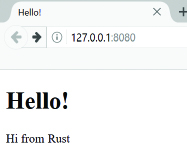
Final Project: Building a Multithreaded Web Server

It’s been a long journey, but we’ve reached the end of the book. In this chapter, we’ll build one more project together to demonstrate some of the concepts we covered in the final chapters, as well as recap some earlier lessons.

For our final project, we’ll make a web server that says “Hello!” and looks like Figure 21-1 in a web browser.

Here is our plan for building the web server:

1. Learn a bit about TCP and HTTP.
2. Listen for TCP connections on a socket.
3. Parse a small number of HTTP requests.
4. Create a proper HTTP response.
5. Improve the throughput of our server with a thread pool.



* + - * 1. Our final shared project

Before we get started, we should mention two details. First, the method we’ll use won’t be the best way to build a web server with Rust. Community members have published a number of production-ready crates available at https://crates.io that provide more complete web server and thread pool implementations than what we’ll build. However, our intention in this chapter is to help you learn, not to take the easy route. Because Rust is a systems programming language, we can choose the level of abstraction we want to work with and can go to a lower level than is possible or practical in other languages.

Second, we will not be using async and await here. Building a thread pool is a big enough challenge on its own, without adding in building an async runtime! However, we will note how async and await might be applicable to some of the same problems we will see in this chapter. Ultimately, as we noted back in Chapter 17, many async runtimes use thread pools for managing their work.

We’ll therefore write the basic HTTP server and thread pool manually so that you can learn the general ideas and techniques behind the crates you might use in the future.

Building a Single-Threaded Web Server

We’ll start by getting a single-threaded web server working. Before we begin, let’s look at a quick overview of the protocols involved in building web servers. The details of these protocols are beyond the scope of this book, but a brief overview will give you the information you need.

The two main protocols involved in web servers are Hypertext Transfer Protocol (HTTP) and Transmission Control Protocol (TCP). Both protocols are request-response protocols, meaning a client initiates requests and a server listens to the requests and provides a response to the client. The contents of those requests and responses are defined by the protocols.

TCP is the lower-level protocol that describes the details of how information gets from one server to another but doesn’t specify what that information is. HTTP builds on top of TCP by defining the contents of the requests and responses. It’s technically possible to use HTTP with other protocols, but in the vast majority of cases, HTTP sends its data over TCP. We’ll work with the raw bytes of TCP and HTTP requests and responses.

Listening to the TCP Connection

Our web server needs to listen to a TCP connection, so that’s the first part we’ll work on. The standard library offers a std::net module that lets us do this. Let’s make a new project in the usual fashion:

$ cargo new hello

Created binary (application) `hello` project

$ cd hello

Now enter the code in Listing 21-1 in src/main.rs to start. This code will listen at the local address 127.0.0.1:7878 for incoming TCP streams. When it gets an incoming stream, it will print Connection established!.

src/main.rs

use std::net::TcpListener;

fn main() {

1 let listener = TcpListener::bind("127.0.0.1:7878").unwrap();

2 for stream in listener.incoming() {

3 let stream = stream.unwrap();

4 println!("Connection established!");

}

}

Listening for incoming streams and printing a message when we receive a stream

Using TcpListener, we can listen for TCP connections at the address 127.0.0.1:7878 1. In the address, the section before the colon is an IP address representing your computer (this is the same on every computer and doesn’t represent the authors’ computer specifically), and 7878 is the port. We’ve chosen this port for two reasons: HTTP isn’t normally accepted on this port, so our server is unlikely to conflict with any other web server you might have running on your machine, and 7878 is rust typed on a telephone.

The bind function in this scenario works like the new function in that it will return a new TcpListener instance. The function is called bind because, in networking, connecting to a port to listen to is known as “binding to a port.”

The bind function returns a Result<T, E>, which indicates that it’s possible for binding to fail, for example, if we ran two instances of our program and so had two programs listening to the same port. Because we’re writing a basic server just for learning purposes, we won’t worry about handling these kinds of errors; instead, we use unwrap to stop the program if errors happen.

The incoming method on TcpListener returns an iterator that gives us a sequence of streams 2 (more specifically, streams of type TcpStream). A single stream represents an open connection between the client and the server. Connection is the name for the full request and response process in which a client connects to the server, the server generates a response, and the server closes the connection. As such, we will read from the TcpStream to see what the client sent and then write our response to the stream to send data back to the client. Overall, this for loop will process each connection in turn and produce a series of streams for us to handle.

For now, our handling of the stream consists of calling unwrap to terminate our program if the stream has any errors 3; if there aren’t any errors, the program prints a message 4. We’ll add more functionality for the success case in the next listing. The reason we might receive errors from the incoming method when a client connects to the server is that we’re not actually iterating over connections. Instead, we’re iterating over connection attempts. The connection might not be successful for a number of reasons, many of them operating system specific. For example, many operating systems have a limit to the number of simultaneous open connections they can support; new connection attempts beyond that number will produce an error until some of the open connections are closed.

Let’s try running this code! Invoke cargo run in the terminal and then load 127.0.0.1:7878 in a web browser. The browser should show an error message like “Connection reset” because the server isn’t currently sending back any data. But when you look at your terminal, you should see several messages that were printed when the browser connected to the server!

Running `target/debug/hello`

Connection established!

Connection established!

Connection established!

Sometimes you’ll see multiple messages printed for one browser request; the reason might be that the browser is making a request for the page as well as a request for other resources, like the favicon.ico icon that appears in the browser tab.

It could also be that the browser is trying to connect to the server multiple times because the server isn’t responding with any data. When stream goes out of scope and is dropped at the end of the loop, the connection is closed as part of the drop implementation. Browsers sometimes deal with closed connections by retrying, because the problem might be temporary.

Browsers also sometimes open multiple connections to the server without sending any requests so that if they do later send requests, those requests can happen more quickly. When this occurs, our server will see each connection, regardless of whether there are any requests over that connection. Many versions of Chrome-based browsers do this, for example; you can disable that optimization by using private browsing mode or using a different browser.

The important factor is that we’ve successfully gotten a handle to a TCP connection!

Remember to stop the program by pressing ctrl-C when you’re done running a particular version of the code. Then, restart the program by invoking the cargo run command after you’ve made each set of code changes to make sure you’re running the newest code.

Reading the Request

Let’s implement the functionality to read the request from the browser! To separate the concerns of first getting a connection and then taking some action with the connection, we’ll start a new function for processing connections. In this new handle\_connection function, we’ll read data from the TCP stream and print it so that we can see the data being sent from the browser. Change the code to look like Listing 21-2.

src/main.rs

1 use std::{

io::{BufReader, prelude::\*},

net::{TcpListener, TcpStream},

};

fn main() {

let listener = TcpListener::bind("127.0.0.1:7878").unwrap();

for stream in listener.incoming() {

let stream = stream.unwrap();

2 handle\_connection(stream);

}

}

fn handle\_connection(mut stream: TcpStream) {

3 let buf\_reader = BufReader::new(&stream);

4 let http\_request: Vec<\_> = buf\_reader

5 .lines()

6 .map(|result| result.unwrap())

7 .take\_while(|line| !line.is\_empty())

.collect();

8 println!("Request: {http\_request:#?}");

}

Reading from the TcpStream and printing the data

We bring std::io::BufReader and std::io::prelude into scope to get access to traits and types that let us read from and write to the stream 1. In the for loop in the main function, instead of printing a message that says we made a connection, we now call the new handle\_connection function and pass the stream to it 2.

