# 세상에서 제일 쉬운 러스트 프로그래밍

## 10장. 에러 처리와 로깅

윤인도

freedomzero91@gmail.com

# 에러 처리의 철학

## 파이썬의 예외 처리

### LBYL(Look before you leap) : "도약하기 전에 살펴보세요"

- 이 코딩 스타일은 호출이나 조회를 하기 전에 명시적으로 전제 조건을 테스트
- 많은 `if` 문이 특징

```
if key in mapping:
    return mapping[key]
```

> 멀티 스레드 환경에서 LBYL 접근 방식은 '보기'와 '도약' 사이에 경쟁 조건이 발생할 위험 존재

## EAFP(Easier to ask for forgiveness than permission) : "용서받는 것이 허락받는 것보다 쉽다"

- 이 코딩 스타일은 유효한 키 또는 속성이 있다고 가정하고, 가정이 거짓으로 판명되면 예외를 포착
- 많은 `try except` 블럭이 특징

```
try:
    file = open("file.txt", "r")
except FileNotFoundError:
    print("File not found")
```

## 러스트의 에러 처리

### 컴파일 타임

LBYL: 컴파일러가 타입/소유권을 미리 검사

### 런타임

EAFP: `Result`를 사용해 에러를 적절히 처리

## panic!

파이썬에서 코드를 즉시 종료시키고 싶다면 `raise`로 에러를 발생

> 정확히는 예외와 에러는 다르지만, 편의상 둘을 묶어서 에러라고 하겠습니다.

```
raise Exception
```

실행 결과

```
Traceback (most recent call last):
  File "/temp/python/main.py", line 1, in <module>
    raise Exception
Exception
```

러스트에서 프로그램이 예상치 못한 오류로 종료되는 경우, 패닉이 발생

- 배열에 잘못된 인덱스를 참조할 때와 같이 패닉이 발생하는 코드를 실행한 경우
- `panic!` 매크로를 직접 실행하는 경우

패닉이 발생하는 코드를 실행(예: 배열에 잘못된 인덱스를 참조하는 경우)

```
fn main() {
    let nums = [1, 2, 3];

    for i in 0..4 {
        println!("{}", nums[i]);
    }
    println!("Finished");
}
```

실행 결과

```
1
2
3
thread 'main' panicked at src/main.rs:5:24:
index out of bounds: the len is 3 but the index is 3
```

`panic!`은 러스트에서 제공하는 편리한 매크로, 프로그램이 즉시 종료

```rust
fn main() {
    panic!("🤯");
}
```

실행 결과

```
thread 'main' panicked at '🤯', src/main.rs:2:5
note: run with `RUST_BACKTRACE=1` environment variable to display
a backtrace
```

여기서 어떤 코드가 문제인지를 알고 싶다면, 컴파일러가 알려준 대로 환경 변수 `RUST_BACKTRACE=1`를 사용해 컴파일하면 됩니다.

> 윈도에서는 `set RUST_BACKTRACE=1`을 먼저 실행한 다음 `cargo run`을 실행합니다.

```
RUST_BACKTRACE=1 cargo run
    Finished dev [unoptimized + debuginfo] target(s) in 0.05s
     Running `target/debug/rust_part`
thread 'main' panicked at '🤯', src/main.rs:2:5
stack backtrace:
   0: rust_begin_unwind
             at /rustc/library/std/src/panicking.rs:575:5
   1: core::panicking::panic_fmt
             at /rustc/library/core/src/panicking.rs:64:14
   2: chat_server::main
             at ./src/main.rs:2:5
   3: core::ops::function::FnOnce::call_once
             at /rustc/library/core/src/ops/function.rs:507:5
note: Some details are omitted, run with `RUST_BACKTRACE=full` for a verbose backtrace.
```

> 릴리즈 모드로 빌드할 경우, 백트레이스는 포함되지 않기 때문에 컴파일 시 별도의 디버그 심볼을 제공해주어야 합니다.

