[Building a Single Threaded Web Server 2](#_Toc509918788)

[Listening to the TCP Connection 2](#_Toc509918789)

[Reading the Request 5](#_Toc509918790)

[A Closer Look at an HTTP Request 7](#_Toc509918791)

[Writing a Response 8](#_Toc509918792)

[Returning Real HTML 9](#_Toc509918793)

[Validating the Request and Selectively Responding 11](#_Toc509918794)

[A Touch of Refactoring 13](#_Toc509918795)

[Turning our Single Threaded Server into a Multithreaded Server 14](#_Toc509918796)

[Simulating a Slow Request in the Current Server Implementation 15](#_Toc509918797)

[Improving Throughput with a Thread Pool 16](#_Toc509918798)

[Code Structure if We Could Spawn a Thread for Each Request 17](#_Toc509918799)

[Creating a Similar Interface for a Finite Number of Threads 18](#_Toc509918800)

[Building the ThreadPool Struct Using Compiler Driven Development 18](#_Toc509918801)

[Validating the Number of Threads in new 23](#_Toc509918802)

[Creating Space to Store the Threads 24](#_Toc509918803)

[A Worker Struct Responsible for Sending Code from the ThreadPool to a Thread 25](#_Toc509918804)

[Sending Requests to Threads Via Channels 28](#_Toc509918805)

[Implementing the execute Method 33](#_Toc509918806)

[Graceful Shutdown and Cleanup 40](#_Toc509918807)

[Implementing the Drop Trait on ThreadPool 40](#_Toc509918808)

[Signaling to the Threads to Stop Listening for Jobs 43](#_Toc509918809)

[Summary 48](#_Toc509918810)

20

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 only says “hello” and looks like Figure 20-1 in a web browser.

insert figure

Figure 20-1: Our final shared project

Here is the plan to build the web server:

Learn a bit about TCP and HTTP

Listen for TCP connections on a socket

Parse a small number of HTTP requests

Create a proper HTTP response

Improve the throughput of our server with a thread pool

But before we get started, we should mention one detail: the method we’ll use won’t be the best way to build a web server with Rust. A number of production-ready crates are available on <https://crates.io>/ that provide more complete web server and thread pool implementations than 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’re 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. We’ll write the basic HTTP server and thread pool manually so 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 the Hypertext Transfer Protocol (HTTP) and the 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 --bin

Created binary (application) `hello` project

$ cd hello

Now enter the code in Listing 20-1 in src/main.rs to start. This code will listen at the 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() {

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

for stream in listener.incoming() {

let stream = stream.unwrap();

println!("Connection established!");

}

}

Listing 20-1: 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. 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 is normally accepted on this port, and 7878 is “rust” typed on a telephone. Note that connecting to port 80 requires administrator privileges; nonadministrators can only listen on ports higher than 1024.

The bind function in this scenario works like the new function in that it will return a new TcpListener instance. The reason the function is called bind is that 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 binding might fail. For example, if we tried to connect to port 80 without being an administrator or if we ran two instances of our program and so had two programs listening to the same port, binding wouldn’t work. 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 (more specifically, streams of type TcpStream). A single stream represents an open connection between the client and the server. A 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, TcpStream will read from itself to see what the client sent, and then allow us to write our response to the stream. 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; if there aren’t any errors, the program prints a message. 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, 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. 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 cargo run 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 we can see the data being sent from the browser. Change the code to look like Listing 20-2.

src/main.rs

use std::io::prelude::\*;

use std::net::TcpListener;

use std::net::TcpStream;

fn main() {

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

for stream in listener.incoming() {

let stream = stream.unwrap();

handle\_connection(stream);

}

}

fn handle\_connection(mut stream: TcpStream) {

let mut buffer = [0; 512];

stream.read(&mut buffer).unwrap();

println!("Request: {}", String::from\_utf8\_lossy(&buffer[..]));

}

Listing 20-2: Reading from the TcpStream and printing the data

We bring std::io::prelude into scope to get access to certain traits that let us read from and write to the stream. 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.

In the handle\_connection function, we’ve made the stream parameter mutable. The reason is that the TcpStream instance keeps track of what data it returns to us internally. It might read more data than we asked for and save that data for the next time we ask for data. It therefore needs to be mut because its internal state might change; usually, we think of “reading” as not needing mutation, but in this case we need the mut keyword.

