20 Финальный проект: сборка многопоточного сервера

Путешествие получилось довольно долгим, но мы дошли до конца книги. В этой главе мы вместе создадим еще один проект, чтобы проиллюстрировать идеи из заключительных глав, а также вспомним некоторые предыдущие уроки.

В итоговом проекте мы создадим веб-сервер, который говорит: «Привет!» и выглядит в браузере так, как показано на рис. 20.1.

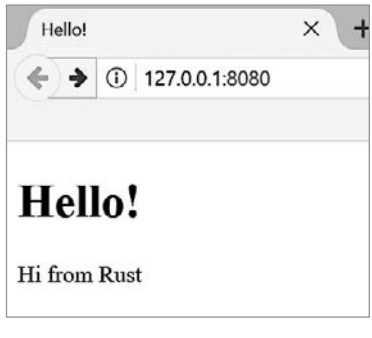

Рис. 20.1. Наш финальный совместный проект

Вот план сборки веб-сервера:

1. Узнать немного о TCP и HTTP.
2. Прослушать TCP-соединения на сокете.
3. Проанализировать небольшое число HTTP-запросов.
4. Создать соответствующий HTTP-ответ.
5. Улучшить пропускную способность сервера с помощью пула потоков исполнения.

Но прежде чем начать, мы должны уточнить одну деталь: метод, который мы будем использовать, — не лучший способ сборки веб-сервера с помощью Rust. Ряд готовых к производству упаковок доступен на https://crates.io/, и все они обеспечивают более полные реализации веб-сервера и пула потоков исполнения, чем та, что сделаем мы.

Однако наша цель — помочь вам освоить материал, а не идти по пути наименьшего сопротивления. Поскольку Rust — это язык системного программирования, мы можем выбрать уровень абстракции, с которым хотим работать, а можем перейти на более низкий уровень, чем это возможно в других языках. Мы напишем базовый HTTP-сервер и пул потоков исполнения вручную, чтобы вы могли усвоить общие идеи и технические приемы, лежащие в основе упаковок, которые вы, возможно, будете использовать в будущем.

Сборка однопоточного сервера

Начнем с того, что приведем в рабочее состояние однопоточный веб-сервер. Прежде чем начать, давайте кратко рассмотрим протоколы, участвующие в создании веб-серверов. Подробности выходят за рамки темы данной книги, поэтому мы кратко предоставим самую важную информацию.

Двумя главными протоколами, используемыми в веб-серверах, являются протокол передачи гипертекста (HTTP) и протокол управления передачей (TCP). Оба являются протоколами запросов-ответов, то есть клиент инициирует запросы, а сервер слушает их и дает клиенту ответ. Содержание этих запросов и ответов определяется протоколами.

Протокол TCP находится на более низком уровне и описывает детали того, как информация поступает из одного сервера на другой, но не указывает, что это за информация. HTTP строится поверх TCP, определяя содержимое запросов и ответов. Технически существует возможность использования HTTP с другими протоколами, но в подавляющем большинстве случаев HTTP отправляет данные по протоколу TCP. Мы будем работать с сырыми байтами запросов и ответов TCP и HTTP.

Прослушивание TCP-соединения

Веб-сервер должен прослушивать TCP-соединение, и поэтому данная часть будет первой, над которой мы будем работать. Стандартная библиотека предлагает модуль `std::net`, позволяющий это сделать. Давайте создадим новый проект обычным образом:

```
$ cargo new hello
     Created binary (application) `hello` project
$ cd hello
```

Теперь для начала введите код из листинга 20.1 в src/main.rs. Этот код будет слушать входящие TCP-потоки по адресу `127.0.0.1:7878`. Получая входящий поток, он будет выводить:

```
Соединение установлено!
```

Листинг 20.1. Прослушивание входящих потоков и печать сообщения при получении потока

src/main.rs

```
use std::net::TcpListener;

fn main() {
 ❶ let listener = TcpListener::bind("127.0.0.1:7878").unwrap();

 ❷ for stream in listener.incoming() {
     ❸ let stream = stream.unwrap();

     ❹ println!("Соединение установлено!");
   }
}
```

Используя `TcpListener`, мы можем прослушивать TCP-соединения по адресу 127.0.0.1: 7878 ❶. В адресе секция перед двоеточием — это IP-адрес, представляющий ваш компьютер (этот адрес одинаков на каждом компьютере и не представляет компьютер определенного автора), а 7878 — это порт. Мы выбрали этот порт по двум причинам: на нем обычно принимается HTTP, а 7878 — это слово rust, набранное на телефоне.

