파이썬 프로그래머를 위한 러스트 입문

윤인도

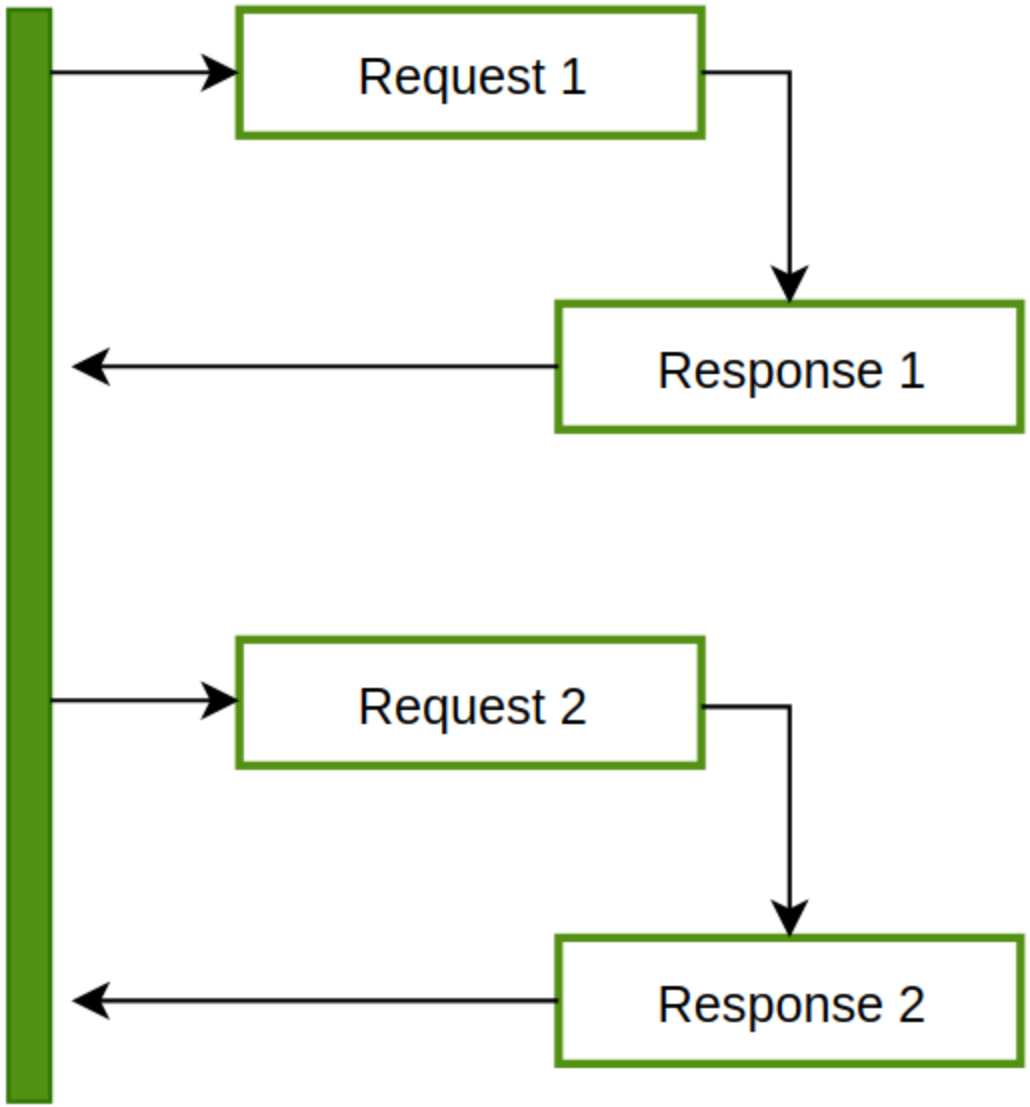
freedomzero91@gmail.com

CH13. 비동기 프로그래밍

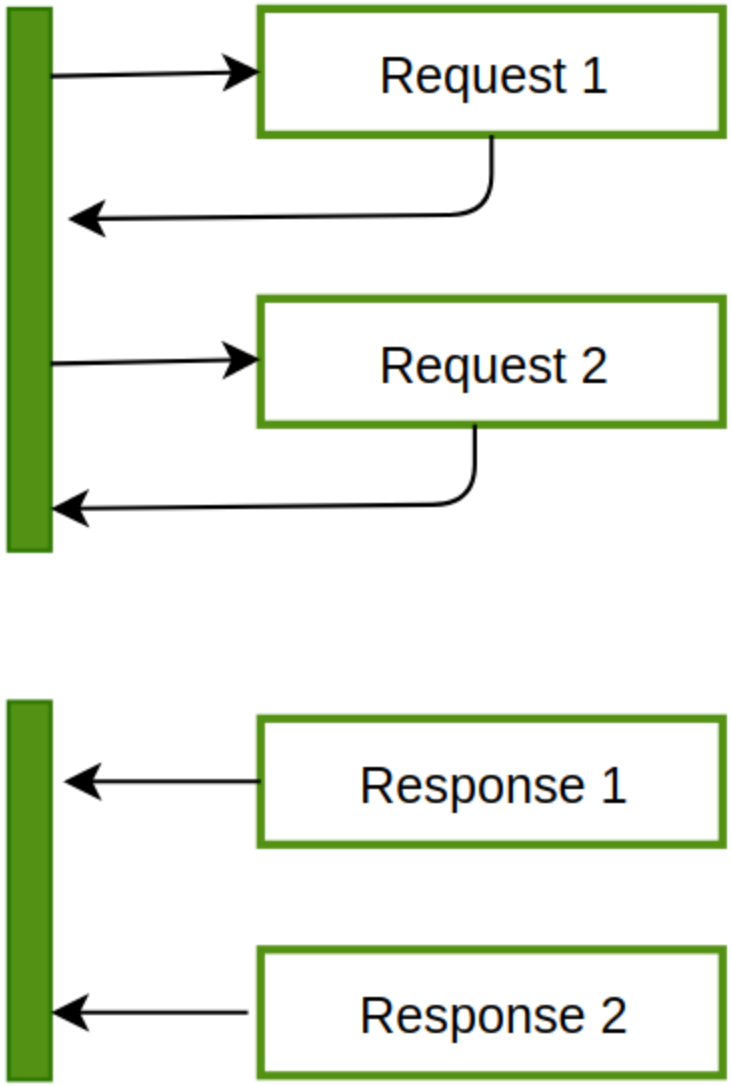
비동기 프로그래밍이란?

비동기 모델에서는 여러 가지 일이 동시에 일어날 수 있습니다. 프로그램에서 오래 실행되는 함수를 호출해도 실행 흐름이 차단되지 않고 프로그램이 계속 실행됩니다. 함수가 완료되면 프로그램은 결과를 알고 액세스합니다.

Synchronous



Asynchronous



비동기 작동 방식

협력적 멀티태스킹

- 파이썬 : 코루틴
- 러스트 : 태스크

운영체제 -> 스레드
런타임 -> 코루틴/태스크

> 협력적 멀티태스킹이란, 멀티태스킹에 참여하는 주체들이 언제든 자신의 실행을 자발적으로 멈추고 다른 주체에게 실행 권한을 넘기는 방식을 의미합니다. 운영체제에서는 스케줄러가 어떤 스레드가 언제 작업을 실행하고 종료할지를 관리하기 때문에 멀티스레딩을 협력적 멀티태스킹이 아닙니다.

tokio

앞에서 설명했듯이, 비동기 함수를 설명하려면 프로그램 내부에서 비동기 함수들의 실행을 관리하는 비동기 런타임이 필요합니다. 러스트는 빌트인 async 런타임이 존재하지 않아서 `tokio` 를 사용해야 합니다. Tokio는 다음 세 가지의 주요 구성 요소를 제공합니다.

- 비동기 코드 실행을 위한 멀티스레드 런타임.
- 표준 라이브러리의 비동기 버전.
- 대규모 라이브러리 에코시스템.

이제 프로젝트에 `tokio` 를 설치하기 위해 `Cargo.toml` 파일에 `tokio` 를 추가합니다. 이때 `features = ["full"]` 을 넣어야 전체 기능을 다 사용할 수 있습니다.

```
[dependencies]
tokio = { version = "1.25.0", features = ["full"] }
```

비동기 함수 만들어보기

파이썬에는 비동기 함수를 만들기 위해서 `asyncio` 라는 내장 라이브러리가 있습니다.

