

HSPACE Rust 특강

Rust Basic #17 - 비동기 프로그래밍, Part 2

Chris Ohk

utilForever@gmail.com

- 비동기식 클라이언트와 서버
 - 서버의 메인 함수
 - 채팅 연결 처리하기 : 비동기 뮤텍스
 - 그룹 테이블 : 동기 뮤텍스
 - 채팅 그룹 : tokio의 브로드캐스트 채널
- 기본 제공 류처와 이그제큐터
 - 웨이커 호출하기 : spawn_blocking
 - block_on 구현하기
- 핀 설정
 - 류처의 두 가지 생애 단계
 - 핀이 설정된 포인터
 - Unpin 트레잇

서버의 메인 함수

HSPACE Rust 특강
Rust Basic #17 - 비동기 프로그래밍, Part 2

- 채팅 서버의 메인 파일인 `main.rs`를 보자.

```
● ● ●

use async_chat::utils::ChatResult;
use async_std::prelude::*;

use std::sync::Arc;

mod connection;
mod group;
mod group_table;

use connection::serve;
```

서버의 메인 함수

HSPACE Rust 특강
Rust Basic #17 - 비동기 프로그래밍, Part 2

- 채팅 서버의 메인 파일인 `main.rs`를 보자.

```
● ● ●

fn main() -> ChatResult<()> {
    let address = std::env::args().nth(1).expect("Usage: server ADDRESS");
    let chat_group_table = Arc::new(group_table::GroupTable::new());

    async_std::task::block_on(async {
        use async_std::{net, task};

        let listener = net::TcpListener::bind(address).await?;
        let mut new_connections = listener.incoming();

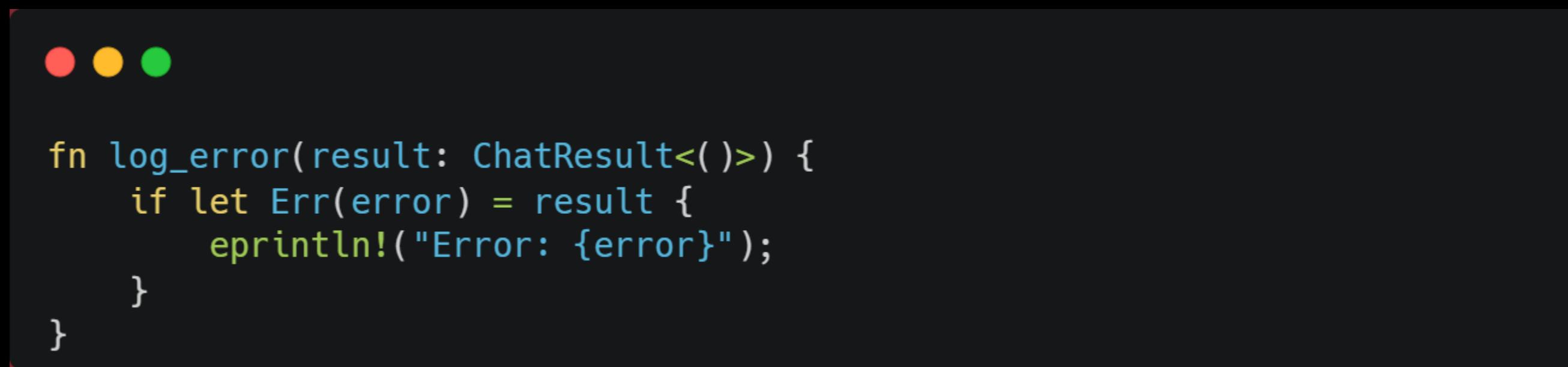
        while let Some(socket_result) = new_connections.next().await {
            let socket = socket_result?;
            let groups = chat_group_table.clone();

            task::spawn(async {
                log_error(serve(socket, groups).await);
            });
        }
    })
}
```

서버의 메인 함수

HSPACE Rust 특강
Rust Basic #17 - 비동기 프로그래밍, Part 2

- 채팅 서버의 메인 파일인 `main.rs`를 보자.



A screenshot of a terminal window with three colored window control buttons (red, yellow, green) at the top. The terminal displays the following Rust code:

```
fn log_error(result: ChatResult<()>) {
    if let Err(error) = result {
        eprintln!("Error: {error}");
    }
}
```

서버의 메인 함수

HSPACE Rust 특강
Rust Basic #17 - 비동기 프로그래밍, Part 2

- **코드 설명**

- 약간의 설정 작업을 하고 나서 `block_on`을 호출해 실제 작업을 하는 `async` 블록을 생성한다.
- 먼저 클라이언트에서 오는 연결을 처리하기 위해서 `TcpListener` 소켓을 만든다.
이 소켓의 `incoming` 메소드는 `std::io::Result<TcpStream>` 값의 스트림을 반환한다.
- 이어서 들어오는 연결마다 `connection::serve` 함수를 실행하는 비동기 태스크를 띄운다.
각 태스크는 `GroupTable` 값의 레퍼런스를 받는데, 이는 서버의 현재 채팅 그룹 목록을 표현하는 값으로
레퍼런스 카운트 기반의 포인터인 `Arc`를 통해서 모든 연결이 공유한다.
- `connection::serve` 함수가 오류를 반환하면 표준 오류 출력에 메시지를 기록하고 태스크를 종료한다.
다른 연결은 계속 그대로 실행된다.

채팅 연결 처리하기

HSPACE Rust 특강
Rust Basic #17 - 비동기 프로그래밍, Part 2

- 다음은 서버의 핵심 요소인 connection 모듈의 serve 함수다.

```
● ● ●

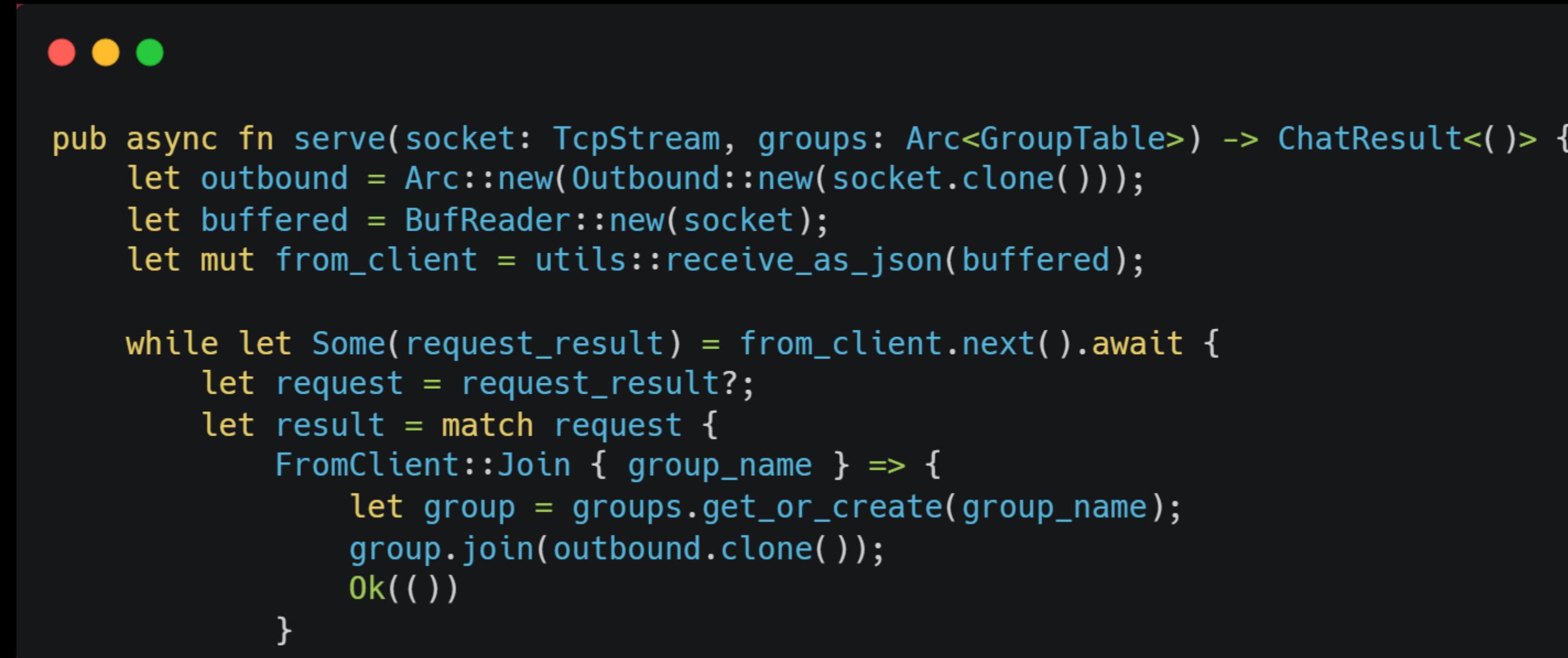
use async_chat::utils::{self, ChatResult};
use async_chat::{FromClient, FromServer};
use async_std::io::BufReader;
use async_std::net::TcpStream;
use async_std::prelude::*;
use async_std::sync::Arc;
use async_std::sync::Mutex;

use crate::group_table::GroupTable;
```

채팅 연결 처리하기

HSPACE Rust 특강
Rust Basic #17 - 비동기 프로그래밍, Part 2

- 다음은 서버의 핵심 요소인 connection 모듈의 serve 함수다.



```
pub async fn serve(socket: TcpStream, groups: Arc<GroupTable>) -> ChatResult<()> {
    let outbound = Arc::new(Outbound::new(socket.clone()));
    let buffered = BufReader::new(socket);
    let mut from_client = utils::receive_as_json(buffered);

    while let Some(request_result) = from_client.next().await {
        let request = request_result?;
        let result = match request {
            FromClient::Join { group_name } => {
                let group = groups.get_or_create(group_name);
                group.join(outbound.clone());
                Ok(())
            }
        }
    }
}
```

채팅 연결 처리하기

HSPACE Rust 특강
Rust Basic #17 - 비동기 프로그래밍, Part 2

- 다음은 서버의 핵심 요소인 connection 모듈의 serve 함수다.

```
● ● ●

FromClient::Post {
    group_name,
    message,
} => match groups.get(&group_name) {
    Some(group) => {
        group.post(message);
        Ok(())
    }
    None => Err(format!("Group '{group_name}' does not exist")),
},
};

if let Err(message) = result {
    let report = FromServer::Error(message);
    outbound.send(report).await?;
}
}

Ok(())
}
```

- **코드 설명**
 - 코드의 대부분을 차지하는 건 들어오는 `FromClient` 값의 스트림을 처리하는 반복문이다.
 - 이 스트림은 버퍼링되는 TCP 스트림을 `receive_as_json`으로 처리해서 만든다.
오류가 발생하면 `FromServer::Error` 패킷을 생성해서 처리를 실패했다고 클라이언트에게 전한다.
 - 클라이언트는 오류 메시지와 더불어 자신이 가입한 채팅 그룹에서 오는 메시지도 받아야 하므로
클라이언트의 연결을 각 그룹과 공유해야 한다.
 - 이럴 때는 그냥 모두에게 `TcpStream`의 복제본을 건네주면 되는데,
이렇게 하면 두 소스가 동시에 패킷을 소켓에 기록했을 때 출력이 서로 뒤섞여서
클라이언트가 잘못된 JSON을 받게 될 가능성이 있다.
따라서 연결을 동시에 접근해도 안전할 수 있도록 정리가 필요하다.

