Konkuk University GDSC + EDGE 스터디

Week 11: Asynchronous Programming, Part 2

Chris Ohk

utilForever@gmail.com

목차

Send를 구현해야 하는 이유

- `spawn`에는 `spawn_local`에는 없는 제약이 하나 있다.
  - 퓨처가 여러 스레드를 오가며 실행되기 때문에 꼭 Send 마커 트레잇을 구현해야 한다는 점이다.
  - 퓨처는 쥐고 있는 값이 전부 Send일 때만 Send가 된다.
    함수 인수와 지역 변수는 물론 심지어 익명의 임시 값까지 전부 다른 스레드로 안전하게 이동될 수 있어야 한다.
  - 늘 그렇듯이 이 요구 사항은 비동기 태스크에만 주어지는 게 아니다. `std::thread::spawn`으로
    Send가 아닌 값을 캡처하는 클로저를 가진 스레드를 시작시키려 할 때도 비슷한 오류가 발생한다.
  - 차이점이라면 `std::thread::spawn`에 넘긴 클로저는 실행을 위해 생성된 스레드에 머무는 반면,
    스레드 풀에 생성된 퓨처는 대기할 때마다 한 스레드에서 다른 스레드로 옮겨갈 수 있다는 것이다.

Send를 구현해야 하는 이유

- 이 제약은 우연히 걸려 넘어지기 쉽다. 다음 코드는 아무런 문제가 없는 것처럼 보인다.

```rust
use async_std::task;
use std::rc::Rc;

async fn reluctant() -> String {
    let string = Rc::new("ref-counted string".to_string());

    some_asynchronous_thing().await;

    format!("Your splendid string: {string}")
}

task::spawn(reluctant());
```

Send를 구현해야 하는 이유

- 이 제약은 우연히 걸려 넘어지기 쉽다. 다음 코드는 아무런 문제가 없는 것처럼 보인다.
  - 비동기 함수의 퓨처는 함수가 `await` 표현식에서 다시 실행을 이어나가는 데 필요한 모든 정보를 쥐고 있어야 한다.
  - `reluctant`의 퓨처는 `await` 이후에 `string`을 써야 하므로 적어도 몇 차례 `Rc<String>` 값을 쥐게 되는데, `Rc` 포인터는 스레드 간에 안전하게 공유할 수 없으므로 퓨처 자체는 `Send`가 될 수 없다.
  - 그러나 `spawn`은 `Send`인 퓨처만 받으므로 Rust는 이 부분으로 인해 오류를 발생시킨다.

Send를 구현해야 하는 이유

- 오류 메시지

```
error: future cannot be sent between threads safely
    |
17  |     task::spawn(reluctant());
    |     ^^^^^^^^^^^ future returned by `reluctant` is not `Send`
    |


    |
127 | T: Future + Send + 'static,
    |             ---- required by this bound in `async_std::task::spawn`
    |
    = help: within `impl Future`, the trait `Send` is not implemented for `Rc<String>`
note: future is not `Send` as this value is used across an await
    |
10  |         let string = Rc::new("ref-counted string".to_string());
    |             ------ has type `Rc<String>` which is not `Send`
11  |
12  |         some_asynchronous_thing().await;
    |         ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
                await occurs here, with `string` maybe used later
...
15  |     }
    |     - `string` is later dropped here
```

Send를 구현해야 하는 이유

- 오류 메시지 분석
  - 길지만 유용한 세부 정보를 많이 담고 있다.
  - 퓨처가 `Send`여야 하는 이유를 설명한다. 이 부분은 `task::spawn`의 요구 사항이다.
  - 어떤 값이 `Send`가 아닌지를 설명한다. `Rc<String>` 타입의 지역 변수 `string`이 여기에 해당한다.
  - `string`이 퓨처에 영향을 주는 이유를 설명한다. 이는 `string`이 앞에 표시된 `await`의 범위 안에 있기 때문이다.

