

Оглавление

| | |
|---|-----------|
| Вступление | 19 |
| Предисловие | 20 |
| Благодарности | 21 |
| Об авторах..... | 22 |
| Введение | 23 |
| Кому подойдет язык Rust..... | 23 |
| Команды разработчиков..... | 23 |
| Студенты..... | 24 |
| Компании | 24 |
| Разработчики открытого исходного кода | 24 |
| Люди, ценящие скорость и стабильность..... | 24 |
| Для кого эта книга | 25 |
| Как пользоваться этой книгой | 25 |
| Ресурсы | 27 |
| От издательства | 27 |
| Глава 1. Начало работы | 28 |
| Установка..... | 28 |
| Установка инструмента rustup в Linux или macOS..... | 29 |
| Установка инструмента rustup в Windows | 30 |
| Обновление и деинсталляция..... | 30 |
| Устранение неисправностей | 30 |
| Локальная документация | 31 |
| Здравствуй, Мир!..... | 31 |
| Создание каталога проектов | 31 |
| Написание и выполнение программы Rust..... | 32 |
| Анатомия программы на языке Rust | 33 |
| Компиляция и выполнение являются отдельными шагами..... | 34 |
| Здравствуй, Cargo! | 35 |
| Создание проекта с помощью Cargo | 35 |
| Построение проекта Cargo и его выполнение | 37 |
| Сборка для релиза | 39 |

| | |
|---|-----------|
| Cargo как общепринятое средство..... | 39 |
| Итоги..... | 40 |
| Глава 2. Программирование игры-угадайки | 41 |
| Настройка нового проекта..... | 41 |
| Обработка загаданного числа..... | 42 |
| Хранение значений с помощью переменных..... | 43 |
| Обработка потенциального сбоя с помощью типа Result | 45 |
| Печать значений с помощью заполнителей макрокоманды println! | 47 |
| Тестирование первой части..... | 47 |
| Генерирование секретного числа | 47 |
| Использование упаковки для получения большей функциональности | 48 |
| Генерирование случайного числа..... | 50 |
| Сравнение загаданного числа с секретным числом..... | 52 |
| Вывод нескольких загаданных чисел с помощью цикличности..... | 56 |
| Выход из игры после правильно угаданного числа | 57 |
| Обработка ввода недопустимых данных | 58 |
| Итоги..... | 60 |
| Глава 3. Концепции программирования | 61 |
| Переменные и изменяемость | 61 |
| Различия между переменными и константами | 63 |
| Затенение | 64 |
| Типы данных | 66 |
| Скалярные типы | 67 |
| Составные типы | 71 |
| Функции | 74 |
| Параметры функций..... | 75 |
| Инструкции и выражения в телах функций..... | 76 |
| Функции с возвращаемыми значениями | 78 |
| Комментарии..... | 80 |
| Управление потоком..... | 81 |
| Выражения if..... | 81 |
| Повторение с помощью циклов | 85 |
| Итоги..... | 89 |
| Глава 4. Концепция владения | 90 |
| Что такое владение? | 90 |
| Правила владения..... | 92 |

| | |
|---|------------|
| Область видимости переменной | 92 |
| Строковый тип | 93 |
| Память и выделение пространства | 94 |
| Владение и функции | 100 |
| Возвращаемые значения и область видимости | 101 |
| Ссылки и заимствование..... | 102 |
| Изменяемые ссылки | 104 |
| Висячие ссылки | 107 |
| Правила ссылок | 108 |
| Срезовой тип | 109 |
| Строковые срезы | 111 |
| Другие срезы | 115 |
| Итоги..... | 115 |
| Глава 5. Использование структур для связанных данных..... | 116 |
| Определение и инстанцирование структур | 116 |
| Использование краткой инициализации полей: когда у переменных и полей одинаковые имена | 118 |
| Создание экземпляров из других экземпляров с помощью синтаксиса обновления структуры..... | 118 |
| Использование кортежных структур без именованных полей для создания разных типов | 119 |
| Unit-подобные структуры без полей | 120 |
| Пример программы с использованием структур | 121 |
| Рефакторинг с использованием кортежей | 122 |
| Рефакторинг с использованием структур: добавление большего смысла | 123 |
| Добавление полезной функциональности с использованием типажей с атрибутом <code>derived</code> | 124 |
| Синтаксис методов | 126 |
| Определение методов | 126 |
| Методы с большим числом параметров | 128 |
| Связанные функции | 129 |
| Несколько блоков <code>impl</code> | 130 |
| Итоги..... | 131 |
| Глава 6. Перечисления и сопоставление с паттернами..... | 132 |
| Определение перечисления..... | 132 |
| Выражение <code>match</code> как оператор управления потоком | 139 |
| Паттерны, которые привязываются к значениям | 141 |

