



38 | 异步处理：Future是什么？它和async/await是什么关系？

2021-11-26 陈天

《陈天·Rust 编程第一课》

[课程介绍 >](#)



讲述：陈天

时长 20:36 大小 18.87M

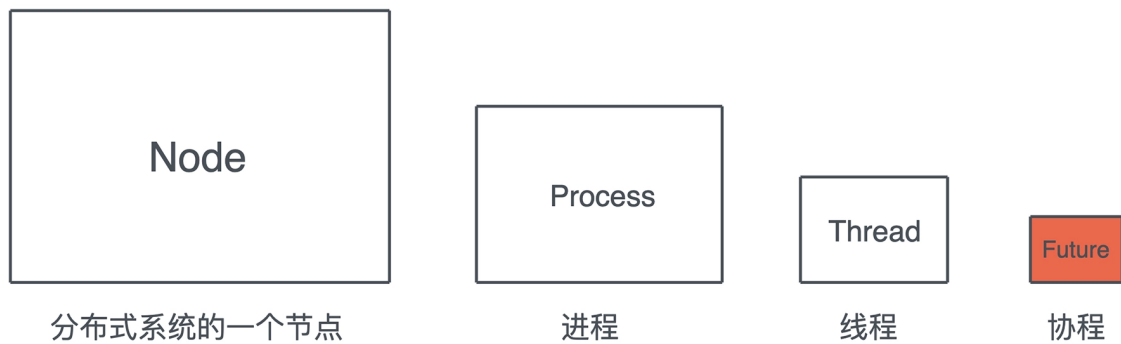


你好，我是陈天。

通过前几讲的学习，我们对并发处理，尤其是常用的并发原语，有了一个比较清晰的认识。并发原语是并发任务之间同步的手段，今天我们要学习的 Future 以及在更高层次上处理 Future 的 async/await，是**产生和运行并发任务**的手段。

不过产生和运行并发任务的手段有很多，async/await 只是其中之一。在一个分布式系统中，并发任务可以运行在系统的某个节点上；在某个节点上，并发任务又可以运行在多个进程中；而在某个进程中，并发任务可以运行在多个线程中；在某个（些）线程上，并发任务可以运行在多个 Promise / Future / Goroutine / Erlang process 这样的协程上。

它们的粒度从大到小如图所示：



在之前的课程里，我们大量应用了线程这种并发工具，在 kv server 的构建过程中，也通过 async/await 用到了 Future 这样的无栈协程。

其实 Rust 的 Future 跟 JavaScript 的 Promise 非常类似。

如果你熟悉 JavaScript，应该熟悉 Promise 的概念，[02](#)也简单讲过，它代表了**在未来的某个时刻才能得到的结果的值**，Promise 一般存在三个状态；

1. 初始状态，Promise 还未运行；
2. 等待（pending）状态，Promise 已运行，但还未结束；
3. 结束状态，Promise 成功解析出一个值，或者执行失败。

只不过 JavaScript 的 Promise 和线程类似，一旦创建就开始执行，对 Promise await 只是为了“等待”并获取解析出来的值；而 Rust 的 Future，只有在主动 await 后才开始执行。

讲到这里估计你也看出来了，谈 Future 的时候，我们总会谈到 async/await。一般而言，**async 定义了一个可以并发执行的任务，而 await 则触发这个任务并发执行**。大多数语言，包括 Rust，async/await 都是一个语法糖（syntactic sugar），它们使用状态机将 Promise/Future 这样的结构包装起来进行处理。

这一讲我们先把内部的实现放在一边，主要聊 Future/async/await 的基本概念和使用方法，下一讲再来详细介绍它们的原理。

为什么需要 Future ？


首先，谈一谈为什么需要 Future 这样的并发结构。

在 Future 出现之前，我们的 Rust 代码都是同步的。也就是说，当你执行一个函数，CPU 处理完函数中的每一个指令才会返回。如果这个函数里有 IO 的操作，实际上，操作系统会把函数对应的线程挂起，放在一个等待队列中，直到 IO 操作完成，才恢复这个线程，并从挂起的位置继续执行下去。

这个模型非常简单直观，代码是一行一行执行的，开发者并不需要考虑哪些操作会阻塞，哪些不会，只关心他的业务逻辑就好。

然而，随着 CPU 技术的不断发展，新世纪应用软件的主要矛盾不再是 CPU 算力不足，而是**过于充沛的 CPU 算力和提升缓慢的 IO 速度之间的矛盾**。如果有大量的 IO 操作，你的程序大部分时间并没有在运算，而是在不断地等待 IO。

