

Rust 비동기 프로그래밍

남정현 - HYPERITHM

junghyun@hyperithm.com

2022.08.01

무엇을 다루나요?

- 비동기 프로그래밍이 왜 필요한가?
- Asynchronous Rust
 - 간단한 규칙들과 주로 사용되는 crates
 - 생각 없이 `async/await` 붙이기
 - 다른 언어와의 차이점

누구를 위한 자료인가요?

- 최소 사양
 - 프로그래밍에 관심이 있는 사람
 - Rust에 관심이 있는 사람
 - Rust의 비동기 프로그래밍에 관심이 있는 사람
 - 복잡한 내용이 나와도 무너지지 않을 강인한 정신력
- 권장 사양
 - 기본적인~중등도의 Rust 이해도
 - 다음의 개념들에 대한 기본 지식: 기본 Rust 문법, Trait, Associated Type, Generics, Lifetime
 - 컴퓨터 구조에 대한 간단한 이해(CPU, 메모리, I/O, ...)
 - Rust에서 `async/await`을 사용해 본 사람

(아마도) 간단한 비유

- 저는 콜센터 상담원이 되었습니다
 - 컨베이어 벨트에서 전화기가 튀어나옵니다
 - 수화기를 들면 상대가 전화번호와 전달할 내용을 말하고 끊습니다
 - 전달할 상대의 전화번호를 누릅니다
 - 신호가 가고 상대가 받을 때까지 기다립니다
 - 상대가 받으면 내용을 전달하고 끊습니다

(아마도) 간단한 비유

- 쉽게쉽게 하려면: 다음을 순서대로 반복하면 됩니다
 - 컨베이어 벨트에서 전화기가 튀어나옵니다
 - 수화기를 들면 상대가 전화번호와 전달할 내용을 말하고 끊습니다
 - 전달할 상대의 전화번호를 누릅니다
 - 신호가 가고 상대가 받을 때까지 기다립니다
 - 상대가 받으면 내용을 전달하고 끊습니다
- 한 사람이 다 해야 하니까 한 바퀴 돌때까지 다른 일을 못 합니다

(아마도) 간단한 비유

- 전화를 거는 사람이 많아져서 한 사람으로는 부족합니다
 - 직원을 4명으로 늘립니다
 - 서서 일하면 안 되니까 의자도 4대로 늘립니다
 - 규칙: 의자에 앉아야지만 말할 수 있습니다
 - 컨베이어 벨트에서 수화기가 나오면 놀고 있는 직원이 알아서 가져갑니다
 - 일단 일 처리량이 4배로 늘어서 좋습니다

(아마도) 간단한 비유

- 전화를 거는 사람이 더 많아지면 어떻게 하죠?
 - 직원을 4→8명으로 늘립니다.
 - 의자도 4→8개로 늘리려 했는데 의자가 비싸서 더 늘릴 수가 없네요...
- 꼼수: 전화를 '쓰고 있을 때만' 앉아있읍시다
 - 다음 상황에서는 서서 구경만 합니다:
 - 수화기가 컨베이어 벨트에서 나올 때까지 기다릴 때
 - 전화를 걸었는데 신호가 가지만 하고 상대가 받지 않을 때
- 일단 8명에서 의자 4개로 일할 수 있게 되었습니다



[공식판매처] [허먼밀러] New

₩2,640,000

sivillage.com

Free shipping

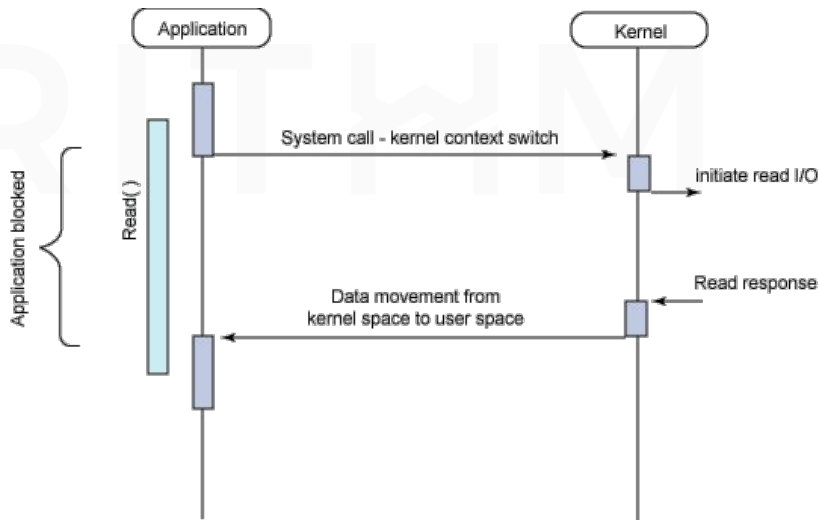
(아마도) 간단한 비유

- 전화를 거는 사람이 더 많아져서 직원을 256명으로 늘렸습니다
 - 직원들: 앉았다 일어났다가 너무 힘듭니다
 - 직원들: 의자 경쟁이 너무 치열해서 합의하기가 힘들어요
- 더 똑똑하게 전화를 걸고 받을 수 있을까요?

