한 번에 끝내는
블록체인 개발 A to Z

Chapter 2

Rust Advanced

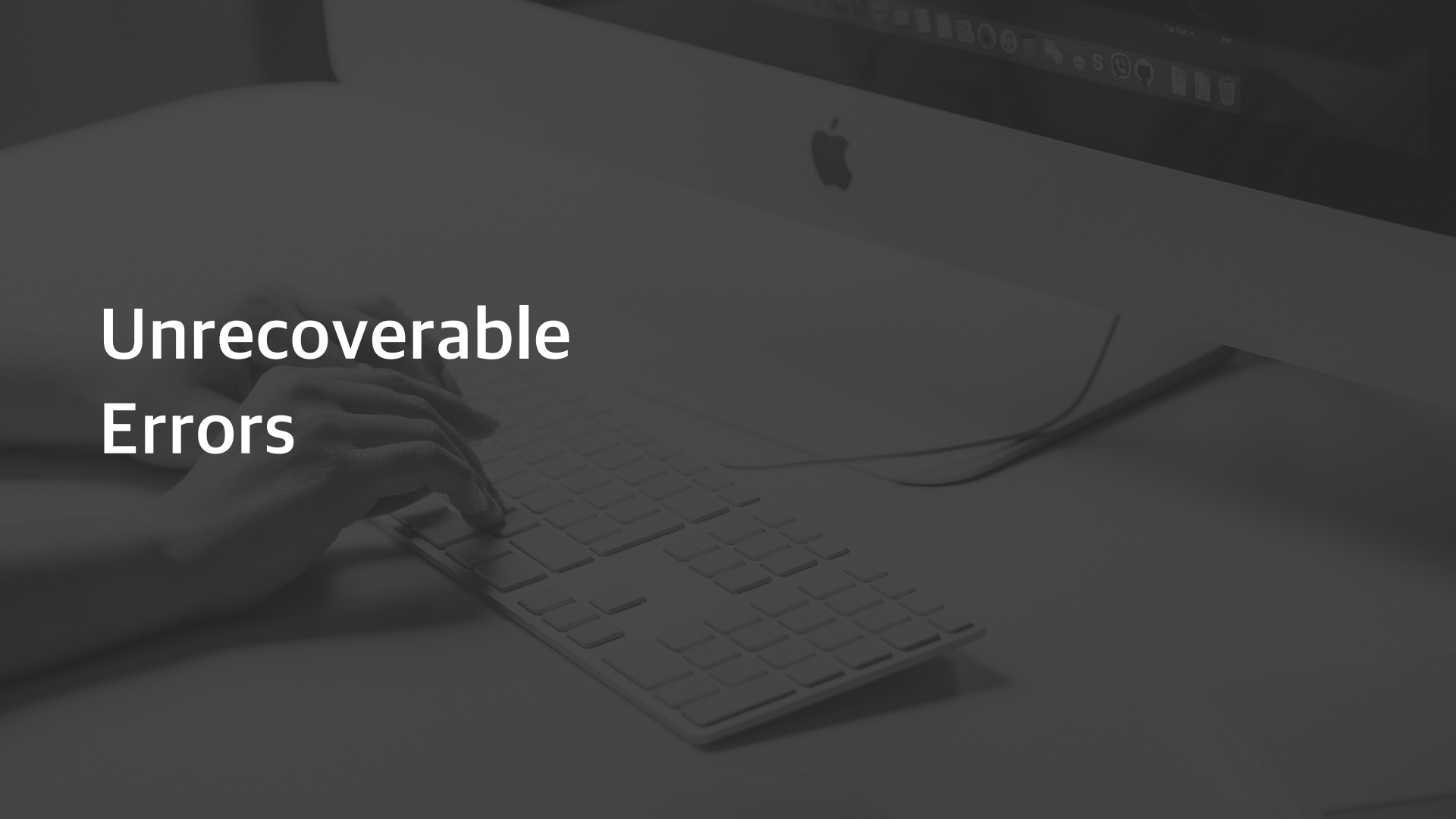
Chapter 2

Rust Advanced

Error Handling

Error Handling

Rust groups errors into two major categories: **recoverable** and **unrecoverable** errors . For a recoverable error, such as a file not found error, we most likely just want to report the problem to the user and retry the operation. Unrecoverable errors are always symptoms of bugs, like trying to access a location beyond the end of an array, and so we want to immediately stop the program.



