بسم الله الرحمن الرحیم

پروژه: برنامه نویسی سیستمی.

نام مدرس: جناب آقای محمد احمدزاده

تهیه کننده: سرکار خانم زهرا صالحی

پاییز:۱۴۰۳

۱.برنامه نویسی Multi-Threading در زبان rust را با ذکر یک مثال توضیح دهید؟

برنامه‌نویسی چند رشته‌ای (Multi-Threading) در زبان Rust به شما این امکان را می‌دهد که چندین کار را به طور همزمان اجرا کنید، که می‌تواند به بهبود کارایی و سرعت برنامه کمک کند. Rust به خاطر سیستم مدیریت حافظه و ایمنی در برابر خطاهای همزمان (Concurrency) شناخته شده است.

در Rust، برای ایجاد رشته‌های جدید می‌توانید از تابع thread::spawn استفاده کنید. این تابع یک رشته جدید ایجاد می‌کند و یک تابع یا کلاسی را به آن می‌دهد تا اجرا کند. همچنین، Rust از مفهوم Ownership و Borrowing برای جلوگیری از مشکلات مربوط به دسترسی همزمان به داده‌ها استفاده می‌کند.

مثال ساده

در این مثال، ما یک برنامه ساده می‌نویسیم که دو رشته را ایجاد می‌کند. هر رشته یک شمارش را انجام می‌دهد و در نهایت نتیجه را چاپ می‌کند.

Rust

Use std::thread;

Use std::time::Duration;

Fn main() {

// ایجاد یک رشته جدید

Let handle1 = thread::spawn(|| {

For I in 1..5 {

Println!("رشته 1: {}", i);

Thread::sleep(Duration::from\_millis(100)); // خوابیدن برای 100 میلی‌ثانیه

}

});

// ایجاد یک رشته دیگر

Let handle2 = thread::spawn(|| {

For I in 1..5 {

Println!("رشته 2: {}", i);

Thread::sleep(Duration::from\_millis(150)); // خوابیدن برای 150 میلی‌ثانیه

}

});

// منتظر ماندن تا هر دو رشته به پایان برسند

Handle1.join().unwrap();

Handle2.join().unwrap();

}

توضیحات کد

وارد کردن ماژول‌ها: ابتدا ماژول‌های thread و time را وارد می‌کنیم.

ایجاد رشته‌ها: با استفاده از thread::spawn دو رشته ایجاد می‌کنیم. هر رشته یک حلقه دارد که اعداد 1 تا 4 را چاپ می‌کند و بین هر چاپ، برای مدت مشخصی می‌خوابد.

انتظار برای پایان رشته‌ها: با استفاده از متد join، برنامه منتظر می‌ماند تا هر دو رشته به پایان برسند. این به ما اطمینان می‌دهد که تمام کارها قبل از پایان برنامه انجام شده‌اند.

نکات مهم

ایمنی: Rust به شما اطمینان می‌دهد که دسترسی به داده‌ها در حین اجرای چند رشته‌ای ایمن باشد. برای اشتراک‌گذاری داده‌ها بین رشته‌ها، می‌توانید از Arc (Atomic Reference Counted) و Mutex (Mutual Exclusion) استفاده کنید.

مدیریت خطا: در این مثال، از unwrap() برای مدیریت خطا استفاده شده است. در برنامه‌های واقعی، بهتر است که خطاها را به طور مناسب مدیریت کنید.

۲.Lazy Loading چیست؟ را با ذکر مثال توضیح دهید؟

Lazy Loading (بارگذاری تنبل) یک الگوی طراحی است که به منظور بهینه‌سازی عملکرد و کاهش زمان بارگذاری در برنامه‌های نرم‌افزاری و وب‌سایت‌ها استفاده می‌شود. در این روش، منابع (مانند تصاویر، داده‌ها یا ماژول‌ها) تنها زمانی بارگذاری می‌شوند که واقعاً به آن‌ها نیاز است، به جای اینکه همه منابع به‌طور همزمان بارگذاری شوند.

مزایای Lazy Loading:

کاهش زمان بارگذاری اولیه: با بارگذاری فقط منابع ضروری، زمان بارگذاری صفحه کاهش می‌یابد.

