Fundamentals of Asynchronous Programming: Async, Await, Futures, and Streams

Many operations we ask the computer to do can take a while to finish. It would be nice if we could do something else while we’re waiting for those long-running processes to complete. Modern computers offer two techniques for working on more than one operation at a time: parallelism and concurrency. Once we start writing programs that involve parallel or concurrent operations, though, we quickly encounter new challenges inherent to *asynchronous programming*, where operations may not finish in the order they were started. This chapter builds on Chapter 16’s use of threads for parallelism and concurrency by introducing an alternative approach to asynchronous programming: Rust’s futures, streams, the async and await syntax that supports them, and the tools for managing and coordinating asynchronous operations.

Let’s consider an example. Say you’re exporting a video you’ve created of a family celebration, an operation that could take anywhere from minutes to hours. The video export will use as much CPU and GPU power as it can. If you had only one CPU core and your operating system didn’t pause that export until it completed—that is, if it executed the export *synchronously*—you couldn’t do anything else on your computer while that task was running. That would be a pretty frustrating experience. Fortunately, your computer’s operating system can, and does, invisibly interrupt the export often enough to let you get other work done simultaneously.

Now say you’re downloading a video shared by someone else, which can also take a while but does not take up as much CPU time. In this case, the CPU has to wait for data to arrive from the network. While you can start reading the data once it starts to arrive, it might take some time for all of it to show up. Even once the data is all present, if the video is quite large, it could take at least a second or two to load it all. That might not sound like much, but it’s a very long time for a modern processor, which can perform billions of operations every second. Again, your operating system will invisibly interrupt your program to allow the CPU to perform other work while waiting for the network call to finish.

The video export is an example of a *CPU-bound* or *compute-bound* operation. It’s limited by the computer’s potential data processing speed within the CPU or GPU, and how much of that speed it can dedicate to the operation. The video download is an example of an *I/O-bound* operation, because it’s limited by the speed of the computer’s *input and output*; it can only go as fast as the data can be sent across the network.

In both of these examples, the operating system’s invisible interrupts provide a form of concurrency. That concurrency happens only at the level of the entire program, though: the operating system interrupts one program to let other programs get work done. In many cases, because we understand our programs at a much more granular level than the operating system does, we can spot opportunities for concurrency that the operating system can’t see.

For example, if we’re building a tool to manage file downloads, we should be able to write our program so that starting one download won’t lock up the UI, and users should be able to start multiple downloads at the same time. Many operating system APIs for interacting with the network are *blocking*, though; that is, they block the program’s progress until the data they’re processing is completely ready.

Note This is how *most* function calls work, if you think about it. However, the term *blocking* is usually reserved for function calls that interact with files, the network, or other resources on the computer, because those are the cases where an individual program would benefit from the operation being *non*-blocking.

We could avoid blocking our main thread by spawning a dedicated thread to download each file. However, the overhead of those threads would eventually become a problem. It would be preferable if the call didn’t block in the first place. It would also be better if we could write in the same direct style we use in blocking code, similar to this:

let data = fetch\_data\_from(url).await;

println!("{data}");

That is exactly what Rust’s *async* (short for *asynchronous*) abstraction gives us. In this chapter, you’ll learn all about async as we cover the following topics:

* How to use Rust’s async and await syntax
* How to use the async model to solve some of the same challenges we looked at in Chapter 16
* How multithreading and async provide complementary solutions that you can combine in many cases

Before we see how async works in practice, though, we need to take a short detour to discuss the differences between parallelism and concurrency.

Parallelism and Concurrency

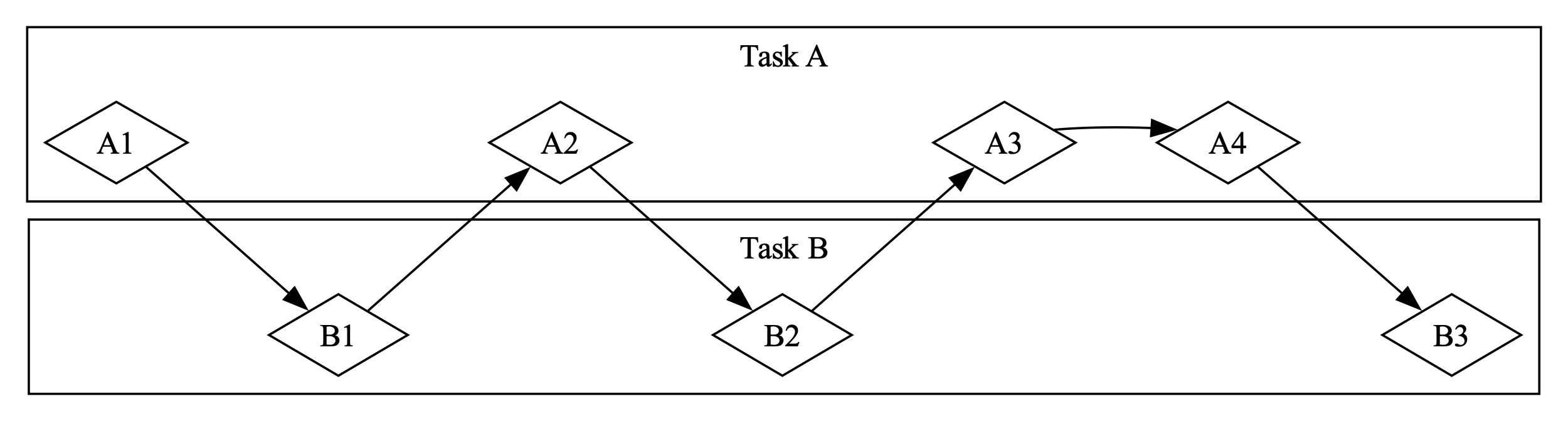
We’ve treated parallelism and concurrency as mostly interchangeable so far. Now we need to distinguish between them more precisely, because the differences will show up as we start working.

Consider the different ways a team could split up work on a software project. You could assign a single member multiple tasks, assign each member one task, or use a mix of the two approaches.

When an individual works on several different tasks before any of them is complete, this is *concurrency*. Maybe you have two different projects checked out on your computer, and when you get bored or stuck on one project, you switch to the other. You’re just one person, so you can’t make progress on both tasks at the exact same time, but you can multitask, making progress on one at a time by switching between them (see Figure 17-1).

[f17001.svg]

<A diagram with stacked boxes labeled Task A and Task B, with diamonds in them representing subtasks. Arrows point from A1 to B1, B1 to A2, A2 to B2, B2 to A3, A3 to A4, and A4 to B3. The arrows between the subtasks cross the boxes between Task A and Task B.>

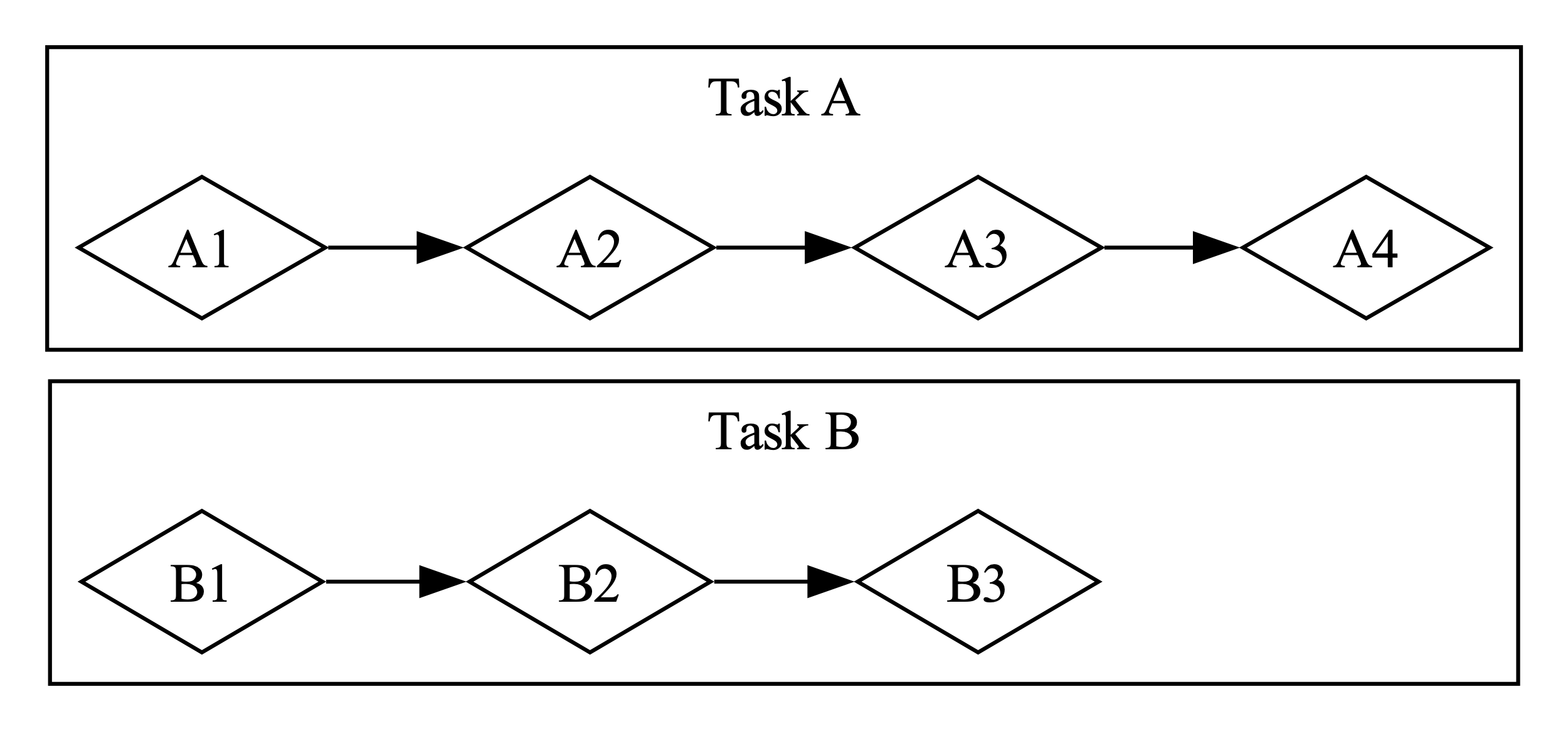


* + - * 1. A concurrent workflow, switching between Task A and Task B

When the team splits up a group of tasks by having each member take one task and work on it alone, this is *parallelism*. Each person on the team can make progress at the exact same time (see Figure 17-2).

[f17002.svg]

<A diagram with stacked boxes labeled Task A and Task B, with diamonds in them representing subtasks. Arrows point from A1 to A2, A2 to A3, A3 to A4, B1 to B2, and B2 to B3. No arrows cross between the boxes for Task A and Task B.>

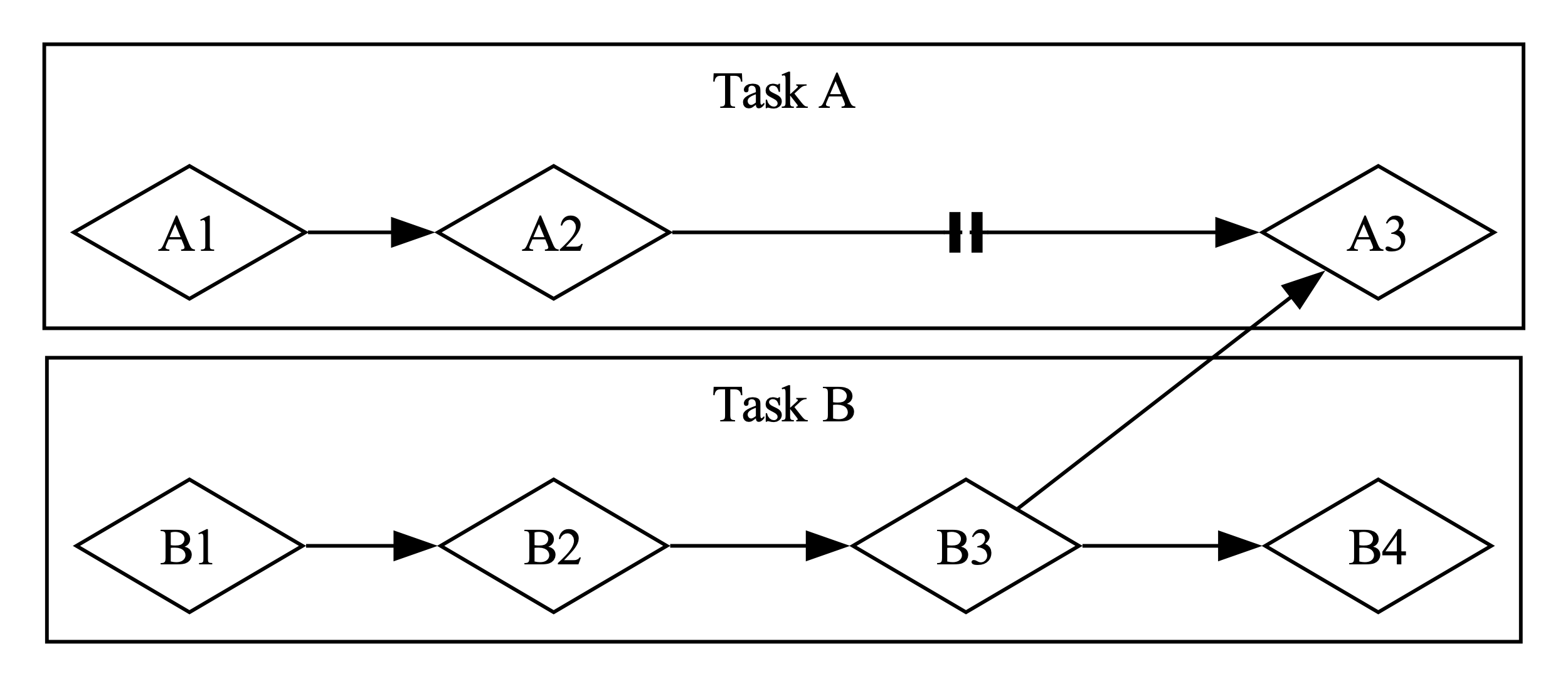


* + - * 1. A parallel workflow, where work happens on Task A and Task B independently

In both of these workflows, you might have to coordinate between different tasks. Maybe you thought the task assigned to one person was totally independent from everyone else’s work, but it actually requires another person on the team to finish their task first. Some of the work could be done in parallel, but some of it was actually *serial*: it could only happen in a series, one task after the other, as in Figure 17-3.

[f17003.svg]

<A diagram with stacked boxes labeled Task A and Task B, with diamonds in them representing subtasks. In Task A, arrows point from A1 to A2, from A2 to a pair of thick vertical lines like a “pause” symbol, and from that symbol to A3. In task B, arrows point from B1 to B2, from B2 to B3, from B3 to A3, and from B3 to B4.>



* + - * 1. A partially parallel workflow, where work happens on Task A and Task B independently until Task A3 is blocked on the results of Task B3.

Likewise, you might realize that one of your own tasks depends on another of your tasks. Now your concurrent work has also become serial.

Parallelism and concurrency can intersect with each other, too. If you learn that a colleague is stuck until you finish one of your tasks, you’ll probably focus all your efforts on that task to “unblock” your colleague. You and your coworker are no longer able to work in parallel, and you’re also no longer able to work concurrently on your own tasks.

The same basic dynamics come into play with software and hardware. On a machine with a single CPU core, the CPU can perform only one operation at a time, but it can still work concurrently. Using tools such as threads, processes, and async, the computer can pause one activity and switch to others before eventually cycling back to that first activity again. On a machine with multiple CPU cores, it can also do work in parallel. One core can be performing one task while another core performs a completely unrelated one, and those operations actually happen at the same time.

When working with async in Rust, we’re always dealing with concurrency. Depending on the hardware, the operating system, and the async runtime we are using (more on async runtimes shortly), that concurrency may also use parallelism under the hood.

Now, let’s dive into how async programming in Rust actually works.