| | |
|--|------------|
| Сопоставление с Option<T> | 142 |
| Совпадения являются исчерпывающими | 143 |
| Заполнитель _ | 144 |
| Сжатое управление потоком с помощью if let | 145 |
| Итоги..... | 146 |
| Глава 7. Управление растущими проектами с помощью пакетов, упаковок и модулей | 147 |
| Пакеты и упаковки | 148 |
| Определение модулей для управления областью видимости и конфиденциальностью | 149 |
| Пути для ссылки на элемент в дереве модулей..... | 151 |
| Демонстрация путей с помощью ключевого слова pub | 154 |
| Начало относительных путей с помощью super | 156 |
| Обозначение структур и перечислений как публичных..... | 157 |
| Введение путей в область видимости с помощью ключевого слова use | 159 |
| Создание идиоматических путей use | 160 |
| Предоставление новых имен с помощью ключевого слова as | 162 |
| Резкспорт имен с использованием pub | 162 |
| Использование внешних пакетов..... | 163 |
| Использование вложенных путей для очистки больших списков use | 164 |
| Оператор glob | 165 |
| Разделение модулей на разные файлы | 165 |
| Итоги..... | 167 |
| Глава 8. Общие коллекции | 168 |
| Хранение списков значений с помощью векторов | 168 |
| Создание нового вектора | 169 |
| Обновление вектора | 169 |
| Отбрасывание вектора отбрасывает его элементы | 170 |
| Чтение элементов вектора | 170 |
| Перебор значений в векторе | 172 |
| Использование перечисления для хранения нескольких типов..... | 173 |
| Хранение текста в кодировке UTF-8 с помощью строк | 174 |
| Что такое тип String?..... | 174 |
| Создание нового экземпляра типа String | 175 |
| Обновление строки | 176 |
| Индексирование в строках | 179 |
| Нарезка строк | 181 |

| | |
|--|------------|
| Методы перебора строк..... | 182 |
| Строки не так просты..... | 182 |
| Хранение ключей со связанными значениями в хеш-отображениях..... | 183 |
| Создание нового хеш-отображения..... | 183 |
| Хеш-отображения и владение..... | 184 |
| Доступ к значениям в хеш-отображении..... | 185 |
| Обновление хеш-отображения..... | 186 |
| Хеширующие функции..... | 188 |
| Итоги..... | 188 |
| Глава 9. Обработка ошибок..... | 190 |
| Неустранимые ошибки и макрокоманда panic!..... | 190 |
| Использование обратной трассировки при вызове panic!..... | 192 |
| Устраняемые ошибки с помощью Result..... | 194 |
| Применение выражения match с разными ошибками..... | 197 |
| Краткие формы для паники в случае ошибки: unwrap и expect..... | 198 |
| Распространение ошибок..... | 200 |
| Паниковать! Или не паниковать!..... | 205 |
| Примеры, прототипный код и тесты..... | 205 |
| Случаи, когда у вас больше информации, чем у компилятора..... | 206 |
| Принципы обработки ошибок..... | 206 |
| Создание настраиваемых типов для проверки допустимости..... | 208 |
| Итоги..... | 210 |
| Глава 10. Обобщенные типы, типаж и жизненный цикл..... | 211 |
| Удаление повторов путем извлечения функции..... | 212 |
| Обобщенные типы данных..... | 214 |
| В определениях функций..... | 214 |
| В определениях структуры..... | 217 |
| В определениях перечислений..... | 219 |
| В определениях методов..... | 220 |
| Производительность кода с использованием обобщений..... | 222 |
| Типаж: определение совместного поведения..... | 223 |
| Определение типажа..... | 223 |
| Реализация типажа в типе..... | 224 |
| Реализации по умолчанию..... | 226 |
| Типаж в качестве параметров..... | 228 |
| Возвращение типов, реализующих типаж..... | 230 |