我们来看一个例子（[🔗代码](#)）：

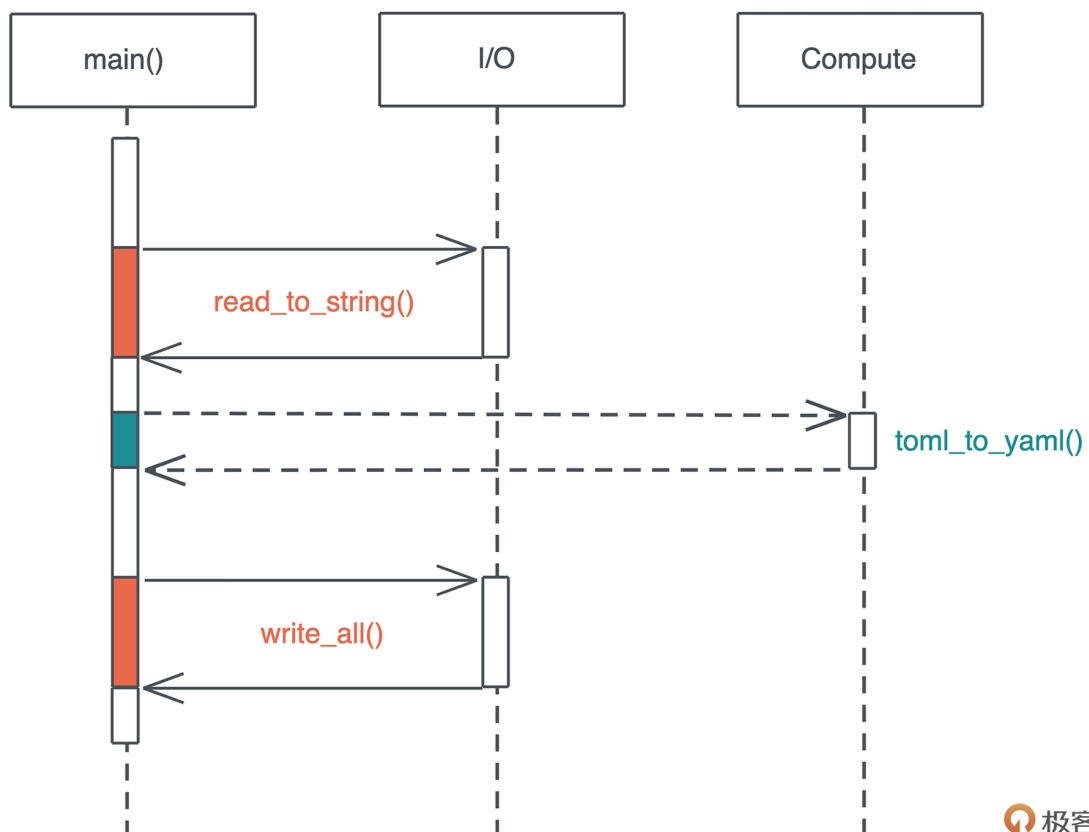
 复制代码

```
1 use anyhow::Result;
2 use serde_yaml::Value;
3 use std::fs;
4
5 fn main() -> Result<()> {
6     // 读取 Cargo.toml , IO 操作 1
7     let content1 = fs::read_to_string("./Cargo.toml")?;
8     // 读取 Cargo.lock , IO 操作 2
9     let content2 = fs::read_to_string("./Cargo.lock")?;
10
11     // 计算
12     let yaml1 = toml2yaml(&content1)?;
13     let yaml2 = toml2yaml(&content2)?;
14
15     // 写入 /tmp/Cargo.yml , IO 操作 3
16     fs::write("/tmp/Cargo.yml", &yaml1)?;
17     // 写入 /tmp/Cargo.lock , IO 操作 4
18     fs::write("/tmp/Cargo.lock", &yaml2)?;
19 }
```

```
20 // 打印
21 println!("{}", yaml1);
22 println!("{}", yaml2);
23
24 Ok(())
25 }
26
27 fn toml2yaml(content: &str) -> Result<String> {
28     let value: Value = toml::from_str(&content)?;
29     Ok(serde_yaml::to_string(&value)?)
30 }
```

这段代码读取 Cargo.toml 和 Cargo.lock 将其转换成 yaml，再分别写入到 /tmp 下。

虽然说这段代码的逻辑并没有问题，但性能有很大的问题。在读 Cargo.toml 时，整个主线程被阻塞，直到 Cargo.toml 读完，才能继续读下一个待处理的文件。整个主线程，只有在运行 toml2yaml 的时间片内，才真正在执行计算任务，之前的读取文件以及之后的写入文件，CPU 都在闲置。



当然，你会辩解，在读文件的过程中，我们不得不等待，因为 toml2yaml 函数的执行有赖于读取文件的结果。嗯没错，但是，这里还有很大的 CPU 浪费：我们读完第一个文件才开

始读第二个文件, 有没有可能两个文件同时读取呢? 这样总共等待的时间是 $\max(\text{time_for_file1}, \text{time_for_file2})$, 而非 $\text{time_for_file1} + \text{time_for_file2}$ 。

这并不难, 我们可以把文件读取和写入的操作放入单独的线程中执行, 比如 ([🔗代码](#)) :

