파이썬 프로그래머를 위한 러스트 입문

윤인도

freedomzero91@gmail.com

CH12. 멀티스레딩

이번 챕터에서는 러스트의 스레드가 어떻게 만들어지는지, 그리고 어떻게 여러 개의 스레드가 안전하게 메모리를 공유하는지를 배워보겠습니다.

스레드 스폰

프로세스에서 스레드를 만드는 것을 스레드를 스폰(spawn)한다고 말합니다.

싱글 스레드 스폰하기

모든 프로그램은 메인 스레드로부터 시작합니다. 메인 스레드가 `main` 함수를 실행하고, 다른 스레드들을 실행시킬 수도 있습니다. 스레드 한 개를 스폰하는 방법은 다음과 같습니다.

```python
from threading import Thread

from time import sleep


def func1():
    for i in range(0, 10):
        print(f"Hello, can you hear me?: {i}")
        sleep(0.001)


thread = Thread(target=func1)
thread.start()

for i in range(0, 5):
    print(f"Hello from the main thread: {i}")
    sleep(0.001)
```

```
Hello, can you hear me?: 0
Hello from the main thread: 0
Hello from the main thread: 1
Hello from the main thread: 2
Hello from the main thread: 3
Hello from the main thread: 4
Hello, can you hear me?: 1
Hello, can you hear me?: 2
Hello, can you hear me?: 3
Hello, can you hear me?: 4
Hello, can you hear me?: 5
Hello, can you hear me?: 6
Hello, can you hear me?: 7
Hello, can you hear me?: 8
Hello, can you hear me?: 9
```

러스트에서는 새로운 스레드를 만들 때 표준 라이브러리의 `std::thread::spawn` 함수를 사용합니다.

```
use std::thread;
use std::time::Duration;

fn main() {
    let handle = thread::spawn(|| {
        for i in 0..10 {
            println!("Hello, can you hear me?: {}", i);
            thread::sleep(Duration::from_millis(1));
        }
    });

    for i in 0..5 {
        println!("Hello from the main thread: {}", i);
        thread::sleep(Duration::from_millis(1));
    }

    handle.join().unwrap();
}
```

실행 결과

```
Hello from the main thread: 0
Hello, can you hear me?: 0
Hello from the main thread: 1
Hello, can you hear me?: 1
Hello from the main thread: 2
Hello, can you hear me?: 2
Hello, can you hear me?: 3
Hello from the main thread: 3
Hello, can you hear me?: 4
Hello from the main thread: 4
Hello, can you hear me?: 5
Hello, can you hear me?: 6
Hello, can you hear me?: 7
Hello, can you hear me?: 8
Hello, can you hear me?: 9
```

대몬(daemon) 스레드 만들기

스폰할 스레드를 대몬 스레드로 만들 수도 있습니다. 대몬 스레드는 백그라운드에서 실행되며, 메인 스레드가 종료되면 함께 종료되는 스레드입니다.

```python
from threading import Thread

from time import sleep


def func1():
    for i in range(0, 10):
        print(f"Hello, can you hear me?: {i}")
        sleep(0.001)


thread = Thread(target=func1, daemon=True)
thread.start()

for i in range(0, 5):
    print(f"Hello from the main thread: {i}")
    sleep(0.001)
```

```
Hello, can you hear me?: 0
Hello from the main thread: 0
Hello, can you hear me?: 1
Hello from the main thread: 1
Hello from the main thread: 2
Hello, can you hear me?: 2
Hello, can you hear me?: 3
Hello from the main thread: 3
Hello from the main thread: 4
Hello, can you hear me?: 4
```

`join` 을 호출하면 메인 스레드가 종료되지 않고, 생성된 스레드가 종료될 때까지 기다립니다. 만일 메인 함수에서 `join` 을 호출하지 않으면, 스폰된 스레드가 실행 중이더라도 메인 스레드가 종료되어 프로그램이 종료됩니다.

```
use std::thread;
use std::time::Duration;

fn main() {
    thread::spawn(|| {
        for i in 0..10 {
            println!("Hello, can you hear me?: {}", i);
            thread::sleep(Duration::from_millis(1));
        }
    });

    for i in 0..5 {
        println!("Hello from the main thread: {}", i);
        thread::sleep(Duration::from_millis(1));
    }
}
```