채팅 연결 처리하기

HSPACE Rust 특강
Rust Basic #17 - 비동기 프로그래밍, Part 2

- 이 부분은 Outbound 타입이 관리한다.

```
● ● ●

pub struct Outbound(Mutex<TcpStream>);

impl Outbound {
    pub fn new(to_client: TcpStream) -> Outbound {
        Outbound(Mutex::new(to_client))
    }

    pub async fn send(&self, packet: FromServer) -> ChatResult<()> {
        let mut guard = self.0.lock().await;

        utils::send_as_json(&mut *guard, &packet).await?;
        guard.flush().await?;

        Ok(())
    }
}
```

- **코드 설명**

- Outbound는 값이 생성될 때 TcpStream의 소유권을 가져다가 한 번에 한 태스크만 쓸 수 있도록 Mutex로 감싸 둔다.
- serve 함수는 클라이언트가 가입한 모든 그룹이 공유된 같은 Outbound 인스턴스를 가리킬 수 있도록 각 Outbound를 레퍼런스 카운트 기반의 포인터인 Arc로 감싸 둔다.
- Outbound::send 호출은 먼저 뮤텍스를 잠궈 내부에 있는 TcpStream을 역참조하는 가드(Guard) 값을 가져온다. 그런 다음 send_as_json을 써서 packet을 전송하고, 끝으로 guard.flush()를 호출해서 어딘가에 있는 버퍼에 전송되다 만 데이터가 남아 있는 일이 없도록 정리한다.
(사실 TcpStream은 데이터를 버퍼링하지 않지만, Write 트레잇이 그런 구현을 허용하므로 만약을 위해 처리해 둔다.)
- 표현식 &mut *guard를 쓰면 Rust가 트레잇 바운드를 만족하는지 볼 때 Deref 강제 변환을 적용하지 않는다는 사실을 회피할 수 있다. 즉, 뮤텍스 가드를 명시적으로 역참조한 다음 보호하고 있는 TcpStream의 변경할 수 있는 레퍼런스를 빌려 오는 식으로 send_as_json이 요구하는 &mut Stream을 산출하는 것이다.

- Outbound는 `std::sync::Mutex`가 아니라 `async_std::sync::Mutex` 타입을 쓴다.
여기에는 세 가지 이유가 있다.
 - 첫 번째로 `std::sync::Mutex`는 태스크가 뮤텍스 가드를 준 채로 중단되면 오작동할 수도 있다.
 - 해당 태스크를 실행하던 스레드가 같은 `Mutex`를 잠그려는 또 다른 태스크를 집어 들면 문제가 터지는데, `Mutex`의 입장에서 보면 이미 자신을 소유하고 있는 스레드가 또 한 번 자신을 잠그려고 하는 셈이기 때문이다.
 - 표준 `Mutex`는 이런 경우를 처리하게끔 설계되지 않아서 패닉이나 교착 상태에 빠진다.
- Rust는 현재 `std::sync::Mutex` 가드의 수명 안에 `await` 표현식이 들어가 있을 때마다 컴파일 시점에 경고를 내보내도록 만드는 작업이 진행중이다.
- `Outbound::send`는 `send_as_json`과 `guard.flush`의 퓨처를 기다리는 동안 락을 주고 있어야 하므로 반드시 `async_std`의 `Mutex`를 써야 한다.

채팅 연결 처리하기

HSPACE Rust 특강
Rust Basic #17 - 비동기 프로그래밍, Part 2

- Outbound는 `std::sync::Mutex`가 아니라 `async_std::sync::Mutex` 타입을 쓴다.
여기에는 세 가지 이유가 있다.
 - 두 번째로 비동기 `Mutex`의 `lock` 메소드는 가드의 퓨처를 반환하기 때문에,
뮤텍스를 잠그려고 기다리는 태스크는 뮤텍스가 준비될 때까지 자신의 스레드를 다른 태스크에게 양보한다.
(뮤텍스가 이미 사용 가능한 상태라면 `lock` 퓨처는 즉시 준비 상태가 되므로 태스크 자체가 중단되는 일이 없다.)
 - 반면 표준 `Mutex`의 `lock` 메소드는 락을 획득할 때까지 기다리는 동안 전체 스레드를 꼼짝 못하게 잡아 둔다.
 - 앞에 있는 코드는 네트워크에 패킷을 전송하는 동안 뮤텍스를 줘고 있으므로 시간이 꽈 걸릴 수도 있다.

채팅 연결 처리하기

HSPACE Rust 특강
Rust Basic #17 - 비동기 프로그래밍, Part 2

- Outbound는 `std::sync::Mutex`가 아니라 `async_std::sync::Mutex` 타입을 쓴다.
여기에는 세 가지 이유가 있다.
 - 세 번째로 표준 Mutex는 락을 건 스레드만 락을 풀 수 있다.
 - 이 규칙을 시행하기 위해 표준 Mutex의 가드 타입은 `Send`를 구현하고 있지 않으며, 따라서 다른 스레드로 전송할 수 없다.
 - 이 말은 이런 가드를 줘고 있는 퓨처 자체도 `Send`를 구현하고 있지 않기 때문에 `spawn`에 넘겨서 스레드 풀을 가지고 실행할 수 없다는 뜻으로, `block_on`이나 `spawn_local`을 써서 실행할 수 밖에 없다.
 - `async_std` Mutex용 가드는 `Send`를 구현하고 있으므로 `spawn`을 통해 실행되는 태스크에서 써도 아무 문제가 없다.

채팅 연결 처리하기

HSPACE Rust 특강
Rust Basic #17 - 비동기 프로그래밍, Part 2

- '비동기 코드에서는 항상 `async_std::sync::Mutex`를 써라'
이 이야기를 하려는 게 아니다. 그렇게 단순하지 않다.
 - 뮤텍스를 주고 있는 동안 아무것도 기다릴 필요가 없을 때도 있고 락을 오래 주고 있지 않을 때도 많기 때문에,
이런 경우에는 표준 라이브러리의 `Mutex`가 훨씬 더 효율적일 수 있다.
 - 다음에 볼 채팅 서버의 `GroupTable` 타입이 바로 이런 경우에 해당한다.

그룹 테이블

HSPACE Rust 특강
Rust Basic #17 - 비동기 프로그래밍, Part 2

- 다음은 group_table.rs의 전체 내용이다.

```
● ● ●

use crate::group::Group;
use std::collections::HashMap;
use std::sync::{Arc, Mutex};

pub struct GroupTable(Mutex<HashMap<Arc<String>, Arc<Group>>>);

impl GroupTable {
    pub fn new() -> GroupTable {
        GroupTable(Mutex::new(HashMap::new()))
    }

    pub fn get(&self, name: &String) -> Option<Arc<Group>> {
        self.0.lock().unwrap().get(name).cloned()
    }

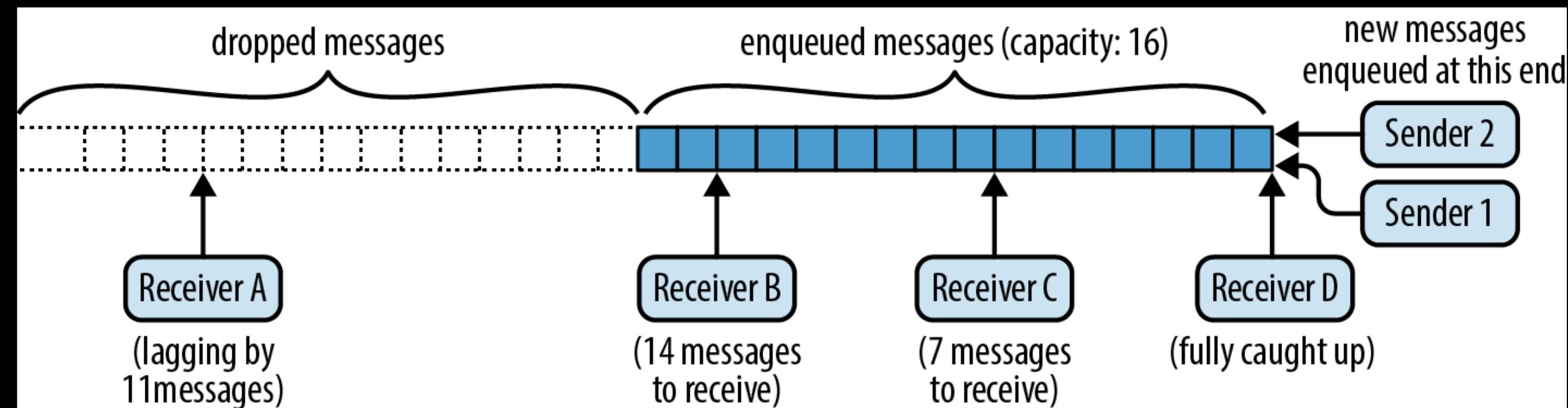
    pub fn get_or_create(&self, name: Arc<String>) -> Arc<Group> {
        self.0
            .lock()
            .unwrap()
            .entry(name.clone())
            .or_insert_with(|| Arc::new(Group::new(name)))
            .clone()
    }
}
```

- **코드 설명**
 - GroupTable은 뮤텍스로 보호되는 간단한 해시 테이블로 채팅 그룹 이름을 실제 그룹에 매핑하며, 둘 다 레퍼런스 카운트 기반의 포인터로 관리한다. get과 get_or_create 메소드는 뮤텍스에 락을 걸고 몇 가지 해시 테이블 작업을 수행한 뒤 복귀하는데, 어쩌면 이 과정에서 할당이 몇 차례 일어날 수도 있다.
 - GroupTable은 평범하고 오래된 std::sync::Mutex를 쓴다.
이 모듈에는 비동기 코드가 전혀 없으므로 피해야 할 await도 없다.
만약 여기에 async_std::sync::Mutex를 썼다면 get과 get_or_create를 비동기 함수로 만들어야 했을 테니, 혜택은 적은데 퓨처를 생성하고 중단하고 재개하는 데서 오는 오버헤드만 떠안는 꼴이 됐을 것이다.
뮤텍스는 일부 해시 연산과 어쩌면 있을지도 모를 몇 차례의 할당 과정에서만 락을 걸면 된다.
 - 채팅 서버의 사용자가 수백만 명으로 불어나서 GroupTable 뮤텍스가 병목이 되면 그땐 이를 비동기로 바꾼다고 해서 문제가 해결되지 않는다. 이럴 때는 HashMap 말고 동시 접근에 특화된 다른 타입을 쓰는 게 더 낫다.
예를 들어, dashmap 크레이트는 그런 타입을 제공한다.