HYPERITM

비동기 프로그래밍이 필요한 시점

- C10K Problem: 10,000개의 클라이언트와 동시에 소통할 수 있는가?
- 쉽지 않다
 - 만약의 근원: Synchronous blocking I/O
 - I/O를 하는 동안에는 프로세스가 '동작그만'
⇒ CPU 자원이 놀게 됨
⇒ 아무리 서버 성능을 올려도 수용량 한계
 - I/O-bound

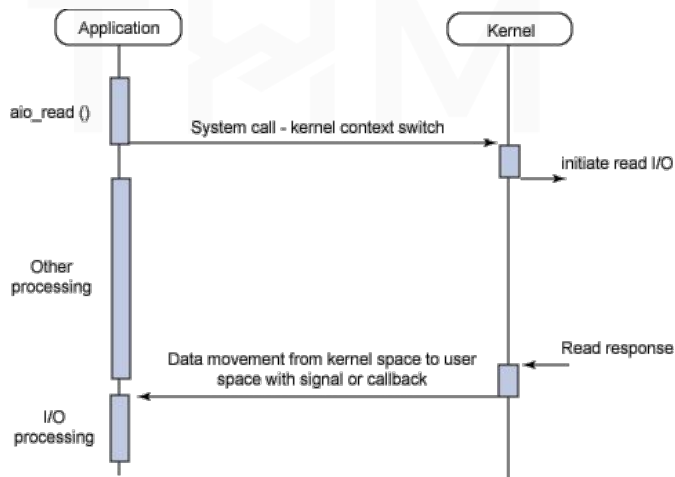


비동기 프로그래밍이 왜 필요한가?

- I/O 병목 줄이기: ‘동작그만’ 하지 말고 다른 일
 - 다른 일: CPU 연산, 다른 I/O 동작 시작하기, ...
- 가장 쉬운 법: process(or thread)를 마구마구 만들자
 - I/O 전담 thread → 연산과 I/O를 분리했지만 병목은 여전함
 - I/O thread pool → I/O 부담이 커지면 확장성에 한계
 - 1 client 1 thread → 클라이언트가 많아지면 메모리 부담
- 결정적으로 thread 간의 전환은 비싸다!!
- 아까의 비유에서...
 - (컨베이어 벨트에서 튀어나온) 수화기 ⇒ 클라이언트(에 연결된 네트워크 소켓)
 - 직원 ⇒ thread
 - 의자 ⇒ CPU 코어

비동기 프로그래밍이 왜 필요한가?

- 결국 process/thread를 사용해서는 확장에 한계가 존재
- I/O 자체가 비동기여야 함 → Asynchronous nonblocking I/O
- 그런데 이것을 어떻게 쓰지…?



Async I/O 표현하기 - (아마도) 간단한 비유

- 똑똑하게 하려면: 전화를 거는 동안 쉬지 않습니다.
 - A가 전화를 겁니다
 - B의 전화번호와 B-내용을 말해줍니다
 - B의 전화번호를 누릅니다
 - 신호가 갑니다. 동시에 새롭게 오는 전화도 받아봅니다
 - C가 전화를 걸었습니다. D의 전화번호와 D-내용을 받습니다
 - B가 D보다 전화를 먼저 받아서 일단 B-내용을 전달했습니다
 - D에게 계속 걸고 있는데 E가 전화를 걸었습니다
 - 이제 E에게 전화를 걸고 있는데 D가 전화를 받았습니다
- 정신 나갈 것 같아요!!!
 - 복잡한 async I/O 로직을 짜는 건 어렵습니다
 - Rust를 사용하면 async한 로직을 더 직관적으로 표현할 수 있습니다

잠깐만요

- Async와 Nonblocking은 (미묘하게) 다릅니다.
 - Async: I/O를 하는 동안 다른 일을 할 수 있다
 - Nonblocking: I/O가 준비될때까지 **기다리지 않는다**
 - 하지만 사용하는 맥락마다 뜻이 다르고 섞어서 쓰는 곳도 많으니 대충 넘어갑시다.
- 동시성(concurrency)과 병렬화(parallelization)도 다릅니다
 - 병렬화: 여러 작업을 '한번에' 진행 (멀티스레딩), 'simultaneous execution'
 - CPU 코어 개수만큼 병렬로 진행한다거나...
 - 동시성: 여러 작업을 '같은 시간 동안' 진행 (시분할, 병렬화, 코루틴, ...), 'simultaneous waiting'
 - 스레드 한 개로도 할 수 있습니다

Asynchronous Rust 써보기

- `cargo add tokio --features full`

```
fn main() {  
    let data = std::fs::read("wow.txt");  
    let handle = thread::spawn(move || { dbg!(data); })  
    handle.join().unwrap();  
}
```

```
#[tokio::main]  
async fn main() {  
    let data = tokio::fs::read("wow.txt").await;  
    let handle = tokio::spawn(async move { async_print(data).await; });  
    handle.await.unwrap();  
}
```

