Checked Exceptions and Result

By Jonathan Louie

Breakdown

- Story time
- Some examples for comparison
- Similarities and differences
- Community discussions
- Key takeaways

Story Time!

“Aren’t Results just Checked Exceptions?”

Examples

Java Example

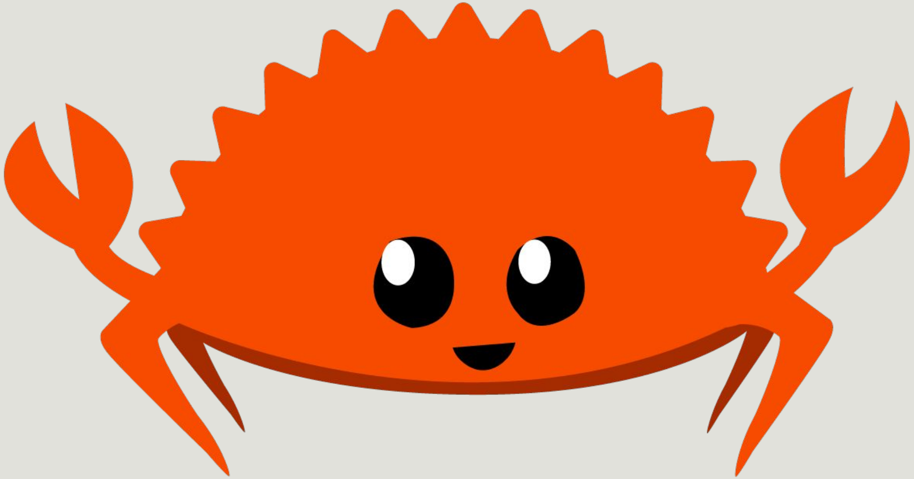
```java
public class Main
{
    public static void greet(Appendable app) throws java.io.IOException {
        app.append("Hello, ");
    }

    public static void greetName(String name, Appendable app) throws java.io.IOException {
        greet(app);
        app.append(name);
        System.out.println(app.toString());
    }

    public static void main(String[] args) {
        StringBuilder sb = new StringBuilder();
        try {
            greetName("Jon", sb);
        } catch (java.io.IOException e) {
            System.err.println("oh no");
        }
    }
}
```

Rust Example

https://play.rust-lang.org/?version=stable&mode=debug&edition=2018&gist=551dc92d373a0934a99ada04b8e59848



Comparison

Similarities

- **Compile-time safety**
- **Propagation**
- **Handle recoverable errors**

Differences?

Big step back

Key difference:

