HANDLING ERRORS DIFFERENTLY WITH RUST

Wijaya Adhisurya, Firmware Engineer

1

ERROR HANDLING THE USUAL WAY

Handling Errors and Special Cases in Mainstream Languages

The C Way

- Sentinel values
- Int return values

The C Way: Sentinel value

```
int read_bytes(unsigned char* dst, int max_read);


Int main() {
	int read = read_bytes(unsigned char* dst, int max_read);


	if (read == -1) {
		//handle error here
	}
}
```

The C Way: Int Return Values

```
typedef int MY_ERROR;

typedef void *MY_HANDLE;

MY_HANDLE createSomething();

MY_ERROR doSomething(MY_HANDLE h, char* name, int size);


int main() {

	MY_HANDLE h = createSomething();
	MY_ERROR err = doSomething(“a name”, 7);

	if (err != 0) {

		//handle error here

	}

}
```

The C++ Way: Exceptions (1)

```
struct MyException : public exception {
   const char * description() const {
      return "My Exception";
   }
};

double division(int a, int b) {
   if( b == 0 ) {
      throw MyException();
   }
   return (a/b);
}
```

The C++ Way: Exceptions (2)

```
int main () {
   int x = 50;
   int y = 0;
   double z = 0;

   try {
       z = division(x, y);
       cout << z << endl;
   } catch (MyException& ex) {
     cerr << ex.description() << endl;
   }

   return 0;
}
```

2

REVIEWING THE USUAL WAYS

Why it’s uncomfortable to handle errors the usual ways

Problems With “The C Way”

- Programmer can pretend or ignore there is no error
- Error type is just an int, it isn’t meaningful
- Handling errors will disrupt the flow of program*

Problems With “The C++ Way”

- Catching runtime exception is hard
- Nested calls makes it more complicated

What We Want in Error Handling

- Handle errors unless we really don’t wan’t to
- Distinctive point of failures
- Errors doesn’t disrupt the flow

3

THE RUST WAYS

How Rust Handle Errors

Rust Have At Least 4 Ways of Errors Handling

- **The original way: Match expression**
- The skip other errors way: if let expression
- The functional way: combinatorial error handling
- **The convenient way*: Try trait**

Introduce Panic, Result, and Option

- Panic: Unhandled error in Rust (unwrap, div by zero, ..)
- Result trait:

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

- Option trait:

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

The original way: Match expression

```
use std::num::ParseIntError;
fn double_number(number_str: &str) -> Result<i32, ParseIntError> {
    match number_str.parse::<i32>() {
        Ok(n) => Ok(2 * n),
        Err(err) => Err(err),
    }
}
fn main() {
    match double_number("10") {
        Ok(n) => assert_eq!(n, 20),
        Err(err) => println!("Error: {:?}", err),
    }
}
```

Pyramid of Doom

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

    let f = match f {
        Ok(file) => file,
        Err(ref error) if error.kind() == ErrorKind::NotFound => {
            match File::create("hello.txt") {
                Ok(fc) => fc,
                Err(e) => {
                    panic!(
                        "Tried to create file but there was a problem: {:?}",
                        e
                    )
                },
            }
        },
        Err(error) => {
            panic!(
                "There was a problem opening the file: {:?}",
                error
            )
        },
    };
}
```

Flattening

```
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),
    }
}
```

End Result

```
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)
}
```

More End Result

```
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)
}
```

NO QUESTION