[📄 复制代码](#)

```
1 use anyhow::{anyhow, Result};
2 use serde_yaml::Value;
3 use std::{
4     fs,
5     thread::{self, JoinHandle},
6 };
7
8 /// 包装一下 JoinHandle, 这样可以提供额外的方法
9 struct MyJoinHandle<T> (JoinHandle<Result<T>>);
10
11 impl<T> MyJoinHandle<T> {
12     /// 等待 thread 执行完 (类似 await)
13     pub fn thread_await(self) -> Result<T> {
14         self.0.join().map_err(|_| anyhow!("failed"))?
15     }
16 }
17
18 fn main() -> Result<()> {
19     // 读取 Cargo.toml, IO 操作 1
20     let t1 = thread_read("./Cargo.toml");
21     // 读取 Cargo.lock, IO 操作 2
22     let t2 = thread_read("./Cargo.lock");
23
24     let content1 = t1.thread_await()?;
25     let content2 = t2.thread_await()?;
26
27     // 计算
28     let yaml1 = toml2yaml(&content1)?;
29     let yaml2 = toml2yaml(&content2)?;
30
31     // 写入 /tmp/Cargo.yml, IO 操作 3
32     let t3 = thread_write("/tmp/Cargo.yml", yaml1);
33     // 写入 /tmp/Cargo.lock, IO 操作 4
34     let t4 = thread_write("/tmp/Cargo.lock", yaml2);
35
36     let yaml1 = t3.thread_await()?;
37     let yaml2 = t4.thread_await()?;
38
39     fs::write("/tmp/Cargo.yml", &yaml1)?;
40     fs::write("/tmp/Cargo.lock", &yaml2)?;
41
42     // 打印
43     println!("{}", yaml1);
```

```
44     println!("{}", yaml2);
45
46     Ok(())
47 }
48
49 fn thread_read(filename: &'static str) -> MyJoinHandle<String> {
50     let handle = thread::spawn(move || {
51         let s = fs::read_to_string(filename)?;
52         Ok:::<_, anyhow::Error>(s)
53     });
54     MyJoinHandle(handle)
55 }
56
57 fn thread_write(filename: &'static str, content: String) -> MyJoinHandle<Strin
58     let handle = thread::spawn(move || {
59         fs::write(filename, &content)?;
60         Ok:::<_, anyhow::Error>(content)
61     });
62     MyJoinHandle(handle)
63 }
64
65 fn toml2yaml(content: &str) -> Result<String> {
66     let value: Value = toml::from_str(&content)?;
67     Ok(serde_yaml::to_string(&value)?)
68 }
```

这样，读取或者写入多个文件的过程并发执行，使等待的时间大大缩短。

但是，如果要同时读取 100 个文件呢？显然，创建 100 个线程来做这样的事情不是一个好主意。在操作系统中，线程的数量是有限的，创建 / 阻塞 / 唤醒 / 销毁线程，都涉及不少的动作，每个线程也都会被分配一个不小的调用栈，所以从 CPU 和内存的角度来看，**创建过多的线程会大大增加系统的开销。**

其实，绝大多数操作系统对 I/O 操作提供了非阻塞接口，也就是说，你可以发起一个读取的指令，自己处理类似 EWOULDBLOCK 这样的错误码，来更好地在同一个线程中处理多个文件的 IO，而不是依赖操作系统通过调度帮你完成这件事。

不过这样就意味着，你需要定义合适的数据结构来追踪每个文件的读取，在用户态进行相应的调度，阻塞等待 IO 的数据结构的运行，让没有等待 IO 的数据结构得到机会使用 CPU，以及当 IO 操作结束后，恢复等待 IO 的数据结构的运行等等。这样的操作粒度更小，可以最大程度利用 CPU 资源。这就是类似 Future 这样的并发结构的主要用途。

然而，如果这么处理，我们需要在用户态做很多事情，包括处理 IO 任务的事件通知、创建 Future、合理地调度 Future。这些事情，统统交给开发者做显然是不合理的。所以，Rust 提供了相应处理手段 `async/await`：**`async` 来方便地生成 Future，`await` 来触发 Future 的调度和执行。**

我们看看，同样的任务，如何用 `async/await` 更高效地处理（[🔗 代码](#)）：

[📄 复制代码](#)

```
1 use anyhow::Result;
2 use serde_yaml::Value;
3 use tokio::{fs, try_join};
4
5 #[tokio::main]
6 async fn main() -> Result<()> {
7     // 读取 Cargo.toml, IO 操作 1
8     let f1 = fs::read_to_string("./Cargo.toml");
9     // 读取 Cargo.lock, IO 操作 2
10    let f2 = fs::read_to_string("./Cargo.lock");
11    let (content1, content2) = try_join!(f1, f2)?;
12
13    // 计算
14    let yaml1 = toml2yaml(&content1)?;
15    let yaml2 = toml2yaml(&content2)?;
16
17    // 写入 /tmp/Cargo.yml, IO 操作 3
18    let f3 = fs::write("/tmp/Cargo.yml", &yaml1);
19    // 写入 /tmp/Cargo.lock, IO 操作 4
20    let f4 = fs::write("/tmp/Cargo.lock", &yaml2);
21    try_join!(f3, f4)?;
22
23    // 打印
24    println!("{}", yaml1);
25    println!("{}", yaml2);
26
27    Ok(())
28 }
29
30 fn toml2yaml(content: &str) -> Result<String> {
31     let value: Value = toml::from_str(&content)?;
32     Ok(serde_yaml::to_string(&value)?)
33 }
```

