HSPACE Rust 특강

Rust Basic #4 - 오류 처리

Chris Ohk

utilForever@gmail.com

오류 처리

- 패닉
- `Result`

패닉

- 프로그램 자체에 있는 버그로 인해 문제가 생기면 프로그램은 패닉에 빠진다.
  - 배열의 범위 밖에 있는 요소에 접근하는 행위
  - 정수를 `0`으로 나누는 행위
  - `Err`이 되어버린 `Result`에 대고 `.expect()`를 호출하는 행위
  - 단언문 실패
- 프로그래머는 실수를 하기 마련이다. 패닉이 발생하면 그다음은 어떻게 되는 걸까?
  - Rust는 우리에게 선택의 기회를 준다. 우리는 스택을 해제하거나 프로세스를 중단할 수 있다.
  - 기본 동작은 스택을 해제하는 것이다.

패닉

- 해제
  - 정수를 `0`으로 나누는 일이 발생하면 어떤 일이 벌어질까?
    - C++에서는 정의되지 않은 동작(Undefined Behavior)을 한다.
    - Rust에서는 패닉에 빠지게 되며, 보통 다음의 절차가 진행된다.
      - 터미널에 오류 메시지가 출력된다.
      - 스택이 해제된다. 현재 함수가 쓰던 임시 값, 지역 변수, 인수는 모두 생성된 순서와 반대로 드롭된다.
        프로그램이 쓰던 `String`이나 `Vec`은 모두 해제되고 열린 File은 모두 닫힌다.
        현재 함수 호출이 정리되고 나면 이번에는 그의 호출부로 이동해서 같은 방법으로 변수와 인수를 드롭한다.
        그리고 스택 끝에 닿을 때까지 이런 식으로 계속해서 그 함수의 호출부로 이동해 정리한다.
      - 끝으로 스레드가 종료된다. 만약 패닉에 빠진 스레드가 메인 스레드였다면 전체 프로세스가 종료된다.

패닉

- 해제
  - 패닉은 크래시(Crash)도 아니고 정의되지 않은 동작도 아니다.
    오히려 자바의 `RuntimeException`이나 C++의 `std::logic_error`에 가깝다.
    동작은 잘 정의되어 있다. 단지 발생하면 안 될 뿐이다.
  - **패닉은 안전하며 Rust의 어떤 안전 규칙도 위반하지 않는다.**
    Rust는 잘못된 배열 접근이든 뭐든 안 좋은 일이 벌어지기 **전에** 잡아낸다.
    패닉에 빠지면 계속 진행하는 것이 위험하므로 Rust는 스택을 해제한다.
  - **패닉은 스레드별로 발생한다.**
    한 스레드가 패닉에 빠져도 다른 스레드는 정상적으로 일을 수행할 수 있다.

패닉

- 중단
  - 스택 해제는 패닉의 기본 동작이지만, 다음 상황에서는 Rust가 스택 해제를 시도하지 않는다.
    - Rust가 첫 번째 패닉을 정리하고 있는 상황에서 `.drop()` 메소드가 **두 번째 패닉을 유발**하면 이는 치명적인 상황으로 간주한다. Rust는 해제를 멈추고 전체 프로세스를 중단한다.
    - Rust의 패닉 동작은 변경이 가능하다. 프로그램을 `-C panic=abort` 옵션으로 컴파일하면 **첫 번째 패닉이 발생하는 즉시** 프로세스가 중단된다.

Result

- Rust에는 예외가 없다. 대신 실패할 수 있는 함수가 다음과 같은 리턴 타입을 갖는다.

```
fn get_weather(location: String) -> Result<WeatherReport, io::Error>
```

- `Result` 타입은 실패 가능성을 암시한다.
  `get_weather()` 함수는 성공 결과 `Ok(weather)`나 실패 결과 `Err(error_value)` 둘 중 하나를 반환하는데,
  `weather`는 새 `WeatherReport` 값이고 `error_value`는 무엇이 잘못됐는지를 설명하는 `io::Error` 값이다.
- Rust는 위 함수를 호출할 때마다 모종의 오류 처리 코드 작성을 요구한다.
  `Result`에 뭔가를 하지 않고서는 `WeatherReport`를 얻을 수 없으며,
  `Result` 값을 사용하지 않으면 컴파일러가 경고를 내보낼 것이다.

