

Hello World

为了开始我们的Tokio之旅，我们先从我们必修的"hello world"示例开始。这个程序将会创建一个TCP流并且写入 " hello,world " 到其中。这与写入非Tokio TCP流的Rust程序之间的区别在于该程序在创建流或者将"hello,world"的时候并不会阻塞程序的执行。

在开始之前你应该对TCP流的工作方式有一定的了解，相信理解[rust标准库](#)实现会对你有很大的帮助。

让我们开始吧。

首先，生成一个新的crate

```
cargo new --bin hello-world  
cd hello-world
```

接下来，在 [Cargo.toml](#) 中添加必要的依赖项：

```
[dependencies]  
tokio = "0.1"
```

在[main.rs](#)中引入包和类型：

```
extern crate tokio;  
  
use tokio::io;  
use tokio::net::TcpStream;  
use tokio::prelude::*;


```

这里我们使用[tokio](#)自己[io](#)和[net](#)模块。这些模块提供了与标准库中网络和I/O操作相同的抽象，只有一点很小的差别：所有操作都是异步执行的。

创建流

第一步是创建 [TcpStream](#).我们将使用Tokio实现的 [TcpStream](#).

```
fn main() {  
    // Parse the address of whatever server we're talking to  
    let addr = "127.0.0.1:6142".parse().unwrap();  
    let client = TcpStream::connect(&addr);  
  
    // Following snippets come here...  
}
```

接下来，我们给 `client` 加点东西。这里的异步任务将创建一个流，一旦这个流创建成功可用于执行进一步操作时就返回它（译者注：此处原文使用动词“yield”，不是通过返回值返回，而是通过调用 `and_then` 的 Lambda 表达式参数给出）。

```
let client = TcpStream::connect(&addr).and_then(|stream| {
    println!("created stream");

    // Process stream here.

    Ok(())
})
.map_err(|err| {
    // All tasks must have an `Error` type of `()`. This forces error
    // handling and helps avoid silencing failures.
    //
    // In our example, we are only going to log the error to STDOUT.
    println!("connection error = {:?}", err);
});
```

对 `TcpStream::connect` 的调用返回创建的 TCP 流的 `Future`。我们将在后面的指南中详细了解 [\[Futures\]](#)，但是现在您可以将一个 `Future` 视为表示将来最终会发生的事情的值（在这种情况下将创建流）。这意味着 `TcpStream::connect` 无需在返回前等待流的创建，而是立即返回一个表示创建 TCP 流操作的值。当这项操作真正执行时，我们会在下面看到。

`and_then` 方法在创建流后返回它。`and_then` 是一个组合器（combinator）函数的示例，用于定义如何处理异步工作。

每个组合器函数都获得必要状态的所有权以及执行的回调，并返回具有附加“步骤”的新 `Future`。这个 `Future` 是表示将在某个时间点完成的某些计算的值。

值得重申的是，返回的 `Future` 是惰性的，即在调用组合器时不会执行任何操作。相反，一旦所有异步操作都被串起来，最终的 `Future` 对象（代表整个任务）就会被“引发”（即运行）。也就是先前定义的操作开始运行的时间。换句话说，到目前为止我们编写的代码实际上并没有创建 TCP 流。

稍后我们将更深入地探讨 `futures`（以及 `streams` 和 `sinks` 的相关概念）。

同样重要的是要注意，在我们实际执行我们的 `Future` 对象之前，我们已经调用 `map_err` 来转换我们可能遇到的任何错误（）。这可以确保我们能够获知错误。

接下来，我们将处理流。

写入数据

我们的目标是把 "hello world\n" 写入到流中。

让我们回到 `TcpStream::connect(addr).and_then` 块：

```
let client = TcpStream::connect(&addr).and_then(|stream| {
    println!("created stream");
```

```
io::write_all(stream, "hello world\n").then(|result| {
    println!("wrote to stream; success={:?}", result.is_ok());
    Ok(())
})
});
```

`io::write_all`函数获取`stream`的所有权，并返回一个`Future`对象，该对象在整个消息写入流后被完成的。`then`用于对写入完成后运行的步骤进行排序。在我们的例子中，我们只是向`STDOUT`写一条消息来表示写完了。

注意`result`是一个包含原始流的`Result`对象。这允许我们对相同的流进行附加读取或写入。但是，我们没有其他任何事情要做，所以我们只需要删除流，它会自动被关闭。

运行客户端任务

到目前为止，我们已经拿到了一个代表我们的程序所要完成操作的`Future`对象，但我们实际上还没有运行它。我们需要一种方法来“引发”这种操作。我们需要一个执行者。

执行程序负责调度异步任务，使其完成。有许多执行器实现可供选择，每个都有不同的优缺点。在此示例中，我们将使用`Tokio`运行时(`Tokio runtime`)的默认执行程序。

```
println!("About to create the stream and write to it...");
tokio::run(client);
println!("Stream has been created and written to.");
```

`tokio::run`启动运行时，阻止当前线程，直到所有生成的任务完成并且所有资源（如文件和套接字）都已被删除。

到目前为止，我们只在执行程序上运行了一个任务，因此该`client`任务是阻止`run`返回的唯一任务。一旦`run`返回，我们可以确定我们的未来已经完成。

你可以在这里[here](#)找到完整的例子。

运行代码

`Netcat`是一种从命令行快速创建TCP套接字的工具。以下命令在先前指定的端口上启动侦听TCP套接字。

```
nc -l -p 6142
```

我们再另开一个终端窗口运行我们的项目。

```
cargo run
```

如果一切顺利，你应该看到`Netcat`打印出`hello world`。

下一步

本指南的下一页将开始深入研究Tokio运行时模型。

Futures

让我们仔细看看futures。Tokio构建在futurescrate之上并使用其运行时模型。这允许它也使用futures库与其他库互操作。

注意：此运行时模型与其他语言中的异步库非常不同。虽然在较高的层面上，API看起来很相似，但代码执行方式却有所不同。

我们将在下一节中仔细研究运行时，但是对运行时的基本了解是理解Future的必要条件。为了获得这种理解，我们首先看一下Rust默认使用的同步模型，看看它与Tokio的异步模型有何不同。

同步模型

首先，让我们简要介绍一下Rust标准库使用的同步（或阻塞）模型。

```
// let socket = ...;
let mut buf = [0; 1024];
let n = socket.read(&mut buf).unwrap();

// Do something with &buf[..n];
```

调用socket.read时，根据socket在其接收缓冲区中是否具有等待处理的数据。如果有待处理的数据，则read将立即返回，buf将填充该数据。但是，如果没有未决数据，则read函数将阻止当前线程，直到收到数据。一旦收到数据，buf将填充这个新接收的数据，并返回读取功能。

为了同时在许多不同的套接字上执行读取，每个套接字需要一个线程。每个套接字使用一个线程不能很好地扩展到大量的套接字。这被称为c10k问题。

非阻塞套接字

在执行像read这样的操作时避免阻塞线程的方法是不阻塞线程！非阻塞套接字允许执行操作，如读取，而不会阻塞线程。当套接字在其接收缓冲区中没有待处理的数据时，read函数立即返回，表明套接字“未准备好”以执行读取操作。

使用Tokio TcpStream时，即使没有要读取的待处理数据，对read的调用也将立即返回一个值（ErrorKind::WouldBlock）。如果没有待处理的数据，则调用者负责稍后再次调用read。诀窍是知道“晚些时候”的时间。

考虑非阻塞读取的另一种方法是“轮询”套接字以读取数据。

Future是围绕这种轮询模型的抽象。Future代表将在“未来某个时刻”提供的值。我们可以轮询Future并询问值是否准备就绪。我们来看看更多细节。

仔细看看Futures

future是表示异步计算完成的值。通常，由于系统中其他位置发生的事件使future完成。虽然我们从基本I/O的角度看待事物，但您可以使用future来表示各种事件，例如：

- 在线程池中执行的数据库查询。查询完成后，future完成，其值是查询的结果。

- 对服务器的RPC调用。当服务器回复时，`future`完成，其值是服务器的响应。
- 超时：当时间到了，`future`就完成了，它的值是`()`。
- 在线程池上运行的长时间运行的CPU密集型任务。任务完成后，`future`完成，其值为任务的返回值。
- 从套接字读取字节。当字节准备就绪时，`future`就完成了 - 根据缓冲策略，字节可能直接返回，或作为副作用写入某个现有缓冲区。

`future`抽象的整个要点是允许异步函数，即不能立即返回值的函数，能够返回一些东西。

例如，异步HTTP客户端可以提供如下所示的`get`函数：

```
pub fn get(&self, uri: &str) -> ResponseFuture { ... }
```

然后，库的用户将使用该函数：

```
let response_future = client.get("https://www.example.com");
```

现在，`response_future`不是实际响应。一旦收到回复，这将是一个`future`。但是，由于调用者具有具体值（`future`），因此他们可以开始使用它。例如，他们可以使用组合器链接计算，以便在收到响应后执行，或者可以将`future`传递给函数。

```
let response_is_ok = response_future
    .map(|response| {
        response.status().is_ok()
    });

track_response_success(response_is_ok);
```

所有与`future`一起采取的行动都不会立即执行任何工作。他们不能，因为他们没有实际的HTTP响应。相反，他们定义了响应`future`完成时要完成的工作。

`futures`箱和`Tokio`都有一系列组合功能，可以用来处理`future`。到目前为止，我们已经看到`and_then`将两个`future`链接在一起，然后允许将`future`链接到前一个，即使前一个错误，映射只是将`future`的值从一种类型映射到另一种类型。

我们将在本指南后面探索更多的组合器。

基于轮询模型的Future

如前所述，Rust`Future`基于轮询模型。这意味着，这意味着，`Future`一旦完成后，它不会负责将数据推送到某个地方，而是依赖于被询问它是否完成。

这是Rust`futures`库的一个独特方面。其他编程语言的大多数`Future`库使用基于推送的模型，其中回调被提供给`Future`，并且计算立即使用计算结果调用回调。

使用基于轮询的模型提供了许多优点，包括作为零成本抽象，即与手动编写异步代码相比，使用Rust Future 没有额外的开销。

我们将在下一节中仔细研究这种基于轮询的模型。

Future 特质 (trait)

Future 特质的定义如下：

```
trait Future {  
    /// The type of the value returned when the future completes.  
    type Item;  
  
    /// The type representing errors that occurred while processing the  
    computation.  
    type Error;  
  
    /// The function that will be repeatedly called to see if the future  
    is  
    /// has completed or not  
    fn poll(&mut self) -> Result<Async<Self::Item>, Self::Error>;  
}
```

现在，了解Future有两种相关类型非常重要： Item和Error。 Item是Future在完成时将产生的值的类型。 错误是如果在导致Future能够完成之前出现错误， Future可能会产生的错误类型。

最后，Futures有一种名为poll的方法。 我们不会在本节中详细介绍轮询模型，因为您不需要了解有关使用组合器的Future的轮询模型。 现在唯一需要注意的是， poll是在tokio运行时调用的，以便查看Future是否已完成。 如果你很好奇： Async是一个带有值的枚举， Ready(item)或者NotReady告诉tokio运行时Future是否完成。

在以后的部分中，我们将从头开始实现Future，包括编写一个poll函数，该函数在Future完成时正确通知tokio运行时。

流(Streams)

流是同类Future的迭代器。 流不会在未来的某个时间点产生值，而是在将来的某个时刻产生一组值。 换句话说，像Future一样，流在未来的某一点上不会产生一个值。 他们随着时间的推移继续产生值。

就像Future一样，只要这些东西在未来的某个时间点在不同的点产生离散值，就可以使用流来表示各种各样的东西。 例如：

- 由用户以不同方式与GUI交互而导致的UI事件。 当事件发生时，流会随着时间的推移向您的应用生成不同的消息。
- 从服务器推送通知。 有时请求/响应模型不是您需要的。 客户端可以与服务器建立通知流，以便能够从服务器接收消息而无需特别请求。
- 传入套接字连接。 当不同的客户端连接到服务器时，连接流将产生套接字连接。

Streams在实现过程中非常类似于future：

```
trait Stream {  
    /// The type of the value yielded by the stream.  
    type Item;  
  
    /// The type representing errors that occurred while processing the  
    computation.  
    type Error;  
  
    /// The function that will be repeatedly called to see if the stream  
    has  
    /// another value it can yield  
    fn poll(&mut self) -> Poll<Option<Self::Item>, Self::Error>;  
}
```

Streams带有他们自己的组合器，在[使用 Future 工作](#)章节更深入地介绍。

运行时

在上一节中，我们探讨了Futures和Streams，它们允许我们表示一个值（在Future的情况下）或一系列值（在Stream的情况下）将在“未来的某个时刻”可用。我们讨论了关于Future和Stream的轮询，运行时将调用它来确定Future或Stream是否已准备好产生值。

最后，我们说需要运行时来轮询Future和Streams来推动它们完成。我们现在将仔细研究运行时。

Tokio 运行时

为了让Future取得进步，必须要调用poll。这是运行时的工作。

运行时负责重复调用Future上的poll，直到返回其值。有许多不同的方法可以做到这一点，因此有许多类型的运行时配置。例如，CurrentThread运行时配置将阻止当前线程并循环遍历所有生成的Futures，并对它们调用poll。ThreadPool配置在线程池中安排Futures。这也是Tokio运行时使用的默认配置。

重要的是要记住，所有Future必须在运行时生成，否则不会执行任何工作。

产生任务

Tokio的一个独特方面是Future可以在运行时从其他Future或流中产生。当我们以这种方式使用Future时，我们通常将它们称为任务。任务是应用程序的“逻辑单元”。它们类似于[Go的goroutine]和[Erlang的过程]，但是异步。换句话说，任务是异步绿色线程。

鉴于任务运行异步逻辑位，它们由Future特征表示。任务完成处理后，任务的Future实现将以()值完成。

任务被传递到运行时，它处理任务的调度。运行时通常在一个或一组线程中调度许多任务。任务不得执行计算量大的逻辑，否则会阻止其他任务执行。因此，不要尝试将斐波那契序列计算为任务！

任务通过使用Future和Tokio中可用的各种组合函数或通过直接实现Tokio特征来构建Tokio来实现。

我们可以使用tokio::spawn生成任务。例如：

```
// Create some kind of future that we want our runtime to execute
let program = my_outer_stream.for_each(|my_outer_value| {
    println!("Got value {:?} from the stream", my_outer_value);

    let task = my_inner_future.and_then(|my_inner_value| {
        println!("Got a value {:?} from second future", my_inner_value);
        Ok(())
    });

    tokio::spawn(task);
    Ok(())
});

tokio::run(program);
```

再次产生任务可以在其他Future或流中产生，允许多个事物同时产生。在上面的例子中，我们从外部流中产生了内在的Future。每当我们从流中获得一个值时，我们就会简单地运行内在的Future。

在下一节中，我们将看一个比我们的hello-world示例更为复杂的示例，该示例将我们迄今为止所学到的所有内容都考虑在内。

示例：Echo服务器

我们将使用到目前为止所覆盖的内容来构建echo服务器。这是一个Tokio应用程序，它包含了我们迄今为止学到的所有内容。服务器将简单地从连接的客户端接收消息，并将收到的相同消息发送回客户端。

我们将能够使用我们在[hello world](#)部分中创建的基本Tcp客户端来测试此echo服务器。

完整的代码可以[在这里](#)找到。

创建

首先，生成一个新的箱子。

```
$ cargo new --bin echo-server
cd echo-server
```

接下来，添加必要的依赖项：

```
[dependencies]
tokio = "0.1"
```

main.rs

```
extern crate tokio;
extern crate futures;

use tokio::io;
use tokio::net::TcpListener;
use tokio::prelude::*;


```

现在，我们为服务器设置必要的结构：

- 绑定[TcpListener](#)到本地端口。
- 定义接受入站连接并处理它们的任务。
- 生成服务器任务。
- 启动Tokio运行时

同样，在执行者上生成服务器任务之前，实际上不会执行任何工作。

```
fn main() {
    let addr = "127.0.0.1:6142".parse().unwrap();
    let listener = TcpListener::bind(&addr).unwrap();
```

```

    // Here we convert the `TcpListener` to a stream of incoming
    connections
    // with the `incoming` method. We then define how to process each
    element in
    // the stream with the `for_each` combinator method
    let server = listener.incoming().for_each(|socket| {
        // TODO: Process socket
        Ok(())
    })
    .map_err(|err| {
        // Handle error by printing to STDOUT.
        println!("accept error = {:?}", err);
    });

    println!("server running on localhost:6142");
    # // `select` completes when the first of the two futures completes.
Since
    # // future::ok() completes immediately, the server won't hang waiting
for
    # // more connections. This is just so the doc test doesn't hang.
    # let server = server.select(futures::future::ok()).then(|_|

Ok(()));

    // Start the server
    //
    // This does a few things:
    //
    // * Start the Tokio runtime
    // * Spawns the `server` task onto the runtime.
    // * Blocks the current thread until the runtime becomes idle, i.e.
all
    //     spawned tasks have completed.
    tokio::run(server);
}

```

在这里，我们创建了一个可以侦听传入TCP连接的[TcpListener](#)。在监听器上，我们调用[incoming](#)，将监听器转换为入站客户端连接流。然后我们调用[for_each](#)，它将产生每个入站客户端连接。目前我们没有对此入站连接做任何事情 - 这是我们的下一步。

一旦我们拥有了我们的服务器，我们就可以将它交给[tokio::run](#)。到目前为止，我们的服务器功能一无所获。由Tokio运行时驱动我们的[Future](#)完成。

注意：我们必须在服务器上调用[map_err](#)，因为[tokio :: run](#)需要一个[Item](#)为[type \(\)](#) 和[Error](#)为[type \(\)](#)的[Future](#)。这是为了确保在将[Future](#)交付给运行时之前处理所有值和错误。

处理连接

既然我们有传入的客户端连接，我们应该处理它们。

我们只想将从套接字读取的所有数据复制回套接字本身（例如“echo”）。我们可以使用标准的[io :: copy](#)函数来做到这一点。

该copy函数有两个参数，从哪里读取以及在哪里写入。但是，我们只有一个参数，使用socket。幸运的是，有一个方法split，它将可读和可写的流分成两半。此操作允许我们独立地处理每个流，例如将它们作为copy函数的两个参数传递。

然后，copy函数返回一个Future，当复制操作完成时，将接收此Future，解析为复制的数据量。

让我们来看看我们再次传递给for_each的闭包。

```
let server = listener.incoming().for_each(|socket| {
    // split the socket stream into readable and writable parts
    let (reader, writer) = socket.split();
    // copy bytes from the reader into the writer
    let amount = io::copy(reader, writer);

    let msg = amount.then(|result| {
        match result {
            Ok((amount, _, _)) => println!("wrote {} bytes", amount),
            Err(e)              => println!("error: {}", e),
        }
        Ok(())
    });

    // spawn the task that handles the client connection socket on to the
    // tokio runtime. This means each client connection will be handled
    // concurrently
    tokio::spawn(msg);
    Ok(())
})
```

如您所见，我们已将socket流拆分为可读写部分。然后我们使用io :: copy从reader读取并写入writer。我们使用then组合器来查看amount未来的Item和Error作为Result打印一些诊断。

对tokio::spawn的调用是关键所在。至关重要的是我们希望所有clients同时取得进展，而不是在完成另一个client时阻止其中一个。为此，我们使用tokio :: spawn函数在后台执行工作。

如果我们没有这样做，那么for_each中块的每次调用都会在一次解决，这意味着我们永远不会同时处理两个客户端连接！

概述

Future 对象，在前文中曾多次提到的，是用来管理异步逻辑的基本构件，也是 Tokio 所使用的底层异步抽象。

一个 future 值代表了一个异步计算的完成。通常来讲，future 会因系统的其他部分产生的事件而完成。我们已经通过基本I/O看过一些具体的案例，然而你可以用一个 future 去表示更多可能的事件，比如：

- **一次数据库查询**，当查询结束时，future 对象被完成，它的值就是查询的结果。
- **一次RPC（远程过程调用）**。当远程服务器响应时，future 对象被完成，它的值就是服务器的相应。
- **一次超时**. 当给定时间耗尽时，future 对象被完成，它的值为 ()。
- **一个耗时较长的CPU密集型任务**，运行在一个线程池中。当任务结束时，future 对象被完成，它的值就是任务的返回值。
- **从一个套接字中读取若干字节**. 当这若干字节准备完毕，future 对象被完成——这取决于你的缓冲策略，这些字节可能被直接返回，或者已函数副作用的方式被写入到预先分配好的缓冲区中。

Tokio 的应用程序是以 future 对象为单位进行构造的，Tokio 掌握这些 future 对象并驱动它们执行。

实现 future

使用Tokio时，实现 future 是很常见的。让我们从一个基本的 future 开始，它不执行异步逻辑，只简单返回一个消息。（经典的“hello world”）

Future 特质

下面是 Future 特质的定义：

```
trait Future {  
    /// The type of the value returned when the future completes.  
    type Item;  
  
    /// The type representing errors that occurred while processing the  
    computation.  
    type Error;  
  
    /// The function that will be repeatedly called to see if the future  
    is  
    /// has completed or not. The `Async` enum can either by `Ready` or  
    /// or `NotReady` and indicates whether the future is ready  
    // to produce a value or not.  
    fn poll(&mut self) -> Result<Async<Self::Item>, Self::Error>;  
}
```

让我们为 "hello world" future 实现它：

```
extern crate futures;  
  
// `Poll` 是 `Result<Async<T>, E>` 类型的一个别名  
use futures::{Future, Async, Poll};  
  
struct HelloWorld;  
  
impl Future for HelloWorld {  
    type Item = String;  
    type Error = ();  
  
    fn poll(&mut self) -> Poll<Self::Item, Self::Error> {  
        Ok(Async::Ready("hello world".to_string()))  
    }  
}
```

Item 和 Error 的关联类型定义了 future 完成时返回的变量类型。Item 代表成功值的类型，Error 代表执行遇到错误时返回的值类型。通常，把不会失败的 future 的 Error 设置为 ()。

Future 使用基于拉取的模型。future 对象的消费者会多次调用 `poll` 函数，这个 future 就会尝试完成。如果这个 future 能够完成，它就会返回 `Ok(Async::Ready(value))`；如果这个 future 因为被内部资源（比如一个TCP套接字）阻塞导致不能完成，它就会返回 `Ok(Async::NotReady)`。

当一个future的 `poll` 函数被调用时，其实现将异步地做尽可能多的工作，直到它逻辑上被某个尚未发生的异步事件阻塞。然后，future 实现将在内部保存它的状态，这样当 `poll` 函数再次被调用时（在收到一个内部事件之后），它会从之前离开的地方继续执行。这里不会做重复工作。