Asynchronous Rust 써보기: 간단한 규칙들

- `async fn`은 `async fn` 안에서만 호출 가능합니다.
- `async fn foo()`을 호출할 때는 `await`을 해 줍니다.
 - `foo()`의 결과를 받고 싶으면 `foo().await;`
 - 안 하면 경고해주니까 까먹어도 괜찮습니다.
- `async fn` 안에서는 `std`의 I/O 함수를 호출하지 않습니다.
 - 대신 `tokio`의 것을 씁니다. (`tokio::fs`, `tokio::net`, ...)
 - 만약에 `std`의 것을 써야 한다면 `tokio::task::spawn_blocking`으로 감쌉니다.
 - `async fn`이 호출하는 모든 함수에 대해서도 동일한 규칙이 적용됩니다.
- 그냥 `fn`에서 `async fn`을 호출하는 것은 가능하긴 합니다.
 - `Runtime::new().block_on()`
 - `Runtime::spawn()`

다른 언어와 비교

- Go: Goroutines
 - Go 런타임이 관리하는 초경량 스레드
 - M:N threading
- Javascript: Promises
 - N:1 threading
- Rust: Futures
 - Executor 구현 마음대로 (3rd-party)
 - `tokio: rt(N:1), rt-multi-thread(M:N)`
 - 중요한 점: Future는 `.await`되지 않는 한 스스로 실행되지 않음
 - 참고로 `tokio::task::spawn`은 스스로 실행됩니다
 - Future는 cooperative하게 처리됩니다. (\leftrightarrow preemptive)
 - `.await` 지점 외에서는 흐름을 뺏아올 수 없습니다.
 - blocking I/O를 하면 안 되는 이유이기도 함

끝

- 질문 있으신 분?

HYPERITHM

심연 속으로

- 여기까지만 다루고 끝내면 재미가 없죠?

HYPERITHM




(진짜) 무엇을 다루나요?

- 비동기 프로그래밍이 왜 필요한가?
- Asynchronous Rust (easy)
 - 간단한 규칙들과 주로 사용되는 crates
 - 생각 없이 `async/await` 붙이기
 - 다른 언어와의 차이점
- Asynchronous Rust (hard)
 - `The Future Trait`
 - `Pin` and `pin-project`
- Caveats of Asynchronous Rust
 - Async trait, recursion, drop ...
- Asynchronous Rust Ecosystem
 - `tokio/tower/reqwest/hyper/tracing`

Async Rust에는 두 개의 세계가 있습니다...

- `async fn, async { }, ...`
 - Rust 컴파일러가 제공하는 ‘문법적 설탕’ (개발자의 편의를 위해 제공되는 특수 문법)
 - 쉽고 간편하다
 - 근데 재미가 떨어짐(???)
- `std::future::Future`
 - Asynchronous Rust의 중추
 - 무설탕 문법의 세계로 들어가봅시다

Outline

	
<code>async { ... } / async fn</code>	
<code>.await</code>	???
???	???

The Future Trait

참고:
trait ~= interface
self = this

```
trait Future {  
    type Output;  
  
    fn poll(self: Pin<&mut Self>, cx: &mut Context<'_>)  
        -> Poll<Self::Output>;  
}
```

The Future Trait

A future represents an asynchronous computation obtained by use of `async`.

A future is a value that might not have finished computing yet. This kind of “asynchronous value” makes it possible for a thread to continue doing useful work while it waits for the value to become available.

The Future Trait

- 정의를 조금 단순화해봅시다

```
trait Future {  
    type Output;  
  
    fn poll(&mut self) -> Poll<Self::Output>;  
}
```

```
pub enum Poll<T> {  
    Ready(T),  
    Pending,  
}
```


The Future Trait

- `type Output: Future`(를 구현하는 어떤 친구)가 반환할 값의 타입
- `fn poll(): Future`(를 구현하는 어떤 친구)야 '준비되었니?'
 - `Poll::Ready(Output)` : 응 '준비되었어' (`Output`을 돌려주며)
 - `Poll::Pending`: 아니
 - 주의: 이미 `Ready`를 반환했으면 다시 `poll()`했을 때는 어떻게 될지 모릅니다
 - Undefined Behavior 빼고 모두 가능(`no-op`, `panic!`, `Pending`, `Ready`)
- `poll()`은 준비 여부에 관계없이 '즉시' 반환합니다
 - Synchronous Rust는 준비될때까지 계속 대기합니다.
 - 그러면 호출자는 준비가 될 때까지(더 정확히: 다시 `poll`할 때까지) 다른 일을 할 수 있습니다.

Poll-based Futures are Inefficient

- `poll()`: Future야 준비됐니?
 - 아니
- `poll()`: Future야 준비됐니?
 - 아니
- `poll()`: Future야 준비됐니?
 - 아니
- ...
- `poll()`: Future야 준비됐니?
 - (드디어) 응
 - 너무나도 비효율적입니다

Waker

- 효율적으로 `poll()` 하려면: 필요할 때만 알려주면 됩니다
 - 타이머: 목표한 시간이 됨, 네트워크: 해당 소켓에 이벤트 발생
- 어떻게 알려주나요?
 - `let waker = cx.waker().clone();`
 - `waker.wake();`

```
trait Future {  
    type Output;  
  
    fn poll(&mut self, cx: &mut Context<'_>)  
        -> Poll<Self::Output>;  
}
```

이 Future는 누가 실행해주나요?