在这段代码里，我们使用了 `tokio::fs`，而不是 `std::fs`，`tokio::fs` 的文件操作都会返回一个 Future，然后可以 `join` 这些 Future，得到它们运行后的结果。`join / try_join` 是用来轮询

多个 Future 的宏，它会依次处理每个 Future，遇到阻塞就处理下一个，直到所有 Future 产生结果。

整个等待文件读取的时间是 `max(time_for_file1, time_for_file2)`，性能和使用线程的版本几乎一致，但是消耗的资源（主要是线程）要少很多。

建议你好好对比这三个版本的代码，写一写，运行一下，感受它们的处理逻辑。注意在最后的 `async/await` 的版本中，我们不能把代码写成这样：

[复制代码](#)

```
1 // 读取 Cargo.toml, IO 操作 1
2 let content1 = fs::read_to_string("./Cargo.toml").await?;
3 // 读取 Cargo.lock, IO 操作 2
4 let content1 = fs::read_to_string("./Cargo.lock").await?;
```

这样写的话，和第一版同步的版本没有区别，因为 `await` 会运行 Future 直到 Future 执行结束，所以依旧是先读取 `Cargo.toml`，再读取 `Cargo.lock`，并没有达到并发的效果。

深入了解


好，了解了 Future 在软件开发中的必要性，来深入研究一下 Future/async/await。

在前面代码撰写过程中，不知道你有没有发现，异步函数（`async fn`）的返回值是一个奇怪的 `impl Future<Output>` 的结构：

```
async fn main() -> Result<(), > {
    // 读取 Cargo.toml, IO 操作 1
    let f1: impl Future<Output = Result<..., >> = fs::read_to_string(path: "./Cargo.toml");
    // 读取 Cargo.lock, IO 操作 2
    let f2: impl Future<Output = Result<..., >> = fs::read_to_string(path: "./Cargo.lock");
    let (content1: String, content2: String) = try_join!(f1, f2)?;
```

我们知道，一般会用 `impl` 关键字为数据结构实现 trait，也就是说接在 `impl` 关键字后面的东西是一个 trait，所以，显然 Future 是一个 trait，并且还有一个关联类型 `Output`。

来看 [Future](#) 的定义：

 复制代码

```
1 pub trait Future {
2     type Output;
3     fn poll(self: Pin<&mut Self>, cx: &mut Context<'_,>) -> Poll<Self::Output>;
4 }
5
6 pub enum Poll<T> {
7     Ready(T),
8     Pending,
9 }
```

除了 `Output` 外，它还有一个 `poll()` 方法，这个方法返回 `Poll < Self::Output`。而 `Poll<T>` 是个 `enum`，包含 `Ready` 和 `Pending` 两个状态。显然，当 `Future` 返回 `Pending` 状态时，活还没干完，但干不下去了，需要阻塞一阵子，等某个事件将其唤醒；当 `Future` 返回 `Ready` 状态时，`Future` 对应的值已经得到，此时可以返回了。

你看，这样一个简单的数据结构，就托起了庞大的 Rust 异步 `async/await` 处理的生态。

回到 `async fn` 的返回值我们接着说，显然它是一个 `impl Future`，那么如果我们给一个普通的函数返回 `impl Future<Output>`，它的行为和 `async fn` 是不是一致呢？来写个简单的实验（[代码](#)）：

 复制代码

```
1 use futures::executor::block_on;
2 use std::future::Future;
3
4 #[tokio::main]
5 async fn main() {
6     let name1 = "Tyr".to_string();
7     let name2 = "Lindsey".to_string();
8
9     say_hello1(&name1).await;
10    say_hello2(&name2).await;
11
12    // Future 除了可以用 await 来执行外，还可以直接用 executor 执行
13    block_on(say_hello1(&name1));
14    block_on(say_hello2(&name2));
15 }
16
17 async fn say_hello1(name: &str) -> usize {
18     println!("Hello {}", name);
19     42
20 }
21
```

```
22 // async fn 关键字相当于一个返回 impl Future<Output> 的语法糖
23 fn say_hello2<'fut>(name: &'fut str) -> impl Future<Output = usize> + 'fut {
24     async move {
25         println!("Hello {}", name);
26         42
27     }
28 }
```

运行这段代码你会发现，say_hello1 和 say_hello2 是等价的，二者都可以使用 await 来执行，也可以将其提供给一个 executor 来执行。

这里我们见到了一个新的名词：executor。

什么是 executor？

你可以把 executor 大致想象成一个 Future 的调度器。对于线程来说，操作系统负责调度；但操作系统不会去调度用户态的协程（比如 Future），所以任何使用了协程来处理并发的程序，都需要有一个 executor 来负责协程的调度。

很多在语言层面支持协程的编程语言，比如 Golang / Erlang，都自带一个用户态的调度器。Rust 虽然也提供 Future 这样的协程，但它在语言层面并不提供 executor，把要不要使用 executor 和使用什么样的 executor 的自主权交给了开发者。好处是，当我的代码中不需要使用协程时，不需要引入任何运行时；而需要使用协程时，可以在生态系统中选择最合适我应用的 executor。

