

HSPACE Rust 특강

Rust Basic #16 - 비동기 프로그래밍, Part 1

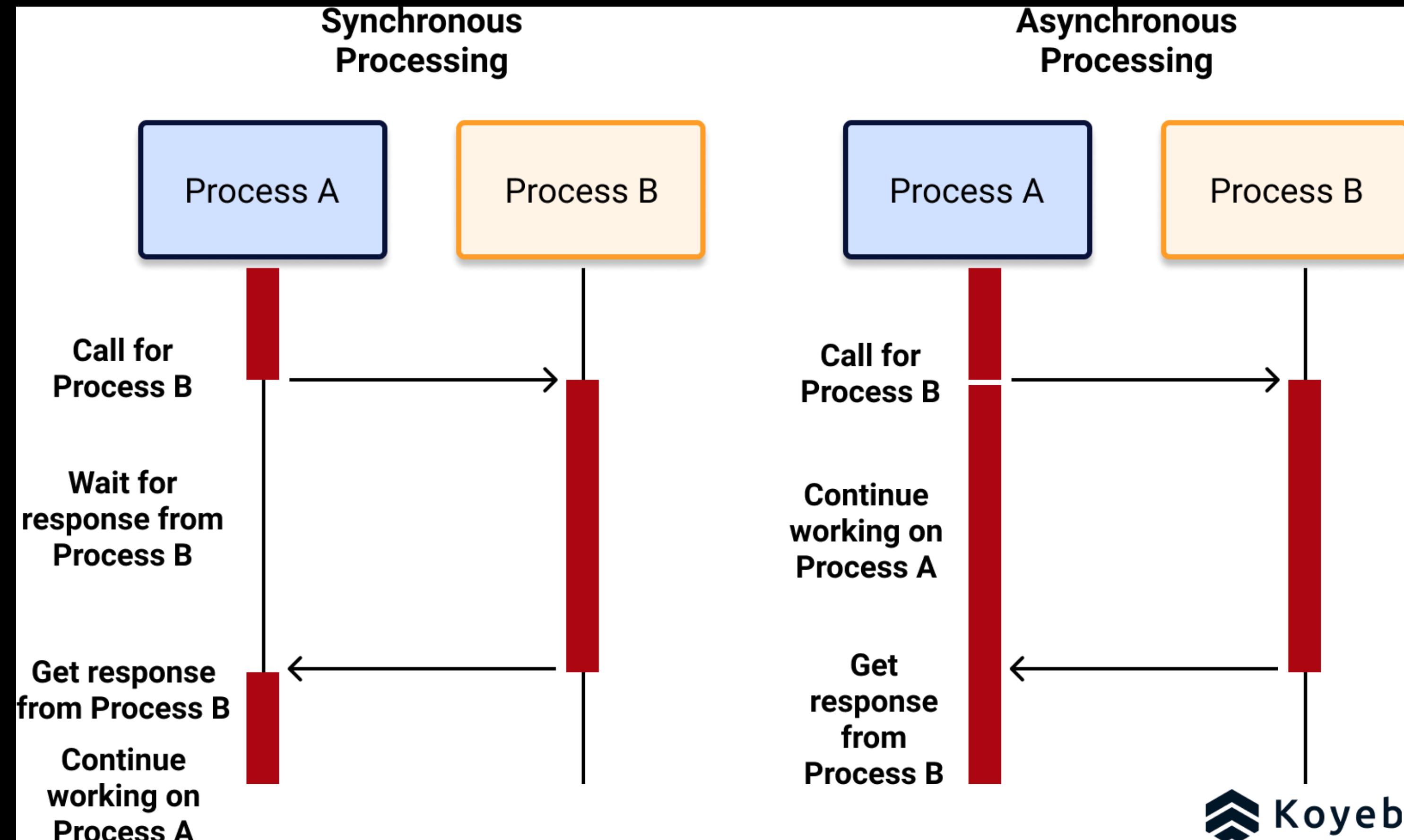
Chris Ohk

utilForever@gmail.com

- 동기 → 비동기
 - 퓨처 (Future)
 - async 함수와 await 표현식
 - block_on : 동기 코드에서 async 함수 호출하기
 - 비동기 태스크 생성하기
 - async 블록
 - async 블록으로 async 함수 만들기
 - async 태스크를 스레드 풀에서 실행하기
 - 퓨처가 Send를 구현해야 하는 이유
 - yield_now와 spawn_blocking : 오래 걸리는 계산
 - 비교하며 알아보는 비동기 설계 전략
 - 진짜 비동기 HTTP 클라이언트

- 비동기식 클라이언트와 서버
 - Error와 Result 타입
 - 프로토콜
 - 사용자 입력 받기 : 비동기 스트림
 - 패킷 보내기
 - 패킷 받기 : 또 다른 비동기 스트림
 - 클라이언트의 메인 함수

- Synchronous vs Asynchronous



- 채팅 서버를 만들고 있다고 생각해 보자.
 - 네트워크 연결마다 들어오는 패킷을 파싱해야 한다.
 - 나가는 패킷을 조립해야 한다.
 - 보안 매개 변수를 관리해야 한다.
 - 채팅 그룹 구독을 추적해야 한다.
- 연결이 많을 때 이 모든 일들을 동시에 관리하려면 구조화가 필요하다.

어떻게 구조를 만들어야 할까?

- 간단하게 들어오는 연결마다 스레드를 하나씩 시작하도록 만드는 방법이 있다.
 - 이 코드는 들어오는 연결마다 `serve` 함수를 실행하는 새 스레드를 생성한다.
 - 이렇게 하면 연결에 필요한 요구 사항을 관리하는 데만 집중할 수 있다는 장점이 있다.

```
use std::{net, thread};

let listener = net::TcpListener::bind(address)?;

for socket_result in listener.incoming() {
    let socket = socket_result?;
    let groups = chat_group_table.clone();

    thread::spawn(|| {
        log_error(serve(socket, groups));
    });
}
```

- 하지만 바이럴이 되어 갑자기 수만 명의 사용자가 몰리면 문제가 터지기 시작한다.
 - 스레드 스택의 크기가 100KB를 넘기는 건 흔한 일이라 수 GB나 되는 서버 메모리가 금새 바닥나고 말 것이다.
 - 스레드는 유용하고 작업을 여러 프로세서에 분배하는 데 필수지만,
이런 메모리 문제 때문에 작업을 나누려면 스레드와 함께 쓸 상호 보완적인 방법이 필요할 때가 많다.

- Rust의 비동기 태스크(Asynchronous Task)를 쓰면
독립된 여러 활동을 하나의 스레드나 워커 스레드 풀에 교차로 배치할 수 있다.
 - 생성 속도가 훨씬 빠르다.
 - 제어를 주고받기에 더 효율적이다.
 - 메모리 오버헤드가 훨씬 적다.
- 따라서 한 프로그램이 동시에 수십만 개의 비동기 태스크를 실행할 수 있다.
 - 물론 네트워크 대역폭, DB 속도, 계산, 작업 고유의 메모리 요구 사항 같은 다른 요인에 의해서 제한될 가능성이 있다.
 - 하지만 태스크를 사용해서 발생하는 메모리 오버헤드는 스레드에 비해서 훨씬 덜 중요하다.

- Rust의 비동기 코드

- 평범한 멀티 스레드 코드와 매우 비슷해 보인다.
- 하지만 I/O나 뮤텍스 획득 같은 블록될 수도 있는 작업의 처리 방식을 약간 달리 가져가야 한다는 점이 다르다.

```
● ● ●

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);
    });
}
```

동기 → 비동기

HSPACE Rust 특강
Rust Basic #16 - 비동기 프로그래밍, Part 1

- 다음 함수를 호출할 때 어떤 일이 벌어질 지 생각해 보자.
 - 주어진 웹 서버 정보를 가지고 TCP 연결을 열어서 오래된 프로토콜로 형식만 겨우 갖추는 HTTP 요청을 보내고 응답을 읽는다.

```
use std::io::prelude::*;
use std::net;

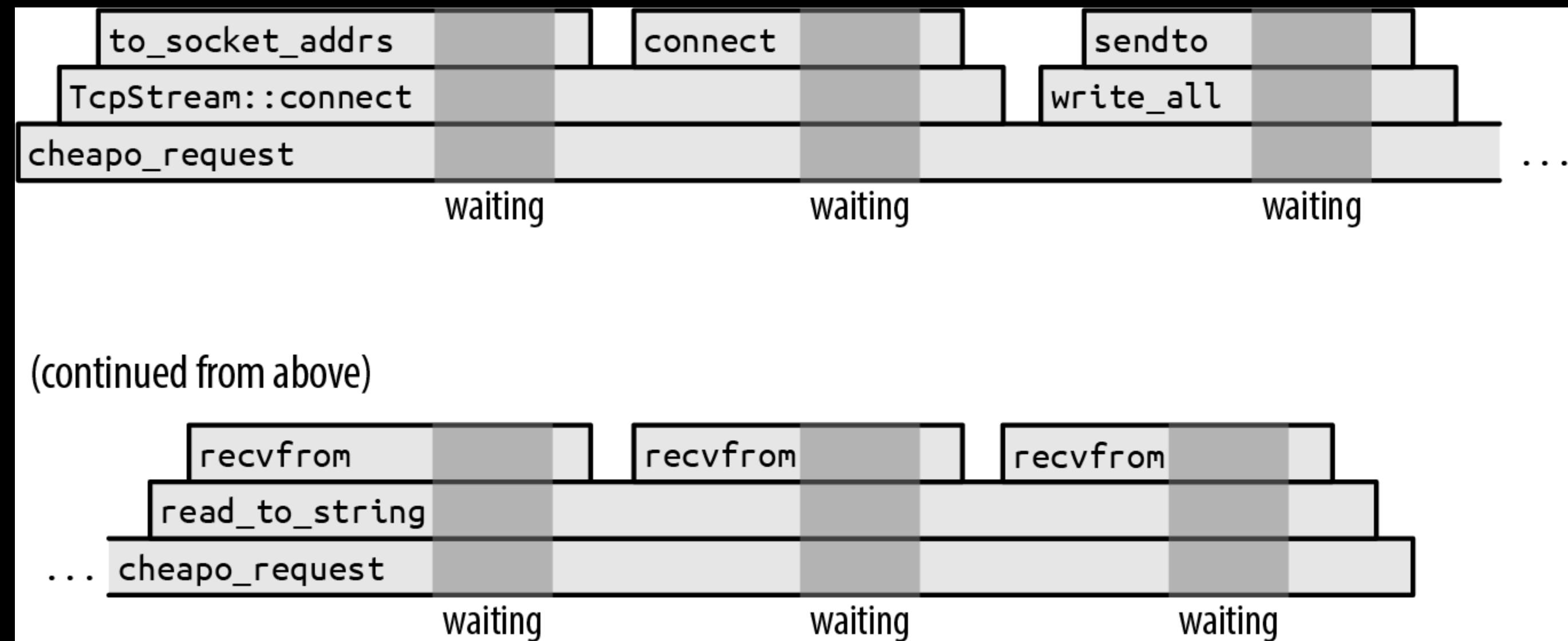
fn cheapo_request(host: &str, port: u16, path: &str) -> std::io::Result<String> {
    let mut socket = net::TcpStream::connect((host, port))?;
    let request = format!("GET {} HTTP/1.1\r\nHost: {}\r\n{}\r\n", path, host, "\r\n");
    socket.write_all(request.as_bytes())?;
    socket.shutdown(net::Shutdown::Write)?;

    let mut response = String::new();
    socket.read_to_string(&mut response)?;
    Ok(response)
}
```