| | |
|--|------------|
| Исправление функции <code>largest</code> с помощью границ типажа | 231 |
| Использование границ типажа для условной реализации методов | 233 |
| Проверка ссылок с помощью жизненных циклов | 235 |
| Предотвращение висячих ссылок с помощью жизненного цикла | 235 |
| Контролер заимствования | 236 |
| Обобщенные жизненные циклы в функциях | 237 |
| Синтаксис аннотаций жизненных циклов | 239 |
| Аннотации жизненных циклов в сигнатурах функций | 240 |
| Мышление в терминах жизненных циклов | 242 |
| Аннотации жизненных циклов в определениях структур | 244 |
| Пропуск жизненного цикла | 244 |
| Аннотации жизненных циклов в определениях методов | 247 |
| Статический жизненный цикл | 248 |
| Параметры обобщенного типа, границы типажа и жизненный цикл вместе | 249 |
| Итоги | 249 |
| Глава 11. Автоматизированные тесты | 250 |
| Как писать тесты | 251 |
| Анатомия функции тестирования | 251 |
| Проверка результатов с помощью макрокоманды <code>assert!</code> | 255 |
| Проверка равенства с помощью макрокоманд <code>assert_eq!</code> и <code>assert_ne!</code> | 257 |
| Добавление сообщений об ошибках для пользователя | 260 |
| Проверка на панику с помощью атрибута <code>should_panic</code> | 261 |
| Использование типа <code>Result<T, E></code> в тестах | 265 |
| Контроль выполнения тестов | 265 |
| Параллельное и последовательное выполнение тестов | 266 |
| Показ результатов функции | 267 |
| Выполнение подмножества тестов по имени | 268 |
| Игнорирование нескольких тестов, только если не запрошено иное | 270 |
| Организация тестов | 271 |
| Модульные тесты | 271 |
| Интеграционные тесты | 273 |
| Интеграционные тесты для двоичных упаковок | 277 |
| Итоги | 277 |
| Глава 12. Проект ввода-вывода: сборка программы командной строки | 278 |
| Принятие аргументов командной строки | 279 |
| Чтение значений аргументов | 279 |
| Сохранение значений аргументов в переменных | 281 |

| | |
|---|-----|
| Чтение файла | 282 |
| Рефакторинг с целью улучшения модульности и обработки ошибок | 283 |
| Разделение обязанностей в двоичных проектах | 284 |
| Исправление обработки ошибок | 289 |
| Извлечение алгоритма из функции main | 292 |
| Разбивка кода в библиотечную упаковку | 295 |
| Развитие функциональности библиотеки с помощью методики разработки на основе тестов | 296 |
| Написание провального теста | 297 |
| Написание кода для успешного завершения теста | 299 |
| Работа с переменными среды | 302 |
| Написание провального теста для функции search, нечувствительной к регистру | 303 |
| Реализация функции search_case_insensitive | 304 |
| Запись сообщений об ошибках в стандартный вывод ошибок вместо стандартного вывода данных | 308 |
| Проверка места, куда записываются ошибки | 308 |
| Запись сообщения об ошибках в стандартный вывод ошибок | 309 |
| Итоги | 310 |