In the handle\_connection function, we create a new BufReader instance that wraps a reference to the stream 3. The BufReader adds buffering by managing calls to the std::io::Read trait methods for us.

We create a variable named http\_request to collect the lines of the request the browser sends to our server. We indicate that we want to collect these lines in a vector by adding the Vec<\_> type annotation 4.

BufReader implements the std::io::BufRead trait, which provides the lines method 5. The lines method returns an iterator of Result<String, std::io::Error> by splitting the stream of data whenever it sees a newline byte. To get each String, we map and unwrap each Result 6. The Result might be an error if the data isn’t valid UTF-8 or if there was a problem reading from the stream. Again, a production program should handle these errors more gracefully, but we’re choosing to stop the program in the error case for simplicity.

The browser signals the end of an HTTP request by sending two newline characters in a row, so to get one request from the stream, we take lines until we get a line that is the empty string 7. Once we’ve collected the lines into the vector, we’re printing them out using pretty debug formatting 8 so that we can take a look at the instructions the web browser is sending to our server.

Let’s try this code! Start the program and make a request in a web browser again. Note that we’ll still get an error page in the browser, but our program’s output in the terminal will now look similar to this:

$ cargo run

Compiling hello v0.1.0 (file:///projects/hello)

Finished `dev` profile [unoptimized + debuginfo] target(s) in 0.42s

Running `target/debug/hello`

Request: [

"GET / HTTP/1.1",

"Host: 127.0.0.1:7878",

"User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10.15; rv:99.0)

Gecko/20100101 Firefox/99.0",

"Accept: text/html,application/xhtml+xml,application/xml;

q=0.9,image/avif,image/webp,\*/\*;q=0.8",

"Accept-Language: en-US,en;q=0.5",

"Accept-Encoding: gzip, deflate, br",

"DNT: 1",

"Connection: keep-alive",

"Upgrade-Insecure-Requests: 1",

"Sec-Fetch-Dest: document",

"Sec-Fetch-Mode: navigate",

"Sec-Fetch-Site: none",

"Sec-Fetch-User: ?1",

"Cache-Control: max-age=0",

]

Depending on your browser, you might get slightly different output. Now that we’re printing the request data, we can see why we get multiple connections from one browser request by looking at the path after GET in the first line of the request. If the repeated connections are all requesting /, we know the browser is trying to fetch / repeatedly because it’s not getting a response from our program.

Let’s break down this request data to understand what the browser is asking of our program.

Looking More Closely at an HTTP Request

HTTP is a text-based protocol, and a request takes this format:

Method Request-URI HTTP-Version CRLF

headers CRLF

message-body

The first line is the request line that holds information about what the client is requesting. The first part of the request line indicates the method being used, such as GET or POST, which describes how the client is making this request. Our client used a GET request, which means it is asking for information.

The next part of the request line is /, which indicates the uniform resource identifier (URI) the client is requesting: A URI is almost, but not quite, the same as a uniform resource locator (URL). The difference between URIs and URLs isn’t important for our purposes in this chapter, but the HTTP spec uses the term URI, so we can just mentally substitute URL for URI here.

The last part is the HTTP version the client uses, and then the request line ends in a CRLF sequence. (CRLF stands for carriage return and line feed, which are terms from the typewriter days!) The CRLF sequence can also be written as \r\n, where \r is a carriage return and \n is a line feed. The CRLF sequence separates the request line from the rest of the request data. Note that when the CRLF is printed, we see a new line start rather than \r\n.

Looking at the request line data we received from running our program so far, we see that GET is the method, / is the request URI, and HTTP/1.1 is the version.

After the request line, the remaining lines starting from Host: onward are headers. GET requests have no body.

Try making a request from a different browser or asking for a different address, such as 127.0.0.1:7878/test, to see how the request data changes.

Now that we know what the browser is asking for, let’s send back some data!

Writing a Response

We’re going to implement sending data in response to a client request. Responses have the following format:

HTTP-Version Status-Code Reason-Phrase CRLF

headers CRLF

message-body

The first line is a status line that contains the HTTP version used in the response, a numeric status code that summarizes the result of the request, and a reason phrase that provides a text description of the status code. After the CRLF sequence are any headers, another CRLF sequence, and the body of the response.

Here is an example response that uses HTTP version 1.1 and has a status code of 200, an OK reason phrase, no headers, and no body:

HTTP/1.1 200 OK\r\n\r\n

The status code 200 is the standard success response. The text is a tiny successful HTTP response. Let’s write this to the stream as our response to a successful request! From the handle\_connection function, remove the println! that was printing the request data and replace it with the code in Listing 21-3.

src/main.rs

fn handle\_connection(mut stream: TcpStream) {

let buf\_reader = BufReader::new(&stream);

let http\_request: Vec<\_> = buf\_reader

.lines()

.map(|result| result.unwrap())

.take\_while(|line| !line.is\_empty())

.collect();

1 let response = "HTTP/1.1 200 OK\r\n\r\n";

2 stream.write\_all(response.as\_bytes()).unwrap(); 3

}

Writing a tiny successful HTTP response to the stream

The first new line defines the response variable that holds the success message’s data 1. Then, we call as\_bytes on our response to convert the string data to bytes 3. The write\_all method on stream takes a &[u8] and sends those bytes directly down the connection 2. Because the write\_all operation could fail, we use unwrap on any error result as before. Again, in a real application, you would add error handling here.

With these changes, let’s run our code and make a request. We’re no longer printing any data to the terminal, so we won’t see any output other than the output from Cargo. When you load 127.0.0.1:7878 in a web browser, you should get a blank page instead of an error. You’ve just handcoded receiving an HTTP request and sending a response!

Returning Real HTML

Let’s implement the functionality for returning more than a blank page. Create the new file hello.html in the root of your project directory, not in the src directory. You can input any HTML you want; Listing 21-4 shows one possibility.

hello.html

<!DOCTYPE html>

<html lang="en">

<head>

<meta charset="utf-8">

<title>Hello!</title>

</head>

<body>

<h1>Hello!</h1>

<p>Hi from Rust</p>

</body>

</html>

A sample HTML file to return in a response

This is a minimal HTML5 document with a heading and some text. To return this from the server when a request is received, we’ll modify handle\_connection as shown in Listing 21-5 to read the HTML file, add it to the response as a body, and send it.

src/main.rs

use std::{

1 fs,

io::{BufReader, prelude::\*},

net::{TcpListener, TcpStream},

};

--snip--

fn handle\_connection(mut stream: TcpStream) {

let buf\_reader = BufReader::new(&stream);

let http\_request: Vec<\_> = buf\_reader

.lines()

.map(|result| result.unwrap())

.take\_while(|line| !line.is\_empty())

.collect();

let status\_line = "HTTP/1.1 200 OK";

let contents = fs::read\_to\_string("hello.html").unwrap();

let length = contents.len();

2 let response = format!(

"{status\_line}\r\n\

Content-Length: {length}\r\n\r\n\

{contents}"

);

stream.write\_all(response.as\_bytes()).unwrap();

}

Sending the contents of hello.html as the body of the response

We’ve added fs to the use statement to bring the standard library’s filesystem module into scope 1. The code for reading the contents of a file to a string should look familiar; we used it when we read the contents of a file for our I/O project in Listing 12-4.