동기 → 비동기

HSPACE Rust 특강
Rust Basic #16 - 비동기 프로그래밍, Part 1

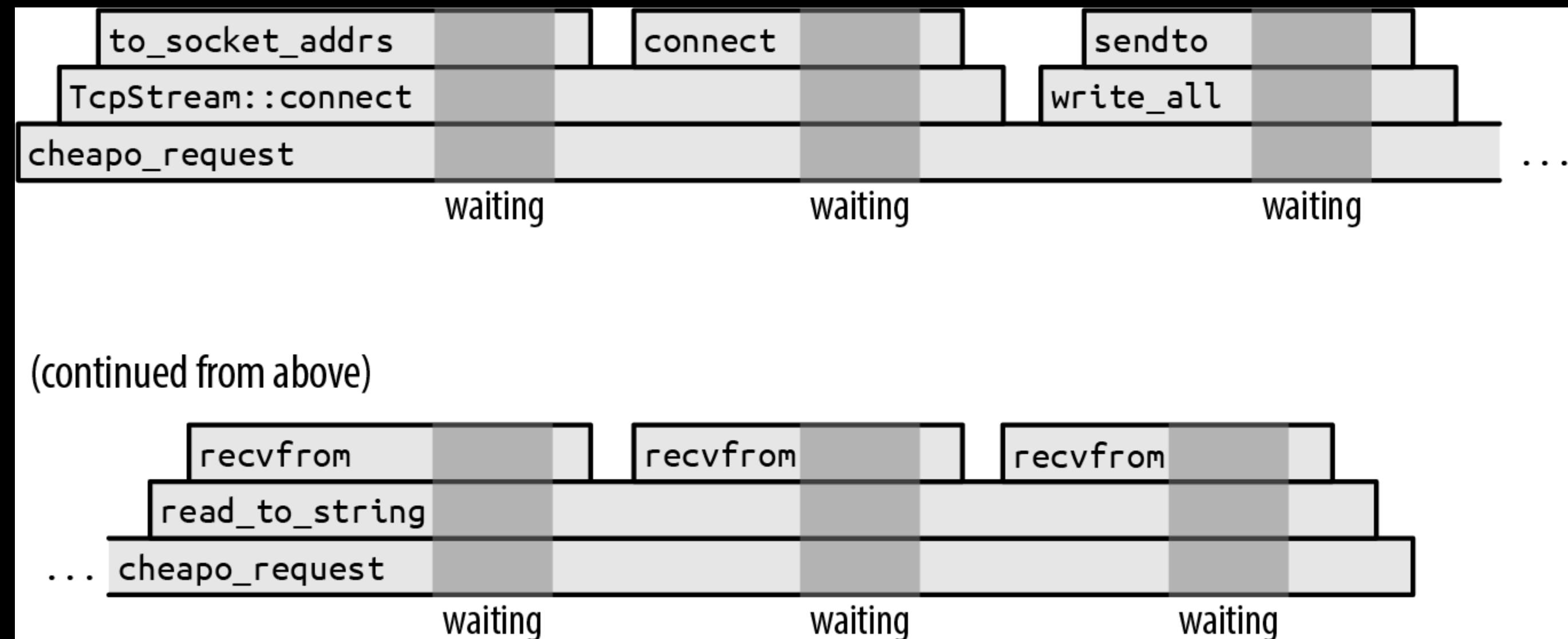
- 이 함수의 실행을 시간의 흐름에 따라 다음과 같이 처리된다.
 - `cheapo_request` 함수는 전체 실행 과정을 점유한다. 그리고 그 과정에서 `TcpStream::connect`와 `TcpStream`에 구현된 `write_all`과 `read_to_string` 같은 Rust 표준 라이브러리에 있는 함수를 호출한다.
 - 그렇게 꼬리에 꼬리를 문 함수 호출을 따라가다 보면 결국 프로그램이 **시스템 호출(System Call)**을 써서 운영체제에게 TCP 연결을 열거나 데이터를 읽고 쓰는 등의 작업을 실제로 처리해 달라고 요청하는 부분이 나온다.



동기 → 비동기

HSPACE Rust 특강
Rust Basic #16 - 비동기 프로그래밍, Part 1

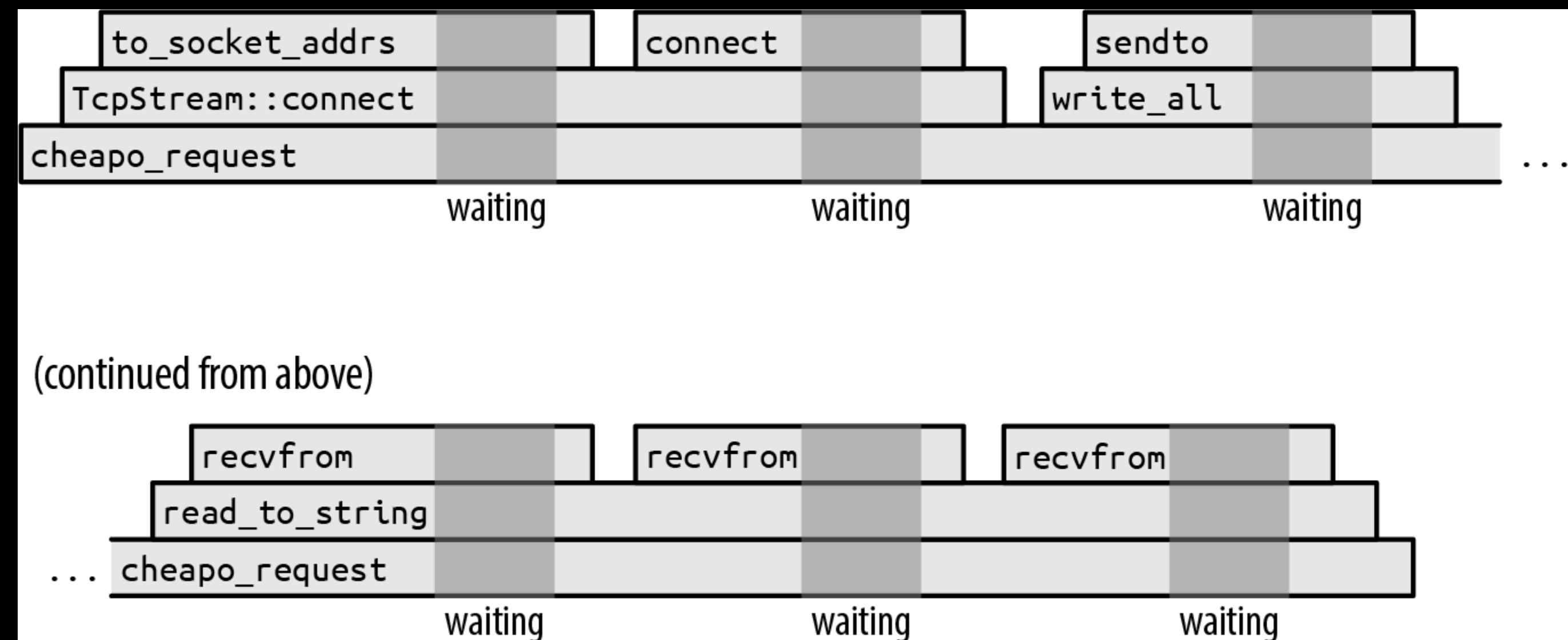
- 이 함수의 실행을 시간의 흐름에 따라 다음과 같이 처리된다.
 - 짙은 갈색 배경으로 된 부분은 운영체제가 시스템 호출을 마칠 때까지 프로그램이 기다리는 시기를 나타낸다.
 - 이 부분은 시간에 비례해서 그리지 않았는데, 그랬다면 전체 다이어그램이 대부분 짙은 갈색으로 채워질 것이다.
 - 실제로 이 함수는 대부분의 시간을 운영체제를 기다리는 데 쓴다.



동기 → 비동기

HSPACE Rust 특강
Rust Basic #16 - 비동기 프로그래밍, Part 1

- 이 함수의 실행을 시간의 흐름에 따라 다음과 같이 처리된다.
 - 이 함수가 시스템 호출이 복귀하기를 기다리는 동안에는 해당 스레드가 블록된다.
즉, 시스템 호출이 끝날 때까지 기다리는 것 외에는 아무것도 할 수 없다.
 - 스레드 스택의 크기가 커지는 건 흔한 일이라서 비슷한 일을 하느라 애쓰는 여러 스레드를 가진 커다란 시스템의 일부가 이런 식으로 작동하면 스레드의 자원을 오로지 기다리는 데만 쓰는 셈이 되므로 비용이 크게 올라간다.



- 이를 해결하려면 시스템 호출이 완료되기를 기다리는 동안 스레드가 다른 작업을 수행할 수 있어야 한다.
- 하지만 어떻게 해야할까? 예를 들어, 소켓에서 응답을 읽는데 사용 중인 함수의 시그니처는 다음과 같다.

```
● ● ●  
fn read_to_string(&mut self, buf: &mut String) -> std::io::Result<usize>;
```

- 이 함수는 **동기(Synchronous)** 버전이다. 따라서 작업이 완료되어야 호출부가 재개된다.
- 운영체제가 일하는 동안 스레드를 다른 용도로 쓰고 싶다면
이 함수의 **비동기(Asynchronous)** 버전을 제공하는 새 I/O 라이브러리가 필요하다.

- Rust는 비동기 작업을 지원하기 위해 std::future::Future 트레이잇을 도입했다.

```
● ● ●

trait Future {
    type Output;

    // For now, read `Pin<&mut Self>` as `&mut Self`.
    fn poll(self: Pin<&mut Self>, cx: &mut Context<'_>) -> Poll<Self::Output>;
}

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

- Rust는 비동기 작업을 지원하기 위해 `std::future::Future` 트레잇을 도입했다.
 - Future는 완료 여부를 테스트할 수 있는 작업을 표현한다.
Future 트레잇의 `poll` 메소드는 절대로 작업이 끝나길 기다리지 않으며, 항상 즉시 복귀한다.
 - 작업이 완료되면 `poll`은 `Poll::Ready(output)`을 반환하는데, 여기서 `output`은 최종 결과다.
 - 그렇지 않으면 `Pending`을 반환하고, 퓨처를 다시 폴링해도 좋은 시점이 되면
Context에 들어 있는 콜백 함수인 웨이커(Waker)를 호출해서 알려줄 것을 약속한다.
이 경우에 퓨처를 가지고 할 수 있는 일이라고는 값이 나올 때까지 `poll`하는 것뿐이라서
이를 비동기 프로그래밍의 피나탸 모델(Pinata Model)이라고 부르기도 한다.
- 최신 운영체제는 모두 폴링 인터페이스를 구현하는데 쓸 수 있는
시스템 호출의 변형을 포함하고 있다.
 - 예를 들어, Unix와 Windows에서 네트워크 소켓을 논블로킹 모드로 설정하면 읽기와 쓰기가 블록될 때
오류를 반환하는데, 이럴 때는 다음에 다시 시도해야 한다.

- 따라서 `read_to_string`의 비동기 버전은 다음과 같은 시그니처를 갖는다.

```
fn read_to_string(&mut self, buf: &mut String) -> impl Future<Output = Result<usize>>;
```

- 리턴 타입을 제외하면 앞에서 봤던 버전과 똑같다. 이 비동기 버전은 `Result<usize>`의 퓨처를 반환한다. 이 `Future`에 대고 `Ready(result)`가 나올 때까지 폴링하면 그때마다 최대한 많은 양의 읽기 작업이 진행된다.
- 최종 `result`는 평범한 I/O 작업과 마찬가지로 성공 값 또는 오류 값을 준다. 이런 식으로 함수의 비동기 버전은 보통 동기 버전과 동일한 인수를 받지만, 리턴 타입은 `Future`로 감싼 패턴을 갖는다.

- 따라서 `read_to_string`의 비동기 버전은 다음과 같은 시그니처를 갖는다.

```
fn read_to_string(&mut self, buf: &mut String) -> impl Future<Output = Result<usize>>;
```

- 이 함수는 호출되더라도 실제로 아무것도 읽지 않는다. 유일하게 실제 작업을 수행할 퓨처를 만들어 반환하는 일만 한다.
- 이 퓨처는 호출로 생기는 요청을 수행하는데 필요한 모든 정보를 줘고 있어야 한다. 예를 들어, 이 함수가 반환하는 퓨처는 호출 대상이 되는 입력 스트림과 들어오는 데이터를 쌓아둘 `String`을 기억해 둬야 한다.
- 사실 이 퓨처는 `self`와 `buf` 레퍼런스를 줘고 있으므로 이 함수의 적절한 시그니처는 다음과 같아야 한다.
(이 시그니처에는 반환되는 퓨처가 `self`와 `buf`가 빌린 값만큼만 살 수 있다는 걸 나타내는 수명이 추가됐다.)