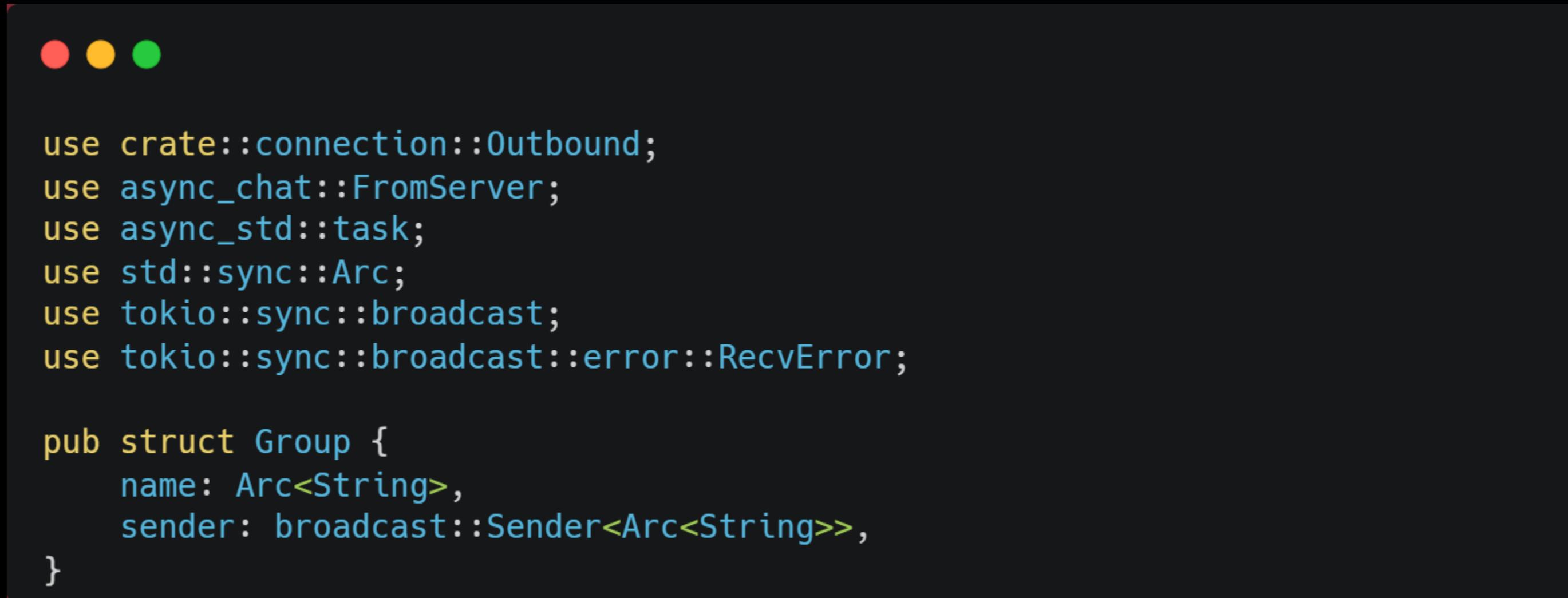
- 서버에서 `group::Group` 타입은 채팅 그룹을 표현한다.
 - 이 타입은 `connection::serve`가 호출하는 두 가지 메소드만 지원하면 되는데, 새 멤버를 추가하는 `join`과 메시지를 보내는 `post`가 바로 그것이다. 보낸 메시지는 모든 멤버에게 전달되어야 한다.
- 배압(Backpressure)을 잘 처리해야 하는 부분이 바로 여기다.
여기에는 요구 사항 몇 가지가 서로 팽팽하게 맞서는 모양새로 돼 있다.
 - 어떤 멤버가 (네트워크 연결이 느리다던지 하는 등의 이유로) 그룹에 전달된 메시지를 그때그때 따라잡을 수 없더라도 그룹 안에 있는 다른 멤버가 영향을 받아서는 안 된다.
 - 비록 좀 뒤쳐지는 멤버가 있더라도 어떻게든 대화에 다시 합류해서 계속 참여할 방법이 있어야 한다.
 - 메시지를 버퍼링하는 데 쓰는 메모리가 한도 없이 늘어나면 안 된다.

- 이런 요구는 다대다 통신 패턴을 구현할 때면 늘 있는 일이다.
 - 그래서 tokio 크레이트는 합리적인 절충안 하나를 구현하고 있는 브로드캐스트 채널(Broadcast Channel) 타입을 제공한다. tokio 브로드캐스트 채널은 서로 다른 여러 스레드가 태스크가 값을 주고 받을 수 있는 값(채팅 메시지)의 큐다.
 - '브로드캐스트' 채널이라고 부르는 이유는 모든 소비자가 전달된 개별 값의 복사본을 받기 때문이다.
(이로 인해 값 타입은 반드시 Clone을 구현해야 된다.)
 - 보통 브로드캐스트 채널은 모든 소비자가 복사본을 받을 때까지 메시지를 큐에 유지해 둔다.
그러나 큐의 길이가 채널을 생성할 때 설정한 최대 용량을 넘어서면 가장 오래된 메시지가 삭제된다.
따라서 이를 따라잡지 못한 소비자는 다음 메시지를 가져오려고 할 때 오류를 받게 되며,
채널은 해당 소비자가 다음번에 현재 시점에서 가장 오래된 메시지를 받아볼 수 있도록 조치한다.

- 다음은 최대 16개의 값을 가질 수 있는 브로드캐스트 채널을 보여 준다.
 - 두 Sender는 메시지를 큐에 넣고 네 Receiver는 메시지를 큐에서 뺀다. (정확하게는 메시지를 큐에서 복사한다.)
 - Receiver B는 14개의 메시지를 덜 받았고, Receiver C는 7개의 메시지를 덜 받았고, Receiver D는 메시지를 전부 받았다. Receiver A는 완전히 뒤쳐져서 아직 받지도 못한 11개의 메시지가 삭제된 상태다.
 - 따라서 다음번에 메시지를 가져오려고 하면 상황 설명이 담긴 오류가 반환되며 실패하고, 다음번에 현재 시점에서 가장 오래된 메시지를 받아볼 수 있도록 조치한다.



- 채팅 서버는 각 채팅 그룹을 `Arc<String>` 값을 실어 나르는 브로드캐스트 채널로 표현한다.
그룹에 메시지를 보내면 현재 시점의 모든 멤버에게 브로드캐스트한다.



```
● ● ●

use crate::connection::Outbound;
use async_chat::FromServer;
use async_std::task;
use std::sync::Arc;
use tokio::sync::broadcast;
use tokio::sync::broadcast::error::RecvError;

pub struct Group {
    name: Arc<String>,
    sender: broadcast::Sender<Arc<String>>,
}
```

- 채팅 서버는 각 채팅 그룹을 `Arc<String>` 값을 실어 나르는 브로드캐스트 채널로 표현한다.
그룹에 메시지를 보내면 현재 시점의 모든 멤버에게 브로드캐스트한다.

```
● ● ●

impl Group {
    pub fn new(name: Arc<String>) -> Group {
        let (sender, _receiver) = broadcast::channel(1000);
        Group { name, sender }
    }

    pub fn join(&self, outbound: Arc<Outbound>) {
        let receiver = self.sender.subscribe();

        task::spawn(handle_subscriber(self.name.clone(), receiver, outbound));
    }

    pub fn post(&self, message: Arc<String>) {
        // This only returns an error when there are no subscribers.
        // A connection's outgoing side can exit, dropping its subscription,
        // slightly before its incoming side, which may end up trying to send
        // a message to an empty group.
        let _ignored = self.sender.send(message);
    }
}
```

- **코드 설명**
 - Group 구조체에는 채팅 그룹의 이름과 더불어 그 그룹의 브로드캐스트 채널에서 Sender를 표현하는 `broadcast::Sender`가 들어 있다.
 - `new` 함수는 `broadcast::channel`을 호출해서 최대 1,000개의 메시지를 가질 수 있는 브로드캐스트 채널을 만든다.
 - `channel` 함수는 Sender와 Receiver를 모두 반환하지만 그룹이 아직 멤버를 갖지 않으므로 현재 시점에서 Receiver는 필요 없다.
 - 새 그룹을 그룹에 추가하기 위해서 `join` 메소드는 Sender의 `subscribe` 메소드를 호출해 해당 채널의 Receiver를 새로 하나 만든다. 그런 다음 Receiver의 메시지를 모니터링해서 이를 다시 클라이언트로 보내는 `handle_subscribe` 함수를 비동기 태스크로 생성해 실행한다.

- **코드 설명**
 - 이런 세부 사항을 알고 나면 post 메소드가 하는 일을 쉽게 이해할 수 있는데, 이 메소드는 단순히 메시지를 브로드캐스트 채널에 보낸다.
 - 채널이 실어 나르는 값은 Arc<String> 값이므로 Receiver마다 메시지의 복사본을 넘겨주더라도 그 메시지의 레퍼런스 카운트만 증가할 뿐 복사나 힙 할당이 전혀 발생하지 않는다.
 - 모든 구독자가 메시지를 전송하고 나면 레퍼런스 카운트는 0으로 떨어지고 메시지는 해제된다.

- `handle_subscriber`의 정의는 다음과 같다.

```
● ● ●

async fn handle_subscriber(
    group_name: Arc<String>,
    mut receiver: broadcast::Receiver<Arc<String>>,
    outbound: Arc<Outbound>,
) {
    loop {
        let packet = match receiver.recv().await {
            Ok(message) => FromServer::Message {
                group_name: group_name.clone(),
                message: message.clone(),
            },
            Err(RecvError::Lagged(n)) => {
                FromServer::Error(format!("Dropped {} messages from {}", group_name))
            }
            Err(RecvError::Closed) => break,
        };

        if outbound.send(packet).await.is_err() {
            break;
        }
    }
}
```

- **코드 설명**
 - 브로드캐스트 채널에서 메시지를 받은 후 이를 다시 고유된 Outbound 값을 통해서 클라이언트로 전송하는 반복문이다.
반복문이 브로드캐스트 채널을 그때그때 따라잡지 못해서 Lagged 오류를 받게 되면
이 사실을 클라이언트에 있는 그대로 알린다.
 - 연결이 끊겼든지 하는 등의 이유로 패킷을 다시 클라이언트로 보내는데 완전히 실패하면
`handle_subscriber`가 반복문을 빠져나와 복귀하므로 비동기 태스크는 종료된다.
그리고 브로드캐스트 채널의 `Receiver`가 드롭되어 채널 구독이 해제된다.
이런 식으로 연결이 끊어지면 다음번에 그룹이 메시지를 보내려 할 때 해당 그룹의 멤버 등록이 취소된다.
 - 단, 채팅 그룹은 그룹 테이블에서 제거하지 않기 때문에 폐쇄되지 않는다.
그러나 완성도를 위해 `handle_subscriber`는 태스크가 종료할 때 발생하는 `Closed` 오류를 처리하도록 되어 있다.

- 채팅 서버 분석
 - TcpListener와 broadcast 채널 같은 비동기 기본 요소를 써서 코드를 작성하는 법과 block_on과 spawn 같은 이그제큐터를 써서 실행을 끌어나가는 법을 보여준다.
이제 이런 것들이 어떤 식으로 구현되어 있는지를 들여다 보자.
 - 핵심은 퓨처가 Poll::Pending을 반환할 때 이그제큐터를 조율해서 적절한 시점에 다시 폴링하게 만드는 방식에 있다.