کاهش مصرف پهنای باند: فقط منابعی که کاربر به آن‌ها نیاز دارد بارگذاری می‌شوند، که این باعث کاهش مصرف داده می‌شود.

بهبود تجربه کاربری: کاربران می‌توانند سریع‌تر به محتوای اولیه دسترسی پیدا کنند.

مثال:

فرض کنید شما یک وب‌سایت دارید که شامل یک گالری از تصاویر است. اگر تمام تصاویر به‌طور همزمان بارگذاری شوند، ممکن است زمان بارگذاری صفحه بسیار طولانی شود. با استفاده از Lazy Loading، می‌توانید به گونه‌ای برنامه‌نویسی کنید که فقط تصاویری که در حال حاضر در نمای کاربر قرار دارند بارگذاری شوند و بقیه تصاویر تنها زمانی بارگذاری شوند که کاربر به پایین صفحه اسکرول کند.

پیاده‌سازی Lazy Loading در JavaScript:

Html

<img src="placeholder.jpg" data-src="image1.jpg" class="lazy">

<img src="placeholder.jpg" data-src="image2.jpg" class="lazy">

<img src="placeholder.jpg" data-src="image3.jpg" class="lazy">

Javascript

Document.addEventListener("DOMContentLoaded", function() {

Const lazyImages = document.querySelectorAll('.lazy');

Const lazyLoad = (image) => {

Const src = image.getAttribute('data-src');

If (src) {

Image.src = src;

Image.classList.remove('lazy');

}

};

Const observer = new IntersectionObserver((entries) => {

Entries.forEach(entry => {

If (entry.isIntersecting) {

lazyLoad(entry.target);

observer.unobserve(entry.target);

}

});

});

lazyImages.forEach(image => {

observer.observe(image);

});

});

توضیح کد:

HTML: تصاویر با data-src مشخص شده‌اند و در ابتدا src به یک تصویر placeholder تنظیم شده است.

JavaScript: با استفاده از IntersectionObserver، برنامه به دنبال تصاویری می‌گردد که در حال حاضر در نمای کاربر قرار دارند. زمانی که یک تصویر در نمای کاربر قرار می‌گیرد، data-src آن به src تبدیل می‌شود و تصویر بارگذاری می‌شود.

نتیجه‌گیری:

Lazy Loading می‌تواند به طور قابل توجهی عملکرد وب‌سایت‌ها و برنامه‌ها را بهبود بخشد و تجربه کاربری بهتری را ارائه دهد.

3.ساختمان داده AVL Tree را در زبان rust پیاده سازی نمایید؟

در اینجا یک پیاده‌سازی ساده از درخت AVL در زبان Rust را برای شما ارائه می‌دهم. درخت AVL یک نوع درخت جستجوی دودویی متوازن است که در آن تفاوت ارتفاع زیر درخت‌های چپ و راست هر گره حداکثر ۱ است. این ویژگی باعث می‌شود که عملیات‌های جستجو، درج و حذف در این درخت‌ها به طور متوسط و در بدترین حالت زمان O(log n) داشته باشند.

پیاده‌سازی درخت AVL در Rust

Rust

#[derive(Debug)]

Struct Node {

Key: i32,

Height: i32,

Left: Option<Box<Node>>,

Right: Option<Box<Node>>,

}

Impl Node {

Fn new(key: i32) -> Self {

Node {

Key,

Height: 1,

Left: None,

Right: None,

}

}

}

Pub struct AVLTree {

Root: Option<Box<Node>>,

}

Impl AVLTree {

Pub fn new() -> Self {

AVLTree { root: None }

}

Pub fn insert(&mut self, key: I32) {

Self.root = self.insert\_node(self.root.take(), key);

}

Fn insert\_node(&mut self, node: Option<Box<Node>>, key: i32) -> Option<Box<Node>> {

Let mut node = match node {

Some(n) => n,

None => return Some(Box::new(Node::new(key))),

};

If key < node.key {

Node.left = self.insert\_node(node.left, key);

} else if key > node.key {

Node.right = self.insert\_node(node.right, key);