실행 결과

```
Hello from the main thread: 0
Hello, can you hear me?: 0
Hello from the main thread: 1
Hello, can you hear me?: 1
Hello from the main thread: 2
Hello, can you hear me?: 2
Hello from the main thread: 3
Hello, can you hear me?: 3
Hello from the main thread: 4
Hello, can you hear me?: 4
Hello, can you hear me?: 5
```

`join` 을 사용해 스레드 기다리기

일반적으로 스레드는 시작된 다음 `join` 함수를 사용해 스레드 작업이 끝날 때까지 기다려집니다.

```python
from threading import Thread

from time import sleep


def func1():
    for i in range(0, 10):
        print(f"Hello, can you hear me?: {i}")
        sleep(0.001)


thread = Thread(target=func1)
thread.start()
thread.join()

for i in range(0, 5):
    print(f"Hello from the main thread: {i}")
    sleep(0.001)
```

실행 결과

```
Hello, can you hear me?: 0
Hello, can you hear me?: 1
Hello, can you hear me?: 2
Hello, can you hear me?: 3
Hello, can you hear me?: 4
Hello, can you hear me?: 5
Hello, can you hear me?: 6
Hello, can you hear me?: 7
Hello, can you hear me?: 8
Hello, can you hear me?: 9
Hello from the main thread: 0
Hello from the main thread: 1
Hello from the main thread: 2
Hello from the main thread: 3
Hello from the main thread: 4
```

이번에는 `join()` 을 사용해 스폰된 스레드가 끝까지 실행되기를 기다렸다가 메인 스레드를 실행합니다.

```
use std::thread;
use std::time::Duration;

fn main() {
    let handle = thread::spawn(|| {
        for i in 0..10 {
            println!("Hello, can you hear me?: {}", i);
            thread::sleep(Duration::from_millis(1));
        }
    });
    handle.join().unwrap();

    for i in 0..5 {
        println!("Hello from the main thread: {}", i);
        thread::sleep(Duration::from_millis(1));
    }
}
```

실행 결과

```
Hello, can you hear me?: 0
Hello, can you hear me?: 1
Hello, can you hear me?: 2
Hello, can you hear me?: 3
Hello, can you hear me?: 4
Hello, can you hear me?: 5
Hello, can you hear me?: 6
Hello, can you hear me?: 7
Hello, can you hear me?: 8
Hello, can you hear me?: 9
Hello from the main thread: 0
Hello from the main thread: 1
Hello from the main thread: 2
Hello from the main thread: 3
Hello from the main thread: 4
```

스레드와 소유권

`std::thread::spawn` 에는 함수 이름을 전달할 수도 있지만, 앞에서처럼 클로저(closure)를 전달하는 경우가 더 많습니다. 클로저를 사용하면 특정 값을 스레드 안으로 이동시키는 것이 가능합니다.

```
let numbers = vec![1, 2, 3];
thread::spawn(move || {
    for n in numbers {
        println!("{n}");
    }
})
.join()
.unwrap();
```

클로저의 리턴값은 스레드로 전달됩니다. 이 값은 `join` 메소드가 호출될 때 `Result`로 감싸져서 리턴됩니다.

```
let numbers = Vec::from_iter(0..=1000);

let t = thread::spawn(move || {
    let len = numbers.len();
    let sum = numbers.into_iter().sum::<usize>();
    sum / len // 1
});

let average = t.join().unwrap(); // 2
println!("average: {average}");
```