- 채팅 클라이언트의 `main` 함수에서 다음 코드를 실행할 때 무슨 일이 벌어지는지 생각해 보자.
 - `block_on`이 맨 처음 `async` 블록의 퓨처를 폴링할 때는 네트워크 연결이 바로 준비되지 못할 게 뻔하므로 `block_on`은 잠자기 상태에 들어간다. 그럼 언제 깨어나야 할까? 어떤 식으로든 네트워크 연결이 준비되면 `TcpStream`이 `block_on`에게 `async` 블록의 퓨처를 다시 폴링해야 한다고 알려줘야 한다. 왜냐하면 이번에 다시 폴링하면 `await`가 완료되고, `async` 블록이 진도를 빨 수 있다는 걸 알고 있는게 바로 `TcpStream`이기 때문이다.

```
task::block_on(async {
    let socket = net::TcpStream::connect(address).await?;
    ...
})
```

- 채팅 클라이언트의 `main` 함수에서 다음 코드를 실행할 때 무슨 일이 벌어지는지 생각해 보자.
 - `block_on` 같은 이그제큐터는 퓨처를 폴링할 때 웨이커(Waker)라고 하는 콜백을 꼭 넘겨야 한다. `Future` 트레잇의 규칙에 따르면 퓨처는 준비 상태가 아닐 때 일단 `Poll::Pending`을 반환하고 나중에 퓨처를 다시 폴링해도 좋은 시점이 되면 웨이커를 호출하도록 조율해야 한다.



```
task::block_on(async {
    let socket = net::TcpStream::connect(address).await?;
    ...
})
```

- 따라서 손으로 쓴 Future의 구현은 다음과 같은 모습일 때가 많다.

```
use std::task::Waker;

struct MyPrimitiveFuture {
    ...
    waker: Option<Waker>,
}

impl Future for MyPrimitiveFuture {
    type Output = ...;

    fn poll(mut self: Pin<&mut Self>, cx: &mut Context<'_>) -> Poll<...> {
        ...

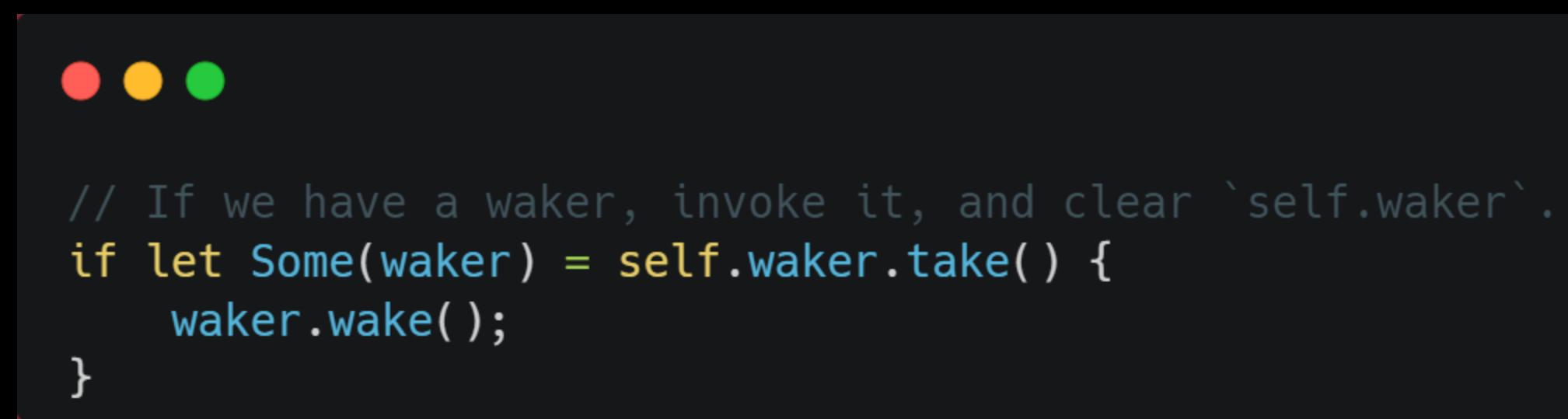
        if ... future is ready ... {
            return Poll::Ready(final_value);
        }

        // Save the waker for later.
        self.waker = Some(cx.waker().clone());

        Poll::Pending
    }
}
```

- **코드 설명**

- 즉, 퓨처의 값이 준비됐으면 이를 반환하고,
그렇지 않으면 Context에 있는 웨이커의 복사본을 어딘가 넣어 두고 Poll::Pending을 반환한다.
- 퓨처를 다시 폴링해도 좋은 시점이 되면 퓨처는 웨이커의 wake 메소드를 호출해서
자신을 폴링했던 마지막 이그제큐터에게 이 사실을 알려야 한다.



```
// If we have a waker, invoke it, and clear `self.waker`.
if let Some(waker) = self.waker.take() {
    waker.wake();
}
```

- 이상적으로는 이그제큐터와 퓨처가 번갈아가며 폴링하고 깨어나기를 반복하는게 바람직하다.
이그제큐터가 퓨처를 폴링하다가 잠자기 상태에 들어가면, 퓨처가 웨이커를 호출해서 이그제큐터를 깨우고,
그럼 다시 이그제큐터가 퓨처를 폴링하다가 잠자기 상태에 들어가고 하는 식으로 말이다.

- **코드 설명**
 - `async` 함수와 블록의 퓨처가 웨이커 자체를 다루는 건 아니다.
단지 주어진 컨텍스트를 자신이 기다리는 서브퓨처에 넘겨서 웨이커를 저장하고 호출하는 의무를 위임할 뿐이다.
채팅 클라이언트에서는 `async` 블록의 퓨처에 대한 첫 번째 폴링이 `TcpStream::connect`의 퓨처를 기다릴 때
컨텍스트를 넘긴다. 이어지는 폴링도 마찬가지로 블록이 기다리는 다음 차례의 퓨처에 자신의 컨텍스트를 넘긴다.
 - `TcpStream::connect`의 퓨처 핸들은 앞서 본 예처럼 폴링된다.
즉, 연결이 준비되길 기다렸다가 웨이커를 호출하는 도우미 스레드에 웨이커를 전달한다.

- **코드 설명**

- Waker는 Clone과 Send를 구현하고 있으므로 퓨처는 언제라도 웨이커의 복사본을 만들어 필요할 때 다른 스레드로 보낼 수 있다. Waker::wake는 웨이커를 소비한다.
그렇지 않은 wake_by_ref 메소드도 있지만 일부 이그제큐터는 소비하는 버전으로 좀 더 효율적으로 구현할 수 있다.
- 이그제큐터가 퓨처를 과도하게 폴링하는 건 전혀 문제될 게 없으며 단지 비효율적일 뿐이다.
하지만 퓨처는 폴링이 실제로 진도를 빨 수 있을 때만 웨이커를 호출하도록 주의해야 한다.
그렇지 않은데 깨워서 폴링하는 주기가 생기면 이그제큐터가 잠자기 상태에 들어가는 걸 방해해서 전력을 낭비하게 되고 프로세서가 다른 태스크에 쏟아야 할 시간을 확보하기 어렵게 된다.

- `spawn_blocking` 구현
 - 앞에서 `spawn_blocking` 함수가 주어진 클로저를 다른 스레드에서 실행시키고 반환값의 퓨처를 반환한다고 설명했다. 이제 `spawn_blocking`을 직접 구현하는 데 필요한 모든 부분이 갖춰졌다. 코드를 단순하게 가져가기 위해서 우리는 `async_std` 버전처럼 스레드 풀을 쓰지 않고 클로저마다 새 스레드를 만든다.
 - `spawn_blocking`이 퓨처를 반환하긴 하지만 이를 `async fn`으로 작성하진 않을 것이다. 그게 아니라 `SpawnBlocking` 구조체를 반환하는 평범한 동기 함수로 만들고 구조체에 직접 `Future`를 구현할 것이다.

웨이커 호출하기

HSPACE Rust 특강
Rust Basic #17 - 비동기 프로그래밍, Part 2

- `spawn_blocking`의 시그니처는 다음과 같다.
 - 클로저를 다른 스레드에 보내고 반환값을 다시 가져와야 하므로 클로저 F와 반환값 T는 둘 다 Send를 구현해야 한다.
 - 또 스레드가 얼마나 오래 실행될지 전혀 모르므로 둘 다 ‘static’이어야 한다.이들 바운드는 전부 `std::thread::spawn`이 요구하는 것과 똑같다.

```
● ● ●

pub fn spawn_blocking<T, F>(closure: F) -> SpawnBlocking<T>
where
    F: FnOnce() -> T,
    F: Send + 'static,
    T: Send + 'static,
```

- `SpawnBlocking<T>`는 클로저의 반환값에 대한 퓨처다.
 - Shared 구조체는 퓨처와 클로저를 실행 중인 스레드 간에 전달을 담당해야 하므로 Arc가 소유하고 Mutex가 보호한다.
 - 퓨처를 폴링하면 `value`가 있는지 보고, 없으면 웨이커를 `waker`에 저장한다.
 - 클로저를 실행하는 스레드는 반환값을 `value`에 저장하고 `waker`가 있으면 호출한다.

```
use std::sync::{Arc, Mutex};
use std::task::Waker;

pub struct SpawnBlocking<T>(Arc<Mutex<Shared<T>>>);

struct Shared<T> {
    value: Option<T>,
    waker: Option<Waker>,
}
```

웨이커 호출하기

HSPACE Rust 특강
Rust Basic #17 - 비동기 프로그래밍, Part 2

- `spawn_blocking<T>`의 전체 정의는 다음과 같다.

```
pub fn spawn_blocking<T, F>(closure: F) -> SpawnBlocking<T>
where
    F: FnOnce() -> T,
    F: Send + 'static,
    T: Send + 'static,
{
    let inner = Arc::new(Mutex::new(Shared {
        value: None,
        waker: None
    }));

    std::thread::spawn({
        let inner = inner.clone();

        move || {
            let value = closure();
            let maybe_waker = {
                let mut guard = inner.lock().unwrap();
                guard.value = Some(value);
                guard.waker.take()
            };

            if let Some(waker) = maybe_waker {
                waker.wake();
            }
        }
    });
}

SpawnBlocking(inner)
```

웨이커 호출하기

HSPACE Rust 특강
Rust Basic #17 - 비동기 프로그래밍, Part 2

- **코드 설명**

- 먼저 Shared 값을 생성한 다음 클로저를 실행하기 위한 스레드를 띄운다.
- 그리고 결과를 Shared의 value 필드에 저장한다.
- 마지막으로 웨이커가 있으면 호출한다.

웨이커 호출하기

HSPACE Rust 특강
Rust Basic #17 - 비동기 프로그래밍, Part 2

- `SpawnBlocking`을 위한 `Future`은 다음처럼 구현할 수 있다.

```
● ● ●

use std::future::Future;
use std::pin::Pin;
use std::task::{Context, Poll};

impl<T: Send> Future for SpawnBlocking<T> {
    type Output = T;

    fn poll(self: Pin<&mut Self>, cx: &mut Context<'_>) -> Poll<T> {
        let mut guard = self.0.lock().unwrap();

        if let Some(value) = guard.value.take() {
            return Poll::Ready(value);
        }

        guard.waker = Some(cx.waker().clone());
        Poll::Pending
    }
}
```

- 코드 설명
 - `SpawnBlocking`을 폴링하면 클로저의 값이 준비됐는지 보고, 그렇다면 소유권을 가져다가 반환한다.
 - 그렇지 않다면 퓨처는 아직 계류 중이므로 컨텍스트에 있는 웨이커의 복사본을 퓨처의 `waker` 필드에 저장한다.
 - `Future`가 `Poll::Ready`를 반환한 뒤에는 다시 폴링하지 말아야 한다.
`await`와 `block_on` 같은 퓨처를 소비하는 일반적인 방식은 전부 이 규칙을 따른다.
`SpawnBlocking` 퓨처를 과도하게 폴링한다고 해서 특별히 끔찍한 일이 일어나는 건 아니지만,
그렇다고 해서 누군가가 그런 경우를 처리하고 있다는 뜻은 아니다. 이는 손으로 쓴 퓨처의 전형적인 특징이다.

block_on 구현하기

HSPACE Rust 특강
Rust Basic #17 - 비동기 프로그래밍, Part 2

- 우리만의 `block_on` 버전을 만들어 보자.
 - 기본 제공 류처를 구현하는데 있어서 간단한 이그제큐터를 구축하는데 필요한 모든 부분을 갖춘 상태다.
 - `async_std` 버전에 비하면 훨씬 단순한데, 예를 들어 `spawn_local`, 태스크 로컬 변수, 중첩 호출은 지원하지 않는다.
 - 그러나 채팅 클라이언트와 서버를 돌리는 데는 충분하다.