Futures and the Async Syntax

The key elements of asynchronous programming in Rust are *futures* and Rust’s async and await keywords.

A *future* is a value that may not be ready now but will become ready at some point in the future. (This same concept shows up in many languages, sometimes under other names such as *task* or *promise*.) Rust provides a Future trait as a building block so that different async operations can be implemented with different data structures but with a common interface. In Rust, futures are types that implement the Future trait. Each future holds its own information about the progress that has been made and what “ready” means.

You can apply the async keyword to blocks and functions to specify that they can be interrupted and resumed. Within an async block or async function, you can use the await keyword to *await a future* (that is, wait for it to become ready). Any point where you await a future within an async block or function is a potential spot for that block or function to pause and resume. The process of checking with a future to see if its value is available yet is called *polling*.

Some other languages, such as C# and JavaScript, also use async and await keywords for async programming. If you’re familiar with those languages, you may notice some significant differences in how Rust handles the syntax. That’s for good reason, as we’ll see!

When writing async Rust, we use the async and await keywords most of the time. Rust compiles them into equivalent code using the Future trait, much as it compiles for loops into equivalent code using the Iterator trait. Because Rust provides the Future trait, though, you can also implement it for your own data types when you need to. Many of the functions we’ll see throughout this chapter return types with their own implementations of Future. We’ll return to the definition of the trait at the end of the chapter and dig into more of how it works, but this is enough detail to keep us moving forward.

This may all feel a bit abstract, so let’s write our first async program: a little web scraper. We’ll pass in two URLs from the command line, fetch both of them concurrently, and return the result of whichever one finishes first. This example will have a fair bit of new syntax, but don’t worry—we’ll explain everything you need to know as we go.

Our First Async Program

To keep the focus of this chapter on learning async rather than juggling parts of the ecosystem, we’ve created the trpl crate (trpl is short for “The Rust Programming Language”). It re-exports all the types, traits, and functions you’ll need, primarily from the futures and tokio crates, available on *https://crates.io*. The futures crate is an official home for Rust experimentation for async code, and it’s actually where the Future trait was originally designed. Tokio is the most widely used async runtime in Rust today, especially for web applications. There are other great runtimes out there, and they may be more suitable for your purposes. We use the tokio crate under the hood for trpl because it’s well tested and widely used.

In some cases, trpl also renames or wraps the original APIs to keep you focused on the details relevant to this chapter. If you want to understand what the crate does, we encourage you to check out its source code at *https://github.com/rust-lang/book/tree/main/packages/trpl*. You’ll be able to see what crate each re-export comes from, and we’ve left extensive comments explaining what the crate does.

Create a new binary project named hello-async and add the trpl crate as a dependency:

$ **cargo new hello-async**

$ **cd hello-async**

$ **cargo add trpl**

Now we can use the various pieces provided by trpl to write our first async program. We’ll build a little command line tool that fetches two web pages, pulls the <title> element from each, and prints out the title of whichever page finishes that whole process first.

Defining the page\_title Function

Let’s start by writing a function that takes one page URL as a parameter, makes a request to it, and returns the text of the <title> element (see Listing 17-1).

src/main.rs

use trpl::Html;

async fn page\_title(url: &str) -> Option<String> {

let response = trpl::get(url).await;

let response\_text = response.text().await;

Html::parse(&response\_text)

.select\_first("title")

.map(|title\_element| title\_element.inner\_html())

}

Defining an async function to get the title element from an HTML page

First, we define a function named page\_title and mark it with the async keyword. Then we use the trpl::get function to fetch whatever URL is passed in and add the await keyword to await the response. To get the text of the response, we call its text method and once again await it with the await keyword. Both of these steps are asynchronous. For the get function, we have to wait for the server to send back the first part of its response, which will include HTTP headers, cookies, and so on and can be delivered separately from the response body. Especially if the body is very large, it can take some time for it all to arrive. Because we have to wait for the *entirety* of the response to arrive, the text method is also async.

We have to explicitly await both of these futures, because futures in Rust are *lazy*: they don’t do anything until you ask them to with the await keyword. (In fact, Rust will show a compiler warning if you don’t use a future.) This might remind you of Chapter 13’s discussion of iterators in the section “Processing a Series of Items with Iterators” on page XX. Iterators do nothing unless you call their next method—whether directly or by using for loops or methods such as map that use next under the hood. Likewise, futures do nothing unless you explicitly ask them to. This laziness allows Rust to avoid running async code until it’s actually needed.

Note This is different from the behavior we saw in the previous chapter when using thread::spawn in “Creating a New Thread with spawn” on page XX, where the closure we passed to another thread started running immediately. It’s also different from how many other languages approach async. But it’s important for Rust to be able to provide its performance guarantees, just as it is with iterators.

Once we have response\_text, we can parse it into an instance of the Html type using Html::parse. Instead of a raw string, we now have a data type we can use to work with the HTML as a richer data structure. In particular, we can use the select\_first method to find the first instance of a given CSS selector. By passing the string "title", we’ll get the first <title> element in the document, if there is one. Because there may not be any matching element, select\_first returns an Option<ElementRef>. Finally, we use the Option::map method, which lets us work with the item in the Option if it’s present, and do nothing if it isn’t. (We could also use a match expression here, but map is more idiomatic.) In the body of the function we supply to map, we call inner\_html on the title\_element to get its content, which is a String. When all is said and done, we have an Option<String>.

Notice that Rust’s await keyword goes *after* the expression you’re awaiting, not before it. That is, it’s a *postfix* keyword. This may differ from what you’re used to if you’ve used async in other languages, but in Rust it makes chains of methods much nicer to work with. As a result, we can change the body of page\_url to chain the trpl::get and text function calls together with await between them, as shown in Listing 17-2.

src/main.rs

let response\_text = trpl::get(url).await.text().await;

Chaining with the await keyword

With that, we have successfully written our first async function! Before we add some code in main to call it, let’s talk a little more about what we’ve written and what it means.

When Rust sees a block marked with the async keyword, it compiles it into a unique, anonymous data type that implements the Future trait. When Rust sees a function marked with async, it compiles it into a non-async function whose body is an async block. An async function’s return type is the type of the anonymous data type the compiler creates for that async block.

Thus, writing async fn is equivalent to writing a function that returns a *future* of the return type. To the compiler, a function definition such as the async fn page\_title in Listing 17-1 is equivalent to a non-async function defined like this:

use std::future::Future;

use trpl::Html;

fn page\_title(url: &str) -> impl Future<Output = Option<String>> {

async move {

let text = trpl::get(url).await.text().await;

Html::parse(&text)

.select\_first("title")

.map(|title| title.inner\_html())

}

}

Let’s walk through each part of the transformed version:

* It uses the impl Trait syntax we discussed back in Chapter 10 in the “Traits as Parameters” section on page XX.
* The returned trait is a Future with an associated type of Output. Notice that the Output type is Option<String>, which is the same as the original return type from the async fn version of page\_title.
* All of the code called in the body of the original function is wrapped in an async move block. Remember that blocks are expressions. This whole block is the expression returned from the function.
* This async block produces a value with the type Option<String>, as just described. That value matches the Output type in the return type. This is just like other blocks you have seen.
* The new function body is an async move block because of how it uses the url parameter. (We’ll talk much more about async versus async move later in the chapter.)

Now we can call page\_title in main.

Determining a Single Page’s Title

To start, we’ll just get the title for a single page. In Listing 17-3, we follow the same pattern we used in Chapter 12 to get command line arguments in the “Accepting Command Line Arguments” section on page XX. Then we pass the first URL page\_title and await the result. Because the value produced by the future is an Option<String>, we use a match expression to print different messages to account for whether the page had a <title>.

src/main.rs

async fn main() {

let args: Vec<String> = std::env::args().collect();

let url = &args[1];

match page\_title(url).await {

Some(title) => println!("The title for {url} was {title}"),

None => println!("{url} had no title"),

}

}

Calling the page\_title function from main with a user-supplied argument

Unfortunately, this code doesn’t compile. The only place we can use the await keyword is in async functions or blocks, and Rust won’t let us mark the special main function as async.

error[E0752]: `main` function is not allowed to be `async`

--> src/main.rs:6:1

|

6 | async fn main() {

| ^^^^^^^^^^^^^^^ `main` function is not allowed to be `async`

The reason main can’t be marked async is that async code needs a *runtime*: a Rust crate that manages the details of executing asynchronous code. A program’s main function can *initialize* a runtime, but it’s not a runtime *itself*. (We’ll see more about why this is the case in a bit.) Every Rust program that executes async code has at least one place where it sets up a runtime and executes the futures.

Most languages that support async bundle a runtime, but Rust does not. Instead, there are many different async runtimes available, each of which makes different tradeoffs suitable to the use case it targets. For example, a high-throughput web server with many CPU cores and a large amount of RAM has very different needs than a microcontroller with a single core, a small amount of RAM, and no heap allocation ability. The crates that provide those runtimes also often supply async versions of common functionality such as file or network I/O.

Here, and throughout the rest of this chapter, we’ll use the run function from the trpl crate, which takes a future as an argument and runs it to completion. Behind the scenes, calling run sets up a runtime that’s used to run the future passed in. Once the future completes, run returns whatever value the future produced.

We could pass the future returned by page\_title directly to run and, once it completed, we could match on the resulting Option<String> as we tried to do in Listing 17-3. However, for most of the examples in the chapter (and most async code in the real world), we’ll be doing more than just one async function call, so instead we’ll pass an async block and explicitly await the result of the page\_title call, as in Listing 17-4.

src/main.rs

fn main() {

let args: Vec<String> = std::env::args().collect();

trpl::run(async {

let url = &args[1];

match page\_title(url).await {

Some(title) => println!("The title for {url} was {title}"),

None => println!("{url} had no title"),

}

})

}

Awaiting an async block with trpl::run

When we run this code, we get the behavior we expected initially:

$ **cargo run "http://www.rust-lang.org"**

The title for http://www.rust-lang.org was

Rust Programming Language

Phew—we finally have some working async code! But before we add the code to race the two sites against each other, let’s briefly turn our attention back to how futures work.

Each *await point*—that is, every place where the code uses the await keyword—represents a place where control is handed back to the runtime. To make that work, Rust needs to keep track of the state involved in the async block so that the runtime can kick off some other work and then come back when it’s ready to try advancing the first one again. This is an invisible state machine, as if you’d written an enum like this to save the current state at each await point:

enum PageTitleFuture<'a> {

Initial { url: &'a str },

GetAwaitPoint { url: &'a str },

TextAwaitPoint { response: trpl::Response },

}

Writing the code to transition between each state by hand would be tedious and error-prone, however, especially when you need to add more functionality and more states to the code later. Fortunately, the Rust compiler creates and manages the state machine data structures for async code automatically. The normal borrowing and ownership rules around data structures all still apply, and happily, the compiler also handles checking those for us and provides useful error messages. We’ll work through a few of those later in the chapter.

Ultimately, something has to execute this state machine, and that something is a runtime. (This is why you may come across references to *executors* when looking into runtimes: an executor is the part of a runtime responsible for executing the async code.)

Now you can see why the compiler stopped us from making main itself an async function back in Listing 17-3. If main were an async function, something else would need to manage the state machine for whatever future main returned, but main is the starting point for the program! Instead, we called the trpl::run function in main to set up a runtime and run the future returned by the async block until it is done.

Note Some runtimes provide macros so you *can* write an async main function. Those macros rewrite async fn main() { ... } to be a normal fn main, which does the same thing we did by hand in Listing 17-4: call a function that runs a future to completion the way trpl::run does.

Now let’s put these pieces together and see how we can write concurrent code.

Racing Our Two URLs Against Each Other

In Listing 17-5, we call page\_title with two different URLs passed in from the command line and race them.

src/main.rs

use trpl::{Either, Html};

fn main() {

let args: Vec<String> = std::env::args().collect();

trpl::run(async {

let title\_fut\_1 = page\_title(&args[1]);

let title\_fut\_2 = page\_title(&args[2]);

let (url, maybe\_title) =

match trpl::race(title\_fut\_1, title\_fut\_2).await {

Either::Left(left) => left,

Either::Right(right) => right,

};

println!("{url} returned first");

match maybe\_title {

Some(title) => println!("Its page title is: '{title}'"),

None => println!("Its title could not be parsed."),

}

})

}

async fn page\_title(url: &str) -> (&str, Option<String>) {

let text = trpl::get(url).await.text().await;

let title = Html::parse(&text)

.select\_first("title")

.map(|title| title.inner\_html());

(url, title)

}

Calling page\_title for two URLs to see which returns first

We begin by calling page\_title for each of the user-supplied URLs. We save the resulting futures as title\_fut\_1 and title\_fut\_2. Remember, these don’t do anything yet, because futures are lazy and we haven’t yet awaited them. Then we pass the futures to trpl::race, which returns a value to indicate which of the futures passed to it finishes first.

Note Under the hood, race is built on a more general function, select, which you will encounter more often in real-world Rust code. A select function can do a lot of things that the trpl::race function can’t, but it also has some additional complexity that we can skip over for now.

Either future can legitimately “win,” so it doesn’t make sense to return a Result. Instead, race returns a type we haven’t seen before, trpl::Either. The Either type is somewhat similar to a Result in that it has two cases. Unlike Result, though, there is no notion of success or failure baked into Either. Instead, it uses Left and Right to indicate “one or the other”:

enum Either<A, B> {

Left(A),

Right(B),

}

The race function returns Left with that future’s output if the first argument wins, and Right with the second future argument’s output if *that* one wins. This matches the order the arguments appear in when calling the function: the first argument is to the left of the second argument.

We also update page\_title to return the same URL passed in. That way, if the page that returns first does not have a <title> we can resolve, we can still print a meaningful message. With that information available, we wrap up by updating our println! output to indicate both which URL finished first and what, if any, the <title> is for the web page at that URL.

You have built a small working web scraper now! Pick a couple URLs and run the command line tool. You may discover that some sites are consistently faster than others, while in other cases the faster site varies from run to run. More importantly, you’ve learned the basics of working with futures, so now we can dig deeper into what we can do with async.

Applying Concurrency with Async

In this section, we’ll apply async to some of the same concurrency challenges we tackled with threads in Chapter 16. Because we already talked about a lot of the key ideas there, in this section we’ll focus on what’s different between threads and futures.

In many cases, the APIs for working with concurrency using async are very similar to those for using threads. In other cases, they end up being quite different. Even when the APIs *look* similar between threads and async, they often have different behavior—and they nearly always have different performance characteristics.

Creating a New Task with spawn\_task

The first operation we tackled in “Creating a New Thread with spawn” on page XX was counting up on two separate threads. Let’s do the same using async. The trpl crate supplies a spawn\_task function that looks very similar to the thread::spawn API, and a sleep function that is an async version of the thread::sleep API. We can use these together to implement the counting example, as shown in Listing 17-6.

src/main.rs

use std::time::Duration;

fn main() {

trpl::run(async {

trpl::spawn\_task(async {

for i in 1..10 {

println!("hi number {i} from the first task!");

trpl::sleep(Duration::from\_millis(500)).await;

}

});

for i in 1..5 {

println!("hi number {i} from the second task!");

trpl::sleep(Duration::from\_millis(500)).await;

}

});

}

Creating a new task to print one thing while the main task prints something else

As our starting point, we set up our main function with trpl::run so that our top-level function can be async.

Note From this point forward in the chapter, every example will include this exact same wrapping code with trpl::run in main, so we’ll often skip it just as we do with main. Don’t forget to include it in your code!

Then we write two loops within that block, each containing a trpl::sleep call, which waits for half a second (500 milliseconds) before sending the next message. We put one loop in the body of a trpl::spawn\_task and the other in a top-level for loop. We also add an await after the sleep calls.

This code behaves similarly to the thread-based implementation—including the fact that you may see the messages appear in a different order in your own terminal when you run it:

hi number 1 from the second task!

hi number 1 from the first task!

hi number 2 from the first task!

hi number 2 from the second task!

hi number 3 from the first task!

hi number 3 from the second task!

hi number 4 from the first task!

hi number 4 from the second task!

hi number 5 from the first task!