```
fn read_to_string<'a>(
    &'a mut self,
    buf: &'a mut String,
) -> impl Future<Output = Result<usize>> + 'a;
```

- `async-std` 크레이트는 `std`가 가진 모든 I/O 작업의 비동기 버전을 제공한다.
 - 여기에는 `read_to_string` 메소드를 가진 비동기 `Read` 트레잇도 포함된다.
 - `async-std`는 `std`의 설계를 충실히 따르며, 가능하면 자체 인터페이스에서 `std`의 타입을 재사용하므로 오류, 결과, 네트워크 주소 등 대부분의 연관데이터가 두 진영을 자유롭게 오갈 수 있다.
 - `std`에 익숙하면 `async-std`를 쓰는 데 도움이 되고 그 반대도 마찬가지다.

- Future 트레이트의 규칙 중 하나는 Poll::Ready를 반환한 퓨처의 경우
다시 폴링되지 않는다고 가정해도 좋다는 것이다.
 - 폴링을 너무 많이 하면 어떤 퓨처는 그냥 Poll::Pending을 계속 반환하고, 또 어떤 퓨처는 패닉에 빠지거나 중단된다.
(하지만 그렇다고 해서 메모리나 스레드 안전성을 위반한다거나 미정의 동작을 일으켜서는 안된다.)
 - Future 트레이트의 fuse 어댑터 메소드는 아무 퓨처나 가져다가 그냥 Poll::Pending을 계속 반환하는 퓨처로
바꿔 놓는다. 그러나 퓨처를 소비하는 일반적인 방법은 모두 이 규칙을 따르므로 대개는 fuse가 필요 없다.

- 폴링이 비효율적으로 들리겠지만 걱정하지 말자.
 - Rust의 비동기 아키텍처는 `read_to_string` 같은 기본 I/O 함수가 제대로 구현되어 있는 한, 꼭 필요할 때만 퓨처를 폴링하면 되도록 꼼꼼히 설계됐다.
 - `poll`이 호출될 때마다 어딘가에 있는 누군가는 `Ready`를 반환하거나 최소한 목표를 향해 앞으로 나아가야 한다. 이게 어떻게 동작하는지는 뒤에서 설명한다.
- 그러나 퓨처를 쓴다는 건 여려모로 어려운 일 같다.
 - 폴링했을 때 `Poll::Pending`이 반환되면 어떻게 해야 하는 걸까? 이럴 때는 해당 스레드가 당장 할 수 있는 다른 일을 찾아야 하는 동시에 적절한 시기가 되면 다시 돌아와서 퓨처를 폴링해야 한다.
 - 사정이 이렇다 보니 프로그램이 누가 대기 중이고 또 준비가 끝나면 무얼 해야 하는지 추적하는 작업으로 가득 차게 된다. 이로 인해 `cheapo_request` 함수는 간결함을 잃게 된다.
 - 좋은 소식은 그러지 않아도 된다는 것이다!

async 함수와 await 표현식

HSPACE Rust 특강
Rust Basic #16 - 비동기 프로그래밍, Part 1

- cheapo_request를 비동기 함수(Asynchronous Function)로 작성하면 다음과 같다.

```
● ● ●

use async_std::io::prelude::*;
use async_std::net;

async fn cheapo_request(host: &str, port: u16, path: &str) -> std::io::Result<String> {
    let mut socket = net::TcpStream::connect((host, port)).await?;
    let request = format!("GET {} HTTP/1.1\r\nHost: {}\r\n\r\n", path, host);

    socket.write_all(request.as_bytes()).await?;
    socket.shutdown(net::Shutdown::Write)?;

    let mut response = String::new();
    socket.read_to_string(&mut response).await?;

    Ok(response)
}
```

async 함수와 await 표현식

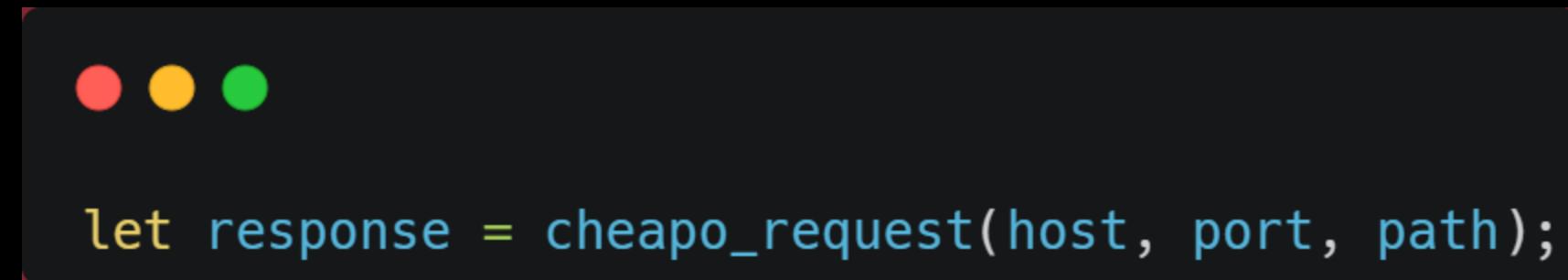
HSPACE Rust 특강
Rust Basic #16 - 비동기 프로그래밍, Part 1

- 비슷해 보이지만, 원래 버전과 다른 점이 있다.
 - 함수가 fn이 아니라 async fn으로 시작한다.
 - async_std 크레이트가 제공하는 TcpStream::connect, write_all, read_to_string의 비동기 버전을 쓴다. 이들은 모두 결과를 퓨처로 반환한다.
 - 퓨처를 반환하는 호출 코드 뒤에 전부 .await가 달려 있다.
이 부분은 퓨처가 준비될 때까지 기다릴 때 쓰는 언어에 내장된 특수 문법이다.
이때 퓨처의 최종 값은 await 표현식의 결과가 된다.
앞에서 나온 함수는 바로 이런 식으로 connect, write_all, read_to_string의 결과를 얻는다.

async 함수와 await 표현식

HSPACE Rust 특강
Rust Basic #16 - 비동기 프로그래밍, Part 1

- 비동기 함수를 호출하면 일반 함수와 달리 본문이 채 실행되기도 전에 즉시 복귀한다.
그러다 보니 호출의 최종 값이 아직 계산되지 않은 상태라 최종 값의 퓨처가 반환된다.
- 예를 들어, 다음 코드를 실행한다고 하자.



```
let response = cheapo_request(host, port, path);
```

- 이때 `response`은 `std::io::Result<String>`의 퓨처가 되며,
`cheapo_request`의 본문은 아직 실행되지 않은 채로 남게 된다.
- 비동기 함수의 리턴 타입은 따로 조정하지 않아도 된다.
Rust는 알아서 `async fn f(...) -> T`를 그냥 `T`가 아니라 `T`의 퓨처를 반환하는 함수로 취급한다.
- `async` 함수가 반환한 퓨처는 함수의 인수와 지역 변수를 위한 공간 등 함수 본문을 실행하는 데 필요한 모든 정보를 감싸 들고 있다. (호출의 스택 프레임을 평범한 Rust 값으로 캡처한 것과 같다.)
따라서 `response`은 `cheapo_request`의 본문을 실행하는 데 필요한 `host, port, path`로 넘어온 값을 주고 있다.

async 함수와 await 표현식

HSPACE Rust 특강
Rust Basic #16 - 비동기 프로그래밍, Part 1

- 퓨처의 구체적인 타입은 컴파일러가 함수의 본문과 인수를 토대로 알아서 생성한다.
 - 이 타입에는 이름이 없으며, 아는 건 오로지 `Future<Output=R>`을 구현하고 있다는 것 뿐이다. 여기서 `R`은 `async` 함수의 리턴 타입이다.
 - 이런 의미에서 비동기 함수의 퓨처는 클로저와 같다.
클로저도 컴파일러가 생성한 `FnOnce`, `Fn`, `FnMut` 트레잇을 구현하고 있는 익명 타입을 갖는다.

async 함수와 await 표현식

HSPACE Rust 특강
Rust Basic #16 - 비동기 프로그래밍, Part 1

- `cheapo_request`가 반환한 퓨처를 처음 폴링하면...
 - 함수 본문의 맨 꼭대기에서 실행이 시작되어 `TcpStream::connect`가 반환한 퓨처의 첫 번째 `await`까지 진행된다.
 - 이 `await` 표현식은 `connect` 퓨처를 폴링해서 준비 상태가 아니면 자신의 호출부에 `Poll::Pending`을 반환한다.
 - `TcpStream::connect`의 퓨처에 대한 폴링이 `Poll::Ready`를 반환하지 않으면, `cheapo_request`의 퓨처에 대한 폴링은 첫 번째 `await`에 막혀서 더 이상 진행을 할 수 없다.

async 함수와 await 표현식

HSPACE Rust 특강
Rust Basic #16 - 비동기 프로그래밍, Part 1

- 따라서 TcpStream::connect(...).await는 대충 다음과 같은 식이라고 보면 된다.

```
● ● ●

{
    // Note: this is pseudocode, not valid Rust
    let connect_future = TcpStream::connect(...);

    'retry_point:
    match connect_future.poll(cx) {
        Poll::Ready(value) => value,
        Poll::Pending => {
            // Arrange for the next `poll` of `cheapo_request`'s
            // future to resume execution at 'retry_point.