此处的“hello world”future 不需要异步处理，是立即就绪的，所以它直接返回 `Ok(Async::Ready(value))`

执行 future

Tokio 负责将 future 对象执行完成。这是通过将 future 传递给 `tokio::run` 函数来实现的。

`tokio::run` 函数接受 `Item` and `Error` 都被设置为 `()` 的 future 作为参数。这是因为 Tokio 只执行 future 而不会对它们的值做任何操作。Tokio 的使用者需要包揽处理 future 中的所有值。

在我们的例子，让我们把 future 打印到标准输出（stdout）中。我们将通过一个 `Display` future 来实现这个效果。

```
extern crate futures;

use futures::{Future, Async, Poll};
use std::fmt;

struct Display<T>(T);

impl<T> Future for Display<T>
where
    T: Future,
    T::Item: fmt::Display,
{
    type Item = ();
    type Error = T::Error;

    fn poll(&mut self) -> Poll<(), T::Error> {
        let value = match self.0.poll() {
            Ok(Async::Ready(value)) => value,
            Ok(Async::NotReady) => return Ok(Async::NotReady),
            Err(err) => return Err(err),
        };
        println!("{}", value);
        Ok(Async::Ready(()))
    }
}
```

`Display` 拥有一个 future 成员，这个 future 成员可以生成被打印显示的对象（译者注：即实现了 `std::fmt::Display` 特质）。当它被拉取时，首先会尝试拉取内部 future 的值。如果内部 future 还 **没有就绪**，这个 `Display` 对象不会被完成。在这种情况下，`Display` 对象也会返回 `NotReady`。

除非调用内部的 `future` 得到一个 `NotReady` 值, `poll` 的实现永远不应当返回 `NotReady`。在后续章节我们将详细解释这一点。

当内部 `future` 产生错误时, `Display` 的 `future` 对象也会产生错误。错误是层层传递上来的。

当 `HelloWorld` 和 `Display` 组合在一起时, 所有的 `Item` 和 `Error` 类型都设置为 `()`, Tokio 就可以直接运行它们:

```
extern crate tokio;

let future = Display(HelloWorld);
tokio::run(future);
```

以上代码的运行结果就是 “hello world” 被输出到标准输出中 (stdout) 。

清理

等待内部 `future` 的模式很常见, 所以有了一个助手宏: `try_ready!`。

我们的 `poll` 函数可以用这个宏重写一下, 像这样:

```
#[macro_use]
extern crate futures;

use futures::{Future, Async, Poll};
use std::fmt;

struct Display<T>(T);

impl<T> Future for Display<T>
where
    T: Future,
    T::Item: fmt::Display,
{
    type Item = ();
    type Error = T::Error;

    fn poll(&mut self) -> Poll<(), T::Error> {
        let value = try_ready!(self.0.poll());
        println!("{}", value);
        Ok(Async::Ready(()))
    }
}
```

异步化

Future 是用于异步管理的。想实现一个可以异步完成的 future , 我们就要正确地处理从内部 future 获得 `Async::NotReady` 的过程。

让我们从实现一个新的 future 开始, 这个 future 将建立与远端的TCP套接字, 然后把对端的 IP 地址写入到标准输出 (stdout) 。

```
extern crate tokio;
#[macro_use]
extern crate futures;

use tokio::net::{TcpStream, tcp::ConnectFuture};
use futures::{Future, Async, Poll};

struct GetPeerAddr {
    connect: ConnectFuture,
}

impl Future for GetPeerAddr {
    type Item = ();
    type Error = ();

    fn poll(&mut self) -> Poll<Self::Item, Self::Error> {
        match self.connect.poll() {
            Ok(Async::Ready(socket)) => {
                println!("peer address = {}", socket.peer_addr().unwrap());
                Ok(Async::Ready(()))
            }
            Ok(Async::NotReady) => Ok(Async::NotReady),
            Err(e) => {
                println!("failed to connect: {}", e);
                Ok(Async::Ready(()))
            }
        }
    }
}

fn main() {
    let addr = "192.168.0.1:1234".parse().unwrap();
    let connect_future = TcpStream::connect(&addr);
    let get_peer_addr = GetPeerAddr {
        connect: connect_future,
    };

    tokio::run(get_peer_addr);
}
```

`GetPeerAddr` 的 future 实现非常类似于上一页的 `Display`。主要的区别在于，这个例子中的 `self.connect.poll()` 在返回连接的套接字之前将 (有可能) 多次返回 `Async::NotReady`。此时，我们的 future 将返回 `NotReady`。

`GetPeerAddr` 所包含的 future 对象 `ConnectFuture` 在 TCP 流建立时被完成。它是由 `TcpStream::connect` 返回的。

当 `GetPeerAddr` 作为参数传递给 `tokio::run` 时，Tokio 将多次调用 `poll` 函数，直到它返回 `Ready`。其中的确切机制将在后续章节介绍。

在实现 `Future` 时，除非我们通过调用内部 future 的 `poll` 函数获得了 `Async::NotReady`，我们的 `poll` 一定不能 返回 `Async::NotReady`。一种理解思路是：当一个 future 被拉取值时，它必须尽其所能的执行任务，直到它被完成或者被内部的 future 阻塞。

链式计算

现在，我们拿着这个建立连接的 future，给它加上 TCP 套接字建立后打印“hello world”的功能。

```
extern crate tokio;
extern crate bytes;
#[macro_use]
extern crate futures;

use tokio::io::AsyncWrite;
use tokio::net::{TcpStream, tcp::ConnectFuture};
use bytes::Bytes, Buf;
use futures::Future, Async, Poll;
use std::io::self, Cursor;

// HelloWorld 有两个状态，即等待连接的状态和已经连接的状态
enum HelloWorld {
    Connecting(ConnectFuture),
    Connected(TcpStream, Cursor<Bytes>),
}

impl Future for HelloWorld {
    type Item = ();
    type Error = io::Error;

    fn poll(&mut self) -> Poll<(), io::Error> {
        use self::HelloWorld::*;

        loop {
            let socket = match *self {
                Connecting(ref mut f) => {
                    try_ready!(f.poll())
                }
                Connected(ref mut socket, ref mut data) => {
                    // 只要缓冲区还有可用的数据，就一直将其写入到套接字中
                    while data.has_remaining() {
                        try_ready!(socket.write_buf(data));
                    }
                }
            }
        }
    }
}
```

```
        return Ok(Async::Ready(()));
    }
};

let data = Cursor::new(Bytes::from_static(b"hello world"));
*self = Connected(socket, data);
}
}

fn main() {
    let addr = "127.0.0.1:1234".parse().unwrap();
    let connect_future = TcpStream::connect(&addr);
    let hello_world = HelloWorld::Connecting(connect_future);

    // 运行之
    tokio::run(hello_world)
}
```

将 future 实现为其可能状态的枚举类型是很常见的用法。这使得 future 实现可以通过枚举值的变化跟踪内部状态。

此例中的 future 被描述为以下状态的枚举：

1. 连接中
2. 将 “hello world” 写入到套接字中

future 从内部包含 `ConnectFuture` 的“连接中”（译者注：即 `Connecting`）状态开始，多次拉取这个内部的 future，直到返回一个套接字，状态变为 `Connected`。

进入 `Connected` 状态，future 开始将数据写入到套接字中。写入操作是通过 `write_buf` 函数完成的。I/O 函数在下一章将会详细介绍。简言之，`write_buf` 就是一个可以无阻塞地将数据写入套接字的函数。如果套接字还没准备好接受写入，`NotReady` 会被返回。如果某些数据（并不一定是全部数据）被写入，`Ready(n)` 会被返回，而这里的 `n` 就是写入字节的个数。cursor 对象也是一个高级封装。

进入 `Connected` 状态之后，只要有剩余数据，future 就必须一直循环写入。因为 `write_buf` 使用 `try_ready!()` 调用，当 `write_buf` 返回 `NotReady` 时，我们的 `poll` 函数也会返回 `NotReady`。

在某些时候，我们 future 中的 `poll` 函数可能会被再次调用。而因为它已经处于 `Connected` 状态，它会直接跳到写数据的地方。

注意 循环非常重要。很多 future 的实现都包括循环。这些循环是很必要的，因为 `poll` 函数只有在所有数据都被写入套接字或者内部 future（比如 `ConnectFuture` 或者 `write_buf`）返回 `NotReady` 时才会返回。

组合器

Future 的实现往往遵循相同的模式。为了减少重复代码，future 库提供了许多被称为“组合器（Combinator）”的工具，它们是这些模式的抽象，多以 [Future] 特质相关的函数的形式存在。

基础构件

让我们回顾之前几页中的 future 实现，看看怎么用组合器去简化它们。

map

map 组合器拥有一个 future 并返回一个新 future，新 future 的值是通过前一个 future 调用某个给定的函数获得的。

这是之前实现的 future Display：

```
impl<T> Future for Display<T>
where
    T: Future,
    T::Item: fmt::Display,
{
    type Item = ();
    type Error = T::Error;

    fn poll(&mut self) -> Poll<(), T::Error> {
        let value = try_ready!(self.0.poll());
        println!("{}", value);
        Ok(Async::Ready(()))
    }
}

fn main() {
    let future = Display(HelloWorld);
    tokio::run(future);
}
```

如果用 map 组合器来写的话，就是这样的：

```
extern crate tokio;
extern crate futures;

use futures::Future;

fn main() {
    let future = HelloWorld.map(|value| {
        println!("{}", value);
    });
}
```

```
    tokio::run(future);
}
```

下面是 `map` 的实现:

```
pub struct Map<A, F> where A: Future {
    future: A,
    f: Option<F>,
}

impl<U, A, F> Future for Map<A, F>
    where A: Future,
          F: FnOnce(A::Item) -> U,
{
    type Item = U;
    type Error = A::Error;

    fn poll(&mut self) -> Poll<U, A::Error> {
        let value = try_ready!(self.future.poll());
        let f = self.f.take().expect("cannot poll Map twice");

        Ok(Async::Ready(f(value)))
    }
}
```

把 `Map` 和我们的 `Display` 放在一起比较，就可以明显看出它们的相似性。`Map` 在 `Display` 调用 `println!` 的相同位置把值传给了给定的函数。

and_then

现在，让我们开始用 `and_then` 组合器重写建立TCP流以及写入“hello world”的future。

`and_then` 组合器允许我们将两个异步操作连接起来。在第一个操作完成时，其值将被传递到一个函数中。该函数会使用该值创建一个新的 future 并使其运行。`and_then` 和 `map` 的区别是 `and_then` 的函数返回一个 future，而 `map` 的函数返回一个值。

最初的实现在 [这里][connect-and-write]。用组合器重写的话，就是这样的:

```
extern crate tokio;
extern crate bytes;
extern crate futures;

use tokio::io;
use tokio::net::TcpStream;
use futures::Future;

fn main() {
    let addr = "127.0.0.1:1234".parse().unwrap();
```

```

let future = TcpStream::connect(&addr)
    .and_then(|socket| {
        io::write_all(socket, b"hello world")
    })
    .map(|_| println!("write complete"))
    .map_err(|_| println!("failed"));

tokio::run(future);
}

```

进一步的计算也可以用链式调用 `and_then` 来连接。比如：

```

fn main() {
    let addr = "127.0.0.1:1234".parse().unwrap();

    let future = TcpStream::connect(&addr)
        .and_then(|socket| {
            io::write_all(socket, b"hello world")
        })
        .and_then(|(socket, _)| {
            // 只读取11个字节
            io::read_exact(socket, vec![0; 11])
        })
        .and_then(|(socket, buf)| {
            println!("got {:?}", buf);
            Ok(())
        });
}

tokio::run(future);
}

```

`and_then` 返回的 future 会像我们在之前手动实现的 future 那样执行。

基本组合器

花时间看一下 Future 特质 和其 模块 的文档来熟悉所有可用的组合器是很值得的。本文仅提供快速简要的概述。

既定的 future

任何值都可以立即生成一个已完成的 future。future 模块中有一些用于创建该类 future 的函数：

- `ok`, 对应 `Result::Ok`, 可以将给定值转化为一个立即就绪的 future, 该 future 可以用于生成原值。
- `err`, 对应 `Result::Err`, 可以将给定错误转化为一个立即就绪的失败的 future, 该 future 所包含的错误即原错误。
- `result` 将一个结果转化为一个立即完成的 future (译者注: `Result::Ok` 或者 `Result::Err` 都是可以的)。

另外，还有一个 `lazy` 函数，允许我们通过一个 闭包 来构建一个 future。这个闭包不会被立即调用，而是在 future 第一次被拉取时调用。

IntoFuture 特质

`IntoFuture` 特质是一个很关键的 API，它代表各种可以被转化为 future 的值。大多数使用 future 的接口实际上都是用它实现的。原因在于：`Result` 实现了这个特质，这就允许我们在很多需要返回 future 的地方直接返回 `Result` 值。

大多数返回 future 的组合器闭包实际上返回的也是一个 `IntoFuture` 实例。

适配器

就像 `Iterator` 那样，`Future` 特质也包含了各种各样的“适配器”方法。这些方法消费当前 future，返回一个新的 future 以提供我们请求的行为。使用这些适配组合器，我们可以：

- 改变一个 future 的类型 (`map`, `map_err`)
- 在一个 future 完成时执行另一个 (`then`, `and_then`, `or_else`)
- 找出两个 future 中哪个先执行完成 (`select`)
- 等待两个 future 都完成 (`join`)
- 转化为一个特质对象 (`Box::new`)
- 将展开式恐慌转化为错误 (`catch_unwind`)

何时使用组合器

使用组合器可以减少陈词滥调，但它们并不总那么合适。由于某些限制，手动实现 `Future` 可能更为常见。

函数式风格

传递给组合器的闭包必须是 '`static`' 的。这就意味着不可能在闭包中加入引用。所有状态的所有权必须被转移到闭包中。这是因为 Rust 的生命周期是基于栈的。使用异步代码，就意味着失去了栈的相关功能。

也正因为如此，使用组合器就会写出函数式风格的代码。让我们比较一下 Future 组合器和异步的 `Result` 组合器。

```
use std::io;

fn get_data() -> Result<Data, io::Error> {
    // ...
}

fn get_ok_data() -> Result<Vec<Data>, io::Error> {
    let mut dst = vec![];
    for _ in 0..10 {
        get_data().and_then(|data| {
            dst.push(data);
            Ok(())
        });
    }
}
```

```
    Ok(dst)
}
```

上面的代码可以工作是因为传递给 `and_then` 的闭包可以获取到 `dst` 的可变借用。Rust 编译器可以保证 `dst` 存活的比闭包更久。

然而使用 `future` 的话，借用 `dst` 就行不通了，必须改为传递 `dst`。像这样：

```
extern crate futures;

use futures::stream, Future, Stream;
use std::io;

fn get_data() -> impl Future<Item = Data, Error = io::Error> {
    // ...
}

fn get_ok_data() -> impl Future<Item = Vec<Data>, Error = io::Error> {
    let mut dst = vec![];

    // Start with an unbounded stream that uses unit values.
    stream::repeat(())
        // Only take 10. This is how the for loop is simulated using a
        functional
        // style.
        .take(10)
        // The `fold` combinator is used here because, in order to be
        // functional, the state must be moved into the combinator. In
        this
        // case, the state is the `dst` vector.
        .fold(dst, move |mut dst, _| {
            // Once again, the `dst` vector must be moved into the nested
            // closure.
            get_data().and_then(move |item| {
                dst.push(item);

                // The state must be included as part of the return value,
                so
                // `dst` is returned.
                Ok(dst)
            })
        })
}
```

还有一种策略，可以配合不可变数据使用，将数据存储在一个 `Arc` 中，然后将句柄的拷贝放到闭包中。有一种比较适用的场景就是将配置值在多个闭包中共享。比如：

```
extern crate futures;
```

```

use futures::future, Future;
use std::io;
use std::sync::Arc;

fn get_message() -> impl Future<Item = String, Error = io::Error> {
    // ...
}

fn print_multi() -> impl Future<Item = (), Error = io::Error> {
    let name = Arc::new("carl".to_string());

    let futures: Vec<_> = (0..1).map(|_| {
        // 拷贝 `name` 句柄，这样依赖多个并发的 future 都可以打印这个 `name` 值。
        let name = name.clone();

        get_message()
            .and_then(move |message| {
                println!("Hello {}, {}", name, message);
                Ok(())
            })
    })
    .collect();

    future::join_all(futures)
        .map(|_| ())
}

```

返回 future

因为组合器经常使用闭包作为它们类型签名的一部分，future 的类型是无法确定的。这就造成 future 的类型无法作为函数签名的一部分。当使用一个 future 作为函数参数时，泛型可以自由运用于几乎所有情况。例如：

```

extern crate futures;

use futures::Future;

fn get_message() -> impl Future<Item = String> {
    // ...
}

fn with_future<T: Future<Item = String>>(f: T) {
    // ...
}

let my_future = get_message().map(|message| {
    format!("MESSAGE = {}", message)
});

with_future(my_future);

```

但是当函数返回 future 的时候，就没有这么简单了。这里是一些各有利弊的可选方法：

- 使用 `impl Future`
- 特质对象
- 手动实现 `Future`

使用 `impl Future`

从 Rust 的 **1.26** 版本开始，`impl Trait` 这一语言特性就可以用来返回组合器 future 了。因此我们可以这样写：

```
fn add_10<F>(f: F) -> impl Future<Item = i32, Error = F::Error>
    where F: Future<Item = i32>,
{
    f.map(|i| i + 10)
}
```

`add_10` 函数的返回值类型是“某个实现了 `Future` 特质的类型”，它还附带类一些相关类型。这就允许我们不需要显式制定 future 的类型而直接返回一个 future。

这种方法的优点是它零开销并且适用于各种各样的情况。但是，使用这种方法从不同代码分支返回 future 的时候可能会有一个问题。例如：

```
if some_condition {
    return get_message()
        .map(|message| format!("MESSAGE = {}", message));
} else {
    return futures::ok("My MESSAGE".to_string());
}
```

从多个代码分支返回

以上代码会导致 `rustc` 输出编译错误：`error[E0308]: if and else have incompatible types` (`if` 和 `else` 存在不匹配的类型)。也就是说返回 `impl Future` 的函数还是必须有一个唯一确定的返回类型。

`impl Future` 语法只是允许我们不明确指定类型。然而，每个组合器类型都有一个不同的类型，这就造成各个条件分支中的返回类型不同。对于以上情况，我们有两种解决方案。第一种时将函数的返回值改为一个 `特质对象`。第二种方法是使用 `Either` 类型：

```
if some_condition {
    return Either::A(get_message()
        .map(|message| format!("MESSAGE = {}", message)));
} else {
    return Either::B(
        future::ok("My MESSAGE".to_string()));
}
```

这就确保了函数有唯一的返回值类型: `Either`.

在多于两个分支的时候, `Either` 枚举必须嵌套使用 (`Either<Either<A, B>, C>`) 或者自行定制一个支持多变量的枚举类型。

这种方案经常用于按条件返回错误的情况。比如:

```
fn my_operation(arg: String) -> impl Future<Item = String> {
    if is_valid(&arg) {
        return Either::A(get_message().map(|message| {
            format!("MESSAGE = {}", message)
        }));
    }

    Either::B(future::err("something went wrong"))
}
```

要在遇到错误时提前返回, 必须把错误放到一个 `Either` 变量中。

相关类型

具有返回 `future` 的函数的特质一定会包含 `future` 相关的类型定义。比如, 我们来看一个简化版本的 Tower 库中的 `Service` 特质:

```
pub trait Service {
    /// Requests handled by the service.
    type Request;

    /// Responses given by the service.
    type Response;

    /// Errors produced by the service.
    type Error;

    /// The future response value.
    type Future: Future<Item = Self::Response, Error = Self::Error>;

    fn call(&mut self, req: Self::Request) -> Self::Future;
}
```

为了实现这个特质, `call` 函数返回的 `future` 必须被明确指定并被设置为 `Future` 的相关类型。在这种情况下, `impl Future` 就不能用了, 我们必须把 `future` 装箱为一个 `特质对象`, 或者自己定制一个 `future`。

特质对象 (Trait objects)

还有一种策略是返回一个装箱的 `future`, 即一个 `[特质对象]`:

```
fn foo() -> Box<Future<Item = u32, Error = io::Error> + Send> {
    // ...
}
```

这种策略的优点是 `Box` 非常易于使用。我们还可以处理之前所述的分支问题，并且任意多个分支都可以：

```
fn my_operation(arg: String) -> Box<Future<Item = String, Error = &'static str> + Send> {
    if is_valid(&arg) {
        if arg == "foo" {
            return Box::new(get_message().map(|message| {
                format!("FOO = {}", message)
            }));
        } else {
            return Box::new(get_message().map(|message| {
                format!("MESSAGE = {}", message)
            }));
        }
    }
}

Box::new(future::err("something went wrong"))
}
```

这种方法的缺点是装箱会产生更多的开销。储存返回的 `future` 值会带来一次内存分配。并且，无论什么时候使用这个 `future`，Rust 都需要通过一次运行时查找（vtable；虚表）来动态地拆箱。这会使装箱的 `future` 在实际运行时稍微慢一些，尽管这种差异往往并不显著。

有一个附加说明可以帮作者们尝试使用 `Box<Future<...>>`，特别是跟 `tokio::run` 一同使用。默认情况下，`Box<Future<...>>` 没有实现 `Send` 特质，无法跨线程发送，即使内部装箱的 `future` 实现了 `Send` 特质。

想确保一个装箱的 `future` 实现 `Send` 特质，必须这样写：

```
fn my_operation() -> Box<Future<Item = String, Error = &'static str> + Send> {
    // ...
}
```

手动实现 Future

最后，当所有以上策略都失败时，我们还是可以退回到手动实现 `Future` 的方法。手动实现可以提供完整的控制，但是因为没有组合器函数能用于这种方法，我们得花更多的精力在那些陈词滥调上了。

何时使用

在你基于 Tokio 的应用程序中，组合器是减少重复代码的有力解决方案，但是就如本章节所述，它们并不是“银弹”。实现定制的future和定制的组合器还是很常见的。这就提出了什么时候使用组合器与手动实现 Future 的选择问题。

根据上述探讨，如果 future 的类型必须被指明并且 Box 是不可接受的开销，那么我们就可以不用组合器。除了这一点，选择什么还取决于组合器间传递的状态的复杂性。

