Designing error types in Rust

✏ 2022-11-15 ✂ 2022-11-16

Introduction

If I had to pick my favorite Rust language feature, that would be its systematic approach to error handling. Sum types, generics (such as `Result<T, E>`), and a holistic standard library design perfectly[1] match my obsession with edge cases. Rust error handling is so good that even Haskell looks bleak and woefully unsafe[2]. This article explains how I approach errors when I design library interfaces in Rust.

Libraries vs. applications

My approach to errors differs depending on whether I am writing a general-purpose library, a background daemon, or a command-line tool.

Applications interface humans. Applications do their job well when they resolve issues without human intervention or, if automatic recovery is impossible or undesirable, provide the user with a clear explanation of how to resolve the issue.

Library code interfaces other code. Libraries do their job well when they recover from errors transparently and provide programmers with a complete list of error cases from which they cannot recover.

This guide targets library design because that is the area with which I am most familiar. However, the core principle of empathy applies equally well to designing machine-machine, human-machine, and human-human interfaces.

Design goals

> *I didn't want to call these guidelines, and I didn't want to call these rules. I wanted them to be goals. These are the things that you should strive for in your code, that are not always easy to accomplish. And maybe you can't always pull them off. But the closer you come, the better your code will be.*
>
> Sean Parent, "C++ Seasoning"

Most issues in the error type design stem from the same root: making error cases easy for the code author at the expense of the caller. All the strategies I describe in this article are applications of the following mantra:

☛ *Be empathetic to your user.*

Imagine yourself having to handle the error. Could you write robust code given the error type and its documentation? Could you translate the error into a message the end user can understand?

Prefer specific enums

Applying familiar error-handling techniques is tempting if you come to Rust from another language. A single error type might seem natural if you wrote a lot of Go.

⊕

```
pub fn frobnicate(n: u64) → anyhow::Result<String> { /* … */ }
```

If you hardened your character with C++ or spent a lot of time working with grpc, having a humongous global error type might seem like a good idea.

⊕

```
pub enum ProjectWideError {
  InvalidInput,
  DatabaseConnectionError,
  Unauthorized,
  FileNotFound,
  // …
}

pub fn frobnicate(n: u64) → Result<String, ProjectWideError> { /
```

These approaches might work fine for you, but I found them unsatisfactory for library design[3] in the long run: they facilitate *propagating* errors (often with little context about the operation that caused the error), not *handling* errors.

When it comes to interface clarity and simplicity, nothing beats algebraic data types (ADTs). Let us use the power of ADTs to fix the `frobnicate` function interface.

⊕

```
pub enum FrobnicateError {
  /// Frobnicate does not accept inputs above this number.
  InputExceeds(u64),
  /// Frobnicate cannot work on mondays. Court order.
  CannotFrobnicateOnMondays,
}

pub fn frobnicate(n: u64) → Result<String, FrobnicateError> { /*
```

Now the type system tells the readers what exactly can go wrong, making *handling* the errors a breeze.

You might think, “I will never finish my project if I define a new enum for each function that can fail.” In my experience, expressing failures using the type system takes less work than documenting all the quirks of the interface. Specific types make writing good documentation easier. They repay their weight in gold when you start testing your code.

Feel free to introduce distinct error types for each function you implement. I am still looking for Rust code that went overboard with distinct error types.

⊕

```
#[test]
fn test_unfrobnicatable() {
  assert_eq!(FrobnicateError::InputExceeds(MAX_FROB_INPUT), frobn
}

#[test]
fn test_frobnicate_on_mondays() {
  sleep_until(next_monday());
  assert_eq!(FrobnicateError::CannotFrobnicateOnMondays, frobnica
}
```

Reserve panics for bugs in your code

> *The `panic!` macro is used to construct errors that represent a bug that has been detected in your program.*
>
> The Rust Standard Library, When to use `panic!` vs `Result`.

The primary purpose of `panics` in Rust is to indicate bugs in your program. Resist the temptation to use panics for input validation if there is a chance that the inputs come from the end user, even if you document panics meticulously. People rarely read documentation; they can easily miss your warnings. Use the type system to guide them.

⊕

```
/// Frobnicates an integer.
///
/// # Panics
///
/// This function panics if
/// * the `n` argument is greater than [MAX_FROB_INPUT].
/// * you call it on Monday.
pub fn frobnicate(n: u64) → String { /* … */ }
```

Feel free to use panics and assertions to check invariants that must hold in *your* code.

⊕

```
pub fn remove_from_tree<K: Ord, V>(tree: &mut Tree<K, V>, key: &K
  let maybe_value = /* … */;
  debug_assert!(tree.balanced());
  debug_assert!(!tree.contains(key));
  maybe_value
}
```

You can panic on invalid inputs if the failure indicates a severe bug

in the caller's program. Good examples are out-of-bound indices or trait implementations that do not obey laws (e.g., if an `Ord` type violates the total order requirements).