            ...
            return Poll::Pending;
        }
    }
}
```

async 함수와 await 표현식

HSPACE Rust 특강
Rust Basic #16 - 비동기 프로그래밍, Part 1

- **await** 표현식

- 퓨처의 소유권을 가져다가 여기에 대고 폴링한다.
- 퓨처가 준비 상태면 표현식의 최종 값이 `await` 표현식의 값이 되고 실행이 계속된다.
그렇지 않으면 자신의 호출부에 `Poll::Pending`을 반환한다.
- 그러나 중요한 건 `cheapo_request`의 퓨처에 대한 다음 폴링이 함수의 맨 꼭대기에서 다시 시작하지 않는다는 점이다.
정확하게는 `connect_future`를 폴링하려고 하는 지점인 함수 중간에서 실행을 재개한다.
퓨처가 준비 상태가 아니면 `async` 함수의 나머지 부분은 진행되지 않는다.

async 함수와 await 표현식

HSPACE Rust 특강
Rust Basic #16 - 비동기 프로그래밍, Part 1

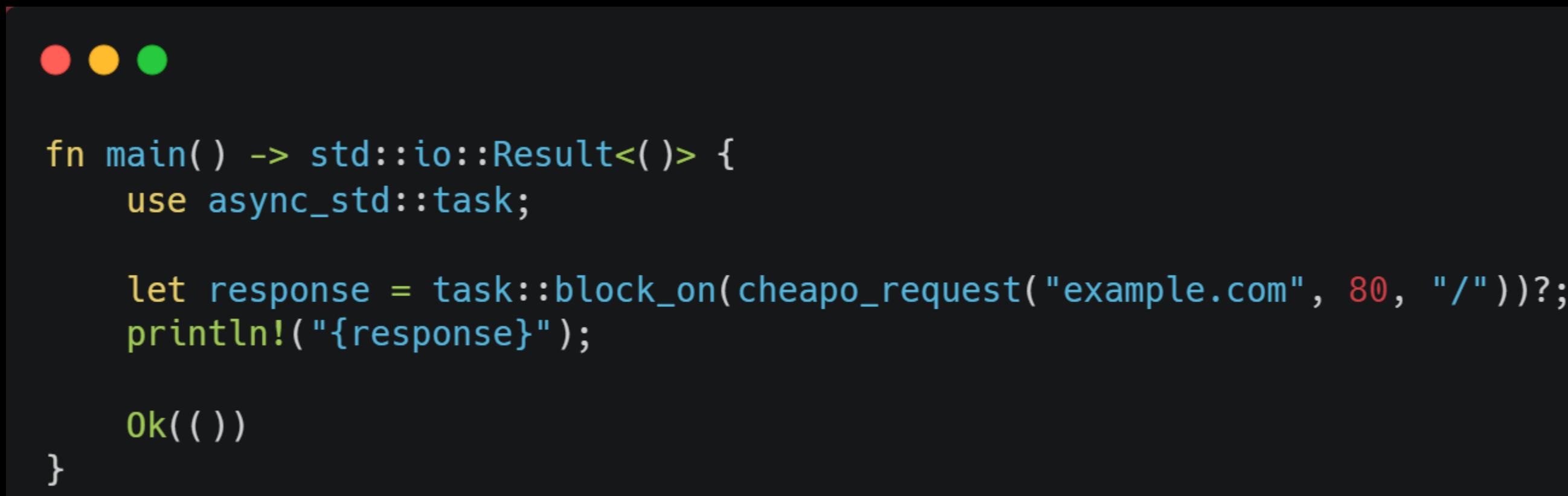
- **await** 표현식
 - 함수는 `cheapo_request`의 퓨처를 폴링할 때마다 본문을 관통하는 `await`들을 징검다리 밟듯이 한 발 한 발 건너며 실행되는데, 단 대기 중인 서브 퓨처가 준비 상태일 때만 다음으로 넘어갈 수 있다.
 - 따라서 `cheapo_request`의 퓨처를 몇 번이나 폴링해야 할지는 서브 퓨처의 행동과 함수 자체의 제어 흐름에 달렸다. `cheapo_request`의 퓨처는 다음 `poll`이 재개되어야 하는 지점을 비롯해 변수, 인수, 임시 값 등 재개에 필요한 모든 지역 상태를 추적한다.
 - 함수 중간에서 실행을 중단하고 재개하는 능력은 `async` 함수 고유의 기능이다.
(일반 함수는 복귀할 때 스택 프레임이 영원히 사라진다.)
 - `await` 표현식은 재개하는 능력에 의존하므로 `async` 함수 안에서만 쓸 수 있다.

block_on

- 어찌 보면 `async` 함수는 남에게 책임을 떠넘길 뿐이다.
 - 사실 `async` 함수에서 퓨처의 값을 가져올 때는 그냥 `await`를 걸어 두면 되니까 쉽다.
 - 그러나 `async` 함수 자체도 퓨처를 반환하므로 이걸 어떤 식으로든 풀링하는 건 이제 호출부의 몫이다. 결국 누군가는 실제로 값을 기다려야 한다.

block_on

- `async_std`의 `task::block_on` 함수를 쓰면
(`main` 함수처럼) 평범한 동기 함수에서도 `cheapo_request`를 호출할 수 있다.
 - 뮤처를 가져다가 값을 산출할 때까지 계속 폴링한다.
 - 비동기 함수의 최종 값을 산출하는 동기 함수이므로 비동기 진영과 동기 진영을 연결하는 어댑터라고 생각할 수 있다.
그러나 블록되는 특성을 갖고 있기 때문에 절대로 `async` 함수 안에서 쓰면 안 된다.
그렇지 않으면 값이 준비될 때까지 전체 스레드가 블록될 것이다.



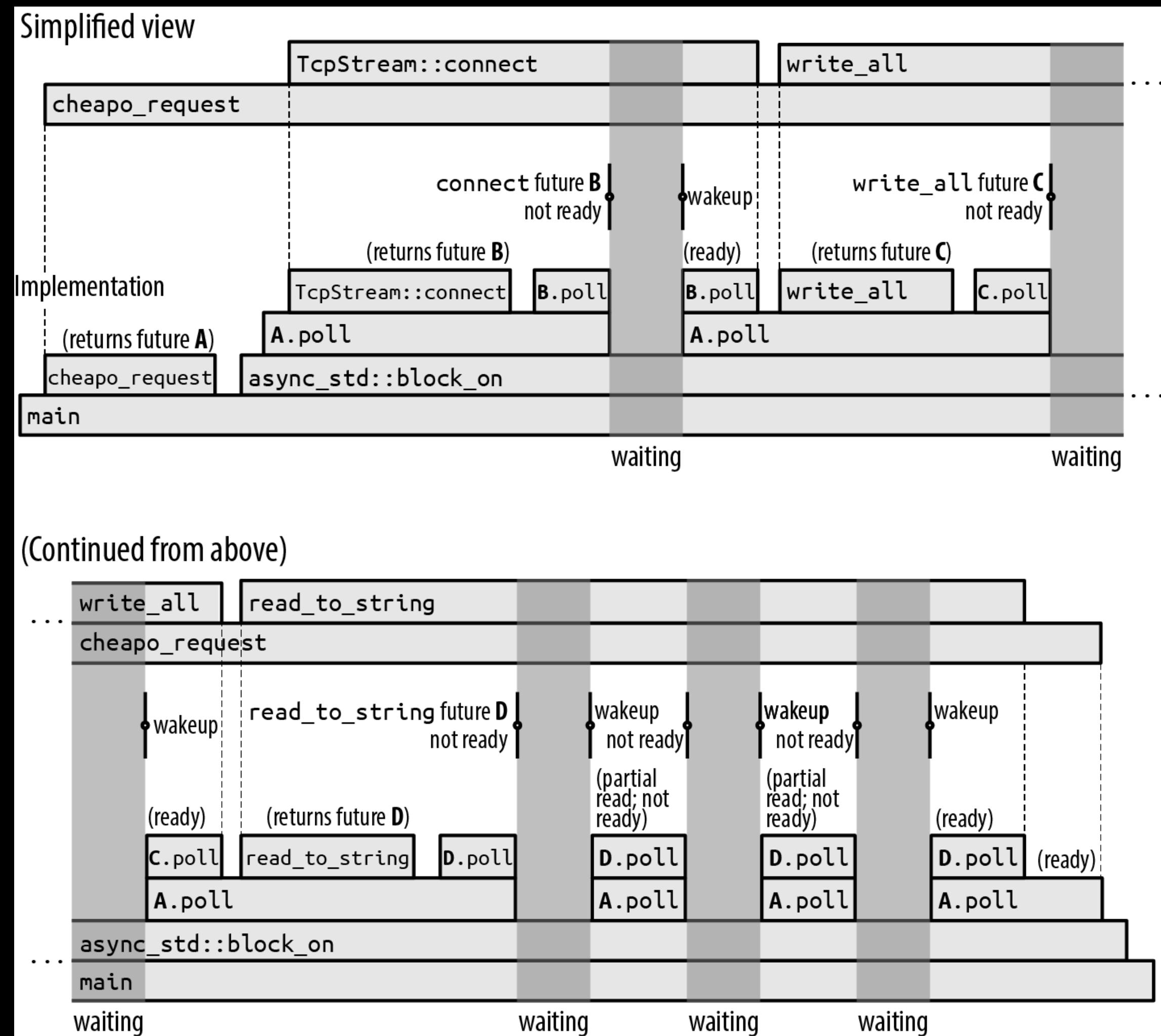
The screenshot shows a terminal window with three colored status indicators (red, yellow, green) at the top. The terminal displays the following Rust code:

```
fn main() -> std::io::Result<()> {
    use async_std::task;

    let response = task::block_on(cheapo_request("example.com", 80, "/"))?;
    println!("{}{response}");
    Ok(())
}
```

block_on

- 비동기 함수의 블로킹 구간



- 비동기 함수의 블로킹 구간
 - 타임라인 위쪽에 있는 "Simplified View"는 프로그램의 비동기 호출을 추상화해서 보여 준다.
`cheapo_request`는 먼저 `TcpStream::connect`를 호출해서 소켓을 손에 쥔 다음
이 소켓을 가지고 `write_all`과 `read_to_string`을 호출하고 복귀한다.
이는 동기 버전이 보여 주는 타임라인과 매우 비슷하다.
 - 그러나 이들 비동기 호출은 각자 여러 단계로 된 처리 과정을 거친다.
퓨처를 만든 다음 준비 상태가 될 때까지 폴링해야 하고, 그 과정에서 다른 서브 퓨처를 만들고 폴링해야 할 수도 있다.
 - 타임라인 아래쪽에 있는 "Implementation"은 이 비동기 동작을 구현하고 있는 실제 동기 호출을 보여 준다.

- 비동기 실행에서는 정확히 무슨 일이 벌어지는가
 - 먼저 main이 cheapo_request를 호출한다. 이 호출은 최종 결과에 대한 퓨처 A를 반환한다.
그런 다음 main이 이 퓨처를 async_std::block_on에 넘긴다. 따라서 폴링이 시작된다.
 - 퓨처 A가 폴링되면 cheapo_request의 본문이 실행되기 시작한다.
이 본문은 TcpStream::connect를 호출해서 소켓에 대한 퓨처 B를 확보하고 여기에 await를 건다.
좀 더 정확히 말하자면 TcpStream::connect에서 오류가 발생할 수 있으므로
이 B는 Result<TcpStream, std::io::Error>의 퓨처다.
 - 퓨처 B가 await에 의해서 폴링된다. 네트워크 연결이 아직 맺어지지 않았으므로
B.poll이 Poll::Pending을 반환하고, 대신 소켓이 준비되면 호출 태스크를 깨우도록 설정한다.
 - 퓨처 B가 준비되지 않았으므로 A.poll이 자신의 호출부인 block_on에 Poll::Pending을 반환한다.
 - block_on이 달리 할 일이 없으므로 잠자기 상태에 들어간다. 이때부터 전체 스레드가 블록된다.

- 비동기 실행에서는 정확히 무슨 일이 벌어지는가
 - B의 연결이 준비되면서 폴링했던 태스크를 깨운다. 그러면 `block_on`이 깨어나서 다시 퓨처 A를 폴링하기 시작한다.
 - A가 폴링되면 `cheapo_request`가 첫 번째 `await`에서 재개되어 다시 B를 폴링한다.
 - 이번에는 B가 준비되었다. 따라서 소켓 생성을 완료하고 A의 `poll`에 `Poll::Ready(Ok(socket))`을 반환한다.
 - 이제 `TcpStream::connect` 비동기 호출이 완료됐다.
따라서 `TcpStream::connect(...).await` 표현식의 값은 `Ok(socket)`이다.
 - `cheapo_request` 본문의 실행이 정상적으로 진행되어,
`format!` 매크로를 써서 요청 문자열을 만들고 이를 `socket.write_all`에 넘긴다.
 - `socket.write_all`은 비동기 함수이므로 자신의 결과에 대한 퓨처 C를 반환하고,
`cheapo_request`는 적절한 절차에 따라 이를 기다린다.

- 비동기 실행에서는 정확히 무슨 일이 벌어지는가
 - 나머지 이야기도 비슷하다. 그림을 보면 `socket.read_to_string`의 퓨처는 준비 상태가 되기까지 총 4번 폴링된다.
 - 매번 깨어날 때마다 소켓에서 데이터를 **조금씩** 읽긴 하지만, `read_to_string`이 입력을 끝까지 다 읽도록 명시해 두고 있어서 여러 번에 걸쳐서 작업을 진행해야 한다.
 - `poll`을 반복해서 호출하는 루프를 작성하는 건 그리 어려운 일이 아니다. 그러나 `async_std::task::block_on`의 백미는 퓨처를 실제로 다시 폴링해도 좋은 시점이 올 때까지 잠자기 상태에 들어갈 수 있어서, 무의미한 `poll` 호출을 남발하느라 프로세서 시간과 배터리 수명을 낭비하지 않는다는 데 있다.
 - `connect`와 `read_to_string` 같은 기본 I/O 함수가 반환하는 퓨처는 `poll`에 넘어온 `Context`에 들어 있는 웨이커를 들고 있다가 `block_on`이 깨어나서 다시 폴링해야 할 때 이를 호출한다. 이게 정확히 어떤 식으로 동작하는지는 뒤에서 간단한 버전의 `block_on`을 직접 구현해 보면서 살펴보기로 하자.

- 정리
 - 앞서 봤던 동기 버전과 마찬가지로 비동기 버전은 대부분의 시간을 작업이 완료되길 기다리는 데 쓴다. 시간축을 실제에 맞게 비례해서 그렸다면 다이어그램은 거의 전체가 짙은 갈색으로 물들고 프로그램이 깨어날 때 일어나는 계산 부분만 가느다란 은색 선처럼 보이게 될 것이다.
 - 지금까지 복잡한 세부 사항을 다뤘다. 보통은 타임라인 위쪽에 있는 단순한 흐름의 관점으로 생각하면 된다. 어떤 함수 호출은 동기로 이뤄지고, 또 어떤 함수 호출은 비동기라 `await`가 필요하겠지만, 결국은 그냥 다 함수 호출일 뿐이다.
 - Rust가 가진 비동기 지원의 성공 여부는 프로그래머가 시시콜콜한 구현의 세부 사항에 허덕이지 않고 실제로 단순한 흐름을 따라 일하도록 도울 수 있느냐에 달렸다.

- 스레드가 기다리는 동안에 다른 일을 하도록 만들기
 - `async_std::task::block_on` 함수는 퓨처의 값이 준비될 때까지 블록된다.
 - 그러나 퓨처 하나가 스레드를 완전히 블록시킨다면 동기 호출이나 다를 바 없다.
 - 이럴 때 쓸 수 있는 게 바로 `async_std::task::spawn_local`이다. 이 함수는 퓨처를 받아다가 풀에 넣는데, `block_on`은 현재 블로킹을 유발한 퓨처가 준비 상태가 아닐 때마다 이 풀에 있는 퓨처를 대상으로 폴링을 시도한다.
 - 따라서 퓨처 여러 개를 `spawn_local`에 넘긴 다음 최종 결과의 퓨처에 `block_on`을 적용하면, `block_on`은 진도를 빼 수 있을 때마다 넘어온 각 퓨처를 폴링하는 식으로 결과가 준비될 때까지 전체 풀을 동시에 실행한다.

비동기 태스크 생성하기

HSPACE Rust 특강
Rust Basic #16 - 비동기 프로그래밍, Part 1

- `spawn_local` 함수는 스레드를 띄울 때 쓰는 표준 라이브러리 함수 `std::thread::spawn`의 비동기 버전이다.
- `std::thread::spawn(c)`는 클로저 `c`를 받아다가 이를 실행하는 스레드를 띄우고 `std::thread::JoinHandle`을 반환한다. 여기에 대고 호출하는 `join` 메소드는 스레드가 끝날 때까지 기다렸다가 `c`가 반환한 것을 그대로 반환한다.
- `async_std::task::spawn_local(f)`는 퓨처 `f`를 받아다가 이를 현재 스레드가 `block_on`을 호출할 때 폴링되는 풀에 넣는다. `spawn_local`은 자체 `async_std::task::JoinHandle` 타입을 반환하는데, 이 자체가 퓨처이므로 여기에 `await`를 걸어서 `f`의 최종 값을 가져올 수 있다.

비동기 태스크 생성하기

HSPACE Rust 특강
Rust Basic #16 - 비동기 프로그래밍, Part 1

- 예를 들어, 전체 HTTP 요청 집합을 동시에 만들고 싶다고 하자.