} else {

Return Some(node); // Duplicate keys are not allowed

}

Node.height = 1 + std::cmp::max(Self::get\_height(&node.left), Self::get\_height(&node.right));

Let balance = Self::get\_balance(&node);

// Rotations

If balance > 1 && key < node.left.as\_ref().unwrap().key {

Return Some(Self::rotate\_right(node));

}

If balance < -1 && key > node.right.as\_ref().unwrap().key {

Return Some(Self::rotate\_left(node));

}

If balance > 1 && key > node.left.as\_ref().unwrap().key {

Node.left = Some(Self::rotate\_left(node.left.take().unwrap()));

Return Some(Self::rotate\_right(node));

}

If balance < -1 && key < node.right.as\_ref().unwrap().key {

Node.right = Some(Self::rotate\_right(node.right.take().unwrap()));

Return Some(Self::rotate\_left(node));

}

Some(node)

}

Fn rotate\_left(node: Box<Node>) -> Box<Node> {

Let mut new\_root = node.right.unwrap();

Node.right = new\_root.left.take();

New\_root.left = Some(node);

New\_root.height = 1 + std::cmp::max(Self::get\_height(&new\_root.left), Self::get\_height(&new\_root.right)));

New\_root

}

Fn rotate\_right(node: Box<Node>) -> Box<Node> {

Let mut new\_root = node.left.unwrap();

Node.left = new\_root.right.take();

New\_root.right = Some(node);

New\_root.height = 1 + std::cmp::max(Self::get\_height(&new\_root.left), Self::get\_height(&new\_root.right)));

New\_root

}

Fn get\_height(node: &Option<Box<Node>>) -> i32 {

Match node {

Some(n) => n.height,

None => 0,

}

}

Fn get\_balance(node: &Box<Node>) -> i32 {

Self::get\_height(&node.left) - Self::get\_height(&node.right)

}

Pub fn in\_order(&self) {

Self.in\_order\_helper(&self.root);

}

Fn in\_order\_helper(&self, node: &OptIon<Box<Node>>) {

If let Some(n) = node {

Self.in\_order\_helper(&n.left);

Println!("{}", n.key);

Self.in\_order\_helper(&n.right);

}

}

}

Fn main() {

Let mut avl\_tree = AVLTree::new();

Avl\_tree.insert(10);

Avl\_tree.insert(20);

Avl\_tree.insert(30);

Avl\_tree.insert(40);

Avl\_tree.insert(50);

Avl\_tree.insert(25);

Println!("In-order traversal of the constructed AVL tree:");

Avl\_tree.in\_order();

}

توضیحات

ساختار Node: هر گره شامل یک کلید، ارتفاع و دو اشاره‌گر به فرزندان چپ و راست است.

ساختار AVLTree: شامل یک اشاره‌گر به ریشه درخت است.

عملیات درج: متد insert برای درج کلید جدید در درخت استفاده می‌شود. این متد از تابع کمکی insert\_node استفاده می‌کند که گره‌ها را به صورت بازگشتی درج می‌کند و در صورت نیاز چرخش‌ها را انجام می‌دهد.

چرخش‌ها: چرخش‌های چپ و راست برای حفظ توازن درخت استفاده می‌شوند.

توازن درخت: با استفاده از ارتفاع زیر درخت‌ها و محاسبه توازن، چرخش‌های لازم انجام می‌شود.

پیمایش درخت: متد in\_order برای پیمایش درخت به صورت مرتب استفاده می‌شود.

۴.یک سرویس RESTful APIجهت پردازش درخواست های JSON بنویسید؟

برای ایجاد یک سرویس RESTful API که درخواست‌های JSON را پردازش کند، می‌توانیم از فریم‌ورک‌های مختلفی مانند Flask (برای پایتون) یا Express (برای Node.js) استفاده کنیم. در اینجا، یک مثال ساده با استفاده از Flask در پایتون ارائه می‌دهم.

