

# Rust异步编程入门

异步编程让程序不通过多线程达到类似多线程的效果

苏林

## 自我介绍

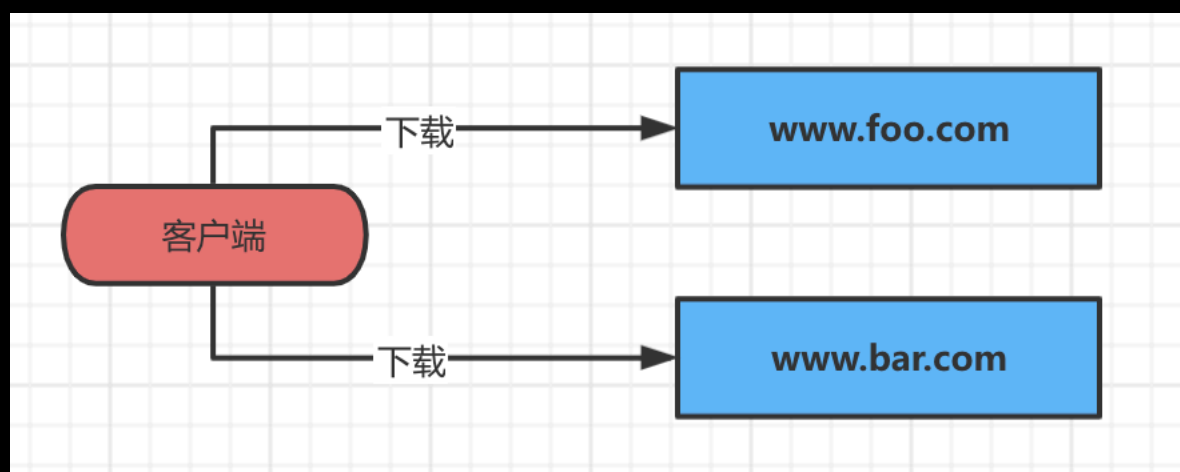
- 前折800互联网研发团队负责人, 10余年一线研发经验
- 目前是多点Dmall技术Leader
- 具有多年的软件开发经验, 熟悉Ruby、Java、Rust等开发语言
- 同时也参与过Rust中文社区日报维护工作.

## 分享内容

- 为什么需要异步
- `async/await` 介绍
- 异步编程模型
- 带领大家实现一个简化版的 `future`

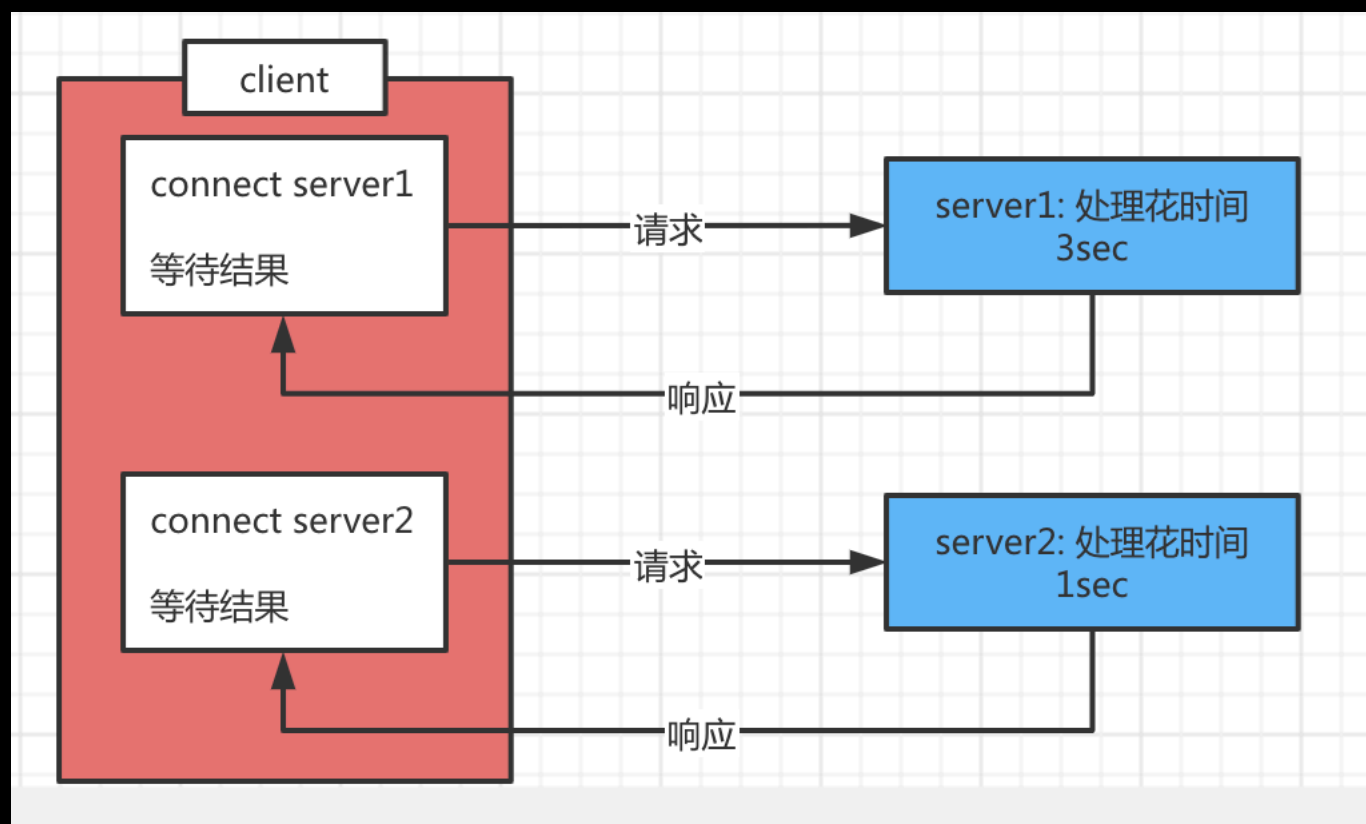
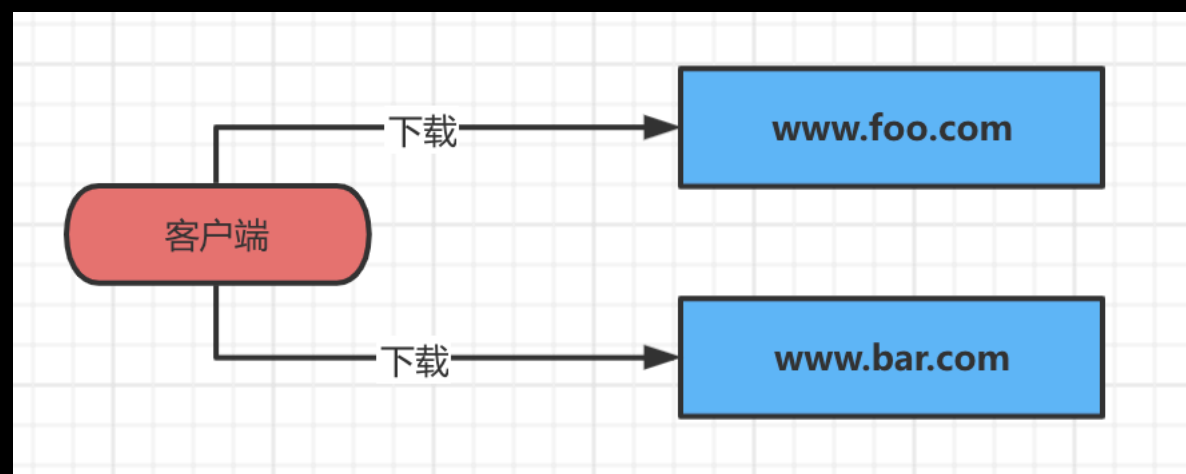
## 为什么需要异步