범위 제한(scoped) 스레드

만일 어떤 스레드가 반드시 특정 범위에서만 존재하는 것이 확실할 경우, 범위 제한 스레드를 만들기 위해서 `std::thread::scope`를 사용하면 됩니다.

```
use std::thread;

fn main() {
    let numbers = vec![1, 2, 3];

    thread::scope(|s| {
        s.spawn(|| {
            println!("length: {}", numbers.len());
        });
        s.spawn(|| {
            for n in &numbers {
                println!("{n}");
            }
        });
    });
}
```

1. 먼저 스레드의 범위를 만들어주기 위해 `std::thread::scope` 함수에 클로저를 전달합니다. 해당 클로저는 즉시 실행되고, 현재 범위를 나타내는 인자 `s` 를 입력으로 받습니다.
2. 그다음 `s` 를 사용해 스레드를 생성합니다. 스레드에 전달되는 클로저는 지역 변수 `numbers` 를 사용할 수 있습니다.
3. 범위가 끝날 때, 실행 중인 스레드들이 종료될 때까지 기다립니다.

하지만 코드를 아래와 같이 바꿔서 `numbers` 에 새로운 값을 넣으려고 하면 컴파일 오류가 발생합니다.

```
use std::thread;

fn main() {
    let mut numbers = vec![1, 2, 3];
    thread::scope(|s| {
        s.spawn(|| {
            numbers.push(1);
        });
        s.spawn(|| {
            numbers.push(2); // Error!
        });
    });
}
```

GIL(Global interpreter lock)

GIL은 한 번에 하나의 스레드만 파이썬 바이트코드를 실행하도록 하기 위해 인터프리터에서 사용되는 락(lock)입니다. 스레드가 GIL을 획득하면 파이썬 바이트코드를 실행할 수 있지만 다른 모든 스레드는 GIL이 해제될 때까지 파이썬 코드를 실행할 수 없도록 차단됩니다.

GIL의 단점

- GIL은 멀티스레드 프로그램, 특히 CPU에 바인딩된 작업을 포함하는 프로그램의 성능에 치명적인 영향을 줍니다.
- 한 스레드가 CPU에 바인딩된 작업을 실행하는 경우, 다른 스레드가 I/O 또는 기타 CPU에 바인딩되지 않은 작업을 대기 중이더라도 작업이 끝날 때까지 기다려야 합니다.

GIL의 한계를 극복하기 위해 파이썬은 몇 가지 기능을 지원합니다.

- 비동기 프로그래밍
- 멀티스레딩
- 멀티프로세싱
- 제한적 GIL 해제

GIL 경합(contention) 문제

아래 코드를 실행해 보면 `count` 함수를 연속으로 두 번 호출하는 것과, 스레드를 사용하는 것과의 실제 실행 속도 차이가 그리 크지 않습니다.

```python
import time
import threading

N = 10000000

def count(n):
    for i in range(n):
        pass

start = time.time()
count(N)
count(N)
print(f"Elapsed time(sequential): {(time.time() - start) * 1000:.3f}ms")

start = time.time()
t1 = threading.Thread(target=count, args=(N,))
t2 = threading.Thread(target=count, args=(N,))

t1.start()
t2.start()

t1.join()
t2.join()

print(f"Elapsed time(threaded): {(time.time() - start) * 1000:.3f}ms")
```

실행 결과

```
Elapsed time(sequential): 0.4786410331726074
Elapsed time(threaded): 0.4163088798522949
```

메모리 공유

스레드 소유권과 레퍼런스 카운팅

지금까지 객체의 소유권을 `move` 키워드를 사용해 클로저로 넘기거나, 범위 제한 스레드를 사용하는 방법을 살펴보았습니다. 두 스레드가 데이터를 공유하는 상황에서, 두 스레드 모두가 나머지 하나보다 더 오래 존재한다는 사실이 보장되지 않는다면, 어떤 스레드도 데이터의 소유권을 가질 수 없습니다. 공유되는 어떤 데이터도 두 스레드보다 더 오래 존재해야만 합니다.