Next, we use format! to add the file’s contents as the body of the success response 2. To ensure a valid HTTP response, we add the Content-Length header, which is set to the size of our response body—in this case, the size of hello.html.

Run this code with cargo run and load 127.0.0.1:7878 in your browser; you should see your HTML rendered!

Currently, we’re ignoring the request data in http\_request and just sending back the contents of the HTML file unconditionally. That means if you try requesting 127.0.0.1:7878/something-else in your browser, you’ll still get back this same HTML response. At the moment, our server is very limited and does not do what most web servers do. We want to customize our responses depending on the request and only send back the HTML file for a well-formed request to /.

Validating the Request and Selectively Responding

Right now, our web server will return the HTML in the file no matter what the client requested. Let’s add functionality to check that the browser is requesting / before returning the HTML file and to return an error if the browser requests anything else. For this we need to modify handle\_connection, as shown in Listing 21-6. This new code checks the content of the request received against what we know a request for / looks like and adds if and else blocks to treat requests differently.

src/main.rs

--snip--

fn handle\_connection(mut stream: TcpStream) {

let buf\_reader = BufReader::new(&stream);

1 let request\_line = buf\_reader

.lines()

.next()

.unwrap()

.unwrap();

2 if request\_line == "GET / HTTP/1.1" {

let status\_line = "HTTP/1.1 200 OK";

let contents = fs::read\_to\_string("hello.html").unwrap();

let length = contents.len();

let response = format!(

"{status\_line}\r\n\

Content-Length: {length}\r\n\r\n\

{contents}"

);

stream.write\_all(response.as\_bytes()).unwrap();

3 } else {

// some other request

}

}

Handling requests to / differently from other requests

We’re only going to be looking at the first line of the HTTP request, so rather than reading the entire request into a vector, we’re calling next to get the first item from the iterator 1. The first unwrap takes care of the Option and stops the program if the iterator has no items. The second unwrap handles the Result and has the same effect as the unwrap that was in the map added in Listing 21-2.

Next, we check the request\_line to see if it equals the request line of a GET request to the / path 2. If it does, the if block returns the contents of our HTML file.

If the request\_line does not equal the GET request to the / path, it means we’ve received some other request. We’ll add code to the else block 3 in a moment to respond to all other requests.

Run this code now and request 127.0.0.1:7878; you should get the HTML in hello.html. If you make any other request, such as 127.0.0.1:7878/something-else, you’ll get a connection error like those you saw when running the code in Listing 21-1 and Listing 21-2.

Now let’s add the code in Listing 21-7 to the else block to return a response with the status code 404, which signals that the content for the request was not found. We’ll also return some HTML for a page to render in the browser indicating the response to the end user.

src/main.rs

--snip--

} else {

1 let status\_line = "HTTP/1.1 404 NOT FOUND";

2 let contents = fs::read\_to\_string("404.html").unwrap();

let length = contents.len();

let response = format!(

"{status\_line}\r\n\

Content-Length: {length}\r\n\r\n

{contents}"

);

stream.write\_all(response.as\_bytes()).unwrap();

}

Responding with status code 404 and an error page if anything other than / was requested

Here, our response has a status line with status code 404 and the reason phrase NOT FOUND 1. The body of the response will be the HTML in the file 404.html 2. You’ll need to create a 404.html file next to hello.html for the error page; again, feel free to use any HTML you want, or use the example HTML in Listing 21-8.

404.html

<!DOCTYPE html>

<html lang="en">

<head>

<meta charset="utf-8">

<title>Hello!</title>

</head>

<body>

<h1>Oops!</h1>

<p>Sorry, I don't know what you're asking for.</p>

</body>

</html>

Sample content for the page to send back with any 404 response

With these changes, run your server again. Requesting 127.0.0.1:7878 should return the contents of hello.html, and any other request, like 127.0.0.1:7878/foo, should return the error HTML from 404.html.

Refactoring

At the moment, the if and else blocks have a lot of repetition: They’re both reading files and writing the contents of the files to the stream. The only differences are the status line and the filename. Let’s make the code more concise by pulling out those differences into separate if and else lines that will assign the values of the status line and the filename to variables; we can then use those variables unconditionally in the code to read the file and write the response. Listing 21-9 shows the resultant code after replacing the large if and else blocks.

src/main.rs

--snip--

fn handle\_connection(mut stream: TcpStream) {

--snip--

let (status\_line, filename) =

if request\_line == "GET / HTTP/1.1" {

("HTTP/1.1 200 OK", "hello.html")

} else {

("HTTP/1.1 404 NOT FOUND", "404.html")

};

let contents = fs::read\_to\_string(filename).unwrap();

let length = contents.len();

let response = format!(

"{status\_line}\r\n\

Content-Length: {length}\r\n\r\n\

{contents}"

);

stream.write\_all(response.as\_bytes()).unwrap();

}

Refactoring the if and else blocks to contain only the code that differs between the two cases

Now the if and else blocks only return the appropriate values for the status line and filename in a tuple; we then use destructuring to assign these two values to status\_line and filename using a pattern in the let statement, as discussed in Chapter 19.

The previously duplicated code is now outside the if and else blocks and uses the status\_line and filename variables. This makes it easier to see the difference between the two cases, and it means we have only one place to update the code if we want to change how the file reading and response writing work. The behavior of the code in Listing 21-9 will be the same as that in Listing 21-7.

Awesome! We now have a simple web server in approximately 40 lines of Rust code that responds to one request with a page of content and responds to all other requests with a 404 response.

Currently, our server runs in a single thread, meaning it can only serve one request at a time. Let’s examine how that can be a problem by simulating some slow requests. Then, we’ll fix it so that our server can handle multiple requests at once.

From a Single-Threaded to a Multithreaded Server

Right now, the server will process each request in turn, meaning it won’t process a second connection until the first connection is finished processing. If the server received more and more requests, this serial execution would be less and less optimal. If the server receives a request that takes a long time to process, subsequent requests will have to wait until the long request is finished, even if the new requests can be processed quickly. We’ll need to fix this, but first we’ll look at the problem in action.

Simulating a Slow Request

We’ll look at how a slowly processing request can affect other requests made to our current server implementation. Listing 21-10 implements handling a request to /sleep with a simulated slow response that will cause the server to sleep for five seconds before responding.

src/main.rs

use std::{

fs,

io::{BufReader, prelude::\*},

net::{TcpListener, TcpStream},

thread,

time::Duration,

};

--snip--

fn handle\_connection(mut stream: TcpStream) {

--snip--

1 let (status\_line, filename) = match &request\_line[..] {

2 "GET / HTTP/1.1" => ("HTTP/1.1 200 OK", "hello.html"),

3 "GET /sleep HTTP/1.1" => {

thread::sleep(Duration::from\_secs(5));

("HTTP/1.1 200 OK", "hello.html")

}

4 \_ => ("HTTP/1.1 404 NOT FOUND", "404.html"),

};

--snip--

}

Simulating a slow request by sleeping for five seconds

We switched from if to match now that we have three cases 1. We need to explicitly match on a slice of request\_line to pattern-match against the string literal values; match doesn’t do automatic referencing and dereferencing, like the equality method does.

The first arm 2 is the same as the if block from Listing 21-9. The second arm 3 matches a request to /sleep. When that request is received, the server will sleep for five seconds before rendering the successful HTML page. The third arm 4 is the same as the else block from Listing 21-9.

You can see how primitive our server is: Real libraries would handle the recognition of multiple requests in a much less verbose way!