비동기 함수를 선언하려면 일반 함수 정의 앞에 `async` 키워드를 붙여주면 됩니다.

```
async def func():
    print("async")
```

비동기 함수는 그냥 호출하게 되면 결과 대신 `coroutine` 객체가 리턴되고, 경고가 발생합니다.

```
>>> print(func())

<coroutine object asynch at 0x10fe8e3b0>
/Users/temp/python/main.py:9: RuntimeWarning: coroutine 'func' was never awaited
  print(func())
RuntimeWarning: Enable tracemalloc to get the object allocation traceback
```

비동기 함수를 호출하려면, `asyncio.run` 과 `await` 키워드를 같이 사용해야 합니다.

```
import asyncio

async def func():
    print("async")

async def main():
    await func()

asyncio.run(main())
```

`main` 을 `async` 로 바꾸고, `#[tokio::main]` 를 `main` 함수에 붙여서 비동기 메인 함수를 만들 수 있습니다. 이제 메인 함수가 비동기 함수이기 때문에 내부에서 `async` 와 `await` 키워드를 사용 가능합니다.

```
use tokio;

async fn func() {
    println!("async");
}

#[tokio::main]
async fn main() {
    func().await
}
```

그렇다면 비동기 함수를 기다리지 않고 프린트해보면 어떨까요? 일단 아래 코드는 컴파일되지 않습니다.

```
use tokio;

async fn func() {
    println!("async");
}

#[tokio::main]
async fn main() {
    println!("{:?}", func()) // 🤯
}
```

실행 결과

```
error[E0277]: `impl Future<Output = i32>` doesn't implement `std::fmt::Display`
 --> src/main.rs:9:22
  |
9 |     println!("{:#}", func())
  |                       ^^^^^^ `impl Future<Output = i32>` cannot be formatted
  |  with the default formatter
```

컴파일 에러를 들여다보면, 함수 `func` 의 리턴값이 `impl Future<Output = i32>` 라는 것을 알 수 있습니다.

> 자바스크립트나 C#과 같은 언어의 비동기 함수들은 프라미스(Promise) 기반으로 만들어져 있어서 비동기 함수를 호출하는 즉시 실행되지만, 파이썬과 러스트는 실행할 수 있을 때 실행하는 방식(lazy execution)을 사용하고 있습니다.

하지만 단순히 `await` 를 사용하기만 해서는 기존의 동기 함수와 비슷한 결과가 나옵니다.

```rust
async fn hello() -> i32 {
    println!("Hello World!");
    tokio::time::sleep(std::time::Duration::from_secs(3)).await;
    println!("waited");
    1
}

async fn bye() -> i32 {
    println!("Goodbye!");
    2
}

#[tokio::main]
async fn main() {
    hello().await; // blocking
    bye().await;
}
```

실행 결과

```
Hello World!
waited
Goodbye!
```

리턴값이 있는 비동기 함수

만일 비동기 함수에서 값을 리턴한다면 그 값을 어떻게 받을 수 있을까요? 파이썬과 러스트 모두, 단순히 기존 동기 함수처럼 코드를 작성하고, 함수 정의에 `async` 키워드만 붙여주면 됩니다. 먼저 파이썬의 경우는 다음과 같습니다. `await` 키워드를 붙여서 호출하고 나면 함수의 리턴값을 얻을 수 있습니다.

```python
import asyncio


async def func():
    return "async"


async def main():
    result = await func()
    print(result)


asyncio.run(main())
```

실행 결과

```
async
```

러스트에서도 함수의 리턴값은 `await` 을 사용하고 나면 얻을 수 있습니다.

```
use tokio;

async fn func<'a>() -> &'a str {
    "async"
}

#[tokio::main]
async fn main() {
    println!("{:?}", func().await)
}
```

실행 결과

```
"async"
```

여러 작업 실행하기

`asyncio.gather` 를 사용하면 여러 개의 비동기 함수를 한꺼번에 실행할 수도 있습니다.

```
import asyncio

async def give_order(order):
    print(f"Processing {order}...")
    await asyncio.sleep(3 - order)
    print(f"Finished {order}")

async def main():
    await asyncio.gather(give_order(1), give_order(2), give_order(3))

asyncio.run(main())
```

실행 결과

```
Processing 1...
Processing 2...
Processing 3...
Finished 3
Finished 2
Finished 1
```

만일 각 함수에 리턴값이 있다면 값이 모아져서 리턴됩니다.

```
import asyncio

async def give_order(order):
    print(f"Processing {order}...")
    await asyncio.sleep(3 - order)
    print(f"Finished {order}")
    return order

async def main():
    results = await asyncio.gather(give_order(1), give_order(2), give_order(3))
    print(results)

asyncio.run(main())
```