Result

- 오류 잡기
  - Result를 다루는 가장 철저한 방법은 match 표현식을 쓰는 것이다.
    이 방법은 다른 언어의 try/catch에 해당한다. 오류를 호출부에 넘기지 않고 직접 처리하고자 할 때 이 방법을 쓴다.

```
match get_weather(hometown) {
    Ok(report) => {
        display_weather(hometown, &report);
    }
    Err(err) => {
        println!("error querying the weather: {err}");
        schedule_weather_retry();
    }
}
```

Result

- 오류 잡기
  - 하지만 match는 코드를 구구절절 늘어놓게 되는 경향이 있다.

    Result<T, E>는 자주 겪는 몇 가지 상황에서 유용하게 쓸 수 있는 메소드들을 모아 제공한다.
    - result.is_ok(), result.is_err()
    - result.ok()
    - result.err()
    - result.unwrap_or(fallback)
    - result.unwrap_or_else(fallback_fn)
    - result.unwrap()
    - result.expect(message)
    - result.as_ref()
    - result.as_mut()

Result

- Result 타입 별칭
  - Rust 문서를 보다 보면 가끔 오류 타입이 생략된 Result를 만날 때가 있다. Result 타입 별칭을 쓰고 있다는 뜻이다.

```rust
fn remove_file(path: &Path) -> Result<()>
```

  - 타입 별칭은 타입 이름을 위한 일종의 축약 표기다. 모듈은 보통 Result 타입 별칭을 정의해서
    그 모듈에 있는 함수들 대부분이 공통으로 사용하는 오류 타입을 반복해 적지 않아도 되게끔 만든다.

```rust
pub type Result<T> = result::Result<T, Error>;
```

    - 앞의 코드는 std::io::Result<T>라는 공개 타입을 정의한다.
      이 타입은 Result<T, E>의 별칭으로, 오류 타입이 std::io::Error로 고정되어 있다.
      이 말은 use std::io;라고 써 두면 Rust가 io::Result<String>을
      Result<String, io::Error>의 축약 표기로 이해한다는 뜻이다.

Result

- 오류 출력하기
  - 경우에 따라서는 오류를 터미널에 덤프하고 다음으로 넘어가는 게 유일한 방법일 때가 있다.
    이를 위해 오류 타입들의 공통 인터페이스인 std::error::Error는 유용하게 쓸 수 있는 몇 가지 메소드를 제공한다.
    - `println!()`
    - `err.to_string()`
    - `err.source()`

Result

- 오류 출력하기
  - 오륫값을 출력하더라도 그 원인은 표시되지 않는다.
    오류에 관한 모든 정보를 빠짐없이 출력하고 싶다면 다음 함수를 사용하자.

```rust
use std::error::Error;
use std::io::{stderr, Write};

// Dump an error message to `stderr`.
//
// If another error happens while building the error message or writing to `stderr`, it is ignored.
fn print_error(mut err: &dyn Error) {
    let _ = writeln!(stderr(), "error: {err}");

    while let Some(source) = err.source() {
        let _ = writeln!(stderr(), "caused by: {source}");
        err = source;
    }
}
```

Result

- 오류 전파하기
  - 오류가 발생하면 보통은 호출부에 처리를 맡기고 싶어 한다. 즉, 호출 스택을 타고 **전파(Propagation)**되길 원한다.
    이를 위해 Rust에는 ? 연산자가 있다. ?는 함수 호출 결과와 같이 Result를 산출하는 모든 표현식에 붙여 쓸 수 있다.

```
let weather = get_weather(hometown)?;
```

  - 위 코드에서 ?의 동작은 함수가 성공 결과를 반환하는지, 오류 결과를 반환하는지에 따라 달라진다.
    - 성공한 경우에는 Result를 풀어서 그 안에 있는 성공 값을 꺼낸다.
      여기서 weather의 타입은 Result<WeatherReport, io::Error>가 아니라 그냥 WeatherReport다.
    - 오류가 발생한 경우에는 즉시 바깥쪽 함수에서 복귀하고 오류 결과를 호출 체인 위로 전달한다.
      이 때문에 ?는 리턴 타입이 Result인 함수에서만 쓸 수 있다.