Start the server using cargo run. Then, open two browser windows: one for http://127.0.0.1:7878 and the other for http://127.0.0.1:7878/sleep. If you enter the / URI a few times, as before, you’ll see it respond quickly. But if you enter /sleep and then load /, you’ll see that / waits until sleep has slept for its full five seconds before loading.

There are multiple techniques we could use to avoid requests backing up behind a slow request, including using async as we did Chapter 17; the one we’ll implement is a thread pool.

Improving Throughput with a Thread Pool

A thread pool is a group of spawned threads that are ready and waiting to handle a task. When the program receives a new task, it assigns one of the threads in the pool to the task, and that thread will process the task. The remaining threads in the pool are available to handle any other tasks that come in while the first thread is processing. When the first thread is done processing its task, it’s returned to the pool of idle threads, ready to handle a new task. A thread pool allows you to process connections concurrently, increasing the throughput of your server.

We’ll limit the number of threads in the pool to a small number to protect us from DoS attacks; if we had our program create a new thread for each request as it came in, someone making 10 million requests to our server could wreak havoc by using up all our server’s resources and grinding the processing of requests to a halt.

Rather than spawning unlimited threads, then, we’ll have a fixed number of threads waiting in the pool. Requests that come in are sent to the pool for processing. The pool will maintain a queue of incoming requests. Each of the threads in the pool will pop off a request from this queue, handle the request, and then ask the queue for another request. With this design, we can process up to N requests concurrently, where N is the number of threads. If each thread is responding to a long-running request, subsequent requests can still back up in the queue, but we’ve increased the number of long-running requests we can handle before reaching that point.

This technique is just one of many ways to improve the throughput of a web server. Other options you might explore are the fork/join model, the single-threaded async I/O model, and the multithreaded async I/O model. If you’re interested in this topic, you can read more about other solutions and try to implement them; with a low-level language like Rust, all of these options are possible.

Before we begin implementing a thread pool, let’s talk about what using the pool should look like. When you’re trying to design code, writing the client interface first can help guide your design. Write the API of the code so that it’s structured in the way you want to call it; then, implement the functionality within that structure rather than implementing the functionality and then designing the public API.

Similar to how we used test-driven development in the project in Chapter 12, we’ll use compiler-driven development here. We’ll write the code that calls the functions we want, and then we’ll look at errors from the compiler to determine what we should change next to get the code to work. Before we do that, however, we’ll explore the technique we’re not going to use as a starting point.

Spawning a Thread for Each Request

First, let’s explore how our code might look if it did create a new thread for every connection. As mentioned earlier, this isn’t our final plan due to the problems with potentially spawning an unlimited number of threads, but it is a starting point to get a working multithreaded server first. Then, we’ll add the thread pool as an improvement, and contrasting the two solutions will be easier.

Listing 21-11 shows the changes to make to main to spawn a new thread to handle each stream within the for loop.

src/main.rs

fn main() {

let listener = TcpListener::bind("127.0.0.1:7878").unwrap();

for stream in listener.incoming() {

let stream = stream.unwrap();

thread::spawn(|| {

handle\_connection(stream);

});

}

}

Spawning a new thread for each stream

As you learned in Chapter 16, thread::spawn will create a new thread and then run the code in the closure in the new thread. If you run this code and load /sleep in your browser, then / in two more browser tabs, you’ll indeed see that the requests to / don’t have to wait for /sleep to finish. However, as we mentioned, this will eventually overwhelm the system because you’d be making new threads without any limit.

You may also recall from Chapter 17 that this is exactly the kind of situation where async and await really shine! Keep that in mind as we build the thread pool and think about how things would look different or the same with async.

Creating a Finite Number of Threads

We want our thread pool to work in a similar, familiar way so that switching from threads to a thread pool doesn’t require large changes to the code that uses our API. Listing 21-12 shows the hypothetical interface for a ThreadPool struct we want to use instead of thread::spawn.

src/main.rs

fn main() {

let listener = TcpListener::bind("127.0.0.1:7878").unwrap();

1 let pool = ThreadPool::new(4);

for stream in listener.incoming() {

let stream = stream.unwrap();

2 pool.execute(|| {

handle\_connection(stream);

});

}

}

Our ideal ThreadPool interface

We use ThreadPool::new to create a new thread pool with a configurable number of threads, in this case four 1. Then, in the for loop, pool.execute has a similar interface as thread::spawn in that it takes a closure that the pool should run for each stream 2. We need to implement pool.execute so that it takes the closure and gives it to a thread in the pool to run. This code won’t yet compile, but we’ll try so that the compiler can guide us in how to fix it.

Building ThreadPool Using Compiler-Driven Development

Make the changes in Listing 21-12 to src/main.rs, and then let’s use the compiler errors from cargo check to drive our development. Here is the first error we get:

$ cargo check

Checking hello v0.1.0 (file:///projects/hello)

error[E0433]: failed to resolve: use of undeclared type `ThreadPool`

--> src/main.rs:11:16

|

11 | let pool = ThreadPool::new(4);

| ^^^^^^^^^^ use of undeclared type `ThreadPool`

Great! This error tells us we need a ThreadPool type or module, so we’ll build one now. Our ThreadPool implementation will be independent of the kind of work our web server is doing. So, let’s switch the hello crate from a binary crate to a library crate to hold our ThreadPool implementation. After we change to a library crate, we could also use the separate thread pool library for any work we want to do using a thread pool, not just for serving web requests.

Create a src/lib.rs file that contains the following, which is the simplest definition of a ThreadPool struct that we can have for now:

src/lib.rs

pub struct ThreadPool;

Then, edit the main.rs file to bring ThreadPool into scope from the library crate by adding the following code to the top of src/main.rs:

src/main.rs

use hello::ThreadPool;

This code still won’t work, but let’s check it again to get the next error that we need to address:

$ cargo check

Checking hello v0.1.0 (file:///projects/hello)

error[E0599]: no function or associated item named `new` found for

struct `ThreadPool` in the current scope

--> src/main.rs:12:28

|

12 | let pool = ThreadPool::new(4);

| ^^^ function or associated item not

found in `ThreadPool`

This error indicates that next we need to create an associated function named new for ThreadPool. We also know that new needs to have one parameter that can accept 4 as an argument and should return a ThreadPool instance. Let’s implement the simplest new function that will have those characteristics:

src/lib.rs

pub struct ThreadPool;

impl ThreadPool {

pub fn new(size: usize) -> ThreadPool {

ThreadPool

}

}

We chose usize as the type of the size parameter because we know that a negative number of threads doesn’t make any sense. We also know we’ll use this 4 as the number of elements in a collection of threads, which is what the usize type is for, as discussed in “Integer Types” on page XX.

Let’s check the code again:

$ cargo check

Checking hello v0.1.0 (file:///projects/hello)

error[E0599]: no method named `execute` found for struct `ThreadPool`

in the current scope

--> src/main.rs:17:14

|

17 | pool.execute(|| {

| -----^^^^^^^ method not found in `ThreadPool`

Now the error occurs because we don’t have an execute method on ThreadPool. Recall from “Creating a Finite Number of Threads” on page XX that we decided our thread pool should have an interface similar to thread::spawn. In addition, we’ll implement the execute function so that it takes the closure it’s given and gives it to an idle thread in the pool to run.