- 이걸 Executor라 합니다.
 - `Poll::Ready`를 받으면 결과가 나왔으니 반환해주고
 - `Poll::Pending`을 받으면 wake 될때까지 기다리다가 다시 `poll()` 하면 됩니다.
- 생각보다 간단합니다?

HYPERITM

Wrap-up: `async`은 ‘문법적 설탕’

- `trait Future`은:
 - ‘즉시’ 값을 반환하지 않는 값을 추상화하는 트레이트
 - 값을 반환할 수 있는지는 `poll()` 로 확인
 - 반환되는 값의 타입은 associated type `Future::Output`
- `async fn(Args) -> Return` 은:
 - `fn(Args) -> impl Future<Output=Return>` 의 설탕
- `async { ... }` 은:
 - `impl Future<Output=T>` 의 설탕
- recap: `impl Trait`이란?
 - 어떤 값이 Trait을 구현한다는 사실만 남겨둔 ‘가면 쓴(opaque) 타입’
 - 컴파일 타임에 타입이 추론되지만 정확히 어떤 타입인지 이름을 알 수 없음
 - 이게 없으면: `xxx.rs`의 6번째 줄부터 16번째 줄까지의 `async { }` 에 타입을 뭘로 줘야 할까요?

여기까지만 보면 참 쉬운데...

- 처음 보는 놈이 있습니다

```
trait Future {  
    type Output;  
  
    fn poll(self: Pin<&mut Self>, cx: &mut Context<'_>)  
        -> Poll<Self::Output>;  
}
```

- 넌 누구냐???



- 넌 누구냐?

Struct std::pin::Pin 

1.33.0 · [source](#) · [-]

```
#[repr(transparent)]  
pub struct Pin<P> { /* private fields */ }
```

[-] A pinned pointer.

This is a wrapper around a kind of pointer which makes that pointer “pin” its value in place, preventing the value referenced by that pointer from being moved unless it implements `Unpin`.

See the `pin module` documentation for an explanation of pinning.

(2)

- 나루호도...
- 일단 Future가 본질적으로 무엇인지 알아야 합니다

Module `std::pin`

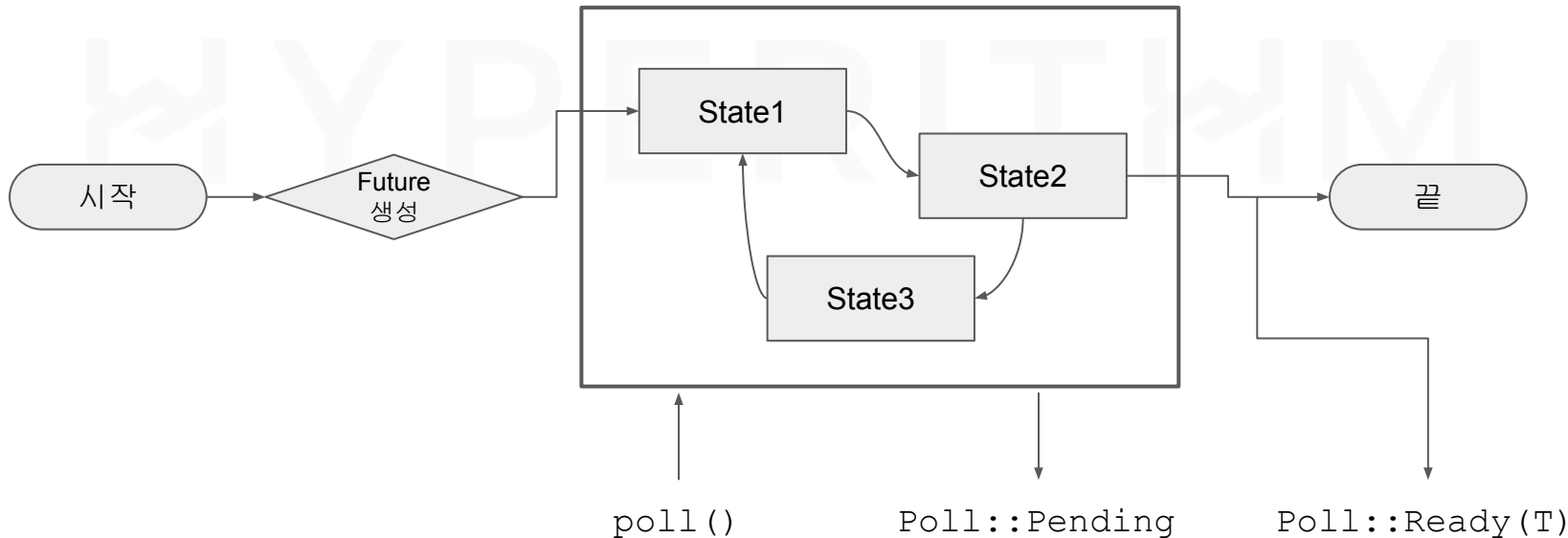
1.33.0 · [source](#) · [-]

[-] Types that pin data to its location in memory.