This version stops as soon as the for loop in the body of the main async block finishes, because the task spawned by spawn\_task is shut down when the main function ends. If you want it to run all the way to the task’s completion, you will need to use a join handle to wait for the first task to complete. With threads, we used the join method to “block” until the thread was done running. In Listing 17-7, we can use await to do the same thing, because the task handle itself is a future. Its Output type is a Result, so we also unwrap it after awaiting it.

src/main.rs

let handle = trpl::spawn\_task(async {

for i in 1..10 {

println!("hi number {i} from the first task!");

trpl::sleep(Duration::from\_millis(500)).await;

}

});

for i in 1..5 {

println!("hi number {i} from the second task!");

trpl::sleep(Duration::from\_millis(500)).await;

}

handle.await.unwrap();

Using await with a join handle to run a task to completion

This updated version runs until *both* loops finish:

hi number 1 from the second task!

hi number 1 from the first task!

hi number 2 from the first task!

hi number 2 from the second task!

hi number 3 from the first task!

hi number 3 from the second task!

hi number 4 from the first task!

hi number 4 from the second task!

hi number 5 from the first task!

hi number 6 from the first task!

hi number 7 from the first task!

hi number 8 from the first task!

hi number 9 from the first task!

So far, it looks like async and threads give us the same basic outcomes, just with different syntax: using await instead of calling join on the join handle, and awaiting the sleep calls.

The bigger difference is that we didn’t need to spawn another operating system thread to do this. In fact, we don’t even need to spawn a task here. Because async blocks compile to anonymous futures, we can put each loop in an async block and have the runtime run them both to completion using the trpl::join function.

In the section “Waiting for All Threads to Finish Using join Handles” on page XX, we showed how to use the join method on the JoinHandle type returned when you call std::thread::spawn. The trpl::join function is similar, but for futures. When you give it two futures, it produces a single new future whose output is a tuple containing the output of each future you passed in once they *both* complete. Thus, in Listing 17-8, we use trpl::join to wait for both fut1 and fut2 to finish. We do *not* await fut1 and fut2 but instead the new future produced by trpl::join. We ignore the output, because it’s just a tuple containing two unit values.

src/main.rs

let fut1 = async {

for i in 1..10 {

println!("hi number {i} from the first task!");

trpl::sleep(Duration::from\_millis(500)).await;

}

};

let fut2 = async {

for i in 1..5 {

println!("hi number {i} from the second task!");

trpl::sleep(Duration::from\_millis(500)).await;

}

};

trpl::join(fut1, fut2).await;

Using trpl::join to await two anonymous futures

When we run this, we see both futures run to completion:

hi number 1 from the first task!

hi number 1 from the second task!

hi number 2 from the first task!

hi number 2 from the second task!

hi number 3 from the first task!

hi number 3 from the second task!

hi number 4 from the first task!

hi number 4 from the second task!

hi number 5 from the first task!

hi number 6 from the first task!

hi number 7 from the first task!

hi number 8 from the first task!

hi number 9 from the first task!

Now, you’ll see the exact same order every time, which is very different from what we saw with threads. That is because the trpl::join function is *fair*, meaning it checks each future equally often, alternating between them, and never lets one race ahead if the other is ready. With threads, the operating system decides which thread to check and how long to let it run. With async Rust, the runtime decides which task to check. (In practice, the details get complicated because an async runtime might use operating system threads under the hood as part of how it manages concurrency, so guaranteeing fairness can be more work for a runtime—but it’s still possible!) Runtimes don’t have to guarantee fairness for any given operation, and they often offer different APIs to let you choose whether or not you want fairness.

Try some of these variations on awaiting the futures and see what they do:

* Remove the async block from around either or both of the loops.
* Await each async block immediately after defining it.
* Wrap only the first loop in an async block, and await the resulting future after the body of second loop.

For an extra challenge, see if you can figure out what the output will be in each case *before* running the code!

Counting Up on Two Tasks Using Message Passing

Sharing data between futures will also be familiar: we’ll use message passing again, but this time with async versions of the types and functions. We’ll take a slightly different path than we did in “Using Message Passing to Transfer Data Between Threads” on page XX to illustrate some of the key differences between thread-based and futures-based concurrency. In Listing 17-9, we’ll begin with just a single async block—*not* spawning a separate task as we spawned a separate thread.

src/main.rs

let (tx, mut rx) = trpl::channel();

let val = String::from("hi");

tx.send(val).unwrap();

let received = rx.recv().await.unwrap();

println!("Got: {received}");

Creating an async channel and assigning the two halves to tx and rx

Here, we use trpl::channel, an async version of the multiple-producer, single-consumer channel API we used with threads back in Chapter 16. The async version of the API is only a little different from the thread-based version: it uses a mutable rather than an immutable receiver rx, and its recv method produces a future we need to await rather than producing the value directly. Now we can send messages from the sender to the receiver. Notice that we don’t have to spawn a separate thread or even a task; we merely need to await the rx.recv call.

The synchronous Receiver::recv method in std::mpsc::channel blocks until it receives a message. The trpl::Receiver::recv method does not, because it is async. Instead of blocking, it hands control back to the runtime until either a message is received or the send side of the channel closes. By contrast, we don’t await the send call, because it doesn’t block. It doesn’t need to, because the channel we’re sending it into is unbounded.

Note Because all of this async code runs in an async block in a trpl::run call, everything within it can avoid blocking. However, the code *outside* it will block on the run function returning. That’s the whole point of the trpl::run function: it lets you *choose* where to block on some set of async code, and thus where to transition between sync and async code. In most async runtimes, run is actually named block\_on for exactly this reason.

Notice two things about this example. First, the message will arrive right away. Second, although we use a future here, there’s no concurrency yet. Everything in the listing happens in sequence, just as it would if there were no futures involved.

Let’s address the first part by sending a series of messages and sleeping in between them, as shown in Listing 17-10.

src/main.rs

let (tx, mut rx) = trpl::channel();

let vals = vec![

String::from("hi"),

String::from("from"),

String::from("the"),

String::from("future"),

];

for val in vals {

tx.send(val).unwrap();

trpl::sleep(Duration::from\_millis(500)).await;

}

while let Some(value) = rx.recv().await {

println!("received '{value}'");

}

Sending and receiving multiple messages over the async channel and sleeping with an await between each message

In addition to sending the messages, we need to receive them. In this case, because we know how many messages are coming in, we could do that manually by calling rx.recv().await four times. In the real world, though, we’ll generally be waiting on some *unknown* number of messages, so we need to keep waiting until we determine that there are no more messages.

In Listing 16-10, we used a for loop to process all the items received from a synchronous channel. Rust doesn’t yet have a way to write a for loop over an *asynchronous* series of items, however, so we need to use a loop we haven’t seen before: the while let conditional loop. This is the loop version of the if let construct we saw back in the section “Concise Control Flow with if let and let else” on page XX. The loop will continue executing as long as the pattern it specifies continues to match the value.

The rx.recv call produces a future, which we await. The runtime will pause the future until it is ready. Once a message arrives, the future will resolve to Some(message) as many times as a message arrives. When the channel closes, regardless of whether *any* messages have arrived, the future will instead resolve to None to indicate that there are no more values and thus we should stop polling—that is, stop awaiting.

The while let loop pulls all of this together. If the result of calling rx.recv().await is Some(message), we get access to the message and we can use it in the loop body, just as we could with if let. If the result is None, the loop ends. Every time the loop completes, it hits the await point again, so the runtime pauses it again until another message arrives.

The code now successfully sends and receives all of the messages. Unfortunately, there are still a couple of problems. For one thing, the messages do not arrive at half-second intervals. They arrive all at once, 2 seconds (2,000 milliseconds) after we start the program. For another, this program also never exits! Instead, it waits forever for new messages. You will need to shut it down using ctrl-c.

Let’s start by examining why the messages come in all at once after the full delay, rather than coming in with delays between each one. Within a given async block, the order in which await keywords appear in the code is also the order in which they’re executed when the program runs.

There’s only one async block in Listing 17-10, so everything in it runs linearly. There’s still no concurrency. All the tx.send calls happen, interspersed with all of the trpl::sleep calls and their associated await points. Only then does the while let loop get to go through any of the await points on the recv calls.

To get the behavior we want, where the sleep delay happens between each message, we need to put the tx and rx operations in their own async blocks, as shown in Listing 17-11. Then the runtime can execute each of them separately using trpl::join, just as in the counting example. Once again, we await the result of calling trpl::join, not the individual futures. If we awaited the individual futures in sequence, we would just end up back in a sequential flow—exactly what we’re trying *not* to do.

src/main.rs

let tx\_fut = async {

let vals = vec![

String::from("hi"),

String::from("from"),

String::from("the"),

String::from("future"),

];

for val in vals {

tx.send(val).unwrap();

trpl::sleep(Duration::from\_millis(500)).await;

}

};

let rx\_fut = async {

while let Some(value) = rx.recv().await {

println!("received '{value}'");

}

};

trpl::join(tx\_fut, rx\_fut).await;

Separating send and recv into their own async blocks and awaiting the futures for those blocks

With the updated code in Listing 17-11, the messages get printed at 500-millisecond intervals, rather than all in a rush after 2 seconds.

The program still never exits, though, because of the way while let loop interacts with trpl::join:

* The future returned from trpl::join completes only once *both* futures passed to it have completed.
* The tx future completes once it finishes sleeping after sending the last message in vals.
* The rx future won’t complete until the while let loop ends.
* The while let loop won’t end until awaiting rx.recv produces None.
* Awaiting rx.recv will return None only once the other end of the channel is closed.
* The channel will close only if we call rx.close or when the sender side, tx, is dropped.
* We don’t call rx.close anywhere, and tx won’t be dropped until the outermost async block passed to trpl::run ends.
* The block can’t end because it is blocked on trpl::join completing, which takes us back to the top of this list.

We could manually close rx by calling rx.close somewhere, but that doesn’t make much sense. Stopping after handling some arbitrary number of messages would make the program shut down, but we could miss messages. We need some other way to make sure that tx gets dropped *before* the end of the function.

Right now, the async block where we send the messages only borrows tx because sending a message doesn’t require ownership, but if we could move tx into that async block, it would be dropped once that block ends. In the Chapter 13 section “Capturing References or Moving Ownership” on page XX, you learned how to use the move keyword with closures, and, as discussed in the Chapter 16 section “Using move Closures with Threads” on page XX, we often need to move data into closures when working with threads. The same basic dynamics apply to async blocks, so the move keyword works with async blocks just as it does with closures.

In Listing 17-12, we change the block used to send messages from async to async move. When we run *this* version of the code, it shuts down gracefully after the last message is sent and received.

src/main.rs

let (tx, mut rx) = trpl::channel();

let tx\_fut = async move {

let vals = vec![

String::from("hi"),

String::from("from"),

String::from("the"),

String::from("future"),

];

for val in vals {

tx.send(val).unwrap();

trpl::sleep(Duration::from\_millis(500)).await;

}

};

let rx\_fut = async {

while let Some(value) = rx.recv().await {

eprintln!("received '{value}'");

}

};

trpl::join(tx\_fut, rx\_fut).await;

A revision of the code from Listing 17-11 that correctly shuts down when complete

This async channel is also a multiple-producer channel, so we can call clone on tx if we want to send messages from multiple futures, as shown in Listing 17-13.

src/main.rs

let (tx, mut rx) = trpl::channel();

let tx1 = tx.clone();

let tx1\_fut = async move {

let vals = vec![

String::from("hi"),

String::from("from"),

String::from("the"),

String::from("future"),

];

for val in vals {

tx1.send(val).unwrap();

trpl::sleep(Duration::from\_millis(500)).await;

}

};

let rx\_fut = async {

while let Some(value) = rx.recv().await {

println!("received '{value}'");

}

};

let tx\_fut = async move {

let vals = vec![

String::from("more"),

String::from("messages"),

String::from("for"),

String::from("you"),

];

for val in vals {

tx.send(val).unwrap();

trpl::sleep(Duration::from\_millis(1500)).await;

}

};

trpl::join3(tx1\_fut, tx\_fut, rx\_fut).await;

Using multiple producers with async blocks

First, we clone tx, creating tx1 outside the first async block. We move tx1 into that block just as we did before with tx. Then, later, we move the original tx into a *new* async block, where we send more messages on a slightly slower delay. We happen to put this new async block after the async block for receiving messages, but it could go before it just as well. The key is the order in which the futures are awaited, not in which they’re created.

Both of the async blocks for sending messages need to be async move blocks so that both tx and tx1 get dropped when those blocks finish. Otherwise, we’ll end up back in the same infinite loop we started out in. Finally, we switch from trpl::join to trpl::join3 to handle the additional future.

Now we see all the messages from both sending futures, and because the sending futures use slightly different delays after sending, the messages are also received at those different intervals:

received 'hi'

received 'more'

received 'from'

received 'the'

received 'messages'

received 'future'

received 'for'

received 'you'

This is a good start, but it limits us to just a handful of futures: two with join, or three with join3. Let’s see how we might work with more futures.

Working with Any Number of Futures

When we switched from using two futures to three, we also had to switch from using join to using join3. It would be annoying to have to call a different function every time we changed the number of futures we wanted to join. Happily, we have a macro form of join to which we can pass an arbitrary number of arguments. It also handles awaiting the futures itself. Thus, we could rewrite the code from Listing 17-13 to use join! instead of join3, as in Listing 17-14.

src/main.rs

trpl::join!(tx1\_fut, tx\_fut, rx\_fut);

Using join! to wait for multiple futures

This is definitely an improvement over swapping between join and join3 and join4 and so on! However, even this macro form only works when we know the number of futures ahead of time. In real-world Rust, though, pushing futures into a collection and then waiting on some or all of them to complete is a common pattern.

To check all the futures in some collection, we’ll need to iterate over and join on *all* of them. The trpl::join\_all function accepts any type that implements the Iterator trait, which you learned about back in “The Iterator Trait and the next Method” on page XX, so it seems like just the ticket. Let’s try putting our futures in a vector and replacing join! with join\_all as shown in Listing 17-15.

let futures = vec![tx1\_fut, rx\_fut, tx\_fut];

trpl::join\_all(futures).await;

Storing anonymous futures in a vector and calling join\_all

Unfortunately, this code doesn’t compile. Instead, we get this error:

error[E0308]: mismatched types

--> src/main.rs:43:37

|

8 | let tx1\_fut = async move {

| \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_-

9 | | let vals = vec![

10 | | String::from("hi"),

11 | | String::from("from"),

... |

19 | | }

20 | | };

| |\_\_\_\_\_\_\_\_\_- the expected `async` block

21 |

22 | let rx\_fut = async {

| \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_-

23 | | while let Some(value) = rx.recv().await {

24 | | println!("received '{value}'");

25 | | }

26 | | };

| |\_\_\_\_\_\_\_\_\_- the found `async` block

...

43 | let futures = vec![tx1\_fut, rx\_fut, tx\_fut];

| ^^^^^^ expected `async` block, found a

different `async` block

|

= note: expected `async` block `{async block@src/main.rs:8:23: 20:10}`

found `async` block `{async block@src/main.rs:22:22: 26:10}`

= note: no two async blocks, even if identical, have the same type

= help: consider pinning your async block and casting it to a trait object

This might be surprising. After all, none of the async blocks returns anything, so each one produces a Future<Output = ()>. Remember that Future is a trait, though, and that the compiler creates a unique enum for each async block, even when they have identical output types. Just as you can’t put two different handwritten structs in a Vec, you can’t mix compiler-generated enums.

To make this work, we need to use *trait objects*, just as we did in “Returning Errors from the run function” on page XX. (We’ll cover trait objects in detail in Chapter 18.) Using trait objects lets us treat each of the anonymous futures produced by these types as the same type, because all of them implement the Future trait.