Unrecoverable Errors

panic!

Sometimes, bad things happen in your code, and there's nothing you can do about it. In these cases, Rust has the **panic!** macro. When the panic! macro executes, your program will print a failure message, unwind and clean up the stack, and then quit. We'll commonly invoke a panic when a bug of some kind has been detected and it's not clear how to handle the problem at the time we're writing our program.

```
fn main() {
    panic!("crash and burn");
}
```

```
$ cargo run
   Compiling panic v0.1.0 (file:///projects/panic)
    Finished dev [unoptimized + debuginfo] target(s) in 0.25s
     Running `target/debug/panic`
thread 'main' panicked at 'crash and burn', src/main.rs:2:5
note: run with `RUST_BACKTRACE=1` environment variable to display a backtrace
```

Backtrace

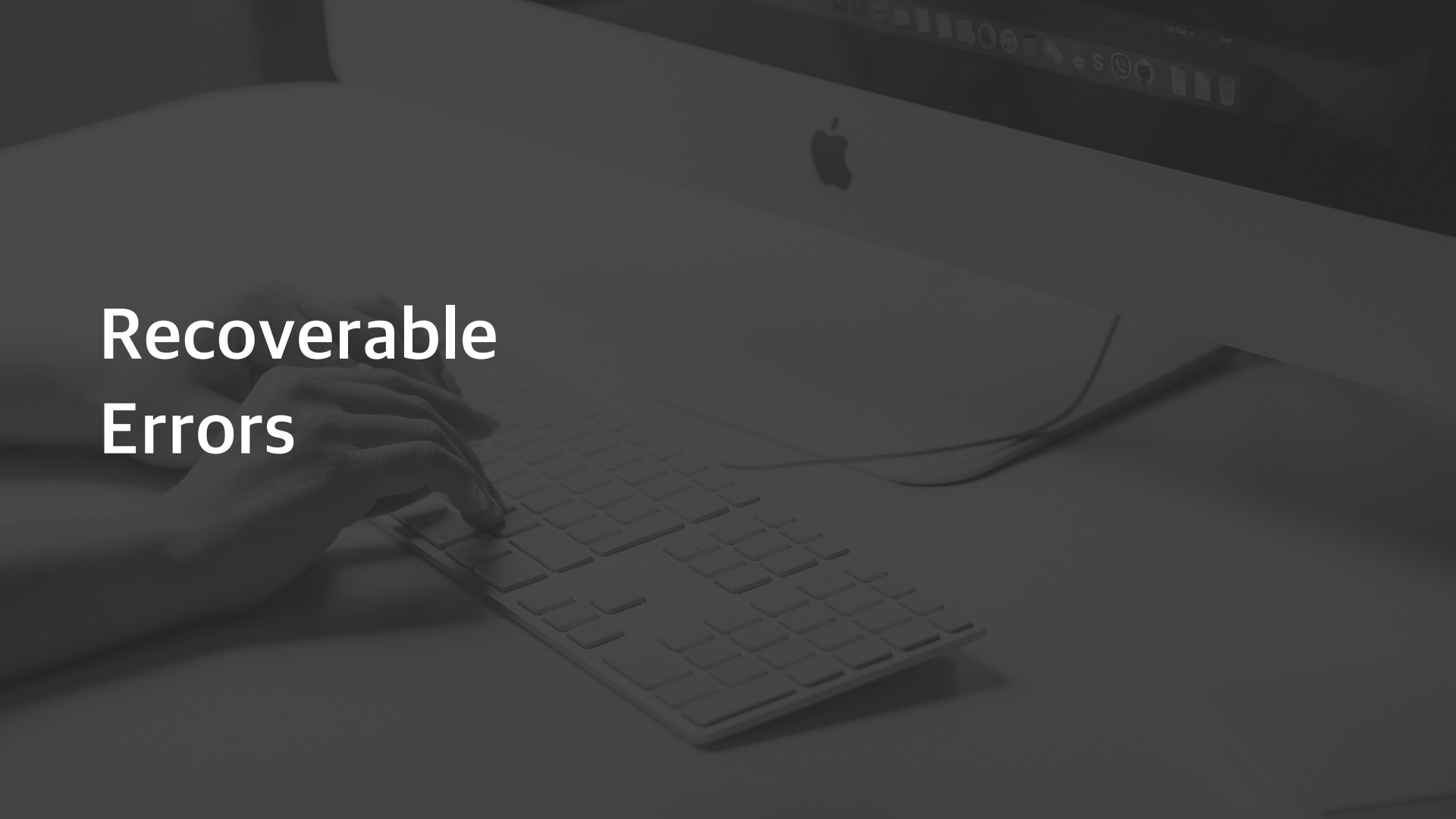
A backtrace is a list of all the functions that have been called to get to this point. Backtraces in Rust work as they do in other languages: the key to reading the backtrace is to start from the top and read until you see files you wrote.

```
fn main() {
    let v = vec![1, 2, 3];

    v[99];
}
```

```
$ cargo run
   Compiling panic v0.1.0 (file:///projects/panic)
    Finished dev [unoptimized + debuginfo] target(s) in 0.27s
     Running `target/debug/panic`
thread 'main' panicked at 'index out of bounds: the len is 3 but the index is 99',
note: run with `RUST_BACKTRACE=1` environment variable to display a backtrace
```

```
$ RUST_BACKTRACE=1 cargo run
thread 'main' panicked at 'index out of bounds: the len is 3 but the index is 99', s
stack backtrace:
   0: rust_begin_unwind
             at /rustc/7eac88abb2e57e752f3302f02be5f3ce3d7adfb4/library/std/src/pani
   1: core::panicking::panic_fmt
             at /rustc/7eac88abb2e57e752f3302f02be5f3ce3d7adfb4/library/core/src/pan
   2: core::panicking::panic_bounds_check
             at /rustc/7eac88abb2e57e752f3302f02be5f3ce3d7adfb4/library/core/src/pan
   3: <usize as core::slice::index::SliceIndex<[T]>>::index
             at /rustc/7eac88abb2e57e752f3302f02be5f3ce3d7adfb4/library/core/src/sli
   4: core::slice::index::<impl core::ops::index::Index<I> for [T]>::index
             at /rustc/7eac88abb2e57e752f3302f02be5f3ce3d7adfb4/library/core/src/sli
   5: <alloc::vec::Vec<T> as core::ops::index::Index<I>>::index
             at /rustc/7eac88abb2e57e752f3302f02be5f3ce3d7adfb4/library/alloc/src/ve
   6: panic::main
             at ./src/main.rs:4
   7: core::ops::function::FnOnce::call_once
             at /rustc/7eac88abb2e57e752f3302f02be5f3ce3d7adfb4/library/core/src/ops
note: Some details are omitted, run with `RUST_BACKTRACE=full` for a verbose backtra
```



Recoverable Errors

Result<T, E>

Most errors aren't serious enough to require the program to stop entirely. Sometimes, when a function fails, it's for a reason that you can easily interpret and respond to.

```
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!("Problem opening the file: {:?}", error),
    };
}
```

Matching Errors

We can take different actions for different failure reasons: if File::open failed because the file doesn't exist, we want to create the file and return the handle to the new file.