It is sometimes useful to have objects that are guaranteed not to move, in the sense that their placement in memory does not change, and can thus be relied upon. A prime example of such a scenario would be building self-referential structs, as moving an object with pointers to itself will invalidate them, which could cause undefined behavior.

How Futures are Constructed

- Future는 결론적으로 FSM(Finite State Machine)입니다
- poll(): State 간의 전환이 정의된 로직



How Futures are Constructed (A Simplified Example)

- 이런 건 직관적으로 설계하기 힘드니까...

- 똑똑하게 하려면: 전화를 거는 동안 쉬지 않습니다.
 - A가 전화를 겁니다
 - B의 전화번호와 B-내용을 말해줍니다
 - 전달할 상대의 전화번호를 누릅니다
 - 신호가 갑니다. 동시에 오는 전화도 받아봅니다
 - C가 전화를 걸었습니다. D의 전화번호와 D-내용을 받습니다
 - B가 D보다 전화를 먼저 받아서 일단 B-내용을 전달했습니다
 - D에게 계속 걸고 있는데 E가 전화를 걸었습니다
 - 이제 E에게 전화를 걸고 있는데 D가 전화를 받았습니다

How Futures are Constructed (A Simplified Example)

- `enum/struct`: '멈춤' 과 '멈춤' 사이를 구분할 상태들을 정의

```
enum CallerState {  
    ListeningForTarget,  
    CallingTarget,  
    SendingMessageToTarget,  
}
```

How Futures are Constructed (A Simplified Example)

- `poll()`: 상태 간의 전환을 정의

```
impl Future for CallerState {  
    fn poll(&mut Self) -> Poll {  
        match *self {  
            ListeningForTarget => {  
                *self = CallingTarget;  
                return Pending;  
            }  
            CallingTarget => {  
                *self = SendingMessageToTarget;  
                return Pending;  
            }  
            SendingMessageToTarget => {  
                return Ready(());  
            }  
        }  
    }  
}
```

```
enum CallerState {  
    ListeningForTarget,  
    CallingTarget,  
    SendingMessageToTarget,  
}
```

How Futures are Constructed (A Simplified Example)

- 좀 더 현실적으로: 직원들이 전화기를 사용해야 함
 - 이걸 어떻게 표현하지?
 - 생각해 보니 자기참조가 발생


```
enum CallerState<'a> {  
    ListeningForTarget(&'a mut Telephone),  
    CallingTarget(&'a mut Telephone),  
    SendingMessageToTarget,  
}
```

```
struct CallCenterState {  
    callers: Vec<CallerState<'what>>,  
    telephones: Vec<Telephone>,  
}  
  
impl Future for CallCenterState {  
    // CallerState를 하나씩 .poll() 해주기  
}
```

지역변수 참조 = 자기참조

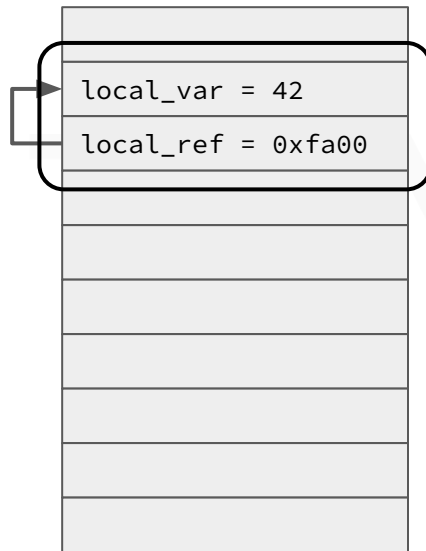
- `async { ... }` 설탕을 써도 자기참조를 피할 수 없음

```
async {  
    let telephone = Telephone;  
    let caller = Caller(&mut telephone);  
    caller.await;  
}
```

```
struct State {  
     var_telephone: Telephone,  
    ref_caller: &'what Telephone,  
}
```

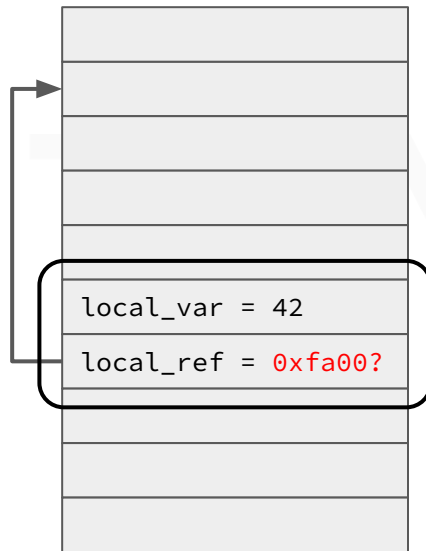
Self-referential structs Cannot be Moved

```
struct MovingFuture {  
    local_var: u16,  
    local_ref: *mut u16  
}
```



Self-referential structs Cannot be Moved

```
struct MovingFuture {  
    local_var: u16,  
    local_ref: *mut u16  
}
```



Wrap-up: Pin이 필요한 이유

- Future는 ‘중간중간 멈추는’ 유한 상태 기계인데
 - Future의 상태에 자신을 가리키는 참조가 섞여 있을 수 있고
 - 자기 참조가 있으면 ‘움직이는’ 순간 참조가 무효화되니까
 - 자기 참조를 가진 Future는 절대 움직일 수 없게 해야 하는데
 - 이걸 Rust 문법 상으로 보장할 수가 없다! (대입/호출/반환 등등이 죄다 move)
- Q: 자기 참조가 없으면 움직여도 되나요?
 - A: 네.
 - 그래서 Unpin trait이 있습니다. (뒤에 나옵니다)

어떻게 값을 고정하나요?