مراحل ایجاد یک RESTful API با Flask:

1. نصب Flask

ابتدا باید Flask را نصب کنید. اگر هنوز آن را نصب نکرده‌اید، می‌توانید از pip استفاده کنید:

Bash

Pip install Flask

1. ایجاد یک فایل Python

یک فایل جدید به نام app.py بسازید و کد زیر را در آن قرار دهید:

Python

From flask import Flask, request, jsonify

App = Flask(\_\_name\_\_)

# داده‌های نمونه

Data = []

# Endpoint برای دریافت داده‌ها

@app.route('/api/data', methods=['GET'])

Def get\_data():

Return jsonify(data), 200

# Endpoint برای اضافه کردن داده‌ها

@app.route('/api/data', methods=['POST'])

Def add\_data():

If request.is\_json:

New\_data = request.get\_json()

Data.append(new\_data)

Return jsonify(new\_data), 201

Return jsonify({"error": "Request must be JSON"}), 400

# Endpoint برای حذف داده‌ها

@app.route('/api/data/<int:data\_id>', methods=['DELETE'])

Def delete\_data(data\_id):

If 0 <= data\_id < len(data):

Deleted\_data = data.pop(data\_Id)

Return jsonify(deleted\_data), 200

Return jsonify({"error": "Data not found"}), 404

If \_\_name\_\_ == '\_\_main\_\_':

App.run(debug=True)

1. توضیحات کد

ایجاد اپلیکیشن Flask: با استفاده از Flask(\_\_name\_\_) یک اپلیکیشن جدید ایجاد می‌کنیم.

داده‌های نمونه: یک لیست خالی برای ذخیره داده‌ها ایجاد کرده‌ایم.

Endpoint برای دریافت داده‌ها: با استفاده از متد GET، داده‌های موجود را برمی‌گردانیم.

Endpoint برای اضافه کردن داده‌ها: با استفاده از متد POST، داده‌های جدید را از درخواست JSON دریافت کرده و به لیست اضافه می‌کنیم.

Endpoint برای حذف داده‌ها: با استفاده از متد DELETE، داده‌های مشخص شده را حذف می‌کنیم.

اجرا: در انتها، اپلیکیشن را با app.run(debug=True) اجرا می‌کنیم.

1. اجرای API

برای اجرای API، در ترمینال به دایرکتوری که فایل app.py در آن قرار دارد بروید و دستور زیر را وارد کنید:

Bash

Python app.py

API شما اکنون در آدرس http://127.0.0.1:5000/api/data در دسترس است.

1. تست API

می‌توانید از ابزارهایی مانند Postman یا curl برای تست API استفاده کنید. به عنوان مثال، برای اضافه کردن داده می‌توانید از دستور curl زیر استفاده کنید:

Bash

Curl -X POST http://127.0.0.1:5000/api/data -H "Content-Type: application/json" -d '{"name": "example", "value": 42}'

نتیجه‌گیری

با استفاده از کد بالا، یک RESTful API ساده برای پردازش درخواست‌های JSON ایجاد کردید. می‌توانید این API را گسترش دهید و ویژگی‌های بیشتری به آن اضافه کنید، مانند اعتبارسنجی داده‌ها، مدیریت خطاها و غیره.

۵.یک سرویس ساده جهت پردازش درخواست های مبتنی بر Web Assembly را بنویسید؟

برای ایجاد یک سرویس ساده جهت پردازش درخواست‌های مبتنی بر WebAssembly (Wasm)، می‌توانیم از فناوری‌های مختلفی استفاده کنیم. در اینجا یک راهنمای گام به گام برای ساخت یک سرویس ساده با استفاده از Node.js و WebAssembly ارائه می‌شود.

گام ۱: نصب Node.js

ابتدا اطمینان حاصل کنید که Node.js بر روی سیستم شما نصب شده است. می‌توانید آن را از سایت رسمی Node.js دانلود و نصب کنید.

