파이썬 프로그래머를 위한 러스트 입문

윤인도

freedomzero91@gmail.com

CH10. 에러 처리

파이썬의 예외 처리

LBYL

도약하기 전에 살펴보세요(Look before you leap). 이 코딩 스타일은 호출이나 조회를 하기 전에 명시적으로 전제 조건을 테스트합니다. 이 스타일은 많은 `if` 문이 있다는 특징이 있습니다.

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

> 멀티 스레드 환경에서 LBYL 접근 방식은 '보기'와 '도약' 사이에 경쟁 조건이 발생할 위험이 있습니다.

EAFP

허락보다 용서받는 것이 더 쉽습니다(Easier to ask for forgiveness than permission). 이 코딩 스타일은 유효한 키 또는 속성이 있다고 가정하고, 가정이 거짓으로 판명되면 예외를 포착합니다. 이 깔끔하고 빠른 스타일은 많은 `try except` 블럭이 있다는 특징이 있습니다.

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

러스트의 에러 처리

따라서 Rust는 LBYL 또는 EAFP와 엄격하게 일치하지는 않지만, 두 접근 방식과 몇 가지 유사점을 공유하며 두 스타일의 장점을 결합한 고유한 오류 처리 방식을 제공합니다.

- `panic!`를 사용하여 치명적인 오류를 처리하는 방법
- 값이 선택 사항이거나 값이 없어도 오류 조건이 아닌 경우 `Option` 열거형을 사용하는 방법
- 오류가 발생할 수 있고 호출자가 문제를 처리해야 하는 경우 `Result` 열거형을 사용하는 방법

panic!

파이썬에서 코드를 즉시 종료시키고 싶다면 예외를 발생시키면 됩니다. 어떤 종류의 예외든 상관없지만, 가장 일반적인 경우를 발생시켜 보면 다음과 같습니다.

```
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 발생한다고 합니다. 패닉이 발생하는 경우는 두 가지입니다.

- 패닉이 발생하는 코드를 실행(예: 배열에 잘못된 인덱스를 참조하는 경우)
- `panic!` 매크로를 직접 실행하는 경우

`panic!`은 러스트에서 제공하는 편리한 매크로로, 프로그램이 즉시 종료됩니다.

```
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.

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

```
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/d5a82bbd26e1ad8b7401f6a718a9c57c96905483/library/std/src/panicking.rs:575:5
   1: core::panicking::panic_fmt
             at /rustc/d5a82bbd26e1ad8b7401f6a718a9c57c96905483/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/d5a82bbd26e1ad8b7401f6a718a9c57c96905483/library/core/src/ops/function.rs:507:5
note: Some details are omitted, run with `RUST_BACKTRACE=full` for a verbose backtrace.
```

Option 열거형

다음과 같은 함수를 생각해 보겠습니다. 입력받는 파라미터에 따라서 결과값이 있을 수도 있고, 없을 수도 있습니다.

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

어떤 계산의 결과가 비어 있거나 없을 가능성이 있는 경우, std 라이브러리의 `Option<T>` 을 써서 이를 표현할 수 있습니다.

암묵적으로 함수의 결과를 처리하는 방법인 `unwrap` 을 사용할 수도 있습니다.

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

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

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

```
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
```

여기서 알 수 있듯이, `unwrap`을 사용하면 `None`이 리턴되는 경우에 패닉이 발생합니다.

따라서 `Option<T>`를 리턴하는 함수에 `unwrap`을 사용하려면 해당 함수가 반드시 `Some`을 리턴하는 것이 확실해야만 합니다.

그렇지 않으면 `match`를 사용해 모든 경우를 처리하는 게 올바른 방법입니다.

unwrap_or...

하지만 때로는 `Some` 인 경우는 그냥 값을 바로 사용하고, 에러가 발생하는 경우만 따로 처리하고 싶은 경우가 있습니다. `if let Some` 을 사용해도 되지만, 함수형 프로그래밍답게 처리하는 세 가지 방법이 존재합니다.

unwrap_or

포함된 Some 값 또는 제공된 기본값을 반환합니다.

unwrap_or에 전달된 인수는 즉시 평가됩니다. 함수 호출의 결과를 전달하는 경우 느리게 평가되는 unwrap_or_else를 사용하는 것이 좋습니다.

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

unwrap_or_else

포함된 Some 값을 반환하거나 클로저에서 계산합니다.

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

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);
```

?

결과가 `Some` 이 아닌 경우 즉시 함수를 종료하고 `None` 을 반환합니다.

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

fn question_mark(bool: bool) -> Option<String> {
    let some = give_some_or_none(bool)?;
    Some(some)
}

fn main() {
    println!("{:?}", question_mark(true));
    println!("{:?}", question_mark(false));
}
```

Result 열거형

`Result` 열거형은 `Option` 열거형과 비슷하지만, `Option` 열거형은 `Some`과 `None`만을 가지고 있지만, `Result` 열거형은 `Ok`와 `Err`를 가지고 있습니다.

