A Guide to Error Handling that Just Works (Part I)

@Bugen Zhao

Error handling in Rust is straightforward: every competent Rust developer knows the libraries like `anyhow` and `thiserror`, writes `?` operator like an expert to make the compiler happy everyday. Error handling in Rust could still be hard: there're tons of opinionated articles or libraries promoting their best practices, leading to an epic debate that never ends.

As a large Rust project, we were all starting to notice that there was something wrong with the error handling practices in RisingWave, but pinpointing the exact problems is challenging. In the past few months, I've taken on the task of exploring the better error-handling practices for RisingWave. Suffering from the letdowns as the dreamlike worlds painted by so-called "best practices" crumbled one by one, I finally came to realization that...

- There's no one-size-fits-all solution for a Rust project as complicated as RisingWave.

- The user-facing changes are the best guideline to this extensive project.
- Achieving a consensus on what is *good* will never be possible. Focus on what is generally considered *bad* and address that instead.
- Embrace the community and practice generosity.

During the process I've tried to do the most improvements on my own, but finally realize that the gained knowledge has to be shared with all our teammates in order to maintain the good status of error-handling in RisingWave, which is why this guide has been written. A considerate portion of the contents is derived from the discussions with reviewers of the refactoring PRs, with special thanks to @Tianxiao Shen for his valuable insights.

So let's get started!

Revisit `trait Error`

Perhaps you have been quite used to defining a new error type with `thiserror`, or interacting with existing error types, but do you really know how the language designers think what an error should look like? The concept is illustrated in the definition of `trait Error` in the standard library, so let's take a glance first.

```
/// `Error` is a trait representing the basic expectations fo
r error values,
/// i.e., values of type `E` in [`Result<T, E>`].
pub trait Error: Debug + Display {
    /// The lower-level source of this error, if any.
    fn source(&self) -> Option<&(dyn Error + 'static)> { ...
}

    /// Provides type based access to context intended for er
ror reports.
    fn provide<'a>(&'a self, request: &mut Request<'a>) { ...
}
}
```

Describe it

First of all, there're two super-traits on the `Error` trait which are both required to describe the error in different circumstances. Specifically,

- `Debug` representation is used when calling `Result::unwrap` or `Result::expect`. Since this is less commonly encountered (in happy paths), there are no specific requirements regarding the format.
  - Most of the error types directly `#[derive(Debug)]` to implement that, including those from standard libraries.
  - `anyhow::Error` customizes to make the debug representation human-readable [ref]. This makes the error message more friendly if one put `anyhow::Result` as the return type of the `main` function, which will call `Termination::report` then `Debug` on the error type.
- `Display` representation is to give a user-friendly description of the error, commonly known as the "error message" which we should pay the main attention to. Followings are the conventions:
  - The message should **be lowercase sentences without trailing punctuation** [ref].
    - BAD: `Failed to connect to server.`
    - GOOD: `failed to connect to server`
  - The message should **only describes itself, without (recursive) formatting on the source (or cause) error** [ref]. We'll talk about the "source" later but I'm sure you can get the idea through the example.
    - BAD: `failed to bind expression: {source}`
      - for example, `failed to bind expression: function "foo" does not exist`
    - GOOD: `failed to bind expression`
  - The message should **not include other stuff** as well, especially the backtraces.
    - BAD: `failed to parse statement "foo"\n\n Backtrace: {backtrace}`
    - GOOD: `failed to parse statement "foo"`

Some of the conventions above might be surprising to you. You might be wondering...

- Isn't there a loss of information if we don't mention the cause of the error?
- How can we effectively debug if we don't include the backtrace?

To answer these questions, let's now go through the methods on the `Error` trait to see how they can work together to provide a concise yet informative message for both users and developers.

How come?

Modern software is structured in layers. It's common that we don't know about the details how external systems or libraries work but only interact with them through interfaces. When there's something wrong within them, we'll get an error based on which we can determine the next steps.

In most cases, we attach our own interpretation (called **context**) based on our own interpretation to create a new error, making the original one as the "source". This is what the `source` method is for.

> The `source` method provide cause information, which is generally used when errors cross "abstraction boundaries" (like modules or crates). [ref]

You might not have made any direct interaction with this method, but you are likely familiar with the attributes like `#[source]` when defining an error type with `thiserror`. `#[source]` will help to implement the `source` method to return the inner error. By the way, `#[from]` implies `#[source]` so we don't need to specify them together.

```
#[derive(thiserror::Error, Debug)]
enum BatchError {
    #[error("failed to run expression")]
    Expr(#[from] ExprError)
}
```

The method helps to maintain the error cause into a **chain**, as the source error can then have its own source again. To visit the source chain, call `Error::source`

recursively on the root error. There's recently a new and unstable `Error::sources` method help to do this as well [ref].