실행 결과

```
Processing 1...
Processing 2...
Processing 3...
Finished 3
Finished 2
Finished 1
[1, 2, 3]
```

러스트에서는 `tokio::join!` 에 기다리고자 하는 함수들을 넣어주면 됩니다.

```rust
async fn give_order(order: u64) -> u64 {
    println!("Processing {order}...");
    tokio::time::sleep(std::time::Duration::from_secs(3 - order)).await;
    println!("Finished {order}");
    order
}

#[tokio::main]
async fn main() {
    let result = tokio::join!(give_order(1), give_order(2), give_order(3));

    println!("{:?}", result);
}
```

실행 결과

```
Processing 1...
Processing 2...
Processing 3...
Finished 3
Finished 2
Finished 1
(1, 2, 3)
```

예제: 빠르게 HTTP 요청 보내보기

파이썬 동기

```
import requests
from random import randint


# The highest Pokemon id
MAX_POKEMON = 898


def fetch(total):
    urls = [
        f"https://pokeapi.co/api/v2/pokemon/{randint(1, MAX_POKEMON)}"
        for _ in range(total)
    ]
    with requests.Session() as session:
        for url in urls:
            response = session.get(url).json()
            yield response["name"]


def main():
    for name in fetch(10):
        print(name)

    print([name for name in fetch(10)])


main()
```

`Cargo.toml` 에 추가

- `rand` : 랜덤값 생성
- `reqwest` : HTTP 요청
- `serde_json` : 데이터 직렬/역직렬화

```
rand = "0.8.5"
reqwest = { version="0.11.16", features = ["blocking", "json"] }
serde_json = "1.0.95"
```

`rand::thread_rng()` 로 1부터 898까지의 무작위 난수 생성

`reqwest::blocking::Client::new()` 동기 방식의 HTTP 클라이언트 생성

`.json::<serde_json::Value>()` 응답 json의 형식을 미리 알 수 없는 경우 사용, 일반적으로는 응답 형식을 알고 있어서 구조체를 사용해 타입을 명시

```rust
use rand::Rng;
use reqwest;
use serde_json;

const MAX_POKEMON: u32 = 898;

fn fetch(total: u32) -> Vec<String> {
    let mut urls = Vec::new();
    for _ in 0..total {
        let url = format!(
            "https://pokeapi.co/api/v2/pokemon/{}",
            rand::thread_rng().gen_range(1..=MAX_POKEMON)
        );
        urls.push(url);
    }
    let client = reqwest::blocking::Client::new();
    let mut names = Vec::new();
    for url in urls {
        let response = client
            .get(&url)
            .send()
            .unwrap()
            .json::<serde_json::Value>()
            .unwrap();
        names.push(response["name"].as_str().unwrap().to_string());
    }
    names
}

fn main() {
    for name in fetch(10) {
        println!("{}", name);
    }

    println!("{:?}", fetch(10));
}
```

실행 결과

```
grumpig
dartrix
celesteela
piloswine
tangrowth
virizion
glastrier
dewpider
hattrem
glameow
['boltund', 'gourgeist-average', 'shellos', 'shiinotic', 'eevee', 'cranidos', 'celesteela',
'solosis', 'houndour', 'landorus-incarnate']
```

파이썬 비동기

```
import asyncio
import aiohttp
from random import randint


# The highest Pokemon id
MAX_POKEMON = 898


async def _fetch(session, url):
    async with session.get(url) as response:
        return await response.json()


async def fetch(total):
    urls = [
        f"https://pokeapi.co/api/v2/pokemon/{randint(1, MAX_POKEMON)}"
        for _ in range(total)
    ]
    async with aiohttp.ClientSession() as session:
        tasks = [_fetch(session, url) for url in urls]
        responses = await asyncio.gather(*tasks)
        for response in responses:
            yield response["name"]


async def main():
    async for name in fetch(10):
        print(name)

    print([name async for name in fetch(10)])


asyncio.run(main())
```