스태틱(static)

`static` 변수는 프로그램 자체가 소유권을 가지기 때문에 반드시 어떤 스레드보다도 오래 존재할 수 있습니다.

```
static X: [i32; 3] = [1, 2, 3];
thread::spawn(|| dbg!(&X));
thread::spawn(|| dbg!(&X));
```

유출(Leaking)

`Box::leak` 함수를 사용하면 `Box`의 소유권을 해제하고 절대 이 값이 삭제되지 않게 할 수 있습니다. 이때부터 `Box`는 프로그램이 종료될 때까지 존재하게 되고 어느 스레드에서도 값을 빌려 갈 수 있게 됩니다.

```
let x: &'static [i32; 3] = Box::leak(Box::new([1, 2, 3]));

thread::spawn(move || dbg!(x));
thread::spawn(move || dbg!(x));
```

레퍼런스 카운팅

*소유권을 공유*해서 해당 데이터의 소유자들을 지속적으로 관리함으로써 더 이상 해당 데이터의 소유자가 없을 때 객체를 삭제할 수 있습니다.

```
use std::rc::Rc;

let a = Rc::new([1, 2, 3]);
let b = a.clone();

assert_eq!(a.as_ptr(), b.as_ptr()); // Same allocation!
```

하지만 `Rc`를 다른 스레드로 보내려고 하면 에러가 발생합니다.

```
    error[E0277]: `Rc` cannot be sent between threads safely
        |
    8   |     thread::spawn(move || dbg!(b));
        |                   ^^^^^^^^^^^^^^^^
        |
```

Arc(Atomic refernce counting)

대신 아토믹(atomically)한 레퍼런스 카운팅을 사용하는 `std::sync::Arc`을 사용할 수 있습니다. `Rc`와 동일한 기능을 제공하지만, `Arc`는 여러 스레드에서 레퍼런스 카운터를 변경하는 것이 허용된다는 점이 다릅니다. 레퍼런스 카운터가 변경되는 작업이 아토믹하게 이루어지기 때문에, 여러 개의 스레드에서 동시에 카운터를 변경하더라도 스레드 안전성이 보장됩니다.

```rust
use std::sync::Arc;

let a = Arc::new([1, 2, 3]);
let b = a.clone();

thread::spawn(move || dbg!(a));
thread::spawn(move || dbg!(b));
```

- (1)에서 배열을 `Arc`를 사용해 메모리에 할당합니다. 이때 레퍼런스 카운터는 1이 됩니다.
- `Arc`를 클론하면 레퍼런스 카운트는 2가 되고, `a`와 `b` 모두 같은 메모리 주소를 사용합니다.
- 각 스레드마다 고유한 `Arc`를 전달받았습니다. 즉 배열이 스레드 사이에 공유되었습니다. 각 스레드에서 `Arc`가 삭제될 때마다 레퍼런스 카운터가 감소하고, 카운터가 0이 되면 배열은 메모리에서 할당 해제됩니다.

뮤텍스(mutex)

뮤텍스는 Mutual exclusion(상호 배제)의 약자로, 뮤텍스는 주어진 시간에 하나의 스레드만 데이터에 액세스할 수 있도록 허용합니다. 뮤텍스를 사용하기 위해서는 두 가지 규칙을 지켜야 합니다.

- 데이터를 사용하기 전에 반드시 잠금을 해제해야 합니다.
- 뮤텍스가 보호하는 데이터를 사용한 후에는 다른 스레드가 잠금을 획득할 수 있도록 데이터의 잠금을 해제해야 합니다.

뮤텍스의 API

뮤텍스 사용 방법을 살펴보기 위해 단일 스레드에서 뮤텍스를 사용하는 것부터 시작해 보겠습니다.

```
use std::sync::Mutex;

fn main() {
    let m = Mutex::new(5);

    {
        let mut num = m.lock().unwrap();
        *num = 6;
    }

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

실행 결과

```
m = Mutex { data: 6, poisoned: false, .. }
```

balance가 인출하고자 하는 금액보다 크다고 판단해 잔고를 인출합니다.