Result

- 오류 전파하기
  - ? 연산자가 어떤 마법을 부리는 게 아니다. 좀 번거롭고 길지만 match 표현식으로도 이와 똑같은 처리를 할 수 있다.

```
let weather = match get_weather(hometown) {
    Ok(success_value) => success_value,
    Err(err) => return Err(err),
};
```

  - 프로그램에서 특히 운영체제와 맞닿아 있는 코드는 오류의 가능성이 곳곳에 배어 있다는 걸 잊지 말자.
    경우에 따라서는 함수의 거의 모든 줄에 ? 연산자가 붙기도 한다.

```
fn move_all(src: &Path, dst: &Path) -> io::Result<()> {
    for entry_result in src.read_dir()? {
        let entry = entry_result?;
        let dst_file = dst.join(entry.file_name());

        fs::rename(entry.path(), dst_file)?;
    }

    Ok(())
}
```

Result

- 여러 오류 타입 다루기
  - 가끔은 다뤄야 할 오류의 종류가 여러 개일 때도 있다. 예를 들어, 텍스트 파일에서 숫자들을 읽는다고 하자.

```rust
use std::io::{self, BufRead};

/// Read integers from a text file.
/// The file should have one number on each line.
fn read_numbers(file: &mut dyn BufRead) -> Result<Vec<i64>, io::Error> {
    let mut numbers = vec![];

    for line_result in file.lines() {
        let line = line_result?;     // Reading lines can fail
        numbers.push(line.parse()?); // Parsing integers can fail
    }

    Ok(numbers)
}
```

Result

- 여러 오류 타입 다루기
  - Rust는 앞의 코드에 대해서 다음과 같은 컴파일러 오류를 낸다.

```
error: `?` couldn't convert the error to `std::io::Error`

  numbers.push(line.parse()?); // Parsing integers can fail
                           ^
              the trait `std::convert::From<std::num::ParseIntError>`
              is not implemented for `std::io::Error`
```

Result

- 여러 오류 타입 다루기
  - 문제는 파일을 한 줄 읽을 때 발생할 수 있는 오류 타입과 정수를 파싱할 때 발생할 수 있는 오류 타입이 다르다는 데 있다.
    - `line_result`의 타입은 `Result<String, std::io::Error>`다.
    - `line.parse()`의 타입은 `Result<i64, std::num::ParseIntError>`다.
    - `read_numbers()` 함수의 리턴 타입은 `io::Error`만 수용한다. 따라서 `Result`는 `ParseIntError`를 `io::Error`로 변환해 대처하려고 하지만, 그런 변환은 정의되어 있지 않기 때문에 타입 오류가 발생하게 된다.
  - 이 문제를 해결하는 방법은 여러 가지가 있다.
    - 자체 오류 타입을 정의하고 다른 여러 오류 타입을 자체 오류 타입으로 바꾸는 변환을 구현한다. (`thiserror` 크레이트)
    - Rust에 내장된 기능을 이용한다. 표준 라이브러리의 모든 오류 타입은 다음과 같은 타입으로 변환될 수 있다.

```rust
type GenericError = Box<dyn std::error::Error + Send + Sync + 'static>;
type GenericResult<T> = Result<T, GenericError>;
```

Result

- '발생할 리 없는' 오류 다루기
  - 때로는 오류가 발생할 리 없다는 걸 알고 있을 때가 있다.

```
if next_char.is_digit(10) {
    let start = current_index;
    current_index = skip_digits(&line, current_index);
    let digits = &line[start..current_index];
    ...
```

  - 예를 들어 구성 파일을 파싱하는 코드를 작성 중이고, 파일의 다음 항목이 숫자 문자열인 상황이라고 하자.
  - 숫자 문자열을 실제 수로 바꿀 때는 표준 메소드인 `parse()`를 사용하면 된다.

```
let num = digits.parse::<u64>();
```

  - 문제는 `str.parse::<u64>()` 메소드가 `Result`를 반환하는 데 있다. 문자열이 다 숫자로 된 건 아니라 실패할 수 있기 때문이다.

```
"bleen".parse::<u64>() // ParseIntError: Invalid digit
```

Result

- '발생할 리 없는' 오류 다루기
  - 때로는 오류가 발생할 리 없다는 걸 알고 있을 때가 있다.

```
if next_char.is_digit(10) {
    let start = current_index;
    current_index = skip_digits(&line, current_index);
    let digits = &line[start..current_index];
    ...
```

  - 하지만 `digits`가 숫자로만 되어 있다는 걸 알고 있다면, 어떻게 하는 게 좋을까?
    가장 좋은 방법은 `Ok`이면 성공 값을 반환하고, `Err`이면 패닉에 빠지는 `Result` 메소드 `.unwrap()`을 쓰는 것이다.
  - 이 방법은 `?`와 비슷하다. 단, 판단 착오로 인해 발생할 리 없다고 여겼던 오류가 발생하게 되면 패닉에 빠진다는 점이 다르다.

```
let num = digits.parse::<u64>().unwrap();
```