Note In the Chapter 8 section “Using an Enum to Store Multiple Types” on page XX, we discussed another way to include multiple types in a Vec: using an enum to represent each type that can appear in the vector. We can’t do that here, though. For one thing, we have no way to name the different types, because they are anonymous. For another, the reason we reached for a vector and join\_all in the first place was to be able to work with a dynamic collection of futures where we only care that they have the same output type.

We start by wrapping each future in the vec! in a Box::new, as shown in Listing 17-16.

src/main.rs

let futures =

vec![Box::new(tx1\_fut), Box::new(rx\_fut), Box::new(tx\_fut)];

trpl::join\_all(futures).await;

Using Box::new to align the types of the futures in a Vec

Unfortunately, this code still doesn’t compile. In fact, we get the same basic error we got before for both the second and third Box::new calls, as well as new errors referring to the Unpin trait. We’ll come back to the Unpin errors in a moment. First, let’s fix the type errors on the Box::new calls by explicitly annotating the type of the futures variable (see Listing 17-17).

src/main.rs

let futures: Vec<Box<dyn Future<Output = ()>>> =

vec![Box::new(tx1\_fut), Box::new(rx\_fut), Box::new(tx\_fut)];

Fixing the rest of the type mismatch errors by using an explicit type declaration

This type declaration is a little involved, so let’s walk through it:

1. The innermost type is the future itself. We note explicitly that the output of the future is the unit type () by writing Future<Output = ()>.
2. Then we annotate the trait with dyn to mark it as dynamic.
3. The entire trait reference is wrapped in a Box.
4. Finally, we state explicitly that futures is a Vec containing these items.

That already made a big difference. Now when we run the compiler, we get only the errors mentioning Unpin. Although there are three of them, their contents are very similar.

error[E0277]: `dyn Future<Output = ()>` cannot be unpinned

--> src/main.rs:49:24

|

49 | trpl::join\_all(futures).await;

| -------------- ^^^^^^^ the trait `Unpin` is not implemented for

`dyn Future<Output = ()>`

| |

| required by a bound introduced by this call

|

= note: consider using the `pin!` macro

consider using `Box::pin` if you need to access the pinned value outside of the

current scope

= note: required for `Box<dyn Future<Output = ()>>` to implement `Future`

note: required by a bound in `join\_all`

--> file:///home/.cargo/registry/src/index.crates.io-1949cf8c6b5b557f/futures-util-0.3.30/

src/future/join\_all.rs:105:14

|

102 | pub fn join\_all<I>(iter: I) -> JoinAll<I::Item>

| -------- required by a bound in this function

...

105 | I::Item: Future,

| ^^^^^^ required by this bound in `join\_all`

error[E0277]: `dyn Future<Output = ()>` cannot be unpinned

--> src/main.rs:49:9

|

49 | trpl::join\_all(futures).await;

| ^^^^^^^^^^^^^^^^^^^^^^^ the trait `Unpin` is not implemented for

`dyn Future<Output = ()>`

|

= note: consider using the `pin!` macro

consider using `Box::pin` if you need to access the pinned value outside of the

current scope

= note: required for `Box<dyn Future<Output = ()>>` to implement `Future`

note: required by a bound in `futures\_util::future::join\_all::JoinAll`

--> file:///home/.cargo/registry/src/index.crates.io-1949cf8c6b5b557f/futures-util-0.3.30/

src/future/join\_all.rs:29:8

|

27 | pub struct JoinAll<F>

| ------- required by a bound in this struct

28 | where

29 | F: Future,

| ^^^^^^ required by this bound in `JoinAll`

error[E0277]: `dyn Future<Output = ()>` cannot be unpinned

--> src/main.rs:49:33

|

49 | trpl::join\_all(futures).await;

| ^^^^^ the trait `Unpin` is not implemented for

`dyn Future<Output = ()>`

|

= note: consider using the `pin!` macro

consider using `Box::pin` if you need to access the pinned value outside of the

current scope

= note: required for `Box<dyn Future<Output = ()>>` to implement `Future`

note: required by a bound in `futures\_util::future::join\_all::JoinAll`

--> file:///home/.cargo/registry/src/index.crates.io-1949cf8c6b5b557f/futures-util-0.3.30/

src/future/join\_all.rs:29:8

|

27 | pub struct JoinAll<F>

| ------- required by a bound in this struct

28 | where

29 | F: Future,

| ^^^^^^ required by this bound in `JoinAll`

For more information about this error, try `rustc --explain E0277`.

error: could not compile `async\_await` (bin "async\_await") due to 3 previous errors

That is a *lot* to digest, so let’s pull it apart. The first part of the message tell us that the first async block (src/main.rs:8:23: 20:10) does not implement the Unpin trait and suggests using pin! or Box::pin to resolve it. Later in the chapter, we’ll dig into a few more details about Pin and Unpin. For the moment, though, we can just follow the compiler’s advice to get unstuck. In Listing 17-18, we start by updating the type annotation for futures, with a Pin wrapping each Box. Second, we use Box::pin to pin the futures themselves.

src/main.rs

let futures: Vec<Pin<Box<dyn Future<Output = ()>>>> =

vec![Box::pin(tx1\_fut), Box::pin(rx\_fut), Box::pin(tx\_fut)];

Using Pin and Box::pin to make the Vec type check

If we compile and run this, we finally get the output we hoped for:

received 'hi'

received 'more'

received 'from'

received 'messages'

received 'the'

received 'for'

received 'future'

received 'you'

Phew!

There’s a bit more to explore here. For one thing, using Pin<Box<T>> adds a small amount of overhead from putting these futures on the heap with Box—and we’re only doing that to get the types to line up. We don’t actually *need* the heap allocation, after all: these futures are local to this particular function. As noted before, Pin is itself a wrapper type, so we can get the benefit of having a single type in the Vec—the original reason we reached for Box—without doing a heap allocation. We can use Pin directly with each future, using the std::pin::pin macro.

However, we must still be explicit about the type of the pinned reference; otherwise, Rust will still not know to interpret these as dynamic trait objects, which is what we need them to be in the Vec. We therefore pin! each future when we define it, and define futures as a Vec containing pinned mutable references to the dynamic future type, as in Listing 17-19.

src/main.rs

let tx1\_fut = pin!(async move {

// *--snip--*

});

let rx\_fut = pin!(async {

// *--snip--*

});

let tx\_fut = pin!(async move {

// *--snip--*

});

let futures: Vec<Pin<&mut dyn Future<Output = ()>>> =

vec![tx1\_fut, rx\_fut, tx\_fut];

Using Pin directly with the pin! macro to avoid unnecessary heap allocations

We got this far by ignoring the fact that we might have different Output types. For example, in Listing 17-20, the anonymous future for a implements Future<Output = u32>, the anonymous future for b implements Future<Output = &str>, and the anonymous future for c implements Future<Output = bool>.

src/main.rs

let a = async { 1u32 };

let b = async { "Hello!" };

let c = async { true };

let (a\_result, b\_result, c\_result) = trpl::join!(a, b, c);

println!("{a\_result}, {b\_result}, {c\_result}");

Three futures with distinct types

We can use trpl::join! to await them, because it allows us to pass in multiple future types and produces a tuple of those types. We *cannot* use trpl::join\_all, because it requires all of the futures passed in to have the same type. Remember, that error is what got us started on this adventure with Pin!

This is a fundamental tradeoff: we can either deal with a dynamic number of futures with join\_all, as long as they all have the same type, or we can deal with a set number of futures with the join functions or the join! macro, even if they have different types. This is the same scenario we’d face when working with any other types in Rust. Futures are not special, even though we have some nice syntax for working with them, and that’s a good thing.

Racing Futures

When we “join” futures with the join family of functions and macros, we require *all* of them to finish before we move on. Sometimes, though, we only need *some* future from a set to finish before we move on—kind of similar to racing one future against another.

In Listing 17-21, we once again use trpl::race to run two futures, slow and fast, against each other.

src/main.rs

let slow = async {

println!("'slow' started.");

trpl::sleep(Duration::from\_millis(100)).await;

println!("'slow' finished.");

};

let fast = async {

println!("'fast' started.");

trpl::sleep(Duration::from\_millis(50)).await;

println!("'fast' finished.");

};

trpl::race(slow, fast).await;

Using race to get the result of whichever future finishes first

Each future prints a message when it starts running, pauses for some amount of time by calling and awaiting sleep, and then prints another message when it finishes. Then we pass both slow and fast to trpl::race and wait for one of them to finish. (The outcome here isn’t too surprising: fast wins.) Unlike when we used race back in “Our First Async Program” on page XX, we just ignore the Either instance it returns here, because all of the interesting behavior happens in the body of the async blocks.

Notice that if you flip the order of the arguments to race, the order of the “started” messages changes, even though the fast future always completes first. That’s because the implementation of this particular race function is not fair. It always runs the futures passed in as arguments in the order in which they’re passed. Other implementations *are* fair and will randomly choose which future to poll first. Regardless of whether the implementation of race we’re using is fair, though, *one* of the futures will run up to the first await in its body before another task can start.

Recall from “Our First Async Program” that at each await point, Rust gives a runtime a chance to pause the task and switch to another one if the future being awaited isn’t ready. The inverse is also true: Rust *only* pauses async blocks and hands control back to a runtime at an await point. Everything between await points is synchronous.

That means if you do a bunch of work in an async block without an await point, that future will block any other futures from making progress. You may sometimes hear this referred to as one future *starving* other futures. In some cases, that may not be a big deal. However, if you are doing some kind of expensive setup or long-running work, or if you have a future that will keep doing some particular task indefinitely, you’ll need to think about when and where to hand control back to the runtime.

By the same token, if you have long-running blocking operations, async can be a useful tool for providing ways for different parts of the program to relate to each other.

But *how* would you hand control back to the runtime in those cases?

Yielding Control to the Runtime

Let’s simulate a long-running operation. Listing 17-22 introduces a slow function.

src/main.rs

fn slow(name: &str, ms: u64) {

thread::sleep(Duration::from\_millis(ms));

println!("'{name}' ran for {ms}ms");

}

Using thread::sleep to simulate slow operations

This code uses std::thread::sleep instead of trpl::sleep so that calling slow will block the current thread for some number of milliseconds. We can use slow to stand in for real-world operations that are both long-running and blocking.

In Listing 17-23, we use slow to emulate doing this kind of CPU-bound work in a pair of futures.

src/main.rs

let a = async {

println!("'a' started.");

slow("a", 30);

slow("a", 10);

slow("a", 20);

trpl::sleep(Duration::from\_millis(50)).await;

println!("'a' finished.");

};

let b = async {

println!("'b' started.");

slow("b", 75);

slow("b", 10);

slow("b", 15);

slow("b", 350);

trpl::sleep(Duration::from\_millis(50)).await;

println!("'b' finished.");

};

trpl::race(a, b).await;

Using thread::sleep to simulate slow operations

Each future hands control back to the runtime only *after* carrying out a bunch of slow operations. If you run this code, you will see this output:

'a' started.

'a' ran for 30ms

'a' ran for 10ms

'a' ran for 20ms

'b' started.

'b' ran for 75ms

'b' ran for 10ms

'b' ran for 15ms

'b' ran for 350ms

'a' finished.

As with our earlier example, race still finishes as soon as a is done. There’s no interleaving between the two futures, though. The a future does all of its work until the trpl::sleep call is awaited, then the b future does all of its work until its own trpl::sleep call is awaited, and finally the a future completes. To allow both futures to make progress between their slow tasks, we need await points so we can hand control back to the runtime. That means we need something we can await!

We can already see this kind of handoff happening in Listing 17-23: if we removed the trpl::sleep at the end of the a future, it would complete without the b future running *at all*. Let’s try using the sleep function as a starting point for letting operations switch off making progress, as shown in Listing 17-24.

src/main.rs

let one\_ms = Duration::from\_millis(1);

let a = async {

println!("'a' started.");

slow("a", 30);

trpl::sleep(one\_ms).await;

slow("a", 10);

trpl::sleep(one\_ms).await;

slow("a", 20);

trpl::sleep(one\_ms).await;

println!("'a' finished.");

};

let b = async {

println!("'b' started.");

slow("b", 75);

trpl::sleep(one\_ms).await;

slow("b", 10);

trpl::sleep(one\_ms).await;

slow("b", 15);

trpl::sleep(one\_ms).await;

slow("b", 35);

trpl::sleep(one\_ms).await;

println!("'b' finished.");

};

Using sleep to let operations switch off making progress

We’ve added trpl::sleep calls with await points between each call to slow. Now the two futures’ work is interleaved:

'a' started.

'a' ran for 30ms

'b' started.

'b' ran for 75ms

'a' ran for 10ms

'b' ran for 10ms

'a' ran for 20ms

'b' ran for 15ms

'a' finished.

The a future still runs for a bit before handing off control to b, because it calls slow before ever calling trpl::sleep, but after that the futures swap back and forth each time one of them hits an await point. In this case, we have done that after every call to slow, but we could break up the work in whatever way makes the most sense to us.

We don’t really want to *sleep* here, though: we want to make progress as fast as we can. We just need to hand back control to the runtime. We can do that directly, using the yield\_now function. In Listing 17-25, we replace all those sleep calls with yield\_now.

src/main.rs

let a = async {

println!("'a' started.");

slow("a", 30);

trpl::yield\_now().await;

slow("a", 10);

trpl::yield\_now().await;

slow("a", 20);

trpl::yield\_now().await;

println!("'a' finished.");

};

let b = async {

println!("'b' started.");

slow("b", 75);

trpl::yield\_now().await;

slow("b", 10);

trpl::yield\_now().await;

slow("b", 15);

trpl::yield\_now().await;

slow("b", 35);

trpl::yield\_now().await;

println!("'b' finished.");

};

Using yield\_now to let operations switch off making progress

This code is both clearer about the actual intent and can be significantly faster than using sleep, because timers such as the one used by sleep often have limits on how granular they can be. The version of sleep we are using, for example, will always sleep for at least a millisecond, even if we pass it a Duration of one nanosecond. Again, modern computers are *fast*: they can do a lot in one millisecond!

You can see this for yourself by setting up a little benchmark, such as the one in Listing 17-26. (This isn’t an especially rigorous way to do performance testing, but it suffices to show the difference here.)

src/main.rs

let one\_ns = Duration::from\_nanos(1);

let start = Instant::now();

async {

for \_ in 1..1000 {

trpl::sleep(one\_ns).await;

}

}

.await;

let time = Instant::now() - start;

println!(

"'sleep' version finished after {} seconds.",

time.as\_secs\_f32()

);

let start = Instant::now();

async {

for \_ in 1..1000 {

trpl::yield\_now().await;

}

}

.await;

let time = Instant::now() - start;

println!(

"'yield' version finished after {} seconds.",

time.as\_secs\_f32()

);

Comparing the performance of sleep and yield\_now

Here, we skip all the status printing, pass a one-nanosecond Duration to trpl::sleep, and let each future run by itself, with no switching between the futures. Then we run for 1,000 iterations and see how long the future using trpl::sleep takes compared to the future using trpl::yield\_now.

The version with yield\_now is *way* faster!

This means that async can be useful even for compute-bound tasks, depending on what else your program is doing, because it provides a useful tool for structuring the relationships between different parts of the program. This is a form of *cooperative multitasking*, where each future has the power to determine when it hands over control via await points. Each future therefore also has the responsibility to avoid blocking for too long. In some Rust-based embedded operating systems, this is the *only* kind of multitasking!

In real-world code, you won’t usually be alternating function calls with await points on every single line, of course. While yielding control in this way is relatively inexpensive, it’s not free. In many cases, trying to break up a compute-bound task might make it significantly slower, so sometimes it’s better for *overall* performance to let an operation block briefly. Always measure to see what your code’s actual performance bottlenecks are. The underlying dynamic is important to keep in mind, though, if you *are* seeing a lot of work happening in serial that you expected to happen concurrently!

Building Our Own Async Abstractions

We can also compose futures together to create new patterns. For example, we can build a timeout function with async building blocks we already have. When we’re done, the result will be another building block we could use to create still more async abstractions.