Lift input validation

Good functions do not panic on invalid inputs. Great functions do not have to validate inputs. Let us consider the following interface of a function that sends an email.

⊕

```
pub enum SendMailError {
  /// One of the addresses passed to send_mail is invalid.
  MalformedAddress { address: String, reason: String },
  /// Failed to connect to the mail server.
  FailedToConnect { source: std::io::Error, reason: String },
  /* … */
}
pub fn send_mail(to: &str, cc: &[&str], body: &str) → SendMailEr
```

Note that our `send_mail` function does at least two things: validating email addresses and sending emails. Such a state of affairs becomes tiresome if you have many functions that expect valid addresses as inputs. One solution is to pepper the code with more types. In this case, we can introduce the `EmailAddress` type that holds only valid email addresses.

⊕

```
/// Represents valid email addresses.
pub struct EmailAddress(String);

impl std::str::FromStr for EmailAddress {
  type Err = MalformedEmailAddress;
  fn from_str(s: &str) → Result<Self, Self::Err> { /* … */ }
}

pub enum SendMailError {
  // no more InvalidAddress!
  FailedToConnect { source: std::io::Error, reason: String },
  /* … */
```

```
}

pub fn send_mail(
    to: &EmailAddress,
    cc: &[&EmailAddress],
    body: &str,
) → SendMailError { /* … */ }
```

If we add more functions working with valid addresses, these functions will not have to run the validation logic and return address validation errors. We also enable the caller to perform address validation earlier, closer to where the program receives that address.

Implement std::error::Error

Implementing the `std::error::Error` trait for error types is like being polite. You should do it even if you do not mean it.

Some callers might care about something other than your beautiful design, shoveling your errors into a `Box<Error>` or `anyhow::Result` and moving on. They may be building a little command line tool that does not need to handle machines with 4096 CPUs. If you implement `std::error::Error` for your error types, you will make their lives easier.

If you find that implementing the `std::error::Error` trait is too much work, try using the `thiserror` package.

⊕

```
use thiserror::Error;

#[derive(Error, Debug)]
pub enum FrobnicateError {
  #[error("cannot frobnicate numbers above {0}")]
  InputExceeds(u64),

  #[error("thy shall not frobnicate on mondays (court order)")]
  CannotFrobnicateOnMondays,
}
```

Define errors in terms of the problem, not a solution

The most common shape of errors I see looks like the following:

⊕

```
pub enum FetchTxError {
  IoError(std::io::Error),
  HttpError(http2::Error),
  SerdeError(serde_cbor::Error),
  OpensslError(openssl::ssl::Error),
}

pub fn fetch_signed_transaction(
  id: Txid,
  pk: &[u8],
) → Result<Option<Tx>, FetchTxError> { /* … */ }
```

This error type does not tell the caller *what* problem you are solving but *how* you solve it. Implementation details leak into the caller's code, causing much pain:

> Such error types encourage unhealthy coding patterns when low-level errors travel up the call stack with minimal context attached. The following error message comes from one program I have to use that often leaves me puzzled and depressed.

```
IO error: Os { code: 2, kind: NotFound, message: "No such file or
```

> Your clients must read the leaked dependencies documentation to learn about possible error cases. Look at `openssl::ssl::Error`, for example. Can you devise a good recovery strategy without knowing which `openssl` library function returned this error?
>
> Your clients must add `openssl` and `serde_cbor` to direct dependencies to handle your errors. If you decide to switch from `openssl` to `libressl` or from `serde_cbor` to `ciborium`, your clients will have to adapt their code.

Let us redesign the `FetchTxError` type, focusing on the well-being of fellow programmers calling that code.

⊕

```
pub enum FetchTxError {
  /// Could not connect to the server.
  ConnectionFailed {
    url: String,
    reason: String,
    cause: Option<std::io::Error>, // ①
  },

  /// Cannot find transaction with the specified txid.
  TxNotFound(Txid), // ②

  /// The object data is not valid CBOR.
  InvalidEncoding { // ③
    data: Bytes,
    error_offset: Option<usize>,
    error_message: String,
  },

  /// The public key is malformed.
  MalformedPublicKey { // ④
    key_bytes: Vec<u8>,
    reason: String,
  },

  /// The transaction signature does not match the public key.
  SignatureVerificationFailed { // ④
    txid: Txid,
    pk: Pubkey,
    sig: Signature,
  },
}
```

```
pub fn fetch_signed_transaction(
  id: Txid,
  pk: &[u8],
) → Result<Tx, FetchTxError> { /* … */ }
```

The new design offers several of improvements:

> The `ConnectionFailed` constructor wraps a low-level `std::io::Error` error. Wrapping works fine here because there is enough context to understand what went wrong.
>
> We replaced the `Option` type with an explicit error constructor, `TxNotFound`, clarifying the meaning of the `None` case.
>
> The `InvalidEncoding` constructor hides the details of the decoding library we use. We can now replace `serde_cbor` without breaking other people's code.
>
> We replaced generic crypto errors with two specific cases: `TxidMismatch` and `SignatureVerificationFailed`. Our fellow programmer has more context to make rational decisions: the `MalformedPublicKey` case indicates that the user supplied the wrong key. The `SignatureVerificationFailed` case can indicate that the peer tampered with the data, so we should try connecting to another peer.