러스트 비동기

```rust
use rand::Rng;
use reqwest;

const MAX_POKEMON: u32 = 898;

async fn fetch(total: u32) -> Vec<String> {
    let mut urls = Vec::new();
    for _ in 0..total {
        let url = format!(
            "https://pokeapi.co/api/v2/pokemon/{}",
            rand::thread_rng().gen_range(1..=MAX_POKEMON)
        );
        urls.push(url);
    }
    let client = reqwest::Client::new();
    let mut names = Vec::new();
    for url in urls {
        let response = client
            .get(&url)
            .send()
            .await
            .unwrap()
            .json::<serde_json::Value>()
            .await
            .unwrap();
        names.push(response["name"].as_str().unwrap().to_string());
    }
    names
}

#[tokio::main]
async fn main() {
    for name in fetch(10).await {
        println!("{}", name);
    }

    println!("{:?}", fetch(10).await);
}
```

rayon

비동기 프로그래밍과는 큰 상관이 없지만, `tokio` 와 자주 비교되는 크레이트인 `rayon` 에 대해서 살펴보겠습니다.

tokio vs rayon

Tokio는 비동기 네트워크 애플리케이션을 구축하는 데 이상적이며, Rayon은 대규모 데이터 컬렉션에 대한 계산을 병렬화하는 데 이상적입니다.

병렬 이터레이터

```
cargo add rayon
```

공식 문서에서 권장하는 사용 방법은 `prelude` 밑에 있는 모든 것을 불러오는 것입니다. 이렇게 하면 병렬 이터레이터와 다른 트레이트를 전부 불러오기 때문에 코드를 훨씬 쉽게 작성할 수 있습니다.

```
use rayon::prelude::*;
```

기존의 순차 계산 함수에 병렬성을 더하려면, 단순히 이터레이터를 `par_iter`로 바꿔주기만 하면 됩니다.

```
use rayon::prelude::*;
use std::time::SystemTime;

fn sum_of_squares(input: &Vec<i32>) -> i32 {
    input
        .par_iter() // ✨
        .map(|&i| {
            std::thread::sleep(std::time::Duration::from_millis(10));
            i * i
        })
        .sum()
}

fn sum_of_squares_seq(input: &Vec<i32>) -> i32 {
    input
        .iter()
        .map(|&i| {
            std::thread::sleep(std::time::Duration::from_millis(10));
            i * i
        })
        .sum()
}

fn main() {
    let start = SystemTime::now();
    sum_of_squares(&(1..100).collect());
    println!("{}ms", start.elapsed().unwrap().as_millis());
    let start = SystemTime::now();
    sum_of_squares_seq(&(1..100).collect());
    println!("{}ms", start.elapsed().unwrap().as_millis());
}
```

실행 결과

```
106ms
1122ms
```

par_iter_mut 는 각 원소의 가변 레퍼런스를 받는 이터레이터입니다.

```
use rayon::prelude::*;

use std::time::SystemTime;

fn plus_one(x: &mut i32) {
    *x += 1;
    std::thread::sleep(std::time::Duration::from_millis(10));
}

fn increment_all_seq(input: &mut [i32]) {
    input.iter_mut().for_each(plus_one);
}

fn increment_all(input: &mut [i32]) {
    input.par_iter_mut().for_each(plus_one);
}

fn main() {
    let mut data = vec![1, 2, 3, 4, 5];

    let start = SystemTime::now();
    increment_all(&mut data);
    println!("{:?} - {}ms", data, start.elapsed().unwrap().as_millis());

    let start = SystemTime::now();
    increment_all_seq(&mut data);
    println!("{:?} - {}ms", data, start.elapsed().unwrap().as_millis());
}
```

실행 결과

```
[2, 3, 4, 5, 6] - 12ms
[3, 4, 5, 6, 7] - 55ms
```

`par_sort` 는 병합 정렬을 응용한 정렬 알고리즘을 사용해 데이터를 병렬적으로 분할해 정렬합니다.

```rust
use rand::Rng;
use rayon::prelude::*;

use std::time::SystemTime;

fn main() {
    let mut rng = rand::thread_rng();
    let mut data1: Vec<i32> = (0..1_000_000).map(|_| rng.gen_range(0..=100)).collect();
    let mut data2 = data1.clone();

    let start = SystemTime::now();
    data1.par_sort();
    println!("{}ms", start.elapsed().unwrap().as_millis());

    let start = SystemTime::now();
    data2.sort();
    println!("{}ms", start.elapsed().unwrap().as_millis());

    assert_eq!(data1, data2);
}
```

실행 결과

```
68ms
325ms
```

Rayon 사용 시 주의사항

멀티스레드도 마찬가지지만 스레드 스폰 및 조인에 시간이 소요되기 때문에 주의