Send를 구현해야 하는 이유

- 해결 방법
  - 첫 번째는 Send가 아닌 값의 범위 안에 await 표현식이 들어가지 않도록 제한해서 그 값이 함수의 퓨처에 저장되지 않게끔 만드는 방법이다.

```rust
async fn reluctant() -> String {
    let return_value = {
        let string = Rc::new("ref-counted string".to_string());
        format!("Your splendid string: {string}")
        // The `Rc<String>` goes out of scope here...
    };

    // ... and thus is not around when we suspend here.
    some_asynchronous_thing().await;

    return_value
}
```

Send를 구현해야 하는 이유

- 해결 방법
  - 두 번째는 Rc 대신 그냥 std::sync::Arc를 쓰는 방법이다.
    Arc는 원자적인 업데이트를 써서 레퍼런스 카운트를 관리하므로 살짝 느리지만 Arc 포인터는 Send다.

```
use async_std::task;
use std::sync::Arc;

async fn reluctant() -> String {
    let string = Arc::new("ref-counted string".to_string());

    some_asynchronous_thing().await;

    format!("Your splendid string: {string}")
}

task::spawn(reluctant());
```

Send를 구현해야 하는 이유

- 해결 방법
  - 결국에는 Send가 아닌 타입을 알아보고 피하는 법을 배우게 되겠지만, 처음에는 좀 당황스러울 수 있다.
  - 예를 들어, 오래된 Rust 코드를 보다 보면 가끔 다음과 같은 제네릭 결과 타입을 쓰는 경우를 만날 때가 있다.

```rust
// Not recommended!
type GenericError = Box<dyn std::error::Error>;
type GenericResult<T> = Result<T, GenericError>;
```

  - GenericError 타입은 박스 처리된 트레잇 오브젝트를 써서
    std::error::Error를 구현하고 있는 임의의 타입으로 된 값을 쥔다.
    그러나 이 타입이 두고 있는 제약은 이게 다라서, 누군가 Error를 구현하고 있는 Send가 아닌 타입을 가졌다면,
    그 타입의 박스 처리된 값을 GenericError로 변환할 수 있다. 이런 가능성 때문에 GenericError는 Send가 아니다.

Send를 구현해야 하는 이유

- 따라서 다음 코드는 동작하지 않는다.

```rust
fn some_fallible_thing() -> GenericResult<i32> {
    ...
}

// This function's future is not `Send`...
async fn unfortunate() {
    // ... because this call's value ...
    match some_fallible_thing() {
        Err(error) => {
            report_error(error);
        }
        Ok(output) => {
            // ... is alive across this await ...
            use_output(output).await;
        }
    }
}

// ... and thus this `spawn` is an error.
async_std::task::spawn(unfortunate());
```

Send를 구현해야 하는 이유

- 컴파일 오류 원인 분석
  - 앞에서 본 예제와 마찬가지로 `Result` 타입을 범인으로 지목하며, 무슨 일이 벌어지고 있는지를 설명한다.
  - Rust는 `some_fallible_thing`의 결과가 `await` 표현식을 포함한 전체 `match` 문에 걸쳐서 존재한다고 생각하기 때문에 `unfortunate`의 퓨처를 `Send`가 아니라고 판단한다.
  - 사실 이 오류는 Rust가 지나치게 신중해서 생기는 문제다.
    `GenericError`를 다른 스레드에 안전하게 보낼 수 없는 건 사실이지만,
    `await`는 결과가 `Ok`일 때만 발생하므로 `use_output`의 퓨처를 기다릴 때는 오륫값이 존재할 수 없다.

Send를 구현해야 하는 이유

- 해결 방법
  - 좀 더 엄격한 제네릭 오류 타입을 쓰는 것이다.

```rust
type GenericError = Box<dyn std::error::Error + Send + Sync + 'static>;
type GenericResult<T> = Result<T, GenericError>;
```

  - 이 트레잇 오브젝트는 기본이 되는 오류 타입이 Send를 구현해야 한다고 명시적으로 요구하기 때문에 아무 문제가 없다.
  - 퓨처가 Send가 아니거나 Send로 바꾸기 어려울 때는 spawn_local을 써서 현재 스레드에서 실행되게 만들면 된다.
    물론 이 경우에는 스레드가 어떤 식으로든 block_on을 호출해야 실행될 기회가 주어지며,
    아쉽지만 작업을 여러 프로세서에서 분배하는 데서 오는 이점은 누릴 수 없다.