常见的 executor 有：

futures 库自带的很简单的 executor，上面的代码就使用了它的 block_on 函数；

tokio 提供的 executor，当使用 #[tokio::main] 时，就隐含引入了 tokio 的 executor；

🔗 [async-std](#) 提供的 executor，和 tokio 类似；

🔗 [smol](#) 提供的 async-executor，主要提供了 block_on。

注意，上面的代码我们混用了 #[tokio::main] 和 futures::executor::block_on，这只是为了展示 Future 使用的不同方式，在正式代码里，不建议混用不同的 executor，会降低程序的性能，还可能引发奇怪的问题。

当我们谈到 executor 时，就不得不提 reactor，它俩都是 [Reactor Pattern](#) 的组成部分，作为构建高性能事件驱动系统的一个很典型模式，Reactor pattern 它包含三部分：

task，待处理的任务。任务可以被打断，并且把控制权交给 executor，等待之后的调度；

executor，一个调度器。维护等待运行的任务（ready queue），以及被阻塞的任务（wait queue）；

reactor，维护事件队列。当事件来临时，通知 executor 唤醒某个任务等待运行。

executor 会调度执行待处理的任务，当任务无法继续进行却又没有完成时，它会挂起任务，并设置好合适的唤醒条件。之后，如果 reactor 得到了满足条件的事件，它会唤醒之前挂起的任务，然后 executor 就有机会继续执行这个任务。这样一直循环下去，直到任务执行完毕。

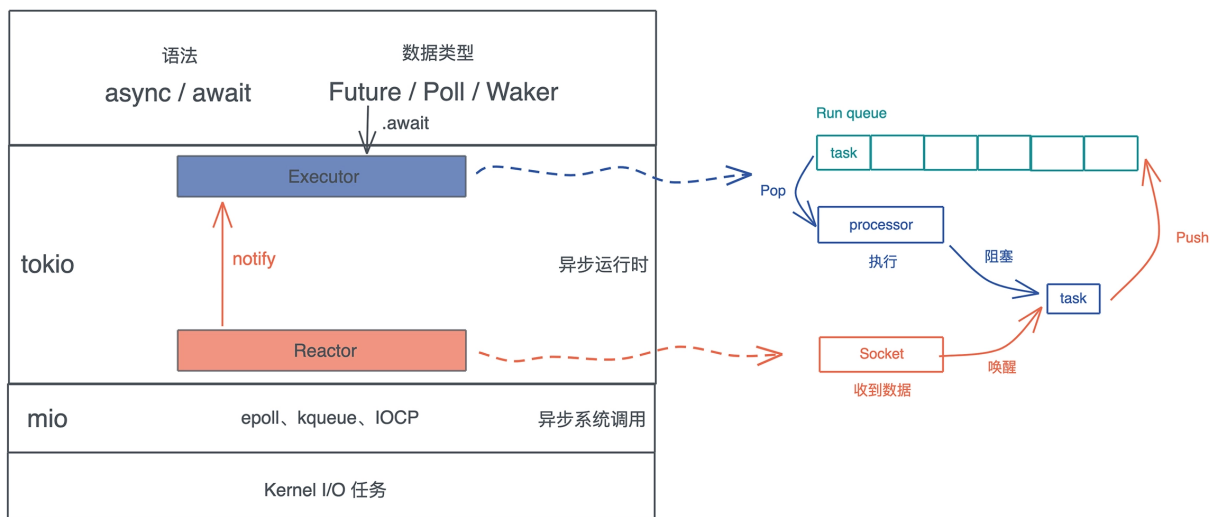
怎么用 Future 做异步处理？

理解了 Reactor pattern 后，Rust 使用 Future 做异步处理的整个结构就清晰了，我们以 tokio 为例：async/await 提供语法层面的支持，Future 是异步任务的数据结构，当 fut.await 时，executor 就会调度并执行它。

tokio 的调度器（executor）会运行在多个线程上，运行线程自己的 ready queue 上的任务（Future），如果没有，就去别的线程的调度器上“偷”一些过来运行。当某个任务无法再继续取得进展，此时 Future 运行的结果是 Poll::Pending，那么调度器会挂起任务，并设置好合适的唤醒条件（Waker），等待被 reactor 唤醒。

而 reactor 会利用操作系统提供的异步 I/O，比如 epoll / kqueue / IOCP，来监听操作系统提供的 IO 事件，当遇到满足条件的事件时，就会调用 Waker.wake() 唤醒被挂起的 Future。这个 Future 会回到 ready queue 等待执行。

整个流程如下：



我们以一个具体的代码示例来进一步理解这个过程（[🔗 代码](#)）：

复制代码

```

1 use anyhow::Result;
2 use futures::{SinkExt, StreamExt};
3 use tokio::net::TcpListener;
4 use tokio_util::codec::{Framed, LinesCodec};
5
6 #[tokio::main]
7 async fn main() -> Result<()> {
8     let addr = "0.0.0.0:8080";
9     let listener = TcpListener::bind(addr).await?;
10    println!("listen to: {}", addr);
11    loop {
12        let (stream, addr) = listener.accept().await?;
13        println!("Accepted: {:?}", addr);
14        tokio::spawn(async move {
15            // 使用 LinesCodec 把 TCP 数据切成一行行字符串处理
16            let framed = Framed::new(stream, LinesCodec::new());
17            // split 成 writer 和 reader
18            let (mut w, mut r) = framed.split();
19            for line in r.next().await {
20                // 每读到一行就加个前缀发回
21                w.send(format!("I got: {}", line?)).await?;
22            }
23            Ok::<_, anyhow::Error>(&())
24        });
25    }
26 }