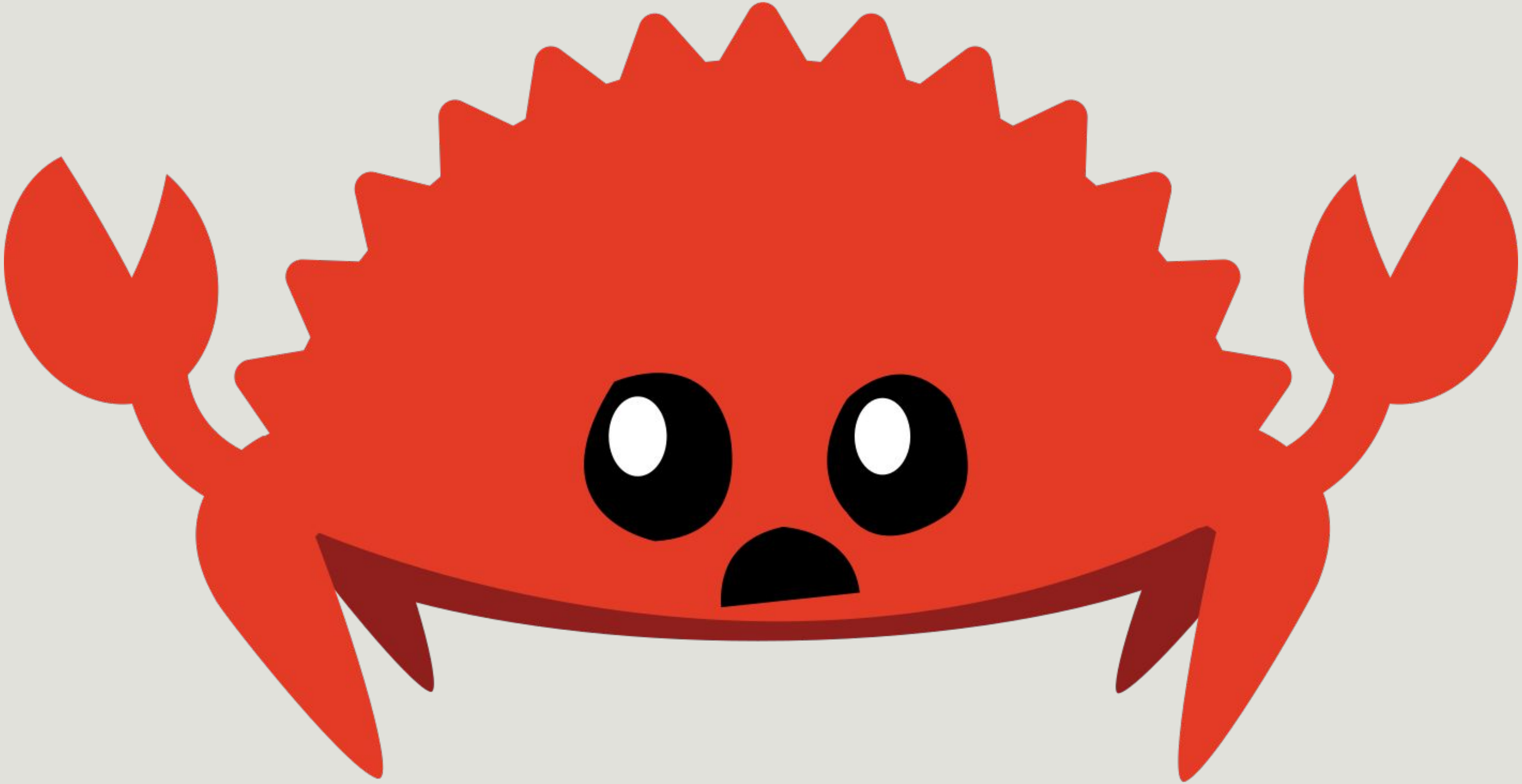
Rust uses “typed errors”

Typed Errors

- **Errors explicit in type signatures**
- **Fallible functions indicated by type signatures**
- **Dependent on type system's expressive power:**
    - **Type safety**
    - **Ergonomics**

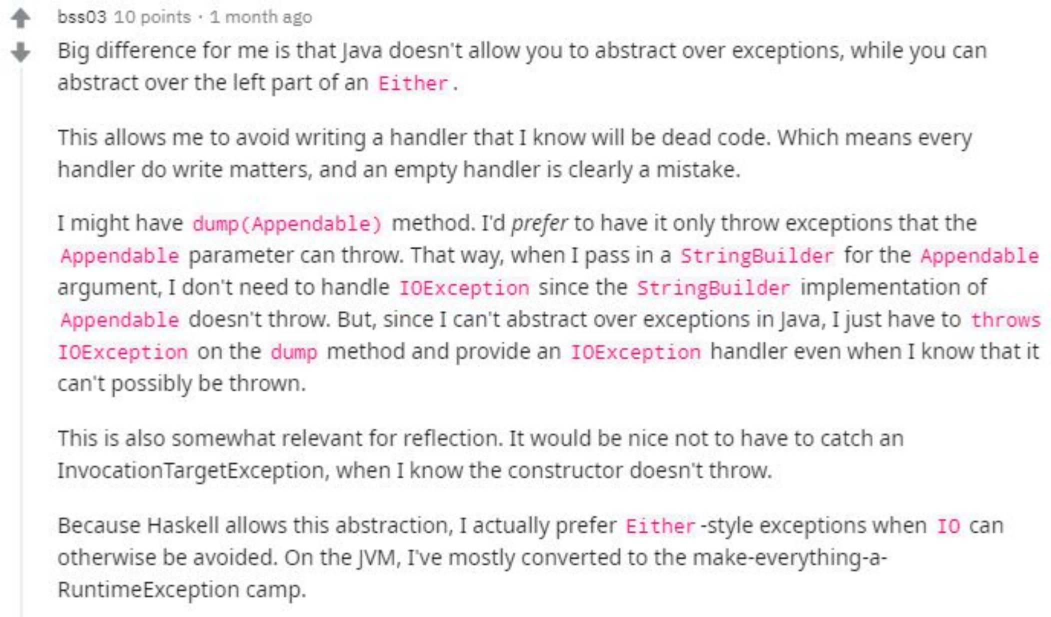
Community Discussions

Why typed errors?



