

세상에서 제일 쉬운 러스트 프로그래밍

13장. 비동기 프로그래밍

윤인도

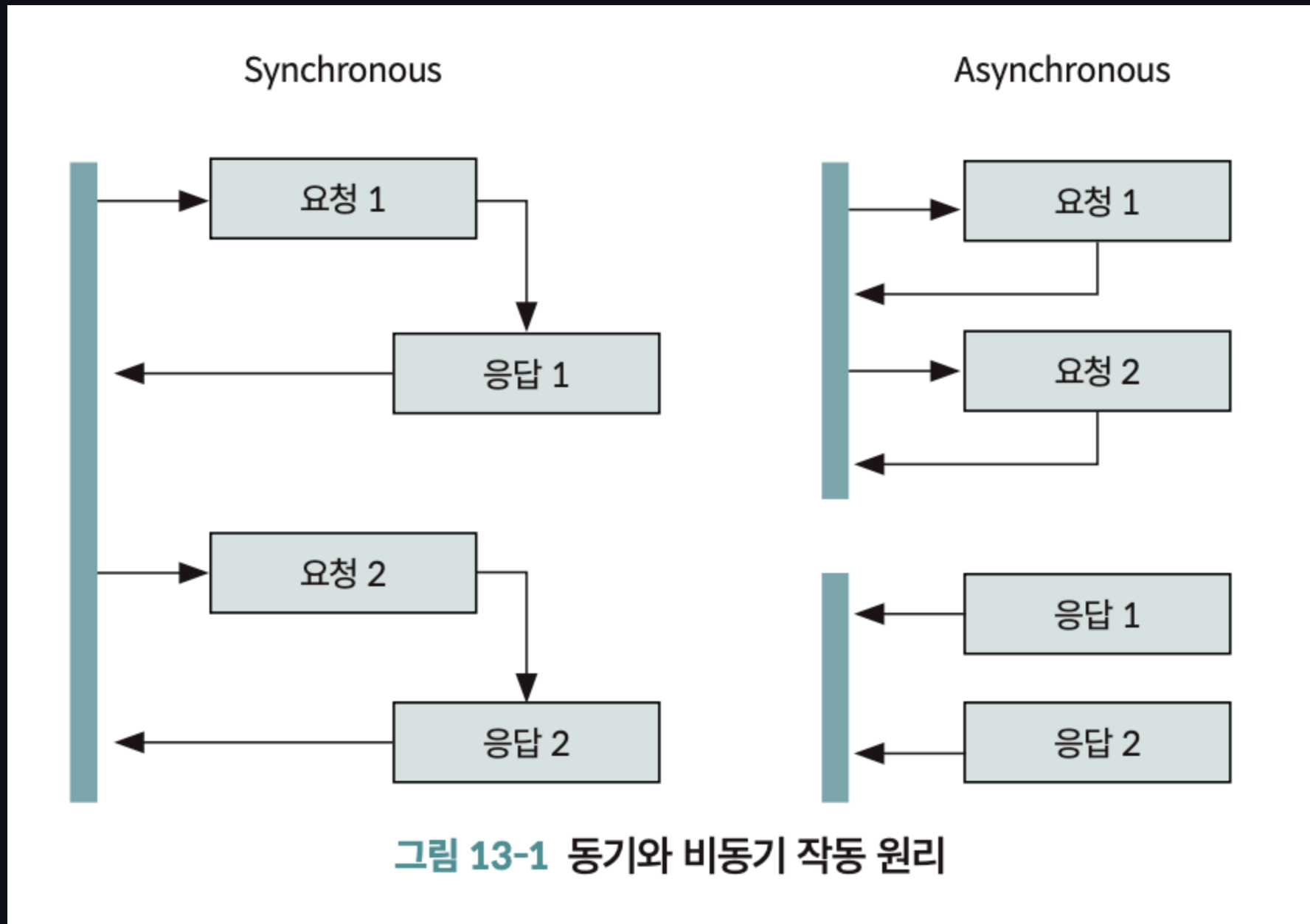
freedomzero91@gmail.com

비동기 프로그래밍

멀티스레드와 비동기의 차이점

특징	선점형 멀티태스킹	협력적 멀티태스킹
	멀티스레딩	비동기
제어 방식	운영체제가 제어	소프트웨어가 제어
장점	CPU 의존 작업 처리 속도 향상	IO 의존 작업 처리 속도 향상, 데이터 경합 방지
단점	데이터 경합 발생 가능성 높음	서드파티가 비동기 방식을 지원해야 함