گام ۲: ایجاد پروژه جدید

یک پوشه جدید برای پروژه خود ایجاد کنید و به آن بروید:

Bash

Mkdir wasm-service

Cd wasm-service

سپس یک فایل package.json جدید ایجاد کنید:

Bash

Npm init -y

گام ۳: نصب وابستگی‌ها

برای این پروژه، ما به express نیاز داریم که یک فریمورک وب برای Node.js است. آن را نصب کنید:

Bash

Npm install express

گام ۴: ایجاد فایل WebAssembly

برای این مثال، ما یک فایل ساده WebAssembly ایجاد خواهیم کرد. فرض کنید که ما یک تابع جمع ساده داریم. ابتدا یک فایل add.c ایجاد کنید:

C

// add.c

Int add(int a, int b) {

Return a + b;

}

سپس، این فایل را با استفاده از Emscripten به WebAssembly تبدیل کنید. ابتدا Emscripten را نصب کنید و سپس دستور زیر را اجرا کنید:

Bash

Emcc add.c -o add.wasm -s WASM=1 -s EXPORTED\_FUNCTIONS='["\_add"]' -s EXTRA\_EXPORTED\_RUNTIME\_METHODS='["cwrap"]'

گام ۵: ایجاد سرویس Express

حالا یک فایل server.js ایجاد کنید و کد زیر را در آن قرار دهید:

Javascript

Const express = require('express');

Const fs = require('fs');

Const path = require('path');

Const app = express();

Const port = 3000;

// بارگذاری WebAssembly

Let wasmModule;

Const loadWasm = async () => {

Const buffer = fs.readFileSync(path.resolve(\_\_dirname, 'add.wasm'));

Const module = await WebAssembly.instantiate(buffer);

Return module.instance.exports;

};

App.use(express.json());

App.post('/add', async (req, res) => {

Const { a, b } = req.body;

If (typeof a !== 'number' || typeof b !== 'number') {

Return res.status(400).json({ error: 'Invalid Input' });

}

If (!wasmModule) {

wasmModule = await loadWasm();

}

Const result = wasmModule.\_add(a, b);

Res.json({ result });

});

App.listen(port, () => {

Console.log(`Server running at http://localhost:${port}`);

});

گام ۶: اجرای سرویس

حالا می‌توانید سرویس را با استفاده از دستور زیر اجرا کنید:

Bash

Node server.js

گام ۷: تست سرویس

برای تست سرویس، می‌توانید از ابزارهایی مانند Postman یا curl استفاده کنید. به عنوان مثال، با استفاده از curl:

Bash

Curl -X POST http://localhost:3000/add -H "Content-Type: application/json" -d '{"a": 5, "b": 3}'

این درخواست باید پاسخ زیر را برگرداند:

Json

{"result":8}

نتیجه‌گیری

دریک سرویس ساده با استفاده از Node.js و WebAssembly ایجاد کردیم که می‌تواند درخواست‌های جمع دو عدد را پردازش کند.

۶.برنامه ای ب زبانrust بنویسید که عملیات CRUD را بروی یک پایگاه داده انجام دهد؟

برای نوشتن یک برنامه CRUD (ایجاد، خواندن، به‌روزرسانی و حذف) به زبان Rust با استفاده از یک پایگاه داده، می‌توانیم از کتابخانه‌هایی مانند Diesel یا SQLx استفاده کنیم. در اینجا، یک مثال ساده با استفاده از Diesel ارائه می‌دهیم.

مراحل ایجاد برنامه CRUD با Diesel

نصب وابستگی‌ها: ابتدا باید Diesel و SQLite را به پروژه خود اضافه کنید. فایل Cargo.toml شما باید به شکل زیر باشد:

Toml

[dependencies]

Diesel = { version = "2.0", features = ["sqlite"] }

Dotenv = "0.15"

ایجاد پایگاه داده: برای ایجاد پایگاه داده، باید یک فایل .env در ریشه پروژه ایجاد کنید و نام پایگاه داده را مشخص کنید:

Ini

DATABASE\_URL=db.sqlite

ایجاد مدل و جدول: برای ایجاد یک جدول و مدل، از دستورات زیر استفاده کنید. ابتدا با اجرای دستور زیر، یک فایل migration ایجاد کنید:

Bash

Diesel migration generate create\_users

سپس در پوشه migrations، فایل up.sql را به شکل زیر ویرایش کنید:

Sql

CREATE TABLE users (

Id INTEGER PRIMARY KEY AUTOINCREMENT,

Name TEXT NOT NULL,

Email TEXT NOT NULL UNIQUE

);