Next, we need to actually read from the stream. We do this in two steps: first, we declare a buffer on the stack to hold the data that is read in. We’ve made the buffer 512 bytes in size, which is big enough to hold the data of a basic request and sufficient for our purposes in this chapter. If we wanted to handle requests of an arbitrary size, buffer management would need to be more complicated; we’ll keep it simple for now. We pass the buffer to stream.read, which will read bytes from the TcpStream and put them in the buffer.

Second, we convert the bytes in the buffer to a string and print that string. The String::from\_utf8\_lossy function takes a &[u8] and produces a String from it. The “lossy” part of the name indicates the behavior of this function when it sees an invalid UTF-8 sequence: it will replace the invalid sequence with �, the U+FFFD REPLACEMENT CHARACTER. You might see replacement characters for characters in the buffer that aren’t filled by request data.

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 [unoptimized + debuginfo] target(s) in 0.42 secs

Running `target/debug/hello`

Request: GET / HTTP/1.1

Host: 127.0.0.1:7878

User-Agent: Mozilla/5.0 (Windows NT 10.0; WOW64; rv:52.0) Gecko/20100101

Firefox/52.0

Accept: text/html,application/xhtml+xml,application/xml;q=0.9,\*/\*;q=0.8

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

Accept-Encoding: gzip, deflate

Connection: keep-alive

Upgrade-Insecure-Requests: 1

������������������������������������

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 Request: GET. 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.

A Closer Look 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.

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. The CRLF sequence can also be written as \r\n: the \r part is a carriage return and \n is a line feed. (These terms come from the typewriter days!) 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.

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

Now we’ll 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, 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 20-3.

src/main.rs

fn handle\_connection(mut stream: TcpStream) {

let mut buffer = [0; 512];

stream.read(&mut buffer).unwrap();

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

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

stream.flush().unwrap();

}

Listing 20-3: Writing a tiny successful HTTP response to the stream

The first new line defines the response variable that holds the success message’s data. Then we call as\_bytes on our response to convert the string data to bytes. The write method on stream takes a &[u8] and sends those bytes directly down the connection.

Because the write operation could fail, we use unwrap on any error result as before. Again, in a real application you would add error-handling here. Finally, flush will wait and prevent the program from continuing until all the bytes are written to the connection; TcpStream contains an internal buffer to minimize calls to the underlying operating system.

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 hand-coded an HTTP request and response!

Returning Real HTML

Let’s implement the functionality for returning more than a blank page. Create a new file, hello.html, in the root of your project directory, not in the src directory. You can input in any HTML you want; Listing 20-4 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>

Listing 20-4: 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 20-5 to read the HTML file, add it to the response as a body, and send it.

src/main.rs

use std::fs::File;

// --snip--

fn handle\_connection(mut stream: TcpStream) {

let mut buffer = [0; 512];

stream.read(&mut buffer).unwrap();

let mut file = File::open("hello.html").unwrap();

let mut contents = String::new();

file.read\_to\_string(&mut contents).unwrap();

let response = format!("HTTP/1.1 200 OK\r\n\r\n{}", contents);

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

stream.flush().unwrap();

}

Listing 20-5: Sending the contents of hello.html as the body of the response

We’ve added a line at the top to bring the standard library’s File into scope. The code for opening files and reading code should look familiar; we used it in Chapter 12 when we read the contents of a file for our I/O project in Listing 12-4 on page XX.

prod: confirm/link xref

Next, we use format! to add the file’s contents as the body of the success response.

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

Currently, we’re ignoring the request data in buffer 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. Our server is very limited and is not 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 return an error if the browser requests anything else. For this we need to modify handle\_connection as shown in Listing 20-6. 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 mut buffer = [0; 512];

stream.read(&mut buffer).unwrap();

let get = b"GET / HTTP/1.1\r\n";

if buffer.starts\_with(get) {

let mut file = File::open("hello.html").unwrap();

let mut contents = String::new();

file.read\_to\_string(&mut contents).unwrap();

let response = format!("HTTP/1.1 200 OK\r\n\r\n{}", contents);

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

stream.flush().unwrap();

} else {

// some other request

}

}

Listing 20-6: Matching the request and handling requests to / differently than other requests

First, we hardcode the data corresponding to the / request into the get variable. Because we’re reading raw bytes into the buffer, we transform get into a byte string by adding the b"" byte string syntax at the start of the content data. Then we check if buffer starts with the bytes in get. If it does, it means we’ve received a well-formed request to /, which is the success case we’ll handle in the if block that returns the contents of our HTML file.