Listing 17-27 shows how we would expect this timeout to work with a slow future.

src/main.rs

let slow = async {

trpl::sleep(Duration::from\_millis(100)).await;

"I finished!"

};

match timeout(slow, Duration::from\_millis(10)).await {

Ok(message) => println!("Succeeded with '{message}'"),

Err(duration) => {

println!("Failed after {} seconds", duration.as\_secs())

}

}

Using our imagined timeout to run a slow operation with a time limit

Let’s implement this! To begin, let’s think about the API for timeout:

* It needs to be an async function itself so we can await it.
* Its first parameter should be a future to run. We can make it generic to allow it to work with any future.
* Its second parameter will be the maximum time to wait. If we use a Duration, that will make it easy to pass along to trpl::sleep.
* It should return a Result. If the future completes successfully, the Result will be Ok with the value produced by the future. If the timeout elapses first, the Result will be Err with the duration that the timeout waited for.

Listing 17-28 shows this declaration.

src/main.rs

async fn timeout<F: Future>(

future\_to\_try: F,

max\_time: Duration,

) -> Result<F::Output, Duration> {

// Here is where our implementation will go!

}

Defining the signature of timeout

That satisfies our goals for the types. Now let’s think about the *behavior* we need: we want to race the future passed in against the duration. We can use trpl::sleep to make a timer future from the duration, and use trpl::race to run that timer with the future the caller passes in.

We also know that race is not fair, polling arguments in the order in which they are passed. Thus, we pass future\_to\_try to race first so it gets a chance to complete even if max\_time is a very short duration. If future\_to\_try finishes first, race will return Left with the output from future\_to\_try. If timer finishes first, race will return Right with the timer’s output of ().

In Listing 17-29, we match on the result of awaiting trpl::race.

src/main.rs

use trpl::Either;

// *--snip--*

fn main() {

trpl::run(async {

let slow = async {

trpl::sleep(Duration::from\_secs(5)).await;

"Finally finished"

};

match timeout(slow, Duration::from\_secs(2)).await {

Ok(message) => println!("Succeeded with '{message}'"),

Err(duration) => {

println!("Failed after {} seconds", duration.as\_secs())

}

}

});

}

async fn timeout<F: Future>(

future\_to\_try: F,

max\_time: Duration,

) -> Result<F::Output, Duration> {

match trpl::race(future\_to\_try, trpl::sleep(max\_time)).await {

Either::Left(output) => Ok(output),

Either::Right(\_) => Err(max\_time),

}

Defining timeout with race and sleep

If the future\_to\_try succeeds and we get a Left(output), we return Ok(output). If the sleep timer elapses instead and we get a Right(()), we ignore the () with \_ and return Err(max\_time) instead.

With that, we have a working timeout built out of two other async helpers. If we run our code, it will print the failure mode after the timeout:

Failed after 2 seconds

Because futures compose with other futures, you can build really powerful tools using smaller async building blocks. For example, you can use this same approach to combine timeouts with retries, and in turn use those with operations such as network calls (one of the examples from the beginning of the chapter).

In practice, you’ll usually work directly with async and await, and secondarily with functions and macros such as join, join\_all, race, and so on. You’ll only need to reach for pin now and again to use futures with those APIs.

We’ve now seen a number of ways to work with multiple futures at the same time. Up next, we’ll look at how we can work with multiple futures in a sequence over time with *streams*. Here are a couple more things you might want to consider first, though:

* We used a Vec with join\_all to wait for all of the futures in some group to finish. How could you use a Vec to process a group of futures in sequence instead? What are the tradeoffs of doing that?
* Take a look at the futures::stream::FuturesUnordered type from the futures crate. How would using it be different from using a Vec? (Don’t worry about the fact that it’s from the stream part of the crate; it works just fine with any collection of futures.)

Streams: Futures in Sequence

So far in this chapter, we’ve mostly stuck to individual futures. The one big exception was the async channel we used. Recall how we used the receiver for our async channel earlier in this chapter in the “Message Passing” section on page XX. The async recv method produces a sequence of items over time. This is an instance of a much more general pattern known as a *stream*.

We saw a sequence of items back in Chapter 13, when we looked at the Iterator trait in “The Iterator Trait and the next Method” section on page XX, but there are two differences between iterators and the async channel receiver. The first difference is time: iterators are synchronous, while the channel receiver is asynchronous. The second difference is the API. When working directly with Iterator, we call its synchronous next method. With the trpl::Receiver stream in particular, we called an asynchronous recv method instead. Otherwise, these APIs feel very similar, and that similarity isn’t a coincidence. A stream is like an asynchronous form of iteration. Whereas the trpl::Receiver specifically waits to receive messages, though, the general-purpose stream API is much broader: it provides the next item the way Iterator does, but asynchronously.

The similarity between iterators and streams in Rust means we can actually create a stream from any iterator. As with an iterator, we can work with a stream by calling its next method and then awaiting the output, as in Listing 17-30.

src/main.rs

let values = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10];

let iter = values.iter().map(|n| n \* 2);

let mut stream = trpl::stream\_from\_iter(iter);

while let Some(value) = stream.next().await {

println!("The value was: {value}");

}

Creating a stream from an iterator and printing its values

We start with an array of numbers, which we convert to an iterator and then call map on to double all the values. Then we convert the iterator into a stream using the trpl::stream\_from\_iter function. Next, we loop over the items in the stream as they arrive with the while let loop.

Unfortunately, when we try to run the code, it doesn’t compile but instead reports that there’s no next method available:

error[E0599]: no method named `next` found for struct `Iter` in the current scope

--> src/main.rs:8:40

|

8 | while let Some(value) = stream.next().await {

| ^^^^

|

= note: the full type name has been written to '~/projects/hello-async/target/debug/deps/async\_await-bbd5bb8f6851cb5f.long-type-18426562901668632191.txt'

= note: consider using `--verbose` to print the full type name to the console

= help: items from traits can only be used if the trait is in scope

help: the following traits which provide `next` are implemented but not in scope; perhaps you

want to import one of them

|

1 + use futures\_util::stream::stream::StreamExt;

|

1 + use std::iter::Iterator;

|

1 + use std::str::pattern::Searcher;

|

1 + use trpl::StreamExt;

|

help: there is a method `try\_next` with a similar name

|

8 | while let Some(value) = stream.try\_next().await {

| ~~~~~~~~

For more information about this error, try `rustc --explain E0599`.

As this output explains, the reason for the compiler error is that we need the right trait in scope to be able to use the next method. Given our discussion so far, you might reasonably expect that trait to be Stream, but it’s actually StreamExt. Short for *extension*, Ext is a common pattern in the Rust community for extending one trait with another.

We’ll explain the Stream and StreamExt traits in a bit more detail at the end of the chapter, but for now all you need to know is that the Stream trait defines a low-level interface that effectively combines the Iterator and Future traits. StreamExt supplies a higher-level set of APIs on top of Stream, including the next method as well as other utility methods similar to those provided by the Iterator trait. Stream and StreamExt are not yet part of Rust’s standard library, but most ecosystem crates use the same definition.

The fix to the compiler error is to add a use statement for trpl::StreamExt, as in Listing 17-31.

src/main.rs

use trpl::StreamExt;

fn main() {

trpl::run(async {

let values = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10];

let iter = values.iter().map(|n| n \* 2);

let mut stream = trpl::stream\_from\_iter(iter);

while let Some(value) = stream.next().await {

println!("The value was: {value}");

}

});

}

Successfully using an iterator as the basis for a stream

With all those pieces put together, this code works the way we want! What’s more, now that we have StreamExt in scope, we can use all of its utility methods, just as with iterators. For example, in Listing 17-32, we use the filter method to filter out everything but multiples of three and five.

src/main.rs

use trpl::StreamExt;

fn main() {

trpl::run(async {

let values = 1..101;

let iter = values.map(|n| n \* 2);

let stream = trpl::stream\_from\_iter(iter);

let mut filtered =

stream.filter(|value| value % 3 == 0 || value % 5 == 0);

while let Some(value) = filtered.next().await {

println!("The value was: {value}");

}

});

}

Filtering a stream with the StreamExt::filter method

Of course, this isn’t very interesting, since we could do the same with normal iterators and without any async at all. Let’s look at what we can do that *is* unique to streams.

Composing Streams

Many concepts are naturally represented as streams: items becoming available in a queue, chunks of data being pulled incrementally from the filesystem when the full data set is too large for the computer’s memory, or data arriving over the network over time. Because streams are futures, we can use them with any other kind of future and combine them in interesting ways. For example, we can batch up events to avoid triggering too many network calls, set timeouts on sequences of long-running operations, or throttle user interface events to avoid doing needless work.

Let’s start by building a little stream of messages as a stand-in for a stream of data we might see from a WebSocket or another real-time communication protocol, as shown in Listing 17-33.

src/main.rs

use trpl::{ReceiverStream, Stream, StreamExt};

fn main() {

trpl::run(async {

let mut messages = get\_messages();

while let Some(message) = messages.next().await {

println!("{message}");

}

});

}

fn get\_messages() -> impl Stream<Item = String> {

let (tx, rx) = trpl::channel();

let messages = ["a", "b", "c", "d", "e", "f", "g", "h", "i", "j"];

for message in messages {

tx.send(format!("Message: '{message}'")).unwrap();

}

ReceiverStream::new(rx)

}

Using the rx receiver as a ReceiverStream

First, we create a function called get\_messages that returns impl Stream<Item = String>. For its implementation, we create an async channel, loop over the first 10 letters of the English alphabet, and send them across the channel.

We also use a new type: ReceiverStream, which converts the rx receiver from the trpl::channel into a Stream with a next method. Back in main, we use a while let loop to print all the messages from the stream.

When we run this code, we get exactly the results we would expect:

Message: 'a'

Message: 'b'

Message: 'c'

Message: 'd'

Message: 'e'

Message: 'f'

Message: 'g'

Message: 'h'

Message: 'i'

Message: 'j'

Again, we could do this with the regular Receiver API or even the regular Iterator API, though, so let’s add a feature that requires streams: adding a timeout that applies to every item in the stream, and a delay on the items we emit, as shown in Listing 17-34.

src/main.rs

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

use trpl::{ReceiverStream, Stream, StreamExt};

fn main() {

trpl::run(async {

let mut messages =

pin!(get\_messages().timeout(Duration::from\_millis(200)));

while let Some(result) = messages.next().await {

match result {

Ok(message) => println!("{message}"),

Err(reason) => eprintln!("Problem: {reason:?}"),

}

}

})

}

Using the StreamExt::timeout method to set a time limit on the items in a stream

We start by adding a timeout to the stream with the timeout method, which comes from the StreamExt trait. Then we update the body of the while let loop, because the stream now returns a Result. The Ok variant indicates a message arrived in time; the Err variant indicates that the timeout elapsed before any message arrived. We match on that result and either print the message when we receive it successfully or print a notice about the timeout. Finally, notice that we pin the messages after applying the timeout to them, because the timeout helper produces a stream that needs to be pinned to be polled.

However, because there are no delays between messages, this timeout does not change the behavior of the program. Let’s add a variable delay to the messages we send, as shown in Listing 17-35.

src/main.rs

fn get\_messages() -> impl Stream<Item = String> {

let (tx, rx) = trpl::channel();

trpl::spawn\_task(async move {

let messages = ["a", "b", "c", "d", "e", "f", "g", "h", "i", "j"];

for (index, message) in messages.into\_iter().enumerate() {

let time\_to\_sleep = if index % 2 == 0 { 100 } else { 300 };

trpl::sleep(Duration::from\_millis(time\_to\_sleep)).await;

tx.send(format!("Message: '{message}'")).unwrap();

}

});

ReceiverStream::new(rx)

}

Sending messages through tx with an async delay without making get\_messages an async function

In get\_messages, we use the enumerate iterator method with the messages array so that we can get the index of each item we’re sending along with the item itself. Then we apply a 100-millisecond delay to even-index items and a 300-millisecond delay to odd-index items to simulate the different delays we might see from a stream of messages in the real world. Because our timeout is for 200 milliseconds, this should affect half of the messages.

To sleep between messages in the get\_messages function without blocking, we need to use async. However, we can’t make get\_messages itself into an async function, because then we’d return a Future<Output = Stream<Item = String>> instead of a Stream<Item = String>>. The caller would have to await get\_messages itself to get access to the stream. But remember: everything in a given future happens linearly; concurrency happens *between* futures. Awaiting get\_messages would require it to send all the messages, including the sleep delay between each message, before returning the receiver stream. As a result, the timeout would be useless. There would be no delays in the stream itself; they would all happen before the stream was even available.

Instead, we leave get\_messages as a regular function that returns a stream, and we spawn a task to handle the async sleep calls.

Note Calling spawn\_task in this way works because we already set up our runtime; had we not, it would cause a panic. Other implementations choose different tradeoffs: they might spawn a new runtime and avoid the panic but end up with a bit of extra overhead, or they may simply not provide a standalone way to spawn tasks without reference to a runtime. Make sure you know what tradeoff your runtime has chosen and write your code accordingly!

Now our code has a much more interesting result. Between every other pair of messages, we see a Problem: Elapsed(()) error:

Message: 'a'

Problem: Elapsed(())

Message: 'b'

Message: 'c'

Problem: Elapsed(())

Message: 'd'

Message: 'e'

Problem: Elapsed(())

Message: 'f'

Message: 'g'

Problem: Elapsed(())

Message: 'h'

Message: 'i'

Problem: Elapsed(())

Message: 'j'

The timeout doesn’t prevent the messages from arriving in the end. We still get all of the original messages, because our channel is *unbounded*: it can hold as many messages as we can fit in memory. If the message doesn’t arrive before the timeout, our stream handler will account for that, but when it polls the stream again, the message may now have arrived.

You can get different behavior if needed by using other kinds of channels or other kinds of streams more generally. Let’s see one of those in practice by combining a stream of time intervals with this stream of messages.

Merging Streams

First, let’s create another stream, which will emit an item every millisecond if we let it run directly. For simplicity, we can use the sleep function to send a message on a delay and combine it with the same approach we used in get\_messages of creating a stream from a channel. The difference is that this time, we’re going to send back the count of intervals that have elapsed, so the return type will be impl Stream<Item = u32>, and we can call the function get\_intervals (see Listing 17-36).

src/main.rs

fn get\_intervals() -> impl Stream<Item = u32> {

let (tx, rx) = trpl::channel();

trpl::spawn\_task(async move {

let mut count = 0;

loop {

trpl::sleep(Duration::from\_millis(1)).await;

count += 1;

tx.send(count).unwrap();

}

});

ReceiverStream::new(rx)

}

Creating a stream with a counter that will be emitted once every millisecond

We start by defining a count in the task. (We could define it outside the task, too, but it’s clearer to limit the scope of any given variable.) Then we create an infinite loop. Each iteration of the loop asynchronously sleeps for one millisecond, increments the count, and then sends it over the channel. Because this is all wrapped in the task created by spawn\_task, all of it—including the infinite loop—will get cleaned up along with the runtime.

This kind of infinite loop, which ends only when the whole runtime gets torn down, is fairly common in async Rust: many programs need to keep running indefinitely. With async, this doesn’t block anything else, as long as there is at least one await point in each iteration through the loop.

Now, back in our main function’s async block, we can attempt to merge the messages and intervals streams, as shown in Listing 17-37.

src/main.rs

let messages = get\_messages().timeout(Duration::from\_millis(200));

let intervals = get\_intervals();

let merged = messages.merge(intervals);

Attempting to merge the messages and intervals streams

We start by calling get\_intervals. Then we merge the messages and intervals streams with the merge method, which combines multiple streams into one stream that produces items from any of the source streams as soon as the items are available, without imposing any particular ordering. Finally, we loop over that combined stream instead of over messages.