오래 걸리는 계산

- 오래 걸리는 계산의 문제점과 해결 방법
  - 퓨처가 자기 스레드를 다른 태스크와 잘 공유하기 위해서는 `poll` 메소드가 가능한 한 늘 빨리 복귀해야 한다.
  - 그러나 오래 걸리는 계산을 수행하고 있으면 다음 `await`에 닿을 때까지 오래 걸릴 수 있고,
    이로 인해서 다른 비동기 태스크가 자기 스레드 차례를 생각보다 오래 기다리게 된다.
  - 이를 피하는 한 가지 방법은 그냥 `await`를 드문드문 수행하는 것이다.
    `async_std::task::yield_now` 함수는 이런 용도를 설계된 간단한 퓨처를 반환한다.

```
while computation_not_done() {
    ... do one medium-sized step of computation ...
    async_std::task::yield_now().await;
}
```

오래 걸리는 계산

- 코드 설명
  - `yield_now`의 퓨처를 처음 폴링하면 `Poll::Pending`이 반환되지만 곧 다시 폴링해도 좋은 시점임을 알려 온다.
  - 이렇게 되면 이 비동기 호출은 스레드를 포기하고 다른 태스크가 실행될 기회를 거머쥐지만,
    곧 다시 원래 호출에게로 차례가 돌아온다.
  - 이때 `yield_now`의 퓨처를 다시 폴링하면 `Poll::Ready(())`가 반환되므로 `async` 함수가 실행을 재개할 수 있다.

오래 걸리는 계산

- 한계 및 해결 방법
  - 하지만 이 접근 방식이 항상 통하는 건 아니다. 외부 크레이트를 써서 오래 걸리는 계산을 수행하거나
    C나 C++를 호출하고 있는 경우에는 해당 코드를 async에 좀 더 적합한 형태로 바꾸기가 어려울 수 있다.
  - 아니면 계산 과정에 있는 모든 경로가 await를 드문드문 수행하도록 만드는 게 어려울 수 있다.
  - 이럴 때는 async_std::task::spawn_blocking을 쓰면 된다.
    이 함수는 클로저를 받아다가 자체 스레드에서 실행시키고 반환값의 퓨처를 반환한다.
    비동기 코드는 계산이 준비될 때까지 자기 스레드를 다른 태스크에 양보해 둔 채로 퓨처를 기다릴 수 있다.
    힘든 일을 별도의 스레드로 빼두면 운영체제가 알아서 프로세서를 잘 공유해 처리한다.

오래 걸리는 계산

- 예시
  - 사용자가 입력한 비밀번호를 인증 데이터베이스에 저장된 해싱된 버전과 비교해야 한다고 하자.
  - 보안을 위해서 비밀번호 검증 작업은 계산 집약적인 과정으로 되어 있어야,
    설령 공격자가 데이터베이스의 복사본을 손에 쥐더라도 엄청난 양의 비밀번호 후보를 일일이 대조해가며 찾을 수 없다.
  - `argonautica` 크레이트는 비밀번호를 저장하기 위한 용도로 특별히 설계된 해시 함수를 제공한다.
  - 적절히 생성된 `argonautica` 해시는 순식간에 검증할 수 있다.
    비동기 애플리케이션에서 `argonautica`를 쓰는 법은 다음과 같다.

오래 걸리는 계산

- 예제 코드
  - 이 함수는 `key`가 데이터베이스 전체에 쓰이는 키일 때 `password`가 `hash`와 일치하면 `Ok(true)`를 반환한다.
    이런 식으로 검증 작업을 클로저에 담아 `spawn_blocking`에 넘기면 비용이 많이 드는 계산이
    자체 스레드에서 실행되므로 다른 사용자 요청의 응답성을 해치지 않는다.

```rust
async fn verify_password(
    password: &str,
    hash: &str,
    key: &str,
) -> Result<bool, argonautica::Error> {
    // Make copies of the arguments, so the closure can be 'static.
    let password = password.to_string();
    let hash = hash.to_string();
    let key = key.to_string();

    async_std::task::spawn_blocking(move || {
        argonautica::Verifier::default()
            .with_hash(hash)
            .with_password(password)
            .with_secret_key(key)
            .verify()
    })
    .await
}
```