“状态必须从多个组合器并发访问”可能是手动实现 Future 的一个适用场景。

待完善 (TODO)：本章节需要更多的例子。如果你有改善本章节的好点子，可以访问 [doc-push](#) 库并以你的想法创建一个问题。

流

流跟 future 很像，但是不只产生一个值，而是可能产生一个或多个值。我们可以把它看作是异步的迭代器。

就像 future 一样，流也可以代表各种各样的事物，只要这些事物可以在未来某几个不同的时间点产生离散的值。比如：

- 由用户和GUI界面交互产生的各种 **UI 事件**。当一个事件发生时，流就会产生一个消息。
- **来自服务器的推送通知**。有时候“请求/响应模式”无法满足需求。客户端可以建立到服务器端的通知流，这样就可以直接接收来自服务器的消息，而不需要明确的请求。
- **收到的套接字连接**。当多个客户端连接到某个服务器时，“连接流”将会生成套接字连接。

Stream 特质

就像 Future 一样，使用 Tokio 时实现 Stream 也是很常见的。Stream 特质的定义如下：

```
trait Stream {
    /// The type of the value yielded by the stream.
    type Item;

    /// The type representing errors that occurred while processing the
    computation.
    type Error;

    /// The function that will be repeatedly called to see if the stream
    has
    /// another value it can yield
    fn poll(&mut self) -> Poll<Option<Self::Item>, Self::Error>;
}
```

关联类型 **Item** 是流所要产生的值的类型。而关联类型 **Error** 则是某些意外发生时产生的错误的类型。**poll** 函数非常类似与 Future 的 **poll** 函数。唯一的区别是它的返回值是一个 **Option<Self::Item>**。

流的实现会将 **poll** 函数调用多次。当下一个值就绪时，返回 **Ok(Async::Ready(Some(value)))**；当流尚未就绪时，返回 **Ok(Async::NotReady)**；当流耗尽不再产生值时，返回 **Ok(Async::Ready(None))**。就像 future 一样，除非内部的流或者 future 返回 **Async::NotReady**，流本身 **一定不能** 返回 **Async::NotReady**。

当流遇到错误时，将返回 **Err(error)**。返回错误**并不**表示流耗尽了。错误可能只是暂时的，调用者可以再次尝试调用 **poll** 函数，流可能还会产生新的值。而如果错误是致命的，下一次调用 **poll** 函数将返回 **Ok(Async::Ready(None))**。

斐波那契数列 (Fibonacci)

下面的例子展示了如何将斐波那契数列实现为一个流。

```
extern crate futures;

use futures::Stream, Poll, Async;

pub struct Fibonacci {
    curr: u64,
    next: u64,
}

impl Fibonacci {
    fn new() -> Fibonacci {
        Fibonacci {
            curr: 1,
            next: 1,
        }
    }
}

impl Stream for Fibonacci {
    type Item = u64;

    // 该流将永远不会产生错误
    type Error = ();

    fn poll(&mut self) -> Poll<Option<u64>, ()> {
        let curr = self.curr;
        let next = curr + self.next;

        self.curr = self.next;
        self.next = next;

        Ok(Async::Ready(Some(curr)))
    }
}
```

要使用流，必须创建一个 future 来消费它。下面的 future 将从一个流中获取 10 个值并打印。

```
#[macro_use]
extern crate futures;

use futures::Future, Stream, Poll, Async;
use std::fmt;

pub struct Display10<T> {
    stream: T,
    curr: usize,
}

impl<T> Display10<T> {
    fn new(stream: T) -> Display10<T> {
        Display10 {
```

```

        stream,
        curr: 0,
    }
}
}

impl<T> Future for Display10<T>
where
    T: Stream,
    T::Item: fmt::Display,
{
    type Item = ();
    type Error = T::Error;

    fn poll(&mut self) -> Poll<(), Self::Error> {
        while self.curr < 10 {
            let value = match try_ready!(self.stream.poll()) {
                Some(value) => value,
                // There were less than 10 values to display, terminate
the
                // future.
                None => break,
            };

            println!("value #{} = {}", self.curr, value);
            self.curr += 1;
        }

        Ok(Async::Ready(()))
    }
}
}

```

现在，斐波那契数列就可以被打印出来了：

```

extern crate tokio;

let fib = Fibonacci::new();
let display = Display10::new(fib);

tokio::run(display);

```

异步化

到目前为止，这个斐波那契流还是同步的。让我们在每两个值之间增加一秒的等待时间来把它变成异步的。而要实现这样的效果，就需要使用 `tokio::timer::Interval`。`Interval` 本身就是一个流，它可以按给定时间间隔产生()值。在间隔时间之外调用 `Interval::poll` 将返回 `Async::NotReady`。

我们把 `Fibonacci` 流改成这样：

```
#[macro_use]
extern crate futures;
extern crate tokio;

use tokio::timer::Interval;
use futures::{Stream, Poll, Async};
use std::time::Duration;

pub struct Fibonacci {
    interval: Interval,
    curr: u64,
    next: u64,
}

impl Fibonacci {
    fn new(duration: Duration) -> Fibonacci {
        Fibonacci {
            interval: Interval::new_interval(duration),
            curr: 1,
            next: 1,
        }
    }
}

impl Stream for Fibonacci {
    type Item = u64;

    // 该流将永远不会产生错误
    type Error = ();

    fn poll(&mut self) -> Poll<Option<u64>, ()> {
        // 等待下一个间隔
        try_ready!(
            self.interval.poll()
                // 如果 Tokio 运行时不可用, interval 可能会拉取失败
                // 在本例中, 错误不做处理
                .map_err(|_| ())
        );

        let curr = self.curr;
        let next = curr + self.next;

        self.curr = self.next;
        self.next = next;

        Ok(Async::Ready(Some(curr)))
    }
}
```

因为 **Display10** 已经支持异步了, 所以不需要修改。

像这样运行这个基于时间间隔限流的斐波那契数列:

```
extern crate tokio;

use std::time::Duration;

let fib = Fibonacci::new(Duration::from_secs(1));
let display = Display10::new(fib);

tokio::run(display);
```

组合器

跟 future 一样，流也可以通过很多组合器来减少重复代码。很多组合器都是以函数的形式存在于 Stream 特质中的。

我们可以使用 unfold 函数来重写斐波那契流：

```
extern crate futures;

use futures::{stream, Stream};

fn fibonacci() -> impl Stream<Item = u64, Error = ()> {
    stream::unfold((1, 1), |(curr, next)| {
        let new_next = curr + next;

        Some(Ok((curr, (next, new_next))))
    })
}
```

同样也跟 future 一样，使用流的组合器也需要函数式的编程风格。并且，impl Stream 也可以作为返回流的函数的返回值类型。返回 future 的策略同样适用于返回流。

Display10 可以使用 take 和 for_each 重新实现：

```
extern crate tokio;
extern crate futures;

use futures::Stream;

tokio::run(
    fibonacci().take(10)
        .for_each(|num| {
            println!("{}!", num);
            Ok(())
        })
);
```

`take` 组合器限制斐波那契流只会产生 10 个值。而 `for_each` 组合器会异步地遍历流的各个值。`for_each` 会消费这个流，并返回 future，每个 future 都会在闭包参数使用一个值执行时被完成。它就是 Rust 中 `for` 循环的异步版。

基本组合器

花时间看一下 [Stream 特质](#) 及模块[mod-dox](#)文档来熟悉各种可用的组合器是很值得的。本文仅提供快速简要的概述。

既定的流

`stream` 模块 包括一些将已有的值和迭代器转化为流的函数。

- `once` 将给定值转化为一个立即就绪的流，它将产生一个值：给定值。
- `iter_ok` 和 `iter_result` 都使用 `IntoIterator` 值并将它们转化为一个立即就绪的流，该流将遍历产生迭代器的值。
- `empty` 返回一个立即产生 `None` 的流。

例如：

```
extern crate tokio;
extern crate futures;

use futures::stream, Stream;

let values = vec!["one", "two", "three"];

tokio::run(
    stream::iter_ok(values).for_each(|value| {
        println!("{}!", value);
        Ok(())
    })
)
```

适配器

像 `Iterator` 一样，`Stream` 特质包括各种各样的“适配器”方法。这些方法都会消费当前流，返回一个新流以提供我们请求的行为。使用这些适配组合器，我们可以：

- 改变一个流的类型 (`[map]`, `[map_err]`, `[and_then]`)
- 处理流产生的错误 (`[or_else]`).
- 过滤流产生的值 (`take`, `[take_while]`, `[skip]`, `[skip_while]`, `[filter]`, `[filter_map]`).
- 异步遍历 (`for_each`, `[fold]`).
- 将多个流组合到一起 (`[zip]`, `[chain]`, `[select]`).

创建任务

基于 Tokio 的应用程序是以任务（task）为单位组织的。任务是较小的独立运行的逻辑单元。类似于 [Go 语言的 goroutine](#) 和 [Erlang 的 process](#)。换句话说，任务是异步的绿色线程（green thread）。创建（spawn）任务与使用同步代码创建线程往往出于相似的原因，但是使用 Tokio 创建任务非常轻量。

之前的一些例子定义 future 并将其传递给 `tokio::run` 函数。这样就会在 Tokio 的运行时上创建一个任务并执行。更多的任务可能需要通过调用 `tokio::spawn` 来创建，这仅限于那些已经作为 Tokio 任务运行的代码。有一个帮助理解的好方法，我们可以把传递给 `tokio::run` 函数的 future 视为“main 函数”。

下面的例子创建了四个任务。

```
extern crate tokio;
extern crate futures;

use futures::future::lazy;

tokio::run(lazy(|| {
    for i in 0..4 {
        tokio::spawn(lazy(move || {
            println!("Hello from task {}", i);
            Ok(())
        }));
    }
    Ok(())
}));
```

`tokio::run` 函数将会一直阻塞，直到传递给它的 future 包括其他所有创建的任务执行完。而在这个例子中，`tokio::run` 将会阻塞至四个任务将内容打印到标准输出（stdout）然后退出。

`lazy` 函数会在 future 第一次被拉取时运行内部的闭包。这里使用它是为了确保 `tokio::spawn` 是从一个任务中调用的。如果不使用 `lazy`，`tokio::spawn` 就会在任务上下文的外部调用，这样就会产生错误了。

与任务通信

就像 Go 语言和 Erlang 那样，任务可以通过传递消息来通信。实际上，使用消息传递来协调任务是非常常见的。相互独立的任务因此可以产生互动。

[[futures](#)] 库提供了一个 [[sync](#)] 模块，这个模块包括了一些通道（channel）类型，它们是跨任务消息传递的理想选择。

- [[oneshot](#)] 是用于发送单个值的通道。
- [[mpsc](#)] 是用于发送多个值（零或多个）的通道。

`oneshot` 非常适用于从一个已创建的任务中获取结果：

```

extern crate tokio;
extern crate futures;

use futures::Future;
use futures::future::lazy;
use futures::sync::oneshot;

tokio::run(lazy(|| {
    let (tx, rx) = oneshot::channel();

    tokio::spawn(lazy(|| {
        tx.send("hello from spawned task");
        Ok(())
    }));
}

rx.and_then(|msg| {
    println!("Got `{:?}`", msg);
    Ok(())
})
.map_err(|e| println!("error = {:?}", e))
}));
```

而 `mpsc` 适用于将流式数据发送到另一个任务中：

```

extern crate tokio;
extern crate futures;

use futures::{stream, Future, Stream, Sink};
use futures::future::lazy;
use futures::sync::mpsc;

tokio::run(lazy(|| {
    let (tx, rx) = mpsc::channel(1_024);

    tokio::spawn({
        stream::iter_ok(0..10).fold(tx, |tx, i| {
            tx.send(format!("Message {} from spawned task", i))
            .map_err(|e| println!("error = {:?}", e))
        })
        .map(|_| ()) // 释放 tx 句柄
    });

    rx.for_each(|msg| {
        println!("Got `{:?}`", msg);
        Ok(())
    })
}));
```

以上两个消息传递原语也将用于后续例子中任务间的协调与通信。

多线程

使用 future 而不创建任务来实现并发也是可以的，这种并发将运行于单线程中。而创建任务则允许 Tokio 运行时在多个线程上调度这些任务。

[多线程 Tokio 运行时](#) 在内部管理多个操作系统线程。它可以仅靠少量物理线程多路复用来运行很多任务。当一个 Tokio 应用创建任务时，这些任务会被提交给运行时环境，运行时环境将自动调度。

何时创建任务

对于大多数软件相关的问题，我们的答案都是要视具体情况来决定。通常来说，你应该尽可能的创建任务。因为你创建的任务越多，就意味着你并行地执行任务的能力就越强。但是，一定要注意，多任务通信会引入通道的开销。

接下来的几个例子将介绍如何创建新任务。

处理入站套接字

创建任务最直接的例子就是网络服务器。它的主任务 (main task) 是用TCP监听器监听入站套接字。当一个新的连接建立时，监听器任务将创建一个新任务来处理对应的套接字。

```
extern crate tokio;
extern crate futures;

use tokio::io;
use tokio::net::TcpListener;
use futures::{Future, Stream};

let addr = "127.0.0.1:0".parse().unwrap();
let listener = TcpListener::bind(&addr).unwrap();

tokio::run({
    listener.incoming().for_each(|socket| {
        // 接受到一个入站套接字
        //
        // 创建一个新任务来处理套接字
        tokio::spawn({
            // 在本例中，直接将 "hello world" 写入到套接字中然后关闭
            io::write_all(socket, "hello world")
                // 释放套接字
                .map(|_| ())
                // 向标准输出 (stdout) 中写入错误信息
                .map_err(|e| println!("socket error = {:?}", e))
        });
        //
        // 接受下一个入站套接字
        Ok(())
    })
    .map_err(|e| println!("listener error = {:?}", e))
});
```

监听任务以及每个套接字的处理任务是完全无关的。它们不需要通信，运行停止也不会影响其它任务。以上，就是一个创建任务的完美用例。

后台处理

另一个例子是创建任务用于执行后台计算来为其它任务提供服务。它的主任务发送数据至后台任务处理，但并不关心数据是否或何时被处理。这样就可以实现一个单独的后台任务对来自多个主任务的数据进行合并处理。

这个例子需要主任务与后台任务通信。通常用 [mpsc] 通道来处理。

下面的例子是一个TCP服务器，它从远端读取数据，并记录接受的字节数。然后，它把接受的字节数发送到一个后台任务。这个后台任务将每30秒打印各套接字任务接受的字节数总和。

```
extern crate tokio;
extern crate futures;

use tokio::io;
use tokio::net::TcpListener;
use tokio::timer::Interval;
use futures::{future, stream, Future, Stream, Sink};
use futures::future::lazy;
use futures::sync::mpsc;
use std::time::Duration;

// 定义后台任务。`rx` 参数是通道的接收句柄。
// 任务将从通道中拉取 `usize` 值（代表套接字读取都字节数）并求总和。
// 所求的总和将每三十秒被打印到标准输出 (stdout) 然后重置为零。
fn bg_task(rx: mpsc::Receiver<usize>)
-> impl Future<Item = (), Error = ()>
{
    // The stream of received `usize` values will be merged with a 30
    // second interval stream. The value types of each stream must
    // match. This enum is used to track the various values.
    #[derive(Eq, PartialEq)]
    enum Item {
        Value(usize),
        Tick,
        Done,
    }

    // 打印总和到标准输出都时间间隔。
    let tick_dur = Duration::from_secs(30);

    let interval = Interval::new_interval(tick_dur)
        .map(|_| Item::Tick)
        .map_err(|_| ());

    // 将流转换为这样的序列:
    // Item(num), Item(num), ... Done
    //
    let items = rx.map(Item::Value)
```

```
.chain(stream::once(Ok(Item::Done)))
// Merge in the stream of intervals
.select(interval)
// Terminate the stream once `Done` is received. This is necessary
// because `Interval` is an infinite stream and `select` will keep
// selecting on it.
.take_while(|item| future::ok(*item != Item::Done));

// With the stream of `Item` values, start our logic.
//
// Using `fold` allows the state to be maintained across iterations.
// In this case, the state is the number of read bytes between tick.
items.fold(0, |num, item| {
    match item {
        // Sum the number of bytes with the state.
        Item::Value(v) => future::ok(num + v),
        Item::Tick => {
            println!("bytes read = {}", num);

            // 重置字节计数器
            future::ok(0)
        }
        _ => unreachable!(),
    }
})
.map(|_| ())
}

// 启动应用
tokio::run(lazy(|| {
    let addr = "127.0.0.1:0".parse().unwrap();
    let listener = TcpListener::bind(&addr).unwrap();

    // 创建用于与后台任务通信的通道。
    let (tx, rx) = mpsc::channel(1_024);

    // 创建后台任务：
    tokio::spawn(bg_task(rx));

    listener.incoming().for_each(move |socket| {
        // 接收到一个入站套接字。
        //
        // 创建新任务处理套接字。
        tokio::spawn({
            // 每个新创建的任务都会拥有发送者句柄的一份拷贝。
            let tx = tx.clone();

            // 在本例中，将 "hello world" 写入套接字然后关闭之。
            io::read_to_end(socket, vec![])
                // 释放套接字
                .and_then(move |(_, buf)| {
                    tx.send(buf.len())
                        .map_err(|_| io::ErrorKind::Other.into())
                })
        })
    })
}))
```

```

        .map(|_| ())
        // 打印错误信息到标准输出
        .map_err(|e| println!("socket error = {:?}", e))
    });

    // 接收下一个入站套接字
    Ok(())
}
.map_err(|e| println!("listener error = {:?}", e))
));

```

协调资源访问

在使用 future 时，协调资源（套接字、数据等等）访问的一个较好的策略时使用消息传递。要实现这样的策略，就需要创建一个专用的任务管理资源，而其它的任务通过发送消息与这个资源交互。

这种模式与之前的例子非常相似，但是这次，任务需要在操作完成后接受一个返回消息。要实现这种模式，`mpsc` 和 `oneshot` 两种通道都会用到。

这个例子通过“乒乓协议 (ping/pong protocol) ”协调对一个 [transport] 的访问。“兵”是发送到 transport 上的，而“兵”是在上面接收到的。主任务 (primary task) 发送一个消息到协调者任务 (coordinator task) 来初始化一个“兵”的请求，协调者任务会将消息往返时间作为请求的响应。主任务通过 `mpsc` 发送给协调者任务的消息包含一个 `oneshot::Sender` 值，协调者任务可以通过它返回响应。

```

extern crate tokio;
extern crate futures;

use tokio::io;
use futures::{future, Future, Stream, Sink};
use futures::future::lazy;
use futures::sync::{mpsc, oneshot};
use std::time::{Duration, Instant};

type Message = oneshot::Sender<Duration>;

struct Transport;

impl Transport {
    fn send_ping(&self) {
        // ...
    }

    fn recv_pong(&self) -> impl Future<Item = (), Error = io::Error> {
        // ...
    }
}

fn coordinator_task(rx: mpsc::Receiver<Message>
-> impl Future<Item = (), Error = ()>
{
    let transport = Transport;

```

```
rx.for_each(move |pong_tx| {
    let start = Instant::now();

    transport.send_ping();

    transport.recv_pong()
        .map_err(|_| ())
        .and_then(move |_| {
            let rtt = start.elapsed();
            pong_tx.send(rtt).unwrap();
            Ok(())
        })
    })
}

/// Request an rtt.
fn rtt(tx: mpsc::Sender<Message>)
-> impl Future<Item = (Duration, mpsc::Sender<Message>), Error = ()>
{
    let (resp_tx, resp_rx) = oneshot::channel();

    tx.send(resp_tx)
        .map_err(|_| ())
        .and_then(|tx| {
            resp_rx.map(|dur| (dur, tx))
                .map_err(|_| ())
        })
}
}

// 启动应用
tokio::run(lazy(|| {
    // 创建用于与后台任务通信的通道。
    let (tx, rx) = mpsc::channel(1_024);

    // 创建后台任务：
    tokio::spawn(coordinator_task(rx));

    // 创建少量任务，它们向协调者任务请求 RTT 时间。
    for _ in 0..4 {
        let tx = tx.clone();

        tokio::spawn(lazy(|| {
            rtt(tx).and_then(|(dur, _)| {
                println!("duration = {:?}", dur);
                Ok(())
            })
        }));
    }
    Ok(())
}));
```

何时不要创建任务

如果通过消息传递进行协调以及同步原语带来的开销已经超过了创建任务带来的并行收益，那么使用一个单独的任务会是更好的选择。

比如，通常来讲，对同一个 TCP 套接字而言，在一个的任务中进行读写要好过把读和写分到两个任务中进行。

运行时模型

流类似于future，但它们不是产生单个值，而是异步地产生一个或多个值。它们可以被认为是异步迭代器。

就像future一样，流可以代表各种各样的东西，只要这些东西在未来的某个时间点在不同的点产生离散的值。例如：

- 由用户以不同方式与GUI交互而导致的UI事件。当事件发生时，流会随着时间的推移向您的应用生成不同的消息。
- 从服务器推送通知。有时请求/响应模型不是您需要的。客户端可以与服务器建立通知流，以便能够从服务器接收消息而无需特别请求。
- 传入套接字连接。当不同的客户端连接到服务器时，连接流将产生套接字连接。

Stream特质

就像Future，使用Tokio时实现Stream很正常。Streams产生许多值，所以让我们从生成fibonacci序列的流开始。

Stream特质如下：

```
trait Stream {  
    /// The type of the value yielded by the stream.  
    type Item;  
  
    /// The type representing errors that occurred while processing the  
    computation.  
    type Error;  
  
    /// The function that will be repeatedly called to see if the stream  
    has  
    /// another value it can yield  
    fn poll(&mut self) -> Poll<Option<Self::Item>, Self::Error>;  
}
```

Item关联类型是流生成的类型。Error关联类型是发生意外情况时产生的错误类型。poll函数与Future的poll函数非常相似。唯一的区别是，这次返回Option <Self :: Item>。

流实现具有多次调用的poll函数。当下一个值准备就绪时，返回Ok (Async :: Ready (Some (value)))。当流未准备好生成值时，将返回Ok (Async :: NotReady)。当流耗尽并且不再产生其他值时，返回Ok (Async :: Ready (None))。

I/O 概述

Rust标准库提供对网络和I/O的支持，例如TCP连接，UDP套接字，读取和写入文件等。但是，这些操作都是同步或阻塞，这意味着当您调用它们时，当前线程可能会停止执行并进入睡眠状态，直到它被解除阻塞。例如，`std :: io :: Read`中的`read`方法将阻塞，直到有数据要读取。在future的世界中，这种行为是不幸的，因为我们希望在等待I / O完成时继续执行我们可能拥有的其他future。