At this point, neither messages nor intervals needs to be pinned or mutable, because both will be combined into the single merged stream. However, this call to merge doesn’t compile! (Neither does the next call in the while let loop, but we’ll come back to that.) This is because the two streams have different types. The messages stream has the type Timeout<impl Stream<Item = String>>, where Timeout is the type that implements Stream for a timeout call. The intervals stream has the type impl Stream<Item = u32>. To merge these two streams, we need to transform one of them to match the other. We’ll rework the intervals stream, because messages is already in the basic format we want and has to handle timeout errors (see Listing 17-38).

src/main.rs

let messages = get\_messages().timeout(Duration::from\_millis(200));

let intervals = get\_intervals()

.map(|count| format!("Interval: {count}"))

.timeout(Duration::from\_secs(10));

let merged = messages.merge(intervals);

let mut stream = pin!(merged);

Aligning the type of the intervals stream with the type of the messages stream

First, we can use the map helper method to transform the intervals into a string. Second, we need to match the Timeout from messages. Because we don’t actually *want* a timeout for intervals, though, we can just create a timeout that’s longer than the other durations we’re using. Here, we create a 10-second timeout with Duration::from\_secs(10). Finally, we need to make stream mutable so that the while let loop’s next calls can iterate through the stream, and pin it so that it’s safe to do so. That gets us *almost* to where we need to be. Everything type checks. If you run this, though, there will be two problems. First, it will never stop! You’ll need to stop it with <span class="keystroke">ctrl-c</span>. Second, the messages from the English alphabet will be buried in the midst of all the interval counter messages:

*--snip--*

Interval: 38

Interval: 39

Interval: 40

Message: 'a'

Interval: 41

Interval: 42

Interval: 43

*--snip--*

Listing 17-39 shows one way to solve these last two problems.

src/main.rs

let messages = get\_messages().timeout(Duration::from\_millis(200));

let intervals = get\_intervals()

.map(|count| format!("Interval: {count}"))

.throttle(Duration::from\_millis(100))

.timeout(Duration::from\_secs(10));

let merged = messages.merge(intervals).take(20);

let mut stream = pin!(merged);

Using throttle and take to manage the merged streams

First, we use the throttle method on the intervals stream so that it doesn’t overwhelm the messages stream. *Throttling* is a way of limiting the rate at which a function will be called—or, in this case, how often the stream will be polled. Once every 100 milliseconds should do, because that’s roughly how often our messages arrive.

To limit the number of items we will accept from a stream, we apply the take method to the merged stream, because we want to limit the final output, not just one stream or the other.

Now when we run the program, it stops after pulling 20 items from the stream, and the intervals don’t overwhelm the messages. We also don’t get Interval: 100 or Interval: 200 or so on, but instead get Interval: 1, Interval: 2, and so on—even though we have a source stream that *can* produce an event every millisecond. That’s because the throttle call produces a new stream that wraps the original stream so that the original stream gets polled only at the throttle rate, not its own “native” rate. We don’t have a bunch of unhandled interval messages we’re choosing to ignore. Instead, we never produce those interval messages in the first place! This is the inherent “laziness” of Rust’s futures at work again, allowing us to choose our performance characteristics.

Interval: 1

Message: 'a'

Interval: 2

Interval: 3

Problem: Elapsed(())

Interval: 4

Message: 'b'

Interval: 5

Message: 'c'

Interval: 6

Interval: 7

Problem: Elapsed(())

Interval: 8

Message: 'd'

Interval: 9

Message: 'e'

Interval: 10

Interval: 11

Problem: Elapsed(())

Interval: 12

There’s one last thing we need to handle: errors! With both of these channel-based streams, the send calls could fail when the other side of the channel closes—and that’s just a matter of how the runtime executes the futures that make up the stream. Up until now, we’ve ignored this possibility by calling unwrap, but in a well-behaved app, we should explicitly handle the error, at minimum by ending the loop so we don’t try to send any more messages. Listing 17-40 shows a simple error strategy: print the issue and then break from the loops.

fn get\_messages() -> impl Stream<Item = String> {

let (tx, rx) = trpl::channel();

trpl::spawn\_task(async move {

let messages = ["a", "b", "c", "d", "e", "f", "g", "h", "i", "j"];

for (index, message) in messages.into\_iter().enumerate() {

let time\_to\_sleep = if index % 2 == 0 { 100 } else { 300 };

trpl::sleep(Duration::from\_millis(time\_to\_sleep)).await;

if let Err(send\_error) = tx.send(format!("Message: '{message}'")) {

eprintln!("Cannot send message '{message}': {send\_error}");

break;

}

}

});

ReceiverStream::new(rx)

}

fn get\_intervals() -> impl Stream<Item = u32> {

let (tx, rx) = trpl::channel();

trpl::spawn\_task(async move {

let mut count = 0;

loop {

trpl::sleep(Duration::from\_millis(1)).await;

count += 1;

if let Err(send\_error) = tx.send(count) {

eprintln!("Could not send interval {count}: {send\_error}");

break;

};

}

});

ReceiverStream::new(rx)

}

Handling errors and shutting down the loops

As usual, the correct way to handle a message send error will vary; just make sure you have a strategy.

Now that we’ve seen a bunch of async in practice, let’s take a step back and dig into a few of the details of how Future, Stream, and the other key traits Rust uses to make async work.

A Closer Look at the Traits for Async

Throughout the chapter, we’ve used the Future, Pin, Unpin, Stream, and StreamExt traits in various ways. So far, though, we’ve avoided getting too far into the details of how they work or how they fit together, which is fine most of the time for your day-to-day Rust work. Sometimes, though, you’ll encounter situations where you’ll need to understand a few more of these details. In this section, we’ll dig in just enough to help in those scenarios, still leaving the *really* deep dive for other documentation.

The Future Trait

Let’s start by taking a closer look at how the Future trait works. Here’s how Rust defines it:

use std::pin::Pin;

use std::task::{Context, Poll};

pub trait Future {

type Output;

fn poll(self: Pin<&mut Self>, cx: &mut Context<'\_>) -> Poll<Self::Output>;

}

That trait definition includes a bunch of new types and also some syntax we haven’t seen before, so let’s walk through the definition piece by piece.

First, Future’s associated type Output says what the future resolves to. This is analogous to the Item associated type for the Iterator trait. Second, Future also has the poll method, which takes a special Pin reference for its self parameter and a mutable reference to a Context type, and returns a Poll<Self::Output>. We’ll talk more about Pin and Context in a moment. For now, let’s focus on what the method returns, the Poll type:

enum Poll<T> {

Ready(T),

Pending,

}

This Poll type is similar to an Option. It has one variant that has a value, Ready(T), and one that does not, Pending. Poll means something quite different from Option, though! The Pending variant indicates that the future still has work to do, so the caller will need to check again later. The Ready variant indicates that the Future has finished its work and the T value is available.

Note With most futures, the caller should not call poll again after the future has returned Ready. Many futures will panic if polled again after becoming ready. Futures that are safe to poll again will say so explicitly in their documentation. This is similar to how Iterator::next behaves.

When you see code that uses await, Rust compiles it under the hood to code that calls poll. If you look back at Listing 17-4, where we printed out the page title for a single URL once it resolved, Rust compiles it into something kind of (although not exactly) like this:

match page\_title(url).poll() {

Ready(page\_title) => match page\_title {

Some(title) => println!("The title for {url} was {title}"),

None => println!("{url} had no title"),

}

Pending => {

// But what goes here?

}

}

What should we do when the future is still Pending? We need some way to try again, and again, and again, until the future is finally ready. In other words, we need a loop:

let mut page\_title\_fut = page\_title(url);

loop {

match page\_title\_fut.poll() {

Ready(value) => match page\_title {

Some(title) => println!("The title for {url} was {title}"),

None => println!("{url} had no title"),

}

Pending => {

// continue

}

}

}

If Rust compiled it to exactly that code, though, every await would be blocking—exactly the opposite of what we were going for! Instead, Rust needs makes sure that the loop can hand off control to something that can pause work on this future to work on other futures and then check this one again later. As we’ve seen, that something is an async runtime, and this scheduling and coordination work is one of its main jobs.

Earlier in the chapter, we described waiting on rx.recv. The recv call returns a future, and awaiting the future polls it. We noted that a runtime will pause the future until it’s ready with either Some(message) or None when the channel closes. With our deeper understanding of the Future trait, and specifically Future::poll, we can see how that works. The runtime knows the future isn’t ready when it returns Poll::Pending. Conversely, the runtime knows the future *is* ready and advances it when poll returns Poll::Ready(Some(message)) or Poll::Ready(None).

The exact details of how a runtime does that are beyond the scope of this book, but the key is to see the basic mechanics of futures: a runtime *polls* each future it is responsible for, putting the future back to sleep when it is not yet ready.

The Pin and Unpin Traits

When we introduced the idea of pinning in Listing 17-17, we ran into a very gnarly error message. Here is the relevant part of it again:

error[E0277]: `{async block@src/main.rs:8:23: 20:10}` cannot be unpinned

--> src/main.rs:46:33

|

46 | trpl::join\_all(futures).await;

| ^^^^^ the trait `Unpin` is not implemented for `{async

block@src/main.rs:8:23: 20:10}`

|

= note: consider using the `pin!` macro

consider using `Box::pin` if you need to access the pinned value outside of the current scope

= note: required for `Box<{async block@src/main.rs:8:23: 20:10}>` to implement `std::future::Future`

note: required by a bound in `JoinAll`

--> ~/.cargo/registry/src/index.crates.io-6f17d22bba15001f/futures-util-0.3.30/src/future/join\_all.rs:29:8

|

27 | pub struct JoinAll<F>

| ------- required by a bound in this struct

28 | where

29 | F: Future,

| ^^^^^^ required by this bound in `JoinAll`

Some errors have detailed explanations: E0277, E0308.

For more information about an error, try `rustc --explain E0277`.

This error message tells us not only that we need to pin the values but also why pinning is required. The trpl::join\_all function returns a struct called JoinAll. That struct is generic over a type F, which is constrained to implement the Future trait. Directly awaiting a future with await pins the future implicitly. That’s why we don’t need to use pin! everywhere we want to await futures.

However, we’re not directly awaiting a future here. Instead, we construct a new future, JoinAll, by passing a collection of futures to the join\_all function. The signature for join\_all requires that the types of the items in the collection all implement the Future trait, and Box<T> implements Future only if the T it wraps is a future that implements the Unpin trait.

That’s a lot to absorb! To really understand it, let’s dive a little further into how the Future trait actually works, in particular around pinning. Look again at the definition of the Future trait:

use std::pin::Pin;

use std::task::{Context, Poll};

pub trait Future {

type Output;

// Required method

fn poll(self: Pin<&mut Self>, cx: &mut Context<'\_>) -> Poll<Self::Output>;

}

The cx parameter and its Context type are the key to how a runtime actually knows when to check any given future while still being lazy. Again, the details of how that works are beyond the scope of this chapter, and you generally only need to think about this when writing a custom Future implementation. We’ll focus instead on the type for self, as this is the first time we’ve seen a method where self has a type annotation. A type annotation for self works like type annotations for other function parameters but with two key differences:

* It tells Rust what type self must be for the method to be called.
* It can’t be just any type. It’s restricted to the type on which the method is implemented, a reference or smart pointer to that type, or a Pin wrapping a reference to that type.

We’ll see more on this syntax in Chapter 18. For now, it’s enough to know that if we want to poll a future to check whether it is Pending or Ready(Output), we need a Pin-wrapped mutable reference to the type.

Pin is a wrapper for pointer-like types such as &, &mut, Box, and Rc. (Technically, Pin works with types that implement the Deref or DerefMut traits, but this is effectively equivalent to working only with pointers.) Pin is not a pointer itself and doesn’t have any behavior of its own like Rc and Arc do with reference counting; it’s purely a tool the compiler can use to enforce constraints on pointer usage.

Recalling that await is implemented in terms of calls to poll starts to explain the error message we saw earlier, but that was in terms of Unpin, not Pin. So how exactly does Pin relate to Unpin, and why does Future need self to be in a Pin type to call poll?

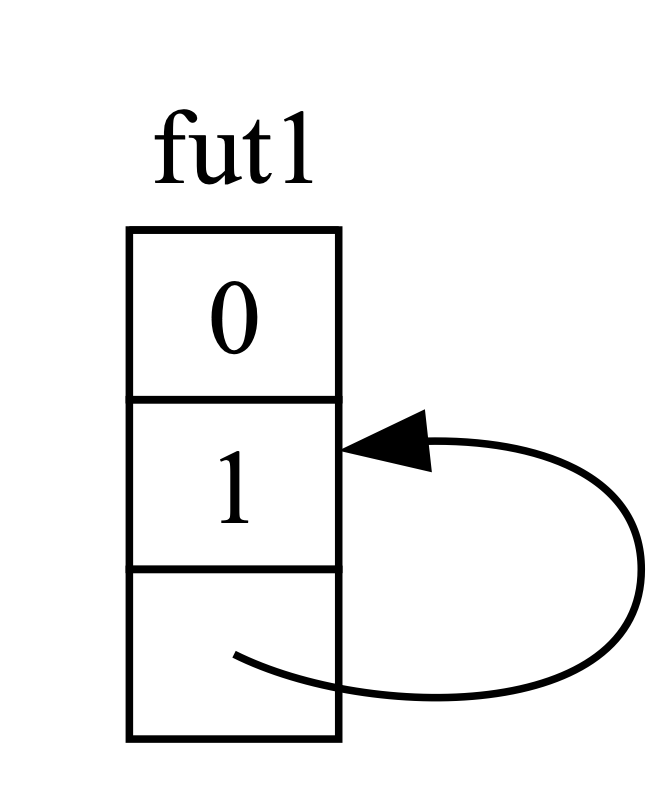
Remember from earlier in this chapter that a series of await points in a future get compiled into a state machine, and the compiler makes sure that state machine follows all of Rust’s normal rules around safety, including borrowing and ownership. To make that work, Rust looks at what data is needed between one await point and either the next await point or the end of the async block. It then creates a corresponding variant in the compiled state machine. Each variant gets the access it needs to the data that will be used in that section of the source code, whether by taking ownership of that data or by getting a mutable or immutable reference to it.

So far, so good: if we get anything wrong about the ownership or references in a given async block, the borrow checker will tell us. When we want to move around the future that corresponds to that block—like moving it into a Vec to pass to join\_all—things get trickier.

When we move a future—whether by pushing it into a data structure to use as an iterator with join\_all or by returning it from a function—that actually means moving the state machine Rust creates for us. And unlike most other types in Rust, the futures Rust creates for async blocks can end up with references to themselves in the fields of any given variant, as shown in the simplified illustration in Figure 17-4.