```
● ● ●

pub async fn many_requests(requests: Vec<(String, u16, String)>) -> Vec<std::io::Result<String>> {
    use async_std::task;

    let mut handles = vec![];
    for (host, port, path) in requests {
        handles.push(task::spawn_local(cheapo_request(&host, port, &path)));
    }

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

    results
}
```

비동기 태스크 생성하기

HSPACE Rust 특강
Rust Basic #16 - 비동기 프로그래밍, Part 1

- 예를 들어, 전체 HTTP 요청 집합을 동시에 만들고 싶다고 하자.
 - 이 함수는 `requests`의 각 요소를 대상으로 `cheapo_request`를 호출하고 각 호출의 퓨처를 `spawn_local`에 넘긴다.
 - 그런 다음 그 결과로 나오는 `JoinHandle`을 벡터에 모아 담은 뒤에 이들 각각에 대해서 `await`를 건다.
 - 조인 핸들에 `await`를 거는 순서는 아무래도 상관없다.
 - 요청은 이미 생성된 상태이므로, 이 스레드가 `block_on`을 호출해서 달리 할 일이 없으면 그때마다 해당 요청의 퓨처가 필요에 따라 폴링될 것이다. 따라서 모든 요청이 동시에 실행된다.
 - 요청이 완료되면 `many_requests`는 결과를 호출부에 반환한다.

비동기 태스크 생성하기

HSPACE Rust 특강
Rust Basic #16 - 비동기 프로그래밍, Part 1

- 이 코드는 대체로 정확하지만, 막상 빌드하면 컴파일 오류가 발생한다.

error: `host` does not live long enough

```
handles.push(task::spawn_local(cheapo_request(&host, port, &path)));
-----^^^^^-----  
|           |  
|           borrowed value does not  
|           live long enough  
argument requires that `host` is borrowed for `'static'  
}  
- `host` dropped here while still borrowed
```

비동기 태스크 생성하기

HSPACE Rust 특강
Rust Basic #16 - 비동기 프로그래밍, Part 1

- 컴파일 오류 원인 분석
 - 비동기 함수에 레퍼런스를 넘기면 반환되는 퓨처가 그 레퍼런스를 들고 있어야 하므로 퓨처는 빌려온 값보다 오래 살 수 없다. 이 제약은 레퍼런스를 들고 있는 모든 값에 똑같이 적용된다.
 - 문제는 `spawn_local()`이 `host`와 `path`가 드롭되기 전에 태스크가 끝난다고 확신할 수 없다는 데 있다. 실제로 `spawn_local`은 수명이 ‘static’인 퓨처만 받는다. 왜냐하면 반환되는 `JoinHandle`을 그냥 무시한 채 태스크를 프로그램의 실행이 끝날 때까지 계속 실행되게 내버려 둘 수도 있기 때문이다. 이는 비동기 태스크만의 문제는 아니라서, `std::thread::spawn`으로 지역 변수의 레퍼런스를 캡처하는 클로저를 스레드에 태워 실행하려는 경우에도 비슷한 오류가 발생한다.

비동기 태스크 생성하기

HSPACE Rust 특강
Rust Basic #16 - 비동기 프로그래밍, Part 1

- 해결 방법
 - 인수의 소유권이 있는 버전을 받는 또 다른 비동기 함수를 만드는 것이다.

```
● ● ●  
  
async fn cheapo_owning_request(host: String, port: u16, path: String) -> std::io::Result<String> {  
    cheapo_request(&host, port, &path).await  
}
```

- &str 레퍼런스 대신 String을 받으므로 퓨처가 host와 path 문자열 자체를 소유하고 수명은 ‘static’이 된다. 빌림 검사기는 이 함수가 cheapo_request의 퓨처에 대고 곧바로 await를 걸고 있다는 걸 알 수 있기 때문에, 이 퓨처가 한 번도 폴링되지 않았더라도 빌려 온 host와 path 변수는 계속 그 근처에 있어야 한다. 따라서 이렇게 하면 모든 문제가 사라진다.

비동기 태스크 생성하기

HSPACE Rust 특강
Rust Basic #16 - 비동기 프로그래밍, Part 1

- `cheapo_owning_request`를 쓰면 다음과 같은 식으로 전체 요청을 할 수 있다.

```
● ● ●  
  
for (host, port, path) in requests {  
    handles.push(task::spawn_local(cheapo_owning_request(host, port, path)));  
}
```

비동기 태스크 생성하기

HSPACE Rust 특강
Rust Basic #16 - 비동기 프로그래밍, Part 1

- `block_on`을 쓰면 동기 함수인 `main`에서 `many_requests`를 호출할 수 있다.
 - 이 코드는 `block_on` 호출 안에서 세 가지 요청을 전부 동시에 실행한다.
 - 각 요청은 다른 요청이 블록되어 있는 동안 기회를 봐서 진도를 빼는데, 이 모든 일이 호출 스레드에서 이루어진다.



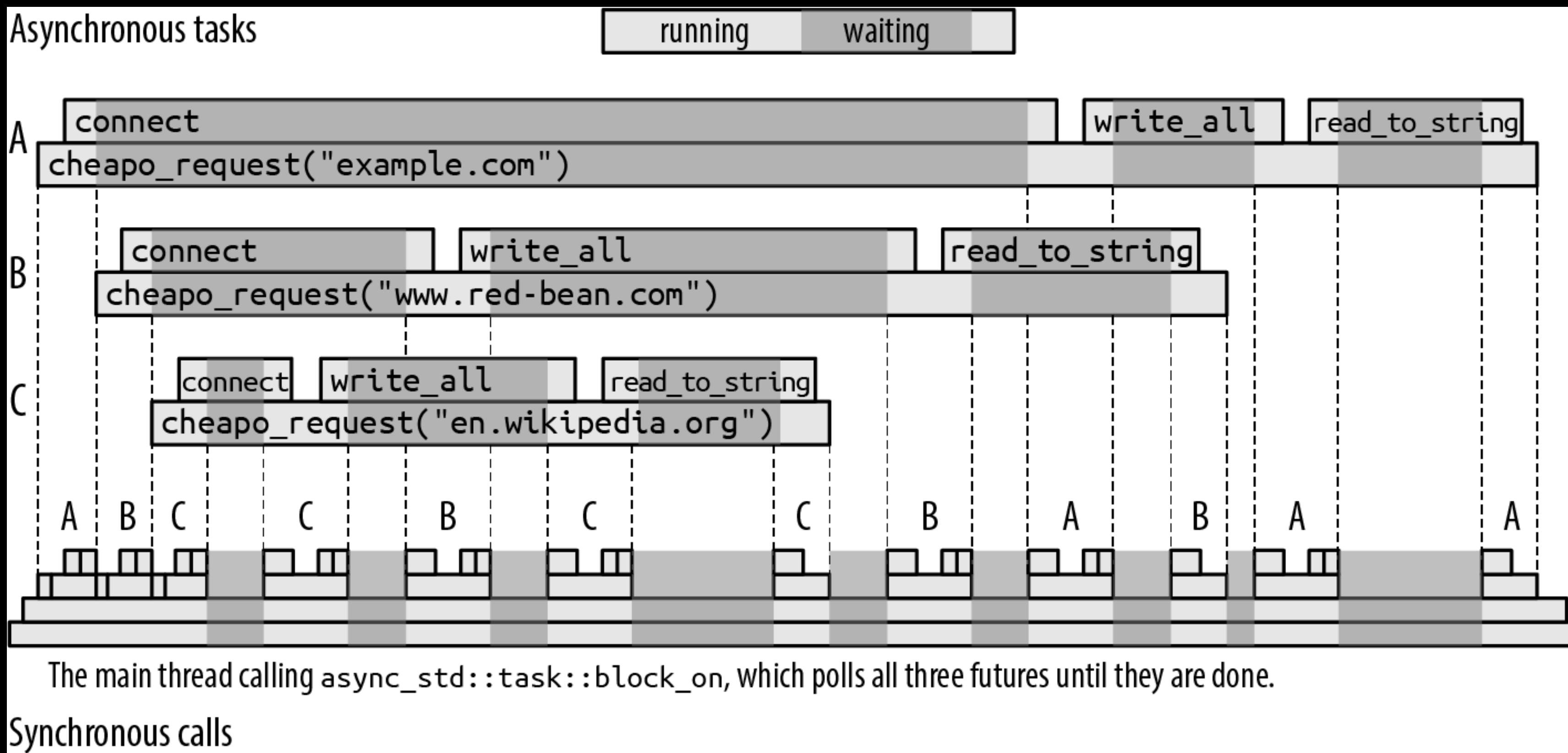
```
let requests = vec![
    ("example.com".to_string(), 80, "/".to_string()),
    ("www.red-bean.com".to_string(), 80, "/".to_string()),
    ("en.wikipedia.org".to_string(), 80, "/".to_string()),
];

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

비동기 태스크 생성하기

HSPACE Rust 특강
Rust Basic #16 - 비동기 프로그래밍, Part 1

- 다음은 이 세 가지 `cheapo_request` 호출의 실행을 보여 준다.