为了实现这一点，Tokio提供了许多标准库I / O资源的非阻塞版本，例如文件操作和TCP，UDP和Unix套接字。它们返回长期运行的future（如接受新的TCP连接），并实现`std :: io :: Read`和`std :: io :: Write`的非阻塞变体，称为`AsyncRead`和`AsyncWrite`。

例如，如果没有可用的数据，则不会阻止非阻塞读取和写入。相反，它们会立即返回一个`WouldBlock`错误，以及一个保证（如`Future :: poll`），它们已安排当前任务在以后可以取得进展时被唤醒，例如当网络数据包到达时。

通过使用非阻塞的TokioI / O类型，如果不能立即执行他们希望执行的I / O，则执行I / O的future不再阻止执行其他future。相反，它只返回`NotReady`，并依赖于任务通知，以便再次调用`poll`，以及它的I / O应该成功而不会阻塞。

在幕后，Tokio使用mio和tokio-fs来跟踪不同future等待的各种I / O资源的状态，并在操作系统的任何状态发生变化时通知操作系统。

一个示例服务器

为了了解这是如何组合在一起的，请考虑这个echo server实现：

```
use tokio::prelude::*;
use tokio::net::TcpListener;

// Set up a listening socket, just like in std::net
let addr = "127.0.0.1:12345".parse().unwrap();
let listener = TcpListener::bind(&addr)
    .expect("unable to bind TCP listener");

// Listen for incoming connections.
// This is similar to the iterator of incoming connections that
// .incoming() from std::net::TcpListener, produces, except that
// it is an asynchronous Stream of tokio::net::TcpStream instead
// of an Iterator of std::net::TcpStream.
let incoming = listener.incoming();

// Since this is a Stream, not an Iterator, we use the for_each
// combinator to specify what should happen each time a new
// connection becomes available.
let server = incoming
    .map_err(|e| eprintln!("accept failed = {:?}", e))
    .for_each(|socket| {
        // Each time we get a connection, this closure gets called.
        // We want to construct a Future that will read all the bytes
    })
}
```

```
// from the socket, and write them back on that same socket.  
//  
// If this were a TcpStream from the standard library, a read or  
// write here would block the current thread, and prevent new  
// connections from being accepted or handled. However, this  
// socket is a Tokio TcpStream, which implements non-blocking  
// I/O! So, if we read or write from this socket, and the  
// operation would block, the Future will just return NotReady  
// and then be polled again in the future.  
//  
// While we *could* write our own Future combinator that does an  
// (async) read followed by an (async) write, we'll instead use  
// tokio::io::copy, which already implements that. We split the  
// TcpStream into a read "half" and a write "half", and use the  
// copy combinator to produce a Future that asynchronously  
// copies all the data from the read half to the write half.  
let (reader, writer) = socket.split();  
let bytes_copied = tokio::io::copy(reader, writer);  
let handle_conn = bytes_copied.map(|amt| {  
    println!("wrote {:?}", amt)  
}).map_err(|err| {  
    eprintln!("I/O error {:?}", err)  
});  
  
// handle_conn here is still a Future, so it hasn't actually  
// done any work yet. We *could* return it here; then for_each  
// would wait for it to complete before it accepts the next  
// connection. However, we want to be able to handle multiple  
// connections in parallel, so we instead spawn the future and  
// return an "empty" future that immediately resolves so that  
// Tokio will _simultaneously_ accept new connections and  
// service this one.  
tokio::spawn(handle_conn)  
});  
  
// The `server` variable above is itself a Future, and hasn't actually  
// done any work yet to set up the server. We need to run it on a Tokio  
// runtime for the server to really get up and running:  
tokio::run(server);
```

更多例子可以在[这里](#)找到。

数据读写

非阻塞I/O

在概述中我们简要提到Tokio的I / O类型实现了`std :: io :: Read`和`std :: io :: Write`的非阻塞变体，称为`AsyncRead`和`AsyncWrite`。这些是Tokio的I / O故事中不可或缺的一部分，在使用I / O代码时很重要。

注意：在本节中，我们将主要讨论`AsyncRead`，但`AsyncWrite`几乎完全相同，仅用于将数据写入I / O资源（如TCP套接字）而不是从中读取。

那么，让我们来看看`AsyncRead`，看看所有的大惊小怪：

```
use std::io::Read;
pub trait AsyncRead: Read {
    // ...
    // various provided methods
    // ...
}
```

呵呵。这里发生了什么？好吧，`AsyncRead`实际上只是从`std :: io`中的`Read`，还有一个额外的合同。`AsyncRead`的文档如下：

此特征继承自`std :: io :: Read`，表示I / O对象是非阻塞的。当字节不可用而不是阻塞当前线程时，所有非阻塞I / O对象都必须返回错误。

最后一部分是至关重要的。如果为类型实现`AsyncRead`，则承诺对其进行调用将永远不会阻塞。相反，如果它不是非阻塞的，则应该返回一个`io :: ErrorKind :: WouldBlock`错误以指示操作将被阻止（例如因为没有可用的数据）。提供的`poll_read`方法：

```
fn poll_read(&mut self, buf: &mut [u8]) -> Poll<usize, std::io::Error> {
    match self.read(buf) {
        Ok(t) => Ok(Async::Ready(t)),
        Err(ref e) if e.kind() == std::io::ErrorKind::WouldBlock => {
            Ok(Async::NotReady)
        }
        Err(e) => Err(e),
    }
}
```

这段代码应该很熟悉。如果你稍微眯一下，`poll_read`看起来很像`Future :: poll`。那是因为这几乎就是它的本质！实现`AsyncRead`的类型本质上就像您可以尝试从中读取数据的`Future`，并且它将通知您它是否已准备好（并且某些数据已被读取）或`NotReady`（并且您将不得不稍后再次轮询）。

使用Future I/O

由于`AsyncRead`（和`AsyncWrite`）几乎都是`Future`，因此您可以轻松地将它们嵌入到您自己的`Future`中，并像轮询任何其他嵌入式`Future`一样轮询它们。你甚至可以使用`try_ready!`根据需要传播错误和`NotReady`。我们将在下一节中详细讨论直接使用这些特质。但是，为了在许多情况下简化生活，Tokio在`tokio::io`中提供了许多有用的组合器，用于在`AsyncRead`和`AsyncWrite`之上执行常见的`I / O`操作。通常，它们提供围绕实现`Future`的`AsyncRead`或`AsyncWrite`类型的包装器，并在给定的读取或写入操作完成时完成。

第一个方便的`I / O`组合器是`read_exact`。它需要一个可变缓冲区（`&mut [u8]`）和`AsyncRead`的实现者作为参数，并返回一个`Future`，它读取足够的字节来填充缓冲区。在内部，返回的`Future`只跟踪它到目前为止读取了多少字节，并继续在`AsyncRead`上发出`poll_ready`（如果需要，返回`NotReady`），直到它完全填满缓冲区。此时，它将使用填充的缓冲区返回`Ready (buf)`。让我们来看看：

```
use tokio::net::tcp::TcpStream;
use tokio::prelude::*;

let addr = "127.0.0.1:12345".parse().unwrap();
let read_8_fut = TcpStream::connect(&addr)
    .and_then(|stream| {
        // We need to create a buffer for read_exact to write into.
        // A Vec<u8> is a good starting point.
        // read_exact will read buffer.len() bytes, so we need
        // to make sure the Vec isn't empty!
        let mut buf = vec![0; 8];

        // read_exact returns a Future that resolves when
        // buffer.len() bytes have been read from stream.
        tokio::io::read_exact(stream, buf)
    })
    .inspect(|(_stream, buf)| {
        // Notice that we get both the buffer and the stream back
        // here, so that we can now continue using the stream to
        // send a reply for example.
        println!("got eight bytes: {:?}", buf);
    });

// We can now either chain more futures onto read_8_fut,
// or if all we wanted to do was read and print those 8
// bytes, we can just use tokio::run to run it (taking
// care to map Future::Item and Future::Error to ()).
```

通常有用的第二个`I / O`组合器是`write_all`。它需要一个缓冲区（`& [u8]`）和`AsyncWrite`的实现者作为参数，并返回一个`Future`，它使用`poll_write`将缓冲区的所有字节写入`AsyncWrite`。当`Future`解析时，整个缓冲区已被写出并刷新。我们可以将它与`read_exact`结合起来，以回应服务器回复的内容：

```
use tokio::net::tcp::TcpStream;
use tokio::prelude::*;

let echo_fut = TcpStream::connect(&addr)
    .and_then(|stream| {
        // We're going to read the first 32 bytes the server sends us
```

```

    // and then just echo them back:
    let mut buf = vec![0; 32];
    // First, we need to read the server's message
    tokio::io::read_exact(stream, buf)
}
.and_then(|(stream, buf)| {
    // Then, we use write_all to write the entire buffer back:
    tokio::io::write_all(stream, buf)
})
.inspect(|(_stream, buf)| {
    println!("echoed back {} bytes: {:?}", buf.len(), buf);
});

// As before, we can chain more futures onto echo_fut,
// or declare ourselves finished and run it with tokio::run.

```

Tokio还带有一个I / O组合器来实现这种复制。它（也许不出所料）称为副本。copy采用AsyncRead和AsyncWrite，并连续将从AsyncRead读出的所有字节写入AsyncWrite，直到poll_read指示输入已关闭并且所有字节都已写出并刷新到输出。这是我们在echo服务器中使用的组合器！它大大简化了我们上面的示例，并使其适用于任何数量的服务器数据！

```

use tokio::net::tcp::TcpStream;
use tokio::prelude::*;

let echo_fut = TcpStream::connect(&addr)
    .and_then(|stream| {
        // First, we need to get a separate read and write handle for
        // the connection so that we can forward one to the other.
        // See "Split I/O resources" below for more details.
        let (reader, writer) = stream.split();
        // Then, we can use copy to send all the read bytes to the
        // writer, and return how many bytes it read/wrote.
        tokio::io::copy(reader, writer)
})
.inspect(|(bytes_copied, r, w)| {
    println!("echoed back {} bytes", bytes_copied);
});

```

很简约！

到目前为止我们谈到的组合器都是针对相当低级别的操作：读取这些字节，写入这些字节，复制这些字节。但是，通常情况下，您希望在更高级别的表示上操作，例如行。Tokio也在那里覆盖！lines接受AsyncRead，并返回一个Stream，它从输入中产生每一行，直到没有更多行要读取：

```

use tokio::net::tcp::TcpStream;
use tokio::prelude::*;

let lines_fut = TcpStream::connect(&addr).and_then(|stream| {
    // We want to parse out each line we receive on stream.
}

```

```
// To do that, we may need to buffer input for a little while
// (if the server sends two lines in one packet for example).
// Because of that, lines requires that the AsyncRead it is
// given *also* implements BufRead. This may be familiar if
// you've ever used the lines() method from std::io::BufRead.
// Luckily, BufReader from the standard library gives us that!
let stream = std::io::BufReader::new(stream);
tokio::io::lines(stream).for_each(|line| {
    println!("server sent us the line: {}", line);
    // This closure is called for each line we receive,
    // and returns a Future that represents the work we
    // want to do before accepting the next line.
    // In this case, we just wanted to print, so we
    // don't need to do anything more.
    Ok(())
})
});
```

tokio :: io中还有更多I / O组合器，您可能需要在决定编写自己的之前先查看一下！

拆分I/O资源

上面的复制示例和echo服务器都包含这个神秘的片段：

```
let (reader, writer) = socket.split();
let bytes_copied = tokio::io::copy(reader, writer);
```

正如上面的评论所解释的那样，我们将TcpStream（套接字）拆分为读half和写half，并使用我们上面讨论的copy组合器生成一个Future，它将读取的一半中的所有数据异步复制到写半。但为什么首先需要这种“分裂”呢？毕竟，`AsyncRead :: poll_ready`和`AsyncWrite :: poll_write`只需要使用`&mut self`。

要回答这个问题，我们需要回顾一下Rust的所有权制度。回想一下，Rust只允许您一次对给定变量进行单个可变引用。但是我们必须传递两个参数来复制，一个用于从哪里读取，一个用于写入。但是，一旦我们将一个可变引用传递给TcpStream作为参数之一，我们就不能构造第二个可变引用作为第二个参数传递给它！我们知道副本不会同时读取和写入，但这不会在副本的类型中表达。

当类型也实现AsyncWrite时，在AsyncRead trait上提供了split方法，如果我们看一下签名，我们就会看到

```
fn split(self) -> (ReadHalf<Self>, WriteHalf<Self>)
where Self: AsyncWrite { ... }
```

返回的ReadHalf实现了AsyncRead，而WriteHalf实现了AsyncWrite。至关重要的是，我们现在有两个单独的指针指向我们的类型，我们可以单独传递。这对于复制(copy)来说非常方便，但它也意味着我们可以将每一半传递给不同的future，并完全独立地处理读写操作！在幕后，分割确保如果我们同时尝试读取和写入，则一次只发生其中一个。

传输

在需要执行I / O的应用程序中，将`AsyncRead`转换为`Stream`（就像行一样）或`AsyncWrite`转换为`Sink`是很常见的。他们经常想要抽象出从线上检索或放置字节的方式，并让他们的大多数应用程序代码处理更方便的“请求”和“响应”类型。这通常称为“构造”：您可以将它们视为接收和发送的应用程序数据的“框架”，而不是将您的连接仅视为字节输入/字节输出。字节构造流通常被称为“传输”。

传输通常使用编解码器实现。例如，`lines`表示一个非常简单的编解码器，它将字节字符串与换行符\n分隔开来，并在将每个帧作为字符串分析之后再传递给应用程序。Tokio提供帮助程序，用于在`tokio :: codec`中实现新的编解码器；为传输实现编码器和解码器特性，并使用`Framed :: new`从字节流中创建一个`Sink + Stream`（如`TcpStream`）。这几乎就像魔术一样！还有一些版本只用于编解码器的读或写端（如线）。让我们看一下编写基于行的编解码器的简单实现（即使存在`LinesCodec`）：

```
extern crate bytes;
use bytes::{BufMut, BytesMut};
use tokio::codec::{Decoder, Encoder};
use tokio::prelude::*;

// This is where we'd keep track of any extra book-keeping information
// our transport needs to operate.
struct LinesCodec;

// Turns string errors into std::io::Error
fn bad_utf8<E>(_: E) -> std::io::Error {
    std::io::Error::new(std::io::ErrorKind::InvalidData, "Unable to decode
input as UTF8")
}

// First, we implement encoding, because it's so straightforward.
// Just write out the bytes of the string followed by a newline!
// Easy-peasy.
impl Encoder for LinesCodec {
    type Item = String;
    type Error = std::io::Error;

    fn encode(&mut self, line: Self::Item, buf: &mut BytesMut) ->
Result<(), Self::Error> {
        // Note that we're given a BytesMut here to write into.
        // BytesMut comes from the bytes crate, and aims to give
        // efficient read/write access to a buffer. To use it,
        // we have to reserve memory before we try to write to it.
        buf.reserve(line.len() + 1);
        // And now, we write out our stuff!
        buf.put(line);
        buf.put_u8(b'\n');
        Ok(())
    }
}

// The decoding is a little trickier, because we need to look for
// newline characters. We also need to handle *two* cases: the "normal"
// case where we're just asked to find the next string in a bunch of
// bytes, and the "end" case where the input has ended, and we need
// to find any remaining strings (the last of which may not end with a
```

```
// newline!
impl Decoder for LinesCodec {
    type Item = String;
    type Error = std::io::Error;

    // Find the next line in buf!
    fn decode(&mut self, buf: &mut BytesMut) -> Result<Option<Self::Item>, Self::Error> {
        Ok(if let Some(offset) = buf.iter().position(|b| *b == b'\n') {
            // We found a newline character in this buffer!
            // Cut out the line from the buffer so we don't return it
            again.
            let mut line = buf.split_to(offset + 1);
            // And then parse it as UTF-8
            Some(
                std::str::from_utf8(&line[..line.len() - 1])
                    .map_err(bad_utf8)?
                    .to_string(),
            )
        } else {
            // There are no newlines in this buffer, so no lines to speak
            of.
            // Tokio will make sure to call this again when we have more
            bytes.
            None
        })
    }

    // Find the next line in buf when there will be no more data coming.
    fn decode_eof(&mut self, buf: &mut BytesMut) ->
    Result<Option<Self::Item>, Self::Error> {
        Ok(match self.decode(buf)? {
            Some(frame) => {
                // There's a regular line here, so we may as well just
                return that.
                Some(frame)
            },
            None => {
                // There are no more lines in buf!
                // We know there are no more bytes coming though,
                // so we just return the remainder, if any.
                if buf.is_empty() {
                    None
                } else {
                    Some(
                        std::str::from_utf8(&buf.take(..))
                            .map_err(bad_utf8)?
                            .to_string(),
                    )
                }
            }
        })
    }
}
```


直接使用 AsyncRead 和 AsyncWrite

到目前为止，我们主要在Tokio提供的I/O组合器的上下文中讨论了`AsyncRead`和`AsyncWrite`。虽然这些通常已经足够，但有时您需要实现自己的组合器，这些组合器希望直接执行异步读取和写入。

使用`AsyncRead`读取数据

`AsyncRead`的核心是`poll_read`方法。它映射了`WouldBlock`错误，指示I/O读取操作将阻塞到`NotReady`，这反过来让我们可以与`future`世界互操作。当你编写一个内部包含`AsyncRead`的`Future`（或类似的东西，比如`Stream`）时，`poll_read`可能就是你要与之交互的方法。

要记住`poll_read`的重要一点是它遵循与`Future :: poll`相同的合同。具体来说，它不能返回`NotReady`，除非它已安排当前任务可以在再次进行时得到通知。这个事实让我们在我们自己的`future`中调查`poll_read`；我们知道当我们从`poll_read`转发`NotReady`时，我们正在维护`poll`合同，因为`poll_read`遵循相同的合同！

Tokio用于确保`poll_read`后来通知当前任务的确切机制超出了本节的范围，但如果感兴趣，可以在Tokio内部的非阻塞I/O部分中阅读更多相关内容。

有了这一切，让我们看看我们如何自己实现`read_exact`方法！

```
#[macro_use]
extern crate futures;
use std::io;
use tokio::prelude::*;

// This is going to be our Future.
// In the common case, this is set to Some(Reading),
// but we'll set it to None when we return Async::Ready
// so that we can return the reader and the buffer.
struct ReadExact<R, T>(Option<Reading<R, T>>);

struct Reading<R, T> {
    // This is the stream we're reading from.
    reader: R,
    // This is the buffer we're reading into.
    buffer: T,
    // And this is how far into the buffer we've written.
    pos: usize,
}

// We want to be able to construct a ReadExact over anything
// that implements AsyncRead, and any buffer that can be
// thought of as a &mut [u8].
fn read_exact<R, T>(reader: R, buffer: T) -> ReadExact<R, T>
where
    R: AsyncRead,
    T: AsMut<[u8]>,
```

```
ReadExact(Some(Reading {
    reader,
    buffer,
    // Initially, we've read no bytes into buffer.
    pos: 0,
}))  
}  
  
impl<R, T> Future for ReadExact<R, T>  
where  
    R: AsyncRead,  
    T: AsMut<[u8]>,  
{  
    // When we've filled up the buffer, we want to return both the buffer  
    // with the data that we read and the reader itself.  
    type Item = (R, T);  
    type Error = io::Error;  
  
    fn poll(&mut self) -> Poll<Self::Item, Self::Error> {  
        match self.0 {  
            Some(Reading {  
                ref mut reader,  
                ref mut buffer,  
                ref mut pos,  
            }) => {  
                let buffer = buffer.as_mut();  
                // Check that we haven't finished  
                while *pos < buffer.len() {  
                    // Try to read data into the remainder of the buffer.  
                    // Just like read in std::io::Read, poll_read *can*  
read  
                    // fewer bytes than the length of the buffer it is  
given,  
                    // and we need to handle that by looking at its return  
                    // value, which is the number of bytes actually read.  
                    //  
                    // Notice that we are using try_ready! here, so if  
poll_read  
                    // returns NotReady (or an error), we will do the  
same!  
                    // We uphold the contract that we have arranged to be  
                    // notified later because poll_read follows that same  
                    // contract, and _it_ returned NotReady.  
                    let n = try_ready!(reader.poll_read(&mut  
buffer[*pos..]));  
                    *pos += n;  
  
                    // If no bytes were read, but there was no error, this  
                    // generally implies that the reader will provide no  
more  
                    // data (for example, because the TCP connection was  
closed  
                    // by the other side).  
                    if n == 0 {  
                        break;  
                    }  
                }  
                Ok((self.0, Poll::Pending))  
            }  
            None => Err(io::Error::new(io::ErrorKind::Other, "no reader"))  
        }  
    }  
}
```

```

        return
Err(io::Error::new(io::ErrorKind::UnexpectedEof, "early eof"));
    }
}
None => panic!("poll a ReadExact after it's done"),
}

// We need to return the reader and the buffer, which we can only
// do by moving them out of self. We do this by taking our state
// and leaving `None`. This _should_ be fine, because poll()
// requires callers to not call poll() again after Ready has been
// returned, so we should only ever see Some(Reading) when poll()
// is called.
let reading = self.0.take().expect("must have seen Some above");
Ok(Async::Ready((reading.reader, reading.buffer)))
}
}
}

```

使用AsyncWrite写入数据

就像`poll_read`是`AsyncRead`的核心部分一样，`poll_write`是`AsyncWrite`的核心。与`poll_read`一样，它映射了`WouldBlock`错误，该错误指示I / O写入操作将阻塞到`NotReady`，这再次允许我们与未来世界互操作。`AsyncWrite`还有一个`poll_flush`，它为`Write`的`flush`方法提供异步模拟。`poll_flush`的作用是确保先前由`poll_write`写入的任何字节都被刷新到基础I/O资源上（例如，在网络数据包中写出）。与`poll_write`类似，它包装`Write::flush`，并将一个`WouldBlock`错误映射到`NotReady`，以指示刷新仍在进行中。

`AsyncWrite`的`poll_write`和`poll_flush`遵循与`Future :: poll`和`AsyncRead :: poll_read`相同的合同，即如果它们返回`NotReady`，它们已经安排当前任务在他们可以再次进展时得到通知。与`poll_read`一样，这意味着我们可以在我们自己的`future`中安全地调用这些方法，并且知道我们也在遵守合同。