```

这是一个简单的 TCP 服务器，服务器每收到一个客户端的请求，就会用 `tokio::spawn` 创建一个异步任务，放入 `executor` 中执行。这个异步任务接受客户端发来的按行分隔（分隔符是 “`\r\n`”）的数据帧，服务器每收到一行，就加个前缀把内容也按行发回给客户端。

你可以用 `telnet` 和这个服务器交互：

[复制代码](#)

```
1 > telnet localhost 8080
2 Trying 127.0.0.1...
3 Connected to localhost.
4 Escape character is '^]'.
5 hello
6 I got: hello
7 Connection closed by foreign host.
```

假设我们在客户端输入了很大的一行数据，服务器在做 `r.next().await` 在执行的时候，收不完一行的数据，因而这个 `Future` 返回 `Poll::Pending`，此时它被挂起。当后续客户端的数据到达时，`reactor` 会知道这个 `socket` 上又有数据了，于是找到 `socket` 对应的 `Future`，将其唤醒，继续接收数据。

这样反复下去，最终 `r.next().await` 得到 `Poll::Ready(Ok(line))`，于是它返回 `Ok(line)`，程序继续往下走，进入到 `w.send()` 的阶段。

从这段代码中你可以看到，在 `Rust` 下使用异步处理是一件非常简单的事情，除了几个你可能不太熟悉的概念，比如今天讲到的用于创建 `Future` 的 `async` 关键字，用于执行和等待 `Future` 执行完毕的 `await` 关键字，以及用于调度 `Future` 执行的运行时 `#[tokio:main]` 外，**整体的代码和使用线程处理的代码完全一致**。所以，它的上手难度非常低，很容易使用。

使用 Future 的注意事项

目前我们已经基本明白 `Future` 运行的基本原理了，也可以在程序的不同部分自如地使用 `Future/async/await` 来进行异步处理。但是，要注意，**不是所有的应用场景都适合用 `async/await`**，在使用的时候，有一些不容易注意到的坑需要我们妥善考虑。

计算密集型任务

当你要处理的任务是 CPU 密集型，而非 IO 密集型，更适合使用线程，而非 Future。

这是因为 Future 的调度是协作式多任务（Cooperative Multitasking），也就是说，除非 Future 主动放弃 CPU，不然它就会一直被执行，直到运行结束。我们看一个例子（[代码](#)）：

[复制代码](#)

```
1 use anyhow::Result;
2 use std::time::Duration;
3
4 // 强制 tokio 只使用一个工作线程，这样 task 2 不会跑到其它线程执行
5 #[tokio::main(worker_threads = 1)]
6 async fn main() -> Result<> {
7     // 先开始执行 task 1 的话会阻塞，让 task 2 没有机会运行
8     tokio::spawn(async move {
9         eprintln!("task 1");
10        // 试试把这句注释掉看看会产生什么结果
11        // tokio::time::sleep(Duration::from_millis(1)).await;
12        loop {}
13    });
14
15    tokio::spawn(async move {
16        eprintln!("task 2");
17    });
18
19    tokio::time::sleep(Duration::from_millis(1)).await;
20    Ok(())
21 }
```

task 1 里有一个死循环，你可以把它想象成是执行时间很长又不包括 IO 处理的代码。运行这段代码，你会发现，task 2 没有机会得到执行。这是因为 task 1 不执行结束，或者不交出 CPU，task 2 没有机会被调度。

如果你的确需要在 tokio（或者其它异步运行时）下运行运算量很大的代码，那么最好使用 yield 来主动让出 CPU，比如 [tokio::task::yield_now\(\)](#)。这样可以避免某个计算密集型的任务饿死其它任务。

异步代码中妥善使用

大部分时候, 标准库的 `Mutex` 可以用在异步代码中, 而且, 这是推荐的用法。然而, 标准库的 `MutexGuard` 不能安全地跨越 `await`, 所以, 当我们需要获得锁之后, 执行异步操作, 必须使用 `tokio` 自带的 `Mutex`, 看下面的例子 ([🔗 代码](#)) :