비동기 태스크 생성하기

HSPACE Rust 특강
Rust Basic #16 - 비동기 프로그래밍, Part 1

- `main` 함수의 실행 과정
 - `many_requests` 호출은 A, B, C라고 이름 붙인 세 가지 비동기 태스크를 생성한다.
 - `block_on`은 먼저 A를 폴링한다. 그러면 A가 `example.com`에 연결하기 시작한다. 그러다가 A가 `Poll::Pending`을 반환하면 `block_on`은 그 즉시 다음 태스크로 눈을 돌려서 B를 폴링하고 같은 식으로 C를 폴링한다. 그러면 B와 C가 각자 정해둔 서버에 연결하기 시작한다.
 - 폴링할 수 있는 퓨처가 전부 `Poll::Pending`을 반환하면, `block_on`은 `TcpStream::connect` 퓨처 중 하나가 다시 폴링해도 좋은 시점임을 알려 올 때까지 잠자기 상태에 들어간다.
 - 이 실행에서는 `en.wikipedia.org` 서버가 다른 것들보다 더 빨리 응답하므로 여기에 해당하는 태스크가 제일 먼저 끝난다. 생성된 태스크가 끝나면 자신의 값을 `JoinHandle`에 넣고 준비 상태로 표시해 두기 때문에 이를 기다리는 `many_requests`가 진도를 빨 수 있다.
 - 이런 식으로 나머지 `cheapo_request` 호출이 성공하거나 오류를 반환하면 마침내 `many_requests` 자체가 복귀한다. 그리고 끝으로 `main`이 `block_on`에게서 결과 벡터를 받는다.

비동기 태스크 생성하기

HSPACE Rust 특강
Rust Basic #16 - 비동기 프로그래밍, Part 1

- `main` 함수의 실행 과정
 - 이 실행은 전부 한 스레드에서 일어나며, 세 가지 `cheapo_request` 호출은 퓨처가 잇따라 폴링되면서 교차로 실행된다. 비동기 호출은 하나의 함수 호출이 완료를 향해 달려가는 모양새를 띠지만, 실상은 퓨처의 `poll` 메소드에 대한 일련의 동기 호출로 되어 있다. 각 `poll` 호출은 빠르게 복귀해서 다른 비동기 호출이 실행될 수 있도록 스레드를 양보한다.

비동기 태스크 생성하기

HSPACE Rust 특강
Rust Basic #16 - 비동기 프로그래밍, Part 1

- 비동기 태스크와 스레드 사이에는 한 가지 염두에 둬야 할 중요한 차이가 있다.
 - 비동기 태스크 간의 전환이 `await` 표현식에서 대기 중인 퓨처가 `Poll::Pending`을 반환할 때만 일어난다는 점이다. 이 말은 `cheapo_request`에 오래 걸리는 계산을 두면, 이게 끝날 때까지 `spawn_local`에 넘긴 다른 태스크가 실행될 기회를 얻지 못한다는 뜻이다.
 - 스레드의 경우는 이런 문제가 발생하지 않는다. 운영체제가 언제든 스레드를 중단시킬 수 있고, 또 프로세서를 독점하는 스레드가 생기지 않도록 타이머를 설정해둘 수 있기 때문이다.
 - 비동기 코드는 스레드를 공유하는 퓨처들의 자발적인 협력에 의존한다. 오래 걸리는 계산과 비동기 코드가 공존해야 한다면 뒤에서 설명하는 몇 가지 옵션을 참고하자.

- Rust는 비동기 함수와 더불어 **비동기 블록(Asynchronous Block)**도 지원한다.

- 평범한 블록문은 마지막 표현식의 값을 반환하지만 async 블록은 마지막 표현식의 값에 대한 퓨처를 반환한다.
- async 블록에서는 await 표현식을 쓸 수 있다.

```
let serve_one = async {
    use async_std::net;

    // Listen for connections, and accept one.
    let listener = net::TcpListener::bind("localhost:8087").await?;
    let (mut socket, _addr) = listener.accept().await?;

    // Talk to client on `socket`.
    ...
};
```

- 코드 설명

- `serve_one`을 퓨처로 초기화한다. 이 퓨처는 폴링되면 수신 대기 상태로 있다가 들어오는 TCP 연결 하나를 처리한다. `async` 함수 호출이 자신의 퓨처가 폴링될 때까지 실행되지 않는 것처럼, 이 블록의 본문은 `serve_one`이 폴링될 때까지 실행되지 않는다.
- `async` 블록 안에서 발생하는 오류에 ? 연산자를 적용하면 바깥쪽 함수가 아니라 그 블록에서 복귀한다. 예를 들어, 앞에 있는 `bind` 호출이 오류를 반환하면 ? 연산자는 이를 `serve_one`의 최종 값으로 반환한다. 마찬가지로 `return` 표현식은 바깥쪽 함수가 아니라 `async` 블록에서 복귀한다.
- `async` 블록이 바깥쪽 코드에 정의된 변수를 참조하면 해당 퓨처는 클로저의 경우처럼 그 값을 캡처한다. 또 `move` 클로저의 경우처럼 블록 앞에 `async move`를 붙이면 단순히 캡처한 값의 레퍼런스를 줘는 게 아니라 그 값의 소유권을 가져올 수 있다.

async 블록

HSPACE Rust 특강
Rust Basic #16 - 비동기 프로그래밍, Part 1

- async 블록은 코드에서 비동기로 실행하고 싶은 부분을 간결하게 분리하는 방법을 제공한다.
 - 예를 들어, 앞에서는 spawn_local이 ‘static 퓨처를 요구하는 바람에 cheapo_owning_request 래퍼 함수를 정의해서 인수의 소유권을 가진 퓨처를 넘겨줘야 했다. async 블록에서는 그냥 cheapo_request를 호출하기만 하면 번거롭게 래퍼 함수를 작성하지 않아도 같은 효과를 낼 수 있다.

```
● ● ●

pub async fn many_requests(requests: Vec<(String, u16, String)>) -> Vec<std::io::Result<String>> {
    use async_std::task;

    let mut handles = vec![];
    for (host, port, path) in requests {
        handles.push(task::spawn_local(async move {
            cheapo_request(&host, port, &path).await
        }));
    }
    ...
}
```

- 코드 설명
 - async move 블록이므로 move 클로저의 경우처럼 퓨처가 String 값인 host와 path의 소유권을 갖는다.
 - 그런 다음 레퍼런스를 cheapo_request에 넘긴다.
 - 빌림 검사기는 이 블록의 await 표현식이 cheapo_request가 변환한 퓨처의 소유권을 갖는다는 걸 알 수 있으므로, host와 path의 레퍼런스는 자신이 빌려 온 캡처된 변수보다 더 오래 살 수 없다.
 - 이 async 블록은 더 적은 상용구로 cheapo_owning_request와 똑같은 일을 해낸다.

async 블록

HSPACE Rust 특강
Rust Basic #16 - 비동기 프로그래밍, Part 1

- 하지만 문제가 될 만한 부분이 하나 있다.
 - async 블록의 경우 async 함수의 인수 뒤에 오는 -> T와 유사한 리턴 타입을 지정하는 문법이 없다는 것이다. 이 점은 ? 연산자를 쓸 때 문제가 될 수 있다.

```
● ● ●

let input = async_std::io::stdin();
let future = async {
    let mut line = String::new();

    // This returns `std::io::Result<usize>`.
    input.read_line(&mut line).await?;

    println!("Read line: {}", line);
    Ok(())
};
```

async 블록

- 하지만 문제가 될 만한 부분이 하나 있다.
 - 이 코드는 다음과 같은 오류를 내며 실패한다.

- 컴파일 오류 원인 분석
 - Rust는 async 블록의 리턴 타입이 무엇인지 알 수 없다.
`read_line` 메소드는 `Result<(), std::io::Error>`를 반환하지만,
? 연산자는 `From` 트레잇을 써서 주어진 오류 타입을 현 상황이 요구하는 무언가로 변환하기 때문에,
async 블록의 리턴 타입은 `From<std::io::Error>`를 구현하고 있는 임의의 타입 `E`에 대해서
`Result<(), E>`가 될 수 있다.

async 블록

HSPACE Rust 특강
Rust Basic #16 - 비동기 프로그래밍, Part 1

- 해결 방법

- 1) async 블록의 리턴 타입을 명시하는 문법이 들어가기를 기다린다.
- 2) 블록의 최종 Ok 타입을 명시해 문제를 우회한다.

```
● ● ●

let future = async {
    ...
    Ok::<(), std::io::Error>(())
};
```

async 함수 만들기

HSPACE Rust 특강
Rust Basic #16 - 비동기 프로그래밍, Part 1

- 비동기 블록은 비동기 함수와 똑같은 효과를 내면서도 좀 더 유연한 또 다른 방법을 제공한다.
- 첫 번째 접근 방식은 `cheapo_request`를 `async` 블록의 퓨처를 반환하는 평범한 동기 함수로 작성한다.

```
use std::future::Future;
use std::io;

fn cheapo_request<'a>(
    host: &'a str,
    port: u16,
    path: &'a str,
) -> impl Future<Output = io::Result<String>> + 'a {
    async move {
        ... function body ...
    }
}
```