비동기 설계 전략

- 여러모로 볼 때 Rust가 비동기 프로그래밍을 바라보는 접근 방식은 다른 언어와 비슷하다.
  - 예를 들어, JavaScript, C#, Rust는 모두 비동기 함수에 `await` 표현식을 쓴다.
  - 그리고 이들 언어에는 모두 완료되지 않은 계산을 표현하는 값이 있다.
  - Rust는 "퓨처", JavaScript는 "프로미스", C#은 "태스크"라고 하지만 모두 기다려야 할 수도 있는 값을 표현한다.
- 하지만 Rust가 폴링을 쓰는 방식은 독특하다.
  - JavaScript와 C#에서는 비동기 함수가 호출되는 즉시 실행을 시작하며,
    기다리던 값이 준비되면 시스템 라이브러리에 내장된 전역 이벤트 루프가 중단된 `async` 함수 호출을 재개한다.
  - 그러나 Rust에서는 `async` 호출이 자신을 폴링해서 작업을 완료하도록 이끌어 줄
    `block_on`, `spawn`, `spawn_local` 같은 함수에 넘겨지기 전에는 아무 일도 하지 않는다.
  - **이그제큐터(Executor)**라고 하는 이들 함수는 다른 언어에서 전역 이벤트 루프가 담당하는 역할을 한다.

비동기 설계 전략

- Rust는 프로그래머가 퓨처를 폴링할 이그제큐터를 고를 수 있게 되어 있으므로 시스템에 내장된 전역 이벤트 루프가 필요 없다.
  - 지금까지 사용한 이그제큐터 함수는 `async-std` 크레이트가 제공하는 것이었지만, 뒤에서 사용할 `tokio` 크레이트도 자체적으로 일련의 유사한 이그제큐터 함수를 정의해 두고 있다.
  - 후반부에는 나만의 이그제큐터를 구현해 본다. 이 세 가지를 전부 같은 프로그램에서 쓸 수 있다.

진짜 비동기 HTTP 클라이언트

- 제대로 된 비동기 HTTP 클라이언트 코드를 살펴 보자.
  - `reqwest`와 `surf`를 포함해서 훌륭한 크레이트가 여럿 있으니 그 중 하나를 골라 사용하면 된다.
  - 여기서는 `async-std`와 `surf`를 사용한다.

진짜 비동기 HTTP 클라이언트

- 제대로 된 비동기 HTTP 클라이언트 코드를 살펴 보자.

```rust
pub async fn many_requests(urls: &[String]) -> Vec<Result<String, surf::Exception>> {
    let client = surf::Client::new();

    let mut handles = vec![];
    for url in urls {
        let request = client.get(&url).recv_string();
        handles.push(async_std::task::spawn(request));
    }

    let mut results = vec![];
    for handle in handles {
        results.push(handle.await);
    }

    results
}
```

진짜 비동기 HTTP 클라이언트

- 제대로 된 비동기 HTTP 클라이언트 코드를 살펴 보자.

```rust
fn main() {
    let requests = &[
        "http://example.com".to_string(),
        "https://www.red-bean.com".to_string(),
        "https://en.wikipedia.org/wiki/Main_Page".to_string(),
    ];

    let results = async_std::task::block_on(many_requests(requests));
    for result in results {
        match result {
            Ok(response) => println!("*** {}\n", response),
            Err(err) => eprintln!("error: {}\n", err),
        }
    }
}
```

비동기식 클라이언트와 서버

- 지금까지 다룬 핵심 아이디어를 한데 모아서 채팅 서버와 클라이언트를 만들어 보자.
  - 먼저 `cargo new --lib async-chat` 명령으로 프로젝트를 만든 후, `Cargo.toml`에 다음과 같이 입력한다.

```toml
[package]
name = "async-chat"
version = "0.1.0"
authors = ["Chris Ohk <utilforever@gmail.com>"]
edition = "2021"

[dependencies]
async-std = { version = "1.7", features = ["unstable"] }
tokio = { version = "1.0", features = ["sync"] }
serde = { version = "1.0", features = ["derive", "rc"] }
serde_json = "1.0"
```

비동기식 클라이언트와 서버

- 지금까지 다룬 핵심 아이디어를 한데 모아서 채팅 서버와 클라이언트를 만들어 보자.
  - `async-std` : 비동기 I/O 기본 요소와 유틸리티가 있는 크레이트
  - `tokio` : `async-std`처럼 비동기 기본 요소가 있는 크레이트. 나온 지 오래되어서 제공하는 기능도 많고 안정적이다.
  - `serde`, `serde_json` : 프로그램에서 채팅 프로토콜은 네트워크를 통해서 주고 받는 데이터를 JSON으로 표현하는데, 두 크레이트는 이 JSON을 생성하고 파싱하는 편리하고 효율적인 도구를 제공한다.