If buffer does not start with the bytes in get, it means we’ve received some other request. We’ll add code to the else block 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 you saw when running the code in Listing 20-1 and Listing 20-2.

Now let’s add the code in Listing 20-7 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 as such to the end user.

src/main.rs

// --snip--

} else {

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

let mut file = File::open("404.html").unwrap();

let mut contents = String::new();

file.read\_to\_string(&mut contents).unwrap();

let response = format!("{}{}", status\_line, contents);

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

stream.flush().unwrap();

}

Listing 20-7: 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. We’re still not returning headers, and the body of the response will be the HTML in the file 404.html. 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 20-8.

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>

Listing 20-8: 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.

A Touch of 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 20-9 shows the resulting 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 buffer.starts\_with(get) {

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

} else {

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

};

let mut file = File::open(filename).unwrap();

let mut contents = String::new();

file.read\_to\_string(&mut contents).unwrap();

let response = format!("{}{}", status\_line, contents);

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

stream.flush().unwrap();

}

Listing 20-9: 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 18.

prod: confirm xref

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 means we have only one place to update the code if we want to change how the file reading and response writing works. The behavior of the code in Listing 20-9 will be the same as that in Listing 20-8.

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, and then fix it so our server can handle multiple requests at once.

Turning Our Single Threaded Server into a Multithreaded Server

Right now, the server will process each request in turn, meaning it won’t process a second connection until the first 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 in the Current Server Implementation

We’ll look at how a slow-processing request can affect other requests made to our current server implementation. Listing 20-10 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::thread;

use std::time::Duration;

// --snip--

fn handle\_connection(mut stream: TcpStream) {

// --snip--

let get = b"GET / HTTP/1.1\r\n";

let sleep = b"GET /sleep HTTP/1.1\r\n";

let (status\_line, filename) = if buffer.starts\_with(get) {

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

} else if buffer.starts\_with(sleep) {

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

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

} else {

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

};

// --snip--

}

Listing 20-10: Simulating a slow request by recognizing /sleep and sleeping for five seconds

This code is a bit messy, but it’s good enough for simulation purposes. We created a second request sleep, whose data our server recognizes. We added an else if after the if block to check for the request to /sleep. When that request is received, the server will sleep for five seconds before rendering the successful HTML page.

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, and then open two browser windows: one for http://localhost:7878/ and the other for http://localhost: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 ways we could change how our web server works to avoid having all requests back up behind a slow request; 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 waiting and ready 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 will allow us to process connections concurrently, increasing the throughput of our server.

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

Rather than spawning unlimited threads, we’ll have a fixed number of threads waiting in the pool. As requests come in, they’ll be 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 of this queue, handle the request, and then ask the queue for another request. With this design, we can process 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 that point.

This technique is just one of many ways to improve the throughput of our web server. Other options you might explore are the fork/join model and the single threaded async I/O model. If you’re interested in this topic, you can read more about other solutions and try to implement them in Rust; 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 it’s structured in the way you want to call it, and 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.

prod: confirm xref

Code Structure If We Could Spawn 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. Listing 20-11 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);

});

}

}

Listing 20-11: 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. But as we mentioned, this will eventually overwhelm the system because we’re making new threads without any limit.

prod: Confirm xref

Creating a Similar Interface for a Finite Number of Threads

We want our thread pool to work in a similar, familiar way so switching from threads to a thread pool doesn’t require large changes to the code that uses our API. Listing 20-12 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();

let pool = ThreadPool::new(4);

for stream in listener.incoming() {

let stream = stream.unwrap();

pool.execute(|| {

handle\_connection(stream);

});

}

}

Listing 20-12: Our ideal ThreadPool interface

We use ThreadPool::new to create a new thread pool with a configurable number of threads, in this case four. Then, in the for loop, pool.execute has a similar interface as thread::spawn in that it takes a closure of the code the pool should run for each stream. We need to implement pool.execute so 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 the compiler can guide us in how to fix it.