We’ll define the execute method on ThreadPool to take a closure as a parameter. Recall from “Moving Captured Values Out of Closures” on page XX that we can take closures as parameters with three different traits: Fn, FnMut, and FnOnce. We need to decide which kind of closure to use here. We know we’ll end up doing something similar to the standard library thread::spawn implementation, so we can look at what bounds the signature of thread::spawn has on its parameter. The documentation shows us the following:

pub fn spawn<F, T>(f: F) -> JoinHandle<T>

where

F: FnOnce() -> T,

F: Send + 'static,

T: Send + 'static,

The F type parameter is the one we’re concerned with here; the T type parameter is related to the return value, and we’re not concerned with that. We can see that spawn uses FnOnce as the trait bound on F. This is probably what we want as well, because we’ll eventually pass the argument we get in execute to spawn. We can be further confident that FnOnce is the trait we want to use because the thread for running a request will only execute that request’s closure one time, which matches the Once in FnOnce.

The F type parameter also has the trait bound Send and the lifetime bound 'static, which are useful in our situation: We need Send to transfer the closure from one thread to another and 'static because we don’t know how long the thread will take to execute. Let’s create an execute method on ThreadPool that will take a generic parameter of type F with these bounds:

src/lib.rs

impl ThreadPool {

--snip--

pub fn execute<F>(&self, f: F)

where

1 F: FnOnce() + Send + 'static,

{

}

}

We still use the () after FnOnce 1 because this FnOnce represents a closure that takes no parameters and returns the unit type (). Just like function definitions, the return type can be omitted from the signature, but even if we have no parameters, we still need the parentheses.

Again, this is the simplest implementation of the execute method: It does nothing, but we’re only trying to make our code compile. Let’s check it again:

$ cargo check

Checking hello v0.1.0 (file:///projects/hello)

Finished `dev` profile [unoptimized + debuginfo] target(s) in

0.24s

It compiles! But note that if you try cargo run and make a request in the browser, you’ll see the errors in the browser that we saw at the beginning of the chapter. Our library isn’t actually calling the closure passed to execute yet!

Note A saying you might hear about languages with strict compilers, such as Haskell and Rust, is “If the code compiles, it works.” But this saying is not universally true. Our project compiles, but it does absolutely nothing! If we were building a real, complete project, this would be a good time to start writing unit tests to check that the code compiles and has the behavior we want.

Consider: What would be different here if we were going to execute a future instead of a closure?

Validating the Number of Threads in new

We aren’t doing anything with the parameters to new and execute. Let’s implement the bodies of these functions with the behavior we want. To start, let’s think about new. Earlier we chose an unsigned type for the size parameter because a pool with a negative number of threads makes no sense. However, a pool with zero threads also makes no sense, yet zero is a perfectly valid usize. We’ll add code to check that size is greater than zero before we return a ThreadPool instance, and we’ll have the program panic if it receives a zero by using the assert! macro, as shown in Listing 21-13.

src/lib.rs

impl ThreadPool {

/// Create a new ThreadPool.

///

/// The size is the number of threads in the pool.

///

1 /// # Panics

///

/// The `new` function will panic if the size is zero.

pub fn new(size: usize) -> ThreadPool {

2 assert!(size > 0);

ThreadPool

}

--snip--

}

Implementing ThreadPool::new to panic if size is zero

We’ve also added some documentation for our ThreadPool with doc comments. Note that we followed good documentation practices by adding a section that calls out the situations in which our function can panic 1, as discussed in Chapter 14. Try running cargo doc --open and clicking the ThreadPool struct to see what the generated docs for new look like!

Instead of adding the assert! macro as we’ve done here 2, we could change new into build and return a Result like we did with Config::build in the I/O project in Listing 12-9. But we’ve decided in this case that trying to create a thread pool without any threads should be an unrecoverable error. If you’re feeling ambitious, try to write a function named build with the following signature to compare with the new function:

pub fn build(

size: usize

) -> Result<ThreadPool, PoolCreationError> {

Creating Space to Store the Threads

Now that we have a way to know we have a valid number of threads to store in the pool, we can create those threads and store them in the ThreadPool struct before returning the struct. But how do we “store” a thread? Let’s take another look at the thread::spawn signature:

pub fn spawn<F, T>(f: F) -> JoinHandle<T>

where

F: FnOnce() -> T,

F: Send + 'static,

T: Send + 'static,

The spawn function returns a JoinHandle<T>, where T is the type that the closure returns. Let’s try using JoinHandle too and see what happens. In our case, the closures we’re passing to the thread pool will handle the connection and not return anything, so T will be the unit type ().

The code in Listing 21-14 will compile, but it doesn’t create any threads yet. We’ve changed the definition of ThreadPool to hold a vector of thread::JoinHandle<()> instances, initialized the vector with a capacity of size, set up a for loop that will run some code to create the threads, and returned a ThreadPool instance containing them.

src/lib.rs

1 use std::thread;

pub struct ThreadPool {

2 threads: Vec<thread::JoinHandle<()>>,

}

impl ThreadPool {

--snip--

pub fn new(size: usize) -> ThreadPool {

assert!(size > 0);

3 let mut threads = Vec::with\_capacity(size);

for \_ in 0..size {

// create some threads and store them in the vector

}

ThreadPool { threads }

}

--snip--

}

Creating a vector for ThreadPool to hold the threads

We’ve brought std::thread into scope in the library crate 1 because we’re using thread::JoinHandle as the type of the items in the vector in ThreadPool 2.

Once a valid size is received, our ThreadPool creates a new vector that can hold size items 3. The with\_capacity function performs the same task as Vec::new but with an important difference: It pre-allocates space in the vector. Because we know we need to store size elements in the vector, doing this allocation up front is slightly more efficient than using Vec::new, which resizes itself as elements are inserted.

When you run cargo check again, it should succeed.

Sending Code from the ThreadPool to a Thread

We left a comment in the for loop in Listing 21-14 regarding the creation of threads. Here, we’ll look at how we actually create threads. The standard library provides thread::spawn as a way to create threads, and thread::spawn expects to get some code the thread should run as soon as the thread is created. However, in our case, we want to create the threads and have them wait for code that we’ll send later. The standard library’s implementation of threads doesn’t include any way to do that; we have to implement it manually.

We’ll implement this behavior by introducing a new data structure between the ThreadPool and the threads that will manage this new behavior. We’ll call this data structure Worker, which is a common term in pooling implementations. The Worker picks up code that needs to be run and runs the code in its thread.

Think of people working in the kitchen at a restaurant: The workers wait until orders come in from customers, and then they’re responsible for taking those orders and filling them.

Instead of storing a vector of JoinHandle<()> instances in the thread pool, we’ll store instances of the Worker struct. Each Worker will store a single JoinHandle<()> instance. Then, we’ll implement a method on Worker that will take a closure of code to run and send it to the already running thread for execution. We’ll also give each Worker an id so that we can distinguish between the different instances of Worker in the pool when logging or debugging.

Here is the new process that will happen when we create a ThreadPool. We’ll implement the code that sends the closure to the thread after we have Worker set up in this way:

1. Define a Worker struct that holds an id and a JoinHandle<()>.
2. Change ThreadPool to hold a vector of Worker instances.
3. Define a Worker::new function that takes an id number and returns a Worker instance that holds the id and a thread spawned with an empty closure.
4. In ThreadPool::new, use the for loop counter to generate an id, create a new Worker with that id, and store the Worker in the vector.

If you’re up for a challenge, try implementing these changes on your own before looking at the code in Listing 21-15.