و در فایل down.sql:

Sql

DROP TABLE users;

حالا با اجرای دستور زیر، migration را اجرا کنید:

Bash

Diesel migration run

نوشتن کد CRUD: حالا می‌توانیم کد CRUD را بنویسیم. در فایل src/main.rs، کد زیر را قرار دهید:

Rust

#[macro\_use]

Extern crate diesel;

Extern crate dotenv;

Use diesel::prelude::\*;

Use dotenv::dotenv;

Use std::env;

Mod schema {

Table! {

Users (id) {

Id -> Integer,

Name -> Text,

Email -> Text,

}

}

}

#[derive(Queryable, Insertable, AsChangeset, Debug)]

#[table\_name = "users"]

Struct User {

Id: i32,

Name: String,

Email: String,

}

Fn establish\_connection() -> SqliteConnection {

Dotenv().ok();

Let database\_url = env::var("DATABASE\_URL").expect("DATABASE\_URL must be set");

SqliteConnection::establish(&database\_url).expect(&format!("Error connecting to {}", database\_url))

}

Fn create\_user(conn: &SqliteConnection, name: &str, email: &str) -> User {

Use schema::users;

Let new\_user = User {

Id: 0,

Name: name.to\_string(),

Email: email.to\_string(),

};

Diesel::insert\_into(users::table)

.values(&new\_user)

.execute(conn)

.expect("Error inserting new user");

New\_user

}

Fn read\_users(conn: &SqliteConnection) -> Vec<User> {

Use schema::users::dsl::\*;

Users.load::<User>(conn).expect("Error loading users")

}

Fn update\_user(conn: &SqliteConnection, user\_id: i32, new\_name: &str) {

Use schema::users::dsl::\*;

Diesel::update(users.find(user\_id))

.set(name.eq(new\_name))

.execute(conn)

.expect("Error updating user");

}

Fn delete\_user(conn: &SqliteConnection, user\_id: i32) {

Use schema::users::dsl::\*;

Diesel::delete(users.find(user\_id))

.execute(conn)

.expect("Error deleting user");

}

Fn main() {

Let connection = establish\_connection();

// ایجاد کاربر جدید

Let user = create\_user(&connection, "Ali", "ali@example.com");

Println!("Created User: {:?}", user);

// خواندن کاربران

Let users = read\_users(&connection);

Println!("All Users: {:?}", users);

// به‌روزرسانی کاربر

Update\_user(&connection, user.id, "Ali Reza");

Println!("Updated User: {:?}", read\_users(&connection));

// حذف کاربر

Delete\_user(&connection, user.Id);

Println!("Users after deletion: {:?}", read\_users(&connection));

}

توضیحات کد:

مدل: مدل User برای نگهداری اطلاعات کاربران ایجاد شده است.

اتصال به پایگاه داده: تابع establish\_connection برای اتصال به پایگاه داده استفاده می‌شود.

عملیات CRUD: توابع create\_user, read\_users, update\_user, و delete\_user برای انجام عملیات CRUD تعریف شده‌اند.

اجرای برنامه:

برای اجرای برنامه، از دستور زیر استفاده کنید:

Bash

Cargo run

این برنامه یک کاربر جدید ایجاد می‌کند، همه کاربران را می‌خواند، یک کاربر را به‌روزرسانی می‌کند و سپس آن را حذف می‌کند.

نکته:

اطمینان حاصل کنید که Diesel CLI را نصب کرده‌اید تا بتوانید migration‌ها را اجرا کنید. برای نصب آن، می‌توانید از دستور زیر استفاده کنید:

Bash

Cargo install diesel\_cli –no-default-features –features sqlite

این کد یک نمونه ساده از CRUD در Rust با استفاده از Diesel و SQLite است. برای پروژه‌های بزرگتر و پیچیده‌تر، ممکن است نیاز به مدیریت خطاها و بهینه‌سازی‌های بیشتری داشته باشید.