Abstracting over errors

https://www.reddit.com/r/haskell/comments/jkhaqa/difference_between_javas_checked_exceptions/



bss03 10 points · 1 month ago

Big difference for me is that Java doesn't allow you to abstract over exceptions, while you can abstract over the left part of an `Either`.

This allows me to avoid writing a handler that I know will be dead code. Which means every handler do write matters, and an empty handler is clearly a mistake.

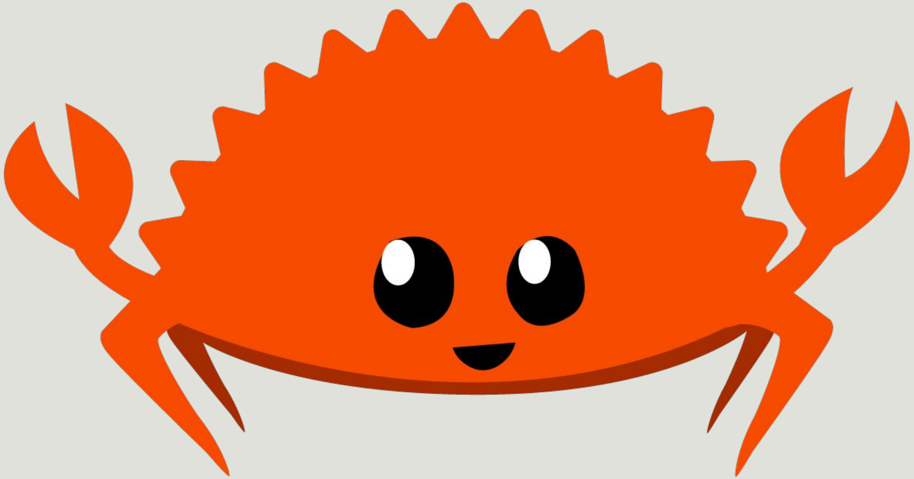
I might have `dump(Appendable)` method. I'd *prefer* to have it only throw exceptions that the `Appendable` parameter can throw. That way, when I pass in a `StringBuilder` for the `Appendable` argument, I don't need to handle `IOException` since the `StringBuilder` implementation of `Appendable` doesn't throw. But, since I can't abstract over exceptions in Java, I just have to `throws IOException` on the `dump` method and provide an `IOException` handler even when I know that it can't possibly be thrown.

This is also somewhat relevant for reflection. It would be nice not to have to catch an InvocationTargetException, when I know the constructor doesn't throw.

Because Haskell allows this abstraction, I actually prefer `Either`-style exceptions when `IO` can otherwise be avoided. On the JVM, I've mostly converted to the make-everything-a-RuntimeException camp.

Back to the previous example…

https://play.rust-lang.org/?version=stable&mode=debug&edition=2018&gist=551dc92d373a0934a99ada04b8e59848



Aside: The never type

- Empty type
- Can be used to indicate infallibility
- Written in Rust as !
- Nightly feature

Are typed errors better than Checked Exceptions?