block_on 구현하기

HSPACE Rust 특강
Rust Basic #17 - 비동기 프로그래밍, Part 2

- 코드는 다음과 같다.

```
● ● ●

use crossbeam::sync::Parker;
use futures_lite::pin;
use std::future::Future;
use std::task::{Context, Poll};
use waker_fn::waker_fn;

fn block_on<F: Future>(future: F) -> F::Output {
    let parker = Parker::new();
    let unparker = parker.unparker().clone();
    let waker = waker_fn(move || unparker.unpark());
    let mut context = Context::from_waker(&waker);

    pin!(future);

    loop {
        match future.as_mut().poll(&mut context) {
            Poll::Ready(value) => return value,
            Poll::Pending => parker.park(),
        }
    }
}
```

block_on 구현하기

HSPACE Rust 특강
Rust Basic #17 - 비동기 프로그래밍, Part 2

- **코드 설명**

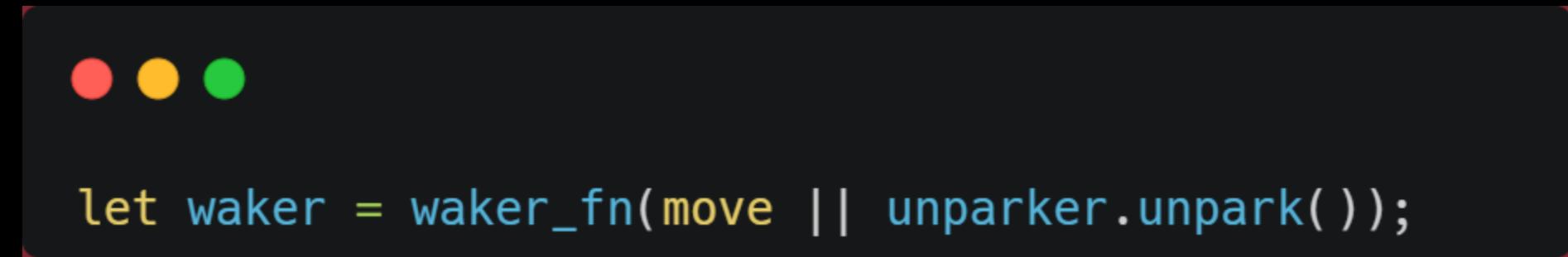
```
● ● ●  
let parker = Parker::new();  
let unparker = parker.unparker().clone();
```

- crossbeam 크레이트의 Parker 타입은 간단한 블로킹 기본 요소다.
- parker.park()를 호출하면 다른 누군가가 그에 대응하는 Unparker에 대고 .unpark()를 호출할 때까지 스레드가 블록된다.
- Unparker는 parker.unparker()를 호출해서 미리 받아둘 수 있다.
- 아직 파킹되지 않은 스레드를 unpark하면 다음 park 호출이 블로킹되지 않고 바로 복귀한다.
- 여기서는 퓨처가 준비 상태가 아닐 때마다 Parker를 써서 대기하고 퓨처에 넘기는 웨이커가 이를 언파킹한다.

block_on 구현하기

HSPACE Rust 특강
Rust Basic #17 - 비동기 프로그래밍, Part 2

- 코드 설명



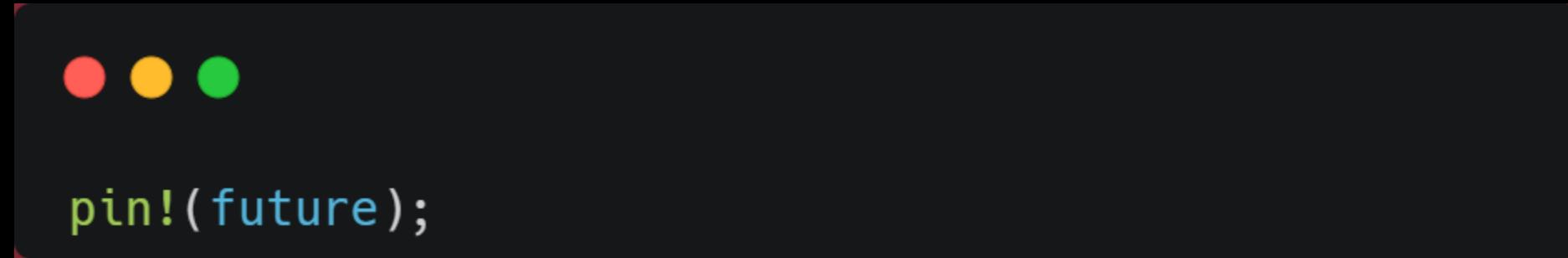
```
let waker = waker_fn(move || unparker.unpark());
```

- waker_fn 크레이트의 waker_fn 함수는 주어진 클로저를 가지고 Waker를 생성한다.
- 여기서는 호출되면 클로저 move || unparker.unpark()를 호출하는 Waker를 만든다.
- std::task::Wake 트레잇을 구현해서 웨이커를 만들 수도 있지만, 여기서는 waker_fn이 좀 더 편하다.

block_on 구현하기

HSPACE Rust 특강
Rust Basic #17 - 비동기 프로그래밍, Part 2

- 코드 설명

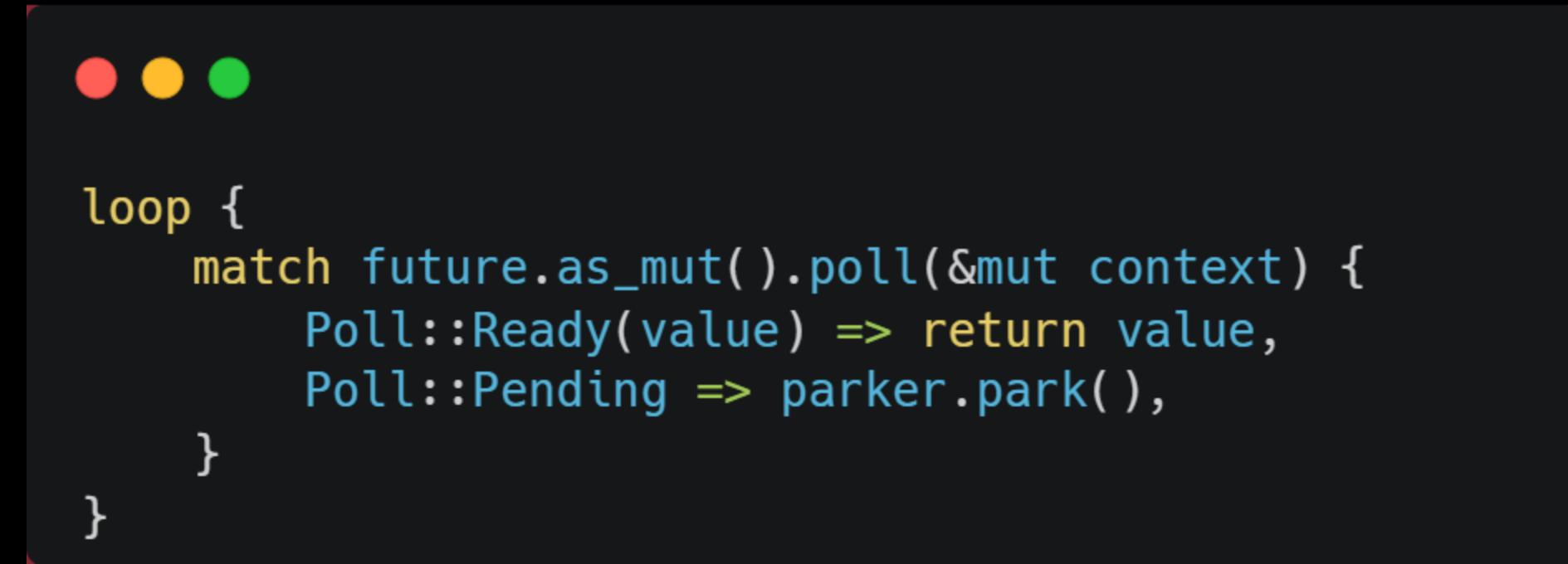


- `pin!` 매크로는 `F` 타입의 퓨처를 줘고 있는 변수가 주어지면 해당 퓨처의 소유권을 가져다가 같은 이름으로 된 `Pin<&mut F>` 타입의 새 변수를 선언한다. 즉, 이 변수는 `F` 타입의 퓨처를 빌려온다.
- 이 코드로 `poll` 메소드가 요구하는 `Pin<&mut Self>`를 확보할 수 있다.
- 비동기 함수와 블록의 퓨처는 `Pin`을 통해서 참조해야 폴링할 수 있는데 왜 그런지는 뒤에서 설명한다.

block_on 구현하기

HSPACE Rust 특강
Rust Basic #17 - 비동기 프로그래밍, Part 2

- 코드 설명



```
loop {
    match future.as_mut().poll(&mut context) {
        Poll::Ready(value) => return value,
        Poll::Pending => parker.park(),
    }
}
```

- 끝으로 폴링 반복문은 아주 단순하다.
웨이커를 들고 있는 컨텍스트를 퓨처에 넘겨서 Poll::Ready가 반환될 때까지 폴링한다.
- Poll::Pending이 반환되면 스레드를 파킹하고 waker가 호출될 때까지 블록 상태에 들어간다. 그런 다음 재시도한다.
- as_mut 호출을 쓰면 소유권을 포기하지 않은 채로 future를 폴링할 수 있는데, 이 부분은 뒤에 더 자세히 설명한다.

- Pin 타입은 Rust가 비동기 함수와 블록을 안전하게 쓰이도록 만드는데 도움을 준다.
 - 먼저 비동기 함수 호출과 블록의 퓨처를 왜 평범한 Rust 값처럼 자유롭게 다룰 수 없는지 살펴본다.
 - 이어서 Pin이 대체 어떤 식으로 작동하기에 이런 퓨처를 안전하게 관리한다고 믿어도 되는지 알아본다.
 - 끝으로 Pin 값을 써서 일하는 법 몇 가지를 살펴본다.