Глава 13. Функциональные средства языка:

| | |
|---|------------|
| итераторы и замыкания | 311 |
| Замыкание: анонимные функции, которые могут захватывать среду | 311 |
| Создание абстракции поведения с помощью замыканий | 312 |
| Логический вывод типа и аннотация замыкания | 317 |
| Ограничения в реализации структуры Cacher | 322 |
| Захватывание среды с помощью замыканий | 323 |
| Обработка серии элементов с помощью итераторов | 326 |
| Типаж Iterator и метод next | 327 |
| Методы, которые потребляют итератор | 328 |
| Методы, которые производят другие итераторы | 329 |
| Использование замыканий, которые захватывают свою среду | 330 |
| Создание собственных итераторов с помощью типажа Iterator | 331 |
| Улучшение проекта ввода-вывода | 334 |
| Удаление метода clone с помощью Iterator | 334 |
| Написание более ясного кода с помощью итераторных адаптеров | 337 |
| Сравнение производительности: циклы против итераторов | 338 |
| Итоги | 340 |

| | |
|--|------------|
| Глава 14. Подробнее о Cargo и Crates.io | 341 |
| Собственная настройка сборок с помощью релизных профилей | 341 |
| Публикация упаковки для Crates.io | 343 |
| Внесение полезных документационных комментариев | 343 |
| Экспорт удобного публичного API с использованием pub | 347 |
| Настройка учетной записи Crates.io | 351 |
| Добавление метаданных в новую упаковку | 351 |
| Публикация в Crates.io | 353 |
| Публикация новой версии существующей упаковки..... | 353 |
| Удаление версий из Crates.io с помощью команды cargo yank..... | 353 |
| Рабочие пространства Cargo..... | 354 |
| Создание рабочего пространства | 354 |
| Создание второй упаковки в рабочем пространстве | 355 |
| Установка двоичных файлов из Crates.io с помощью команды cargo install..... | 360 |
| Расширение Cargo с помощью индивидуальных команд..... | 361 |
| Итоги..... | 361 |
| Глава 15. Умные указатели..... | 362 |
| Использование <code>Box<T></code> для указания на данные в куче | 363 |
| Использование <code>Box<T></code> для хранения данных в куче | 364 |
| Применение рекурсивных типов с помощью умных указателей <code>Box</code> | 365 |
| Трактовка умных указателей как обыкновенных ссылок с помощью типажа <code>Deref</code> | 369 |
| Следование по указателю к значению с помощью оператора разыменования | 370 |
| Использование <code>Box<T></code> в качестве ссылки | 371 |
| Определение собственного умного указателя..... | 371 |
| Трактовка типа как ссылки путем реализации типажа <code>Deref</code> | 372 |
| Скрытые принудительные приведения типов посредством <code>deref</code> с функциями и методами..... | 374 |
| Как принудительное приведение типа посредством <code>deref</code> взаимодействует с изменяемостью | 375 |
| Выполнение кода при очистке с помощью типажа <code>Drop</code> | 376 |
| Досрочное отбрасывание значения с помощью <code>std::mem::drop</code> | 378 |
| <code>Rc<T></code> — умный указатель подсчета ссылок..... | 380 |
| Применение <code>Rc<T></code> для совместного использования данных..... | 380 |
| Клонирование <code>Rc<T></code> увеличивает число ссылок | 383 |
| <code>RefCell<T></code> и паттерн внутренней изменяемости | 384 |