## unwrap

# "안에 들어 있는 내용물을 꺼내기 위해 포장을 뜯는다"

`Option`이나 `Result` 안에 있는 값을 꺼낸다

## unwrap

`unwrap` 함수는 입력받는 파라미터가 true라면 결괏값인 문자열을 `Some`으로 감싸서 리턴하고, `false`라면 `None`이 리턴

```
fn give_some_or_none(some: bool) -> Option<String> {
    if some {
        Some(String::from("💖"))
    } else {
        None
    }
}
```

함수의 결과가 항상 `Some`이라고 가정하고 `Some`의 내용물을 꺼낼 수 있음

```
fn main() {
    println!("{}", give_some_or_none(true).unwrap());
}
```

실행 결과

```
💖
```

그런데 만일 `give_some_or_none` 함수에 `false`가 주어진다면?

```
fn main() {
    println!("{}", give_some_or_none(false).unwrap());
}
```

실행 결과

```
thread 'main' panicked at 'called `Option::unwrap()` on a `None`
value', src/main.rs:10:45
```

- `unwrap`을 사용하면 `None`이 리턴되는 경우에 패닉이 발생!
- 따라서 `Option<T>`를 리턴하는 함수에 `unwrap`을 사용하려면 해당 함수가 반드시 `Some`을 리턴하는 것이 확실해야만 함
- 그렇지 않으면 `match`를 사용해 모든 경우를 처리

사실은 대부분의 경우에서 `unwrap`을 쓰지 않는 것이 가장 좋음

`Result` 열거형에도 `unwrap`을 사용가능

```rust
use std::fmt::Error;

fn give_ok_or_err(bool: bool) -> Result<String, Error> {
    if bool {
        Ok(String::from("💖"))
    } else {
        Err(Error)
    }
}
```

`Ok`가 리턴되면 `Some`으로 감싸진 값을 리턴

```
fn main() {
    let result = give_ok_or_err(true).unwrap();
    println!("{}", result);
}
```

실행 결과

```
💖
```

`Err`가 리턴되면 즉시 패닉이 발생하고 프로그램이 중단

```rust
fn main() {
    let result = give_ok_or_err(false).unwrap();
    println!("{}", result);
}
```

실행 결과

```
thread 'main' panicked at 'called `Result::unwrap()` on an `Err` value: Error', src/main.rs:12:40
note: run with `RUST_BACKTRACE=1` environment variable to display a backtrace
```

`Some` / `Ok` 인 경우는 그냥 값을 바로 사용하고, 에러가 발생하는 경우만 따로 처리하려면?

`if let ~` 을 사용해도 되지만, 함수형 프로그래밍답게 처리하는 세 가지 방법이 존재합니다.

## unwrap_or

> "unwrap"해봐, 만일 아니면...

- 결과가 `Some` / `Ok` 면 내부의 값
- `None` / `Err` 라면 함수에서 입력받은 값을 기본값으로 사용

## Option

```
fn main() {
    assert_eq!(Some("car").unwrap_or("bike"), "car");
    assert_eq!(None.unwrap_or("bike"), "bike");
}
```

## Result

```
fn main() {
    type StrResult<'a> = Result<&'a str, std::fmt::Error>;
    assert_eq!(StrResult::Ok("car").unwrap_or("bike"), "car");
    assert_eq!(StrResult::Err(std::fmt::Error {}).unwrap_or("bike"), "bike");
}
```

## unwrap_or_else

포함된 값을 리턴하거나 클로저에서 새로운 결과를 계산합니다.

이때 클로저에서는 환경 캡처가 가능합니다.

## Option

```
fn main(){
    let k = 10;
    assert_eq!(Some(4).unwrap_or_else(|| 2 * k), 4);
    assert_eq!(None.unwrap_or_else(|| 2 * k), 20);
}
```

## Result

```
fn main() {
    let k = 10;
    assert_eq!(Ok(4).unwrap_or_else(|_: &str| 2 * k), 4);
    assert_eq!(Err("foo").unwrap_or_else(|_: &str| 2 * k), 20);
}
```

## 참고 1

`unwrap_or_else`의 시그니처

```
pub fn unwrap_or_else<F>(self, op: F) -> T
where
    F: FnOnce(E) -> T,
```

클로저는 에러가 발생할 경우 에러 내용을 `&str` 타입으로 입력받아서 에러의 종류에 따른 서로 다른 결과물을 내는 것이 가능합니다.

