The Kids are Ok(())

Understanding collections of Results and Options

by Kevin Martin
@6b766e

Why Not Null?

"I call it my billion-dollar mistake. It was the invention of the null reference in 1965. At that time, I was designing the first comprehensive type system for references in an object oriented language (ALGOL W). My goal was to ensure that all use of references should be absolutely safe, with checking performed automatically by the compiler. But I couldn't resist the temptation to put in a null reference, simply because it was so easy to implement. This has led to innumerable errors, vulnerabilities, and system crashes, which have probably caused a billion dollars of pain and damage in the last forty years."

-- Tony Hoare @ QCon 2009

Why Not Exceptions?

“Almost anything in software can be implemented, sold, and even used given enough determination. There is nothing a mere scientist can say that will stand against the flood of a hundred million dollars. But there is one quality that cannot be purchased in this way - and that is reliability. The price of reliability is the pursuit of the utmost simplicity. [...]

Gradually these objectives have been sacrificed in favor of power, supposedly achieved by a plethora of features and notational conventions, many of them unnecessary and some of them, like exception handling, even dangerous.”

-- Tony Hoare

Why Types?

By using the type system to express the ideas of the absence or failure, we put these logical patterns into terms that a compiler can understand. This means that we can statically analyze our program and verify whether something is formally correct* at compile time, rather than execution time.

* Formal verification is a complex topic.

A Simple Example

```
enum Option<T> {
    Some(T),
    None,
}
```

```
enum Result<T, E> {
    Ok(T),
    Err(E),
}
```

```
use std::fs::File;

fn main() {
    let f = File::open("hello.txt");
    let f = match f {
        Ok(file) => file,
        Err(error) => {
            panic!("There was a problem opening the file: {:?}", error)
        },
    };
}
```

Handling collections of these can get difficult however. Especially when the contained objects are themselves non-trivial types...

std::vec::Vec<std::result::Result<std::option::Option<gimli::AttributeValue<gimli::EndianBuf<'_, gimli::RunTimeEndian>>>, gimli::Error>>

```
std::vec::Vec<
    std::result::Result<
        std::option::Option<
            gimli::AttributeValue<gimli::EndianBuf<'_, gimli::RunTimeEndian>>
        >,
        gimli::Error
    >
>
```

A vector of results. Each element, if the operation was successful, is potentially an attribute value, in the form of a buffer containing data with some lifetime, whose data is organized according to some entianity that will be known at runtime.

How do we iterate through collections like this, while keeping our code human readable?

Is that even possible?

How do we iterate through collections like this, while keeping our code human readable?

Is that even possible?

Yes!

Rust includes a number of useful methods and operators to operate on these concisely, helping you to avoid tedious boilerplate, and pyramids of doom.

async fn mgattozzi() → impl Puns
@mgattozzi

Follow

Replying to @6b766e @mycoliza

How many layers deep of match are you?
Like 2 or 3
You are like a baby watch this

```
match match match match match true {
    _ => println!("lol"),
}{
    _ => println!("lol"),
}{
    _ => println!("lol"),
}{
    _ => println!("lol"),
}{
    _ => println!("lol this compiles"),
}
```

9:20 PM - 6 Jun 2018

The ? Operator

“The ? placed after a Result value is defined to work in almost the same way as the match expressions we defined to handle the Result values in Listing 9-6. If the value of the Result is an Ok, the value inside the Ok will get returned from this expression, and the program will continue. If the value is an Err, the value inside the Err will be returned from the whole function as if we had used the return keyword so the error value gets propagated to the calling code.”

- The Rust Programming Language, Chapter 9 Section 2 - “Recoverable Errors with Result”

```
use std::io;
use std::io::Read;
use std::fs::File;

fn read_username_from_file() -> Result<String, io::Error> {
    let mut s = String::new();

    File::open("hello.txt")?.read_to_string(&mut s)?;

    Ok(s)
}
```

Useful Combinators

The Option and Result types each offer a **map** method. Option's version of map is useful because it converts an Option<T> into an Option<U>, given some f: T -> U. Values that are None will be mapped to None.

This can be more readable than a match statement, and is useful if you only care about transforming the values that do exist.

Demo:

https://play.rust-lang.org/?gist=629f702fcf4c0042e0d78928460bba50&version=stable&mode=debug

Useful Combinators

filter_map will filter out Option results that are None.

and_then is useful for working with functions that return Options, and helps you avoid types like `Option<Option<Option<T>>>`

```
fn sq(x: u32) -> Option<u32> { Some(x * x) }
fn nope(_: u32) -> Option<u32> { None }
assert_eq!(Some(2).and_then(sq).and_then(sq), Some(16));
assert_eq!(Some(2).and_then(nope).and_then(sq), None);
```

Useful Combinators

The **ok_or_else** method is useful for combining an Option into a Result, if the value is None. This is especially helpful when combined with the ? operator, if you consider an empty value to be an error.

```
let x = Some("foo");
assert_eq!(x.ok_or_else(|| 0), Ok("foo"));

let x: Option<&str> = None;
assert_eq!(x.ok_or_else(|| 0), Err(0));
```

Note: Use this instead of `ok_or`, because `ok_or_else` is lazily evaluated.

Collecting Results

The `collect` method can be used for a lot of nifty tricks, but one of the most powerful is its ability to manipulate collections of results. This is an especially useful trick if we want to consider the presence of any error as representative of a failure.

This is best demonstrated with a quick demo.

Demo!

References and Further Reading

Rust by Example Tutorials

- https://rustbyexample.com/error/iter_result.html
- https://rustbyexample.com/error/option_unwrap/map.html
- https://rustbyexample.com/error/option_unwrap/and_then.html
- https://rustbyexample.com/error/result/result_map.html

References and Further Reading

C.A.R. Hoare. "The Emperor's Old Clothes". 1980 Turing Award Lecture:

- http://zoo.cs.yale.edu/classes/cs422/2014/bib/hoare81emperor.pdf

C.A.R. Hoare. “Null References: The Billion Dollar Mistake”. 2009 QCon Lecture

The Rust Programming Language 2nd Ed. “Enums and Pattern Matching” Ch. 6.1:

- https://doc.rust-lang.org/book/second-edition/ch06-01-defining-an-enum.html