Building the ThreadPool Struct Using Compiler Driven Development

Make the changes in Listing 20-12 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

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

error[E0433]: failed to resolve. Use of undeclared type or module `ThreadPool`

--> src\main.rs:10:16

|

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

| ^^^^^^^^^^^^^^^ Use of undeclared type or module

`ThreadPool`

error: aborting due to previous error

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. We could also use the separate thread pool library for whatever work we want to do, not just for serving web requests.

Create a src/lib.rs 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 create a new directory, src/bin, and move the binary crate rooted in src/main.rs into src/bin/main.rs. Doing so will make the library crate the primary crate in the hello directory; we can still run the binary in src/bin/main.rs using cargo run. After moving the main.rs file, edit it to bring the library crate in and bring ThreadPool into scope by adding the following code to the top of src/bin/main.rs:

src/bin/main.rs

extern crate hello;

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

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

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

`hello::ThreadPool` in the current scope

--> src/bin/main.rs:13:16

|

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

| ^^^^^^^^^^^^^^^ function or associated item not found in

`hello::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 .

prod: confirm/link xref (ch 3)

Let’s check the code again:

$ cargo check

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

warning: unused variable: `size`

--> src/lib.rs:4:16

|

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

| ^^^^

|

= note: #[warn(unused\_variables)] on by default

= note: to avoid this warning, consider using `\_size` instead

error[E0599]: no method named `execute` found for type `hello::ThreadPool` in the current scope

--> src/bin/main.rs:18:14

|

18 | pool.execute(|| {

| ^^^^^^^

Now we get a warning and an error. Ignoring the warning for a moment, the error occurs because we don’t have an execute method on ThreadPool. Recall from “Creating a Similar Interface for 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 it takes the closure it’s given and gives it to an idle thread in the pool to run.

prod: link xref

We’ll define the execute method on ThreadPool to take a closure as a parameter. Recall from “Storing Closures Using Generic Parameters and the Fn Traits” on pageXX 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:

prod: confirm/link xref (ch 13)

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

where

F: FnOnce() -> T + Send + 'static,

T: Send + 'static

The F parameter is the one we’re concerned with here; the T 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 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 F with these bounds:

src/lib.rs

impl ThreadPool {

// --snip--

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

where

F: FnOnce() + Send + 'static

{

}

}

We still use the () after FnOnce because this FnOnce represents a closure that takes no parameters and doesn’t return a value. 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, we’ll add the simplest implementation of the execute method, which does nothing, just to make our code compile. Let’s check it again:

$ cargo check

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

warning: unused variable: `size`

--> src/lib.rs:4:16

|

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

| ^^^^

|

= note: #[warn(unused\_variables)] on by default

= note: to avoid this warning, consider using `\_size` instead

warning: unused variable: `f`

--> src/lib.rs:8:30

|

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

| ^

|

= note: to avoid this warning, consider using `\_f` instead

We’re receiving only warnings now, which means 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 built 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.

Validating the Number of Threads in new

We’ll continue to get warnings because 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 have the program panic if it receives a zero by using the assert! macro, as shown in Listing 20-13.

src/lib.rs

impl ThreadPool {

/// Create a new ThreadPool.

///

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

///

/// # Panics

///

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

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

assert!(size > 0);

ThreadPool

}

// --snip--

}

Listing 20-13: Implementing ThreadPool::new to panic if size is zero

We’ve 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, as discussed in Chapter 14. Try running cargo doc --open and clicking the ThreadPool struct to see what the generate docs for new look like!

prod: confirm xref

Instead of adding the assert! macro as we’ve done here, we could make new return a Result like we did with Config::new in the I/O project in Listing 12-9 on page XX. 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 version of new with the following signature to compare both versions:

prod: link xref

fn new(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 it. 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 + 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 20-14 will compile but 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

use std::thread;

pub struct ThreadPool {

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

}

impl ThreadPool {

// --snip--

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

assert!(size > 0);

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

for \_ in 0..size {

// create some threads and store them in the vector

}

ThreadPool {

threads

}

}

// --snip--

}

Listing 20-14: Creating a vector for ThreadPool to hold the threads

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

Once a valid size is received, our ThreadPool creates a new vector that can hold size items. We haven’t used the with\_capacity function in this book yet, which performs the same task as Vec::new but with an important difference: it preallocates 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, you’ll get a few more warnings, but it should succeed.