通过官方教程的一个例子, 来理解一下

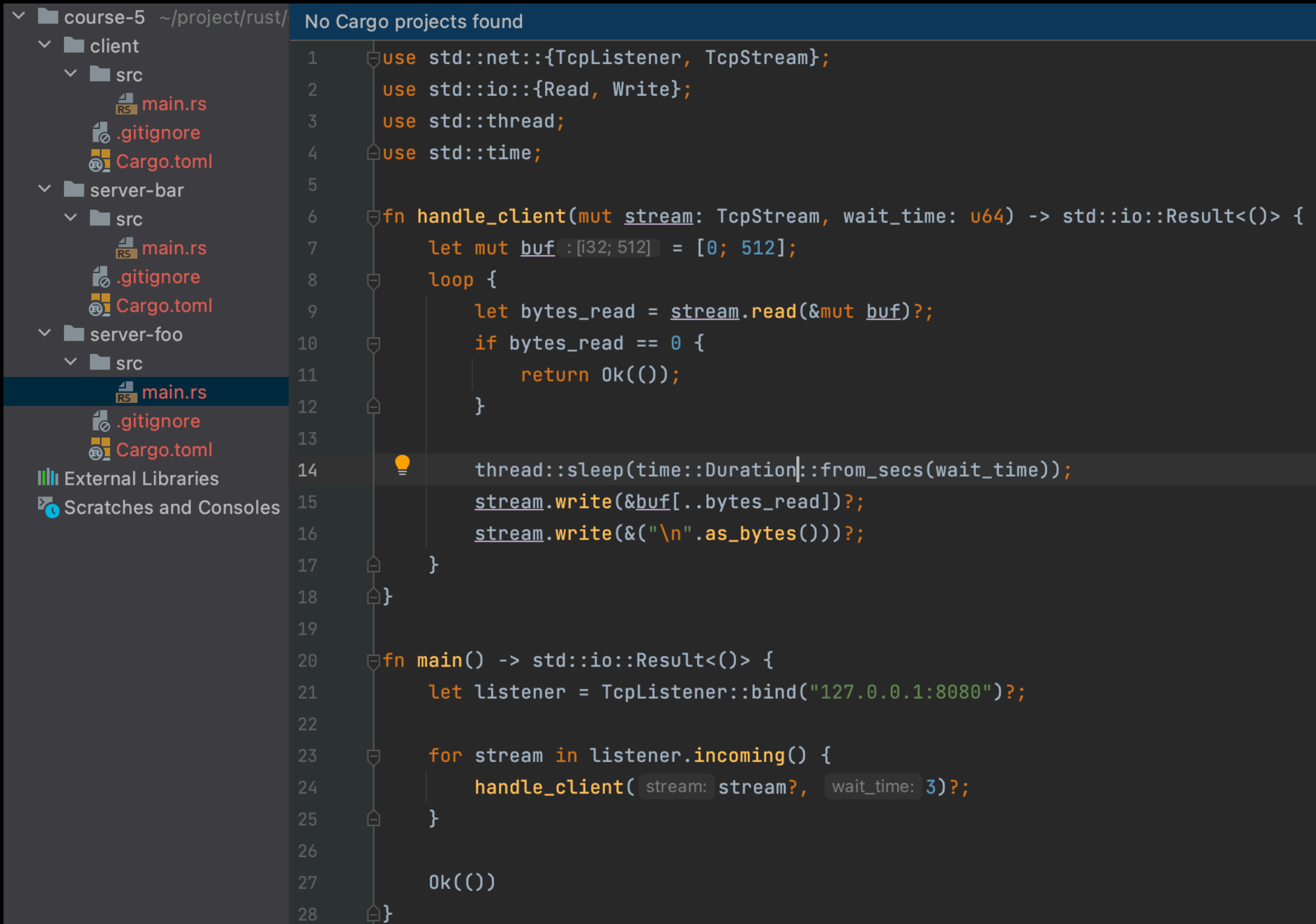


## 为什么需要异步

通过官方教程的一个例子, 来理解一下

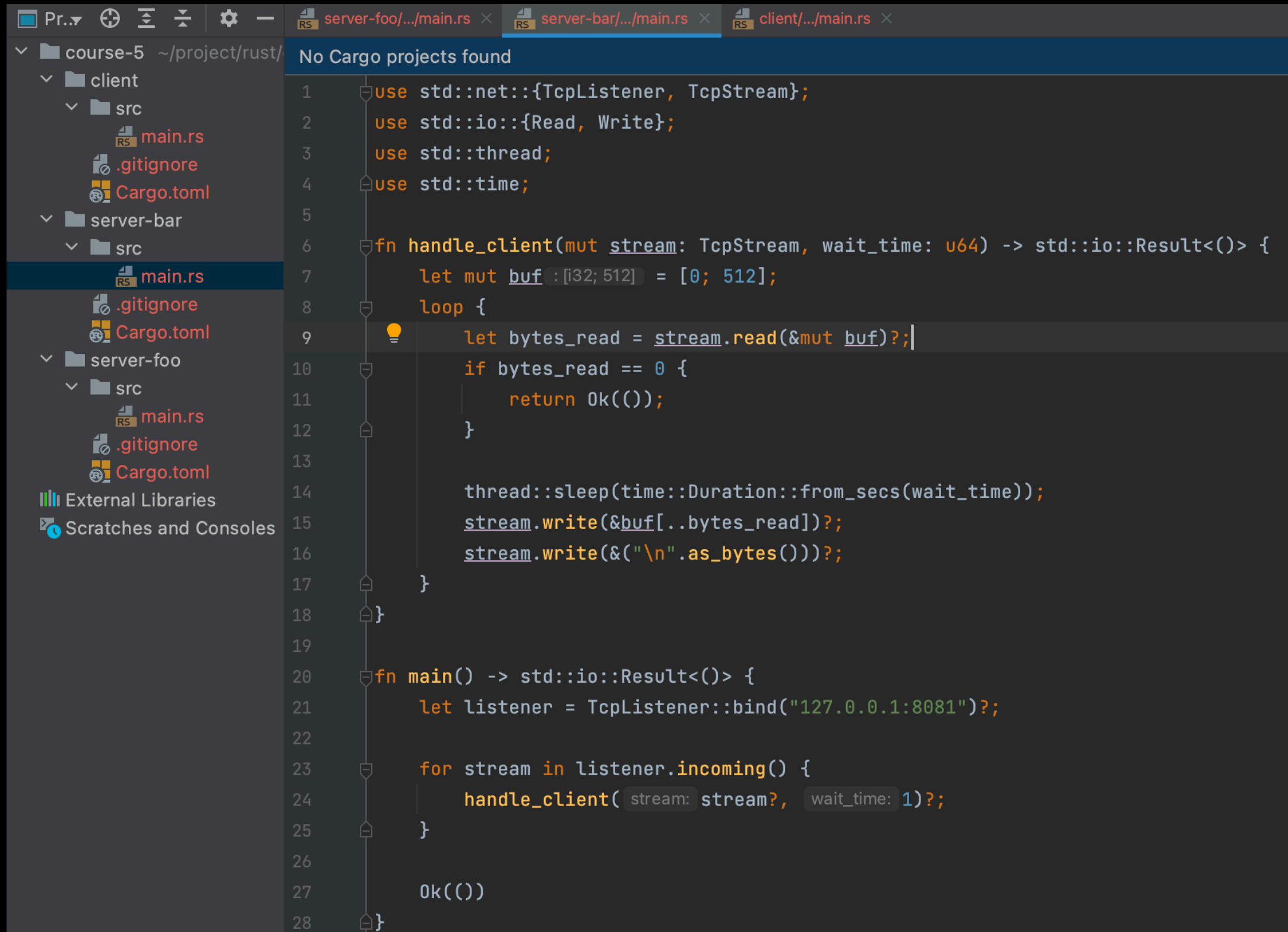


# 为什么需要异步



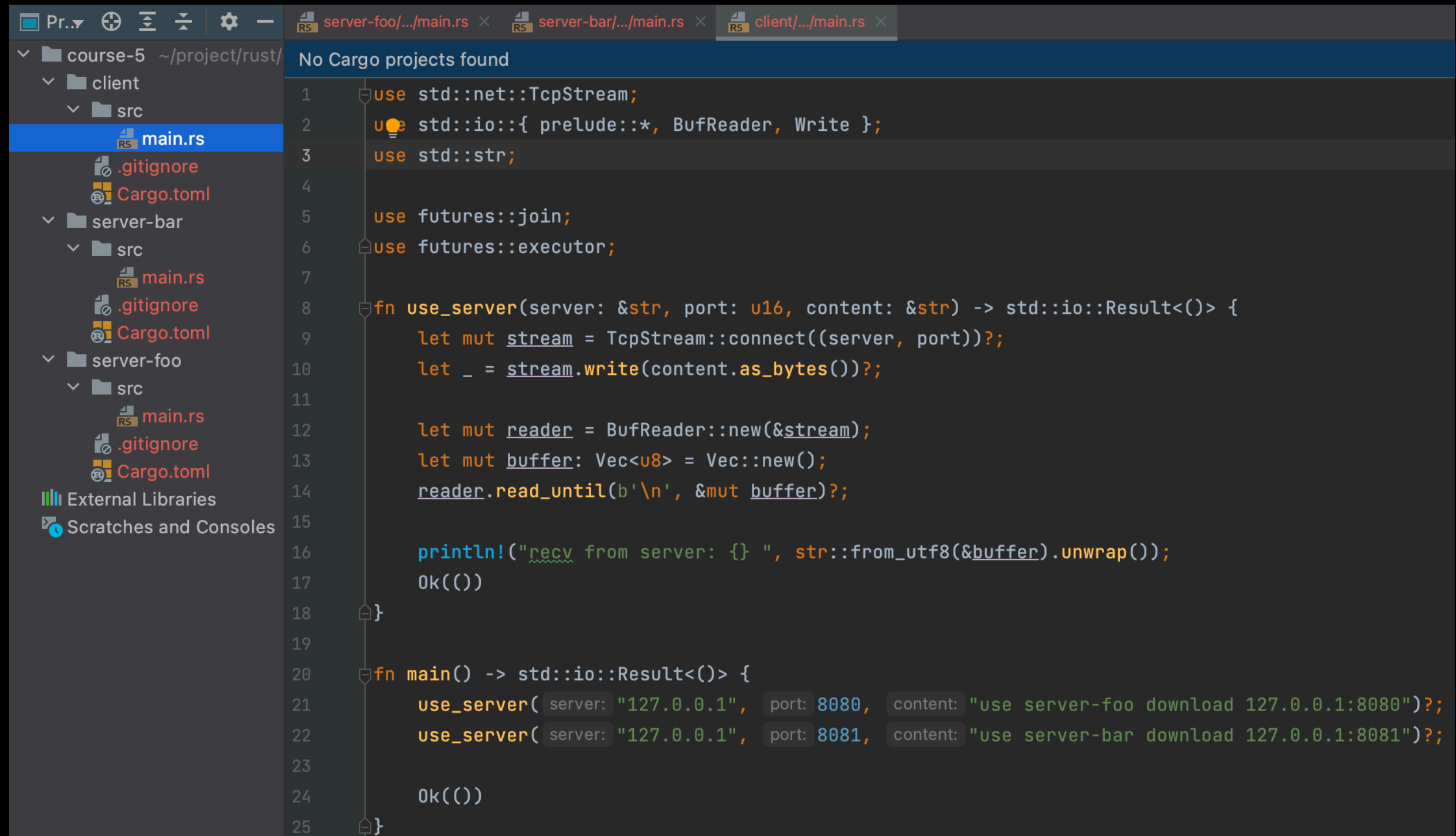
```
1 use std::net::{TcpListener, TcpStream};
2 use std::io::{Read, Write};
3 use std::thread;
4 use std::time;
5
6 fn handle_client(mut stream: TcpStream, wait_time: u64) -> std::io::Result<()> {
7     let mut buf : [i32; 512] = [0; 512];
8     loop {
9         let bytes_read = stream.read(&mut buf)?;
10        if bytes_read == 0 {
11            return Ok(());
12        }
13
14        thread::sleep(time::Duration::from_secs(wait_time));
15        stream.write(&buf[..bytes_read])?;
16        stream.write(&("\n".as_bytes()))?;
17    }
18 }
19
20 fn main() -> std::io::Result<()> {
21     let listener = TcpListener::bind("127.0.0.1:8080")?;
22
23     for stream in listener.incoming() {
24         handle_client( stream: stream?, wait_time: 3)?;
25     }
26
27     Ok(())
28 }
```

# 为什么需要异步



```
server-foo/.../main.rs x server-bar/.../main.rs x client/.../main.rs x
No Cargo projects found
1 use std::net::{TcpListener, TcpStream};
2 use std::io::{Read, Write};
3 use std::thread;
4 use std::time;
5
6 fn handle_client(mut stream: TcpStream, wait_time: u64) -> std::io::Result<()> {
7     let mut buf : [i32; 512] = [0; 512];
8     loop {
9         let bytes_read = stream.read(&mut buf)?;
10        if bytes_read == 0 {