퓨처의 두 가지 생애 단계

HSPACE Rust 특강
Rust Basic #17 - 비동기 프로그래밍, Part 2

- 다음에 있는 간단한 비동기 함수를 보자.
 - 이 함수는 주어진 주소를 가지고 TCP 연결을 열어서 해당 서버가 보내온 것을 전부 String으로 반환한다.
 - 상자로 표시한 지점은 **재개 지점(Resumption Point)**으로 비동기 함수의 코드에서 실행이 중단될 수도 있는 지점이다.

```
● ● ●

use async_std::io::prelude::*;
use async_std::{io, net};

async fn fetch_string(address: &str) -> io::Result<String> {
    1
    let mut socket = net::TcpStream::connect(address).await?;
    let mut buf = String::new(); 2

    socket.read_to_string(&mut buf).await?;
    3
    Ok(buf)
}
```

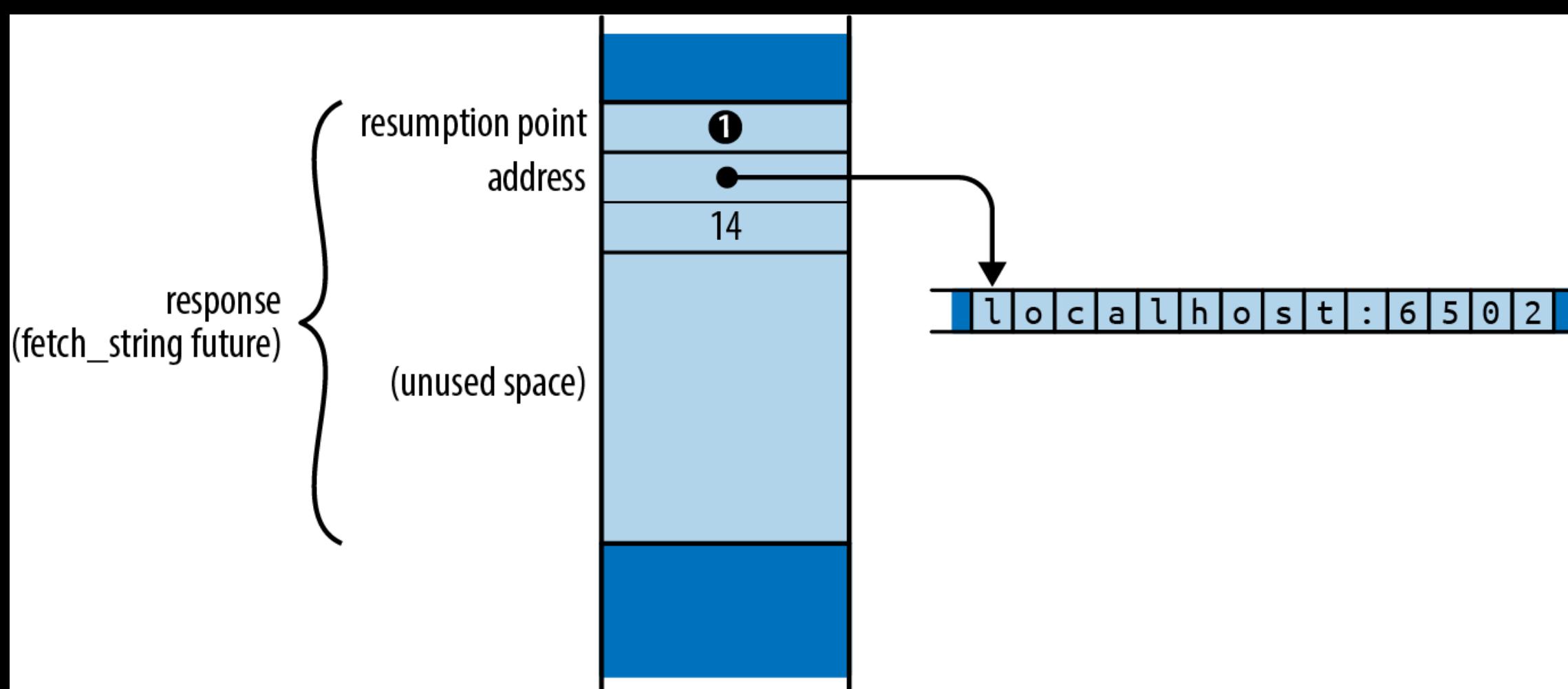
퓨처의 두 가지 생애 단계

HSPACE Rust 특강
Rust Basic #17 - 비동기 프로그래밍, Part 2

- 이 함수를 다음처럼 await 없이 호출한다고 가정해 보자.

```
● ● ●  
let response = fetch_string("localhost:6502");
```

- response는 fetch_string을 주어진 인수를 가지고 맨 앞에서부터 실행해 나가는 퓨처다. 이 퓨처는 아래의 메모리 구조를 갖는다.
- 방금 만들었으므로 실행이 함수 본문의 맨 꼭대기에 있는 재개 지점 [1]에서 시작되어야 한다. 이 상태에서 퓨처가 진도를 빼는데 필요한 값은 함수 인수 뿐이다.



퓨처의 두 가지 생애 단계

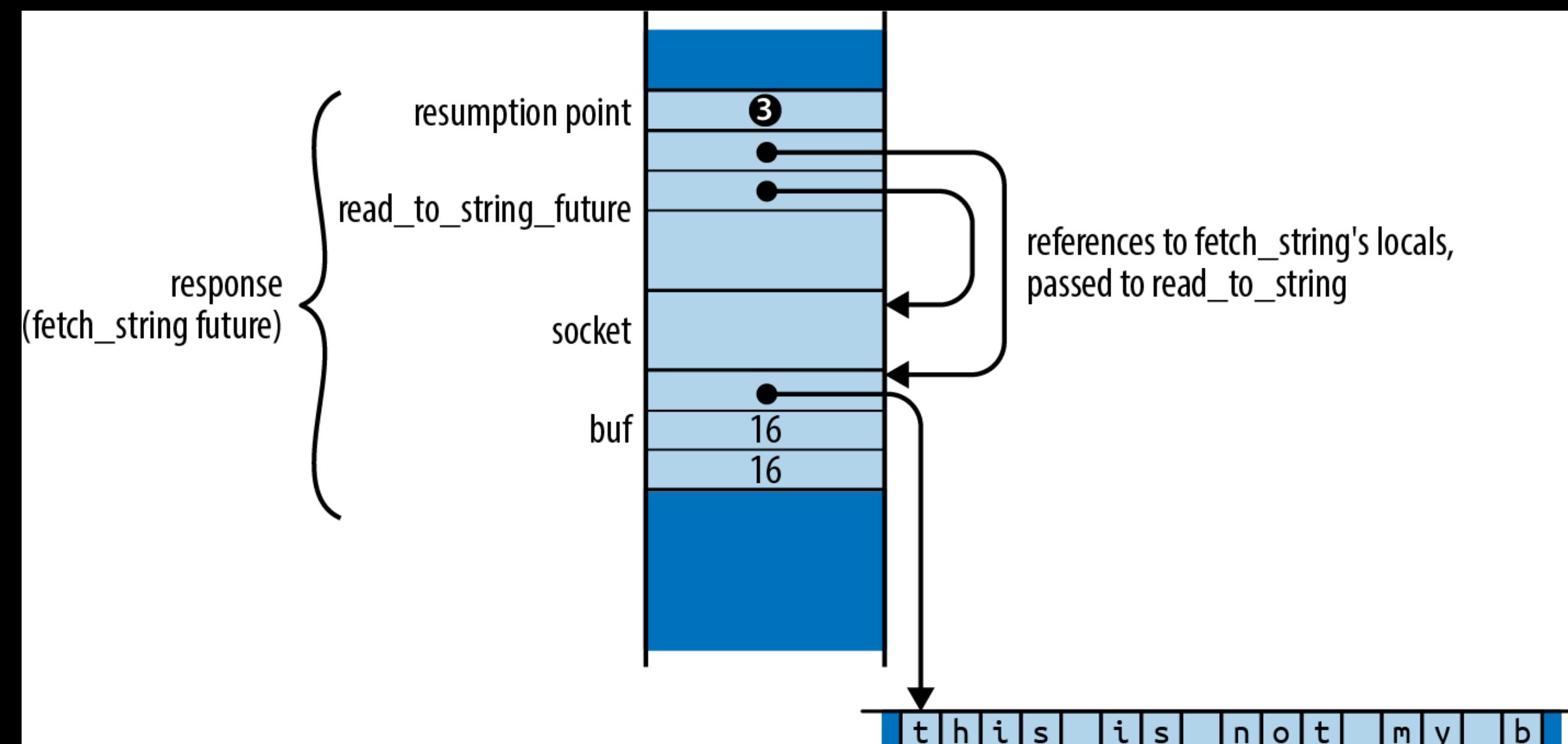
HSPACE Rust 특강
Rust Basic #17 - 비동기 프로그래밍, Part 2

- 이제 response를 몇 번 폴링해서 함수 본문의 아래 지점에 도달했다고 하자.



```
socket.read_to_string(&mut buf).await?;
```

- 그리고 read_to_string의 결과가 아직 준비되지 않아서 폴링이 Poll::Pending을 반환한다고 하자. 이 시점에서 퓨처의 모습은 아래의 메모리 구조를 갖는다.



퓨처의 두 가지 생애 단계

HSPACE Rust 특강
Rust Basic #17 - 비동기 프로그래밍, Part 2

- 퓨처는 다음에 폴링될 때 실행을 재개할 수 있도록 늘 필요한 모든 정보를 줘고 있어야 한다.
 - 재개 지점 [3]. 실행이 `read_to_string`의 퓨처를 폴링하는 `await`에서 재개되어야 함을 나타낸다.
 - 해당 재개 지점에 살아 있는 변수 `socket`과 `buf`.
`address`의 값은 함수에서 더 이상 쓰이지 않으므로 여기에 포함되지 않는다.
 - `await` 표현식이 폴링하고 있는 `read_to_string` 서브 퓨처.
- `read_to_string` 호출이 `socket`과 `buf`의 레퍼런스를 빌려왔다는 걸 눈여겨보자.
 - 동기 함수에서는 지역 변수가 전부 스택에 살지만, 비동기 함수에서는 `await` 이후에도 살아 있는 지역 변수가 퓨처 안에 위치해 있어야 다시 폴링될 때 쓰일 수 있다.
 - 이런 변수의 레퍼런스를 빌려 오면 퓨처의 일부를 빌려오게 된다.
 - 하지만 Rust에서는 빌려온 상태에 있는 값을 옮겨서는 안된다.

퓨처의 두 가지 생애 단계

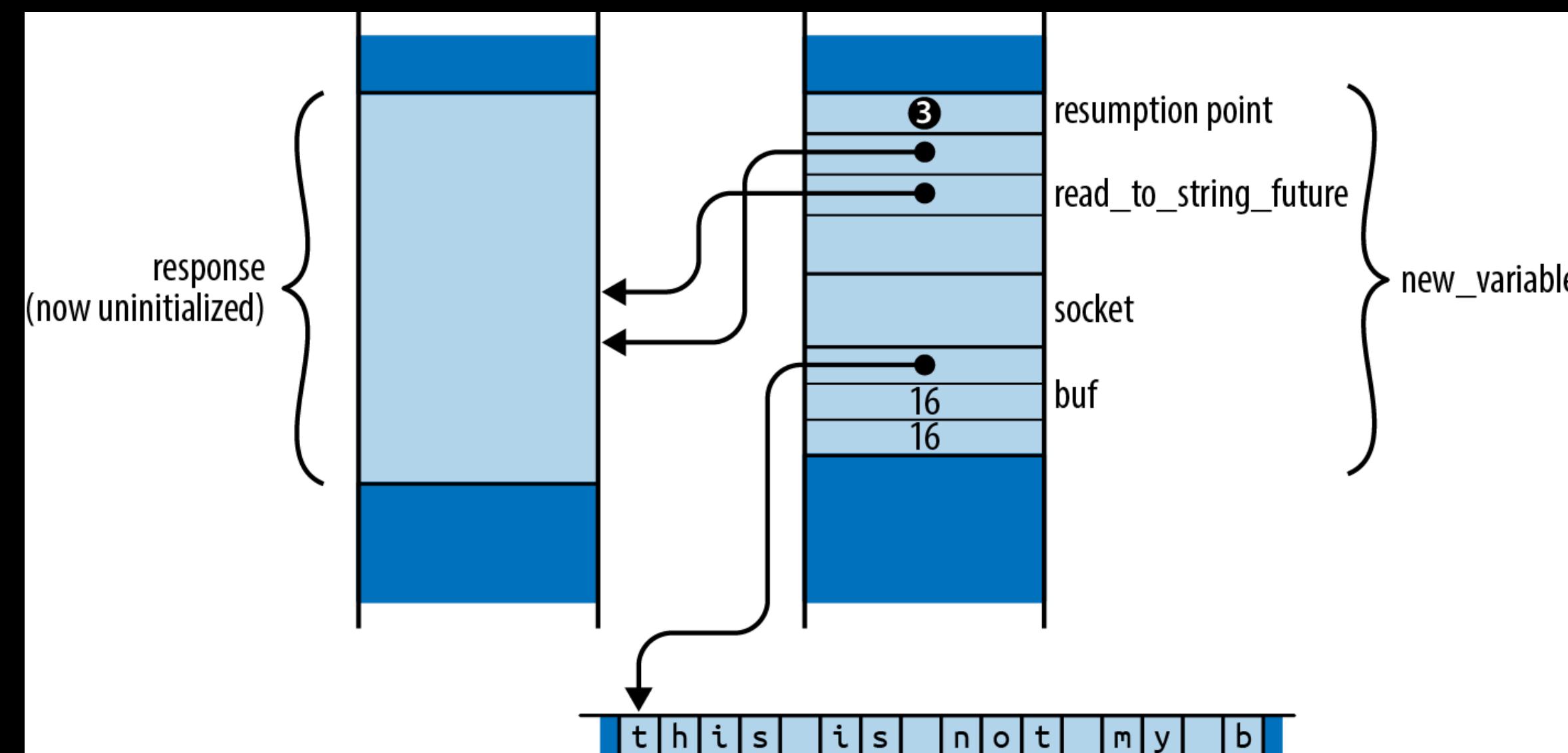
HSPACE Rust 특강
Rust Basic #17 - 비동기 프로그래밍, Part 2