If I needed to call `fetch_signed_transaction`, I prefer the latter interface. Which interface would you choose? Which interface will be easier to test?

Do not wrap errors, embed them

We have already seen the tactic of embedding error cases in the previous section. This tactic eases interface comprehension so much that it deserves more attention.

Imagine that we are working on a little library that verifies

cryptographic signatures. We want to support ECDSA and BLS signatures. We start from the path of the least resistance.

⊕

```
pub enum Algorithm { Ecdsa, Bls12381 };

pub enum VerifySigError {
  EcdsaError { source: ecdsa::Error, context: String },
  BlsError { source: bls12_381_sign::Error, context: String },
}

pub fn verify_sig(
  algorithm: Algorithm,
  pk: Bytes,
  sig: Bytes,
  msg_hash: Hash,
) → Result<(), VerifySigError> { /* … */ }
```

There are a few issues with that `verify_sig` function design.

> There is an implicit assumption that if the caller passes the `Ecdsa` as the `algorithm`, the error can be only `EcdsaError`. It should be clear from the semantics, but the type system does not enforce this invariant.
>
> The error type leaks implementation details to the caller.
>
> If we extend the list of supported algorithms, the caller might have to modify all call sites.

We can address these issues by removing one layer of nesting and embedding error cases from `ecdsa::Error` and `bls12_381_sign::Error` into the `VerifySigError` error type. The result is a clear and self-descriptive error type conveying to your callers that you care about them.

⊕

```
pub enum Algorithm { Ecdsa, Bls12381 };

pub enum VerifySigError {
  MalformedPublicKey { pk: Bytes, reason: String },
```

```
    MalformedSignature { sig: Bytes, reason: String },
    SignatureVerificationFailed {
      algorithm: Algorithm,
      pk: Bytes,
      sig: Bytes,
      reason: String
    },
    // …
  }

  pub fn verify_sig(
    algorithm: Algorithm,
    pk: Bytes,
    sig: Bytes,
    msg_hash: Hash,
  ) → Result<(), VerifySigError> { /* … */ }
```

There are a few cases when wrapping errors makes sense:

> Wrapping `std::io::Error` is acceptable if you include enough context, such as the attempted operation and the paths involved. `std::io::Error` does not bring extra dependencies and is familiar to any seasoned Rust programmer, so it adds little cognitive load. `std::io::Errors` also can contain low-level OS error codes that can help diagnose tricky cases.
>
> It is often acceptable to convert a lower-level error to a string and attach that string to your errors, as long as the containing error type constructor is descriptive enough. However, you should check that these strings do not contain sensitive information, such as email addresses or secret keys.

You might prefer to wrap a `Box<dyn Error>` instead of converting the error to string so the caller can downcast the error, delay the conversion to string, and traverse the error stack using the `source` method. I found that boxing errors does not help me much in practice:

> If the caller needs to access information from the original

error programmatically, embed the relevant bits or add more type constructors. Downcasting is a short-term solution.

The client must depend on the same semantic version of the transitive dependency to downcast the error. The client code can silently break if the versions diverge (`0.3` in the client code vs. `0.4` in your code, for example).

The error types become impossible to clone and serialize (my errors often cross process boundaries).

Resources

There is a lot of research on error-handling approaches. Yet the practical application of those ideas in real-world programming interfaces is an art requiring good taste and human compassion. The following resources made the most profound imprint on my thinking about errors.

Catch me if you can: Looking for type-safe, hierarchical, lightweight, polymorphic and efficient error management in OCaml by David Teller, Arnaud Spiwack, and Till Varoquaux. This article demonstrates how features of a high-level functional language give rise to a powerful new way of dealing with errors.

The Error vs. Exception article on Haskell Wiki has a few through-provoking parallels between `panics` (called “errors” in the article) and recoverable errors (called “exceptions”).

Parse, don’t validate by Alexis King is a beautiful introduction to type-driven design and error handling.

The Trouble with Typed Errors by Matt Parsons. I share Matt’s passion for precisely expressing errors in types, even though I would not try to replicate his Haskell-specific ideas in Rust.

You can discuss this article on Reddit.

Similar articles

When Rust hurts

Rust at scale: packages, crates, and modules

Tutorial: stable-structures

Scaling Rust builds with Bazel

←IC internals: the ICP ledger
IC internals: Internet Identity storage→