11            return Ok(());
12        }
13
14        thread::sleep(time::Duration::from_secs(wait_time));
15        stream.write(&buf[..bytes_read])?;
16        stream.write(&("\n".as_bytes()))?;
17    }
18 }
19
20 fn main() -> std::io::Result<()> {
21     let listener = TcpListener::bind("127.0.0.1:8081")?;
22
23     for stream in listener.incoming() {
24         handle_client( stream: stream?, wait_time: 1)?;
25     }
26
27     Ok(())
28 }
```

# 为什么需要异步

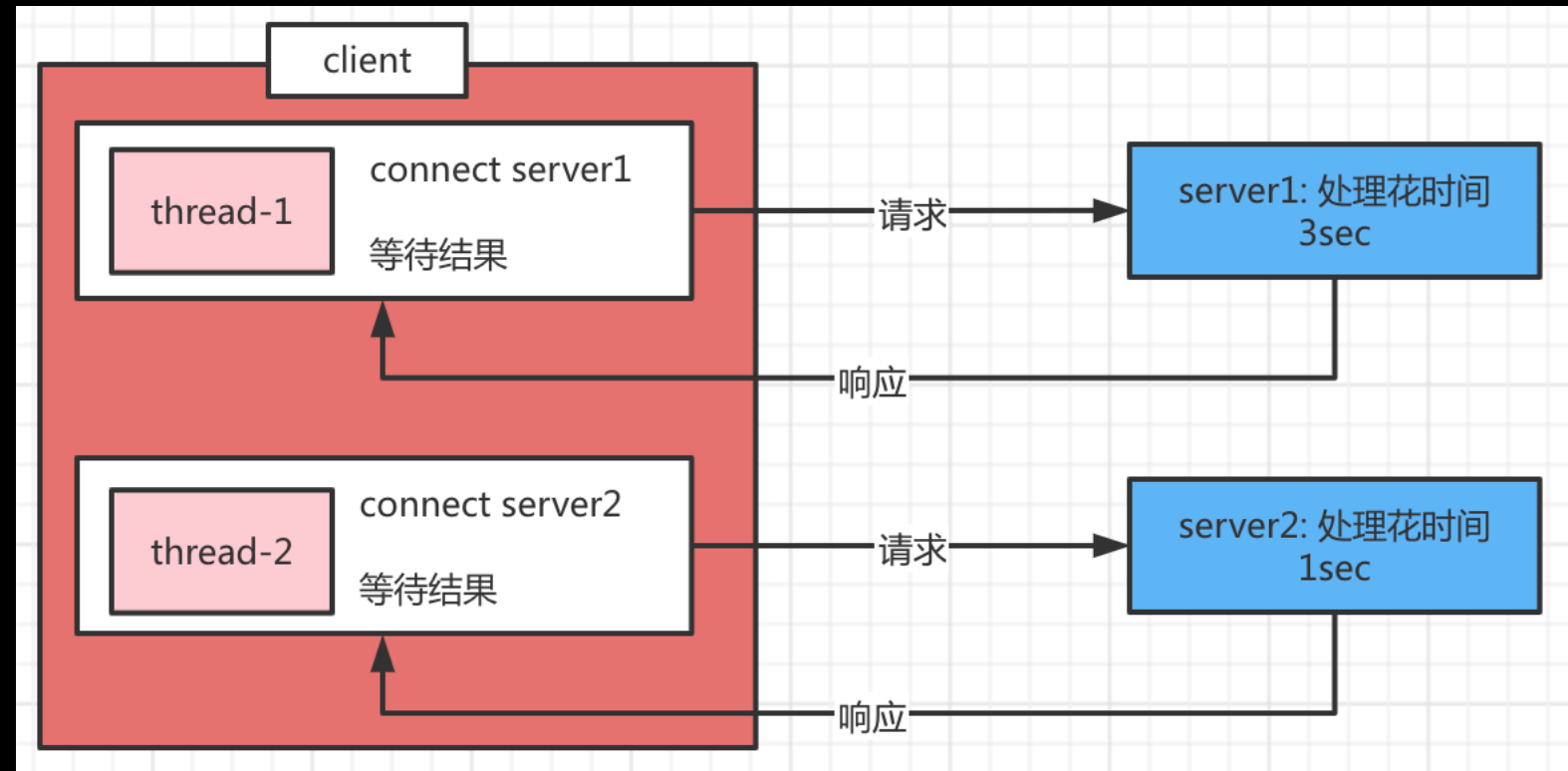
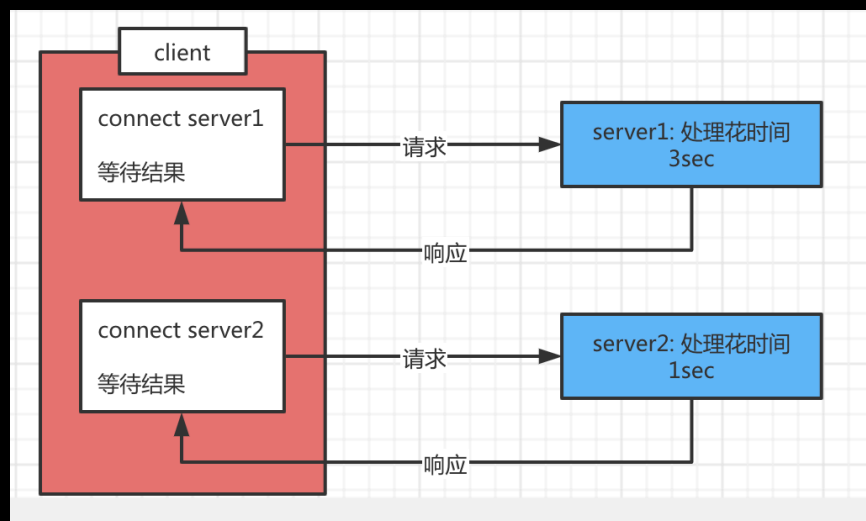
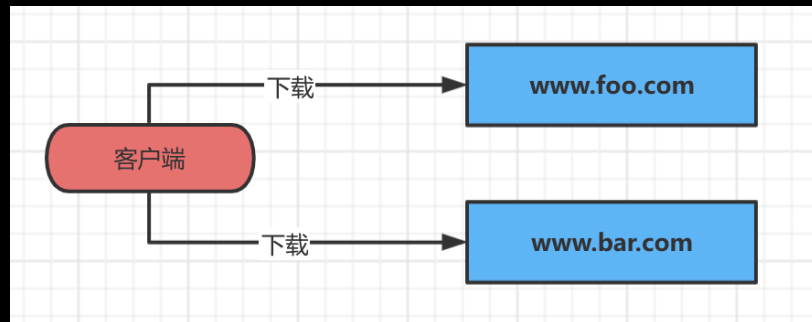


```
1 use std::net::TcpStream;
2 use std::io::{prelude::*, BufReader, Write};
3 use std::str;
4
5 use futures::join;
6 use futures::executor;
7
8 fn use_server(server: &str, port: u16, content: &str) -> std::io::Result<()> {
9     let mut stream = TcpStream::connect((server, port))?;
10    let _ = stream.write(content.as_bytes())?;
11
12    let mut reader = BufReader::new(&stream);
13    let mut buffer: Vec<u8> = Vec::new();
14    reader.read_until(b'\n', &mut buffer)?;
15
16    println!("recv from server: {} ", str::from_utf8(&buffer).unwrap());
17    Ok(())
18 }
19
20 fn main() -> std::io::Result<()> {
21     use_server(server: "127.0.0.1", port: 8080, content: "use server-foo download 127.0.0.1:8080")?;
22     use_server(server: "127.0.0.1", port: 8081, content: "use server-bar download 127.0.0.1:8081")?;
23
24     Ok(())
25 }
```