비동기식 클라이언트와 서버

- 지금까지 다룬 핵심 아이디어를 한데 모아서 채팅 서버와 클라이언트를 만들어 보자.
  - 채팅 애플리케이션의 전체 구조

```
async-chat
├── Cargo.toml
└── src
    ├── lib.rs
    ├── utils.rs
    └── bin
        ├── client.rs
        └── server
            ├── main.rs
            ├── connection.rs
            ├── group.rs
            └── group_table.rs
```

Error와 Result 타입

- utils 모듈은 애플리케이션 전반에 걸쳐서 사용할 결과와 오류 타입을 정의한다.
  - Send와 Sync 바운드는 다른 스레드에 생성된 태스크가 실패할 때 오류를 메인 스레드로 안전하게 보고할 수 있게 만든다.
  - 실제 애플리케이션에서는 이와 비슷한 Error와 Result 타입을 제공하는 anyhow 크레이트를 쓰는 게 좋다.

```rust
use std::error::Error;

pub type ChatError = Box<dyn Error + Send + Sync + 'static>;
pub type ChatResult<T> = Result<T, ChatError>;
```

프로토콜

- 라이브러리 크레이트는 전체 채팅 프로토콜을 두 가지 타입으로 담아낸다.

```rust
use serde::{Deserialize, Serialize};
use std::sync::Arc;

pub mod utils;

#[derive(Debug, Serialize, Deserialize, PartialEq)]
pub enum FromClient {
    Join {
        group_name: Arc<String>,
    },
    Post {
        group_name: Arc<String>,
        message: Arc<String>,
    },
}

#[derive(Debug, Serialize, Deserialize, PartialEq)]
pub enum FromServer {
    Message {
        group_name: Arc<String>,
        message: Arc<String>,
    },
    Error(String),
}
```

프로토콜

- 코드 설명
  - 열거체 `FromClient`는 클라이언트가 서버에 보낼 수 있는 패킷을 표현한다.
    이를 통해서 그룹 가입을 요청하고 가입된 임의의 그룹에 메시지를 보낼 수 있다.
  - 열거체 `FromServer`는 서버가 되돌려 보낼 수 있는 것, 즉 일부 그룹에 보낸 메시지와 오류 메시지를 포함한다.
    평범한 `String` 대신 레퍼런스 카운트를 쓰는 `Arc<String>`을 쓰기 때문에
    서버가 그룹을 관리하고 메시지를 전달할 때 문자열의 복사본을 만들 필요가 없다.
  - `#[derive]`는 `serde` 크레이트에게 `FromClient`와 `FromServer`를 위한 `Serialize`와 `Deserialize` 트레잇의 구현을 생성해 달라고 이야기한다. 그 덕분에 보내는 쪽에서는 `serde_json::to_string`을 호출해서 이를 JSON 값으로 바꿀 수 있고, 받는 쪽에서는 `serde_json::from_str`를 호출해서 이를 다시 Rust 타입으로 바꿀 수 있다.

사용자 입력 받기

- 채팅 클라이언트의 첫 임무는 사용자의 명령을 읽고 대응하는 패킷을 서버에 보내는 것이다.

```rust
use async_chat::utils::{self, ChatResult};
use async_std::io;
use async_std::net;
use async_std::prelude::*;

async fn send_commands(mut to_server: net::TcpStream) -> ChatResult<()> {
    println!(
        "Commands:\n\
        join GROUP\n\
        post GROUP MESSAGE...\n\
        Type Control-D (on Unix) or Control-Z (Windows) to close the connection."
    );

    let mut command_lines = io::BufReader::new(io::stdin()).lines();

    while let Some(command_result) = command_lines.next().await {
        let command = command_result?;
        let request = match parse_command(&command) {
            Some(request) => request,
            None => continue,
        };

        utils::send_as_json(&mut to_server, &request).await?;
        to_server.flush().await?;
    }

    Ok(())
}
```

사용자 입력 받기

- 코드 설명
  - `async_std::io::stdin`을 호출해서 클라이언트의 표준 입력에 대한 비동기 핸들을 가져다가 버퍼링을 위해서 `async_std::io::BufReader`로 감싼 뒤에 `lines`를 호출해서 사용자의 입력을 한 줄씩 처리한다.
  - 이 과정에서 각 줄을 `FromClient` 값에 대응하는 명령으로 파싱해 보고 성공하면 그 값을 서버에 보낸다. 사용자가 인식할 수 없는 명령을 입력하면 `parse_command`가 오류 메시지를 출력한 뒤 `None`을 반환하므로, `send_commands`가 다시 반복문을 실행할 수 있다.
  - 사용자가 파일의 끝에 해당하는 문자를 입력하면 `lines` 스트림은 `None`을 반환하고 `send_commands`는 복귀한다.