Not exactly...

https://www.parsonsmatt.org/2018/11/03/trouble_with_typed_errors.html

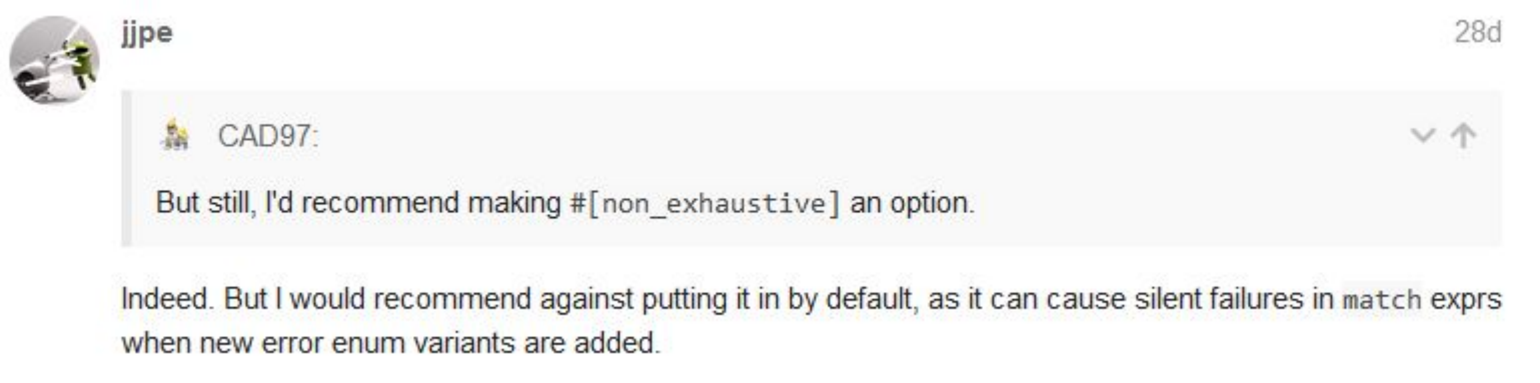
- Monolithic error types allow functions to produce errors they shouldn't
- Monolithic error types force error handlers to handle cases that can't happen
- Wildcard pattern matching can introduce subtle bugs

Example

https://play.rust-lang.org/?version=stable&mode=debug&edition=2018&gist=57d0048a05b6883c4e9b7640f9f17953

Exhaustivity

- Pro: Added typesafety prevents subtle bugs
- Con: New variants/members introduce breaking changes



jjpe 28d

> CAD97:
>
> But still, I'd recommend making `#[non_exhaustive]` an option.

Indeed. But I would recommend against putting it in by default, as it can cause silent failures in `match` exprs when new error enum variants are added.

A criticism of Rust's error handling system

https://www.reddit.com/r/rust/comments/jdvtu4/javas_error_handling_system_is_better_than_that/

- Loss of explicit individual errors
- Inability to add/remove single errors
- Loss of stack traces

A rebuttal

https://degoes.net/articles/bifunctor-io

- Use Either/Result to compose error types and recover individual errors
- Use type-level sets to add/remove errors from a set of errors
- Alternatively, define a new type that is narrower/wider

Type-level Sets in Rust?

- **polyerror crate: https://users.rust-lang.org/t/errors-in-rust-can-now-be-handled-more-ergonomically-cleanly-and-simply-introducing-a-new-error-crate/51527**
- **Anonymous sum types?**
  - **Ordering problem: https://www.parsonsmatt.org/2018/11/03/trouble_with_typed_errors.html**

Anonymous sum types

Ordering 1

Result<A, Result<B, C>>

A | B | C

(A, B, C)

