Документация

Посмотреть, как это работает, можно здесь
https://vk.com/video229844553_456239292

«Парадигма» программирования

Я пошёл сложным, но выгодным в дальнейшей перспективе, путём – реализовал на Future-ах. Да, это крайне неудобный способ, но это единственный метод в Rust создать ассинхронный сервер, когда даже один поток с частотой 3.3ггц может обрабатывать тысячи HTTP-запросов, а если ядер несколько, например, 4, то можно обрабатывать и ещё в 4 раза больше запросов.

В таком случае для каждого запроса создаётся специальный лёгкий поток(или задача), который может пребывать длительное время в состоянии ожидания каких-либо данных от удалённых ресурсов, а в это время может выполняться какой-то ещё лёгкий процесс, нативные потоки ОС же не могут обеспечить столь высокой скорости переключения контекстов потоков по аппаратным причинам, а сами потоки требуют немало памяти под стек. Конечно, производительность будет замедляться из-за конкуренции за данные между лёгкими процессами, но это уже напрямую зависит от архитектуры.

К сожалению, использование футур предполагает фактически создание подобия AST этих задач при помощи Rust-а, что превращает организацию какого-то цикла или выбора match в кошмар. Более того, есть одна особенность: футуры, образующие AST вкладываются друг в друга и образуют сложные трудночитаемые структуры вида <<<<<<>>>>>, имена которых довольно скоро начинают весить 3МБ, компилятору не хватает на них памяти и он начинает тормозить. Поэтому резонно что бы функции/методы не вкладывались в функции, вызывающие их, особенно это необходимо для API, методы которых вызываются многими функциями. Поэтому разделим функции на футурах по этому критерию на:

- Встраиваемые, вида -> impl Future<Item = Option<String>, Error = Error> {
- Невстраиваемые, вида -> Box<Future<Item = String, Error = Error>> {

Встраиваемые, можно считать, являются лишь вынесенными наружу для удобства функциями, аналогично макросам. Невстраиваемые удовлетворяют трейту, поэтому можно даже применить динамический полиморфизм(TraitObject + dynamic dispatch).

Всем функциям полезно возвращать Error из библиотеки failure, такая ошибка крайне напоминает исключения. Особенно важно следующее свойство: при динамическом полиморфизме функции могут вернуть совершенно различные ошибки, что при enum Error было бы затруднительно т. к. в таком случае все возможные ошибки должны быть перечислены.

Архитектура

Проект разбит на несколько крейтов, и каждый крейт предлагает свой API и специализируется на чём-то определённом, это позволяет распараллелить разработку проекта. Api для работы с картинками отделено от самого сервера и называется ImageApi

Крейты следующие:

- Сам сервер simple_web_server и вложенные в него крейты(для компактности), для работы с картинками он использует ImageApi
- ImageApi позволяет сохранять картинки в «БД», умеет делать и http запросы для скачивания файлов по URL, для масштабирования картинок использует OpenCV
- OpenCV_sys является FFI обёрткой над OpenCV
- OpenCV является высокоуровневой обёрткой над OpenCV

Крейт SimpleWebServer

SimpleWebServer использует следующие крейты как зависимости:

- failure, failure_derive для ошибок
- futures, предоставляющие футуры
- actix_web, содержащий фреймворк/шедулер задач/лёгких потоков tokio, а так же массу средств для работы с http/web

При старте сервера создаётся дочерний нативный поток, в котором и запускается HTTP-server. Другой же, main, остаётся ждать Ctrl+C, после получения этого сигнала он завершает http_server. Так же устанавливается обработчик сигнала Ctrl+C, который посылает команду потоку main при помощи канала.

Arctix умеет осуществлять маршрутизацию http запросов, поэтому созданы следующие сервисы:

- пустой запрос / по get создаёт страницу загрузки изображений на сервер (невстроенная функция ImageApp::show_image),
- пустой запрос / по post их загрузку на сервер(upload) (невстроенная функция ImageApp::put_image)
- запрос /get_image по get загрузку изображений с сервера (download) (невстроенная функция ImageApp::get_image).

ImageApp

Невстроенная функция ImageApp::show_image создаёт страницу просмотра/загрузки изображения, вызывая встроенную функцию ImageApp::create_page. Если ImageApp::create_page завершается ошибкой(фатальной), отправляет пользователю описание ошибки.

Встроенная функция create_page вызывается до и после загрузки картинки функциями ImageApp::show_image и put_image соответственно, и создаёт страницу, либо возвращает ошибку, если ошибка фатальна.

- Сначала она формирует описание всех положительных результатов при загрузке изображений
- Потом формирует описание всех ошибок
- Потом формирует форму загрузки изображения/ий
- Запрашивает у ImageApi список всех картинок в БД, или выводит ошибку, если такова произошла
- Формирует список изображений в http, далее клиент загружает эти изображения по url, делая запрос /get_image по get

Невстроенная функция ImageApp::get_image отправляет пользователю запрашиваемую картинку. Данные картинки она получает при помощи вызова метода ImageApi::get_image, если произошла ошибка, возвращает текст ошибки

Встроенная функция ImageApp::put_image позволяет загрузить изображения на сервер. Поддерживает Multipart, URL, base-64.