## 참고 2

`unwrap_or`에 전달된 인수는 즉시 평가됩니다. 함수 호출의 결과를 전달하는 경우 느리게 평가되는 `unwrap_or_else`를 사용하는 것이 좋습니다.

```
fn main() {
    let closure = || {
        std::thread::sleep(std::time::Duration::from_secs(1));
        return 1;
    };
    let some = Some(1);

    let start = std::time::Instant::now();
    some.unwrap_or(closure());
    println!("unwrap_or: {:?}", std::time::Instant::now() - start);

    let start = std::time::Instant::now();
    some.unwrap_or_else(closure);
    println!("unwrap_or_else: {:?}", std::time::Instant::now() - start);
}
```

실행 결과

```
unwrap_or: 1.000254959s
unwrap_or_else: 42ns
```

`unwrap_or` 에서는 실제로는 값이 필요 없는데도 불구하고 클로저가 즉시 실 행돼 1초 간 지연이 발생하지만, `unwrap_or_else` 에서는 값이 필요없다면 클로저가 실행되지 않아 코드가 빨리 실행

## unwrap_or_default

포함된 Some 값 또는 기본값을 반환합니다.

self 인자를 소비한 다음 Some이면 포함된 값을 반환하고, None이면 해당 타입의 기본값을 반환합니다.

```
let x: Option<u32> = None;
let y: Option<u32> = Some(12);

assert_eq!(x.unwrap_or_default(), 0);
assert_eq!(y.unwrap_or_default(), 12);
```

## expect

- `unwrap`은 코드가 반드시 성공한다는 확신이 있을 때만 사용한
- 하지만 패닉이 발생한 이유나 해결 방법에 관해서는 설명하지 않음

```rust
use std::env;
fn main() {
    env::var("RUST_LOG").unwrap();
}
```

실행 결과

```
thread 'main' panicked at src/main.rs:3:26:
called `Result::unwrap()` on an `Err` value: NotPresent
note: run with `RUST_BACKTRACE=1` environment variable to display a backtrace
```

사용자가 지정한 메세지로 추가 설명이 가능

```
use std::env;
fn main() {
    env::var("RUST_LOG").expect("Run setup.sh before running this program");
}
```

실행 결과

```
thread 'main' panicked at 'Run setup.sh before running this program', src/main.rs:3:26
note: run with `RUST_BACKTRACE=1` environment variable to display a backtrace
```

## 커스텀 예외 정의하기

프로그램의 특정 부분에서 발생할 수 있는 문제들을 유형화해서 제대로 처리할 수 있도록 하는 방법

단순히 파일이 없는 것이 아닌, 아직 다운로드되지 않았음 : `FileNotDownloaded`

```
import os
def get_content():
     filepath = os.path.join(os.path.pardir, "test.txt")
     with open(filepath, "r") as f:
         return f.read()
class FileNotDownloaded(Exception):
     pass
if __name__ == '__main__':
    try:
        content = get_content()
        print(content)
    except FileNotFoundError as exc:
        raise FileNotDownloaded("File is not yet downloaded") from None
```

실행 결과

```
Traceback (most recent call last):
   File "/temp/python/main.py", line 19, in <module>
    raise FileNotDownloaded("File is not yet downloaded") from None
__main__.FileNotDownloaded: File is not yet downloaded
```

러스트 코드

```rust
use std::{ fmt, fs::File, io::{Error, ErrorKind, Read}, path::Path };

fn get_content() -> Result<String, Error> {
    let mut content = String::new();
    let filepath = Path::new(std::env::current_dir().unwrap()
        .parent().unwrap()).join("test.txt");
    File::open(filepath)?.read_to_string(&mut content)?;
    Ok(content)
}

#[derive(Debug, Clone)]
struct FileNotDownloaded;

impl fmt::Display for FileNotDownloaded {
    fn fmt(&self, f: &mut fmt::Formatter) -> fmt::Result {
        write!(f, "File is not yet downloaded!")
    }
}
```

```rust
fn main() {
    let content = get_content().unwrap_or_else(|_| {
        panic!("{}", FileNotDownloaded);
    });
    println!("{}", content);
}
```

실행 결과

```
thread 'main' panicked at 'File is not yet downloaded', src/main.rs:24:9
note: run with `RUST_BACKTRACE=1` environment variable to display a backtrace
```