A Worker Struct Responsible for Sending Code from the ThreadPool to a Thread

We left a comment in the for loop in Listing 20-14 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 start up 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. 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 we can distinguish between the different workers in the pool when logging or debugging.

First, let’s make these changes to what happens 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:

Define a Worker struct that holds an id and a JoinHandle<()>.

Change ThreadPool to hold a vector of Worker instances.

Define a Worker::new function that takes an id number and returns a Worker instance that holds the allocated id and a thread spawned with an empty closure.

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 20-15.

Ready? Here is Listing 20-15 with one way to make the preceding modifications.

src/lib.rs

use std::thread;

pub struct ThreadPool {

workers: Vec<Worker>,

}

impl ThreadPool {

// --snip--

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

assert!(size > 0);

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

for id in 0..size {

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

}

ThreadPool {

workers

}

}

// --snip--

}

struct Worker {

id: usize,

thread: thread::JoinHandle<()>,

}

impl Worker {

fn new(id: usize) -> Worker {

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

Worker {

id,

thread,

}

}

}

Listing 20-15: 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. We use the counter in the for loop as an argument to Worker::new, and we store each new Worker in the vector named workers.

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

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

Now we’ll tackle the problem 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 code to run from a queue held in the ThreadPool and send that code to its thread to run.

In Chapter 16, you learned about channels—a simple way to communicate between two threads—that 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:

prod: confirm xref

The ThreadPool will create a channel and hold on to the sending side of the channel.

Each Worker will hold on to the receiving side of the channel.

We’ll create a new Job struct that will hold the closures we want to send down the channel.

The execute method will send the job it wants to execute down the sending side of the channel.

In its thread, the Worker will loop over its receiving side of the channel and execute the closures of any jobs it receives.

Let’s start by creating a channel in ThreadPool::new and holding the sending side in the ThreadPool instance, as shown in Listing 20-16. 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

// --snip--

use std::sync::mpsc;

pub struct ThreadPool {

workers: Vec<Worker>,

sender: mpsc::Sender<Job>,

}

struct Job;

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 {

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

}

ThreadPool {

workers,

sender,

}

}

// --snip--

}

Listing 20-16: Modifying ThreadPool to store the sending end of a channel that sends Job instances

In ThreadPool::new, we create our new channel and have the pool hold the sending end. This will successfully compile, still with warnings.

Let’s try passing a receiving end of the channel into each worker as the thread pool creates them. We know we want to use the receiving end in the thread that the workers spawn, so we’ll reference the receiver parameter in the closure. The code in Listing 20-17 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 {

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(|| {

receiver;

});

Worker {

id,

thread,

}

}

}

Listing 20-17: Passing the receiving end of the channel to the workers

We’ve made some small and straightforward changes: we pass the receiving end of the channel into Worker::new, and then we use it inside the closure.

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

$ cargo check

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

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

--> src/lib.rs:27:42

|

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

| ^^^^^^^^ value moved here in

previous iteration of loop

|

= note: move occurs because `receiver` has type

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

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. Even if we could, that is not the technique we would want to use; instead, we want to distribute the jobs across threads by sharing the single receiver between all the workers.

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 workers own the receiver, and Mutex will ensure that only one worker gets a job from the receiver at a time. Listing 20-18 shows the changes we need to make.

prod: confirm xrefs

src/lib.rs

use std::sync::Arc;

use std::sync::Mutex;

// --snip--

impl ThreadPool {

// --snip--

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

assert!(size > 0);

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

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)));

}

ThreadPool {

workers,

sender,

}

}

// --snip--

}

impl Worker {

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

// --snip--

}

}

Listing 20-18: Sharing the receiving end of the channel between the workers using Arc and Mutex

In ThreadPool::new, we put the receiving end of the channel in an Arc and a Mutex. For each new worker, we clone the Arc to bump the reference count so the workers can share ownership of the receiving end.