Ready? Here is Listing 21-15 with one way to make the preceding modifications.

src/lib.rs

use std::thread;

pub struct ThreadPool {

1 workers: Vec<Worker>,

}

impl ThreadPool {

--snip--

pub fn new(size: usize) -> ThreadPool {

assert!(size > 0);

let mut workers = Vec::with\_capacity(size);

2 for id in 0..size {

3 workers.push(Worker::new(id));

}

ThreadPool { workers }

}

--snip--

}

4 struct Worker {

id: usize,

thread: thread::JoinHandle<()>,

}

impl Worker {

5 fn new(id: usize) -> Worker {

6 let thread = thread::spawn(|| {});

7 Worker { id, thread } 8

}

}

Modifying ThreadPool to hold Worker instances instead of holding threads directly

We’ve changed the name of the field on ThreadPool from threads to workers because it’s now holding Worker instances instead of JoinHandle<()> instances 1. We use the counter in the for loop 2 as an argument to Worker::new, and we store each new Worker in the vector named workers 3.

External code (like our server in src/main.rs) doesn’t need to know the implementation details regarding using a Worker struct within ThreadPool, so we make the Worker struct 4 and its new function 5 private. The Worker::new function uses the id we give it 7 and stores a JoinHandle<()> instance 8 that is created by spawning a new thread using an empty closure 6.

Note If the operating system can’t create a thread because there aren’t enough system resources, thread::spawn will panic. That will cause our whole server to panic, even though the creation of some threads might succeed. For simplicity’s sake, this behavior is fine, but in a production thread pool implementation, you’d likely want to use std::thread::Builder and its spawn method that returns Result instead.

This code will compile and will store the number of Worker instances we specified as an argument to ThreadPool::new. But we’re still not processing the closure that we get in execute. Let’s look at how to do that next.

Sending Requests to Threads via Channels

The next problem we’ll tackle is that the closures given to thread::spawn do absolutely nothing. Currently, we get the closure we want to execute in the execute method. But we need to give thread::spawn a closure to run when we create each Worker during the creation of the ThreadPool.

We want the Worker structs that we just created to fetch the code to run from a queue held in the ThreadPool and send that code to its thread to run.

The channels we learned about in Chapter 16—a simple way to communicate between two threads—would be perfect for this use case. We’ll use a channel to function as the queue of jobs, and execute will send a job from the ThreadPool to the Worker instances, which will send the job to its thread. Here is the plan:

1. The ThreadPool will create a channel and hold on to the sender.
2. Each Worker will hold on to the receiver.
3. We’ll create a new Job struct that will hold the closures we want to send down the channel.
4. The execute method will send the job it wants to execute through the sender.
5. In its thread, the Worker will loop over its receiver and execute the closures of any jobs it receives.

Let’s start by creating a channel in ThreadPool::new and holding the sender in the ThreadPool instance, as shown in Listing 21-16. The Job struct doesn’t hold anything for now but will be the type of item we’re sending down the channel.

src/lib.rs

use std::{sync::mpsc, thread};

pub struct ThreadPool {

workers: Vec<Worker>,

sender: mpsc::Sender<Job>,

}

struct Job;

impl ThreadPool {

--snip--

pub fn new(size: usize) -> ThreadPool {

assert!(size > 0);

1 let (sender, receiver) = mpsc::channel();

let mut workers = Vec::with\_capacity(size);

for id in 0..size {

workers.push(Worker::new(id));

}

ThreadPool { workers, sender } 2

}

--snip--

}

Modifying ThreadPool to store the sender of a channel that transmits Job instances

In ThreadPool::new, we create our new channel 1 and have the pool hold the sender 2. This will successfully compile.

Let’s try passing a receiver of the channel into each Worker as the thread pool creates the channel. We know we want to use the receiver in the thread that the Worker instances spawn, so we’ll reference the receiver parameter in the closure. The code in Listing 21-17 won’t quite compile yet.

src/lib.rs

impl ThreadPool {

--snip--

pub fn new(size: usize) -> ThreadPool {

assert!(size > 0);

let (sender, receiver) = mpsc::channel();

let mut workers = Vec::with\_capacity(size);

for id in 0..size {

1 workers.push(Worker::new(id, receiver));

}

ThreadPool { workers, sender }

}

--snip--

}

--snip--

impl Worker {

fn new(id: usize, receiver: mpsc::Receiver<Job>) -> Worker {

let thread = thread::spawn(|| {

2 receiver;

});

Worker { id, thread }

}

}

Passing the receiver to each Worker

We’ve made some small and straightforward changes: We pass the receiver into Worker::new 1, and then we use it inside the closure 2.

When we try to check this code, we get this error:

$ cargo check

Checking hello v0.1.0 (file:///projects/hello)

error[E0382]: use of moved value: `receiver`

--> src/lib.rs:26:42

|

21 | let (sender, receiver) = mpsc::channel();

| -------- move occurs because `receiver` has

type `std::sync::mpsc::Receiver<Job>`, which does not implement the

`Copy` trait

...

26 | workers.push(Worker::new(id, receiver));

| ^^^^^^^^ value moved

here, in previous iteration of loop

The code is trying to pass receiver to multiple Worker instances. This won’t work, as you’ll recall from Chapter 16: The channel implementation that Rust provides is multiple producer, single consumer. This means we can’t just clone the consuming end of the channel to fix this code. We also don’t want to send a message multiple times to multiple consumers; we want one list of messages with multiple Worker instances such that each message gets processed once.

Additionally, taking a job off the channel queue involves mutating the receiver, so the threads need a safe way to share and modify receiver; otherwise, we might get race conditions (as covered in Chapter 16).

Recall the thread-safe smart pointers discussed in Chapter 16: To share ownership across multiple threads and allow the threads to mutate the value, we need to use Arc<Mutex<T>>. The Arc type will let multiple Worker instances own the receiver, and Mutex will ensure that only one Worker gets a job from the receiver at a time. Listing 21-18 shows the changes we need to make.

src/lib.rs

use std::{

sync::{Arc, Mutex, mpsc},

thread,

};

--snip--

impl ThreadPool {

--snip--

pub fn new(size: usize) -> ThreadPool {

assert!(size > 0);

let (sender, receiver) = mpsc::channel();

1 let receiver = Arc::new(Mutex::new(receiver));

let mut workers = Vec::with\_capacity(size);

for id in 0..size {

workers.push(

Worker::new(id, Arc::clone(&receiver)) 2

);

}

ThreadPool { workers, sender }

}

--snip--

}

--snip--

impl Worker {

fn new(

id: usize,

receiver: Arc<Mutex<mpsc::Receiver<Job>>>,

) -> Worker {

--snip--

}

}

Sharing the receiver among the Worker instances using Arc and Mutex

In ThreadPool::new, we put the receiver in an Arc and a Mutex 1. For each new Worker, we clone the Arc to bump the reference count so that the Worker instances can share ownership of the receiver 2.

With these changes, the code compiles! We’re getting there!