Result

- 오류 무시하기
  - 가끔은 오류를 완전히 무시하고 싶을 때도 있다.
    하지만 그랬다가는 Rust 컴파일러가 사용하지 않는 `Result` 값이 있다며 경고를 내보낸다.

```
writeln!(stderr(), "error: {err}"); // Warning: Unused result
```

  - 이럴 때는 `let _ = ...` 관용구를 쓰면 경고를 잠재울 수 있다.

```
let _ = writeln!(stderr(), "error: {err}"); // OK, Ignore result
```

Result

- main()에서 오류 처리하기
  - 보통 main()은 리턴 타입이 Result가 아니라서 ?를 쓸 수 없다.
  - main()에서 오류를 처리하는 가장 간단한 방법은 .expect()를 쓰는 것이다.

```
fn main() {
    calculate_tides().expect("error");
}
```

  - 또는 main()의 타입 시그니처를 바꿔서 Result 타입을 반환하게 만들면 ?를 쓸 수 있다.

```
fn main() -> Result<(), TideCalcError> {
    let tides = calculate_tides()?;
    print_tides(tides);
    Ok(())
}
```

Result

- 사용자 정의 오류 타입 선언하기
  - 새 JSON 파서를 작성 중이고, 여기에 자체 오류 타입을 넣고 싶다고 하자.

```rust
#[derive(Debug, Clone)]
pub struct JsonError {
    pub message: String,
    pub line: usize,
    pub column: usize,
}
```

  - 이 타입의 오류를 발생시키고 싶을 때는 다음처럼 작성하면 된다.

```rust
return Err(JsonError {
    message: "expected ']' at end of array".to_string(),
    line: current_line,
    column: current_column
});
```

Result

- 사용자 정의 오류 타입 선언하기
  - 이렇게만 해도 충분하지만, 라이브러리 사용자들의 기대에 발맞춰 이 오류 타입을 표준 오류 타입처럼 동작하게 만들고 싶다면 할 일이 좀 더 남아 있다.

```rust
use std::fmt;

// Errors should be printable.
impl fmt::Display for JsonError {
    fn fmt(&self, f: &mut fmt::Formatter) -> Result<(), fmt::Error> {
        write!(f, "{} ({}:{})", self.message, self.line, self.column)
    }
}

// Errors should implement the std::error::Error trait,
// but the default definitions for the Error methods are fine.
impl std::error::Error for JsonError { }
```

Result

- 사용자 정의 오류 타입 선언하기
  - 오류 처리도 외부 크레이트의 도움을 받으면 일이 훨씬 더 쉽고 간단해진다.
    오류 처리용 크레이트는 종류가 꽤 다양한데, 그중에서도 `thiserror`가 가장 많이 쓰인다.

```rust
use thiserror::Error;

#[derive(Error, Debug)]
#[error("{message:} ({line:}, {column})")]
pub struct JsonError {
    message: String,
    line: usize,
    column: usize,
}
```

Result

- 왜 Result일까?
  - Rust는 오류가 발생할 수 있는 모든 위치에서 프로그래머가 모종의 결정을 내린 뒤
    그것을 코드에 기록할 것을 요구한다. 이렇게 하면 오류가 방치되어 잘못 처리되는 일이 줄기 때문에 좋다.
  - 가장 일반적인 결정은 오류가 전파되도록 만드는 것인데, 여기에 필요한 코드는 ? 한 문자 뿐이다.
    따라서 C와 Go처럼 오류 배관 작업으로 인해 코드가 어수선해지는 일이 없다.
    게다가 가독성이 좋아서 코드를 조금만 봐도 오류가 전파되는 모든 곳을 한눈에 파악할 수 있다.
  - 오류의 가능성이 모든 함수의 리턴 타입에 명시되어 있기 때문에 실패할 수 있거나 없는 함수를 명확히 구분할 수 있다.
    실패할 수 없는 함수를 실패할 수 있는 함수로 바꾸는 일은 곧 리턴 타입을 바꾸는 일이므로,
    컴파일러가 함수의 사용처를 모두 업데이트할 수 있도록 도와줄 것이다.
  - Rust는 Result 값의 사용 여부를 확인하기 때문에 실수로 오류를 무시하고 넘어가는 일이 생길 수 없다.
  - Result는 평범한 데이터 타입이므로 성공 결과와 오류 결과를 같은 컬렉션 안에 담을 수 있는데,
    이렇게 하면 부분적인 성공을 쉽게 모델링할 수 있다.

감사합니다.

utilForever@gmail.com

https://github.com/utilForever

Facebook, Twitter: @utilForever