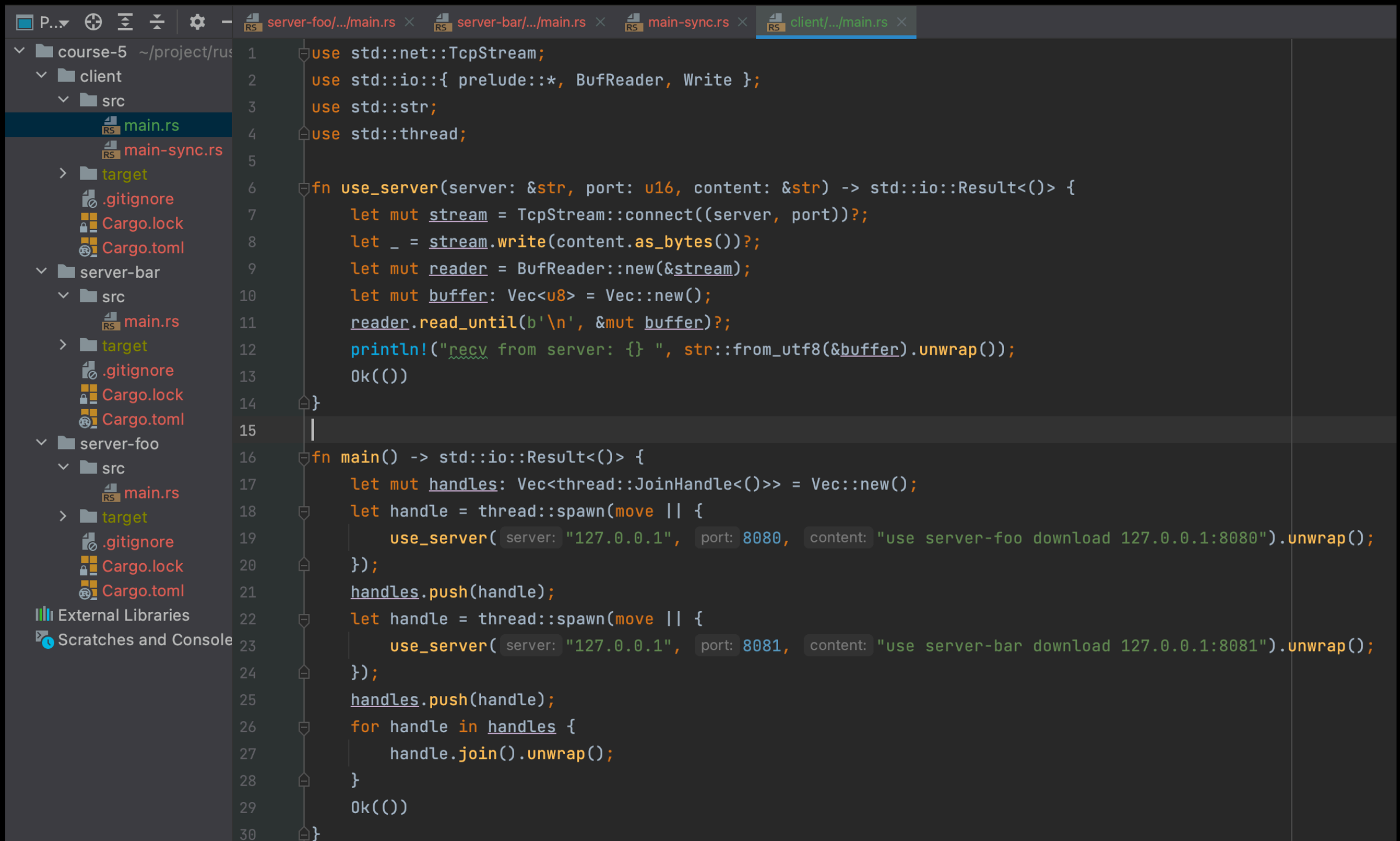
# 为什么需要异步

通过官方教程的一个例子, 来理解一下



# 为什么需要异步

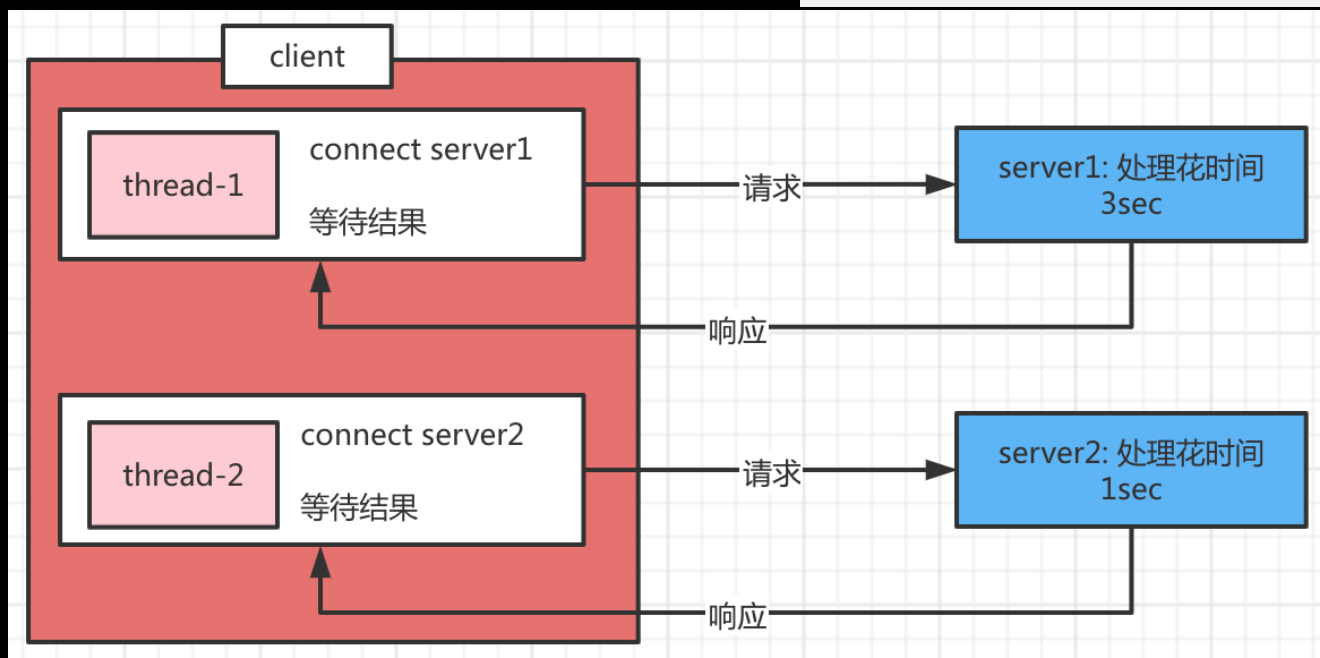
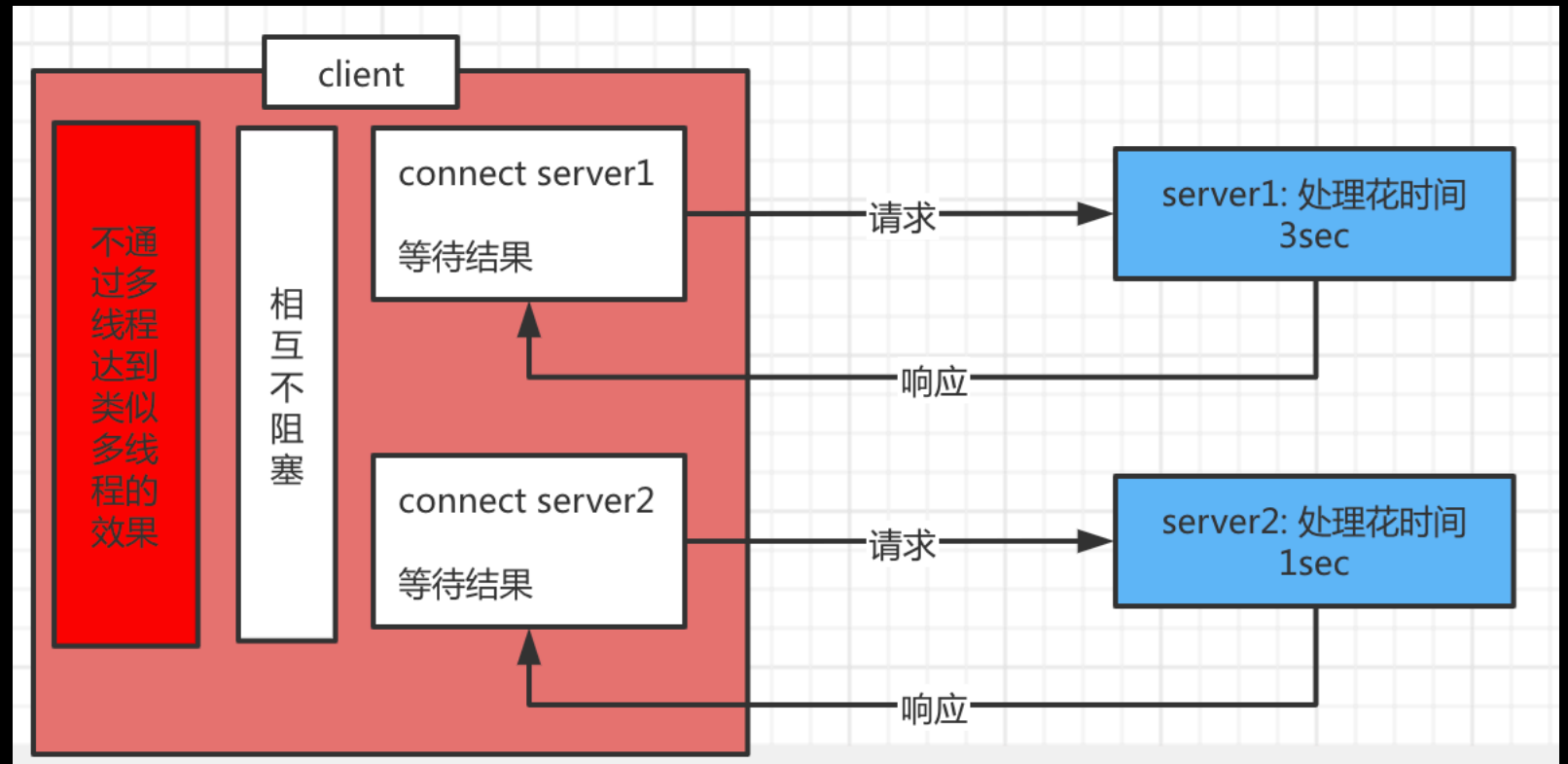
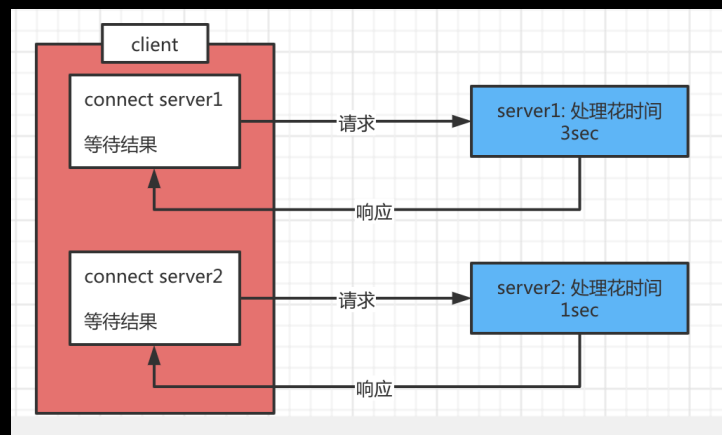
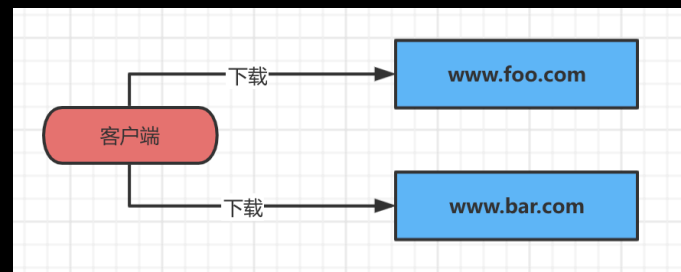
通过官方教程的一个例子, 来理解一下



```
1 use std::net::TcpStream;
2 use std::io::{prelude::*, BufReader, Write};
3 use std::str;
4 use std::thread;
5
6 fn use_server(server: &str, port: u16, content: &str) -> std::io::Result<()> {
7     let mut stream = TcpStream::connect((server, port))?;
8     let _ = stream.write(content.as_bytes())?;
9     let mut reader = BufReader::new(&stream);
10    let mut buffer: Vec<u8> = Vec::new();
11    reader.read_until(b'\n', &mut buffer)?;
12    println!("recv from server: {}", str::from_utf8(&buffer).unwrap());
13    Ok(())
14}
15
16 fn main() -> std::io::Result<()> {
17     let mut handles: Vec<thread::JoinHandle<()>> = Vec::new();
18     let handle = thread::spawn(move || {
19         use_server(server: "127.0.0.1", port: 8080, content: "use server-foo download 127.0.0.1:8080").unwrap();
20     });
21     handles.push(handle);
22     let handle = thread::spawn(move || {
23         use_server(server: "127.0.0.1", port: 8081, content: "use server-bar download 127.0.0.1:8081").unwrap();
24     });
25     handles.push(handle);
26     for handle in handles {
27         handle.join().unwrap();
28     }
29     Ok(())
30 }
```