Being able to provide the source chain explains why we don't have to refer to the source (or inner) error while implementing `Display`: **the root-level error can choose its own way to composite the sources into a final error message**, which is called **report** by convention [ref].

If you are observant, you might have already noticed how we apply this in RisingWave's user-facing error reporting via psql, where each line represents a source error.

```
ERROR: Failed to run the query

Caused by these errors (recent errors listed first):
  1: Failed to get/set session config
  2: Invalid value `maybe` for `rw_implicit_flush`
  3: Invalid bool
```

In the meanwhile, we connect the error source chain in a single line to get them printed in the logs, which can be much more concise for our developers to read and analyze.

```
failed to collect barrier: Actor 233 exited unexpectedly: Exe
cutor error: Chunk operation error: Division by zero
```

It's not hard to find that maintaining the source chain in a structured way leads to much more flexibility than directly embedding them in the `Display` implementation of a single error. We'll cover the part for how you should format the error into reports and benefit from this later.

Provide any stuff

An error message can actually be much fancier and more informative than the multi-line one above. For example,

- Include **span** information to indicate the location of the syntax error for users.
- Instruct users how to fix the error with some **hints** or **suggestions**.

- Display the captured **backtrace** showing where the error first occurred in the source code.

The need for a more user-friendly error message can be quite varying depending on the application, that's why the trait defines another method named `provide` allowing an error to provide *any* kind of context to the outside world.

Not being stabilized, this method has not been widely used by the ecosystem. However, there are still conventions that an error should...

- Call `Error::provide` on the source error, if exists.
- Provide a `std::backtrace::Backtrace` if captured, which is the primary purpose of this method at present.

```
fn provide<'a>(&'a self, request: &mut std::error::Request<'a
>) {
    if let Some(backtrace) = &self.backtrace {
        request.provide_ref::<Backtrace>(backtrace);
    }
    if let Some(source) = &self.source {
        source.provide(request);
    }
}
```

To request a value from an error, call `std::error::request_ref`. Similar to `source`, this is not something we typically encounter in our daily lives either. Error reports will handle this for us, again, which will be covered later.

```
if let Some(backtrace) = std::error::request_ref::<Backtrace>
(&error) {
    println!("Backtrace:\n{backtrace}");
}
```

World's complicated

You may now find the error friends you meet everyday can be much more powerful than you thought. However, the world is complicated. Do you also know

that not all stuff named “error” is actually an `Error`?

This might be mainly because there’s no `Error` trait bound on the type parameter `E` in `Result<T, E>`. Some interfaces returning `Result` actually mean the more general `Either`, while others may simply forget to implement the trait on the error type. There usually won’t be a problem until you want to make it a source of a new error.

Another different case is `anyhow::Error`. Yes, `anyhow::Error` is not an `Error` 😄. It's not that it doesn't *want* to be, but unfortunately it *cannot* be. To explain it in short:

```
// `anyhow::Error` aims to be the container of any kinds of e
rror types:
impl<E: Error> From<E> for anyhow::Error { .. }

// So if...
impl Error for anyhow::Error { .. }

// We'll get it conflict with the blanket implementation from
`std`:
impl<T> From<T> for T { .. }
```

Blame the compiler, no reservation! The limitation makes it more difficult to write generic code that works with all `Error` types as desired, since the large piece for `anyhow` support is missing. However, if you didn’t notice this fun fact, it’s likely because of those clever type tricks that make `anyhow::Error` behave like a normal `Error` type. Let’s discuss this later if there’s a chance.

Formatting the Error

Now that we’ve mastered the basic knowledge of how errors should behave, let’s move on to something more practical. I’m going to cover the topics in a top-down manner to avoid losing ourselves in this long journey. So first, imagine you’ve got an error from some other folks, how should we format it to get it displayed to the users or appeared in the logs?

Make it a `Report`

We've already known the concept of source chain and how it should be leveraged to create an error report. `thiserror_ext::Report` can handle all the stuff for us [ref]. You can check the documentation on docs.rs to find the detail usages, or in simple terms...