Функция `bind` в этом сценарии работает как функция `new` в том, что она возвращает новый экземпляр типа `TcpListener`. Причина, по которой указанная функция называется `bind` (то есть «привязать»), заключается в том, что в сети подключение к порту для прослушивания называется привязкой к порту.

Функция `bind` возвращает экземпляр типа `Result<T, E>`, который говорит о том, что привязка может быть неуспешной. Например, для подключения к порту 80 требуются права администратора (неадминистраторы могут прослушивать только порты выше 1024), поэтому, если мы попытаемся подключиться к порту 80, не будучи администратором, то привязка не будет работать. Еще один пример, когда привязка не будет работать: если мы запускаем два экземпляра программы, и поэтому две программы прослушивают один и тот же порт. Поскольку мы пишем базовый сервер только для целей обучения, мы не будем беспокоиться об обработке таких ошибок. Мы используем метод `unwrap`, чтобы остановить программу в случае ошибок.

Метод `incoming` в типе `TcpListener` возвращает итератор, который дает последовательность потоков ❷ (более конкретно, потоков типа `TcpStream`). Один поток представляет собой открытое соединение между клиентом и сервером. Соединение — это имя для полного процесса запроса и ответа, в котором клиент подключается к серверу, сервер генерирует ответ и закрывает соединение. Таким образом, `TcpStream` будет читать из себя, чтобы увидеть, что отправил клиент, а затем будет

давать писать ответ в поток. В целом этот цикл `for` будет обрабатывать каждое соединение по очереди и производить серию потоков для обработки.

Пока что обработка потока состоит из вызова метода `unwrap` для завершения программы, если в потоке есть ошибки ❸. Если ошибок нет, то программа выводит сообщение ❹. Мы добавим больше функциональности для случая успеха в следующем листинге. Причина, по которой мы можем получать ошибки от метода `incoming`, когда клиент подключается к серверу, заключается в том, что фактически мы перебираем не соединения, а попытки соединения. Соединение может не быть успешным по ряду причин, многие из которых зависят от операционной системы. Например, многие операционные системы имеют лимит на число одновременно открытых соединений, которые они могут поддерживать. Новые попытки соединения, превышающие это число, будут выдавать ошибку до тех пор, пока некоторые из открытых соединений не будут закрыты.

Давайте попробуем выполнить этот код! Вызовите команду `cargo run` в терминале, а затем загрузите 127.0.0.1:7878 в веб-браузере. Браузер должен выдать сообщение об ошибке наподобие «Connection reset» («Сброс соединения»), так как сервер в данный момент не отправляет обратно никаких данных. Но посмотрев на терминал, вы должны увидеть несколько сообщений, которые были выведены, когда браузер соединился с сервером!

```
     Running `target/debug/hello`
Соединение установлено!
Соединение установлено!
Соединение установлено!
```

Иногда вы увидите, что со стороны браузера выводится несколько сообщений для одного запроса. Причина может быть в том, что браузер делает запрос страницы, а также других ресурсов, таких как иконка favicon.ico, которая появляется на вкладке браузера.

Возможно также, что браузер пытается соединиться с сервером несколько раз, потому что сервер не отвечает данными. Когда переменная `stream` выходит из области видимости и отбрасывается в конце цикла, соединение закрывается как часть реализации функции `drop`. Браузеры иногда регулируют закрытые соединения путем повторной попытки, потому что проблема может быть временной. Важно то, что мы успешно получили дескриптор для TCP-соединения!

Не забудьте остановить программу, нажав Ctrl-C, когда закончите выполнение отдельной версии кода. После внесения любых изменений в код заново выполните команду `cargo run`, чтобы работать с последней версией кода.

Чтение запроса

Давайте реализуем функциональность чтения запроса из браузера! Чтобы сначала получить соединение и затем выполнить некие действия с ним, мы начнем новую

функцию обработки соединений. В этой новой функции `handle_connection` мы будем читать данные из TCP-потока и печатать их, чтобы видеть данные, отправляемые из браузера. Измените код так, чтобы он выглядел как в листинге 20.2.

Листинг 20.2. Чтение из потока TcpStream и печать данных

src/main.rs

```
❶ use std::io::prelude::*;
  use std::net::TcpStream;
  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();

        ❷ handle_connection(stream);
      }
  }

  fn handle_connection(❸mut stream: TcpStream) {
    ❹ let mut buffer = [0; 512];

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

    ❻ println!("Запрос: {}", String::from_utf8_lossy(&buffer[..]));
  }
```

Мы вводим `std::io::prelude` в область видимости, чтобы получить доступ к некоторым типажам, позволяющим читать и писать в поток ❶. В цикле `for` в функции `main`, вместо того, чтобы печатать сообщение о том, что у нас есть соединение, мы теперь вызываем новую функцию `handle_connection` и передаем ей переменную `stream` ❷.