async 함수 만들기

HSPACE Rust 특강
Rust Basic #16 - 비동기 프로그래밍, Part 1

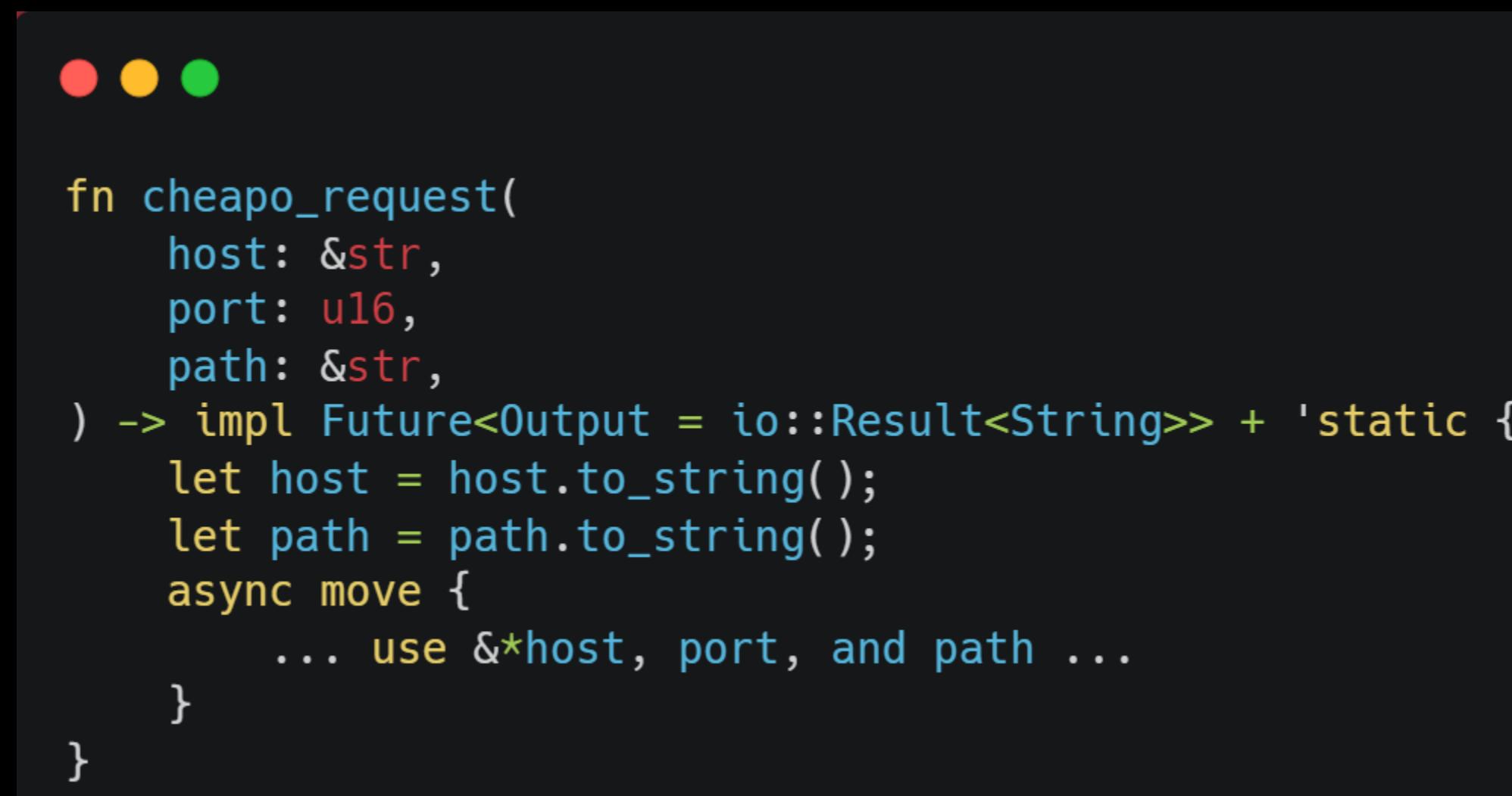
- **코드 설명**

- 함수를 호출하면 그 즉시 async 블록의 값에 대한 퓨처가 반환된다.
- 이 퓨처는 함수의 인수를 캡처하며, 비동기 함수가 반환한 퓨처처럼 행동한다.
- async fn 문법을 쓰고 있는 게 아니므로 리턴 타입에 impl Future를 적어 주어야 하지만, 호출부 입장에서는 두 정의가 다 같은 함수 시그니처를 가진 서로 호환되는 구현이다.

async 함수 만들기

HSPACE Rust 특강
Rust Basic #16 - 비동기 프로그래밍, Part 1

- 비동기 블록은 비동기 함수와 똑같은 효과를 내면서도 좀 더 유연한 또 다른 방법을 제공한다.
 - 두 번째 접근 방식은 `cheapo_request`와 `spawn_local`을 함께 쓰기 위한 전략으로, `cheapo_request`를 동기 함수로 바꾼 다음 그 안에 인수의 완전한 소유권을 가진 복사본을 만들어서 이를 캡처하는 ‘`static` 퓨처를 반환한다.
 - 함수가 호출되자마자 결과에 대한 퓨처를 생성하기에 앞서 뭔가 계산이 필요한 경우에 유용하다.



```
● ● ●

fn cheapo_request(
    host: &str,
    port: u16,
    path: &str,
) -> impl Future<Output = io::Result<String>> + 'static {
    let host = host.to_string();
    let path = path.to_string();
    async move {
        ... use &*host, port, and path ...
    }
}
```

async 함수 만들기

HSPACE Rust 특강
Rust Basic #16 - 비동기 프로그래밍, Part 1

- **코드 설명**

- async 블록이 host와 path를 &str 레퍼런스가 아니라 소유권이 있는 String 값으로 캡처한다.
- 이 퓨처는 실행하는데 필요한 모든 데이터를 소유하고 있으므로 ‘static 수명을 갖는다.
(시그니처에는 + ‘static이라고 적어 두었지만 -> impl 리턴 타입에는 ‘static이 기본값이므로 생략해도 무방하다.)
- 이 버전의 cheapo_request는 ‘static인 퓨처를 반환하므로 이를 직접 spawn_local에 넘길 수 있다.

```
● ● ●

let join_handle = async_std::task::spawn_local(
    cheapo_request("areweasyncyet.rs", 80, "/")
);

... other work ...

let response = join_handle.await?;
```

스레드 풀에서 실행하기

HSPACE Rust 특강
Rust Basic #16 - 비동기 프로그래밍, Part 1

- 지금까지 살펴본 예는 대부분의 시간을 I/O를 기다리는 데 쓰지만, 작업 부하 중 일부는 프로세서 작업과 블로킹이 섞여 있다.
- 프로세서 하나로 감당하기 어려울 만큼 계산할 게 많을 때는 `async_std::task::spawn`을 써서 퓨처를 워커 스레드 풀에서 실행할 수 있다.
이 워커 스레드 풀은 준비 상태에 있는 퓨처를 폴링해서 진도를 빼는 일만 전담한다.

스레드 풀에서 실행하기

HSPACE Rust 특강
Rust Basic #16 - 비동기 프로그래밍, Part 1

- `async_std::task::spawn`의 사용법은 `async_std::task::spawn_local`과 같다.
 - `spawn`은 `spawn_local`과 마찬가지로 `JoinHandle` 값을 반환하며, 여기에 대고 `await`를 걸면 퓨처의 최종 값을 얻을 수 있다. 그러나 `spawn_local`과 달리 퓨처가 폴링되고 말고는 `block_on` 호출과 관계가 없다.
 - 실제로 `spawn()`이나 `spawn_local`보다 더 널리 쓰인다. 왜냐하면 사람들은 자신의 작업 부하에 계산과 블로킹이 섞여 있다 하더라도 머신의 리소스를 균형 있게 쓰고 싶어 하기 때문이다.

```
use async_std::task;

let mut handles = vec![];
for (host, port, path) in requests {
    handles.push(task::spawn(async move {
        cheapo_request(&host, port, &path).await
    }));
}
...
}
```

스레드 풀에서 실행하기

HSPACE Rust 특강
Rust Basic #16 - 비동기 프로그래밍, Part 1

- `spawn`을 쓸 때 유의할 점
 - 스레드 풀이 늘 바쁘게 돌아가다 보니 어떤 스레드가 내 퓨처를 폴링하게 될지 모른다.
`async` 호출은 한 스레드에서 실행을 시작한 뒤에 `await` 표현식에서 블록되었다가 다른 스레드에서 재개될 수도 있다.
따라서 `async` 함수 호출을 하나의 연결된 코드 실행으로 단순화해서 바라보는 게 합리적이더라도,
(실제로 비동기 함수와 `await` 표현식의 목적은 이를 그런 식으로 생각하게끔 만드는 것이다)
실제로 호출은 다른 여러 스레드에서 수행될 수도 있다.
 - 스레드 로컬 스토리지를 쓰고 있다면 `await` 표현식 앞에서 넣어 둔 데이터가
나중에 완전히 다른 무언가로 대체되는 걸 보고 당황해할 수도 있다.
이런 일이 벌어지는 이유는 이제 태스크를 풀에 있는 다른 스레드가 폴링하기 때문이다.
이게 문제가 된다면 태스크 로컬 스토리지를 써야 하는데,
자세한 내용은 `async-std` 크레이트의 문서에서 `task_local!` 매크로 부분을 참고하자.

Send를 구현해야 하는 이유

HSPACE Rust 특강
Rust Basic #16 - 비동기 프로그래밍, Part 1

- spawn에는 spawn_local에는 없는 제약이 하나 있다.
 - 퓨처가 여러 스레드를 오가며 실행되기 때문에 꼭 Send 마커 트레잇을 구현해야 한다는 점이다.
 - 퓨처는 주고 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::spawn으로 Send가 아닌 값을 캡처하는 클로저를 가진 스레드를 시작시키려 할 때도 비슷한 오류가 발생한다.
 - 차이점이라면 std::thread::spawn에 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

HSPACE Rust 특강
Rust Basic #16 - 비동기 프로그래밍, Part 1

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

```
● ● ●

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

HSPACE Rust 특강
Rust Basic #16 - 비동기 프로그래밍, Part 1

- 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 주고 있어야 한다.
 - `reluctant`의 퓨처는 `await` 이후에 `string`을 써야 하므로 적어도 몇 차례 `Rc<String>` 값을 주게 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

HSPACE Rust 특강
Rust Basic #16 - 비동기 프로그래밍, Part 1

- 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

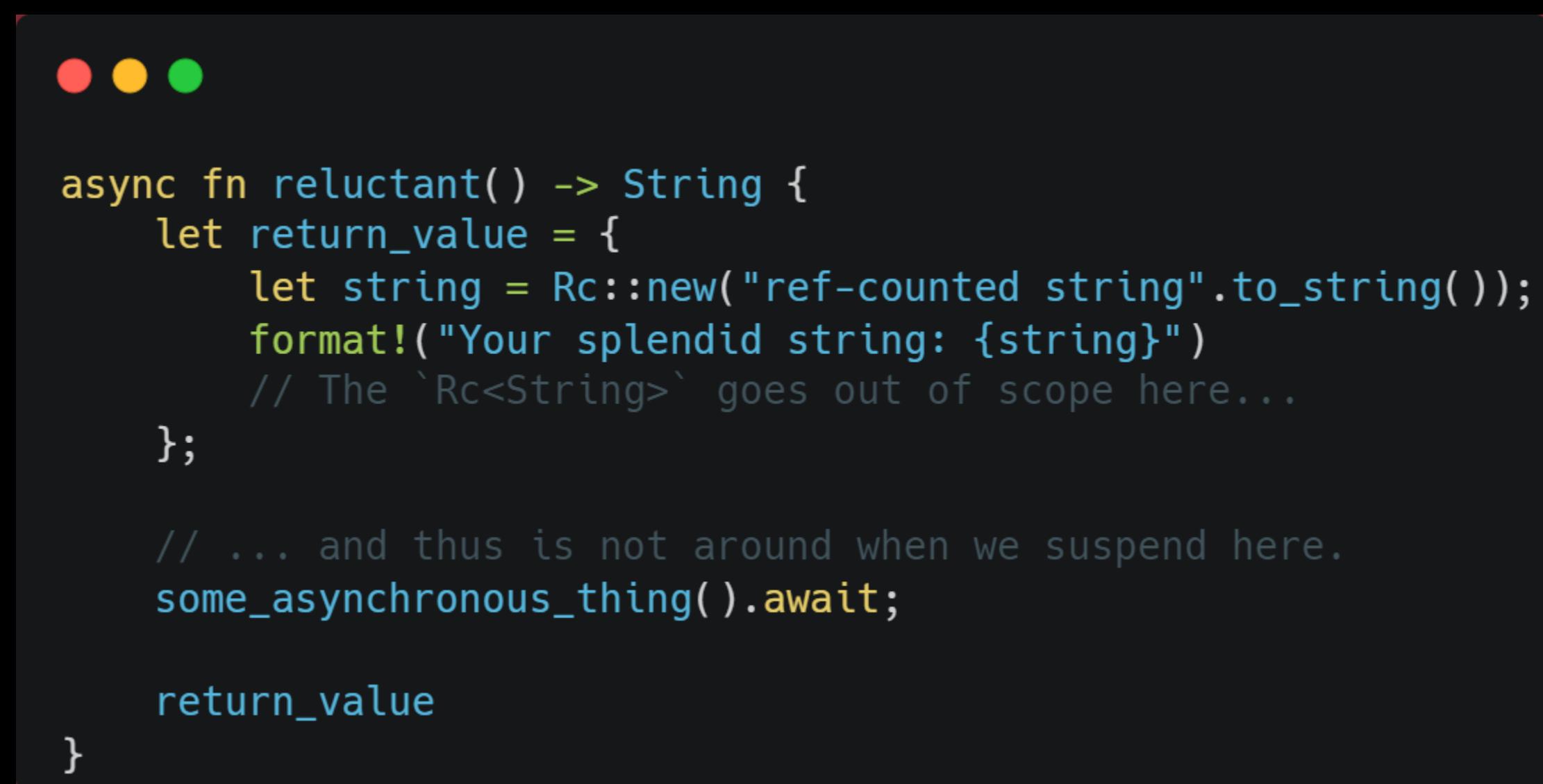
HSPACE Rust 특강
Rust Basic #16 - 비동기 프로그래밍, Part 1

- 오류 메시지 분석
 - 길지만 유용한 세부 정보를 많이 담고 있다.
 - 퓨처가 Send여야 하는 이유를 설명한다. 이 부분은 `task::spawn`의 요구 사항이다.
 - 어떤 값이 Send가 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

HSPACE Rust 특강
Rust Basic #16 - 비동기 프로그래밍, Part 1

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



The screenshot shows a terminal window with three colored dots (red, yellow, green) at the top. The terminal displays the following Rust code:

```
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

HSPACE Rust 특강
Rust Basic #16 - 비동기 프로그래밍, Part 1

- 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

HSPACE Rust 특강
Rust Basic #16 - 비동기 프로그래밍, Part 1

- 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.

```
// 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 준다. 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

HSPACE Rust 특강
Rust Basic #16 - 비동기 프로그래밍, Part 1

- 따라서 다음 코드는 동작하지 않는다.