- 이 퓨처를 새 위치로 옮긴다고 가정해 보자.



```
let new_variable = response;
```

- Rust에는 사용중인 레퍼런스를 전부 찾아다가 이를 적절하게 바로잡을 수단이 없다.
따라서 이들 레퍼런스는 새 위치에 있는 socket과 buf를 가리키는게 아니라 원래 위치에 있는
이제는 미초기화 상태가 되어 버린 response를 계속 가리킨다. 이들은 아래와 같이 대상을 읽은 포인터가 된다.



퓨처의 두 가지 생애 단계

HSPACE Rust 특강
Rust Basic #17 - 비동기 프로그래밍, Part 2

- 동작 설명
 - 빌려온 값이 이동되는 걸 방지하는 건 보통 빌림 검사기의 몫이다.
빌림 검사기는 변수를 소유 관계 트리의 뿌리로 여기지만,
스택에 저장된 변수와 달리 퓨처에 저장된 변수는 퓨처 자체가 이동되면 함께 이동된다.
이 말은 `socket`과 `buf`를 빌려 오는 일이 단지 `fetch_string`이 자기 변수를 가지고 할 수 있는 일의 종류 뿐만 아니라,
호출부가 이를 줘고 있는 퓨처인 `response`를 가지고 안전하게 할 수 있는 일의 종류에도 영향을 준다는 뜻이다.
`async` 함수의 퓨처는 빌림 검사기의 사각지대로,
Rust가 메모리 안전성에 관한 약속을 지켜내려면 어떤 식으로든 해결해야 한다.
 - 이 문제에 대한 Rust의 해결책은 퓨처가 처음 생성될 때는 언제든 이동해도 안전하며,
폴링될 때만 안전하지 않게 된다는 통찰력에 기초한다.
비동기 함수 호출을 통해서 방금 막 생성된 퓨처는 재개 지점과 인수 값만 들고 있다.
이들은 아직 실행되지 않은 비동기 함수의 본문 안에만 존재한다. 퓨처를 폴링해야만 이들의 내용을 빌려올 수 있다.

퓨처의 두 가지 생애 단계

HSPACE Rust 특강
Rust Basic #17 - 비동기 프로그래밍, Part 2

- 이를 통해서 모든 퓨처는 두 가지 생애 단계를 갖는다는 걸 알 수 있다.
 - 첫 번째 단계는 퓨처가 생성될 때 시작된다. 함수의 본문이 실행되기 전이므로 아직 빌려올 수 있는 부분이 없다. 이때는 임의의 다른 Rust 값으로 옮겨도 안전하다.
 - 두 번째 단계는 퓨처가 처음으로 폴링될 때 시작된다. 함수의 본문이 실행되고 나면 퓨처에 저장된 변수의 레퍼런스를 빌려올 수 있고 또 빌려온 채로 대기할 수 있다. 한번 폴링이 시작된 퓨처는 안전하게 이동할 수 없을지도 모른다고 봐야 한다.

퓨처의 두 가지 생애 단계

HSPACE Rust 특강
Rust Basic #17 - 비동기 프로그래밍, Part 2

- 이를 통해서 모든 퓨처는 두 가지 생애 단계를 갖는다는 걸 알 수 있다.
 - 퓨처를 `block_on`과 `spawn`에 넘기고 `race`와 `fuse` 같은 어댑터 메소드를 호출할 수 있는 건 모두 첫 번째 생애 단계의 유연성 덕분이다. 이들은 전부 퓨처를 값으로 받는다. 사실 맨 처음 퓨처를 생성한 비동기 함수 호출도 결과를 호출부에 반환해야 했는데, 이 역시 이동이었다.
 - 두 번째 생애 단계로 들어가기 위해서는 퓨처를 폴링해야 한다.
`poll` 메소드에는 퓨처를 `Pin<&mut Self>` 값으로 넘겨야 한다.
`Pin`은 (`&mut Self` 같은) 포인터 타입의 래퍼로 포인터의 용법에 제약을 둬서 (`Self` 같은) 참조 대상을 다시는 이동할 수 없도록 만든다. 따라서 퓨처를 폴링하면 먼저 `Pin`으로 감싼 포인터를 생성해야 한다.
 - 이것이 바로 Rust가 퓨처를 안전하게 지켜내기 위한 전략이다. 퓨처는 폴링하기 전에는 이동해도 위험하지 않다. 퓨처는 `Pin`으로 감싼 포인터를 생성하기 전에는 폴링할 수 없다. 그리고 그렇게 하고 난 퓨처는 이동할 수 없다.

핀이 설정된 포인터

HSPACE Rust 특강
Rust Basic #17 - 비동기 프로그래밍, Part 2

- Pin 타입
 - 퓨처 포인터의 래퍼로 포인터의 용법에 제약을 둬서 한 번 폴링된 퓨처를 이동될 수 없도록 만든다.
이 제약은 이동되어도 상관없는 퓨처의 경우에는 해제될 수 있지만
비동기 함수와 블록의 퓨처를 안전하게 폴링하는 데는 필수다.
 - 여기서 포인터란 말은 Deref와 옵션으로 DerefMut를 구현하고 있는 임의의 타입을 말한다.
Pin으로 감싼 포인터를 **핀이 설정된 포인터(Pinned Pointer)**라고 한다.
`Pin<&mut T>`와 `Pin<Box<T>>`가 대표적이다.

핀이 설정된 포인터

HSPACE Rust 특강
Rust Basic #17 - 비동기 프로그래밍, Part 2

- 표준 라이브러리에 있는 Pin의 정의는 다음처럼 단순하다.



```
● ● ●
pub struct Pin<P> {
    pointer: P,
}
```

- pointer 필드가 pub가 아니라는 점을 눈여겨보자.
이 말은 Pin을 생성하거나 사용하려면 이 타입이 제공하는 특별한 용도의 메소드를 사용할 수 밖에 없다는 뜻이다.

핀이 설정된 포인터

HSPACE Rust 특강
Rust Basic #17 - 비동기 프로그래밍, Part 2

- 비동기 함수나 블록의 퓨처가 주어질 때 이를 대상으로 핀이 설정된 포인터를 얻는 방법
 - futures-lite 크레이트의 pin! 매크로는 T 타입의 기존 변수를 Pin<&mut T> 타입의 새 변수로 가린다. 새 변수는 스택에 있는 익명의 임시 위치로 이동된 원본의 값을 가리킨다. 이 변수가 범위를 벗어나면 그 값은 드롭된다. 우리가 만든 block_on에서는 pin!을 써서 폴링하고 싶은 퓨처를 고정시켰다.
 - 표준 라이브러리의 Box::pin 생성자는 임의의 타입 T로 된 값의 소유권을 가져다가 힙으로 옮기고 Pin<Box<T>>를 반환한다.
 - Pin<Box<T>>는 From<Box<T>>를 구현하고 있으므로 Pin::from(boxed)는 boxed의 소유권을 가져다가 힙에 있는 같은 T를 가리키는 핀이 설정된 박스를 돌려준다.
 - 이들 퓨처의 핀이 설정된 포인터를 얻는 방법은 모두 퓨처의 소유권을 포기하는 과정을 수반하며, 이를 되돌릴 방법이 없다. 물론 핀이 설정된 포인터 자체는 원하는 대로 자유롭게 옮겨 다닐 수 있지만, 포인터가 이동한다고 해서 참조 대상이 같이 이동하는 건 아니다. 따라서 퓨처의 핀이 설정된 포인터를 소유하고 있다는 건 그 퓨처의 이동 능력을 영구적으로 포기했다는 증거가 된다.

핀이 설정된 포인터

HSPACE Rust 특강
Rust Basic #17 - 비동기 프로그래밍, Part 2

- 모든 `Pin<T>` 타입에는 포인터를 역참조해서 `poll`이 요구하는 `Pin<&mut T>`를 반환하는 `as_mut` 메소드가 있다.
 - 따라서 핀이 설정된 퓨처를 폴링하고 싶으면 이 메소드를 쓰면 된다.
 - `as_mut` 메소드는 소유권을 포기하지 않은 채로 폴링할 때도 도움이 될 수 있다.

우리가 만든 `block_on`에서는 이를 딱 그런 용도로 썼다.

```
pin!(future);

loop {
    match future.as_mut().poll(&mut context) {
        Poll::Ready(value) => return value,
        Poll::Pending => parker.park(),
    }
}
```

- **코드 설명**

- 여기서 `pin!` 매크로는 `future`를 `Pin<&mut F>`로 다시 선언하기 때문에 이를 `poll`에 바로 넘길 수 있다.
- 그러나 변경할 수 있는 레퍼런스는 `Copy`가 아니므로 `Pin<&mut F>`도 `Copy`가 될 수 없다.
이 말은 `future.poll()`을 바로 호출하면 `future`의 소유권을 빼앗겨서
반복문의 다음 반복에서는 변수가 미초기화 상태로 남게 된다는 뜻이다.
- 여기서는 이를 피하려고 `future.as_mut()`를 호출해서 반복문을 매번 반복할 때마다
새로운 `Pin<&mut F>`를 다시 빌려 온다.
- 핀이 설정된 퓨처의 `&mut` 레퍼런스를 얻는 방법은 없다.
그랬다가는 `std::mem::replace`나 `std::mem::swap`을 써서 기존 퓨처를 꺼내다가
그 자리에 다른 퓨처를 집어넣을 수 있었을 것이다.