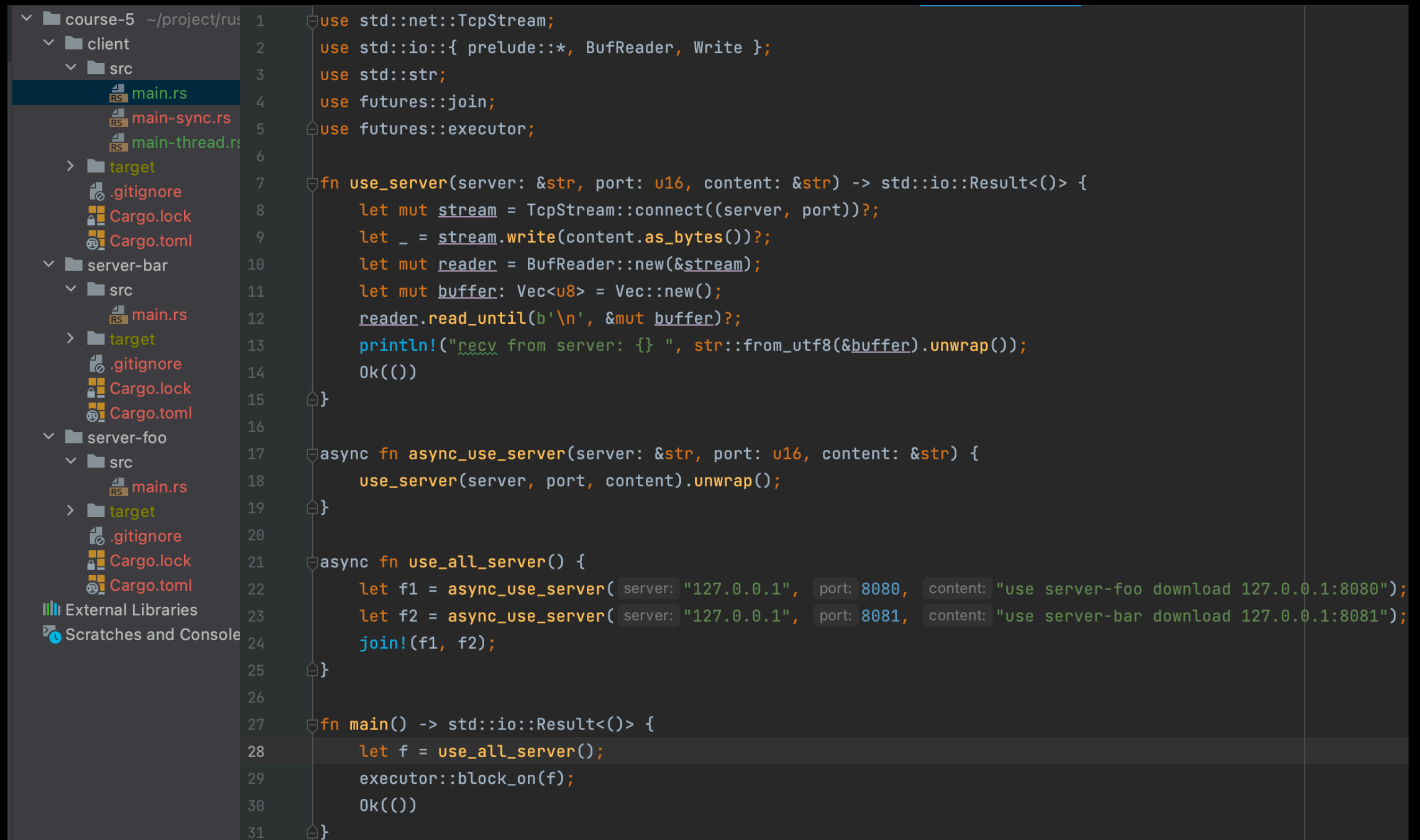
# 为什么需要异步

通过官方教程的一个例子, 来理解一下



# 为什么需要异步

通过官方教程的一个例子, 来理解一下



```
1 use std::net::TcpStream;
2 use std::io::{prelude::*, BufReader, Write};
3 use std::str;
4 use futures::join;
5 use futures::executor;
6
7 fn use_server(server: &str, port: u16, content: &str) -> std::io::Result<()> {
8     let mut stream = TcpStream::connect((server, port))?;
9     let _ = stream.write(content.as_bytes())?;
10    let mut reader = BufReader::new(&stream);
11    let mut buffer: Vec<u8> = Vec::new();
12    reader.read_until(b'\n', &mut buffer)?;
13    println!("recv from server: {} ", str::from_utf8(&buffer).unwrap());
14    Ok(())
15}
16
17 async fn async_use_server(server: &str, port: u16, content: &str) {
18     use_server(server, port, content).unwrap();
19}
20
21 async fn use_all_server() {
22     let f1 = async_use_server(server: "127.0.0.1", port: 8080, content: "use server-foo download 127.0.0.1:8080");
23     let f2 = async_use_server(server: "127.0.0.1", port: 8081, content: "use server-bar download 127.0.0.1:8081");
24     join!(f1, f2);
25}
26
27 fn main() -> std::io::Result<()> {
28     let f = use_all_server();
29     executor::block_on(f);
30     Ok(())
31}
```

## async\await 简单介绍

async/await是Rust编写异步的内置工具

async将一个代码块转化为实现了future特征的状态机  
那么转化为future后有什么作用呢？

## async\await 简单介绍

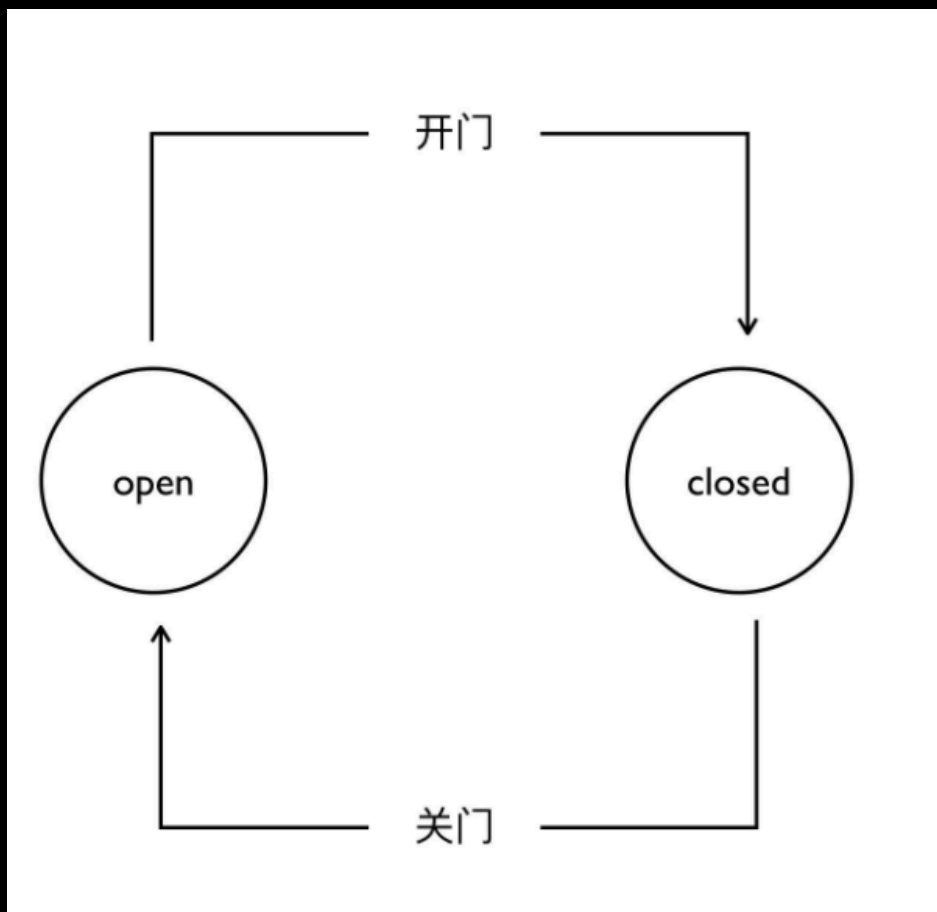
async/await是Rust编写异步的内置工具

async将一个代码块转化为实现了future特征的状态机.

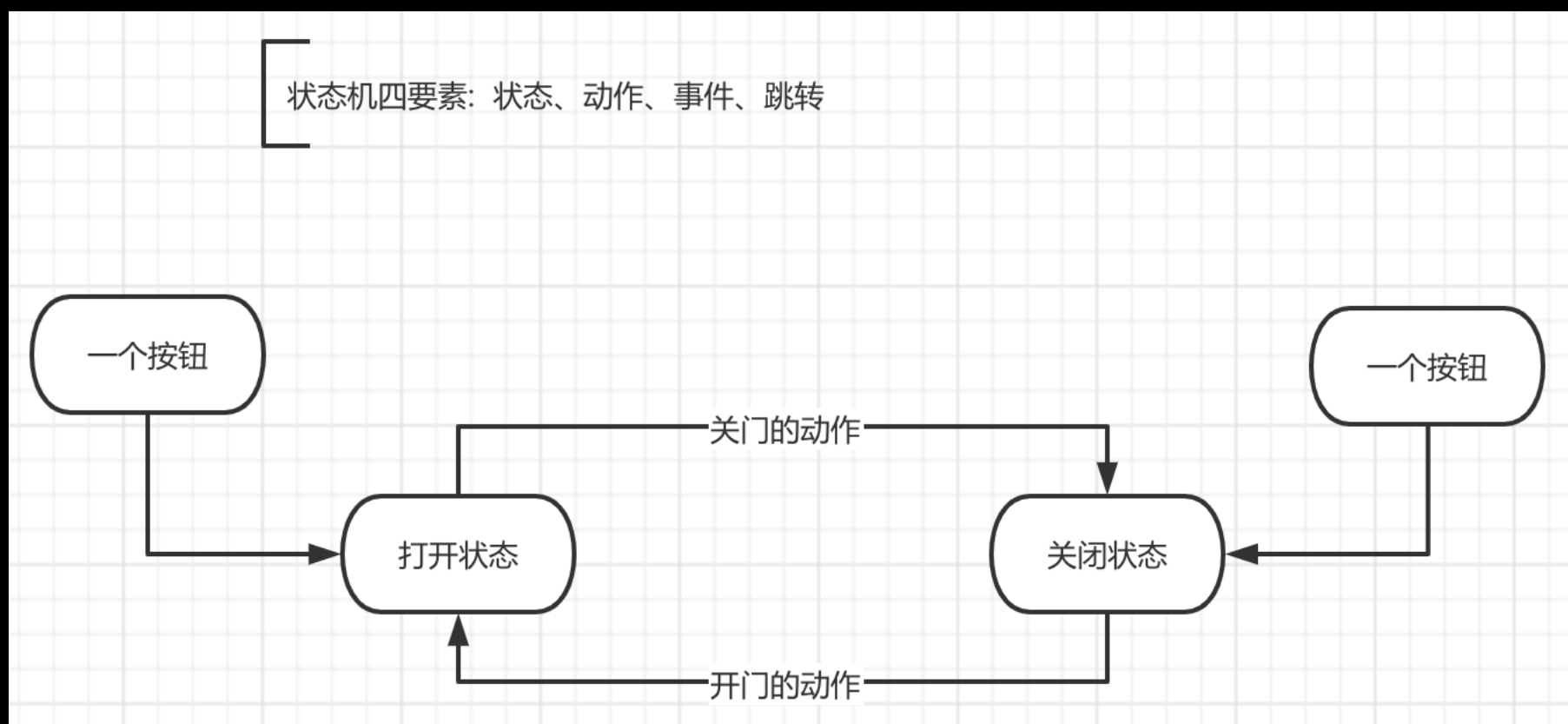
那么转化为future后有什么作用呢?