참고:

Deref trait은 'dereferencing' 연산을 오버라이드할 수 있는 trait입니다.

ex: $\&T \rightarrow T$

- `struct Pin<P>`
 - P는 T를 가리키는 포인터라고 정의합니다. (`&T`, `&mut T`, `Box<T>` 등등)
 - P가 포인터가 아니면요?: 못 씁니다. (P: Deref 제한이 있음)
 - Pin이 T를 고정하는 법: Deref/DerefMut 구현을 없애버림
- `Pin::new(ptr)`
 - T: Unpin이어야 합니다. (=자기참조가 없음)
 - 이 경우에는 Pin을 마음대로 풀 수도 있습니다(`Pin::into_inner`).
- `Pin::new_unchecked(ptr)` 🙅
 - unsafe입니다. (아래의 조건을 문법적으로 보장할 수 없으므로)
 - !Unpin을 한번 Pin하면 절대 그 값을 움직이면 안 됩니다. `Pin<P>`가 사라져도요.
 - `let ptr = &value;`
 - `{ let pinned = Pin::new_unchecked(ptr); }`
 - 이 시점에서 pinned가 없으니 value를 움직일 수 있습니다 ← 움직이는 순간 규칙 위반

어떻게 값을 고정하나요?



- `Box::pin(T)` 🍌
 - 제일 간단한 방법
 - `Box`를 만들어서(=할당을 해서) 거기에 `T`를 넣고 `Pin<Box<T>>`로 감쌉니다
 - 아주 살짝 틀린 용어: 'pinning on the heap'
 - 장점: `Pin<P>`가 스코프 밖으로 움직일 수 있습니다
 - 단점: 할당이 필요합니다(공짜가 아님)
- `tokio::pin!`, `futures::pin_mut!` 🍌
 - `Pin::new_unchecked`의 안전한 매크로 버전
 - `Pin::new_unchecked::pinned`를 만들고도 `value`에 접근 가능한 게 문제
 - → 그러면 이름을 똑같이 해서 shadow하면 되겠네?
 - 장점: 추가 비용이 없음
 - 단점: 스코프 안에서만 고정&사용 가능('~ on the stack'), 서드파티 의존성([Nightly](#)에는 있습니다)

주의

- *Premature optimization is the root of all evil* — Donald Knuth
- 일반적으로는 그냥 `Box::pin(async { ... })` 해도 충분히 빠릅니다.

HYPERITHM

Outline

	
<code>async { ... } / async fn</code>	<code>std::future::Future</code>
<code>.await</code>	<code>???</code>
<code>???</code>	<code>???</code>

어떻게 필드를 고정하나요?

- 두 Future를 받아서 빨리 끝나는 값을 반환하는 Future
 - 이런 친구를 ‘combinator’라 합니다

```
struct Race<Fut1, Fut2> {  
    fut1: Fut1,  
    fut2: Fut2,  
}
```

```
impl<Fut1, Fut2> Future for Race  
where  
    Fut1: Future,  
    Fut2: Future {  
  
}
```

어떻게 필드를 고정하나요?

- Pinned struct에서 필드를 어떻게 가져오지?
- 가져온 필드가 pin된 상태이긴 한가?
 - = *structured pinning*

```
fn poll(self: Pin<&mut Self>, cx: Context) -> Poll {  
    if let Ready(x) = self.fut1.poll(cx) {  
        return Ready(x);  
    }  
    if let Ready(x) = self.fut2.poll(cx) {  
        return Ready(x);  
    }  
    return Pending;  
}
```

Structured Pinning

- struct/enum을 정의할 때 결정하기 나름입니다
- No: struct가 고정되어도 이 필드는 움직일 수 있다
 - 제한이 없으니 추가로 생각할 게 없고 편하긴 함
 - 실용적이지 않을 때가 있음(`self.fut1`)
- Yes: struct가 고정되었다면 이 필드도 움직일 수 없다
 - 구현이 까다로움
 - `pin-project`를 쓰면 편합니다

pin-project

```
use pin_project::pin_project;
```

```
#[pin_project]
struct Race<Fut1, Fut2> {
    #[pin]
    fut1: Fut1,
    #[pin]
    fut2: Fut2,
}
```

pin-project

- `this.fut1`과 `this.fut2`는 각각 `Pin<&mut Fut1>`, `Pin<&mut Fut2>`
- `project(): #[pin_project]` 매크로가 생성해 준 메서드

```
fn poll(self: Pin<&mut Self>, cx: Context) -> Poll {  
    let this = self.project();  
    if let Ready(x) = this.fut1.poll(cx) {  
        return Ready(x);  
    }  
    if let Ready(x) = this.fut2.poll(cx) {  
        return Ready(x);  
    }  
    return Pending;  
}
```



ready! ()

```
ready!(fut) ⇒  
if let Ready(val) = fut {  
    val  
} else {  
    return Pending;  
}
```