핀이 설정된 포인터

HSPACE Rust 특강
Rust Basic #17 - 비동기 프로그래밍, Part 2

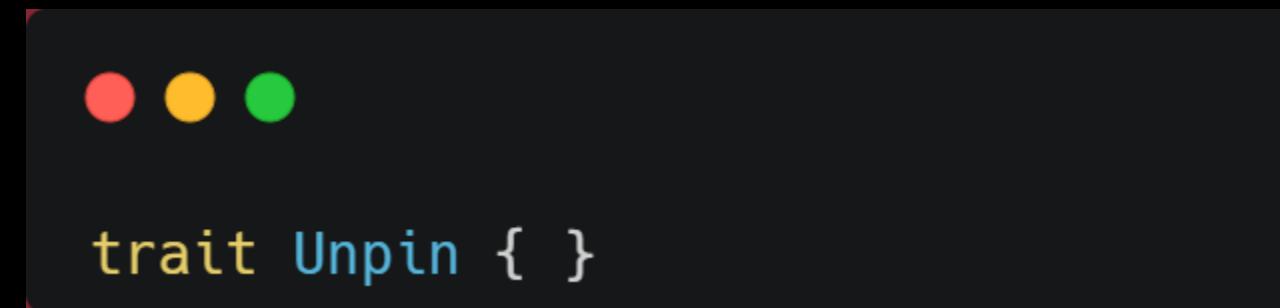
- **코드 설명**

- 평범한 비동기 코드에서 퓨처의 핀 설정을 두고 걱정할 필요가 없는 이유는 ('기다리기'나 '이그제큐터에게 넘기기'처럼) 퓨처의 값을 얻을 때 쓰는 대부분의 방법이 내부적으로 퓨처의 소유권을 가져다가 핀 설정을 알아서 관리해 주기 때문이다.
- 예로 우리가 만든 `block_on`은 퓨처의 소유권을 가져다가 `pin!` 매크로를 써서 폴링에 필요한 `Pin<&mut F>`를 만든다. `await` 표현식도 내부적으로 퓨처의 소유권을 가져다가 `pin!` 매크로와 비슷한 접근 방식을 쓴다.

Unpin 트레잇

HSPACE Rust 특강
Rust Basic #17 - 비동기 프로그래밍, Part 2

- 하지만 모든 퓨처를 이런 식으로 꼼꼼히 처리해야 하는 건 아니다.
 - 앞서 언급한 `SpawnBlocking` 타입처럼 평범한 타입을 위해 손으로 쓴 `Future`의 구현 같은 경우에는 핀이 설정된 포인터를 만들고 사용하는데 따르는 제약이 불필요하다.
- 이런 지속형 타입은 `Unpin` 마커 트레잇을 구현한다.



- Rust에 있는 대부분의 타입은 컴파일러의 특별한 지원을 써서 자동으로 `Unpin`을 구현하고 있다. 비동기 함수와 블록의 퓨처는 이 규칙에서 예외다.
- `Pin`은 `Unpin` 타입에 아무런 제약도 부과하지 않는다. `Pin::new`를 써서 평범한 포인터를 가지고 핀이 설정된 포인터를 만들 수 있고, `Pin::into_inner`를 써서 다시 포인터를 가져올 수 있다. 또한, `Pin` 자체가 포인터의 `Deref`와 `DerefMut` 구현을 제공한다.

Unpin 트레잇

HSPACE Rust 특강
Rust Basic #17 - 비동기 프로그래밍, Part 2

- 예를 들면, `String`은 `Unpin`을 구현하고 있으므로 다음처럼 작성할 수 있다.
 - 심지어 `Pin<&mut String>`을 만든 뒤에도 문자열의 변경할 수 있는 접근을 전부 사용할 수 있으며, `into_inner`가 `Pin`을 소비하는 바람에 변경할 수 있는 레퍼런스가 사라지더라도 문자열을 새 변수로 옮길 수 있다. 따라서 `Unpin` 타입의 경우(대부분의 타입)에는 `Pin`이 그 타입의 포인터를 감싸는 별 볼 일 없는 래퍼에 불과하다.
 - 이 말은 사용자 정의 `Unpin` 타입에 `Future`를 구현할 때 `poll` 구현이 `self`를 `Pin<&mut Self>`가 아니라 마치 `&mut Self`인 것처럼 다룰 수 있다는 뜻이다. 핀 설정과 관련된 건 대부분 무시해도 된다.

```
● ● ●

let mut string = "Pinned?".to_string();
let mut pinned: Pin<&mut String> = Pin::new(&mut string);

pinned.push_str(" Not");
Pin::into_inner(pinned).push_str(" so much.");

let new_home = string;
assert_eq!(new_home, "Pinned? Not so much.");
```

Unpin 트레잇

HSPACE Rust 특강
Rust Basic #17 - 비동기 프로그래밍, Part 2

- F가 Unpin을 구현하지 않더라도 Pin<&mut F>와 Pin<Box<F>>가 Unpin을 구현한다는 걸 알게 되면 놀랄 수도 있겠다.
 - Pin이 Unpin이 될 수 있다니 잘 이해되지 않지만 각각의 의미를 곰곰이 짚어 보면 말이 된다는 걸 알 수 있다. F는 한 번 폴링되고 나면 안전하게 이동할 수 없지만, F의 포인터는 폴링 여부와 상관없이 늘 안전하게 이동할 수 있다. 포인터만 이동하고 조심히 다뤄야 하는 참조 대상은 이동하지 않는다.
 - 이 부분은 비동기 함수나 블록의 퓨처를 Unpin 퓨처만 받는 함수에 넘기고 싶을 때 유용하다. F가 Unpin이 아니더라도 Pin<Box<F>>는 Unpin이므로 비동기 함수나 블록 퓨처에 Box::pin을 적용하면 힘 할당 비용은 들겠지만 어디서나 쓸 수 있는 퓨처를 얻을 수 있다.
 - Pin에는 대상 타입이 Unpin이 아니더라도 포인터와 그의 대상을 가지고 원하는 일을 할 수 있게 해주는 안전하지 않은 여러 메소드가 있다. 그러나 Rust는 이들 메소드가 올바르게 쓰이고 있는지 확인할 길이 없다. 이들 메소드를 쓰는 코드의 안전성을 담보하는 건 여러분의 몫이다.

비동기 코드는 언제 써야 좋을까

HSPACE Rust 특강
Rust Basic #17 - 비동기 프로그래밍, Part 2

- 비동기 코드는 멀티스레드 코드보다 작성하기 더 까다롭다.
 - 올바른 I/O와 동기화 기본 요소를 써야 한다.
 - 오래 걸리는 계산을 직접 쪼개거나 다른 스레드로 분리해야 한다.
 - 스레드를 쓰는 코드에서는 하지 않는 편 설정 같은 다른 세부 사항을 관리해야 한다.
- 그럼 비동기 코드가 가져다주는 구체적인 이점은 뭘까?

비동기 코드는 언제 써야 좋을까

HSPACE Rust 특강
Rust Basic #17 - 비동기 프로그래밍, Part 2

- 다음의 두 가지 주장을 자주 듣게 될 텐데 꼼꼼히 따져보면 꼭 그런 건 아니다.
 - '비동기 코드는 I/O에 적합하다.'
 - 이 말은 부정확하다. 애플리케이션이 I/O를 기다리는데 시간을 쓰고 있다면 이를 비동기로 만들어도 I/O가 더 빨라지지 않는다.
 - 요즘 쓰이는 비동기 I/O 인터페이스가 동기 버전을 더 효율적으로 만들어 주는 게 아니다. 운영체제가 하는 일은 어느 쪽이든 똑같다. (사실 준비 상태가 아닌 비동기 I/O 연산은 나중에 다시 시도해야 하므로 완료하는 데 필요한 시스템 호출은 1개가 아니라 2개다.)
 - '비동기 코드는 멀티스레드 코드보다 작성하기 더 쉽다.'
 - JavaScript와 Python 같은 언어에서는 맞는 말일 수 있다. 프로그래머가 `async/await`라는 잘 갖춰진 틀 안에서 동시성을 표현한다. 이 틀은 실행 스레드가 하나인데다 중단(Interruption)이 `await` 표현식에서만 일어나므로 데이터의 일관성을 지켜내기 위해서 뮤텍스를 써야할 일이 거의 없다. 그냥 데이터를 변경하는 도중에만 `await`를 쓰지 않으면 된다. 명시적인 허락이 있을 때만 태스크가 전환되면 코드를 이해하기 훨씬 쉬워진다.
 - 그러나 스레드 문제가 거의 없는 Rust에는 이 주장이 통하지 않는다. 프로그램이 컴파일되고 나면 데이터 경합이 생기지 않는다. 비결정적인 행동은 이를 다루도록 설계된 뮤텍스, 채널, 원자성 등과 같은 동기화 기능 안에 격리된다. 따라서 비동기 코드라고 해서 다른 스레드가 영향을 미칠만한 곳을 파악하는데 도움이 되는 특별한 이점이 있는 게 아니다. 모든 게 안전한 Rust 코드에서는 이 부분이 늘 명확하다.
 - 게다가 Rust의 비동기 지원은 스레드와 결합해서 쓸 때 그 진가를 발휘한다. 이를 포기하는 건 말도 안되는 일이다.

비동기 코드는 언제 써야 좋을까

HSPACE Rust 특강
Rust Basic #17 - 비동기 프로그래밍, Part 2

- 그렇다면 비동기 코드의 진정한 이점은 무엇일까?
 - 비동기 태스크는 메모리 사용량이 적다. Linux에서는 스레드의 메모리 사용량이 사용자와 커널 공간을 모두 합해 20KB가 넘는다. 퓨처의 메모리 사용량은 그보다 훨씬 적다. 우리가 만든 채팅 서버의 퓨처는 크기가 수백 바이트 정도이며, 이마저도 Rust 컴파일러가 개선되면서 더 작아지고 있다.
 - 비동기 태스크는 생성 속도가 더 빠르다. Linux에서는 스레드를 생성하는데 약 15us가 걸린다. 반면 비동기 태스크를 띄우는데 약 300ns가 걸리므로 약 50배 정도 차이가 난다.
 - 비동기 태스크는 운영체제 스레드보다 컨텍스트 스위치가 더 빠르다. Linux에서는 0.2us 대 1.7us 정도다. 하지만 이들 수치는 최고의 조건 아래서 얻은 결과다. 스위치가 I/O 준비로 인한 것이라면 두 비용 모두 1.7us로 뛴다. 스레드나 태스크 간의 스위치가 서로 다른 프로세서 코어에 일어났는지의 여부도 큰 차이를 만든다. 코어 간의 통신은 매우 느리다.

비동기 코드는 언제 써야 좋을까

HSPACE Rust 특강
Rust Basic #17 - 비동기 프로그래밍, Part 2

- 여기에 비동기 코드가 풀 수 있는 문제의 종류를 알 수 있는 힌트가 숨어 있다.
 - 예를 들면, 비동기 서버는 태스크당 메모리 사용량이 적으므로 더 많은 연결을 동시에 처리할 수 있다. 또한 설계를 독립된 여러 태스크가 서로 자연스럽게 소통하는 구조로 가져갈 때는 낮은 태스크당 비용, 짧은 생성 시간, 빠른 컨텍스트 스위치가 전부 중요한 이점으로 작용한다. 이런 이유 때문에 채팅 서버를 비동기 프로그래밍의 전형적인 예로 들지만, 멀티 플레이어 게임과 네트워크 라우터 역시 좋은 예가 될 수 있다.
 - 다른 상황에서는 비동기식 코드를 쓸 때 얻을 수 있는 이점이 불분명하다. 프로그램에 계산량이 많은 작업을 수행하거나 I/O가 끝나길 기다리며 앉아 있는 스레드 풀이 있을 때는 앞서 나열한 이점이 성능에 큰 영향을 미치지 않을 수도 있다. 이럴 때는 계산을 최적화하든지, 더 빠른 네트워크 연결을 찾든지, 실제로 제한 인자에 영향을 미치는 다른 무언가를 해야 한다.

감사합니다.

utilForever@gmail.com

<https://github.com/utilForever>

Facebook, Twitter: @utilForever