۷.کتابخانه های Parsing را در زبان rust را نام ببرید ؟و عملکرد یک parser را در قالب یک مثال توضیح دهید؟

در زبان Rust، چندین کتابخانه برای Parsing وجود دارد که می‌توانند به شما در تجزیه و تحلیل داده‌ها کمک کنند. در زیر به برخی از این کتابخانه‌ها اشاره می‌کنم:

کتابخانه‌های Parsing در Rust

Nom: این کتابخانه یک پارسر ترکیبی است که به شما امکان می‌دهد تا با استفاده از توابع کوچک و قابل ترکیب، داده‌ها را تجزیه و تحلیل کنید. nom به خاطر سرعت و کارایی‌اش شناخته شده است.

Pest: این کتابخانه یک پارسر مبتنی بر گرامر است که به شما امکان می‌دهد گرامر خود را به صورت واضح و خوانا تعریف کنید. Pest برای تجزیه و تحلیل زبان‌های پیچیده مناسب است.

Regex: اگر به دنبال تجزیه و تحلیل رشته‌های متنی با استفاده از عبارات منظم هستید، کتابخانه regex می‌تواند مفید باشد. این کتابخانه به شما اجازه می‌دهد الگوهای پیچیده را برای تجزیه و تحلیل متن تعریف کنید.

Serde: این کتابخانه بیشتر برای سریال‌سازی و دی‌سریال‌سازی داده‌ها استفاده می‌شود، اما می‌تواند برای تجزیه و تحلیل فرمت‌های داده‌ای مانند JSON و YAML نیز مفید باشد.

عملکرد یک Parser

یک پارسر وظیفه تجزیه و تحلیل داده‌ها را بر عهده دارد. به عنوان مثال، فرض کنید که می‌خواهیم یک رشته متنی ساده را که شامل اعداد و عملگرهای ریاضی است، تجزیه کنیم. هدف ما این است که یک درخت تجزیه (Parse Tree) از این داده‌ها بسازیم.

مثال: تجزیه یک عبارت ریاضیغ

فرض کنید ما یک رشته ورودی داریم:

Plaintext

"3 + 5 \* 2"

ما می‌خواهیم این عبارت را به صورت زیر تجزیه کنیم:

شناسایی اعداد و عملگرها

تعیین ترتیب عمل (در اینجا، ضرب قبل از جمع انجام می‌شود)

پیاده‌سازی با استفاده از nom

در زیر یک مثال ساده از استفاده از کتابخانه nom برای تجزیه یک عبارت ریاضی آورده شده است:

Rust

Use nom::{

Iresult,

Character::complete::{digit1, char},

Combinator::map,

Sequence::preceded,

};

Fn parse\_number(input: &str) -> Iresult<&str, i32> {

Map(digit1, |s: &str| s.parse::<I32>().unwrap())(Input)

}

Fn parse\_expression(input: &str) -> Iresult<&str, i32> {

Let (input, left) = parse\_number(Input)?;

Let (input, \_) = char(' ')(Input)?;

Let (input, op) = char('+')(Input)?;

Let (input, \_) = char(' ')(Input)?;

Let (input, right) = parse\_number(Input)?;

Let result = left + right; // اینجا فقط جمع را انجام می‌دهیم

Ok((input, result))

}

Fn main() {

Let expression = "3 + 5";

Match parse\_expression(expression) {

Ok((remaining, result)) => {

Println!("Result: {}, Remaining: {}", result, remaining);

}

Err(err) => {

Println!("Error: {:?}", err);

}

}

}

توضیحات کد

Parse\_number: این تابع یک عدد را از ورودی تجزیه می‌کند.

Parse\_expression: این تابع یک عبارت ساده (در اینجا فقط جمع) را تجزیه می‌کند. ابتدا عدد اول را تجزیه می‌کند، سپس عملگر و عدد دوم را تجزیه می‌کند و در نهایت نتیجه جمع را محاسبه می‌کند.

Main: در اینجا یک رشته ورودی به تابع parse\_expression داده می‌شود و نتیجه آن چاپ می‌شود.

۸.عملکرد کتابخانه Syslnfo در زبان rust چیست؟ با ذکر یک مثال ساده توضیح دهید؟

کتابخانه sysinfo در زبان Rust برای جمع‌آوری اطلاعات مربوط به سیستم عامل و منابع سیستم طراحی شده است. این کتابخانه به شما این امکان را می‌دهد که اطلاعاتی مانند استفاده از CPU، حافظه، دیسک، و فرایندهای در حال اجرا را به راحتی بدست آورید.