- Poll::Pending을 ‘밖으로 전파’시키는 역할: .await 과 비슷함

```
fn poll(self: Pin<&mut Self>, cx: Context) -> Poll {  
    let this = self.project();  
    if let Ready(x) = this.fut1.poll(cx) {  
        return Ready(x);  
    }  
    return ready!(this.fut2.poll(cx));  
}
```

Outline

	
<code>async { ... } / async fn</code>	<code>std::future::Future</code>
<code>.await</code>	<code>ready!() + pin-project</code>
???	???

그래서 굳이 을 빼는 이유가 뭔가요?

- 0.1%의 성능 손해도 용납할 수 없는 경우
 - `Box::pin` (이건 엄밀히 말하면 설탕은 아니지만...)
 - 라이브러리 작성할 때는 최대한 최적화해두면 최소 고통 다수 행복(???) 을 유도할 수 있음
- 동시에 여러 개의 Future를 실행해야 하는 경우
 - `futures-util` 등에 `join` 등의 combinator를 쓸 수는 있음
 - 사용이 불가능한 옛지 케이스 등이 있을 수 있음
- 문법 설탕이 지원해주지 않는 케이스😱를 손으로 작성해야 할 때
 - 위에서 말한 동시에 Future 실행하기
 - `Asynchronous Trait`



Asynchronous Traits

- `trait std::io::Read`
 - File도 읽을 수 있고, TcpStream도 읽을 수 있고, ...

```
pub trait Read {  
    fn read(&mut self, buf: &mut [u8])  
    -> Result<usize>;  
}
```

- `tokio::io::AsyncRead`
 - File도 비동기로 읽을 수 있고, TcpStream도 비동기로 읽을 수 있고, ...

```
pub trait AsyncRead {  
    async fn read(  
        &mut self, buf: &mut [u8]  
    ) -> Result<usize>;  
}
```

진짜 AsyncRead

- 그냥 `async`으로 선언하면 안 되나요?

```
pub trait AsyncRead {  
    fn poll_read(  
        self: Pin<&mut Self>,  
        cx: &mut Context<'_>,  
        buf: &mut ReadBuf<'_>  
    ) -> Poll<Result<()>>;  
}
```


안 됩니다

- Recap: `async fn`은 `fn -> impl Future`의 설탕
- `fn read(&'a mut self) -> impl Future<Output=Result> + 'a` 로 `desugar`
- Trait method에서 `impl Trait`은 associated type이 됨
 - `trait AsyncRead { type Read<'a>: Future<'a>; fn read(&mut self) -> Self::Read<'_>; }`
 - AKA *type-alias-impl-trait* <https://github.com/rust-lang/rust/issues/63063> → unstable
 - 근데 associated type에 타입 파라미터가? → Generic Associated Type
 - GAT는 아직 unstable(=nightly only)
- 이외에도 복잡한 문제들이 한 트럭 섞여 있음

대신에 `async-trait` 을 쓰시다

- Asynchronous trait의 향기를 느끼게 해주는 마법의 crate

```
#[async_trait]
trait Read{
    async fn read(&mut self) -> Result;
}
```

- 대신에 공짜가 아닙니다: 호출할 때마다 `Box::pin + dyn`으로 감쌘
 - 그리고 매크로가 코드를 생성해주기 때문에 documentation 등이 아래처럼 변합니다
 - 실제로 생성되는 정의:

```
trait Read{
    fn read(&'a mut self)
        -> Pin<Box<dyn Future<Output=Result>
            + 'a>>;
}
```

할당을 용납할 수 없는 경우

- 설당을 빼면 됩니다
- `poll_{name}`

```
pub trait AsyncRead {  
    fn poll_read(  
        self: Pin<&mut Self>,  
        cx: &mut Context<'_>,  
        buf: &mut ReadBuf<'_>  
    ) -> Poll<Result<()>>;  
}
```

할당을 용납할 수 없는 경우



- 그 다음 `poll_{name}` 을 감싸는 convenience method를 만듭니다

```
pub trait AsyncReadExt {  
    fn read<'a>(  
        &'a mut self,  
        buf: &'a mut [u8]  
    ) -> Read<'a, Self>  
    where  
        Self: Unpin  
    {  
        Read {  
            reader,  
            buf,  
        }  
    }  
}
```

```
pub struct Read<'a, R: ?Sized>  
{  
    reader: &'a mut R,  
    buf: &'a mut [u8],  
}  
  
impl<'a, R> Future for Read<'a, R>  
    where R: Unpin
```

```
// Example  
tokio::pin!(file); // if file is !Unpin  
file.read().await?;
```