[📄 复制代码](#)

```
1 use anyhow::Result;
2 use std::{sync::Arc, time::Duration};
3 use tokio::sync::Mutex;
4
5 struct DB;
6
7 impl DB {
8     // 假装在 commit 数据
9     async fn commit(&mut self) -> Result<usize> {
10         Ok(42)
11     }
12 }
13
14 #[tokio::main]
15 async fn main() -> Result<()> {
16     let db1 = Arc::new(Mutex::new(DB));
17     let db2 = Arc::clone(&db1);
18
19     tokio::spawn(async move {
20         let mut db = db1.lock().await;
21         // 因为拿到的 MutexGuard 要跨越 await, 所以不能用 std::sync::Mutex
22         // 只能用 tokio::sync::Mutex
23         let affected = db.commit().await?;
24         println!("db1: Total affected rows: {}", affected);
25         Ok::<_, anyhow::Error>(())
26     });
27
28     tokio::spawn(async move {
29         let mut db = db2.lock().await;
30         let affected = db.commit().await?;
31         println!("db2: Total affected rows: {}", affected);
32
33         Ok::<_, anyhow::Error>(())
34     });
35
36     // 让两个 task 有机会执行完
37     tokio::time::sleep(Duration::from_millis(1)).await;
38
39     Ok(())
40 }
```

这个例子模拟了一个数据库的异步 `commit()` 操作。如果我们需要在多个 `tokio task` 中使用这个 `DB`，需要使用 `Arc<Mutex<DB>>`。然而，`db1.lock()` 拿到锁后，我们需要运行 `db.commit().await`，这是一个异步操作。

前面讲过，因为 `tokio` 实现了 `work-stealing` 调度，**Future 有可能在不同的线程中执行，普通的 `MutexGuard` 编译直接就会出错**，所以需要使用 `tokio` 的 `Mutex`。更多信息可以看 [📄 文档](#)。

在这个例子里，我们又见识到了 `Rust` 编译器的伟大之处：如果一件事，它觉得你不能做，会通过编译器错误阻止你，而不是任由编译通过，然后让程序在运行过程中听天由命，让你无休止地和捉摸不定的并发 `bug` 斗争。

使用 `channel` 做同步

在一个复杂的应用程序中，会兼有计算密集和 `IO` 密集的任务。前面说了，要避免在 `tokio` 这样的异步运行时中运行大量计算密集型的任务，一来效率不高，二来还容易饿死其它任务。

所以，一般的做法是我们使用 `channel` 来在线程和 `future` 两者之间做同步。看一个例子：

 复制代码

```
1 use std::thread;
2
3 use anyhow::Result;
4 use blake3::Hasher;
5 use futures::{SinkExt, StreamExt};
6 use rayon::prelude::*;
7 use tokio::{
8     net::TcpListener,
9     sync::{mpsc, oneshot},
10 };
11 use tokio_util::codec::{Framed, LinesCodec};
12
13 pub const PREFIX_ZERO: &[u8] = &[0, 0, 0];
14
15 #[tokio::main]
16 async fn main() -> Result<()> {
17     let addr = "0.0.0.0:8080";
18     let listener = TcpListener::bind(addr).await?;
19     println!("listen to: {}", addr);
20
21     // 创建 tokio task 和 thread 之间的 channel
```

```
22 let (sender, mut receiver) = mpsc::unbounded_channel::<(String, oneshot::S
23
24 // 使用 thread 处理计算密集型任务
25 thread::spawn(move || {
26     // 读取从 tokio task 过来的 msg, 注意这里用的是 blocking_recv, 而非 await
27     while let Some((line, reply)) = receiver.blocking_recv() {
28         // 计算 pow
29         let result = match pow(&line) {
30             Some((hash, nonce)) => format!("hash: {}, once: {}", hash, non
31             None => "Not found".to_string(),
32         };
33         // 把计算结果从 oneshot channel 里发回
34         if let Err(e) = reply.send(result) {
35             println!("Failed to send: {}", e);
36         }
37     }
38 });
39
40 // 使用 tokio task 处理 IO 密集型任务
41 loop {
42     let (stream, addr) = listener.accept().await?;
43     println!("Accepted: {:?}", addr);
44     let sender1 = sender.clone();
45     tokio::spawn(async move {
46         // 使用 LinesCodec 把 TCP 数据切成一行行字符串处理
47         let framed = Framed::new(stream, LinesCodec::new());
48         // split 成 writer 和 reader
49         let (mut w, mut r) = framed.split();
50         for line in r.next().await {
51             // 为每个消息创建一个 oneshot channel, 用于发送回复
52             let (reply, reply_receiver) = oneshot::channel();
53             sender1.send((line?, reply))?;
54
55             // 接收 pow 计算完成后的 hash 和 nonce
56             if let Ok(v) = reply_receiver.await {
57                 w.send(format!("Pow calculated: {}", v)).await?;
58             }
59         }
60         Ok::<_, anyhow::Error>::Ok(())
61     });
62 }
63 }
64
65 // 使用 rayon 并发计算 u32 空间下所有 nonce, 直到找到有头 N 个 0 的哈希
66 pub fn pow(s: &str) -> Option<(String, u32)> {
67     let hasher = blake3_base_hash(s.as_bytes());
68     let nonce = (0..u32::MAX).into_par_iter().find_any(|n| {
69         let hash = blake3_hash(hasher.clone(), n).as_bytes().to_vec();
70         &hash[..PREFIX_ZERO.len()] == PREFIX_ZERO
71     });
72     nonce.map(|n| {
73         let hash = blake3_hash(hasher, &n).to_hex().to_string();
```

```
74         (hash, n)
75     })
76 }
77
78 // 计算携带 nonce 后的哈希
79 fn blake3_hash(mut hasher: blake3::Hasher, nonce: &u32) -> blake3::Hash {
80     hasher.update(&nonce.to_be_bytes()[..]);
81     hasher.finalize()
82 }
83
84 // 计算数据的哈希
85 fn blake3_base_hash(data: &[u8]) -> Hasher {
86     let mut hasher = Hasher::new();
87     hasher.update(data);
88     hasher
89 }
```