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 Aliases Create Type Synonyms” on page XX in Chapter 19, type aliases allow us to make long types shorter. Look at Listing 20-19.

prod: confirm/link xref (ch 19)

src/lib.rs

// --snip--

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

impl ThreadPool {

// --snip--

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

where

F: FnOnce() + Send + 'static

{

let job = Box::new(f);

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

}

}

// --snip--

Listing 20-19: 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, we send that job down the sending end of the channel. We’re calling unwrap on send for the case that sending fails, which 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 20-20 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.lock().unwrap().recv().unwrap();

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

(\*job)();

}

});

Worker {

id,

thread,

}

}

}

Listing 20-20: Receiving and executing the jobs in the worker’s thread

Here, we first call lock on the receiver to acquire the mutex, and then call unwrap to panic on any errors. 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. A final unwrap moves past any errors here as well, which might occur if the thread holding the sending side of the channel has shut down, similar to how the send method returns Err if the receiving side shuts down.

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

Theoretically, this code should compile. Unfortunately, the Rust compiler isn’t perfect yet, and we get this error:

error[E0161]: cannot move a value of type std::ops::FnOnce() +

std::marker::Send: the size of std::ops::FnOnce() + std::marker::Send cannot be

statically determined

--> src/lib.rs:63:17

|

63 | (\*job)();

| ^^^^^^

This error is fairly cryptic because the problem is fairly cryptic. To call a FnOnce closure that is stored in a Box<T> (which is what our Job type alias is), the closure needs to move itself out of the Box<T> because the closure takes ownership of self when we call it. In general, Rust doesn’t allow us to move a value out of a Box<T> because Rust doesn’t know how big the value inside the Box<T> will be: recall in Chapter 15 that we used Box<T> precisely because we had something of an unknown size that we wanted to store in a Box<T> to get a value of a known size.

As you saw in Listing 17-15 on page XX, we can write methods that use the syntax self: Box<Self>, which allows the method to take ownership of a Self value stored in a Box<T>. That’s exactly what we want to do here, but unfortunately Rust won’t let us: the part of Rust that implements behavior when a closure is called isn’t implemented using self: Box<Self>. So Rust doesn’t yet understand that it could use self: Box<Self> in this situation to take ownership of the closure and move the closure out of the Box<T>.

prod: confirm/link xrefs

Rust is still a work in progress with places where the compiler could be improved, but in the future, the code in Listing 20-20 should work just fine. People just like you are working to fix this and other issues! After you’ve finished this book, we would love for you to join in.

But for now, let’s work around this problem using a handy trick. We can tell Rust explicitly that in this case we can take ownership of the value inside the Box<T> using self: Box<Self>; then, once we have ownership of the closure, we can call it. This involves defining a new trait FnBox with the method call\_box that will use self: Box<Self> in its signature, defining FnBox for any type that implements FnOnce(), changing our type alias to use the new trait, and changing Worker to use the call\_box method. These changes are shown in Listing 20-21.

src/lib.rs

trait FnBox {

fn call\_box(self: Box<Self>);

}

impl<F: FnOnce()> FnBox for F {

fn call\_box(self: Box<F>) {

(\*self)()

}

}

type Job = Box<FnBox + Send + 'static>;

// --snip--

impl Worker {

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

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

loop {

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

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

job.call\_box();

}

});

Worker {

id,

thread,

}

}

}

Listing 20-21: Adding a new trait FnBox to work around the current limitations of Box<FnOnce()>

First, we create a new trait named FnBox. This trait has the one method call\_box, which is similar to the call methods on the other Fn\* traits except that it takes self: Box<Self> to take ownership of self and move the value out of the Box<T>.

Next, we implement the FnBox trait for any type F that implements the FnOnce() trait. Effectively, this means that any FnOnce() closures can use our call\_box method. The implementation of call\_box uses (\*self)() to move the closure out of the Box<T> and call the closure.

We now need our Job type alias to be a Box of anything that implements our new trait FnBox. This will allow us to use call\_box in Worker when we get a Job value. Implementing the FnBox trait for any FnOnce() closure means we don’t have to change anything about the actual values we’re sending down the channel.

Finally, in the closure run in the thread in Worker::new, we use call\_box instead of invoking the closure directly. Now Rust is able to recognize that what we want to do is fine.

This trick is very sneaky and complicated. Don’t worry if it doesn’t make perfect sense; someday, it will be completely unnecessary.

By implementing this trick, our thread pool is 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 is never used: `workers`

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

|

7 | workers: Vec<Worker>,

| ^^^^^^^^^^^^^^^^^^^^

|

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

warning: field is never used: `id`

--> src/lib.rs:61:5

|

61 | id: usize,

| ^^^^^^^^^

|

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

warning: field is never used: `thread`

--> src/lib.rs:62:5

|

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

| ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^

|

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

Finished dev [unoptimized + debuginfo] target(s) in 0.99 secs

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.