Tokio使用相同的机制来管理`poll_write`和`poll_flush`的通知，就像它对`poll_read`一样，你可以在Tokio内部的非阻塞I/O部分中阅读更多相关内容。

关闭

`AsyncWrite`还添加了一个不属于`Write: shutdown`的方法。从其文件：

启动或尝试关闭此编写器，在I/O连接完全关闭时返回成功。

此方法旨在用于I/O连接的异步关闭。例如，这适用于在代理连接上实现TLS连接的关闭或调用`TcpStream :: shutdown`。协议有时需要清除最终的数据或以其他方式执行正常的关闭握手，适当地读取/写入更多数据。此方法是实现正常关闭逻辑的此类协议的钩子。

这总结很好地关闭：它是一种告诉作者不再有数据的方法，并且它应该以底层I/O协议所需的任何方式指示。例如，对于TCP连接，这通常需要关闭TCP通道的写入端，以便另一端以读取形式接收和返回0字节的文件结束。您通常可以将关闭视为在`Drop`的实现中同步完成的操作；只是在异步世界中，你不能轻易地在`Drop`中做一些事情，因为你需要有一个继续轮询你的作家的执行者！

请注意，在实现`AsyncWrite`和`AsyncRead`的类型的写入 "half" 上调用`shutdown`不会关闭读取"half"。您通常可以继续随意读取数据，直到另一方关闭相应的写入"一半"。

使用`AsyncWrite`的示例

不用多说，让我们来看看我们如何实现

```
#[macro_use]
extern crate futures;
use std::io;
use tokio::prelude::*;

// This is going to be our Future.
// It'll seem awfully familiar to ReadExact above!
// In the common case, this is set to Some(Writing),
// but we'll set it to None when we return Async::Ready
// so that we can return the writer and the buffer.
struct WriteAll<W, T>(<code>Option<Writing<W, T>></code>);

struct Writing<W, T> {
    // This is the stream we're writing into.
    writer: W,
    // This is the buffer we're writing from.
    buffer: T,
    // And this is much of the buffer we've written.
    pos: usize,
}

// We want to be able to construct a WriteAll over anything
// that implements AsyncWrite, and any buffer that can be
// thought of as a &[u8].
fn write_all<W, T>(writer: W, buffer: T) -> WriteAll<W, T>
where
    W: AsyncWrite,
    T: AsRef<[u8]>,
{
    WriteAll(<code>Some(Writing {
        writer,
        buffer,
        // Initially, we've written none of the bytes from buffer.
        pos: 0,
    })</code>)
}

impl<W, T> Future for WriteAll<W, T>
where
    W: AsyncWrite,
    T: AsRef<[u8]>,
{
    // When we've written out the entire buffer, we want to return
    // both the buffer and the writer so that the user can re-use them.
    type Item = (W, T);
}
```

```
type Error = io::Error;

fn poll(&mut self) -> Poll<Self::Item, Self::Error> {
    match self.0 {
        Some(Writing {
            ref mut writer,
            ref buffer,
            ref mut pos,
        }) => {
            let buffer = buffer.as_ref();
            // Check that we haven't finished
            while *pos < buffer.len() {
                // Try to write the remainder of the buffer into the
writer.
                // Just like write in std::io::Write, poll_write *can*
write
                // fewer bytes than the length of the buffer it is
given,
                // and we need to handle that by looking at its return
                // value, which is the number of bytes actually
written.
                //
                // We are using try_ready! here, just like in
poll_read in
                // ReadExact, so that if poll_write returns NotReady
(or an
                // error), we will do the same! We uphold the contract
that
                // we have arranged to be notified later because
poll_write
                // follows that same contract, and _it_ returned
NotReady.
                let n = try_ready!
(writer.poll_write(&buffer[*pos..]));
                *pos += n;

                // If no bytes were written, but there was no error,
this
                // generally implies that something weird happened
under us.
                // We make sure to turn this into an error for the
caller to
                // deal with.
                if n == 0 {
                    return Err(io::Error::new(
                        io::ErrorKind::WriteZero,
                        "zero-length write",
                    ));
                }
            }
        }
    }
    None => panic!("poll a WriteAll after it's done"),
}
}
```

```
// We use the same trick as in ReadExact to ensure that we can
return
    // the buffer and the writer once the entire buffer has been
written out.
    let writing = self.0.take().expect("must have seen Some above");
    Ok(Async::Ready((writing.writer, writing.buffer)))
}
}
```

Futures

在指南早期暗示的future是用于管理异步逻辑的构建块。它们是Tokio使用的底层异步抽象。

future的实施由future crate提供。但是，为方便起见，Tokio重新导出了许多类型。

Futures是什么

future是表示异步计算完成的值。通常，由于系统中某处发生的事件使future完成。虽然我们从基本I/O的角度看待事物，但您可以使用future来表示各种事件，例如：

- 在线程池中执行的数据库查询。当数据库查询完成时，future完成，其值是查询的结果。
- 对服务器的RPC调用。当服务器回复时，future完成，其值是服务器的响应。
- 超时事件。当时间到了，future就完成了，它的值是()。
- 在线程池上运行的长时间运行的CPU密集型任务。任务完成后，future完成，其值为任务的返回值。
- 从套接字读取字节。当字节准备就绪时，future就完成了 - 根据缓冲策略，字节可能直接返回，或作为副作用写入某个现有缓冲区。

future抽象的整个要点是允许异步函数，即不能立即返回值的函数，将会返回一些东西。

例如，异步HTTP客户端可以提供如下所示的get函数：

```
pub fn get(&self, uri: &str) -> ResponseFuture { ... }
```

然后，库的用户将使用该函数：

```
let response_future = client.get("https://www.example.com");
```

现在，response_future不是实际响应。这是个一旦收到响应，就会完成的future。但是，调用者要具有一个具体的东西（future）使他们可以开始使用它。例如，它们可以链式计算在接收到响应时执行，或者它们可能将future传递给函数。

```
let response_is_ok = response_future
    .map(|response| {
        response.status().is_ok()
    });
track_response_success(response_is_ok);
```

所有与future一起采取的行动都不会立即执行任何工作。他们不能，因为他们没有实际的HTTP响应。相反，他们定义了响应future完成时要完成的工作。

future crate和Tokio都有一系列组合功能，可以用来处理future。

实现future

使用Tokio时，实现Future是很常见的，因此适应它是很重要的。

如前一节所述，Rust future是基于轮询的。这是Rust future库的一个独特方面。其他编程语言的大多数future库使用基于推送的模型，其中回调被提供给future，并且计算立即调用计算结果回调。

使用基于轮询的模型提供了许多优点，包括作为零成本抽象，即，与手动编写异步代码相比，使用Rust future没有额外的开销。

future的特点如下：

```
trait Future {  
    /// The type of the value returned when the future completes.  
    type Item;  
  
    /// The type representing errors that occurred while processing the  
    /// computation.  
    type Error;  
  
    fn poll(&mut self) -> Result<Async<Self::Item>, Self::Error>;  
}
```

您可能会注意到这与用于实现异步任务的trait完全相同。这是因为一旦计算完成，异步任务就是“正好”的future，其值为()。

通常，当您实现Future时，您将定义一个由子（或内部）future组成的计算。在这种情况下，future的实现会尝试调用内部future，如果内部future未准备好，则返回NotReady。

以下示例是由另一个返回usize并将使该值加倍的future组成的future：

```
pub struct Doubler<T> {  
    inner: T,  
}  
  
pub fn double<T>(inner: T) -> Doubler<T> {  
    Doubler { inner }  
}  
  
impl<T> Future for Doubler<T>  
where T: Future<Item = usize>  
{  
    type Item = usize;  
    type Error = T::Error;  
  
    fn poll(&mut self) -> Result<Async<usize>, T::Error> {  
        match self.inner.poll()? {  
            Async::Ready(v) => Ok(Async::Ready(v * 2)),  
            ...  
        }  
    }  
}
```

```
        Async::NotReady => Ok(Async::NotReady),  
    }  
}  
{
```

当Doubler的future被轮询时，它会调查其内的future。如果内部future尚未准备好，Doubler future将返回NotReady。如果内心的future已经准备就绪，那么Doubler的future会使返回值加倍并返回Ready。

因为上面的匹配模式很常见，所以`future crate`提供了一个宏：`try_ready!`。它类似于`try!`或`?`，但它也返回`NotReady`。上面的`poll`函数可以使用`try_ready`重写！如下：

```
fn poll(&mut self) -> Result<Async<usize>, T::Error> {
    let v = try_ready!(self.inner.poll());
    Ok(Async::Ready(v * 2))
}
```

返回NotReady

当一个任务返回NotReady时，一旦它转换到就绪状态，执行者就会被通知。这使执行者能够有效地调度任务。

当函数返回`Async :: NotReady`时，在状态转换为“就绪”时通知执行程序至关重要。否则，任务将无限挂起，永远不会再运行。

对于大多数 `future` 的实现，这是可传递的。当 `future` 实现是子 `future` 的组合时，当至少一个内部 `future` 返回 `NotReady` 时，外部 `future` 仅返回 `NotReady`。因此，一旦内部 `future` 转变为就绪状态，外部 `future` 将转变为就绪状态。在这种情况下，`NotReady` 合约已经满足，因为内部 `future` 将在准备就绪时通知执行者。

最内层的future，有时也被称为“资源”，是负责通知执行人的人。这是通过对task :: current () 返回的任务调用notify来完成的

在执行者调用任务轮询之前，它将任务上下文设置为线程局部变量。然后，最内部的future从线程本地访问上下文，以便一旦其准备状态改变就能够通知任务。

我们将在后面的部分中更深入地探索实施资源和任务系统。除非你从内部的**future**获得**NotReady**，否则这里的关键是不要返回**NotReady**。

一个更复杂的future

让我们看一下稍微复杂的future实现。在这种情况下，我们将实现一个取得主机名，进行DNS解析，然后建立与远程主机的连接的future。我们假设存在一个如下所示的resolve函数：

```
pub fn resolve(host: &str) -> ResolveFuture:
```

其中**ResolveFuture**是一个返回**SocketAddr**的**future**。

实现future的步骤是：

1. 调用**resolve**以获取**ResolveFuture**实例。
2. 调用**ResolveFuture :: poll**直到它返回一个**SocketAddr**。
3. 将**SocketAddr**传递给**TcpStream :: connect**。
4. 调用**ConnectFuture :: poll**直到它返回**TcpStream**。
5. 使用**TcpStream**完成外部**future**。

我们将使用枚举来跟踪**future**的状态。

```
enum State {
    // Currently resolving the host name
    Resolving(ResolveFuture),

    // Establishing a TCP connection to the remote host
    Connecting(ConnectFuture),
}
```

ResolveAndConnect的**future**定义为：

```
pub struct ResolveAndConnect {
    state: State,
}
```

```
pub fn resolve_and_connect(host: &str) -> ResolveAndConnect {
    let state = State::Resolving(resolve(host));
    ResolveAndConnect { state }
}

impl Future for ResolveAndConnect {
    type Item = TcpStream;
    type Error = io::Error;

    fn poll(&mut self) -> Result<Async<TcpStream>, io::Error> {
        use self::State::*;

        loop {
            let addr = match self.state {
                Resolving(ref mut fut) => {
                    try_ready!(fut.poll())
                }
                Connecting(ref mut fut) => {
                    return fut.poll();
                }
            };
            let connecting = TcpStream::connect(&addr);
            self.state = Connecting(connecting);
        }
    }
}
```

```
    }  
}
```

这解释了Future如何实现状态机。这个future可以是两种状态中的任何一种：

1. Resolving
2. Connecting

每次调用poll时，我们都会尝试将状态机推进到下一个状态。

现在，我们刚刚实现的future基本上是AndThen，所以我们可能只是使用该组合器而不是重新实现它。

```
resolve(my_host)  
    .and_then(|addr| TcpStream::connect(&addr))
```

任务

任务是应用程序的“逻辑单元”。它们类似于Go的goroutine和Erlang的进程，但是异步。换句话说，任务是异步绿色线程。

鉴于任务运行异步逻辑位，它们由Future特征表示。任务完成处理后，任务的future实现将以 () 值完成。

任务被传递给执行程序，执行程序处理任务的调度。执行程序通常在一组或一组线程中调度许多任务。**任务不得执行计算繁重的逻辑，否则将阻止其他任务执行**。因此，不要尝试将斐波那契序列计算为任务。

任务通过直接实施Future特征或通过使用future和tokio crate中可用的各种组合函数构建future来实现。

下面是一个使用HTTP get从URI获取值并缓存结果的示例。

1. 检查缓存以查看是否存在URI条目。
2. 如果没有条目，请执行HTTP get。
3. 将响应存储在缓存中。
4. 返回response。

整个事件序列也包含超时，以防止无限制的执行时间。

```
// The functions here all return `Box<Future<...>>`. This is one
// of a number of ways to return futures. For more details on
// returning futures, see the "Returning futures" section in
// "Going deeper: Futures".

/// Get a URI from some remote cache.
fn cache_get(uri: &str)
    -> Box<Future<Item = Option<String>, Error = Error>>
{ ... }

fn cache_put(uri: &str, val: String)
    -> Box<Future<Item = (), Error = Error>>
{ ... }

/// Do a full HTTP get to a remote URL
fn http_get(uri: &str)
    -> Box<Future<Item = String, Error = Error>>
{ ... }

fn fetch_and_cache(url: &str)
    -> Box<Future<Item = String, Error = Error>>
{
    // The URL has to be converted to a string so that it can be
    // moved into the closure. Given futures are asynchronous,
    // the stack is not around anymore by the time the closure is called.
    let url = url.to_string();

    let response = http_get(&url)
```

```

        .and_then(move |response| {
            cache_put(&url, response.clone())
                .map(|_| response)
        });

    Box::new(response)
}

let url = "https://example.com";

let response = cache_get(url)
    .and_then(|resp| {
        // `Either` is a utility provided by the `futures` crate
        // that enables returning different futures from a single
        // closure without boxing.
        match resp {
            Some(resp) => Either::A(future::ok(resp)),
            None => {
                Either::B(fetch_and_cache(url))
            }
        }
    });
}

// Only let the task run for up to 20 seconds.
//
// This uses a fictional timer API. Use the `tokio-timer` crate for
// all your actual timer needs.
let task = Timeout::new(response, Duration::from_secs(20));

my_executor.spawn(task);

```

由于这些步骤对于完成任务都是必需的，因此将它们全部分组到同一任务中是有意义的。

但是，如果不是在缓存未命中时更新缓存，而是希望在一个时间间隔内更新缓存值，那么将其拆分为多个任务是有意义的，因为这些步骤不再直接相关。

```

let url = "https://example.com";

// An Interval is a stream that yields `()` on a fixed interval.
let update_cache = Interval::new(Duration::from_secs(60))
    // On each tick of the interval, update the cache. This is done
    // by using the same function from the previous snippet.
    .for_each(|_| {
        fetch_and_cache(url)
            .map(|resp| println!("updated cache with {}", resp))
    });

// Spawn the cache update task so that it runs in the background
my_executor.spawn(update_cache);

```

```
// Now, only get from the cache.  
// (NB: see next section about ensuring the cache is up to date.)  
let response = cache_get(url);  
let task = Timeout::new(response, Duration::from_secs(20));  
  
my_executor.spawn(task);
```

消息传递

就像Go和Erlang一样，任务可以使用消息传递进行通信。实际上，使用消息传递来协调多个任务是很常见的。这允许独立任务仍然相互作用。

`future`包提供了一个同步模块，其中包含一些适合跨任务传递消息的通道类型。

- `oneshot`是一个用于发送一个值的通道。
- `mpsc`是用于发送许多（零个或多个）值的通道。

前面的例子并不完全正确。鉴于任务同时执行，无法保证缓存更新任务在其他任务尝试从缓存中读取时将第一个值写入缓存。

这是使用消息传递的完美情况。高速缓存更新任务可以发送消息，通知其他任务它已使用初始值启动了高速缓存。

```
let url = "https://example.com";  
  
let (primed_tx, primed_rx) = oneshot::channel();  
  
let update_cache = fetch_and_cache(url)  
    // Now, notify the other task that the cache is primed  
    .then(|_| primed_tx.send(()))  
    // Then we can start refreshing the cache on an interval  
    .then(|_| {  
        Interval::new(Duration::from_secs(60))  
            .for_each(|_| {  
                fetch_and_cache(url)  
                    .map(|resp| println!("updated cache with {}", resp))  
            })  
    })  
    );  
  
    // Spawn the cache update task so that it runs in the background  
    my_executor.spawn(update_cache);  
  
    // First, wait for the cache to primed  
    let response = primed_rx  
        .then(|_| cache_get(url));  
  
    let task = Timeout::new(response, Duration::from_secs(20));  
  
    my_executor.spawn(task);
```

任务通知

使用Tokio构建的应用程序被构造为一组并发运行的任务。这是服务器的基本结构：

```
let server = listener.incoming().for_each(|socket| {
    // Spawn a task to process the connection
    tokio::spawn(process(socket));

    Ok(())
})
.map_err(|_| () ); // Just drop the error

tokio::run(server);
```

在这种情况下，我们为每个入站服务器套接字生成一个任务。但是，也可以实现处理同一套接字上所有入站连接的服务器future：

```
pub struct Server {
    listener: TcpListener,
    connections: Vec<Box<Future<Item = (), Error = io::Error> + Send>>,
}

impl Future for Server {
    type Item = ();
    type Error = io::Error;

    fn poll(&mut self) -> Result<Async<()>, io::Error> {
        // First, accept all new connections
        loop {
            match self.listener.poll_accept()? {
                Async::Ready((socket, _)) => {
                    let connection = process(socket);
                    self.connections.push(connection);
                }
                Async::NotReady => break,
            }
        }

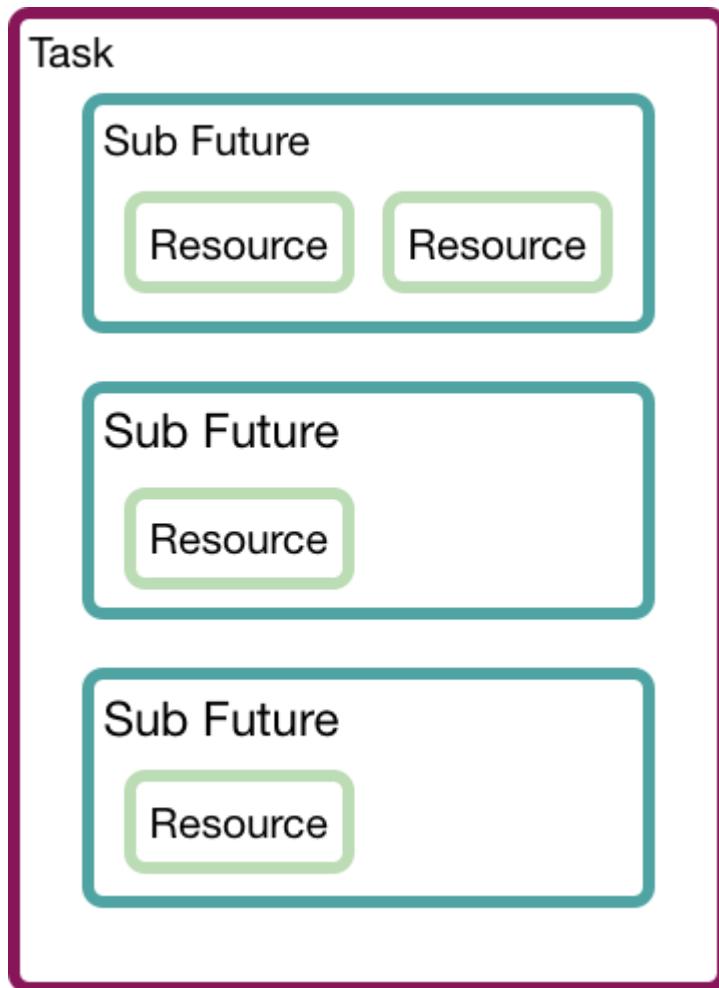
        // Now, poll all connection futures.
        let len = self.connections.len();

        for i in (0..len).rev() {
            match self.connections[i].poll()? {
                Async::Ready(_) => {
                    self.connections.remove(i);
                }
                Async::NotReady => {}
            }
        }
    }
}
```

```
// `NotReady` is returned here because the future never actually
// completes. The server runs until it is dropped.
Ok(Async::NotReady)
}
}
```

这两种策略在功能上是等效的，但具有明显不同的运行时特性。

通知发生在任务级别。该任务不知道哪个子future触发了通知。因此，无论何时轮询任务，都必须尝试轮询所有子future。



在此任务中，有三个子future可以进行轮询。如果其中一个子future所包含的资源转为“就绪”，则任务本身会收到通知，并会尝试轮询其所有三个子future。其中一个将推进，这反过来推进任务的内部状态。

关键是尽量减少任务，尽可能少地完成每项任务。这就是为什么服务器为每个连接生成新任务而不是在与侦听器相同的任务中处理连接的原因。

好吧，实际上有一种方法可以让任务知道哪个子future使用FuturesUnordered触发了通知，但通常正确的做法是生成一个新任务。

运行时模型

现在我们将介绍Tokio /**future**运行时模型。 Tokio构建在**future**箱顶部并使用其运行时模型。 这允许它也使用**future**箱与其他图书馆互操作。

注意：此运行时模型与其他语言中的异步库非常不同。 虽然在较高的层面上， API看起来很相似，但代码执行方式却有所不同。

同步模型

首先，让我们简要谈谈同步（或阻塞）模型。 这是Rust标准库使用的模型。

```
// let socket = ...;
let mut buf = [0; 1024];
let n = socket.read(&mut buf).unwrap();

// Do something with &buf[..n];
```

调用socket.read时，无论套接字在其缓冲区中是否有待处理数据，如果有待处理的数据，则读取的调用将立即返回，并且buf将填充该数据。如果没有未决数据，则read函数将阻止当前线程，直到收到数据。此时，buf将填充此新接收的数据，并且将返回**read**函数

为了同时在许多不同的套接字上并发执行读取，每个套接字需要一个线程。 每个套接字使用一个线程不能很好地扩展到大量的套接字。 这被称为c10k问题。

非阻塞套接字

在执行像**read**这样的操作时避免阻塞线程的方法是不阻塞线程！ 当套接字在其接收缓冲区中没有未决数据时，**read**函数立即返回，表明套接字“未准备好”以执行读取操作。

使用Tokio **TcpStream**时，如果没有要读取的待处理数据，则对**read**的调用将返回类型**ErrorKind :: WouldBlock**的错误。 此时，调用者负责稍后再次调用**read**。 诀窍是知道“晚些时候”的时间。