-> 在同步方法中调用阻塞函数(async转化的函数), 会阻塞整个线程, 但是, 阻塞的future会让出线程控制权, 允许其它future运行.

## 什么是状态机



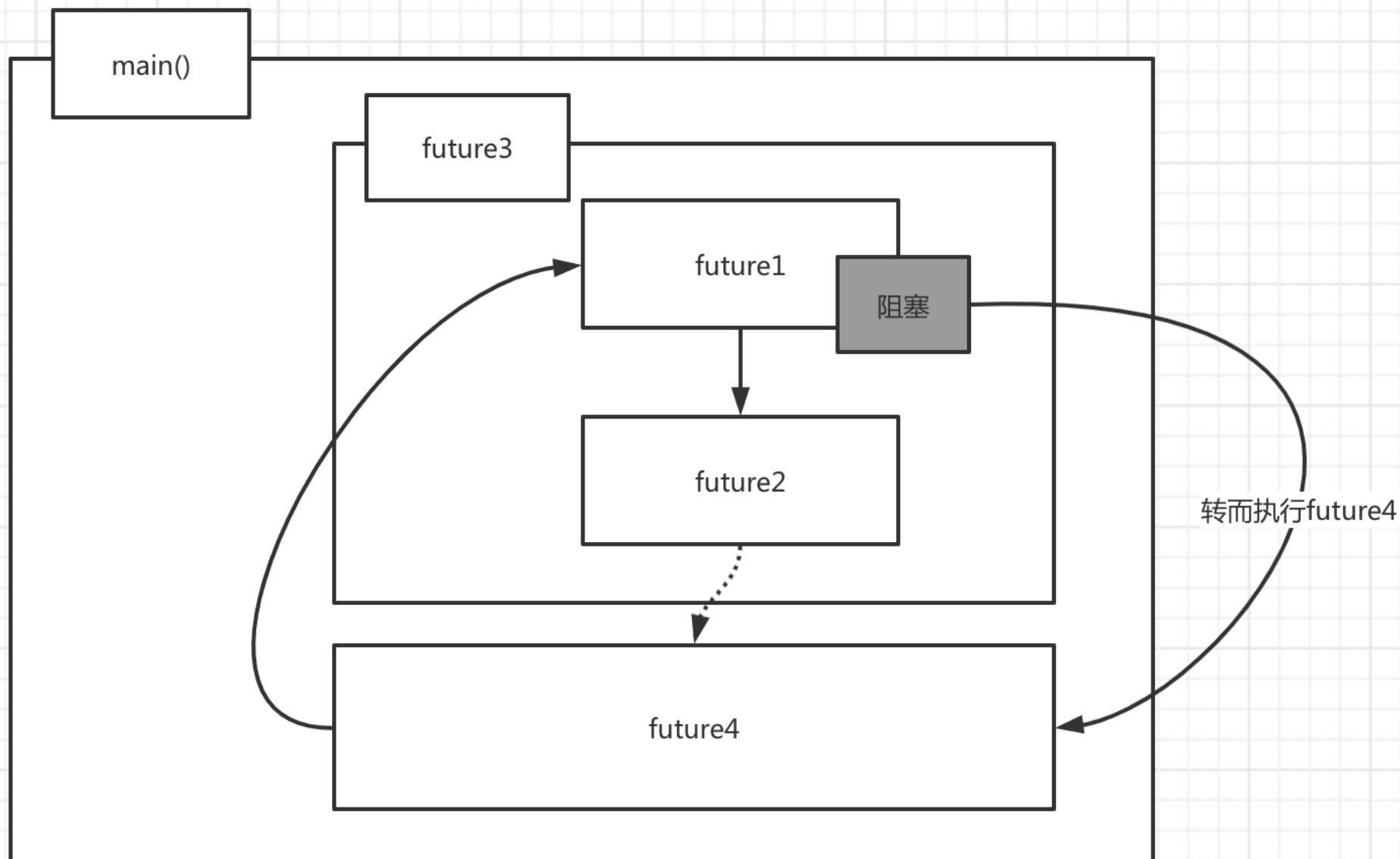
# 什么是状态机



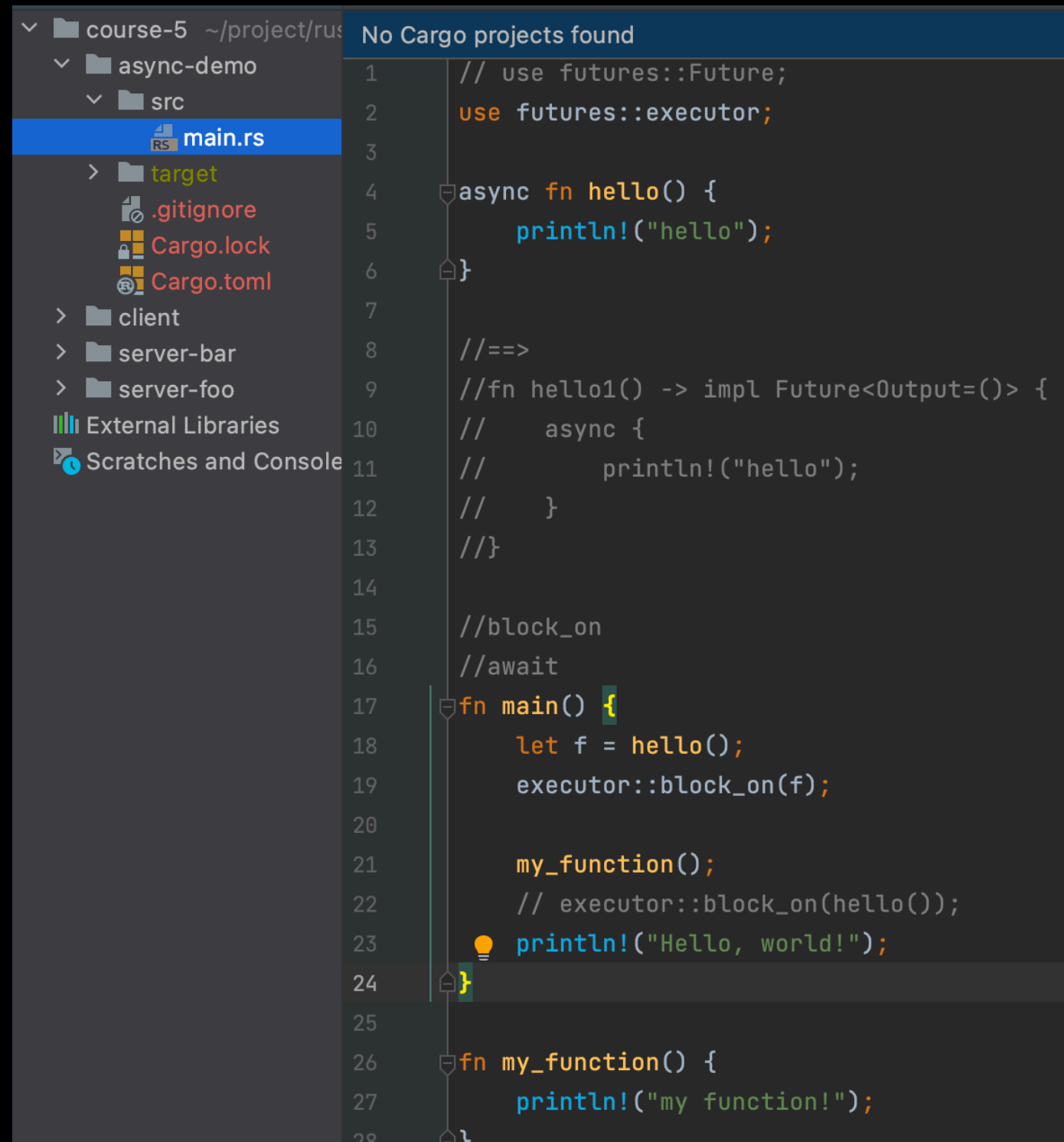


## 通过一个图来说明 那么转化为future后有什么作用呢?

在同步方法中调用阻塞函数(async转化的函数), 会阻塞整个线程, 但是, 阻塞的future会让出线程控制权, 允许其它future运行.



