File Management with Error Handling

Mustakimur R. Khandaker

Introduction

The File struct represents a file that has been opened (it wraps a file descriptor), and gives read and/or write access to the underlying file.

- Since many things can go wrong when doing file I/O, all the File methods return the io::Result<T> type, which is an alias for Result<T, io::Error>.
- This makes the failure of all I/O operations explicit.
  - The programmer can see all the failure paths, and should handle them in a proactive manner.

Rust groups errors into two major categories:

- Recoverable error, such as a file not found error, it's reasonable 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.

Error Handling

Rust: Errors

Most languages support error handling in a single approach known as Exceptions.

- Instead, Rust has the type:
  - Result<T, E> for recoverable errors.
  - panic! macro that stops execution when the program encounters an unrecoverable error.

panic!

Rust has the panic! macro.

- When the panic! macro executes, the program will:
  - print a failure message
  - unwind and clean up the stack
  - and then quit.
- This most commonly occurs when a bug of some kind has been detected and it's not clear to the programmer how to handle the error.

Unwinding means Rust walks back up the stack and cleans up the data from each function it encounters.

- The walking back and cleanup is an expensive task.
- The alternative is to immediately abort, which ends the program without cleaning up.
  - Memory that the program was using will then need to be cleaned up by the operating system.
  - Want to force Rust to abort on panic, change Cargo.toml:

```
[profile.release]
panic = 'abort'
```

panic! to Debug

Let's try calling panic! in a simple program:

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

When we run the program, following output will be shown (never use `--release`):

```
$ 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 output looks like:

```
$ RUST_BACKTRACE=1 cargo run
thread 'main' panicked at 'index out of bounds: the len is 3 but the index is 99', src/main.rs:4:5
stack backtrace:
   0: rust_begin_unwind
             at /rustc/7eac88abb2e57e752f3302f02be5f3ce3d7adfb4/library/std/src/panicking.rs:483
   1: core::panicking::panic_fmt
             at /rustc/7eac88abb2e57e752f3302f02be5f3ce3d7adfb4/library/core/src/panicking.rs:85
   ……
   5: <alloc::vec::Vec<T> as core::ops::index::Index<I>>::index
             at /rustc/7eac88abb2e57e752f3302f02be5f3ce3d7adfb4/library/alloc/src/vec.rs:1982
   6: panic::main
             at ./src/main.rs:4
   7: core::ops::function::FnOnce::call_once
             at /rustc/7eac88abb2e57e752f3302f02be5f3ce3d7adfb4/library/core/src/ops/function.rs:227
note: Some details are omitted, run with `RUST_BACKTRACE=full` for a verbose backtrace.
```

Recoverable Errors with Result

Recoverable Error Scenario:
If we try to open a file and that operation fails because the file doesn't exist, we might want to create the file instead of terminating the process.

The Result enum is defined as having two variants, Ok and Err, as follows:

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

The T and E are generic type parameters.

- T represents the type of the value that will be returned in a success case within the Ok variant.
- E represents the type of the error that will be returned in a failure case within the Err variant.

File I/O

Open a File

1.0.0[−][src]Struct std::fs::File

The open static method can be used to open a file in read-only mode.

- A File owns a resource, the file descriptor and takes care of closing the file when it is droped.

File::open() returns a Result<T, E>.

- T has been filled in with the type of the success value, std::fs::File, which is a file handle.
  - the value in the variable f will be an instance of Ok that contains the file handle.
- E used in the error value is std::io::Error.
  - the value in f will be an instance of Err that contains more information about the kind of error that happened.

```
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),
    };
} // `file` goes out of scope, and the "hello.txt" file gets closed
```

Create if Fails

If File::open failed because the file doesn't exist, we want to create the file and return the handle to the new file.
If File::open failed for any other reason— we want the code to panic!

```
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)
            }
        },
    };
}
```

Return an Error

Instead of handling the error within the Error function, we can return the error to the calling code.

- This is known as propagating the error and gives more control to the calling code (to dictate how the error should be handled).

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

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

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

A shortcut of the above can be implemented using ? operator at the end of expression where we expects a return (except of main() unless explicitly expected a return) of Result <T, E>.

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

Read File Content

```rust
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),
    }
}
```

VS

```rust
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)
}
```

Read/Write File

1.0.0[−][src]Struct std::fs::OpenOptions

Writes a three-line message to a file, then reads it back a line at a time with the Lines iterator created by BufRead::lines.

- File implements Read which provides BufReader trait.
- File::create opens a File for writing.
- File::open for reading.
- To append a file, use OpenOptions.

```
use std::fs::File;
use std::io::{Write, BufReader, BufRead,
Error};

fn main() -> Result<(), Error> {
    let path = "lines.txt";

    let mut output = File::create(path)?;
    write!(output, "Rust\nis\nFun")?;

    let input = File::open(path)?;
    let buffered = BufReader::new(input);

    for line in buffered.lines() {
        println!("{}", line?);
    }

    Ok(())
}
```

```
fn file_append_write() {
    let mut file = OpenOptions::new()
        .write(true)
        .append(true)
        .open("my-file")
        .unwrap();

    if let Err(e) = writeln!(file, "A new line!") {
        eprintln!("Couldn't write to file: {}", e);
    }
}
```

Directory Traversal

List of Modified Files

```rust
use std::error::Error;
use std::{env, fs};

fn directory_traversal() -> Result<(), Box<dyn Error>> {
    let current_dir = env::current_dir()?;
    println!("Entries modified in the last 24 hours in {:?}:", current_dir);

    for entry in fs::read_dir(current_dir)? {
        let entry = entry?;
        let path = entry.path();

        let metadata = fs::metadata(&path)?;
        let last_modified = metadata.modified()?.elapsed()?.as_secs();

        if last_modified < 24 * 3600 && metadata.is_file() {
            println!(
                "Last modified: {:?} seconds, is read only: {:?}, size: {:?} bytes, filename: {:?}",
                last_modified,
                metadata.permissions().readonly(),
                metadata.len(),
                path.file_name().ok_or("No filename")?
            );
        }
    }

    Ok(())
}
```

< File Management and Error Handling />