`match` 문을 사용해 에러 종류에 따라 다른 처리

```
#[derive(Debug, Clone)]
struct FileNotDownloaded;
impl fmt::Display for FileNotDownloaded {
    fn fmt(&self, f: &mut fmt::Formatter) -> fmt::Result {
        write!(f, "File is not yet downloaded")
    }
}
#[derive(Debug, Clone)]
struct WrongContentError;
impl fmt::Display for WrongContentError {
    fn fmt(&self, f: &mut fmt::Formatter) -> fmt::Result {
        write!(f, "Unexpected content")
    }
}
enum GetContentError {
    FileNotDownloaded,
    WrongContentError,
}
```

```rust
fn get_content() -> Result<String, GetContentError> {
    let mut content = String::new();
    let filepath = Path::new(std::env::current_dir().unwrap()
        .parent().unwrap()).join("test.txt");
    match File::open(filepath) {
        Ok(mut file) => {
            file.read_to_string(&mut content).unwrap();
            if content != "Hello Rust!" {
                return Err(GetContentError::WrongContentError);
            }
            Ok(content)
        }
        Err(_) => return Err(GetContentError::FileNotDownloaded),
    }
}
```

```rust
fn main() {
    match get_content() {
        Ok(content) => println!("{}", content),
        Err(e) => match e {
            GetContentError::FileNotDownloaded => println!(
                "Please download the file first"
            ),
            GetContentError::WrongContentError => println!(
                "Make sure you downloaded the
                correct file"
            ),
        },
    }
}
```

# 에러 로깅

## 😇 러스트는 로그 설정이 꽤나 까다롭다

- 내장 크레이트인 `log`가 존재하지만, 인터페이스(또는 파사드façade)만 정의돼 있음
- 따라서 서드파티 크레이트를 반드시 설치
- 여기서는 가장 인기 있는 `tracing`과 `tracing-subscriber`를 사용

`Cargo.toml` 파일에 다음을 추가

```
[dependencies]
tracing = "0.1.37"
tracing-subscriber = "0.3.17"
```

## 기본 사용법

```rust
use tracing::{debug, error, info, warn};
use tracing_subscriber::fmt;

fn main() {
    fmt()
        .with_max_level(tracing::Level::DEBUG) // 로그 레벨 설정
        .event_format(fmt::format().with_thread_ids(true) // 출력 내용 설정
        .with_thread_names(true)) // 여기서는 스레드 ID와 이름 출력
        .init();
    debug!("This is a debug message");
    info!("This is an info message");
    warn!("This is a warning message");
    error!("This is an error message");
}
```

실행 결과

```
2023-08-16T08:00:58.710375Z DEBUG main ThreadId(01) rust_part: This is a debug message
2023-08-16T08:00:58.710468Z  INFO main ThreadId(01) rust_part: This is an info message
2023-08-16T08:00:58.710493Z  WARN main ThreadId(01) rust_part: This is a warning message
2023-08-16T08:00:58.710516Z ERROR main ThreadId(01) rust_part: This is an error message
```

`util.rs` 파일을 만들고 다음 코드를 추가

```rust
use tracing::{error, warn};

pub fn sqaure(num: i32) -> i32 {
    if num == 0 {
        error!("num is 0!");
        return num;
    } else if num == 1 {
        warn!("num is 1!");
        return num;
    }
    num * num
}
```

`main.rs` 파일에 다음 코드를 추가

```rust
mod util;
use tracing::{debug, info};
use tracing_subscriber::fmt;
use util::sqaure;
fn main() {
    info!("Starting..."); // 출력되지 않음!
    fmt()
        .with_max_level(tracing::Level::DEBUG)
        .event_format(fmt::format().with_thread_ids(true)
        .with_thread_names(true))
        .init();
    for i in 0..3 {
        debug!("i = {} and i^2 is {}", i, sqaure(i));
    }
}
```

실행 결과

```
2023-08-16T12:50:41.068772Z ERROR main ThreadId(01) rust_part::util: num is 0!
2023-08-16T12:50:41.068838Z DEBUG main ThreadId(01) rust_part: i = 0 and i^2 is 0
2023-08-16T12:50:41.068853Z  WARN main ThreadId(01) rust_part::util: num is 1!
2023-08-16T12:50:41.068863Z DEBUG main ThreadId(01) rust_part: i = 1 and i^2 is 1
2023-08-16T12:50:41.068872Z DEBUG main ThreadId(01) rust_part: i = 2 and i^2 is 4
```

> tracing이 아닌 다른 트레이트를 사용하더라도, 로깅 설정을 하는 부분을 제외하고는 대부분 `debug!`, `error!` 와 같은 매크로를 같은 이름으로 사용합니다.