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

fn main() {
    let f = File::open("hello.txt");

    let f = match f {
        Ok(file) => file,
        Err(error) => match error.kind() {
            ErrorKind::NotFound => match File::create("hello.txt") {
                Ok(fc) => fc,
                Err(e) => panic!("Problem creating the file: {:?}", e),
            },
            other_error => {
                panic!("Problem opening the file: {:?}", other_error)
            }
        },
    };
}
```

Matching Errors (2)

We can take different actions for different failure reasons: if File::open failed because the file doesn't exist, we want to create the file and return the handle to the new file.

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

fn main() {
    let f = File::open("hello.txt").unwrap_or_else(|error| {
        if error.kind() == ErrorKind::NotFound {
            File::create("hello.txt").unwrap_or_else(|error| {
                panic!("Problem creating the file: {:?}", error);
            })
        } else {
            panic!("Problem opening the file: {:?}", error);
        }
    });
}
```

Shortcuts for Panic

The **unwrap** method is a shortcut method implemented just like the match expression. If the **Result** value is the **Ok** variant, **unwrap** will return the value inside the **Ok**. If the **Result** is the **Err** variant, **unwrap** will call the **panic!** macro for us.

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

fn main() {
    let f = File::open("hello.txt").unwrap();
}
```

```
thread 'main' panicked at 'called `Result::unwrap()` on an `Err` value: Error {
repr: Os { code: 2, message: "No such file or directory" } }',
src/libcore/result.rs:906:4
```

Shortcuts for Panic (2)

Similarly, the **expect** method lets us also choose the **panic!** error message. Using **expect** instead of **unwrap** and providing good error messages can convey your intent and make tracking down the source of a panic easier.

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

fn main() {
    let f = File::open("hello.txt").expect("Failed to open hello.txt");
}
```

```
thread 'main' panicked at 'Failed to open hello.txt: Error { repr: Os { code:
2, message: "No such file or directory" } }', src/libcore/result.rs:906:4
```

Propagating Errors

When a function's implementation calls something that might fail, instead of handling the error within the function itself, you can return the error to the calling code so that it can decide what to do. This is known as propagating the error and gives more control to the calling code, where there might be more information or logic that dictates how the error should be handled than what you have available in the context of your code.

```rust
use std::fs::File;
use std::io::{self, Read};

fn read_username_from_file() -> Result<String, io::Error> {
    let f = File::open("hello.txt");

    let mut f = match f {
        Ok(file) => file,
        Err(e) => return Err(e),
    };

    let mut s = String::new();

    match f.read_to_string(&mut s) {
        Ok(_) => Ok(s),
        Err(e) => Err(e),
    }
}
```

Shortcut for Propagating Errors

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. 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 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.

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

fn read_username_from_file() -> Result<String, io::Error> {
    let mut f = File::open("hello.txt")?;
    let mut s = String::new();
    f.read_to_string(&mut s)?;
    Ok(s)
}
```

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

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)
}
```

? Operator Usage

The ? operator can only be used in functions whose return type is compatible with the value the ? is used on. This is because the ? operator is defined to perform an early return of a value out of the function, in the same manner as the match expression

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

fn main() {
    let f = File::open("hello.txt")?;
}
```

```
$ cargo run
   Compiling error-handling v0.1.0 (file:///projects/error-handling)
error[E0277]: the `?` operator can only be used in a function that returns `Result`
 --> src/main.rs:4:36
  |
3 | / fn main() {
4 | |     let f = File::open("hello.txt")?;
  | |                                    ^ cannot use the `?` operator in a functio
5 | | }
  | |_- this function should return `Result` or `Option` to accept `?`
  |
  = help: the trait `FromResidual<Result<Infallible, std::io::Error>>` is not imple

For more information about this error, try `rustc --explain E0277`.
error: could not compile `error-handling` due to previous error
```

? Operator Usage (2)

The error message also mentioned that ? can be used with Option<T> values as well. As with using ? on Result, you can only use ? on Option in a function that returns an Option. The behavior of the ? operator when called on an Option<T> is similar to its behavior when called on a Result<T, E>: if the value is None, the None will be returned early from the function at that point. If the value is Some, the value inside the Some is the resulting value of the expression and the function continues.

```
fn last_char_of_first_line(text: &str) -> Option<char> {
    text.lines().next()?.chars().last()
}
```