В функции `handle_connection` мы сделали параметр `stream` изменяемым ❸. Причина в том, что внутренне экземпляр типа `TcpStream` отслеживает то, какие данные он нам возвращает. Он может прочитать данные в объеме, превышающем тот, который мы запрашивали, и сохранить их для следующего запроса. Следовательно, ему нужно ключевое слово `mut`, потому что его внутреннее состояние может измениться. Обычно мы думаем, что «чтение» не нуждается в изменении, но в данном случае указанное ключевое слово необходимо.

Далее нам фактически нужно читать из потока. Мы делаем это в два этапа: во-первых, объявляем переменную `buffer` в стеке для хранения считываемых данных ❹. Мы создали буфер размером 512 байт, которого будет достаточно для хранения данных базового запроса и для других целей этой главы. Если бы мы хотели обрабатывать запросы произвольного размера, то управление буфером было бы сложнее. Мы пока оставим его простым. Мы передаем буфер методу `stream.read`, который будет читать байты из `TcpStream` и помещать их в буфер ❺.

Затем мы конвертируем байты в буфере в экземпляр типа `String` и печатаем его ❻. Функция `String::from_utf8_lossy` берет `&[u8]` и производит из него экземпляр типа `String`. Часть имени «lossy» («с потерями») указывает на поведение этой функции при виде недопустимой последовательности UTF-8: она будет заменять недопустимую последовательность символом `U+FFFD`, то есть символом замены, `U+FFFD`. Вы можете увидеть символы замены для символов в буфере, которые не заполнены данными запроса.

Давайте испытаем этот код! Выполните программу и снова сделайте запрос в веб-браузере. Обратите внимание, мы все равно получим страницу ошибки в браузере, но данные программы в терминале теперь будут выглядеть примерно так:

```
$ 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
������������������������������������������
```

В зависимости от вашего браузера вы можете получить немного другие данные. Теперь, печатая данные запроса, мы можем понять, почему получается несколько соединений из одного запроса со стороны браузера, посмотрев на путь после `Request: GET`. Если все повторяющиеся соединения запрашивают ресурс `/`, то мы знаем, что браузер многократно пытается получить ресурс `/`, потому что он не получает ответа от программы.

Давайте разложим данные запроса, чтобы увидеть, что именно браузер запрашивает у программы.

HTTP-запрос

HTTP — это текстовый протокол, запрос принимает такой формат:

```
Method Request-URI HTTP-Version CRLF
headers CRLF
message-body
```

Первая строка формата — это строка запроса, содержащая информацию о том, что клиент запрашивает. Первая часть строки запроса указывает на используемый метод, например `GET` или `POST`, который описывает, как клиент делает запрос. Наш клиент использовал запрос `GET`.

Следующая часть строки запроса — это символ `/`, который указывает на единый идентификатор ресурса (URI), запрашиваемый клиентом: URI почти, но не совсем совпадает с единым локатором ресурсов (URL). Разница между URI- и URL-адресами в этой главе нам не важна, но спецификация HTTP использует термин URI, поэтому здесь можно просто мысленно подставить URL вместо URI.

Последняя часть — это версия HTTP, которую клиент использует, а затем строка запроса заканчивается последовательностью CRLF. (CRLF расшифровывается как *carriage return and line feed*, то есть «возврат каретки и подача строки», эти термины остались со времен пишущих машинок!) Последовательность CRLF также может быть записана как `\r\n`, где `\r` — это возврат каретки, а `\n` — подача строки. Последовательность CRLF отделяет строку запроса от остальных данных запроса. Обратите внимание, во время печати CRLF мы видим начало новой строки, а не `\r\n`.

Глядя на данные строки запроса, полученные из программы в ее настоящем состоянии, мы видим, что запрос имеет метод `GET`, URI запроса `/` и версию `HTTP/1.1`.

После строки запроса остальные строки, начиная с `Host:` и далее, являются заголовками. Запросы `GET` не имеют тела.

Попробуйте сделать запрос из другого браузера или запросить другой адрес, например 127.0.0.1:7878/test, чтобы увидеть, как изменяются данные запроса.

Теперь, зная, что именно запрашивает браузер, давайте отправим назад некоторые данные!