| | |
|--|------------|
| Соблюдение правил заимствования во время выполнения с помощью RefCell<T> | 384 |
| Внутренняя изменяемость: изменяемое заимствование неизменяемого значения..... | 386 |
| Наличие нескольких владельцев изменяемых данных путем сочетания Rc<T> и RefCell<T> | 392 |
| Циклы в переходах по ссылкам приводят к утечке памяти..... | 394 |
| Создание цикла в переходах по ссылкам..... | 394 |
| Предотвращение циклов в переходах по ссылкам: превращение Rc<T> в Weak<T>..... | 397 |
| Итоги..... | 403 |
| Глава 16. Конкурентность без страха | 404 |
| Использование потоков исполнения для одновременного выполнения кода..... | 405 |
| Создание нового потока с помощью spawn..... | 407 |
| Ожидание завершения работы всех потоков с использованием дескрипторов join..... | 408 |
| Использование замыкания move с потоками..... | 410 |
| Использование передачи сообщений для пересылки данных между потоками..... | 413 |
| Каналы и передача владения | 416 |
| Отправка нескольких значений и ожидание приемника | 417 |
| Создание нескольких производителей путем клонирования передатчика..... | 418 |
| Конкурентность совместного состояния..... | 420 |
| Использование мьютексов для обеспечения доступа к данным из одного потока за один раз..... | 420 |
| Сходства между RefCell<T>/Rc<T> и Mutex<T>/Arc<T> | 428 |
| Расширяемая конкурентность с типажми Send и Sync | 428 |
| Разрешение передавать владение между потоками с помощью Send..... | 429 |
| Разрешение доступа из нескольких потоков исполнения с помощью Sync | 429 |
| Реализовывать Send и Sync вручную небезопасно..... | 430 |
| Итоги..... | 430 |
| Глава 17. Средства объектно-ориентированного программирования | 431 |
| Характеристики объектно-ориентированных языков..... | 431 |
| Объекты содержат данные и поведение | 432 |
| Инкапсуляция, которая скрывает детали реализации | 432 |
| Наследование как система типов и как совместное использование кода | 434 |
| Использование типажных объектов, допускающих значения разных типов | 435 |
| Определение типажа для часто встречающегося поведения | 436 |

| | |
|---|------------|
| Реализация типажа | 438 |
| Типажные объекты выполняют динамическую диспетчеризацию | 441 |
| Объектная безопасность необходима для типажных объектов..... | 442 |
| Реализация объектно-ориентированного паттерна проектирования | 443 |
| Определение поста и создание нового экземпляра в состоянии черновика ... | 445 |
| Хранение текста поста | 446 |
| Делаем пустой черновик | 446 |
| Запрос на проверку статьи изменяет ее состояние | 447 |
| Добавление метода approve, который изменяет поведение метода content | 449 |
| Компромиссы паттерна переходов между состояниями | 452 |
| Итоги..... | 457 |
| Глава 18. Паттерны и сопоставление..... | 458 |
| Где могут использоваться паттерны | 459 |
| Ветви выражения match | 459 |
| Условные выражения if let..... | 459 |
| Условные циклы while let..... | 461 |
| Циклы for | 461 |
| Инструкции let..... | 462 |
| Параметры функций..... | 463 |
| Опровержимость: возможность несовпадения паттерна | 464 |
| Синтаксис паттернов | 466 |
| Сопоставление литералов | 466 |
| Сопоставление именованных переменных | 466 |
| Несколько паттернов..... | 468 |
| Сопоставление интервалов значений с помощью синтаксиса ... | 468 |
| Деструктурирование для выделения значений | 469 |
| Игнорирование значений в паттерне..... | 474 |
| Дополнительные условия с ограничителями совпадений..... | 479 |
| Привязки @..... | 481 |
| Итоги..... | 482 |
| Глава 19. Продвинутое средства | 483 |
| Небезопасный Rust | 483 |
| Небезопасные сверхспособности | 484 |
| Применение оператора разыменования к сырому указателю..... | 485 |
| Вызов небезопасной функции или метода | 487 |