After learning about the while let loop in Chapter 18, you might be wondering why we didn’t write the worker thread as shown in Listing 20-22.

prod: confirm xref

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 {} got a job; executing.", id);

job.call\_box();

}

});

Worker {

id,

thread,

}

}

}

Listing 20-22: 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. But this method can also result in the lock being held longer than intended if we don’t think carefully about the lifetime of the MutexGuard<T>. Because the values in the while expression remain in scope for the duration of the block, the lock remains held for the duration of the call to job.call\_box(), meaning other workers cannot receive jobs.

By using loop instead and acquiring the lock and a job within the block rather than outside it, the MutexGuard returned from the lock method is dropped as soon as the let job statement ends. This ensures that the lock is held during the call to recv, but it is released before the call to job.call\_box(), allowing multiple requests to be serviced concurrently.

Graceful Shutdown and Cleanup

The code in Listing 20-21 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.

Now we’ll implement the Drop trait to call join on each of the threads in the pool so 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 only accept two requests before gracefully shutting down its thread pool.

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 on to make sure they finish their work. Listing 20-23 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) {

for worker in &mut self.workers {

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

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

}

}

}

Listing 20-23: Joining each thread when the thread pool goes out of scope

First, we loop through each of the thread pool workers. 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 is shutting down, and then we call join on that worker’s thread. 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 borrowed content

--> src/lib.rs:65:13

|

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

| ^^^^^^ cannot move out of borrowed content

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 join can consume the thread. We did this in Listing 17-15: if Worker holds an Option<thread::JoinHandle<()> instead, we can 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 will have a Some variant in thread, and when we want to clean up a worker, we’ll replace Some with None so the worker doesn’t have a thread to run.

So we know we want to update the definition of Worker like this:

src/lib.rs

struct Worker {

id: usize,

thread: Option<thread::JoinHandle<()>>,

}

Now let’s lean on the compiler to find the other places that need to change. Checking this code, we get two errors:

error[E0599]: no method named `join` found for type

`std::option::Option<std::thread::JoinHandle<()>>` in the current scope

--> src/lib.rs:65:27

|

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

| ^^^^

error[E0308]: mismatched types

--> src/lib.rs:89:13

|

89 | thread,

| ^^^^^^

| |

| expected enum `std::option::Option`, found struct

`std::thread::JoinHandle`

| help: try using a variant of the expected type: `Some(thread)`

|

= note: expected type `std::option::Option<std::thread::JoinHandle<()>>`

found type `std::thread::JoinHandle<\_>`

Let’s address the second error, which points to the code at the end of Worker::new; we need to wrap the thread value in Some when we create a new Worker. Make the following changes to fix this error:

src/lib.rs

impl Worker {

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

// --snip--

Worker {

id,

thread: Some(thread),

}

}

}

The first error is in our Drop implementation. We mentioned earlier that we intended to call take on the Option value to move thread out of worker. The following changes will do so:

Filename: src/lib.rs

impl Drop for ThreadPool {

fn drop(&mut self) {

for worker in &mut self.workers {

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

if let Some(thread) = worker.thread.take() {

thread.join().unwrap();

}

}

}

}

As discussed in Chapter 17, the take method on Option takes the Some variant out and leaves None in its place. We’re using if let to destructure the Some and get the thread; then we call join on the thread. If a worker’s thread is already None, we know that worker has already had its thread cleaned up, so nothing happens in that case.

prod: confirm xref

Signaling to the Threads to Stop Listening for Jobs

With all the changes we’ve made, our code compiles without any warnings. But the bad news is 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 this implementation, the main thread will block forever waiting for the first thread to finish.

To fix this problem, we’ll modify the threads so they listen for either a Job to run or a signal that they should stop listening and exit the infinite loop. Instead of Job instances, our channel will send one of these two enum variants:

src/lib.rs

enum Message {

NewJob(Job),

Terminate,

}

This Message enum will either be a NewJob variant that holds the Job the thread should run, or it will be a Terminate variant that will cause the thread to exit its loop and stop.