```
import threading
import time

balance = 100

def withdraw(amount):
    global balance
    if balance >= amount:
        time.sleep(0.01)
        balance -= amount
        print(f"Withdrawal successful. Balance: {balance}")
    else:
        print("Insufficient balance.")

def main():
    t1 = threading.Thread(target=withdraw, args=(50,))
    t2 = threading.Thread(target=withdraw, args=(75,))
    t1.start()
    t2.start()
    t1.join()
    t2.join()

if __name__ == '__main__':
    main()
```

실행 결과

```
Withdrawal successful. Balance: 50
Withdrawal successful. Balance: -25
```

스레드가 lock을 획득했을 때만 데이터에 접근 가능하도록 하면 이러한 문제를 막을 수 있습니다.

```
import time
import threading

balance = 100
lock = threading.Lock()

def withdraw(amount):
    global balance
    thread_id = threading.get_ident()
    with lock:
        print(f"Thread {thread_id}: Checking balance...")
        if balance >= amount:
            time.sleep(1)
            balance -= amount
            print(f"Thread {thread_id}: Withdrawal successful. Balance: {balance}")
        else:
            print(f"Thread {thread_id}: Insufficient balance.")

def check_lock():
    while lock.locked():
        print("Lock is locked.")
        time.sleep(0.1)

def main():
    t1 = threading.Thread(target=withdraw, args=(50,))
    t2 = threading.Thread(target=withdraw, args=(75,))
    t3 = threading.Thread(target=check_lock)
    t1.start()
    t2.start()
    t3.start()
    t1.join()
    t2.join()
    t3.join()

if __name__ == '__main__':
    main()
```

실행 결과

```
Thread 123145503645696: Checking balance...
Lock is locked.
Lock is locked.
Lock is locked.
Lock is locked.
Lock is locked.
Lock is locked.
Lock is locked.
Lock is locked.
Lock is locked.
Lock is locked.
Lock is locked.
Thread 123145503645696: Withdrawal successful. Balance: 50
Thread 123145520435200: Checking balance...
Thread 123145520435200: Insufficient balance.
```

다음 코드는 10개의 스레드가 100번씩 1씩 증가하는 값을 더하는 코드입니다.

```rust
use std::sync::Arc;
use std::thread;
use std::time::Duration;

fn withdraw(balance: &mut i32, amount: i32) {
    if *balance >= amount {
        thread::sleep(Duration::from_millis(10));
        *balance -= amount;
        println!("Withdrawal successful. Balance: {balance}");
    } else {
        println!("Insufficient balance.");
    }
}

fn main() {
    let mut balance = Arc::new(100);

    let t1 = thread::spawn(move || {
        withdraw(&mut balance, 50); // 🤯
    });

    let t2 = thread::spawn(move || {
        withdraw(&mut balance, 75);
    });

    t1.join().unwrap();
    t2.join().unwrap();
}
```

뮤텍스를 사용!

```rust
use std::sync::{Arc, Mutex};
use std::thread;
use std::time::Duration;

fn withdraw(balance: Arc<Mutex<i32>>, amount: i32) {
    let mut balance = balance.lock().unwrap();
    println!("{:?}: Checking balance.", thread::current().id());
    if *balance >= amount {
        thread::sleep(Duration::from_millis(100));
        *balance -= amount;
        println!("Withdrawal successful. Balance: {balance}");
    } else {
        println!("Insufficient balance.");
    }
}

fn check_lock(balance: Arc<Mutex<i32>>) {
    while let Err(_) = balance.try_lock() {
        println!("Lock is locked.");
        thread::sleep(Duration::from_millis(10));
    }
}
```