패킷 보내기

- 클라이언트와 서버는 패킷을 네트워크 소켓에 전송하기 위해서
  라이브러리 크레이트의 `utils` 모듈에 있는 `send_as_json` 함수를 쓴다.

```rust
use async_std::prelude::*;
use serde::Serialize;
use std::marker::Unpin;

pub async fn send_as_json<S, P>(outbound: &mut S, packet: &P) -> ChatResult<()>
where
    S: async_std::io::Write + Unpin,
    P: Serialize,
{
    let mut json = serde_json::to_string(&packet)?;
    json.push('\n');

    outbound.write_all(json.as_bytes()).await?;

    Ok(())
}
```

패킷 보내기

- 코드 설명
  - `packet`의 JSON 표현을 `String`으로 만들고, 끝에 새 줄을 넣어서 전부 `outbound`에 기록한다.
  - `where` 절을 보면 `send_as_json`이 상당히 유연하다는 걸 알 수 있다.
    보낼 패킷의 타입 `P`는 `serde::Serialize`를 구현하고 있는 것이라면 무엇이든 될 수 있다.
    출력 스트림 `S`는 출력 스트림을 위한 `std::io::Write` 트레잇의 비동기 버전인 `async_std::io::Write`를 구현하고 있는 것이라면 무엇이든 될 수 있다.
  - 이 조건이라면 `FromClient`와 `FromServer` 값을 비동기 `TcpStream`에 보내기에 충분하다.
    `send_as_json`의 정의를 제네릭으로 가져가면 놀랍게도 스트림이나 패킷 타입의 세부 사항에 의존하지 않게 된다.
    `send_as_json`은 이들 트레잇이 가진 메소드만 쓸 수 있다.

패킷 보내기

- 코드 설명
  - S에 붙은 `Unpin` 제약 조건은 `write_all` 메소드를 쓰는 데 필요하다.
  - 마지막에 패킷을 `outbound` 스트림에 바로 직렬화하지 않고 임시 `String`에 직렬화한 다음, `outbound`에 기록한다. `serde_json` 크레이트는 값을 출력 스트림에 바로 직렬화하는 함수를 제공하지만, 이들 함수는 동기 스트림만 지원한다. 비동기 스트림에 기록하기 위해서는 `serde_json`과 `serde` 크레이트 양쪽 모두가 가진 타입 의존성이 없는 코어 부분을 많이 바꿔야 하는데, 왜냐하면 여기에 관여된 트레잇이 동기 메소드를 중심으로 설계됐기 때문이다.

패킷 받기

- 서버와 클라이언트는 패킷을 받기 위해서 `utils` 모듈의 `receive_as_json` 함수를 쓴다.

```rust
use serde::de::DeserializeOwned;

pub fn receive_as_json<S, P>(inbound: S) -> impl Stream<Item = ChatResult<P>>
where
    S: async_std::io::BufRead + Unpin,
    P: DeserializeOwned,
{
    inbound.lines().map(|line_result| -> ChatResult<P> {
        let line = line_result?;
        let parsed = serde_json::from_str::<P>(&line)?;

        Ok(parsed)
    })
}
```

패킷 받기

- 코드 설명
  - 이 함수는 버퍼링되는 비동기 TCP 소켓 `async_std::io::BufReader<TcpStream>`에서 `FromClient`와 `FromServer` 값을 받는다.
  - 스트림 타입 `S`는 버퍼링되는 입력 바이트 스트림을 표현하는 `std::io::BufRead`의 비동기 버전인 `async_std::io::BufRead`를 구현하고 있어야 한다.
  - 패킷 타입 `P`는 `serde`가 가진 `Deserialize` 트레잇의 좀 더 엄격한 버전인 `DeserializeOwned`를 구현하고 있어야 한다. 효율성을 위해서 `Deserialize`는 역직렬화된 버퍼에서 직접 내용을 빌려다가 `&str`와 `&[u8]` 값을 산출하는 식으로 데이터 복사를 피할 수 있다. 하지만 여기서는 그렇게 해봐야 좋을 게 없는데, 역직렬화한 값을 호출부에 반환해야 해서 값이 자기가 파싱되어 나온 버퍼보다 더 오래 살 수 있어야 하기 때문이다. `DeserializeOwned`를 구현하고 있는 타입은 항상 자기가 역직렬화되어 나온 버퍼와 독립적이다.