考虑非阻塞读取的另一种方法是“轮询”套接字以读取数据。

轮询模型

轮询套接字数据的策略可以推广到任何操作。 例如，在轮询模型中获取“小部件”的函数看起来像这样：

```
fn poll_widget() -> Async<Widget> { ... }
```

此函数返回**Async <Widget>**，其中**Async**是**Ready (Widget)** 或**NotReady**的枚举。 **Async**枚举由**future**箱提供，是轮询模型的构建块之一。

现在，让我们定义一个没有使用此**poll_widget**函数的组合器的异步任务。 该任务将执行以下操作：

1. 获取小部件。
2. 将小部件打印到STDOUT。
3. 终止任务。

为了定义任务，我们实现了Future trait。

```
//轮询单个小部件并将其写入STDOUT的任务。
pub struct MyTask;

impl Future for MyTask {
    type Item = ();
    type Error = ();

    fn poll(&mut self) -> Result<Async<()>, ()> {
        match poll_widget() {
            Async::Ready(widget) => {
                println!("widget={:?}", widget);
                Ok(Async::Ready(()))
            }
            Async::NotReady => {
                return Ok(Async::NotReady);
            }
        }
    }
}
```

重要提示： 返回Async :: NotReady具有特殊含义。有关详细信息，请参阅下一节。

需要注意的关键是，当调用MyTask :: poll时，它会立即尝试获取小部件。如果对poll_widget的调用返回NotReady，则该任务无法继续进行。然后任务返回NotReady，表明它尚未准备好完成处理。

任务实现不会阻止。相反，“将来的某个时间”，执行者将再次调用MyTask :: poll。将再次调用poll_widget。如果poll_widget已准备好返回窗口小部件，则该任务又可以打印窗口小部件。然后，可以通过返回Ready来完成任务。

执行者(Executors)

为了使任务取得进展，必须调用MyTask :: poll。这就是执行者的工作。

执行程序负责反复调用任务轮询，直到返回Ready。有很多不同的方法可以做到这一点。例如，CurrentThread执行者将阻止当前线程并遍历所有生成的任务，并对它们调用poll。ThreadPool在线程池中调度任务。这也是运行时使用的默认执行者。

必须在执行者上生成所有任务，否则不会执行任何工作。

在最简单的情况下，执行者可能看起来像这样：

```
pub struct SpinExecutor {
    tasks: VecDeque<Box<Future<Item = (), Error = ()>>>,
}
```

```
impl SpinExecutor {
    pub fn spawn<T>(&mut self, task: T)
    where T: Future<Item = (), Error = ()> + 'static
    {
        self.tasks.push_back(Box::new(task));
    }

    pub fn run(&mut self) {
        while let Some(mut task) = self.tasks.pop_front() {
            match task.poll().unwrap() {
                Async::Ready(_) => {}
                Async::NotReady => {
                    self.tasks.push_back(task);
                }
            }
        }
    }
}
```

当然，这不会非常有效。执行程序在一个繁忙的循环中运转并尝试轮询所有任务，即使任务将再次返回NotReady。

理想情况下，执行者可以通过某种方式知道任务的“准备就绪”状态何时被改变，即当轮询调用返回Ready时。执行者看起来像这样：

```
pub fn run(&mut self) {
    loop {
        while let Some(mut task) = self.ready_tasks.pop_front() {
            match task.poll().unwrap() {
                Async::Ready(_) => {}
                Async::NotReady => {
                    self.not_ready_tasks.push_back(task);
                }
            }
        }

        if self.not_ready_tasks.is_empty() {
            return;
        }

        // Put the thread to sleep until there is work to do
        self.sleep_until_tasks_are_ready();
    }
}
```

当任务从“未准备好”变为“准备好”时能够得到通知是future任务模型的核心。我们将很快进一步深入研究。

Tokio与I/O

tokio crate带有TCP和UDP网络类型。与std中的类型不同，Tokio的网络类型基于轮询模型，并在其准备状态发生变化（接收数据并刷写写入缓冲区）时通知任务执行程序。在tokio :: net模块中，您将找到TcpListener, TcpStream和UdpSocket等类型。

所有这些类型都提供了future的API以及poll API。

Tokio网络类型由基于Mio的反应器提供动力，默认情况下，它在后台线程上懒洋洋地启动。有关详细信息，请参阅reactor文档。

使用Future API 我们已经在本指南的前面已经看到了一些传入函数以及tokio_io :: io中的助手。

这些助手包括：

- incoming: 入站TCP连接流。
- read_exact: 准确读取n个字节到缓冲区。
- read_to_end: 将所有字节读入缓冲区。
- write_all: 写入缓冲区的全部内容。
- copy: 将字节从一个I / O句柄复制到另一个I / O句柄。

很多这些函数/帮助程序都是AsyncRead和AsyncWrite特性的通用函数。这些特征类似于std的Read和Write，但仅适用于“future感知”的类型，即遵循强制属性：

- 调用读取或写入是非阻塞的，它们永远不会阻塞调用线程。
- 如果一个调用会以其他方式阻塞，那么会返回一个带有此类WillBlock的错误。如果发生这种情况，则当前future的任务计划在I / O再次准备就绪时接收通知（取消停放）

请注意 AsyncRead和AsyncWrite类型的用户应使用poll_read和poll_write，而不是直接调用read和write。

例如，以下是如何接受连接，从它们读取5个字节，然后将5个字节写回套接字：

```
let server = listener.incoming().for_each(|socket| {
    println!("accepted socket; addr={:?}", socket.peer_addr().unwrap());

    let buf = vec![0; 5];

    let connection = io::read_exact(socket, buf)
        .and_then(|(socket, buf)| {
            io::write_all(socket, buf)
        })
        .then(|_| Ok(())); // Just discard the socket and buffer

    // Spawn a new task that processes the socket:
    tokio::spawn(connection);

    Ok(())
})
```

使用Poll API

手动实现Future时将使用基于Poll的API，您需要返回Async。当您需要实现自己的处理自定义逻辑的组合器时，这非常有用。

例如，这就是如何为TcpStream实现read_exact的future。

```
pub struct ReadExact {
    state: State,
}

enum State {
    Reading {
        stream: TcpStream,
        buf: Vec<u8>,
        pos: usize,
    },
    Empty,
}

impl Future for ReadExact {
    type Item = (TcpStream, Vec<u8>);
    type Error = io::Error;

    fn poll(&mut self) -> Result<Async<Self::Item>, io::Error> {
        match self.state {
            State::Reading {
                ref mut stream,
                ref mut buf,
                ref mut pos
            } => {
                while *pos < buf.len() {
                    let n = try_ready!({
                        stream.poll_read(&mut buf[*pos..])
                    });
                    *pos += n;
                    if n == 0 {
                        let err = io::Error::new(
                            io::ErrorKind::UnexpectedEof,
                            "early eof");
                        return Err(err)
                    }
                }
            State::Empty => panic!("poll a ReadExact after it's done"),
        }

        match mem::replace(&mut self.state, State::Empty) {
            State::Reading { stream, buf, .. } => {
                Ok(Async::Ready((stream, buf)))
            }
        }
    }
}
```

```
        State::Empty => panic!(),
    }
}
```

数据报(Datagrams)

请注意，大多数讨论都是围绕I / O或字节流进行的，而UDP重要的不是！但是，为了适应这种情况，`UdpSocket`类型还提供了许多方便的方法：

- `send_dgram`允许您表示将数据报作为`future`发送，如果无法立即发送整个数据报，则返回错误。
- `recv_dgram`表示将数据报读入缓冲区，产生缓冲区和来自的地址。

示例：聊天服务器

我们将使用到目前为止已经涵盖的内容来构建聊天服务器。这是一个非平凡的Tokio服务器应用程序。

服务器将使用基于行的协议。行以`\r\n`结束。这与telnet兼容，因此我们只使用telnet作为客户端。当客户端连接时，它必须通过发送包含其“缺口”的行来标识自己（即，用于在其 `Peer` 中标识客户端的某个名称）。

识别出客户端后，所有发送的行都以`[nick]:`为前缀，并广播给所有其他连接的客户端。

完整的代码可以[在这里](#)找到。请注意，Tokio提供了一些尚未涵盖的额外抽象，这些抽象将使聊天服务器能够用更少的代码编写。

首先，生成一个新的箱子。

```
$ cargo new --bin line-chat
cd line-chat
```

接下来，添加必要的依赖项：

```
[dependencies]
tokio = "0.1"
tokio-io = "0.1"
futures = "0.1"
bytes = "0.4"
```

```
extern crate tokio;
#[macro_use]
extern crate futures;
extern crate bytes;

use tokio::io;
use tokio::net::{TcpListener, TcpStream};
use tokio::prelude::*;
use futures::sync::mpsc;
use futures::future::{self, Either};
use bytes::{BytesMut, Bytes, BufMut};

use std::collections::HashMap;
use std::net::SocketAddr;
use std::sync::{Arc, Mutex};

/// Shorthand for the transmit half of the message channel.
type Tx = mpsc::UnboundedSender<Bytes>;

/// Shorthand for the receive half of the message channel.
type Rx = mpsc::UnboundedReceiver<Bytes>;
```

现在，我们为服务器设置必要的结构。这些步骤与Hello World中使用的步骤相同！例：

- 将TcpListener绑定到本地端口。
- 定义接受入站连接并处理它们的任务。
- 启动Tokio运行时
- 产生服务器任务。

同样，在执行程序上生成服务器任务之前，实际上不会发生任何工作。

```
fn main() {
    let addr = "127.0.0.1:6142".parse().unwrap();
    let listener = TcpListener::bind(&addr).unwrap();

    let server = listener.incoming().for_each(move |socket| {
        // TODO: Process socket
        Ok(())
    })
    .map_err(|err| {
        // Handle error by printing to STDOUT.
        println!("accept error = {:?}", err);
    });

    println!("server running on localhost:6142");

    // Start the server
    //
    // This does a few things:
    //
    // * Start the Tokio runtime (reactor, threadpool, etc...)
    // * Spawns the `server` task onto the runtime.
    // * Blocks the current thread until the runtime becomes idle, i.e.
all
    //     spawned tasks have completed.
    tokio::run(server);
}
```

Chat State

聊天服务器要求从一个客户端接收的消息被广播到所有其他连接的客户端。这将使用通过mpsc通道传递的消息来完成。

每个客户端套接字都将由任务管理。每个任务都有一个关联的mpsc通道，用于接收来自其他客户端的消息。所有这些通道的发送一半存储在Rc单元中以使它们可访问。

在这个例子中，我们将使用无界通道。理想情况下，渠道永远不应该是无限制的，但在这种情况下处理背压有点棘手。我们将把通道限制在后面专门用于处理背压的部分。

以下是共享状态的定义方式（上面已完成Tx类型别名）：

```
struct Shared {
    peers: HashMap<SocketAddr, Tx>,
}
```

然后，在main函数的最顶部，创建状态实例。此状态实例将移动到接受传入连接的任务中。

```
let state = Arc::new(Mutex::new(Shared::new()));
```

现在我们可以处理传入的连接。服务器任务更新为：

```
listener.incoming().for_each(move |socket| {
    process(socket, state.clone());
    Ok(())
})
```

服务器任务将所有套接字以及服务器状态的克隆传递给进程函数。我们来定义那个功能。它将具有这样的结构：

```
fn process(socket: TcpStream, state: Arc<Mutex<Shared>>) {
    // Define the task that processes the connection.
    let task = unimplemented!();

    // Spawn the task
    tokio::spawn(task);
}
```

对tokio :: spawn的调用将在当前的Tokio运行时生成一个新任务。所有工作线程都保留对存储在线程局部变量中的当前运行时的引用。注意，尝试从Tokio运行时外部调用tokio :: spawn将导致恐慌。

所有连接处理逻辑必须能够理解协议。该协议是基于行的，由\r\n终止。它不是在字节流级别工作，而是更容易在帧级工作，即使用表示原子消息的值。

我们实现了一个包含套接字的编解码器，并公开了一个采用和消耗行的API。

线性编解码器

对于采用字节流类型（AsyncRead + AsyncWrite）并在帧级别公开读写API的类型，编解码器是一个松散术语。tokio-io crate为编写编解码器提供了额外的帮助，在这个例子中，我们将手动完成。

Lines编解码器定义如下：

```
struct Lines {
    socket: TcpStream,
    rd: BytesMut,
```

```

        wr: BytesMut,
    }

impl Lines {
    /// Create a new `Lines` codec backed by the socket
    fn new(socket: TcpStream) -> Self {
        Lines {
            socket,
            rd: BytesMut::new(),
            wr: BytesMut::new(),
        }
    }
}

```

从套接字读取的数据缓冲到rd中。读取完整行后，将返回给调用者。调用者提交以写入套接字的行被缓冲到wr中，然后刷新。

这是读取一半的实现方式：

```

impl Stream for Lines {
    type Item = BytesMut;
    type Error = io::Error;

    fn poll(&mut self) -> Result<Async<Option<Self::Item>>, Self::Error> {
        // First, read any new data that might have been received
        // off the socket
        //
        // We track if the socket is closed here and will be used
        // to inform the return value below.
        let sock_closed = self.fill_read_buf()?.is_ready();

        // Now, try finding lines
        let pos = self.rd.windows(2)
            .position(|bytes| bytes == b"\r\n");

        if let Some(pos) = pos {
            // Remove the line from the read buffer and set it
            // to `line`.
            let mut line = self.rd.split_to(pos + 2);

            // Drop the trailing \r\n
            line.split_off(pos);

            // Return the line
            return Ok(Async::Ready(Some(line)));
        }

        if sock_closed {
            Ok(Async::Ready(None))
        } else {
            Ok(Async::NotReady)
        }
    }
}

```

```

        }
    }

impl Lines {
    fn fill_read_buf(&mut self) -> Result<Async<()>, io::Error> {
        loop {
            // Ensure the read buffer has capacity.
            //
            // This might result in an internal allocation.
            self.rd.reserve(1024);

            // Read data into the buffer.
            //
            // The `read_buf` fn is provided by `AsyncRead`.
            let n = try_ready!(self.socket.read_buf(&mut self.rd));

            if n == 0 {
                return Ok(Async::Ready(()));
            }
        }
    }
}

```

该示例使用字节包中的BytesMut。这为在网络环境中处理字节序列提供了一些很好的实用程序。Stream实现产生的BytesMut值只包含一行。

与往常一样，实现返回Async的函数的关键是永远不会返回Async :: NotReady，除非函数实现收到Async :: NotReady本身。在此示例中，仅当fill_read_buf返回NotReady时才返回NotReady，如果TcpStream :: read_buf返回NotReady，则fill_read_buf仅返回NotReady。

```

struct Lines {
    socket: TcpStream,
    rd: BytesMut,
    wr: BytesMut,
}

impl Lines {
    fn buffer(&mut self, line: &[u8]) {
        // Push the line onto the end of the write buffer.
        //
        // The `put` function is from the `BufMut` trait.
        self.wr.put(line);
    }

    fn poll_flush(&mut self) -> Poll<(), io::Error> {
        // As long as there is buffered data to write, try to write it.
        while !self.wr.is_empty() {
            // Try to write some bytes to the socket
            let n = try_ready!(self.socket.poll_write(&self.wr));

            // As long as the wr is not empty, a successful write should
        }
    }
}

```

```

        // never write 0 bytes.
        assert!(n > 0);

        // This discards the first `n` bytes of the buffer.
        let _ = self.wr.split_to(n);
    }

    Ok(Async::Ready(()))
}
}

fn main() {}

```

调用者通过调用缓冲区对所有行进行排队。这会将该行附加到内部wr缓冲区。然后，一旦所有数据排队，调用者就会调用poll_flush，它会对套接字进行实际写入操作。poll_flush仅在所有排队数据成功写入套接字后才返回Ready。

与读取半部分类似，仅在函数实现收到NotReady本身时返回NotReady。

Lines编解码器在进程函数中使用如下：

```

fn process(socket: TcpStream, state: Arc<Mutex<Shared>>) {
    // Wrap the socket with the `Lines` codec that we wrote above.
    let lines = Lines::new(socket);

    // The first line is treated as the client's name. The client
    // is not added to the set of connected peers until this line
    // is received.
    //
    // We use the `into_future` combinator to extract the first
    // item from the lines stream. `into_future` takes a `Stream`
    // and converts it to a future of `(first, rest)` where `rest`
    // is the original stream instance.
    let connection = lines.into_future()
        // `into_future` doesn't have the right error type, so map
        // the error to make it work.
        .map_err(|(e, _)| e)
        // Process the first received line as the client's name.
        .and_then(|(name, lines)| {
            let name = match name {
                Some(name) => name,
                None => {
                    // TODO: Handle a client that disconnects
                    // early.
                    unimplemented!();
                }
            };
            //
            // TODO: Rest of the process function
        });
}

```

广播消息

下一步是实现处理实际聊天功能的连接处理逻辑，即从一个客户端向所有其他客户端广播消息。

为了实现这一点，我们将明确地实现一个Future，它接受Lines编解码器实例并处理广播逻辑。这个逻辑处理：

- 在其消息通道上接收消息并将其写入套接字。
- 从套接字接收消息并将其广播给所有 Peer。

完全使用组合器实现此逻辑也是可能的，但需要使用拆分，但尚未涉及。此外，这提供了一个机会，可以看到如何手动实现一个非平凡的 future。

以下是处理连接的广播逻辑的 future 定义：

```
struct Peer {
    /// Name of the peer. This is the first line received from the client.
    name: BytesMut,

    /// The TCP socket wrapped with the `Lines` codec.
    lines: Lines,

    /// Handle to the shared chat state.
    state: Arc<Mutex<Shared>>,

    /// Receive half of the message channel.
    ///
    /// This is used to receive messages from peers. When a message is
    received
    /// off of this `Rx`, it will be written to the socket.
    rx: Rx,

    /// Client socket address.
    ///
    /// The socket address is used as the key in the `peers` HashMap. The
    /// address is saved so that the `Peer` drop implementation can clean
    up its
    /// entry.
    addr: SocketAddr,
}
```

并且创建如下 Peer 实例：

```
impl Peer {
    fn new(name: BytesMut,
           state: Arc<Mutex<Shared>>,
           lines: Lines) -> Peer
    {
        // Get the client socket address
        let addr = lines.socket.peer_addr().unwrap();
```

```
// Create a channel for this peer
let (tx, rx) = mpsc::unbounded();

// Add an entry for this `Peer` in the shared state map.
state.lock().unwrap()
    .peers.insert(addr, tx);

Peer {
    name,
    lines,
    state,
    rx,
    addr,
}
}
```

为其他 **Peer** 创建 `mpsc` 通道，以将其消息发送到此新创建的 **Peer**。在创建信道之后，将发送半部分插入 **Peer** 映射中。此条目在 **Peer** 的 `drop` 实现中删除。

```
impl Drop for Peer {
    fn drop(&mut self) {
        self.state.lock().unwrap().peers
            .remove(&self.addr);
    }
}
```

这是实现

```
impl Future for Peer {
    type Item = ();
    type Error = io::Error;

    fn poll(&mut self) -> Poll<(), io::Error> {
        // Receive all messages from peers.
        loop {
            // Polling an `UnboundedReceiver` cannot fail, so `unwrap`
            // here is safe.
            match self.rx.poll().unwrap() {
                Async::Ready(Some(v)) => {
                    // Buffer the line. Once all lines are buffered,
                    // they will be flushed to the socket (right
                    // below).
                    self.lines.buffer(&v);
                }
                _ => break,
            }
        }
    }
}
```

```

// Flush the write buffer to the socket
let _ = self.lines.poll_flush()?;

// Read new lines from the socket
while let Async::Ready(line) = self.lines.poll()? {
    println!("Received line ({:?}) : {:?}", self.name, line);

    if let Some(message) = line {
        // Append the peer's name to the front of the line:
        let mut line = self.name.clone();
        line.put(": ");
        line.put(&message);
        line.put("\r\n");

        // We're using `Bytes`, which allows zero-copy clones
        // (by storing the data in an Arc internally).
        //
        // However, before cloning, we must freeze the data.
        // This converts it from mutable -> immutable,
        // allowing zero copy cloning.
        let line = line.freeze();

        // Now, send the line to all other peers
        for (addr, tx) in &self.state.lock().unwrap().peers {
            // Don't send the message to ourselves
            if *addr != self.addr {
                // The send only fails if the rx half has been
                // dropped, however this is impossible as the
                // `tx` half will be removed from the map
                // before the `rx` is dropped.
                tx.unbounded_send(line.clone()).unwrap();
            }
        }
    } else {
        // EOF was reached. The remote client has disconnected.
        // There is nothing more to do.
        return Ok(Async::Ready(()));
    }
}

// As always, it is important to not just return `NotReady`
// without ensuring an inner future also returned `NotReady`.
//
// We know we got a `NotReady` from either `self.rx` or
// `self.lines`, so the contract is respected.
Ok(Async::NotReady)
}
}

```

剩下的就是连接刚刚实施的Peer `future`。为此，将客户端连接任务（在`process`函数中定义）扩展为使用Peer。

```

let connection = lines.into_future()
    .map_err(|(e, _)| e)
    .and_then(|(name, lines)| {
        // If `name` is `None`, then the client disconnected without
        // actually sending a line of data.
        //
        // Since the connection is closed, there is no further work
        // that we need to do. So, we just terminate processing by
        // returning `future::ok()`.

        //
        // The problem is that only a single future type can be
        // returned from a combinator closure, but we want to
        // return both `future::ok()` and `Peer` (below).

        //
        // This is a common problem, so the `futures` crate solves
        // this by providing the `Either` helper enum that allows
        // creating a single return type that covers two concrete
        // future types.

        let name = match name {
            Some(name) => name,
            None => {
                // The remote client closed the connection without
                // sending any data.
                return Either::A(future::ok(()));
            }
        };

        println!("`{:?}` is joining the chat", name);

        // Create the peer.
        //
        // This is also a future that processes the connection, only
        // completing when the socket closes.

        let peer = Peer::new(
            name,
            state,
            lines);

        // Wrap `peer` with `Either::B` to make the return type fit.
        Either::B(peer)
    })
    // Task futures have an error of type `()` , this ensures we handle
    // the error. We do this by printing the error to STDOUT.
    .map_err(|e| {
        println!("connection error = {:?}", e);
    });
}

```

除了添加Peer之外，还会处理name == None。在这种情况下，远程客户端在识别自身之前终止。

返回多个 `future` (`name == None` handler和 `Peer`) 通过将返回的 `future` 包装在 `Either` 中来处理。要么是枚举，要为每个变体接受不同的 `future` 类型。这允许返回多个 `future` 类型而不到达 `trait` 对象。