Wrap-up: From Sugared to De-sugared

	
<code>async { ... } / async fn</code>	<code>std::future::Future</code>
<code>.await</code>	<code>ready!() + pin-project</code>
<code>#[async-trait]</code>	<code>fn poll_{name} + impl Future for {Name}</code>

Asynchronous Rust Ecosystem

- Runtime
 - Future들을 실행하는 Executor 및 I/O, timing 등의 API를 제공
 - Tokio, async-std, smol 등
 - I/O API 등은 서로 호환이 안 됩니다: `tokio::io`는 `tokio` 런타임에서만 사용 가능
 - 다른 런타임의 API와 붙으려면 compatibility layer를 붙이면 됨([example](#): `tokio` ↔ `futures-rs`)
- 생태계 경쟁에서 사실상 Tokio가 판정승
 - `reqwest`/`axum`/`hyper`/`tower`/`mio`/`tracing`으로 연계되는 강력한 자체 생태계
 - 애네들 위에서 지어진 상당수의 서드파티 crate가 `tokio` API 기반
 - 그리고 AWS가 밀어줍니다
 - *Ultimately, open source communities like Rust are about people, and we started hiring Rust and Tokio committers to ensure they'd have the time and resources necessary to further improve Rust. ([ref](#), emphasis mine)*
- `async-std`, `smol`도 충분히 좋음
 - 참고: 애네들과 별개로 '특수 목적' executor도 있음. [Gloomio](#), [embassy-executor](#)

Asynchronous Rust Ecosystem

- Hyper
 - 고성능 HTTP/1, HTTP/2 프로토콜 구현체 (HTTP/3도 개발 중)
 - 클라이언트(`hyper::Client`), 서버(`hyper::Server`) 구현 모두 있음
 - 클라이언트 구현은 curl 백엔드로 시험 적용됨
 - 서버 구현은 TechEmpower 웹 프레임워크 벤치마크 상위권 차지
- Reqwest
 - `hyper::Client`를 잘 감싸놓은 HTTP 클라이언트 라이브러리
 - 파이썬의 `requests` 와 비슷
- Axum
 - `hyper::Server`를 잘 감싸놓은 웹 서버 프레임워크
 - 무난한 API (Express 느낌?)

Asynchronous Rust Ecosystem

- Tower

- Service와 Layer라는 추상화를 제공하는 라이브러리(Future처럼)
- Service: `async fn(Request) -> Response`
 - `hyper::Client: async fn(http::Request) -> http::Response`
- Layer: `fn(Service) -> AnotherService`
 - Request나 Response 혹은 둘 다를 입출력 전후에 감싸줄 수 있음
 - Service를 감싸서 rate limit을 건다거나, 로깅을 한다거나, 타임아웃을 넣는다거나...
- Service, Layer들의 combinator, 기본 Layer 등을 제공

Asynchronous Rust Ecosystem

- [Mio](#)
 - 저수준 I/O API 제공
 - epoll(linux), kqueue(unix/macOS), ...
 - tokio의 근간

HYPERITHM

Final Wrap-up

- Asynchronous Rust의 원리를 배웠습니다
 - `async` \Rightarrow Future/poll/Pin/... \Rightarrow `epoll(7)` /kqueue \Rightarrow hardware
 - 기존 Rust의 장점인 강력한 타입 시스템과 메모리 안전성을 챙기면서 프로그래밍할 수 있습니다
- Asynchronous Rust는 어렵습니다
 - 비동기 프로그래밍은 원래 어렵습니다
 - 어려운 주제가 나오면 다른 언어(ex: Go)는 런타임 비용을 지불하고 추상화합니다
 - ex: Memory safety(GC), polymorphism(dynamic dispatch)
 - Rust는 zero-cost abstraction을 지향합니다.
 - 추상화에 따른 런타임 비용이 없습니다(=추상화 없이 손으로 쓴 것과 성능이 같음)
 - *we've added a comparison of minihhttp against a directly-coded state machine version in Rust (see "raw mio" in the link). The two are within 0.3% of each other.*
 - 대신에 개발이 힘듭니다: 컴파일러와 개발자가 비용을 지불합니다
 - 그럼 이 중에서 가장 힘든 건 누구일까요?
 - 정답은 컴파일러 개발자입니다.

We're Hiring!

- <https://hiring.hyperithm.com>
- 하이퍼리즘은 대한민국 최고의 암호화폐 자산 운용사입니다
- 시장을 이기는 차세대 전략을 위한 트레이딩 프레임워크를 개발하고 있습니다
 - 당연히 Asynchronous Rust를 광범위하게 사용 중입니다.
- Rust에 관심이 있거나 Rust를 사랑한다면 부담없이 지원 ㄱ ㄱ

References

- [Asynchronous Programming in Rust](#)
 - Asynchronous Rust의 좋은 입문서니 필독
- [IBM Developer: *Boost application performance using asynchronous I/O*](#)
- [Dan Kegel, *The C10K problem*](#)
- [Niko Matsakis, why async fn in traits are hard](#)
- [Aaron Turon, Zero-cost futures in Rust](#)

(진짜) 끝

- 감사합니다.
- 질문 마구마구 해주세요

HYPERITHM