Implementing the execute Method

Let’s finally implement the execute method on ThreadPool. We’ll also change Job from a struct to a type alias for a trait object that holds the type of closure that execute receives. As discussed in “Type Synonyms and Type Aliases” on page XX, type aliases allow us to make long types shorter for ease of use. Look at Listing 21-19.

src/lib.rs

--snip--

type Job = Box<dyn FnOnce() + Send + 'static>;

impl ThreadPool {

--snip--

pub fn execute<F>(&self, f: F)

where

F: FnOnce() + Send + 'static,

{

1 let job = Box::new(f);

2 self.sender.send(job).unwrap();

}

}

--snip--

Creating a Job type alias for a Box that holds each closure and then sending the job down the channel

After creating a new Job instance using the closure we get in execute 1, we send that job down the sending end of the channel 2. We’re calling unwrap on send for the case that sending fails. This might happen if, for example, we stop all our threads from executing, meaning the receiving end has stopped receiving new messages. At the moment, we can’t stop our threads from executing: Our threads continue executing as long as the pool exists. The reason we use unwrap is that we know the failure case won’t happen, but the compiler doesn’t know that.

But we’re not quite done yet! In the Worker, our closure being passed to thread::spawn still only references the receiving end of the channel. Instead, we need the closure to loop forever, asking the receiving end of the channel for a job and running the job when it gets one. Let’s make the change shown in Listing 21-20 to Worker::new.

src/lib.rs

--snip--

impl Worker {

fn new(

id: usize,

receiver: Arc<Mutex<mpsc::Receiver<Job>>>,

) -> Worker {

let thread = thread::spawn(move ||

loop {

let job = receiver

1 .lock()

2 .unwrap()

3 .recv()

4 .unwrap();

println!("Worker {id} got a job; executing.");

job();

}

);

Worker { id, thread }

}

}

Receiving and executing the jobs in the Worker instance’s thread

Here, we first call lock on the receiver to acquire the mutex 1, and then we call unwrap to panic on any errors 2. Acquiring a lock might fail if the mutex is in a poisoned state, which can happen if some other thread panicked while holding the lock rather than releasing the lock. In this situation, calling unwrap to have this thread panic is the correct action to take. Feel free to change this unwrap to an expect with an error message that is meaningful to you.

If we get the lock on the mutex, we call recv to receive a Job from the channel 3. A final unwrap moves past any errors here as well 4, which might occur if the thread holding the sender has shut down, similar to how the send method returns Err if the receiver shuts down.

The call to recv blocks, so if there is no job yet, the current thread will wait until a job becomes available. The Mutex<T> ensures that only one Worker thread at a time is trying to request a job.

Our thread pool is now in a working state! Give it a cargo run and make some requests:

$ cargo run

Compiling hello v0.1.0 (file:///projects/hello)

warning: field `workers` is never read

--> src/lib.rs:7:5

|

6 | pub struct ThreadPool {

| ---------- field in this struct

7 | workers: Vec<Worker>,

| ^^^^^^^

|

= note: `#[warn(dead\_code)]` on by default

warning: fields `id` and `thread` are never read

--> src/lib.rs:48:5

|

47 | struct Worker {

| ------ fields in this struct

48 | id: usize,

| ^^

49 | thread: thread::JoinHandle<()>,

| ^^^^^^

warning: `hello` (lib) generated 2 warnings

Finished `dev` profile [unoptimized + debuginfo] target(s) in

1.40s

Running `target/debug/hello`

Worker 0 got a job; executing.

Worker 2 got a job; executing.

Worker 1 got a job; executing.

Worker 3 got a job; executing.

Worker 0 got a job; executing.

Worker 2 got a job; executing.

Worker 1 got a job; executing.

Worker 3 got a job; executing.

Worker 0 got a job; executing.

Worker 2 got a job; executing.

Success! We now have a thread pool that executes connections asynchronously. There are never more than four threads created, so our system won’t get overloaded if the server receives a lot of requests. If we make a request to /sleep, the server will be able to serve other requests by having another thread run them.

Note If you open /sleep in multiple browser windows simultaneously, they might load one at a time in five-second intervals. Some web browsers execute multiple instances of the same request sequentially for caching reasons. This limitation is not caused by our web server.

This is a good time to pause and consider how the code in Listings 21-18, 21-19, and 21-20 would be different if we were using futures instead of a closure for the work to be done. What types would change? How would the method signatures be different, if at all? What parts of the code would stay the same?

After learning about the while let loop in Chapter 17 and Chapter 19, you might be wondering why we didn’t write the Worker thread code as shown in Listing 21-21.

src/lib.rs

--snip--

impl Worker {

fn new(

id: usize,

receiver: Arc<Mutex<mpsc::Receiver<Job>>>,

) -> Worker {

let thread = thread::spawn(move || {

while let Ok(job) = receiver.lock().unwrap().recv() {

println!("Worker {id} got a job; executing.");

job();

}

});

Worker { id, thread }

}

}

An alternative implementation of Worker::new using while let

This code compiles and runs but doesn’t result in the desired threading behavior: A slow request will still cause other requests to wait to be processed. The reason is somewhat subtle: The Mutex struct has no public unlock method because the ownership of the lock is based on the lifetime of the MutexGuard<T> within the LockResult<MutexGuard<T>> that the lock method returns. At compile time, the borrow checker can then enforce the rule that a resource guarded by a Mutex cannot be accessed unless we hold the lock. However, this implementation can also result in the lock being held longer than intended if we aren’t mindful of the lifetime of the MutexGuard<T>.

The code in Listing 21-20 that uses let job = receiver.lock().unwrap().recv().unwrap(); works because with let, any temporary values used in the expression on the right-hand side of the equal sign are immediately dropped when the let statement ends. However, while let (and if let and match) does not drop temporary values until the end of the associated block. In Listing 21-21, the lock remains held for the duration of the call to job(), meaning other Worker instances cannot receive jobs.

Graceful Shutdown and Cleanup

The code in Listing 21-20 is responding to requests asynchronously through the use of a thread pool, as we intended. We get some warnings about the workers, id, and thread fields that we’re not using in a direct way that reminds us we’re not cleaning up anything. When we use the less elegant ctrl-C method to halt the main thread, all other threads are stopped immediately as well, even if they’re in the middle of serving a request.

Next, then, we’ll implement the Drop trait to call join on each of the threads in the pool so that they can finish the requests they’re working on before closing. Then, we’ll implement a way to tell the threads they should stop accepting new requests and shut down. To see this code in action, we’ll modify our server to accept only two requests before gracefully shutting down its thread pool.

One thing to notice as we go: None of this affects the parts of the code that handle executing the closures, so everything here would be the same if we were using a thread pool for an async runtime.

Implementing the Drop Trait on ThreadPool

Let’s start with implementing Drop on our thread pool. When the pool is dropped, our threads should all join to make sure they finish their work. Listing 21-22 shows a first attempt at a Drop implementation; this code won’t quite work yet.

src/lib.rs

impl Drop for ThreadPool {

fn drop(&mut self) {

1 for worker in &mut self.workers {

2 println!("Shutting down worker {}", worker.id);

3 worker.thread.join().unwrap();

}

}

}

Joining each thread when the thread pool goes out of scope

First, we loop through each of the thread pool workers 1. We use &mut for this because self is a mutable reference, and we also need to be able to mutate worker. For each worker, we print a message saying that this particular Worker instance is shutting down 2, and then we call join on that Worker instance’s thread 3. If the call to join fails, we use unwrap to make Rust panic and go into an ungraceful shutdown.