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

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

fn main() {
    for bool in [true, false].iter() {
        let result = give_ok_or_err(*bool);
        match result {
            Ok(value) => println!("value: {}", value),
            Err(e) => println!("error: {}", e),
        }
    }
}
```

Unwrap...

unwrap

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

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

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

unwrap_or

`unwrap_or`는 `Result<T, E>`를 `T`로 변환합니다. 결과가 `Ok(v)`이면 `v`를 반환하고, `Err(e)`이면 인수를 반환합니다.

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

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

fn main() {
    let result = give_ok_or_err(false).unwrap_or(String::from("💔"));
    println!("{}", result);
}
```

ok_or...

`ok_or`와 `ok_or_else`는 Option<T>를 Result<T, E>로 변환합니다.

ok_or

`Option<T>`를 `Result<T, E>`로 변환하여 일부(`v`)를 `Ok(v)`로, 없음(`None`)을 `Err(err)`로 매핑합니다.

`ok_or에` 전달된 인수는 즈ㄱ시 평가되며, 함수 호출의 결과를 전달하는 경우 느리게 평가되는 `ok_or_else`를 사용하는 것이 좋습니다.

```rust
fn main() {
    let x = Some("foo");
    assert_eq!(x.ok_or(0), Ok("foo"));

    let x: Option<&str> = None;
    assert_eq!(x.ok_or(0), Err(0));
}
```

ok_or_else

`Option<T>`를 `Result<T, E>`로 변환하여 일부(v)를 Ok(v)로, 없음을 Err(err())로 매핑합니다.

```
fn main() {
    let x = Some("foo");
    assert_eq!(x.ok_or_else(|| 0), Ok("foo"));

    let x: Option<&str> = None;
    assert_eq!(x.ok_or_else(|| 0), Err(0));
}
```

?

결과가 `Ok` 이 아닌 경우 즉시 함수를 종료하고 `Err` 을 반환합니다.

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

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

fn question_mark(bool: bool) -> Result<String, Error> {
    let result = give_ok_or_err(bool)?;
    Ok(result)
}

fn main() {
    let result = question_mark(true);
    println!("{:?}", result);
    let result = question_mark(false);
    println!("{:?}", result);
}
```

커스텀 예외 정의하기

```
import os


def get_content():
    filepath = os.path.join(os.path.pardir, "test.txt")
    with open(filepath, "r") as f:
        return f.read()


class CustomError(Exception):
    pass


if __name__ == '__main__':
    try:
        with open("hello.txt", "r") as file:
            file.read()
    except FileNotFoundError as exc:
        print(exc)
    except:
        print("Unexpected error")

    try:
        content = get_content()
    except:
        raise CustomError
    print(content)
```

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

impl fmt::Display for CustomError {
    fn fmt(&self, f: &mut fmt::Formatter) -> fmt::Result {
        write!(f, "File not found!")
    }
}

fn main() {
    let file = File::open("hello.txt");
    match file {
        Ok(file) => println!("{:?}", file),
        Err(error) => match error.kind() {
            ErrorKind::NotFound => println!("{:?}", error),
            _ => println!("Unexpected error"),
        },
    }

    let content = get_content().unwrap_or_else(|_| {
        panic!("{}", CustomError);
    });
    println!("{}", content);
}
```

에러 로깅

```
[package]
name = "rust_part"
version = "0.1.0"
edition = "2021"

# See more keys and their definitions at https://doc.rust-lang.org/cargo/reference/manifest.html

[dependencies]
log = "0.4"
pretty_env_logger = "0.4.0"


[dev-dependencies]
```

기본 사용법

```
extern crate pretty_env_logger;
#[macro_use] extern crate log;

fn main() {
    pretty_env_logger::init();

    trace!("a trace example");
    debug!("deboogging");
    info!("such information");
    warn!("o_O");
    error!("boom");
}
```

```
RUST_LOG=trace cargo run
```

```
 TRACE rust_part > a trace example
 DEBUG rust_part > deboogging
 INFO  rust_part > such information
 WARN  rust_part > o_O
 ERROR rust_part > boom
```

커스터마이징

```
use log::{debug, error, info, trace, warn};

const LOG_LEVEL: log::Level = log::Level::Trace;

fn main() {
    let mut log_builder = pretty_env_logger::formatted_timed_builder();
    log_builder.parse_filters(LOG_LEVEL.as_str()).init();

    trace!("a trace example");
    debug!("deboogging");
    info!("such information");
    warn!("o_O");
    error!("boom");
}
```

```
2023-03-04T10:21:20.575Z TRACE rust_part > a trace example
2023-03-04T10:21:20.575Z DEBUG rust_part > deboogging
2023-03-04T10:21:20.575Z INFO  rust_part > such information
2023-03-04T10:21:20.575Z WARN  rust_part > o_O
2023-03-04T10:21:20.575Z ERROR rust_part > boom
```