패킷 받기

- 함수 분석
  - `receive_as_json` 자체는 비동기 함수가 아니라는 점을 눈여겨보자.
    이 함수는 `async` 값인 스트림을 반환하는 평범한 함수다.
  - Rust의 비동기 지원을 '그냥 모든 곳에 `async`와 `.await`를 붙이기만 하면 되는 것'이라는 식의 인식에서 벗어나 보다 깊이 이해하면, 이처럼 언어를 최대한 활용하는 깔끔하고 유연하면서도 효율적인 정의가 나올 가능성이 열린다.

패킷 받기

- `receive_as_json`을 어떻게 쓰는지 알아 보기 위해서 `handle_replies` 함수를 보자.

```rust
use async_chat::FromServer;

async fn handle_replies(from_server: net::TcpStream) -> ChatResult<()> {
    let buffered = io::BufReader::new(from_server);
    let mut reply_stream = utils::receive_as_json(buffered);

    while let Some(reply) = reply_stream.next().await {
        match reply? {
            FromServer::Message {
                group_name,
                message,
            } => {
                println!("message posted to {group_name}: {message}");
            }
            FromServer::Error(message) => {
                println!("error from server: {message}");
            }
        }
    }

    Ok(())
}
```

패킷 받기

- 코드 설명
  - 서버에서 데이터를 받는 소켓을 가져다가 (`async_std` 버전의) `BufReader`로 감싼 뒤에 `receive_as_json`에 넘겨서 들어오는 `FromServer` 값의 스트림을 얻는다.
  - 그런 다음 `while let` 반복문을 써서 들어오는 응답을 처리하고, 오류 결과를 검사하고 각 서버 응답을 사용자가 볼 수 있게 출력한다.

클라이언트의 메인 함수

- 이제 채팅 클라이언트의 메인 함수를 보자.

```rust
use async_std::task;

fn main() -> ChatResult<()> {
    let address = std::env::args().nth(1).expect("Usage: client ADDRESS:PORT");

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

        let to_server = send_commands(socket.clone());
        let from_server = handle_replies(socket);

        from_server.race(to_server).await?;

        Ok(())
    })
}
```

클라이언트의 메인 함수

- 코드 설명
  - 명령줄에서 서버의 주소를 가져온 뒤에 일련의 비동기 함수를 호출해야 하므로, 함수의 나머지 부분을 비동기 블록으로 감싼 다음 이 블록의 퓨처를 `async_std::task::block_on`에 넘겨서 실행한다.
  - 연결을 설정하고 난 뒤에 `send_commands`와 `handle_replies` 함수를 나란히 실행해서 타이핑하고 있는 동안에 도착하는 다른 이의 메시지를 볼 수 있게 만든다.
    파일의 끝을 나타내는 문자를 입력하거나 서버와의 연결이 끊어지면 프로그램은 종료되어야 한다.

클라이언트의 메인 함수

- 코드 설명
  - 하지만 그동안 배웠던 걸 고려할 때 어쩌면 다음과 같은 코드를 기대했을 수도 있겠다.

```rust
let to_server = task::spawn(send_commands(socket.clone()));
let from_server = task::spawn(handle_replies(socket));

to_server.await?;
from_server.await?;
```

  - 이전 코드는 두 조인 핸들을 모두 기다리기 때문에 두 태스크가 **모두** 끝나야 프로그램이 종료된다.
  - 여기서는 **둘 중 하나**가 끝나면 그 즉시 종료되게 만들고 싶다. 이럴 때 쓰라고 있는 게 바로 퓨처의 `race` 메소드다.
    `from_server.race(to_server)` 호출은 `from_server`가 `to_server`를 모두 폴링하는 새 퓨처를 반환하고,
    둘 중 하나가 준비되면 그 즉시 `Poll::Ready(v)`를 반환한다.
  - 이때 두 퓨처의 출력 타입은 반드시 같아야 하며,
    먼저 끝난 퓨처의 값이 최종 값이 된다. 아직 끝나지 않은 퓨처는 드롭된다.

참고 자료

- https://www.rust-lang.org/
- https://doc.rust-lang.org/book/
- https://tourofrust.com/
- The Rust Programming Language (No Starch Press, 2019)
- Programming Rust, 2nd Edition (O’Reilly, 2021)

Thank you!