定时器

在编写基于网络的应用程序时，通常需要根据时间执行操作。

- 在一段时间后运行一些代码。
- 取消运行时间过长的运行操作。
- 以一定间隔重复执行操作。

这些用例通过使用计时器模块中提供的各种计时器API来处理。

一段时间后运行代码

在这种情况下，我们希望在一段时间后执行任务。为此，我们使用Delay API。我们要做的就是写“Hello world!”到终端，但此时可以采取任何行动。

```
use tokio::prelude::*;
use tokio::timer::Delay;

use std::time::{Duration, Instant};

fn main() {
    let when = Instant::now() + Duration::from_millis(100);
    let task = Delay::new(when)
        .and_then(|_| {
            println!("Hello world!");
            Ok(())
        })
        .map_err(|e| panic!("delay errored; err={:?}", e));

    tokio::run(task);
}
```

上面的示例创建了一个新的Delay实例，该实例将在future 100毫秒内完成。新函数需要一个Instant，所以我们计算从现在起100毫秒的瞬间。

到达瞬间后，延迟future完成，从而导致执行and_then块。

与所有future一样，延迟是懒惰的。简单地创建一个新的Delay实例什么都不做。该实例必须用于生成到Tokio运行时的任务。运行时预先配置了一个计时器实现，以驱动Delay实例完成。在上面的示例中，这是通过将任务传递给tokio :: run来完成的。使用tokio :: spawn也可以。

计时耗时操作

在编写健壮的网络应用程序时，确保在合理的时间内完成操作至关重要。在等待来自外部，可能不受信任的来源的数据时尤其如此。

Timeout类型确保操作在指定的时刻完成。

```
use tokio::io;
use tokio::net::TcpStream;
use tokio::prelude::*;

use std::time::{Duration, Instant};

fn read_four_bytes(socket: TcpStream)
    -> Box<Future<Item = (TcpStream, Vec<u8>), Error = ()>>
{
    let buf = vec![0; 4];
    let fut = io::read_exact(socket, buf)
        .timeout(Duration::from_secs(5))
        .map_err(|_| println!("failed to read 4 bytes by timeout"));

    Box::new(fut)
}
```

上面的函数接受一个套接字并返回一个从套接字读取4个字节后完成的future。读取必须在5秒内完成。通过在读取future上调用超时来确保这一点，持续时间为5秒。

超时函数由FutureExt定义，包含在前奏中。因此，使用tokio :: prelude :: *也会导入FutureExt，因此我们可以在所有future上调用超时，以便要求它们在指定的瞬间完成。

如果在没有读取完成的情况下达到超时，则自动取消读取操作。当io :: read_exact返回的future被删除时会发生这种情况。由于延迟的运行时模型，删除future会导致操作被取消。

在间隔时间段上运行代码

在一个时间间隔内重复运行代码对于在套接字上发送PING消息或经常检查配置文件等情况很有用。这可以通过重复创建延迟值来实现。但是，因为这是一种常见模式，所以提供了Interval。

Interval类型实现Stream，以指定的速率产生。

```
use tokio::prelude::*;
use tokio::timer::Interval;

use std::time::{Duration, Instant};

fn main() {
    let task = Interval::new(Instant::now(), Duration::from_millis(100))
        .take(10)
        .for_each(|instant| {
            println!("fire; instant={:?}", instant);
            Ok(())
        })
        .map_err(|e| panic!("interval errored; err={:?}", e));

    tokio::run(task);
}
```

上面的例子创建了一个Interval，从现在开始每100毫秒产生一次（第一个参数是Interval应该首先触发的瞬间）。

默认情况下，即时流是无界的，即它将永久地以请求的间隔继续产生。该示例使用Stream :: take来限制Interval产生的次数，此处限制为10个事件的序列。因此，该示例将运行0.9秒，因为立即生成10个值中的第一个。

计时器的注意事项

Tokio计时器的粒度为1毫秒。任何较小的间隔都会向上舍入到最接近的毫秒。定时器在用户域中实现（即，不使用像linux上的timerfd这样的操作系统定时器）。它使用分层散列计时器轮实现，在创建，取消和触发超时时提供有效的恒定时间复杂度。

Tokio运行时包括每个工作线程一个计时器实例。这意味着，如果运行时启动4个工作线程，则将有4个计时器实例。这允许在大多数情况下避免同步，因为当使用计时器时，任务将在位于当前线程上的状态下操作。

也就是说，计时器实现是线程安全的，并支持从任何线程使用。

基础组合器

我们在`future`和`streams`概述中看到了一些最重要的组合。在这里，我们将再看一些。值得花一些时间使用特质文档来熟悉可用的全系列组合器 ([cheatsheet](#)) 。

一些具体的`future`和`streams`

任何价值都可以变成一个立即完整的`future`。`future`模块中有一些功能可用于创建这样的`future`:

- `ok`, 这类似于`Result :: Ok`: 它将你给它的值视为一个立即成功的`future`。
- `err`, 类似于`Result :: Err`: 它将您提供的值视为立即失败的`future`。
- `result`, 将结果提升到一个立即完整的`future`。

对于流，有一些“立即就绪”的流:

- `iter`, 它创建一个流，产生与底层迭代器相同的项。迭代器生成`Result`值，第一个错误终止带有该错误的流。
- `once`, 从结果中创建单元素流。

除了这些构造函数之外，还有一个函数，`lazy`，它允许您构建一个`future`，给出一个闭包，以便在以后按需生成该`future`。

IntoFuture

要了解的关键API是`IntoFuture` trait，它是可以转换为`future`的价值的 trait。您认为采取`future`的大多数API实际上都适用于此 trait。关键原因： trait是为`Result`实现的，允许您在预期`future`的许多地方返回结果值。

适配器

与`Iterator`一样，`Future`、`Stream`和`Sink` trait都配备了广泛的“适配器”方法。这些方法都使用接收对象并返回一个新的包装对象。对于`future`，您可以使用适配器：

- 更改`future`的类型 (`map`, `map_err`)
- 一个完成后运行另一个`future` (`then`, `and_then`, `or_else`)
- 弄清楚两个`future`中的哪一个先解决 (`select`)
- 等两个`future`都完成 (`join`)
- 转换为`trait object` (`Box :: new`)
- 将展开转换为错误 (`catch_unwind`)

对于流，有一大组适配器，包括：

- 许多与`Iterator`有共同点，如`map`, `fold`, `collect`, `filter`, `zip`, `take`, `skip`等。请注意`fold`和`collect`产生`future`，因此它们的结果是异步计算的。
- 用`future`排序的适配器 (`then`, `and_then`, `or_else`)
- 用于组合流的附加适配器 (`merge`, `select`)

`Sink` trait目前具有较少的适配器

最后，可以使用拆分适配器将既是流又是接收器的对象分解为单独的流和接收对象。

所有适配器都是零成本的，这意味着内部没有内存分配，并且实现将优化到您手动编写的内容。

错误处理

future, **streams**和**sinks**都将错误处理视为核心问题：它们都配备了相关的错误类型，并且各种适配器方法以合理的方式解释错误。例如：

- 序列组合器: `then`, `and_then`, `or_else`, `map`和`map_err`所有链错误类似于标准库中的`Result`类型。因此，例如，如果您使用`and_then`链接`future`并且第一个`future`因错误而失败，那么链式`future`永远不会运行。
- 像`select`和`join`这样的组合也可以处理错误。对于`select`，以任何方式完成的第一个`future`会产生一个答案，传播错误，但如果你想继续使用它，还可以访问另一个`future`。对于`join`，如果任何将来产生错误，则整个连接会产生该错误。

默认情况下，`future`对恐慌没有任何特殊处理。但是，在大多数情况下，`future`最终作为线程池中的任务运行，您需要捕获它们产生的任何恐慌并将其传播到其他地方。`catch_unwind`适配器可用于在不关闭工作线程的情况下将恐慌重新引入`Result`。

返回Futures

在处理 `future` 时，您可能需要做的第一件事就是返回 `future`。然而，与迭代器一样，这样做可能有点棘手。有几种选择，从大多数到最不符合人体工程学：

- Trait objects
- impl Trait
- Named types
- Custom types

Trait objects

首先，您始终可以选择返回一个Box `trait` 对象：

```
fn foo() -> Box<Future<Item = u32, Error = io::Error>> {  
    // ...  
}
```

这个策略的好处是它很容易写下来（只是一个Box）并且易于创建。就 `future` 的方法变化而言，这也是最灵活的，因为任何类型的 `future` 都可以作为不透明的Box Future返回。

这种方法的缺点是，在构建 `future` 时需要运行时分配，在使用该 `future` 时需要动态分派。Box 需要在堆上分配，然后将 `future` 放在里面。请注意，尽管这是此处唯一的分配，否则在执行 `future` 时不会进行任何分配。

通常可以通过仅在您想要返回的 `future` 长链的末尾装箱来降低成本，这仅需要整个链的单一分配和动态调度。

impl Trait

如果您使用的Rust版本大于1.26，那么您可以使用语言功能impl Trait。此语言功能将允许，例如：

```
fn add_10<F>(f: F) -> impl Future<Item = i32, Error = F::Error>  
    where F: Future<Item = i32>,  
{  
    f.map(|i| i + 10)  
}
```

这里我们用指定的关联类型指示返回类型是“实现Future的东西”。除此之外，我们通常会像往常一样使用 `future` 的组合器。

这种方法的优点在于它是零开销，没有Box需要，它对于 `future` 的实现是最大的灵活性，因为实际的返回类型是隐藏的，并且它符合人体工程学，因为它类似于上面的漂亮Box示例。

这种方法的缺点是只使用Box更灵活 - 如果你可能返回两种不同类型的Future, 然后你仍然必须返回Box `<Future <Item = F :: Item, Error = F :: Error>`而不是`impl Future <Item = F :: Item, Error = F :: Error>`。然而, 好消息是这种情况很少见; 一般来说, 它应该是一个向后兼容的扩展, 用于将返回类型从Box更改为`impl Trait`。

Named types

如果您不想返回Box并希望坚持使用旧版本的Rust, 另一种选择是直接编写返回类型:

```
fn add_10<F>(f: F) -> Map<F, fn(i32) -> i32>
    where F: Future<Item = i32>,
{
    fn do_map(i: i32) -> i32 { i + 10 }
    f.map(do_map)
}
```

这里我们将返回类型命名为编译器看到的完全一样。map函数返回Map结构, 该结构在内部包含future和执行map的函数。

这种方法的优点是它没有以前Box的运行时开销, 并且可以在1.26之前的Rust版本上运行。

然而, 缺点是, 通常很难命名这种类型。有时类型可能会变得非常大或完全无法命名。这里我们使用函数指针 (`fn (i32) -> i32`), 但我们理想情况下使用闭包。不幸的是, 返回类型暂时无法命名闭包。它还会导致非常详细的签名, 并向客户泄露实施细节。

Custom types

最后, 您可以将具体的返回类型包装在一个新类型中, 并为它实现`future`。例如:

```
struct MyFuture {
    inner: Sender<i32>,
}

fn foo() -> MyFuture {
    let (tx, rx) = oneshot::channel();
    // ...
    MyFuture { inner: tx }
}

impl Future for MyFuture {
    // ...
}
```

在这个例子中, 我们返回一个自定义类型MyFuture, 我们直接为它实现Future trait。此实现利用了底层的`Oneshot <i32>`, 但也可以在此处实现任何其他类型的协议。

这种方法的好处是它不需要Box分配, 它仍然是最大的灵活性。MyFuture的实现细节对外界是隐藏的, 因此可以在不破坏其他情况的情况下进行更改。

然而，这种方法的缺点是，这是返回 `future`最不符合人体工程学的方法。

使用构建流

Tokio有助手将字节流转换为帧流。字节流的示例包括TCP连接，管道，文件对象以及标准输入和输出文件描述符。在Rust中，流很容易识别，因为它们实现了读写 **trait**。

框架消息的最简单形式之一是行分隔消息。每条消息都以`n`字符结尾。让我们看一下如何使用tokio实现行分隔消息流。

编写编解码器

编解码器实现了**tokio_codec :: Decoder**和**tokio_codec :: Encoder trait**。它的工作是将帧转换为字节和从字节转换。这些 **trait**与**tokio_codec :: Framed**结构一起使用，以提供字节流的缓冲，解码和编码。

让我们看一下**LinesCodec**结构的简化版本，它实现了行分隔消息的解码和编码。

```
pub struct LinesCodec {
    // Stored index of the next index to examine for a `'\n` character.
    // This is used to optimize searching.
    // For example, if `decode` was called with `abc` , it would hold `3` ,
    // because that is the next index to examine.
    // The next time `decode` is called with `abcde\n` , the method will
    // only look at `de\n` before returning.
    next_index: usize,
}
```

这里的注释解释了，由于字节被缓冲直到找到一行，因此每次收到数据时从缓冲区的开头搜索`n`是很浪费的。保持缓冲区的最后长度并在收到新数据时从那里开始搜索更有效。

当在底层流上接收数据时，调用Decoder :: decode方法。该方法可以生成一个帧或返回Ok (None) 来表示它需要更多的数据来生成一个帧。解码方法负责通过使用BytesMut方法将其拆分来删除不再需要缓冲的数据。如果未删除数据，缓冲区将继续增长。

我们来看看如何为**LinesCodec**实现Decoder :: decode。

```
fn decode(&mut self, buf: &mut BytesMut) -> Result<Option<String>, io::Error> {
    // Look for a byte with the value '\n' in buf. Start searching from
    // the search start index.
    if let Some(newline_offset) =
        buf[self.next_index..].iter().position(|b| *b == b'\n')
    {
        // Found a '\n' in the string.

        // The index of the '\n' is at the sum of the start position + the
        // offset found.
        let newline_index = newline_offset + self.next_index;
    }
}
```

```

        // Split the buffer at the index of the '\n' + 1 to include the
        '\n'.
        // `split_to` returns a new buffer with the contents up to the
index.
        // The buffer on which `split_to` is called will now start at this
index.
    let line = buf.split_to(newline_index + 1);

        // Trim the '\n` from the buffer because it's part of the
protocol,
        // not the data.
    let line = &line[..line.len() - 1];

        // Convert the bytes to a string and panic if the bytes are not
valid utf-8.
    let line = str::from_utf8(&line).expect("invalid utf8 data");

        // Set the search start index back to 0.
    self.next_index = 0;

        // Return Ok(Some(...)) to signal that a full frame has been
produced.
    Ok(Some(line.to_string()))
} else {
    // '\n' not found in the string.

        // Tell the next call to start searching after the current length
of the buffer
        // since all of it was scanned and no '\n' was found.
    self.next_index = buf.len();

        // Ok(None) signifies that more data is needed to produce a full
frame.
    Ok(None)
}
}

```

当必须将帧写入底层流时，将调用Encoder :: encode方法。必须将帧写入作为参数接收的缓冲区。写入缓冲区的数据将在准备好发送数据时写入流。

现在让我们看看如何为LinesCodec实现Encoder :: encode。

```

fn encode(&mut self, line: String, buf: &mut BytesMut) -> Result<(),>
{
    // It's important to reserve the amount of space needed. The `bytes`>
API
    // does not grow the buffers implicitly.
    // Reserve the length of the string + 1 for the '\n'.
    buf.reserve(line.len() + 1);
}

```

```
// String implements IntoBuf, a trait used by the `bytes` API to work
with
// types that can be expressed as a sequence of bytes.
buf.put(line);

// Put the '\n' in the buffer.
buf.put_u8(b'\n');

// Return ok to signal that no error occurred.
Ok(())
}
```

编码信息通常更简单。 这里我们只需保留所需的空间并将数据写入缓冲区。

使用编解码器

使用编解码器的最简单方法是使用Framed结构。 它是实现自动缓冲的编解码器的包装器。 Framed结构体既是Stream又是Sink。 因此，您可以从中接收帧并向其发送帧。

您可以使用AsyncRead :: framed方法使用任何实现AsyncRead和AsyncWrite **trait**的类型创建Framed结构。

```
TcpStream::connect(&addr).and_then(|sock| {
    let framed_sock = Framed::new(sock, LinesCodec::new());
    framed_sock.for_each(|line| {
        println!("Received line {}", line);
        Ok(())
    })
});
```

构建一个运行时

运行时 - 运行事件驱动的应用程序所需的所有部分 - 已经可用。如果你只想使用tokio，你不需要知道这个。但是，知道底层发生了什么可能是有用的，既可以在出现问题时更多地了解细节，也可以在运行时生成器支持之外进行自定义。

我们将构建一个单线程运行时，因为它组合起来稍微简单一些。并不是说默认的多线程版本在概念上会更复杂，但是有更多的移动部分。了解这里的细节可以成为读取默认运行时代码的垫脚石。

可以在git存储库中找到此处讨论的完整，有效的[示例](#)。

Park trait

异步世界本质上是在等待某事发生（并且能够一次等待多个事物）。毫无疑问，抽象等待是一种特质。它叫做Park。

这个想法是，如果没有更好的事情要做，控制权将被传递到Park直到发生一些有趣的事情并且控制权再次被带走或直到某个指定的时间过去。Park如何花费这段时间。它可以做一些有用的事情（处理后台作业）或者只是以某种方式阻止线程。

有些东西是底层的Park实现 - 它们以某种方式阻止了线程。实现 trait的其他事情只是将park调用委托给它们包装的一些底层对象（带有一些附加功能），允许将东西堆叠在一起。

常用的组件

我们肯定需要一个Reactor来接受来自操作系统的外部事件（比如可读的网络套接字）。它是通过mio crate阻塞epoll，kqueue或其他依赖于操作系统的原语来实现的。这不能将等待委托给任何其他东西，因此反应堆会进入堆栈的底部。

反应堆能够通过网络和类似事件通知我们的 future数据，但我们需要一个执行者来实际运行它们。我们将使用CurrentThread执行程序，因为我们正在构建单线程运行时。使用任何其他适合您需求的执行程序。当没有准备好的运行的 future时，执行者需要在下面的Park等待。它没有实现Park，因此它必须位于整个堆栈的顶部。

虽然不是绝对必要，但是能够运行延迟的 future - 超时和类似的是有用的。因此，我们将Timer置于中间位置 - 幸运的是，它可以放置在一个Park的顶部并且还可以实现Park。对于基于IO的 future反应堆而言，这与超时类似。

此外，可以添加任何自定义图层。一个例子可能是某种闲置的簿记组件 - 如果被要求等待和交错，让它下面的 Park也拿起事件，它会尝试重复做一些工作。如果没有簿记要做，它只会委托等待。

这就是反应堆，计时器和执行器的创建在代码中的样子：

```
let reactor = Reactor::new()?;
// The reactor itself will get consumed by timer,
// so we keep a handle to communicate with it.
let reactor_handle = reactor.handle();
let timer = Timer::new(reactor);
let timer_handle = timer.handle();
let mut executor = CurrentThread::new_with_park(timer);
```

这样，如果要执行 `future`，它们将首先执行。然后，一旦它用完了准备好的 `future`，它将寻找触发超时。这可能会产生一些更准备好的 `future`（接下来会执行）。如果没有超时触发，则计时器计算反应堆可以安全阻塞的时间并让它等待外部事件。

全局状态

我们已经构建了完成实际工作的组件。但我们需要一种方法来构建并向他们提交工作。我们可以通过把手这样做，但要做到这一点，我们将不得不携带它们远离人体工程学。

为了避免繁琐的几个句柄传递，内置运行时将它们存储在线程本地存储中。tokio中的几个模块有一个 `with_default` 方法，它接受相应的句柄和一个闭包。它将句柄存储在线程本地存储中并运行闭包。然后它在关闭完成后恢复TLS的原始值。

这样我们就可以在设置所有默认值的情况下运行 `future`，因此可以自由使用它们：

```
// Binds an executor to this thread
let mut enter = tokio_executor::enter()
    .expect("Multiple executors at once");
// Set the defaults before running the closure
let result = tokio_reactor::with_default(
    &reactor_handle,
    &mut enter,
    |enter| timer::with_default(
        &timer_handle,
        enter,
        |enter| {
            let mut default_executor =
                current_thread::TaskExecutor::current();
            tokio_executor::with_default(
                &mut default_executor,
                enter,
                |enter| executor.enter(enter).block_on(f)
            )
        }
    )
);

```

有一些值得注意的事情。首先，输入事物只是确保我们不会同时在同一个线程上运行多个执行程序。运行多个执行程序会使其中一个被阻止，这将以非常有用的方式起作用，因此这是防止脚步。

其次，我们希望使用与默认执行程序和默认当前线程执行程序相同的执行程序，并且还运行执行程序（不仅在不再等待的情况下将 `future` 产生到它上）。要做到这两点，我们需要两个可变的引用，这是不可能的。为了解决这个问题，我们设置了当前的线程执行器（它实际上在 `executor.block_on` 调用中设置了自己，或者任何类似的线程执行器）。我们使用 `TaskExecutor` 作为默认值，它是当前线程执行程序在使用时配置的代理。

最后，`block_on` 将执行单个 `future` 将完成（并将处理在执行程序中生成的任何其他 `future`，但如果 `f` 先完成，它将不会等待它们完成）。`future` 的结果是通过所有 `with_default` 调用向上冒泡，并且可以以任何其他方式返回或使用。如果你想等待所有其他 `future` 也完成，那么还有 `executor.run`，可以在之后执行。

运行时模型

使用Tokio编写的应用程序组织在大量小的非阻塞任务中。Tokio任务类似于[goroutine](#)或者[Erlang进程](#)，但是是非阻塞的。它们设计为轻量级，可以快速生成，并保持较低的调度开销。它们也是非阻塞的，因为无法立即完成的此类操作必须立即返回。它们返回一个表示操作正在进行的值，而不是返回操作的结果，表明操作正在进行中。

非阻塞执行

使用[Future trait](#)实现Tokio任务：

```
struct MyTask {
    my_resource: MyResource,
}

impl Future for MyTask {
    type Item = ();
    type Error = ();

    fn poll(&mut self) -> Poll<Self::Item, Self::Error> {
        match self.my_resource.poll() {
            Ok(Async::Ready(value)) => {
                self.process(value);
                Ok(Async::Ready(()))
            }
            Ok(Async::NotReady) => Ok(Async::NotReady),
            Err(err) => {
                self.process_err(err);
                Ok(Async::Ready(()))
            }
        }
    }
}
```