We need to adjust the channel to use values of type Message rather than type Job, as shown in Listing 20-24.

src/lib.rs

pub struct ThreadPool {

workers: Vec<Worker>,

sender: mpsc::Sender<Message>,

}

// --snip--

impl ThreadPool {

// --snip--

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

where

F: FnOnce() + Send + 'static

{

let job = Box::new(f);

self.sender.send(Message::NewJob(job)).unwrap();

}

}

// --snip--

impl Worker {

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

Worker {

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

loop {

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

match message {

Message::NewJob(job) => {

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

job.call\_box();

},

Message::Terminate => {

println!("Worker {} was told to terminate.", id);

break;

},

}

}

});

Worker {

id,

thread: Some(thread),

}

}

}

Listing 20-24: Sending and receiving Message values and exiting the loop if a Worker receives Message::Terminate

To incorporate the Message enum, we need to change Job to Message in two places: the definition of ThreadPool and the signature of Worker::new. The execute method of ThreadPool needs to send jobs wrapped in the Message::NewJob variant. Then, in Worker::new where a Message is received from the channel, the job will be processed if the NewJob variant is received, and the thread will break out of the loop if the Terminate variant is received.

With these changes, the code will compile and continue to function in the same way as it has been. But we’ll get a warning because we aren’t creating any messages of the Terminate variety. Let’s fix this warning by changing our Drop implementation to look like Listing 20-25.

src/lib.rs

impl Drop for ThreadPool {

fn drop(&mut self) {

println!("Sending terminate message to all workers.");

for \_ in &mut self.workers {

self.sender.send(Message::Terminate).unwrap();

}

println!("Shutting down all workers.");

for worker in &mut self.workers {

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

if let Some(thread) = worker.thread.take() {

thread.join().unwrap();

}

}

}

}

Listing 20-25: Sending Message::Terminate to the workers before calling join on each worker thread

We’re now iterating over the workers twice: once to send one Terminate message for each worker and once to call join on each worker’s thread. If we tried to send a message and join immediately in the same loop, we couldn’t guarantee that the worker in the current iteration would be the one to get the message from the channel.

To better understand why we need two separate loops, imagine a scenario with two workers. If we used a single loop to iterate through each worker, on the first iteration a terminate message would be sent down the channel and join called on the first worker’s thread. If that first worker was busy processing a request at that moment, the second worker would pick up the terminate message from the channel and shut down. We would be left waiting on the first worker to shut down, but it never would because the second thread picked up the terminate message. Deadlock!

To prevent this scenario, we first put all of our Terminate messages on the channel in one loop; then we join on all the threads in another loop. Each worker will stop receiving requests on the channel once it gets a terminate message. So, we can be sure that if we send the same number of terminate messages as there are workers, each worker will receive a terminate message before join is called on its thread.

To see this code in action, let’s modify main to only accept two requests before gracefully shutting down the server, as shown in Listing 20-26.

src/bin/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.");

}

Listing 20-26: Shut 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 the graceful shutdown and cleanup 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 [unoptimized + debuginfo] target(s) in 1.0 secs

Running `target/debug/hello`

Worker 0 got a job; executing.

Worker 3 got a job; executing.

Shutting down.

Sending terminate message to all workers.

Shutting down all workers.

Shutting down worker 0

Worker 1 was told to terminate.

Worker 2 was told to terminate.

Worker 0 was told to terminate.

Worker 3 was told to terminate.

Shutting down worker 1

Shutting down worker 2

Shutting down worker 3

You might see a different ordering of workers and messages printed. We can see how this code works from the messages: workers zero and three got the first two requests, and then on the third request the server stopped accepting connections. When the ThreadPool goes out of scope at the end of main, its Drop implementation kicks in, and the pool tells all workers to terminate. The workers each print a message when they see the terminate message, and then the thread pool calls join to shut down each worker thread.

Notice one interesting aspect of this particular execution: the ThreadPool sent the terminate messages down the channel, and before any worker received the messages, we tried to join worker 0. Worker 0 had not yet received the terminate message, so the main thread blocked waiting for worker 0 to finish. In the meantime, each of the workers received the termination messages. When worker 0 finished, the main thread waited for the rest of the workers to finish. At that point, they had all received the termination message and were able to shut down.

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 the website for this book to download the full code for this chapter for reference.

We could do here more! 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 peoples’ 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.