在这个例子里，我们使用了之前撰写的 TCP server，只不过这次，客户端输入过来的一行文字，会被计算出一个 POW (Proof of Work) 的哈希：调整 nonce，不断计算哈希，直到哈希的头三个字节全是零为止。服务器要返回计算好的哈希和获得该哈希的 nonce。这是一个典型的计算密集型任务，所以我们需要使用线程来处理它。

而在 tokio task 和 thread 间使用 channel 进行同步。我们使用了一个 ubounded MPSC channel 从 tokio task 侧往 thread 侧发送消息，每条消息都附带一个 oneshot channel 用于 thread 侧往 tokio task 侧发送数据。

建议你仔细读读这段代码，最好自己写一遍，感受一下使用 channel 在计算密集型和 IO 密集型任务同步的方式。如果你用 telnet 连接，发送 “hello world!”，会得到不同的哈希和 nonce，它们都是正确的结果：

[复制代码](#)

```
1 > telnet localhost 8080
2 Trying 127.0.0.1...
3 Connected to localhost.
4 Escape character is '^]'.
5 hello world!
6 Pow calculated: hash: 0000006e6e9370d0f60f06bdc288efafa203fd99b9af0480d040b2cc
7 Connection closed by foreign host.
8
9 > telnet localhost 8080
10 Trying 127.0.0.1...
11 Connected to localhost.
12 Escape character is '^]'.
13 hello world!
```

```
14 Pow calculated: hash: 000000e23f0e9b7aeba9060a17ac676f3341284800a2db843e2f0e85
15 Connection closed by foreign host.
```

小结

通过拆解 `async fn` 有点奇怪的返回值结构, 我们学习了 Reactor pattern, 大致了解了 tokio 如何通过 executor 和 reactor 共同作用, 完成 Future 的调度、执行、阻塞, 以及唤醒。这是一个完整的循环, 直到 Future 返回 `Poll::Ready(T)`。

在学习 Future 的使用时, 估计你也发现了, 我们可以对比线程来学习, 可以看到, 下列代码的结构多么相似:

[复制代码](#)

```
1 fn thread_async() -> JoinHandle<usize> {
2     thread::spawn(move || {
3         println!("hello thread!");
4         42
5     })
6 }
7
8 fn task_async() -> impl Future<Output = usize> {
9     async move {
10         println!("hello async!");
11         42
12     }
13 }
```

在使用 Future 时, 主要有 3 点注意事项:

1. 我们要避免在异步任务中处理大量计算密集型的工作;
2. 在使用 Mutex 等同步原语时, 要注意标准库的 `MutexGuard` 无法跨越 `.await`, 所以, 此时要使用对异步友好的 Mutex, 如 `tokio::sync::Mutex`;
3. 如果要在线程和异步任务间同步, 可以使用 channel。

今天为了帮助你深入理解, 我们写了很多代码, 每一段你都可以再仔细阅读几遍, 把它们搞懂, 最好自己也能直接写出来, 这样你对 Future 才会有更深的理解。


思考题

想想看, 为什么标准库的 Mutex 不能跨越 await? 你可以把文中使用 `tokio::sync::Mutex` 的代码改成使用 `std::sync::Mutex`, 并对使用的接口做相应的改动 (把 `lock().await` 改成 `lock().unwrap()`), 看看编译器会报什么错。对着错误提示, 你明白为什么了么?

欢迎在留言区分享你的学习感悟和思考。今天你完成 Rust 学习的第 38 次打卡啦, 感谢你的收听, 如果你觉得有收获, 也欢迎你分享给身边的朋友, 邀他一起讨论。我们下节课见。

分享给需要的人, Ta订阅后你可得 **20 元现金奖励**

 生成海报并分享

 赞 9  提建议

© 版权归极客邦科技所有, 未经许可不得传播售卖。页面已增加防盗追踪, 如有侵权极客邦将依法追究其法律责任。

上一篇 37 | 阶段实操 (4): 构建一个简单的KV server-网络安全

下一篇 39 | 异步处理: async/await内部是怎么实现的?

精选留言 (3)

 写留言



CyNevis

2021-12-01

标准库的 Mutex 不能跨越 await, 盲猜一手是不是标准库的Mutex实现是依赖线程绑定, 得去看代码是怎么实现的

展开 ∨





罗杰 


2021-11-26

代码中的 `toml::from_str` 编译不过, 但在 `play.rust-lang.org` 中竟然可以编译通过, 很神奇, 我在本地添加了 toml 库, 并且 `use toml` 之后, 代码就可以正常运行了。

作者回复: playground 把常见的 crate 都添加了









罗杰

2021-11-26

今天的内容要好好消化一下...

展开