비동기 작동 방식



비동기 프로그램 만들기

비동기 런타임

- 러스트는 빌트인 `async` 런타임이 존재하지 않아서 `tokio` 를 사용
- `features = ["full"]` 을 넣어야 전체 기능을 다 사용 가능

`[dependencies]`

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

비동기 함수 만들기

비동기 함수를 동기 방식으로 호출 시 에러 발생

```
async def func(): # `async` 키워드
    return 1
print(func())
```

실행 결과

```
<coroutine object async 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 사용

```
import asyncio # 내장 비동기 런타임

async def func():
    return 1

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

asyncio.run(main())
```

비동기 런타임 만들기

- `main` 을 `async` 로 변경
- `#[tokio::main]` 를 `main` 함수에 추가

```
use tokio;

async fn func() -> i32 {
    1
}

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

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

```
use tokio;

async fn func() {
    1
}

#[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.

여러 작업 실행하기

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

```
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!
```

`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.

```
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 보내기

실제로 여러 개의 IO 요청을 보낼 때 동기보다 비동기 방식이 훨씬 효율적인지 직접 확인해보겠습니다.

동기 방식

```
import time
import requests
from random import randint

MAX_POKEMON = 898 # The highest Pokemon id

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():
    start = time.time()
    for name in fetch(10):
        print(name)
    print(f"Time taken: {time.time() - start:.2f}s")

main()
```

실행 결과

```
kabutops  
frogadier  
mankey  
lucario  
mienfoo  
pancham  
voltorb  
nuzleaf  
minccino  
aurorus  
Time taken: 2.76s
```

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

```

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() {  
    let start = std::time::Instant::now();  
    for name in fetch(10) {  
        println!("{}", name);  
    }  
  
    println!("Time taken: {:?}", start.elapsed());  
}
```

실행 결과

```
grumpig  
dartrix  
celesteela  
piloswine  
tangrowth  
virizion  
glastrier  
dewpider  
hattrem  
glameow  
Time taken: 1.618283835s
```

비동기 방식

비동기 요청을 보낼 수 있는 `aiohttp` 라이브러리를 사용

```
pip install aiohttp
```

```
import asyncio
import aiohttp
from random import randint

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]
        # 여러 개의 요청을 동시에 수행
        responses = await asyncio.gather(*tasks)
        for response in responses:
            yield response["name"]
```

```
async def main():  
    async for name in fetch(10):  
        print(name)  
  
start = time.time()  
asyncio.run(main())  
print(f"Time taken: {time.time() - start:.3f}")
```

실행 결과

```
...  
Time taken: 0.4111156463623047
```

러스트 비동기

```
use rand::Rng;
use request;

const MAX_POKEMON: u32 = 898;

async fn fetch(id: u32) -> String {
    let url = format!("https://pokeapi.co/api/v2/pokemon/{}", id);
    let client = request::Client::new();
    let response = client
        .get(&url)
        .send()
        .await
        .unwrap()
        .json::<serde_json::Value>()
        .await
        .unwrap();
    response["name"].as_str().unwrap().to_string()
}
```

```
#[tokio::main]
async fn main() {
    let start = std::time::Instant::now();
    let mut tasks = tokio::task::JoinSet::new();
    for _ in 0..10 {
        let id = rand::thread_rng().gen_range(1..=MAX_POKEMON);
        tasks.spawn(fetch(id));
    }
    while let Some(res) = tasks.join_next().await {
        println!("{}", res.unwrap());
    }
    println!("Time taken: {:?}", start.elapsed());
}
```

실행 결과

```
...
Time taken: 693.418242ms
```

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

```
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!("{}", start.elapsed().unwrap().as_millis());

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

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

실행 결과

68ms
325ms