```rust
fn main() {
    let balance = Arc::new(Mutex::new(100));

    let balance1 = Arc::clone(&balance);
    let t1 = thread::spawn(move || {
        withdraw(Arc::clone(&balance1), 50);
    });
    let balance2 = Arc::clone(&balance);
    let t2 = thread::spawn(move || {
        withdraw(Arc::clone(&balance2), 75);
    });
    let balance3 = Arc::clone(&balance);
    let t3 = thread::spawn(move || {
        check_lock(Arc::clone(&balance3));
    });

    t1.join().unwrap();
    t2.join().unwrap();
    t3.join().unwrap();
}
```

실행 결과

```
ThreadId(2): Checking balance.
Lock is locked.
Lock is locked.
Lock is locked.
Lock is locked.
Lock is locked.
Lock is locked.
Lock is locked.
Lock is locked.
Lock is locked.
Lock is locked.
Lock is locked.
Withdrawal successful. Balance: 50
ThreadId(3): Checking balance.
Insufficient balance.
```

메시지 전달

스레드 간에 데이터를 공유하는 방법 중 하나로 널리 쓰이는 것 중 하나가 바로 MPSC입니다. 다중 생성자-단일 소비자(Multiple {roducer Single Consumer)란 뜻으로, 여러 개의 스레드에서 하나의 스레드로 데이터를 보내는 방식입니다.

파이썬에서는 공식적으로 MPSC를 만드는 방법이 없기 때문에, 스레드 안정성이 보장되는 큐 자료형인 `Queue` 를 사용해 이를 구현해 보겠습니다.

```
import threading
import time
from queue import Queue

channel = Queue(maxsize=3)


def producer():
    for msg in ["hello", "from", "the", "other", "side"]:
        print(f"Producing {msg}...")
        channel.put(msg)


def consumer():
    while not channel.empty():
        item = channel.get()
        print(f"Consuming {item}...")
        channel.task_done()
        time.sleep(0.01)


producer_thread = threading.Thread(target=producer)
producer_thread.start()

consumer_thread = threading.Thread(target=consumer)
consumer_thread.start()

producer_thread.join()
consumer_thread.join()
```

채널은 `Sender`와 `Receiver`로 구성됩니다.

```
use std::sync::mpsc;
use std::thread;
use std::time::Duration;

fn main() {
    let (tx, rx) = mpsc::channel();

    thread::spawn(move || {
        for msg in vec!["hello", "from", "the", "other", "side"] {
            let val = String::from(msg);
            println!("Producing {}...", val);
            tx.send(val).unwrap();
            thread::sleep(Duration::from_millis(10));
        }
    });

    for re in rx {
        println!("Consuming {}...", re);
    }
}
```

`try_recv` 를 사용하면 다음과 같이 할 수 있습니다.

```rust
use std::sync::mpsc;
use std::thread;
use std::time::Duration;

fn main() {
    let (tx, rx) = mpsc::channel();

    thread::spawn(move || {
        let val = String::from("hello");
        thread::sleep(Duration::from_millis(1000));
        tx.send(val).unwrap();
    });

    loop {
        println!("Waiting for the signal...");
        if let Ok(received) = rx.try_recv() {
            println!("Message: {}", received);
            break;
        }

        thread::sleep(Duration::from_millis(300));
    }
}
```

rx가 어떻게 앞의 메시지를 다 기다릴 수 있는지? Receiver, 즉 `rx` 는 송신자 `tx` 가 메시지를 전송할 때까지 기다리려고 시도하며, 해당 채널이 끊어지면 오류를 반환합니다.

```rust
use std::sync::mpsc;
use std::thread;
use std::time::Duration;

fn main() {
    let (tx, rx) = mpsc::channel();

    let tx1 = tx.clone();
    thread::spawn(move || {
        for msg in vec!["hello", "from", "the", "other", "side"] {
            let val = String::from(msg);
            thread::sleep(Duration::from_millis(100));
            tx1.send(val).unwrap();
        }
    });

    thread::spawn(move || {
        let val = String::from("bye");
        thread::sleep(Duration::from_millis(1000));
        tx.send(val).unwrap();
    });

    for re in rx {
        println!("{}", re);
    }
}
```