Ordering 2

Result<A, Result<C, B>>

A | C | B

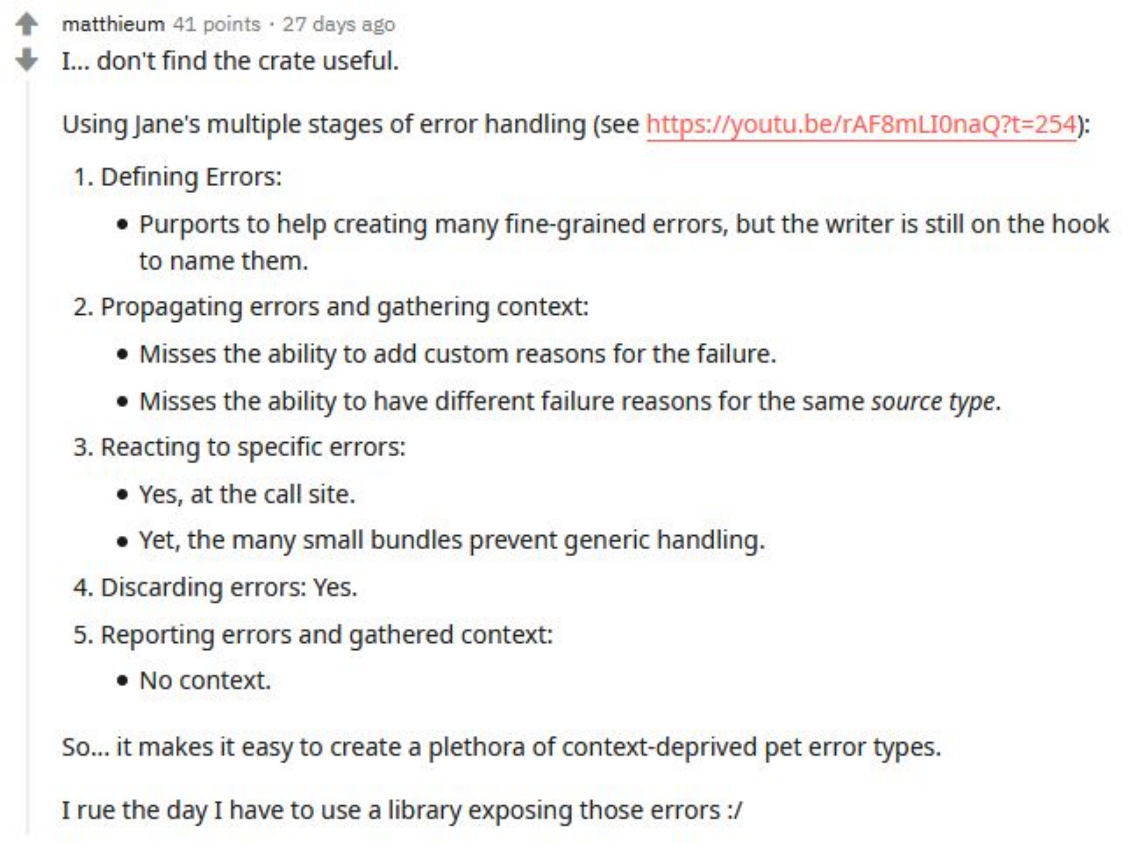
(A, C, B)

What about the stack traces?

https://www.fpcomplete.com/blog/error-handling-is-hard/

- Both Rust and Haskell struggle with this
- Stack traces are helpful, but not a panacea
- Context is key

Is polyerror the answer?



matthieum 41 points · 27 days ago

I... don't find the crate useful.

Using Jane's multiple stages of error handling (see https://youtu.be/rAF8mLI0naQ?t=254):

1. Defining Errors:
   - Purports to help creating many fine-grained errors, but the writer is still on the hook to name them.
2. Propagating errors and gathering context:
   - Misses the ability to add custom reasons for the failure.
   - Misses the ability to have different failure reasons for the same *source type*.
3. Reacting to specific errors:
   - Yes, at the call site.
   - Yet, the many small bundles prevent generic handling.
4. Discarding errors: Yes.
5. Reporting errors and gathered context:
   - No context.

So... it makes it easy to create a plethora of context-deprived pet error types.

I rue the day I have to use a library exposing those errors :/

Jane Lusby's talk

https://www.youtube.com/watch?v=rAF8mLI0naQ

https://github.com/rust-lang/project-error-handling

- Libraries => Error defining (thiserror)
- Applications => Error reporting (anyhow, eyre)
- Use non-exhaustive enums to avoid API breakage
  - Trade-offs!

Performance implications?

https://www.reddit.com/r/rust/comments/k5wk7r/is_rust_leaving_performance_on_the_table_by/

https://www.youtube.com/watch?v=rAF8mLI0naQ

- Performance cost when using Results for errors, even on happy path
  - Stack size proportional to size of error type (use boxing when too large)
- Generating stack traces also incurs a performance penalty
- Benchmark and profile your code!

Key takeaways

Key Takeaways

- **There is no silver bullet for error handling**
- **Typed errors are flexible**
    - **Think very carefully about what works best for the situation!**
- **Checked exceptions are different, but not strictly worse than typed errors**
- **This is not the end of the story!**