Here is the error we get when we compile this code:

error[E0507]: cannot move out of `worker.thread` which is behind a

mutable reference

--> src/lib.rs:52:13

|

52 | worker.thread.join().unwrap();

| ^^^^^^^^^^^^^ ------ `worker.thread` moved due to

this method call

| |

| move occurs because `worker.thread` has type

`JoinHandle<()>`, which does not implement the `Copy` trait

|

note: `JoinHandle::<T>::join` takes ownership of the receiver `self`,

which moves `worker.thread`

The error tells us we can’t call join because we only have a mutable borrow of each worker and join takes ownership of its argument. To solve this issue, we need to move the thread out of the Worker instance that owns thread so that join can consume the thread. One way to do this is to take the same approach we took in Listing 18-15. If Worker held an Option<thread::JoinHandle<()>>, we could call the take method on the Option to move the value out of the Some variant and leave a None variant in its place. In other words, a Worker that is running would have a Some variant in thread, and when we wanted to clean up a Worker, we’d replace Some with None so that the Worker wouldn’t have a thread to run.

However, the only time this would come up would be when dropping the Worker. In exchange, we’d have to deal with an Option<thread::JoinHandle<()>> anywhere we accessed worker.thread. Idiomatic Rust uses Option quite a bit, but when you find yourself wrapping something you know will always be present in an Option as a workaround like this, it’s a good idea to look for alternative approaches to make your code cleaner and less error-prone.

In this case, a better alternative exists: the Vec::drain method. It accepts a range parameter to specify which items to remove from the vector and returns an iterator of those items. Passing the .. range syntax will remove every value from the vector.

So, we need to update the ThreadPool drop implementation like this:

src/lib.rs

impl Drop for ThreadPool {

fn drop(&mut self) {

for worker in self.workers.drain(..) {

println!("Shutting down worker {}", worker.id);

1 worker.thread.join().unwrap();

}

}

}

This resolves the compiler error and does not require any other changes to our code. Note that, because drop can be called when panicking, the unwrap 1 could also panic and cause a double panic, which immediately crashes the program and ends any cleanup in progress. This is fine for an example program, but it isn’t recommended for production code.

Signaling to the Threads to Stop Listening for Jobs

With all the changes we’ve made, our code compiles without any warnings. However, the bad news is that this code doesn’t function the way we want it to yet. The key is the logic in the closures run by the threads of the Worker instances: At the moment, we call join, but that won’t shut down the threads, because they loop forever looking for jobs. If we try to drop our ThreadPool with our current implementation of drop, the main thread will block forever, waiting for the first thread to finish.

To fix this problem, we’ll need a change in the ThreadPool drop implementation and then a change in the Worker loop.

First, we’ll change the ThreadPool drop implementation to explicitly drop the sender before waiting for the threads to finish. Listing 21-23 shows the changes to ThreadPool to explicitly drop sender. Unlike with the thread, here we do need to use an Option to be able to move sender out of ThreadPool with Option::take.

src/lib.rs

pub struct ThreadPool {

workers: Vec<Worker>,

sender: Option<mpsc::Sender<Job>>,

}

--snip--

impl ThreadPool {

pub fn new(size: usize) -> ThreadPool {

--snip--

ThreadPool {

workers,

sender: Some(sender),

}

}

pub fn execute<F>(&self, f: F)

where

F: FnOnce() + Send + 'static,

{

let job = Box::new(f);

self.sender

.as\_ref()

.unwrap()

.send(job)

.unwrap();

}

}

impl Drop for ThreadPool {

fn drop(&mut self) {

1 drop(self.sender.take());

for worker in self.workers.drain(..) {

println!("Shutting down worker {}", worker.id);

worker.thread.join().unwrap();

}

}

}

Explicitly dropping sender before joining the Worker threads

Dropping sender 1 closes the channel, which indicates no more messages will be sent. When that happens, all the calls to recv that the Worker instances do in the infinite loop will return an error. In Listing 21-24, we change the Worker loop to gracefully exit the loop in that case, which means the threads will finish when the ThreadPool drop implementation calls join on them.

src/lib.rs

impl Worker {

fn new(id: usize, receiver: Arc<Mutex<mpsc::Receiver<Job>>>) -> Worker {

let thread = thread::spawn(move || loop {

let message = receiver.lock().unwrap().recv();

match message {

Ok(job) => {

println!("Worker {id} got a job; executing.");

job();

}

Err(\_) => {

println!("Worker {id} disconnected; shutting down.");

break;

}

}

});

Worker { id, thread }

}

}

Explicitly breaking out of the loop when recv returns an error

To see this code in action, let’s modify main to accept only two requests before gracefully shutting down the server, as shown in Listing 21-25.

src/main.rs

fn main() {

let listener = TcpListener::bind("127.0.0.1:7878").unwrap();

let pool = ThreadPool::new(4);

for stream in listener.incoming().take(2) {

let stream = stream.unwrap();

pool.execute(|| {

handle\_connection(stream);

});

}

println!("Shutting down.");

}

Shutting down the server after serving two requests by exiting the loop

You wouldn’t want a real-world web server to shut down after serving only two requests. This code just demonstrates that the graceful shutdown and cleanup is in working order.

The take method is defined in the Iterator trait and limits the iteration to the first two items at most. The ThreadPool will go out of scope at the end of main, and the drop implementation will run.

Start the server with cargo run and make three requests. The third request should error, and in your terminal, you should see output similar to this:

$ cargo run

Compiling hello v0.1.0 (file:///projects/hello)

Finished `dev` profile [unoptimized + debuginfo] target(s) in

1.0s

Running `target/debug/hello`

Worker 0 got a job; executing.

Shutting down.

Shutting down worker 0

Worker 3 got a job; executing.

Worker 1 disconnected; shutting down.

Worker 2 disconnected; shutting down.

Worker 3 disconnected; shutting down.

Worker 0 disconnected; shutting down.

Shutting down worker 1

Shutting down worker 2

Shutting down worker 3

You might see a different ordering of Worker IDs and messages printed. We can see how this code works from the messages: Worker instances 0 and 3 got the first two requests. The server stopped accepting connections after the second connection, and the Drop implementation on ThreadPool starts executing before Worker 3 even starts its job. Dropping the sender disconnects all the Worker instances and tells them to shut down. The Worker instances each print a message when they disconnect, and then the thread pool calls join to wait for each Worker thread to finish.

Notice one interesting aspect of this particular execution: The ThreadPool dropped the sender, and before any Worker received an error, we tried to join Worker 0. Worker 0 had not yet gotten an error from recv, so the main thread blocked, waiting for Worker 0 to finish. In the meantime, Worker 3 received a job and then all threads received an error. When Worker 0 finished, the main thread waited for the rest of the Worker instances to finish. At that point, they had all exited their loops and stopped.

Congrats! We’ve now completed our project; we have a basic web server that uses a thread pool to respond asynchronously. We’re able to perform a graceful shutdown of the server, which cleans up all the threads in the pool. See https://nostarch.com/rust-programming-language-2nd-edition to download the full code for this chapter for reference.

We could do more here! If you want to continue enhancing this project, here are some ideas:

* Add more documentation to ThreadPool and its public methods.
* Add tests of the library’s functionality.
* Change calls to unwrap to more robust error handling.
* Use ThreadPool to perform some task other than serving web requests.
* Find a thread pool crate on https://crates.io and implement a similar web server using the crate instead. Then, compare its API and robustness to the thread pool we implemented.

Summary

Well done! You’ve made it to the end of the book! We want to thank you for joining us on this tour of Rust. You’re now ready to implement your own Rust projects and help with other people’s projects. Keep in mind that there is a welcoming community of other Rustaceans who would love to help you with any challenges you encounter on your Rust journey.