```
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

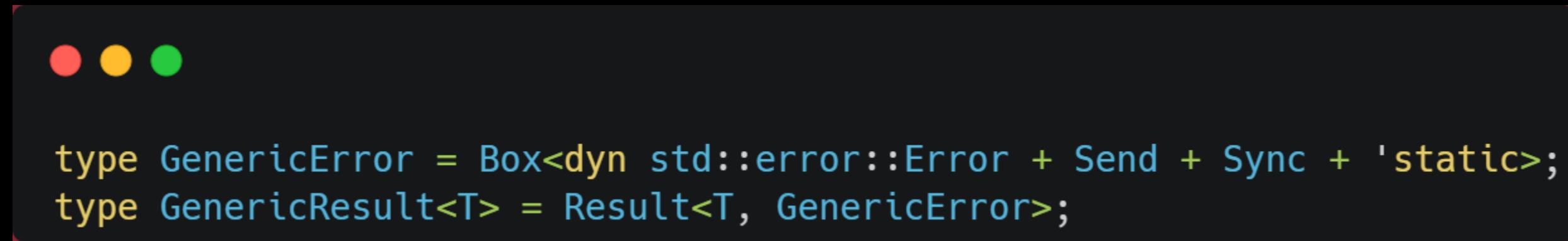
HSPACE Rust 특강
Rust Basic #16 - 비동기 프로그래밍, Part 1

- 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 오류값이 존재할 수 없다.

Send를 구현해야 하는 이유

HSPACE Rust 특강
Rust Basic #16 - 비동기 프로그래밍, Part 1

- 해결 방법
 - 좀 더 엄격한 제네릭 오류 타입을 쓰는 것이다.



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

- 이 트레이트 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 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 await에 닿을 때까지 오래 걸릴 수 있고, 이로 인해서 다른 비동기 태스크가 자기 스레드 차례를 생각보다 오래 기다리게 된다.
- 이를 피하는 한 가지 방법은 그냥 await를 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

HSPACE Rust 특강
Rust Basic #16 - 비동기 프로그래밍, Part 1

- 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.

- **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 `async`에 좀 더 적합한 형태로 바꾸기가 어려울 수 있다.
- 아니면 계산 과정에 있는 모든 경로가 `await`를 드문드문 수행하도록 만드는 게 어려울 수 있다.
- 이럴 때는 `async_std::task::spawn_blocking`을 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

HSPACE Rust 특강
Rust Basic #16 - 비동기 프로그래밍, Part 1

- 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 argonautica를 쓰는 법은 다음과 같다.

오래 걸리는 계산

HSPACE Rust 특강
Rust Basic #16 - 비동기 프로그래밍, Part 1

- 예제 코드

- 이 함수는 key가 데이터베이스 전체에 쓰이는 키일 때 password가 hash와 일치하면 Ok(true)를 반환한다.
이런 식으로 검증 작업을 클로저에 담아 spawn_blocking에 넘기면 비용이 많이 드는 계산이 자체 스레드에서 실행되므로 다른 사용자 요청의 응답성을 해치지 않는다.

```
● ● ●

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
}
```

- 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

HSPACE Rust 특강
Rust Basic #16 - 비동기 프로그래밍, Part 1

- 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

HSPACE Rust 특강
Rust Basic #16 - 비동기 프로그래밍, Part 1

- 제대로 된 비동기 HTTP 클라이언트 코드를 살펴 보자.
 - reqwest와 surf를 포함해서 훌륭한 크레이트가 여럿 있으니 그 중 하나를 골라 사용하면 된다.
 - 여기서는 async-std와 surf를 사용한다.

진짜 비동기 HTTP 클라이언트

HSPACE Rust 특강
Rust Basic #16 - 비동기 프로그래밍, Part 1

- 제대로 된 비동기 HTTP 클라이언트 코드를 살펴 보자.

```
● ● ●

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));
    }

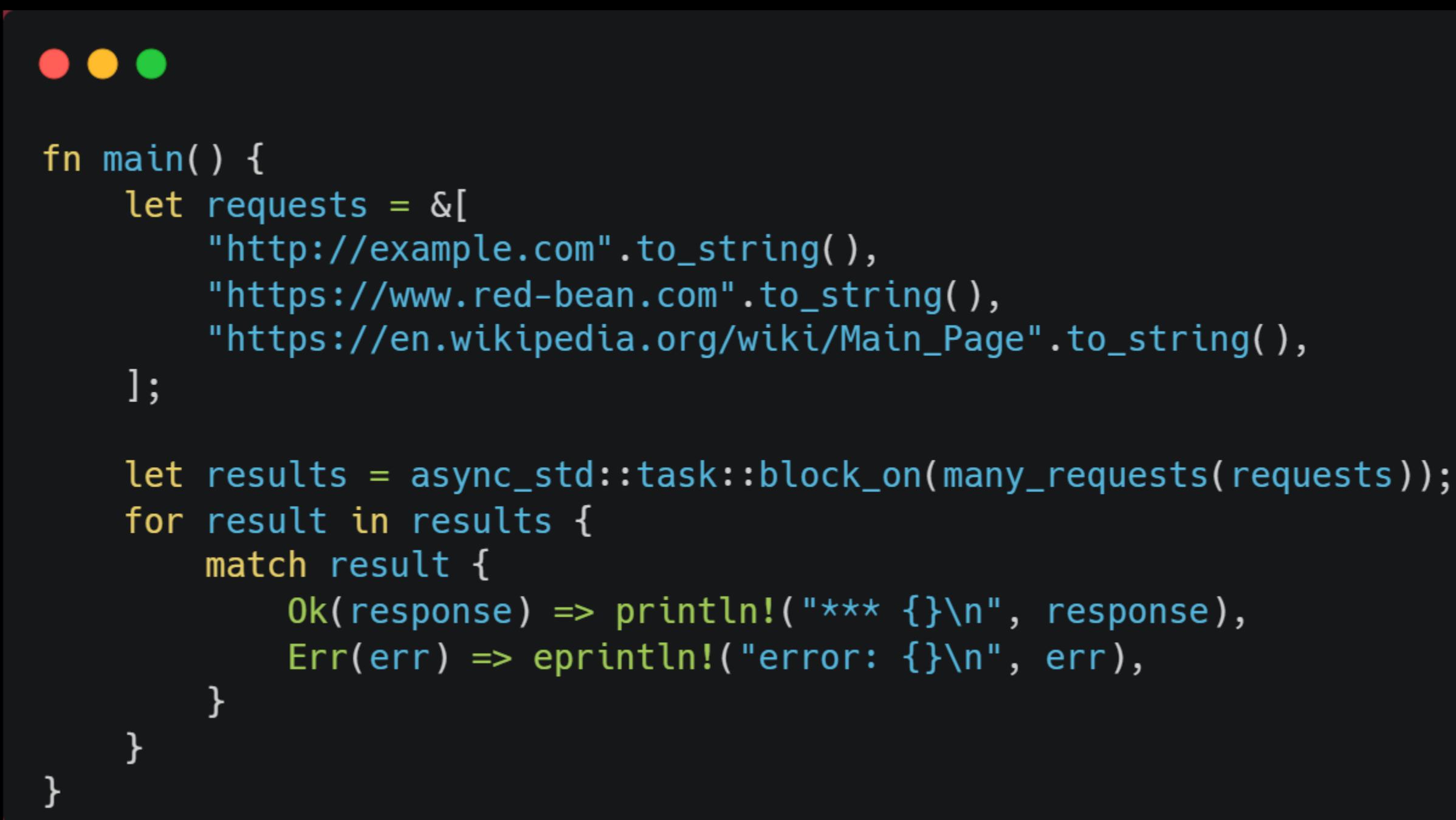
    let mut results = vec![];
    for handle in handles {
        results.push(handle.await);
    }

    results
}
```

진짜 비동기 HTTP 클라이언트

HSPACE Rust 특강
Rust Basic #16 - 비동기 프로그래밍, Part 1

- 제대로 된 비동기 HTTP 클라이언트 코드를 살펴 보자.



```
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

HSPACE Rust 특강
Rust Basic #16 - 비동기 프로그래밍, Part 1

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

```
● ● ●

[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

HSPACE Rust 특강
Rust Basic #16 - 비동기 프로그래밍, Part 1

- 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

HSPACE Rust 특강
Rust Basic #16 - 비동기 프로그래밍, Part 1

- 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

HSPACE Rust 특강
Rust Basic #16 - 비동기 프로그래밍, Part 1

- 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.

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

pub type ChatError = Box;
pub type ChatResult<T> = Result<T, ChatError>;
```

- 라이브러리 크레이트는 전체 채팅 프로토콜을 두 가지 타입으로 담아낸다.

```
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),
}
```

- **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

HSPACE Rust 특강
Rust Basic #16 - 비동기 프로그래밍, Part 1

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

```
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

HSPACE Rust 특강
Rust Basic #16 - 비동기 프로그래밍, Part 1

- **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

HSPACE Rust 특강
Rust Basic #16 - 비동기 프로그래밍, Part 1

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

```
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(())
}
```

- 코드 설명

- packet의 JSON 표현을 String으로 만들고, 끝에 새 줄을 넣어서 전부 outbound에 기록한다.
- where 절을 보면 send_as_json이 상당히 유연하다는 걸 알 수 있다.
보낼 패킷의 타입 P는 serde::Serialize를 구현하고 있는 것이라면 무엇이든 될 수 있다.
출력 스트림 S는 출력 스트림을 위한 std::io::Write 트레잇의 비동기 버전인 async_std::io::Write를 구현하고 있는 것이라면 무엇이든 될 수 있다.
- 이 조건이라면 FromClient와 FromServer 값을 비동기 TcpStream에 보내기에 충분하다.
send_as_json의 정의를 제네릭으로 가져가면 놀랍게도 스트림이나 패킷 타입의 세부 사항에 의존하지 않게 된다.
send_as_json은 이들 트레잇이 가진 메소드만 쓸 수 있다.

- **코드 설명**
 - S에 붙은 Unpin 제약 조건은 write_all 메소드를 쓰는 데 필요하다.
 - 마지막에 패킷을 outbound 스트림에 바로 직렬화하지 않고 임시 String에 직렬화한 다음, outbound에 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

HSPACE Rust 특강
Rust Basic #16 - 비동기 프로그래밍, Part 1

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

```
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)
    })
}
```

- 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.

- 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.

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

```
● ● ●

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(())
}
```

- 코드 설명
 - 서버에서 데이터를 받는 소켓을 가져다가 (async_std 버전의) BufReader로 감싼 뒤에 receive_as_json에 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

HSPACE Rust 특강
Rust Basic #16 - 비동기 프로그래밍, Part 1

- 이제 채팅 클라이언트의 메인 함수를 보자.

```
● ● ●

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

HSPACE Rust 특강
Rust Basic #16 - 비동기 프로그래밍, Part 1

- **코드 설명**
 - 명령줄에서 서버의 주소를 가져온 뒤에 일련의 비동기 함수를 호출해야 하므로, 함수의 나머지 부분을 비동기 블록으로 감싼 다음 이 블록의 끝처를 `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

HSPACE Rust 특강
Rust Basic #16 - 비동기 프로그래밍, Part 1

- **코드 설명**

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

```
● ● ●

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 하나가 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.

감사합니다.

utilForever@gmail.com

<https://github.com/utilForever>

Facebook, Twitter: @utilForever