Функция устроена довольно сложно:

- она формирует stream полей запроса при помощи вызовов встраиваемой функции ImageApp::read_field
- не все поля могут быть заданы, их не следует обрабатывать вообще, поэтому их фильтруем filter
- Option<Field> превращаем в Field .map(|field| field.unwrap())
- Превращяем поток полей в вектор
- Обрабатываем вектор полей:
- Для каждого поля вызываем встроенную функцию AppImage::process_put_image, которая при помощи вызова ImageApi::put_image загружает изображение в БД, возвращает результат, в т.ч. и ошибку
- Превращает поток результатор в вектор
- Обрабатываем ошибки/результаты, сортируем их по двум векторам ошибок и положительных результатов
- Создаёт страницу с описанием результатов загрузки изображений

Встроенная функция ImageApp::read_field читает поля в Multipart-запросе и возвращает Some(field_name, field_data), если поле не пусто, и None, если значение поля пусто, в случае ошибки может вернуть ошибку

Встроенная функция ImageApp::process_put_image загружает картинку на сервер, при этом определяет, является ли поле запроса тектом или бинарными данными и вызывает соответствующую реализацию метода ImageApi::put_image. Таким образом, ImageApi::put_image получает на вход довольно чистые данные. Может вернуть None если вдруг данных нет. Если заявленно, что это текстовая информация, но при этом она не соотвутствует Utf-8, возвращает ошибку.

ImageApi

Невстроенная функция put_image загружает картинку на сервер. Функция принимает либо тектовые данные, либо бинарные. Если данные бинарные, то вызывается невстроенная функция upload_image. Если это текст и он Base64, то upload_image_base64, если нет, то если текст URL, то картинка скачивается функцией download_image, иначе возвращается ошибка.

Невстроенная функция upload_image принимает бинарные данные картинки, и записывает большую и миниатюру картинки в БД, в качестве БД используется директория на жёстком диске, а картинки пронумерованы. Следует отметить, что после каждого запуска информация из БД «стирается» и картинки снова начинают нумероваться с нуля – устройства сложной БД не требуется по заданию, да к тому же усложняет развёртывание примера.

Функция работает следующим образом:

- сохраняет картинку в БД, вызывая функция write_image
- Изменяет размер картинки, и сохраняет миниатюру под другим названием при помощи OpenCV, вызывая функцию ResizeImage
- Создаёт результат, повествующий о успешной загрузке картинки

Функция на Rust write_image записывает картинку на жёсткий диск.

Функция на Rust resize_image создаёт миниатюру картинки и записывает её на диск при помощи высокоуровневой обёртки над OpenCV

Функция на Rust is_image_base64 проверяет, является ли текст base64

Встроенная функция upload_image_base64 получает на вход строку в base64, декодирует base64 в бинарные данные и запускает функцию upload_image, поэтому возвращает либо сообщение о успешной загрузке, либо ошибку.

Встроенная фукнция download_image умеет скачивать картинку по данному URL. При помощи специального Http-клиента в библиотеке actix производится запрос другому серверу, при этом данный лёгкий поток засыпает и просыпается, когда приходит ответ. Далее либо картинка загружается при помощи функции upload_image, либо возвращается ошибка.

Невстроенная функция get_image возвращает бинарные данные картинки(полной и миниатюры в зависимости от URI), но существует лишь что бы обернуть встроенную функцию load_image в Box.

Встроенная функция load_image загружает картинку из БД(просто из директории) и возвращает либо данные, либо ошибку. Эта функция может быть доработана для использования полноценных БД

Невстроенная функция get_images_list возвращает список идентификаторов существующих картинок в БД, может быть расширена для использования полноценной БД

OpenCV_sys

OpenCV sys является FFI обёрткой над OpenCL. Следует отметить, что использовать OpenCL из Rust стало затруднительно, поскольку разработчики решили выпилить C-шный API, а C++ Rust использовать не может.. Но кое-как удалось заставить это работать. Что-бы полноценно использовать OpenCV, необходимо сделать обёртку над C++ интерфейсом, которая имеет C-шный API, либо использовать готовый crate, «но это уже совсем другая история».

Оборачивает функции:
- cvLoadImage
- cvSaveImage
- cvResize
- cvCreateImage
- cvGetErrStatus

OpenCV

OpenCV является высокоуровневой обёрткой над OpenCL, использующей крейт OpecCV_sys и предоставляющее высокоуровневое Rust-овское API

Собственно предоставляет структуру Image, которая имеет следующие методы:

- create
- open
- save
- resize

Тесты

В файле image_api.rs перечислено несколько тестов, в том числе и для методов API-футур, причём эти методы запускаются не параллельно, потому что могут возникать конфликты при работе с файлами или БД.

Библиотеки-зависимости

Необходимо установить OpenCL и OpenSSL, настроить переменные окружения.