[f17004.svg]

<A single-column, three-row table representing a future, fut1, which has data values 0 and 1 in the first two rows and an arrow pointing from the third row back to the second row, representing an internal reference within the future.>

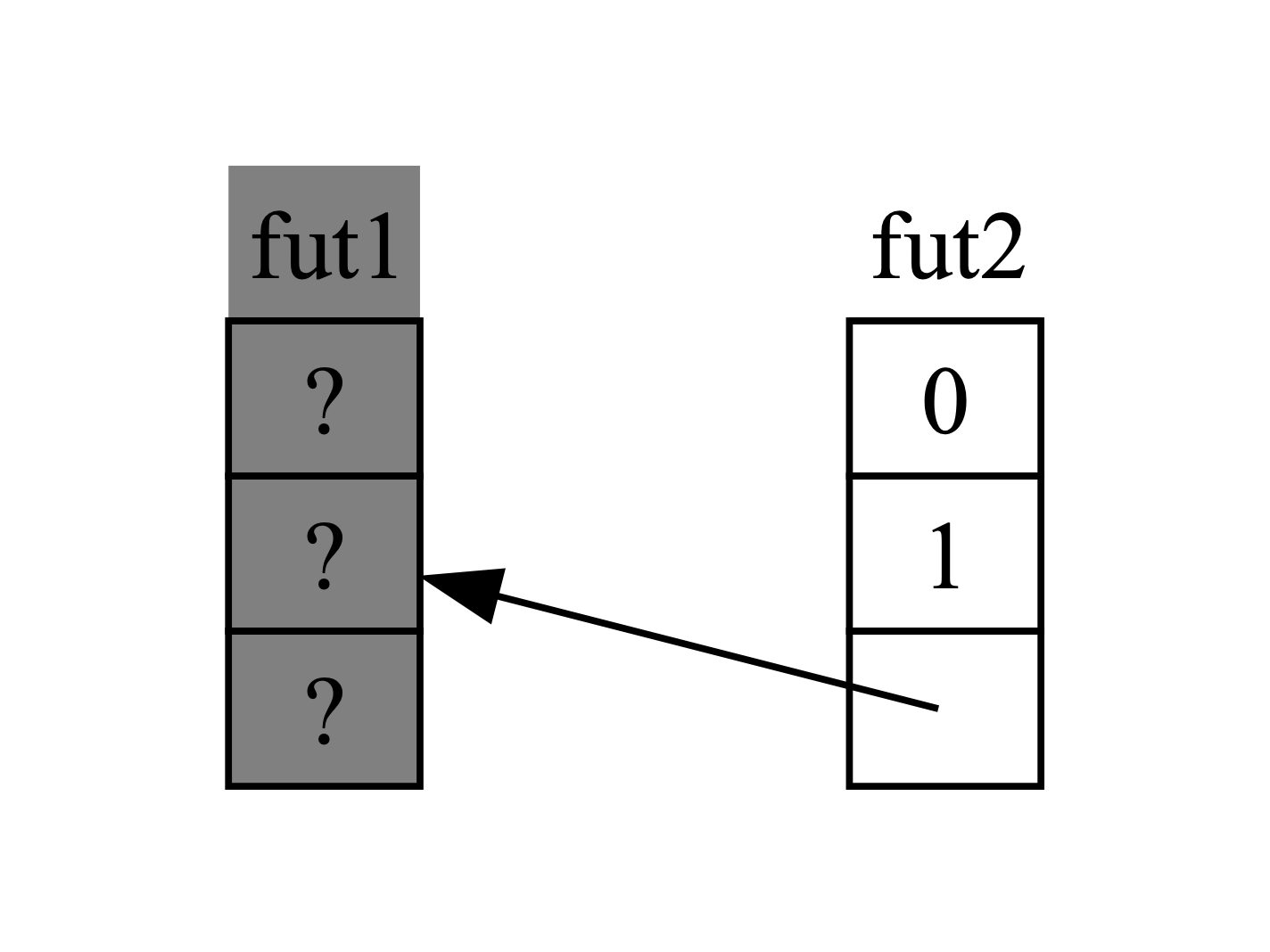


* + - * 1. A self-referential data type

By default, though, any object that has a reference to itself is unsafe to move, because references always point to the actual memory address of whatever they refer to (see Figure 17-5). If you move the data structure itself, those internal references will be left pointing to the old location. However, that memory location is now invalid. For one thing, its value will not be updated when you make changes to the data structure. For another—more important—thing, the computer is now free to reuse that memory for other purposes! You could end up reading completely unrelated data later.

[f17005.svg]

<Two tables, depicting two futures, fut1 and fut2, each of which has one column and three rows, representing the result of having moved a future out of fut1 into fut2. The first, fut1, is grayed out, with a question mark in each index, representing unknown memory. The second, fut2, has 0 and 1 in the first and second rows and an arrow pointing from its third row back to the second row of fut1, representing a pointer that is referencing the old location in memory of the future before it was moved.>



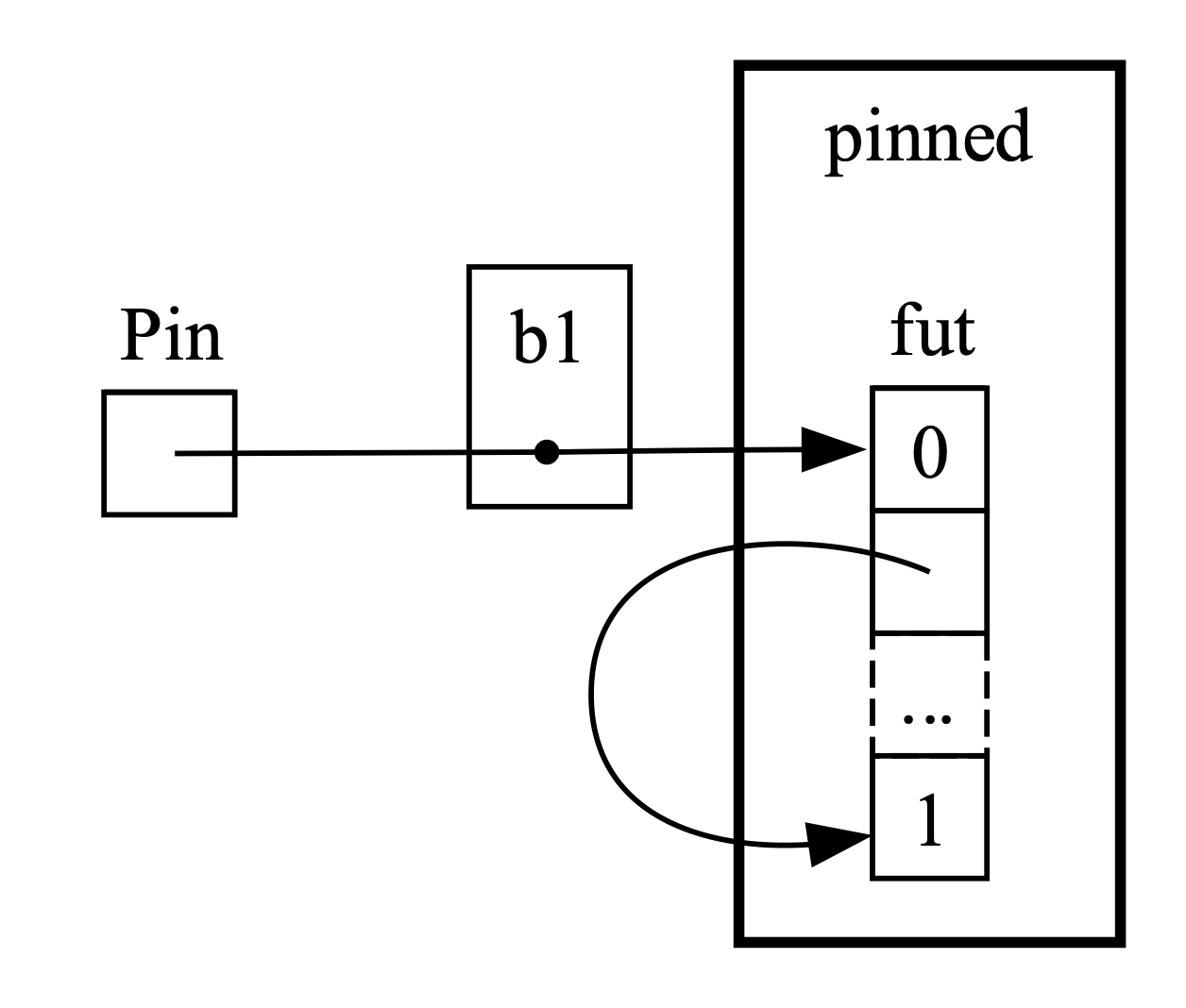
* + - * 1. The unsafe result of moving a self-referential data type

Theoretically, the Rust compiler could try to update every reference to an object whenever it gets moved, but that could add a lot of performance overhead, especially if a whole web of references needs updating. If we could instead make sure the data structure in question *doesn’t move in memory*, we wouldn’t have to update any references. This is exactly what Rust’s borrow checker is for: in safe code, it prevents you from moving any item with an active reference to it.

Pin builds on that to give us the exact guarantee we need. When we *pin* a value by wrapping a pointer to that value in Pin, it can no longer move. Thus, if you have Pin<Box<SomeType>>, you actually pin the SomeType value, *not* the Box pointer. Figure 17-6 illustrates this process.

[f17006.svg]

<Three boxes laid out side by side. The first is labeled “Pin”, the second “b1”, and the third “pinned”. Within “pinned” is a table labeled “fut”, with a single column; it represents a future with cells for each part of the data structure. Its first cell has the value “0”, its second cell has an arrow coming out of it and pointing to the fourth and final cell, which has the value “1” in it, and the third cell has dashed lines and an ellipsis to indicate there may be other parts to the data structure. All together, the “fut” table represents a future which is self-referential. An arrow leaves the box labeled “Pin”, goes through the box labeled “b1” and has terminates inside the “pinned” box at the “fut” table.>

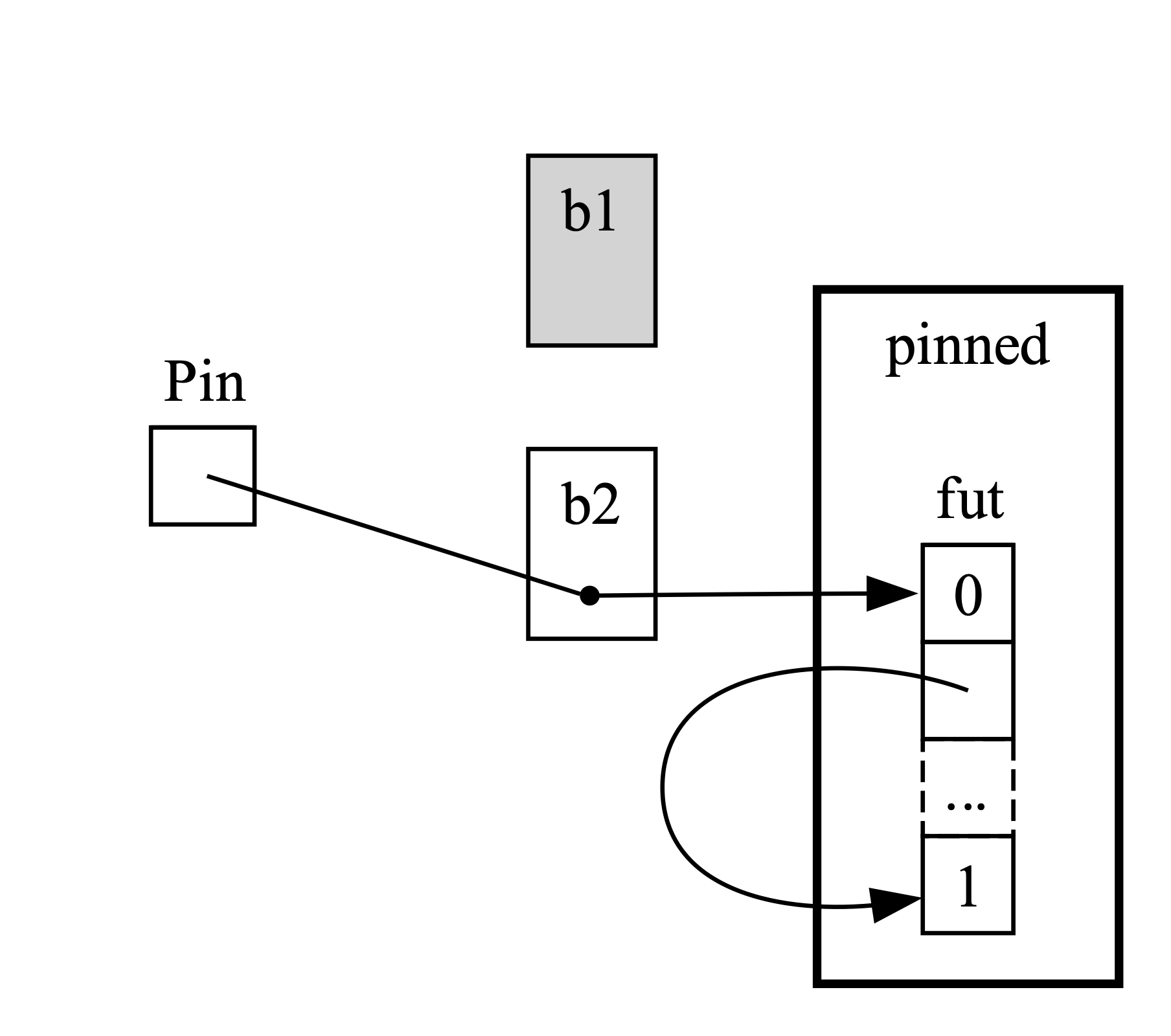


* + - * 1. Pinning a Box that points to a self-referential future type

In fact, the Box pointer can still move around freely. Remember: we care about making sure the data ultimately being referenced stays in place. If a pointer moves around, *but the data it points to* is in the same place, as in Figure 17-7, there’s no potential problem. (As an independent exercise, look at the docs for the types as well as the std::pin module and try to work out how you’d do this with a Pin wrapping a Box .) The key is that the self-referential type itself cannot move, because it is still pinned.

[f17007.svg]

<Four boxes laid out in three rough columns, identical to the previous diagram with a change to the second column. Now there are two boxes in the second column, labeled “b1” and “b2”, “b1” is grayed out, and the arrow from “Pin” goes through “b2” instead of “b1”, indicating that the pointer has moved from “b1” to “b2”, but the data in “pinned” has not moved.>



* + - * 1. Moving a Box which points to a self-referential future type

However, most types are perfectly safe to move around, even if they happen to be behind a Pin pointer. We only need to think about pinning when items have internal references. Primitive values such as numbers and Booleans are safe because they obviously don’t have any internal references. Neither do most types you normally work with in Rust. You can move around a Vec, for example, without worrying. Given what we have seen so far, if you have a Pin<Vec<String>>, you’d have to do everything via the safe but restrictive APIs provided by Pin, even though a Vec<String> is always safe to move if there are no other references to it. We need a way to tell the compiler that it’s fine to move items around in cases like this—and that’s where Unpin comes into play.

Unpin is a marker trait, similar to the Send and Sync traits we saw in Chapter 16, and thus has no functionality of its own. Marker traits exist only to tell the compiler it’s safe to use the type implementing a given trait in a particular context. Unpin informs the compiler that a given type does *not* need to uphold any guarantees about whether the value in question can be safely moved.

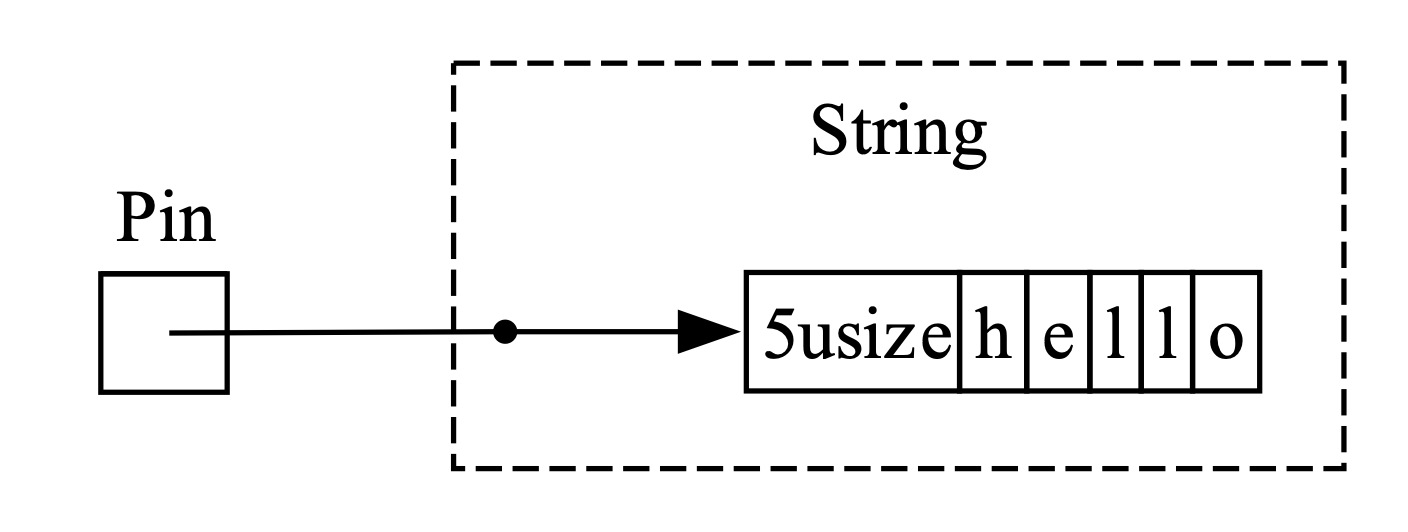
Just as with Send and Sync, the compiler implements Unpin automatically for all types where it can prove it is safe. A special case, again similar to Send and Sync, is where Unpin is *not* implemented for a type. The notation for this is impl !Unpin for *SomeType*, where *SomeType* is the name of a type that *does* need to uphold those guarantees to be safe whenever a pointer to that type is used in a Pin.