- Instead of writing `format!("error: {}", error)` , use
  - `format!("error: {}", error.as_report())` if you want a concise inline representation
  - `format!("error: {:#}", error.as_report())` if you want a pretty multi-line representation
- If you want to include the backtrace in the report, add an extra `?` to use `Debug` format:
  - `format!("error: {:?}", error.as_report())` for inline
  - `format!("error: {:#?}", error.as_report())` for multiline
- Use the following sugars if you just want `to_string`:

```
pub trait AsReport: Sealed {
        ...
    fn to_report_string(&self) -> String { ... }
    fn to_report_string_with_backtrace(&self) -> String {
... }
    fn to_report_string_pretty(&self) -> String { ... }
    fn to_report_string_pretty_with_backtrace(&self) -> St
ring { ... }
}
```

So simple, right? But wait, I must now clarify that in most cases, calling `format` on error report **is not what you want, or even sometimes bad**.

Format in `tracing`?

In RisingWave, we leverage `tracing` to emit runtime logs. At first glance, it may seem like just a `println` with level-filtering support, but this is far from accurate.

The most powerful functionality of `tracing` is the support for structured logging to gain better observability for the system [ref].

This topic is too extensive to cover in this article. However, all you need to know now is that, instead of formatting everything into the log message like the old-`println` way, record the variable parts into **fields** as much as possible. This is to make the logs more machine-readable so that we can do analysis on them programmatically.

Here's an example:

```
// BAD
tracing::info!(
    "failed to parse column `{}` for source {}, error: {}",
    name, id, error.as_report(),
);

// GOOD
tracing::info!(
    name,
    source_id = id,
    error = %error.as_report(),
 // error = ?error.as_report() /* with backtrace */
    "failed to parse column",
);
```

The `%` before `error.as_report()` indicates that the error field will be a string formatted with `Display` trait on the report, which will be one-liner without backtrace as you've already known. If you want the backtrace, replace it with `?` to use the `Debug` representation.

Backtrace meh?

When should we include the backtrace in the error report? Here are some tips to consider:

- If the error occurs on happy and critical paths, **do not** include since it introduces overhead while resolving the symbols.

- If the error occurs frequently, **do not** print since it can be really verbose!
- If the report will be shown to users, **do not** print the backtrace. Imagine what the user looks like when scared by hundreds of lines of incantation. 👻🌬️😨
- If the error is simple and self-explanatory, or soon gets resolved after being created, **do not** print the backtrace since it's likely to be meaningless.
- Only if the error is significant, unexpected, and complicated, print the backtrace. A typical example is the error logging after an actor exited (failed).

```
// Intentionally use `?` on the report to also include the
backtrace.
tracing::error!(actor_id, error = ?err.as_report(), "actor
exit with error");
```

BTW, I would also like to emphasize that, log the error **only if you're going to ignore or resolve it**. This approach guarantees that the error will only be logged once to avoid cluttering the logs, as it will eventually be resolved during propagation (otherwise we get panic).

Format in `anyhow`?

`anyhow::anyhow!` is again a stuff that feels quite similar to `format!`. However, it must be pointed out that formatting error (report) in `anyhow!` is generally a bad idea. Consider the following example:

```
return Err(
    anyhow!("failed to fetch offset: {}", mysql_error.as_repo
rt())
);
```

The intention of this line is to create a new `anyhow::Error` indicating that we failed to fetch the offset, preserving the cause of the original error from the external MySQL library. Having the knowledge of how `Display` and `source` are supposed to work on an error type, I believe that it's not hard to figure out why this is not a good practice. To be clear:

- The description (message) of the new error will contain the description of the cause `mysql_error`, which violates the convention mentioned above.
- The source chain of the new error is not well-maintained. The `source` method will return `None` in this case.

The best way for doing this is to attach context through `anyhow::Error::context`. We'll discuss about that in the "error construction" section later.

```
return Err(
    anyhow!(mysql_error).context("failed to fetch offset")
);
```

The machine power

As you can imagine, formatting an error without using `Report` can be problematic in most time: we may lose the information from the entire source chain! Luckily, we can leverage the power from machine to identify the problems.

Thanks to `cargo-dylint` which allows everyone to write his own lint rules with the exactly same experience as `cargo-clippy`, I've also created one named `format_error` to cover the problem. It has been integrated into CI for a while. As a result, if you introduce some error formatting without practicing the best, you can refer to the instructions provided to fix it.

Given that the lint may not able to cover all the edge cases and the suggestions can be inaccurate, please kindly be sure to have a good understanding of this guide before taking actions. 🥰

To be continued...

That's all for the very first part of this guide series, while the journey is far from over. In the upcoming parts, we'll dive into some lower-level topics by shifting our focus from error *consumption* to *production*, including...

- how to define an error type, choose `thiserror` or `anyhow` ?
- the best practices to construct an error instance respectively with both libraries,

- and more interesting stories or tricks in the ecosystem.

Stay tuned for more updates!