| | |
|--|------------|
| Обращение к изменяемой статической переменной или ее модифицирование | 492 |
| Реализация небезопасного типажа | 493 |
| Когда использовать небезопасный код | 494 |
| Продвинутые типажи | 494 |
| Детализация заполнительных типов в определениях типажей с помощью связанных типов | 494 |
| Параметры обобщенного типа по умолчанию и перегрузка операторов | 496 |
| Полный синтаксис для устранения неоднозначности: вызов методов с одинаковым именем | 498 |
| Использование супертипажей, требующих функциональности одного типажа внутри другого типажа | 502 |
| Использование паттерна newtype для реализации внешних типажей во внешних типах | 504 |
| Продвинутые типы | 505 |
| Использование паттерна newtype для безопасности типов и абстракции | 506 |
| Создание синонимов типов с помощью псевдонимов типов | 506 |
| Тип never, который никогда не возвращается | 508 |
| Динамически изменяемые типы и типаж Sized | 510 |
| Продвинутые функции и замыкания | 512 |
| Указатели функций | 512 |
| Возвращающие замыкания | 514 |
| Макрокоманды | 515 |
| Разница между макрокомандами и функциями | 516 |
| Декларативные макрокоманды с помощью macro_rules! для общего метапрограммирования | 516 |
| Процедурные макрокоманды для генерирования кода из атрибутов | 519 |
| Как написать настраиваемую макрокоманду derive | 520 |
| Макрокоманды, подобные атрибутам | 525 |
| Макрокоманды, подобные функциям | 526 |
| Итоги | 527 |
| Глава 20. Финальный проект: сборка многопоточного сервера | 528 |
| Сборка однопоточного сервера | 529 |
| Прослушивание TCP-соединения | 529 |
| Чтение запроса | 531 |
| HTTP-запрос | 533 |
| Написание ответа | 534 |
| Возвращение реального HTML | 535 |

| | |
|---|------------|
| Проверка запроса и выборочный ответ | 537 |
| Небольшой рефакторинг | 538 |
| Превращение однопоточного сервера в многопоточный..... | 540 |
| Моделирование медленного запроса в текущей реализации сервера | 540 |
| Повышение пропускной способности с помощью пула потоков исполнения | 541 |
| Корректное отключение и очистка | 561 |
| Реализация типажа Drop для ThreadPool | 562 |
| Подача потокам сигнала об остановке прослушивания заданий | 564 |
| Итоги..... | 569 |
| Приложение А. Ключевые слова | 570 |
| Ключевые слова, употребляемые в настоящее время | 570 |
| Ключевые слова, зарезервированные для использования в будущем | 572 |
| Сырые идентификаторы | 572 |
| Приложение Б. Операторы и символы | 574 |
| Операторы | 574 |
| Неоператорные символы | 576 |
| Приложение В. Генерируемые типы | 581 |
| Debug для вывода рабочей информации | 582 |
| PartialEq и Eq для сравнений равенств..... | 582 |
| PartialOrd и Ord для сравнений порядка..... | 583 |
| Clone и Copy для дублирования значений..... | 583 |
| Хеш для отображения значения в значение фиксированного размера..... | 584 |
| Default для значений по умолчанию | 585 |
| Приложение Г. Полезные инструменты разработки | 586 |
| Автоматическое форматирование с помощью rustfmt | 586 |
| Исправляйте код с помощью rustfix | 586 |
| Статический анализ кода с помощью Clippy..... | 588 |
| Интеграция с IDE с помощью языкового сервера Rust Language Server..... | 589 |
| Приложение Д. Редакции..... | 590 |

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.

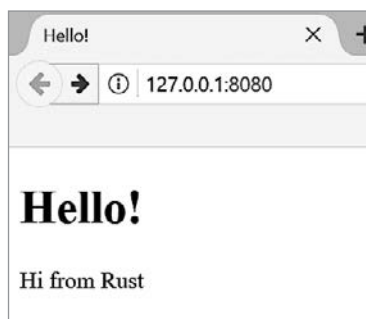


Рис. 20.1. Наш финальный совместный проект

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-поток 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(❸ mut 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 прочесть 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 . 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 , 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()).unwrap();
    ❸ 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 ❷. Метод `write` для `stream` берет `&[u8]` и посылает эти байты непосредственно по соединению ❸.

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 сверху, 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 `/`.