使用[tokio :: spawn](#)或通过调用[executor](#)对象上的[Spawn](#)方法将任务提交给 [executor](#)。[poll](#)函数驱动任务。没有调用[poll](#)就什么都不做。在任务上调用[poll](#)直到[Ready \(\(\) \)](#)返回是 [executor](#)的工作。

[MyTask](#)将从[my_resource](#)接收一个值并处理它。一旦值处理完毕，任务就完成了他的逻辑并结束。这会返回 [Ok \(Async :: Ready \(\(\) \)\)](#)。

为了完成处理，任务取决于[my_resource](#)提供的值。鉴于[my_resource](#)是一个非阻塞任务，它在调用 [my_resource.poll \(\)](#) 时，可能准备好或者还没准备好提供值。如果它准备就绪，它返回 [Ok \(Async :: Ready \(value\)\)](#)。如果没有准备好，它会返回 [Ok \(Async :: NotReady\)](#)。

当资源未准备好提供值时，这意味着该任务本身还没准备好完成，任务的[poll](#)函数也返回 [NotReady](#)。

在未来的某个时刻，资源将随时准备提供值。资源使用任务系统向 [executor](#)发信号给 [executor](#)通知它已准备好。[executor](#)安排任务，导致[MyTask :: poll](#)又叫了一遍。这一次，假设[my_resource](#)准备就绪，那

么值就是从`my_resource.poll()` 返回并且任务完成。

协作调度

协作调度用于在 `executor` 上调度任务。单个 `executor` 将通过一小组线程管理许多任务。将有比线程更多的任务。这也没有抢占。这个意味着当任务被安排执行时，它会阻止当前线程直到 `poll` 函数返回。

因此，实现 `poll` 在很短的时间内执行才是重要的。对于 I/O 绑定的应用程序，通常会发生这种情况。但是，如果任务预计必须长时间运行，则应该推迟工作到 `blocking pool` 或将计算分解为更小的块和在每个块执行之后 `yield` 回来。

任务系统

任务系统是资源通知 `executor` 准备就绪的系统。任务由消耗资源的非阻塞逻辑组成。在上面的示例中，`MyTask` 使用单个资源 `my_resource`，但没有限制任务可以使用的资源数量。

当任务正在执行并尝试使用未准备好的资源时，它在该资源上被逻辑阻塞，即任务无法进一步处理，直到资源准备就绪。Tokio 跟踪阻塞当前任务的资源以进行推进。当一个依赖资源准备就绪，`executor` 安排任务。这是通过跟踪 **当任务在资源中表现兴趣完成** 完成的。

当 `MyTask` 执行，尝试使用 `my_resource` 和 `my_resource` 返回 `NotReady` 时，`MyTask` 隐含表示对 `my_resource` 资源感兴趣。对此，任务和资源是连接的。什么时候资源准备就绪，任务再次被安排。

task :: current 和 Task :: notify

通过两个 API 完成跟踪兴趣并通知准备情况的变化：

- `task::current`
- `Task::notify`

当调用 `my_resource.poll()` 时，如果资源准备就绪，则立即返回值而不使用任务系统。如果资源 **没有准备** 好，通过调用 `task::current() -> Task` 来获取当前任务的句柄。这是通过读取 `executor` 设置的线程局部变量集获得此句柄。

一些外部事件（在网络上接收的数据，后台线程完成计算等...）将导致 `my_resource` 准备好生成它的值。那时，准备好 `my_resource` 的逻辑将调用从 `task :: current` 获得的任务句柄上的 `notify`。这个表示准备就绪会改变 `executor`，`executor` 随后安排任务执行。

如果多个任务表示对资源感兴趣，则只有 `last` 任务这样做会得到通知。资源旨在从单一任务使用。

Async :: NotReady

任何返回 `Async` 的函数都必须遵守 `contract`（契约）。当返回 `NotReady`，当前任务 **必须** 已经注册准备就绪的变更通知。以上部分讨论了资源的含义。对于任务逻辑，这意味着无法返回 `NotReady` 除非资源已返回“`NotReady`”。通过这样做，`contract` 得到了传承。当前任务已注册通知，因为已从资源收到 `NotReady`。

必须非常小心避免在没有从资源收到 `NotReady` 的情况下返回 `NotReady`。例如，以下任务中，任务实现结果永远不会完成。

```

use futures::{Future, Poll, Async};

enum BadTask {
    First(Resource1),
    Second(Resource2),
}

impl Future for BadTask {
    type Item = ();
    type Error = ();

    fn poll(&mut self) -> Poll<Self::Item, Self::Error> {
        use BadTask::*;

        let value = match *self {
            First(ref mut resource) => {
                try_ready!(resource.poll())
            }
            Second(ref mut resource) => {
                try_ready!(resource.poll());
                return Ok(Async::Ready(()));
            }
        };

        *self = Second(Resource2::new(value));
        Ok(Async::NotReady)
    }
}

```

上面实现的问题是`Ok (Async :: NotReady)` 是在将状态转换为`Second`后立即返回。在这转换中，没有资源返回`NotReady`。当任务本身返回时`NotReady`，它违反了`contract`，因为任务将来不会被通知。

通常通过添加循环来解决这种情况：

```

use futures::{Future, Poll, Async};

fn poll(&mut self) -> Poll<Self::Item, Self::Error> {
    use BadTask::*;

    loop {
        let value = match *self {
            First(ref mut resource) => {
                try_ready!(resource.poll())
            }
            Second(ref mut resource) => {
                try_ready!(resource.poll());
                return Ok(Async::Ready(()));
            }
        };

        *self = Second(Resource2::new(value));
    }
}

```

思考它的一种方法是任务的poll函数不能返回，直到由于其资源不能进一步取得进展而准备就绪或明确yields（见下文）。

另请注意，返回Async的函数只能从一个任务调用。换句话说，这些函数只能从已经提交给tokio :: spawn或其他任务spawn函数调用

Yielding

有时，任务必须返回NotReady而不是在资源上被阻塞。这通常发生在运行计算很大且任务想要的时候将控制权交还executor以允许其执行其他future。

Yielding是通过通知当前任务并返回“NotReady”完成：

```
use futures::task;
use futures::Async;

// Yield the current task. The executor will poll this task next
// iteration through its run list.
task::current().notify();
return Ok(Async::NotReady);
```

Yield可用于分解CPU昂贵的计算：

```
struct Count {
    remaining: usize,
}

impl Future for Count {
    type Item = ();
    type Error = ();

    fn poll(&mut self) -> Poll<Self::Item, Self::Error> {
        while self.remaining > 0 {
            self.remaining -= 1;

            // Yield every 10 iterations
            if self.remaining % 10 == 0 {
                task::current().notify();
                return Ok(Async::NotReady);
            }
        }

        Ok(Async::Ready(()))
    }
}
```

executor

`executor`员负责驱动完成许多任务。任务是产生于 `executor`之上，是在`executor`需要调用它的`poll`函数的时候。`executor`挂钩到任务系统以接收资源准备通知。

通过将任务系统与 `executor`实现分离，具体执行和调度逻辑可以留给 `executor`实现。`tokio`提供两个 `executor`实现，每个实现具有独特的特点：`current_thread`和`thread_pool`。

当任务首次在`executor`之上生成时，`executor`用`Spawn`将其包装。这将任务逻辑与任务状态绑定（这主要是遗留原因所需要的）。`executor`通常会将任务存储在堆，通常是将它存储在`Box`或`Arc`中。当 `executor`选择一个执行任务，它调用`Spawn :: poll_future_notify`。此函数确保将任务上下文设置为线程局部变量像`task :: current`能够读取它。

当调用`poll_future_notify`时，`executor`也是传递通知句柄和标识符。这些参数包含在由`task :: current`返回的任务句柄中，也是有关任务与`executor`连接的方式。

notify句柄是`Notify`的实现，标识符是`executor`用于查找当前任务的值。当调用`Task::notify, notify`函数使用提供的标识符调用notify句柄。该函数的实现负责执行调度逻辑。

实现 `executor`的一种策略是将每个任务存储在`Box`和使用链接列表来跟踪计划执行的任务。当调用`Notify :: notify`，然后将与之关联的任务标识符被推送到`scheduled`链表的末尾。当 `executor`运行时，它从链表的前端弹出并执行任务如上所述。

请注意，本节未介绍 `executor`的运行方式。细节这留给 `executor`实现。一个选项是 `executor`产生一个或多个线程并将这些线程专用于排出`scheduled`链表。另一个是提供一个`MyExecutor :: run`函数阻塞当前线程并排出`scheduled`链表。

资源， drivers和运行时

资源是叶子`futures`，即未以其他`futures`实施的`futures`。它们是使用上述任务系统的类型与 `executor`互动。资源类型包括TCP和UDP套接字，定时器，通道，文件句柄等。Tokio应用程序很少需要实现资源。相反，他们使用Tokio或第三方包装箱提供的资源。

通常，资源本身不能起作用而是需要drivers。例如，Tokio TCP套接字由`Reactor`支持。`Reactor`是socket资源driver。单个driver可以为大量资源实例提供动力。要使用该资源，drivers必须在某处运行这个过程。Tokio提供网络资源的drivers (`tokio-reactor`)，文件资源 (`tokio-fs`) 和定时器 (`tokio-timer`)。提供解耦driver组件允许用户选择他们想要使用的组件。每个driver可以单独使用或与其他driver结合使用。

正因为如此，为了使用Tokio并成功执行任务，一个应用程序必须启动 `executor`和必要的drivers作为应用程序的任务依赖的资源。这需要大量的样板。为了管理样板，Tokio提供了几个运行时选项。运行时是与所有必需drivers捆绑在一起的`executor`，以便为Tokio的资源提供动力。不是单独管理所有各种Tokio组件，而是在一次调用中创建并启动运行时。

Tokio提供并发运行时和单线程运行时。并发运行时基于多线程、工作窃取 `executor`。单线程运行时执行当前线程上的所有任务和drivers。用户可以选择最适合应用的运行时。

Future

如上所述，任务是使用`Future trait`实现的。这个特点不仅限于实施任务。一个 `Future`是表示一个非阻塞计算的值在未来的某个时间完成。任务是一个计算没有输出。Tokio中的许多资源都用`Future`实现。例如，超时是`Future`在达到截止日期后完成。

该 `trait`包括许多与`Future`值一起工作的有用的组合器。

通过对应用特定类型实现 Future 来构建应用或使用组合器来定义应用程序逻辑。通常两者兼而有之策略是最成功的。

非阻塞I/O

本节介绍Tokio提供的网络资源和drivers。这个组件提供Tokio的主要功能之一：非阻塞，事件驱动，由适当的操作系统原语提供的网络（epoll，kqueue，IOCP，...）。它以资源和drivers模式为模型在上一节中描述。

网络drivers使用mio构建，网络资源由后备实现Evented的类型。

本指南将重点介绍TCP类型。其他网络资源（UDP，unix插座，管道等）遵循相同的模式。

网络资源。

网络资源是由网络句柄和对为资源供电的driver的引用组成的类型，例如TcpListener和TcpStream。最初，在首次创建资源时，driver指针可能是None：

```
let listener = TcpListener::bind(&addr).unwrap();
```

在这种情况下，尚未设置对driver的引用。但是，如果使用带有Handle引用的构造函数，则driver引用将设置为给定句柄表示的driver：

```
let listener = TcpListener::from_std(std_listener, &my_reactor_handle);
```

一旦driver与资源相关联，就会将其设置为该资源的生命周期，不能改变。相关的driver负责接收网络资源的操作系统事件并通知对该资源表示兴趣的任务。

使用资源

资源类型包括以poll_为前缀和在返回类型中包含Async的非阻塞函数。这些函数与任务系统关联，应该从任务中使用，并作为[Future]实现一部分使用。例如， TcpStream提供[poll_read]和[poll_write]。

TcpListener提供[poll_accept]。

这里有一个使用[poll_accept]]接受来自侦听器的入站套接字并通过生成新任务来处理它们的任务：

```
struct Acceptor {
    listener: TcpListener,
}

impl Future for Acceptor {
    type Item = ();
    type Error = ();

    fn poll(&mut self) -> Poll<Self::Item, Self::Error> {
        loop {
            let (socket, _) = try_ready!(self.listener.poll_accept());
            // ...
        }
    }
}
```

```

        // Spawn a task to process the socket
        tokio::spawn(process(socket));
    }
}
}

```

资源类型还可以包括返回 `future` 的函数。这些是使用 `poll_*` 函数提供附加功能的帮助程序。例如，`TcpStream` 提供了一个返回 `future` 的 `[connect]` 函数。一旦 `TcpStream` 与对等方建立了连接（或未能成功），这个 `future` 就会完成。

使用组合器连接 `TcpStream`：

```

tokio::spawn({
    let connect_future = TcpStream::connect(&addr);

    connect_future
        .and_then(|socket| process(socket))
        .map_err(|_| panic!())
});

```

`future` 也可以直接用于其他 `future` 的实现：

```

struct ConnectAndProcess {
    connect: ConnectFuture,
}

impl Future for ConnectAndProcess {
    type Item = ();
    type Error = ();

    fn poll(&mut self) -> Poll<Self::Item, Self::Error> {
        let socket = try_ready!(self.connect.poll());
        tokio::spawn(process(socket));
        Ok(Async::Ready(()))
    }
}

```

使用 driver 注册资源

当使用 `TcpListener :: poll_accept`（或任何 `poll_*` 函数）时，如果资源已准备好立即返回，那么它将会这样做。在这种情况下 `poll_accept`，准备就绪意味着有一个套接字在队列中等待被接受。如果资源 **没有** 准备就绪，即没有待接受的套接字，然后资源要求 `driver` 一旦准备好就通知当前任务。

第一次 `NotReady` 由资源返回，如果资源没有明确地使用 `Handle` 参数分配一个 `driver`，则资源将使用 `driver` 实例注册自身。这是通过查看与当前执行上下文关联的网络 `driver` 来完成的。

执行上下文的默认driver使用本地线程存储，使用`with_default`设置，并使用`[Handle :: current]`访问。运行时负责确保，从闭包内传递到`with_default`过程轮询任务。调用`[Handle :: current]`访问本地线程由`with_default`设置，以便将句柄返回给当前执行上下文的driver。

Handle :: current vs Handle :: default

`Handle :: current`和`Handle :: default`都返回一个`Handle`实例。然而，它们略有不同。大多数情况下，`Handle :: default`就是期望的行为。

`Handle :: current`为当前driver **立即**读取存储在driver中的线程局部变量。这意味着`Handle :: current`必须从设置默认driver的执行上下文中调用。`Handle :: current`当句柄将被发送到不同的执行上下文使用并且用户希望使用特定的反应器（reactor）时使用（参见下面的示例）。

另一方面，`[Handle :: default]`懒惰地读取线程局部变量。这允许从执行上下文之外获取`Handle`实例。使用资源时，句柄将访问线程局部变量，如上一节中所述。

例如：

```
fn main() {
    let addr: SocketAddr = "127.0.0.1:0".parse().unwrap();
    let std_listener = ::std::net::TcpListener::bind(&addr).unwrap();
    let listener = TcpListener::from_std(std_listener,
        &Handle::default()).unwrap();

    tokio::run({
        listener.incoming().for_each(|socket| {
            tokio::spawn(process(socket));
            Ok(())
        })
        .map_err(|_| panic!("error"))
    });
}
```

在这个例子中，`incoming()`返回通过调用`poll_accept`实现的一个`future`。该`future`产生于具有网络driver配置作为执行上下文的一部分的运行之上。当在执行上下文中调用`poll_accept`时，即当读取线程本地driver与`TcpListener`实例相关联。

但是，如果直接使用`tokio-threadpool`，那么产生threadpool `executor`之上的任务就会将无法访问reactor：

```
let pool = ThreadPool::new();
let listener = TcpListener::bind(&addr).unwrap();

pool.spawn({
    listener.incoming().for_each(|socket| {
        // This will never get called due to the listener not being able
        // to
        // function.
        unreachable!();
    })
});
```

```

    })
    .map_err(|_| panic!("error"))
});

```

为了使上面的示例工作，必须为线程池的执行上下文设置反应器（reactor）。有关更多信息，请参阅[building a runtime](#)细节。或者，可以使用[\[Handle :: current\]](#)获得的Handle：

```

let pool = ThreadPool::new();

// This does not run on the pool.
tokio::run(future::lazy(move || {
    // Get the handle
    let handle = Handle::current();

    let std_listener = std::net::TcpListener::bind(&addr).unwrap();

    // This eagerly links the listener with the handle for the current
    // reactor.
    let listener = TcpListener::from_std(std_listener, &handle).unwrap();

    pool.spawn({
        listener.incoming().for_each(|socket| {
            // Do something with the socket
            Ok(())
        })
        .map_err(|_| panic!())
    });

    Ok(())
}) );

```

网络driver

为所有Tokio的网络类型提供动力的driver是[\[Reactor\]](#)Crate中的[\[tokio-reactor\]](#)类型。它是使用**mio**实现的。调用[\[Reactor :: turn\]](#)使用[\[mio :: Poll :: poll\]](#)获取已注册网络资源的操作系统事件。然后它使用[\[task system\]](#)通知每个网络资源已注册的任务。任务被调度为在其关联的[executor](#)上运行，然后任务将网络资源视为就绪并且调用[poll_ *](#)函数返回[Async :: Ready](#)。

将driver与资源链接

driver必须跟踪向其注册的每个资源。虽然实际实现更复杂，但可以将其视为对单元共享状态的共享引用，类似于：

```

struct Registration {
    // The registration needs to know its ID. This allows it to remove
    state
    // from the reactor when it is dropped.
    id: Id,
}

```

```

    // The task that owns the resource and is registered to receive
    // readiness
    // notifications from the driver.
    //
    // If `task` is `Some`, we **definitely** know that the resource
    // is not ready because we have not yet received an operating system
    event.
    // This allows avoiding syscalls that will return `NotReady`.
    //
    // If `task` is `None`, then the resource **might** be ready. We can
    try the
    // syscall, but it might still return `NotReady`.
    task: Option<task::Task>,
}

struct TcpListener {
    mio_listener: mio::TcpListener,
    registration: Option<Arc<Mutex<Registration>>>,
}

struct Reactor {
    poll: mio::Poll,
    resources: HashMap<Id, Arc<Mutex<Registration>>>,
}

```

这不是真正的实现，而是用于演示行为的简化版本。在实践中，没有Mutex，每个资源实例没有分配单元，并且reactor不使用HashMap。真正的实现在[here](#)

首次使用资源时，它会向driver注册：

```

impl TcpListener {
    fn poll_accept(&mut self) -> Poll<TcpStream, io::Error> {
        // If the registration is not set, this will associate the
        `TcpListener`
            // with the current execution context's reactor.
        let registration = self.registration.get_or_insert_with(|| {
            // Access the thread-local variable that tracks the reactor.
            Reactor::with_current(|reactor| {
                // Registers the listener, which implements
                `mio::Evented`.
                    // `register` returns the registration instance for the
                resource.
                    reactor.register(&self.mio_listener)
            })
        });
    }

    if registration.task.is_none() {
        // The task is `None`, this means the resource **might** be
        ready.
        match self.mio_listener.accept() {

```

```

        Ok(socket) => {
            let socket = mio_socket_to_tokio(socket);
            return Ok(Async::Ready(socket));
        }
        Err(ref e) if e.kind() == WouldBlock => {
            // The resource is not ready, fall through to task
            registration
        }
        Err(e) => {
            // All other errors are returned to the caller
            return Err(e);
        }
    }
}

// The task is set even if it is already `Some`, this handles the
case where
// the resource is moved to a different task than the one stored
in
// `self.task`.
registration.task = Some(task::current());
Ok(Async::NotReady)
}
}

```

请注意，每个资源只有一个task字段。其含义是资源一次只能从一个任务中使用。如果TcpListener :: poll_accept返回NotReady，注册当前任务和将监听器发送到另一个调用poll_accept的任务并视为NotReady，然后第二个任务是唯一一个在套接字准备好被接受后将接收通知的任务。资源可能会支持跟踪不同操作的不同任务。例如， TcpStream内部有两个任务字段：一个用于通知read准备好了，另一个用于通知write准备好了。这允许从不同的任务调用TcpStream :: poll_read和TcpStream :: poll_write。

[mio :: Poll]作为register上面使用的函数的一部分，将事件类型注册到驱动程序的实例中。。同样，本指南使用了简化的实现与实际tokio-reactor的实现不匹配，但足以理解tokio-reactor的行为方式。

```

impl Reactor {
    fn register<T: mio::Evented>(&mut self, evented: &T) ->
    Arc<Mutex<Registration>> {
        // Generate a unique identifier for this registration. This
        identifier
        // can be converted to and from a Mio Token.
        let id = generate_unique_identifier();

        // Register the I/O type with Mio
        self.poll.register(
            evented,
            id.into_token(),
            mio::Ready::all(),
            mio::PollOpt::edge());

        let registration = Arc::new(Mutex::new(Registration {
            id,
            task: None,

```

```

        });

        self.resources.insert(id, registration.clone());

        registration
    }
}

```

运行driver

driver需要运行才能使其相关资源正常工作。如果driver无法运行，资源永远不会准备就绪。使用Runtime时会自动处理运行driver，但了解它是如何工作的很有用。如果你对真正的实现感兴趣，那么[tokio-reactor] real-impl源码是最好的参考。

当资源注册到driver时，它们也会注册Mio，运行driver在循环中执行以下步骤：

- 1)调用[Poll :: poll]来获取操作系统事件。
- 2)发送所有事件到适当的注册过的资源。

上面的步骤是通过调用Reactor :: turn来完成的。循环部分是取决于我们。这通常在后台线程中完成或嵌入executor中作为一个Park实现。有关详细信息，请参阅runtime guide。

```

loop {
    // `None` means never timeout, blocking until we receive an operating
    // system
    // event.
    reactor.turn(None);
}

```

turn的实现执行以下操作：

```

fn turn(&mut self) {
    // Create storage for operating system events. This shouldn't be
    // created
    // each time `turn` is called, but doing so does not impact behavior.
    let mut events = mio::Events::with_capacity(1024);

    self.poll.poll(&mut events, timeout);

    for event in &events {
        let id = Id::from_token(event.token());

        if let Some(registration) = self.resources.get(&id) {
            if let Some(task) = registration.lock().unwrap().task.take() {
                task.notify();
            }
        }
    }
}

```

```
    }  
}
```

任务在其executor上进行调度会通知任务会结果。当任务再次运行，它将再次调用poll_accept函数。这次，task插槽将是None。这意味着应该尝试系统调用，并且这次poll_accept将返回一个被接受的套接字（可能允许虚假事件）。