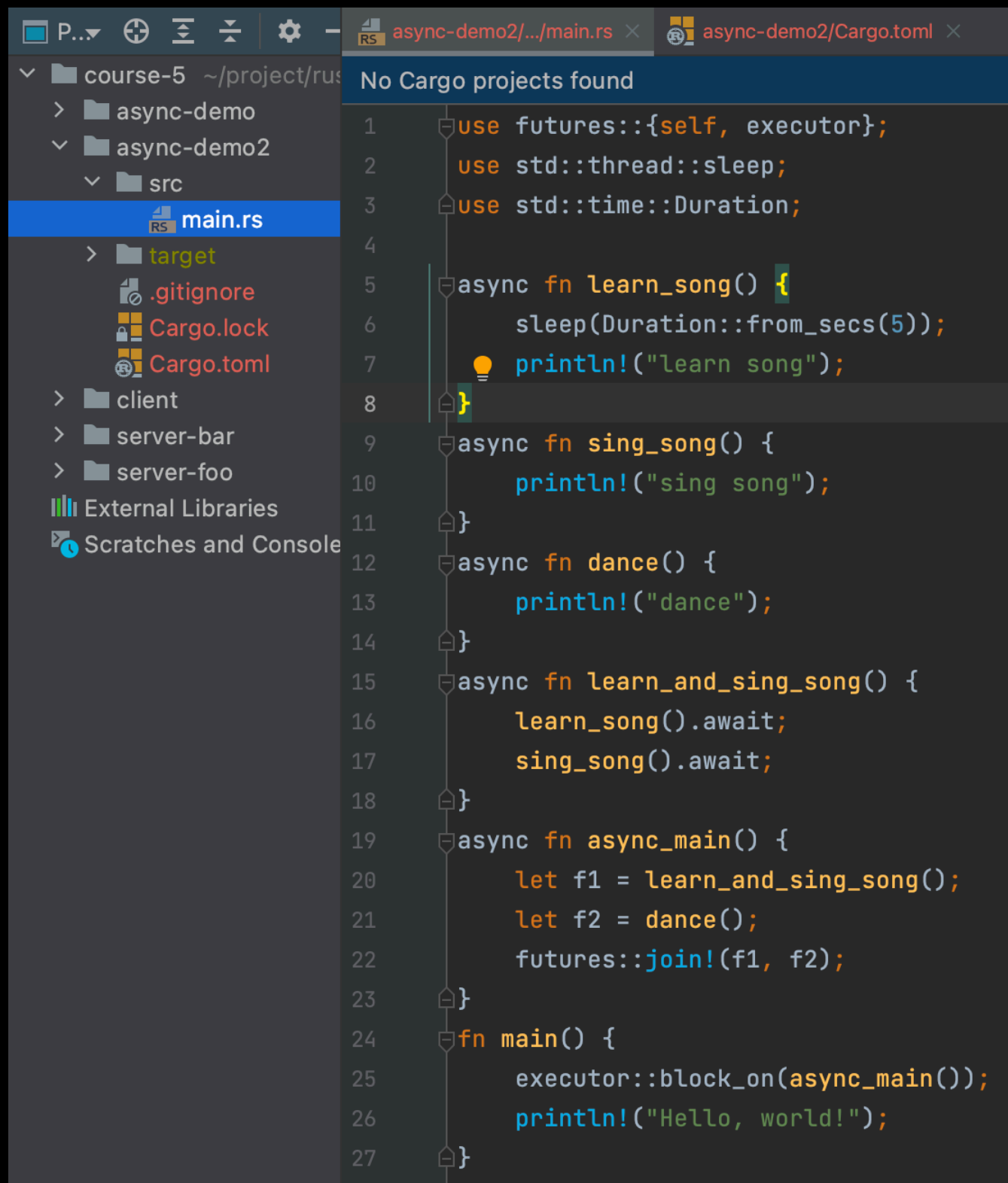
## 通过一段代码, 了解async关键字的作用会



```
1 // use futures::Future;
2 use futures::executor;
3
4 async fn hello() {
5     println!("hello");
6 }
7
8 //==>
9 //fn hello1() -> impl Future<Output=()> {
10 //    async {
11 //        println!("hello");
12 //    }
13 //}
14
15 //block_on
16 //await
17 fn main() {
18     let f = hello();
19     executor::block_on(f);
20
21     my_function();
22     // executor::block_on(hello());
23     println!("Hello, world!");
24 }
25
26 fn my_function() {
27     println!("my function!");
28 }
```

## 通过block\_on来执行

## 通过await来执行

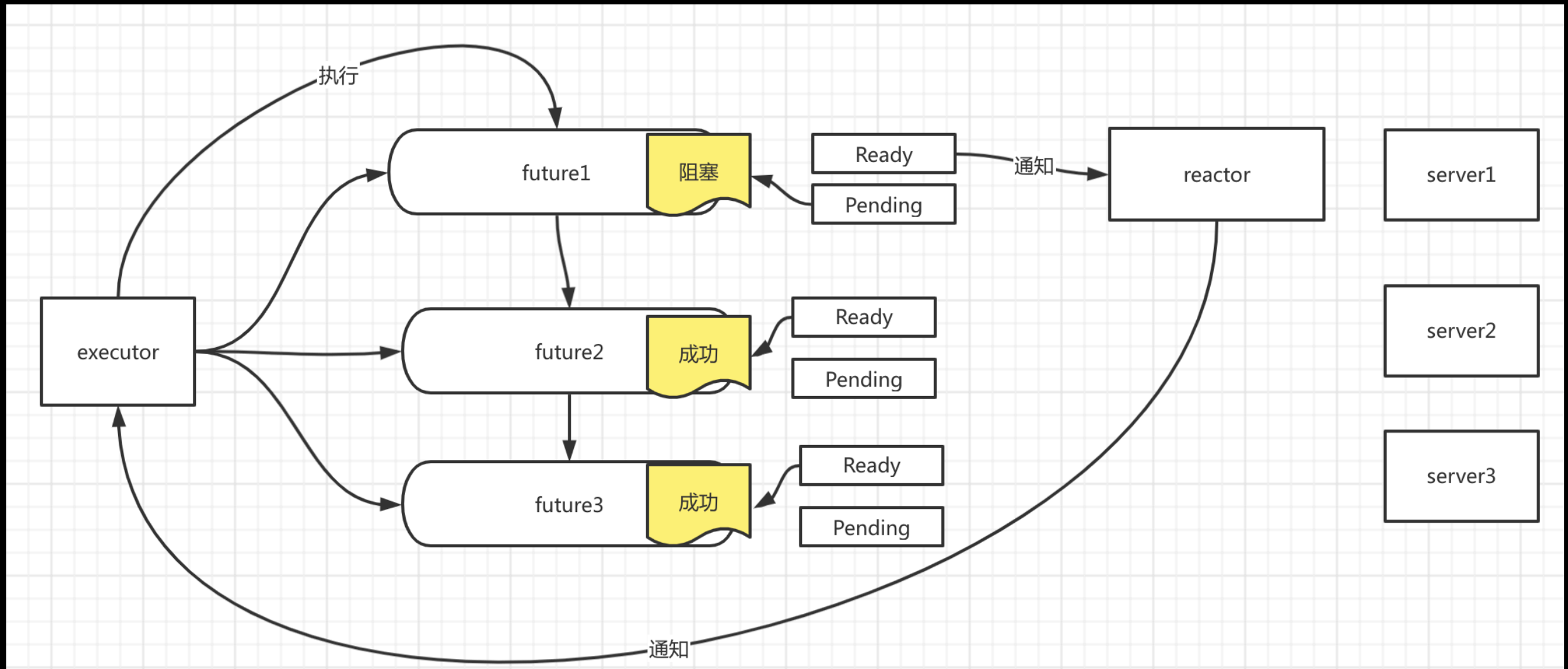


The screenshot shows an IDE with a file explorer on the left and a code editor on the right. The file explorer shows a project structure with folders like 'course-5', 'async-demo', 'async-demo2', and 'src'. The 'main.rs' file is selected in the 'src' folder. The code editor displays the following Rust code:

```
1 use futures::{self, executor};
2 use std::thread::sleep;
3 use std::time::Duration;
4
5 async fn learn_song() {
6     sleep(Duration::from_secs(5));
7     println!("learn song");
8 }
9
10 async fn sing_song() {
11     println!("sing song");
12 }
13
14 async fn dance() {
15     println!("dance");
16 }
17
18 async fn learn_and_sing_song() {
19     learn_song().await;
20     sing_song().await;
21 }
22
23 async fn async_main() {
24     let f1 = learn_and_sing_song();
25     let f2 = dance();
26     futures::join!(f1, f2);
27 }
28
29 fn main() {
30     executor::block_on(async_main());
31     println!("Hello, world!");
32 }
```

## join!语法

- 异步编程模型



- 带领大家实现一个简化版的Rust异步编程框架

# QA环节

加群一起交流Rust & Datafuse