In other words, there are two things to keep in mind about the relationship between Pin and Unpin. First, Unpin is the “normal” case, and !Unpin is the special case. Second, whether a type implements Unpin or !Unpin *only* matters when you’re using a pinned pointer to that type like Pin<&mut *SomeType*>.

To make that concrete, think about a String: it has a length and the Unicode characters that make it up. We can wrap a String in Pin, as seen in Figure 17-8. However, String automatically implements Unpin, as do most other types in Rust.

[f17008.svg]

<AU: Please add Alt Text here>

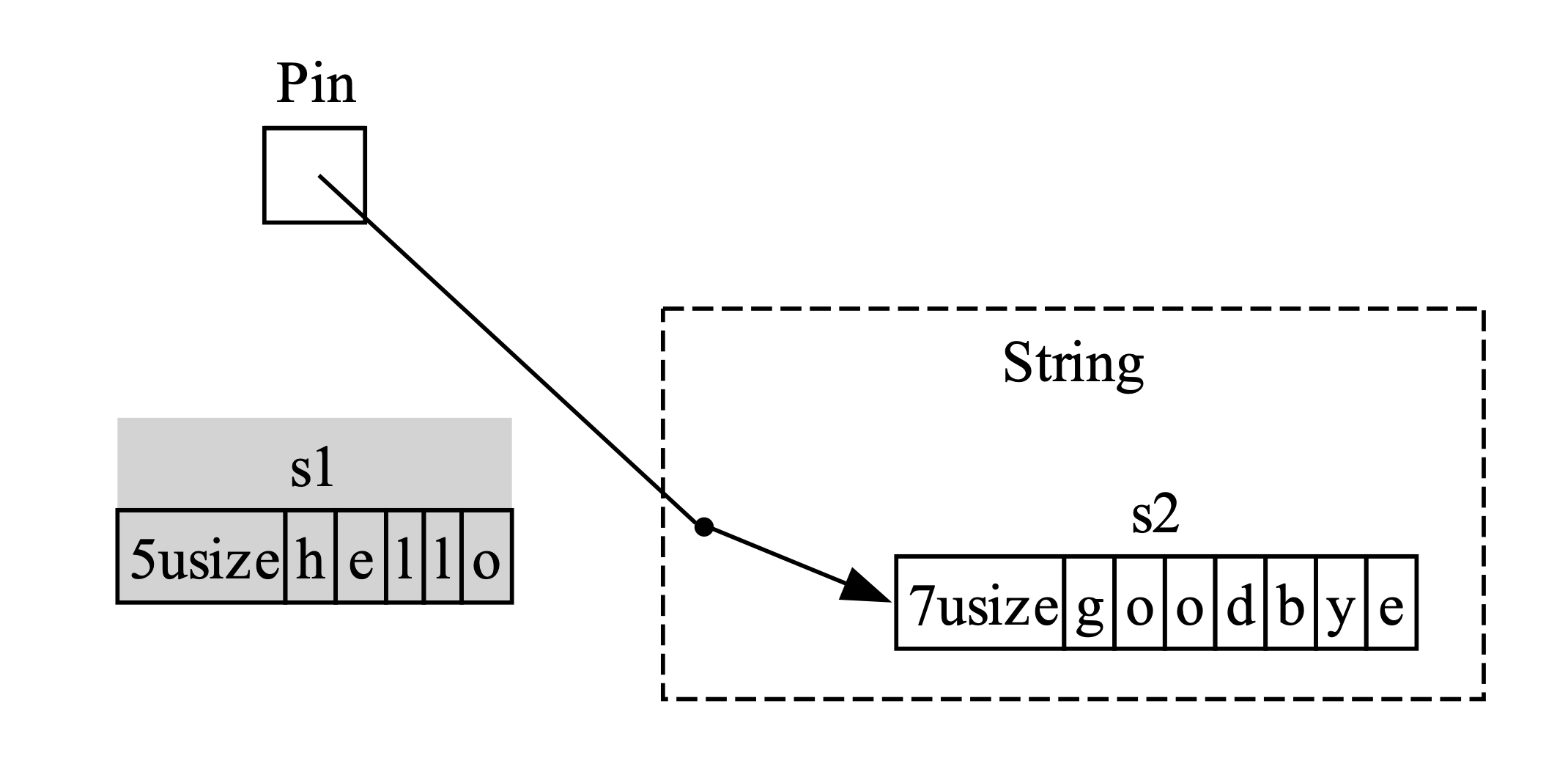


* + - * 1. Pinning a String; the dotted line indicates that the String implements the Unpin trait and thus is not pinned

As a result, we can do things that would be illegal if String implemented !Unpin instead, such as replacing one string with another at the exact same location in memory as in Figure 17-9. This doesn’t violate the Pin contract, because String has no internal references that make it unsafe to move around. That is precisely why it implements Unpin rather than !Unpin.

[f17009.svg]

<AU: Please add Alt Text here>



* + - * 1. Replacing the String with an entirely different String in memory

Now we know enough to understand the errors reported for that join\_all call from back in Listing 17-17. We originally tried to move the futures produced by async blocks into a Vec<Box<dyn Future<Output = ()>>>, but as we’ve seen, those futures may have internal references, so they don’t automatically implement Unpin. Once we pin them, we can pass the resulting Pin type into the Vec, confident that the underlying data in the futures will *not* be moved.

Pin and Unpin are mostly important for building lower-level libraries, or when you’re building a runtime itself, rather than for day-to-day Rust code. When you see these traits in error messages, though, now you’ll have a better idea of how to fix your code!

Note This combination of Pin and Unpin makes it possible to safely implement a whole class of complex types in Rust that would otherwise prove challenging because they’re self-referential. Types that require Pin show up most commonly in async Rust today, but every once in a while, you might see them in other contexts, too.

The specifics of how Pin and Unpin work, and the rules they’re required to uphold, are covered extensively in the API documentation for std::pin, so if you’re interested in learning more, that’s a great place to start.

If you want to understand how things work under the hood in even more detail, see Chapters 2 and 4 of *Asynchronous Programming in Rust*, available at *https://rust-lang.github.io/async-book*.

The Stream Trait

Now that you have a deeper grasp on the Future, Pin, and Unpin traits, we can turn our attention to the Stream trait. As you learned earlier in the chapter, streams are similar to asynchronous iterators. Unlike Iterator and Future, however, Stream has no definition in the standard library as of this writing, but there *is* a very common definition from the futures crate used throughout the ecosystem.

Let’s review the definitions of the Iterator and Future traits before looking at how a Stream trait might merge them together. From Iterator, we have the idea of a sequence: its next method provides an Option<Self::Item>. From Future, we have the idea of readiness over time: its poll method provides a Poll<Self::Output>. To represent a sequence of items that become ready over time, we define a Stream trait that puts those features together:

use std::pin::Pin;

use std::task::{Context, Poll};

trait Stream {

type Item;

fn poll\_next(

self: Pin<&mut Self>,

cx: &mut Context<'\_>

) -> Poll<Option<Self::Item>>;

}

The Stream trait defines an associated type called Item for the type of the items produced by the stream. This is similar to Iterator, where there may be zero to many items, and unlike Future, where there is always a single Output, even if it’s the unit type ().

Stream also defines a method to get those items. We call it poll\_next, to make it clear that it polls in the same way Future::poll does and produces a sequence of items in the same way Iterator::next does. Its return type combines Poll with Option. The outer type is Poll, because it has to be checked for readiness, just as a future does. The inner type is Option, because it needs to signal whether there are more messages, just as an iterator does.

Something very similar to this definition will likely end up as part of Rust’s standard library. In the meantime, it’s part of the toolkit of most runtimes, so you can rely on it, and everything we cover next should generally apply!

In the example we saw in the section on streaming, though, we didn’t use poll\_next *or* Stream, but instead used next and StreamExt. We *could* work directly in terms of the poll\_next API by hand-writing our own Stream state machines, of course, just as we *could* work with futures directly via their poll method. Using await is much nicer, though, and the StreamExt trait supplies the next method so we can do just that:

trait StreamExt: Stream {

async fn next(&mut self) -> Option<Self::Item>

where

Self: Unpin;

// other methods...

}

Note The actual definition we used earlier in the chapter looks slightly different than this, because it supports versions of Rust that did not yet support using async functions in traits. As a result, it looks like this:

fn next(&mut self) -> Next<'\_, Self> where Self: Unpin;

That Next type is a struct that implements Future and allows us to name the lifetime of the reference to self with Next<'\_, Self> so that await can work with this method.

The StreamExt trait is also the home of all the interesting methods available to use with streams. StreamExt is automatically implemented for every type that implements Stream, but these traits are defined separately to enable the community to iterate on convenience APIs without affecting the foundational trait.

In the version of StreamExt used in the trpl crate, the trait not only defines the next method but also supplies a default implementation of next that correctly handles the details of calling Stream::poll\_next. This means that even when you need to write your own streaming data type, you *only* have to implement Stream, and then anyone who uses your data type can use StreamExt and its methods with it automatically.

That’s all we’re going to cover for the lower-level details on these traits. To wrap up, let’s consider how futures (including streams), tasks, and threads all fit together!

Putting It All Together: Futures, Tasks, and Threads

As we saw in Chapter 16, threads provide one approach to concurrency. We’ve seen another approach in this chapter: using async with futures and streams. If you’re wondering when to choose one method over the other, the answer is: it depends! And in many cases, the choice isn’t threads *or* async but rather threads *and* async.

Many operating systems have supplied threading-based concurrency models for decades now, and many programming languages support them as a result. However, these models are not without their tradeoffs. On many operating systems, they use a fair bit of memory for each thread, and they come with some overhead for starting up and shutting down. Threads are also only an option when your operating system and hardware support them. Unlike mainstream desktop and mobile computers, some embedded systems don’t have an OS at all, so they also don’t have threads.

The async model provides a different—and ultimately complementary—set of tradeoffs. In the async model, concurrent operations don’t require their own threads. Instead, they can run on tasks, as when we used trpl::spawn\_task to kick off work from a synchronous function in the streams section. A task is similar to a thread, but instead of being managed by the operating system, it’s managed by library-level code: the runtime.

In the previous section, we saw that we could build a stream by using an async channel and spawning an async task we could call from synchronous code. We can do the exact same thing with a thread. In Listing 17-40, we used trpl::spawn\_task and trpl::sleep. In Listing 17-41, we replace those with the thread::spawn and thread::sleep APIs from the standard library in the get\_intervals function.

src/main.rs

fn get\_intervals() -> impl Stream<Item = u32> {

let (tx, rx) = trpl::channel();

// This is \*not\* `trpl::spawn` but `std::thread::spawn`!

thread::spawn(move || {

let mut count = 0;

loop {

// Likewise, this is \*not\* `trpl::sleep` but `std::thread::sleep`!

thread::sleep(Duration::from\_millis(1));

count += 1;

if let Err(send\_error) = tx.send(count) {

eprintln!("Could not send interval {count}: {send\_error}");

break;

};

}

});

ReceiverStream::new(rx)

}

Using the std::thread APIs instead of the async trpl APIs for the get\_intervals function

If you run this code, the output is identical to that of Listing 17-40. And notice how little changes here from the perspective of the calling code. What’s more, even though one of our functions spawned an async task on the runtime and the other spawned an OS thread, the resulting streams were unaffected by the differences.

Despite their similarities, these two approaches behave very differently, although we might have a hard time measuring it in this very simple example. We could spawn millions of async tasks on any modern personal computer. If we tried to do that with threads, we would literally run out of memory!

However, there’s a reason these APIs are so similar. Threads act as a boundary for sets of synchronous operations; concurrency is possible *between* threads. Tasks act as a boundary for sets of *asynchronous* operations; concurrency is possible both *between* and *within* tasks, because a task can switch between futures in its body. Finally, futures are Rust’s most granular unit of concurrency, and each future may represent a tree of other futures. The runtime—specifically, its executor—manages tasks, and tasks manage futures. In that regard, tasks are similar to lightweight, runtime-managed threads with added capabilities that come from being managed by a runtime instead of by the operating system.

This doesn’t mean that async tasks are always better than threads (or vice versa). Concurrency with threads is in some ways a simpler programming model than concurrency with async. That can be a strength or a weakness. Threads are somewhat “fire and forget”; they have no native equivalent to a future, so they simply run to completion without being interrupted except by the operating system itself. That is, they have no built-in support for *intratask concurrency* the way futures do. Threads in Rust also have no mechanisms for cancellation—a subject we haven’t covered explicitly in this chapter but was implied by the fact that whenever we ended a future, its state got cleaned up correctly.

These limitations also make threads harder to compose than futures. It’s much more difficult, for example, to use threads to build helpers such as the timeout and throttle methods we built earlier in this chapter. The fact that futures are richer data structures means they can be composed together more naturally, as we have seen.

Tasks, then, give us *additional* control over futures, allowing us to choose where and how to group them. And it turns out that threads and tasks often work very well together, because tasks can (at least in some runtimes) be moved around between threads. In fact, under the hood, the runtime we’ve been using—including the spawn\_blocking and spawn\_task functions—is multithreaded by default! Many runtimes use an approach called *work stealing* to transparently move tasks around between threads, based on how the threads are currently being utilized, to improve the system’s overall performance. That approach actually requires threads *and* tasks, and therefore futures.

When thinking about which method to use when, consider these rules of thumb:

* If the work is *very parallelizable*, such as processing a bunch of data where each part can be processed separately, threads are a better choice.
* If the work is *very concurrent*, such as handling messages from a bunch of different sources that may come in at different intervals or different rates, async is a better choice.

And if you need both parallelism and concurrency, you don’t have to choose between threads and async. You can use them together freely, letting each play the part it’s best at. For example, Listing 17-42 shows a fairly common example of this kind of mix in real-world Rust code.

src/main.rs

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

fn main() {

let (tx, mut rx) = trpl::channel();

thread::spawn(move || {

for i in 1..11 {

tx.send(i).unwrap();

thread::sleep(Duration::from\_secs(1));

}

});

trpl::run(async {

while let Some(message) = rx.recv().await {

println!("{message}");

}

});

}

Sending messages with blocking code in a thread and awaiting the messages in an async block

We begin by creating an async channel, then spawn a thread that takes ownership of the sender side of the channel. Within the thread, we send the numbers 1 through 10, sleeping for a second between each. Finally, we run a future created with an async block passed to trpl::run just as we have throughout the chapter. In that future, we await those messages, just as in the other message-passing examples we have seen.

To return to the scenario we opened the chapter with, imagine running a set of video encoding tasks using a dedicated thread (because video encoding is compute-bound) but notifying the UI that those operations are done with an async channel. There are countless examples of these kinds of combinations in real-world use cases.

Summary

This isn’t the last you’ll see of concurrency in this book. The project in Chapter 21 will apply these concepts in a more realistic situation than the simpler examples discussed here and compare problem-solving with threading versus tasks and futures more directly.

No matter which of these approaches you choose, Rust gives you the tools you need to write safe, fast, concurrent code—whether for a high-throughput web server or an embedded operating system.

Next, we’ll talk about idiomatic ways to model problems and structure solutions as your Rust programs get bigger. In addition, we’ll discuss how Rust’s idioms relate to those you might be familiar with from object-oriented programming.