Написание ответа

Теперь мы реализуем отправку данных в ответ на запрос клиента. Ответы имеют следующий формат:

```
HTTP-Version Status-Code Reason-Phrase CRLF
headers CRLF
message-body
```

Первая строка отклика — это строка состояния, содержащая версию HTTP, используемую в ответе, числовой код состояния, который резюмирует результат запроса, и причину с текстовым описанием кода состояния. После последовательности CRLF идут любые заголовки, еще одна последовательность CRLF и тело ответа.

Вот пример ответа, который использует HTTP версии 1.1, имеет код состояния `200`, причину `OK`, без заголовков и без тела:

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

Код состояния 200 — это стандартный ответ об успешном выполнении. Текст представляет собой крошечный успешный HTTP-ответ. Давайте запишем это

в поток как наш ответ на успешный запрос! Из функции `handle_connection` удалите инструкцию `println!`, которая печатала данные запроса, и замените ее кодом из листинга 20.3.

Листинг 20.3. Запись крошечного успешного HTTP-ответа в поток

src/main.rs

```
fn handle_connection(mut stream: TcpStream) {
    let mut buffer = [0; 512];

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

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

  ❷ stream.write(response.as_bytes()❸).unwrap();
  ❹ stream.flush().unwrap();
}
```

Первая новая строка определяет переменную `response`, которая содержит данные сообщения об успехе ❶. Затем мы вызываем метод `as_bytes` для переменной `response`, чтобы конвертировать строковые данные в байты ❸. Метод `write` для `stream` берет `&[u8]` и посылает эти байты непосредственно по соединению ❷.

Поскольку операция `write` может не сработать, мы используем метод `unwrap` для любого результата с ошибкой, как и раньше. Опять же, в реальном приложении вы бы добавили сюда обработку ошибок. Наконец, метод `flush` будет ждать и препятствовать продолжению программы до тех пор, пока все байты не будут записаны в соединение ❹. Тип `TcpStream` содержит внутренний буфер для минимизации вызовов опорной операционной системы.

Внеся эти изменения, давайте выполним код и сделаем запрос. Мы больше не печатаем данные в терминал, поэтому в нем будут только данные из Cargo. Когда вы загрузите 127.0.0.1:7878 в веб-браузере, вы должны получить пустую страницу вместо ошибки. Вы только что вручную закодировали HTTP-запрос и ответ!

Возвращение реального HTML

Давайте реализуем функциональность для возвращения не только пустой страницы. В корневом каталоге проекта, но не в каталоге src, создайте новый файл hello.html. Вы можете ввести любой HTML, который хотите. В листинге 20.4 показан один из вариантов.

Листинг 20.4. Пример HTML-файла для возвращения в ответе

hello.html

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="utf-8">
```

```
    <title>Привет!</title>
  </head>
  <body>
    <h1>Привет!</h1>
    <p>Привет от Rust</p>
  </body>
</html>
```

Это минимальный документ HTML5 с заголовком и некоторым текстом. Для того чтобы вернуть его с сервера при получении запроса, мы изменим функцию `handle_connection`, как показано в листинге 20.5, так, чтобы прочитать HTML-файл, добавить его в ответ в качестве тела и отправить.

Листинг 20.5. Отправка содержимого hello.html в качестве тела ответа

src/main.rs

```
❶ use std::fs;
  // --пропуск--

  fn handle_connection(mut stream: TcpStream) {
      let mut buffer = [0; 512];
      stream.read(&mut buffer).unwrap();

      let contents = fs::read_to_string("hello.html").unwrap();

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

      stream.write(response.as_bytes()).unwrap();
      stream.flush().unwrap();
  }
```

Мы добавили строку вверху, чтобы ввести модуль файловой системы стандартной библиотеки в область видимости ❶. Код для чтения содержимого файла в строку должен выглядеть знакомо: мы использовали его в главе 12, когда читали содержимое файла для проекта ввода-вывода в листинге 12.4.

Далее мы используем макрокоманду `format!` для добавления содержимого файла в тело ответа об успешном выполнении ❷.

Выполните этот код с помощью команды `cargo run` и загрузите 127.0.0.1:7878 в свой браузер. Вы должны увидеть, что HTML отображен на экране!

В настоящее время мы игнорируем данные запроса в переменной `buffer` и просто отправляем обратно содержимое HTML-файла в безусловном порядке. Это означает, что если вы попытаетесь запросить в браузере *127.0.0.1:7878/что-то-еще*, то все равно получите тот же HTML-ответ. Наш сервер очень ограничен и не делает того, что делает большинство веб-серверов. Мы хотим настраивать ответы индивидуально, в зависимости от запроса, и отправлять обратно HTML-файл только для хорошо сформированного запроса ресурса /.