عملکرد کتابخانه sysinfo

کتابخانه sysinfo می‌تواند اطلاعات زیر را ارائه دهد:

اطلاعات CPU: تعداد هسته‌ها، بار CPU و غیره.

اطلاعات حافظه: مقدار حافظه کل، حافظه آزاد، و حافظه استفاده شده.

اطلاعات دیسک: فضای دیسک و وضعیت آن.

فرایندها: لیست فرایندهای در حال اجرا و اطلاعات مربوط به هر کدام.

مثال ساده

در زیر یک مثال ساده از استفاده از کتابخانه sysinfo برای نمایش اطلاعات حافظه و CPU آورده شده است:

Rust

Use sysinfo::{System, SystemExt, ProcessorExt};

Fn main() {

// ایجاد یک شیء از نوع System

Let mut system = System::new\_all();

// به‌روزرسانی اطلاعات سیستم

System.refresh\_all();

// نمایش اطلاعات حافظه

Println!("Total memory: {} KB", system.total\_memory());

Println!("Used memory: {} KB", system.used\_memory());

Println!("Free memory: {} KB", system.free\_memory());

// نمایش اطلاعات CPU

For processor in system.processors() {

Println!("Processor: {} - Usage: {}%", processor.name(), processor.cpu\_usage());

}

}

توضیح کد

وارد کردن کتابخانه: با استفاده از use sysinfo::{System, SystemExt, ProcessorExt}; کتابخانه sysinfo را وارد می‌کنیم.

ایجاد شیء System: با let mut system = System::new\_all(); یک شیء از نوع System ایجاد می‌کنیم که می‌تواند اطلاعات مختلف سیستم را جمع‌آوری کند.

به‌روزرسانی اطلاعات: با system.refresh\_all(); اطلاعات سیستم را به‌روز می‌کنیم.

نمایش اطلاعات حافظه: با استفاده از توابع total\_memory(), used\_memory(), و free\_memory() اطلاعات مربوط به حافظه را نمایش می‌دهیم.

نمایش اطلاعات CPU: با یک حلقه for بر روی پردازنده‌ها، نام هر پردازنده و میزان استفاده از آن را نمایش می‌دهیم.

نتیجه‌گیری

کتابخانه sysinfo ابزاری قدرتمند برای جمع‌آوری اطلاعات سیستم در زبان Rust است. با استفاده از این کتابخانه، می‌توانید به راحتی اطلاعات مفیدی از سیستم خود بدست آورید و در برنامه‌های خود استفاده کنید.

۹.برنامه ای برای انجام یک پردازش ساده بر روی یک Image بنویسید؟

برای انجام یک پردازش ساده بر روی یک تصویر، می‌توانیم از کتابخانه‌های محبوبی مانند OpenCV یا PIL (Pillow) در زبان برنامه‌نویسی Python استفاده کنیم. در اینجا، یک برنامه ساده نوشته می‌شود که یک تصویر را بارگذاری کرده و آن را به تصویر سیاه و سفید تبدیل می‌کند.

مراحل انجام کار:

نصب کتابخانه‌های مورد نیاز: ابتدا باید کتابخانه Pillow یا OpenCV را نصب کنید. برای این کار می‌توانید از pip استفاده کنید:

Bash

Pip install pillow

یا برای OpenCV:

Bash

Pip install opencv-python

نوشتن کد: در اینجا یک مثال با استفاده از Pillow آورده شده است:

Python

From PIL import Image

# بارگذاری تصویر

Image\_path = 'path/to/your/Image.jpg' # مسیر تصویر خود را وارد کنید

Image = Image.open(image\_path)

# تبدیل تصویر به سیاه و سفید

Bw\_image = image.convert('L')

# ذخیره تصویر جدید

Bw\_image.save('path/to/save/black\_and\_white\_Image.jpg') # مسیر ذخیره تصویر سیاه و سفید

توضیحات کد:

بارگذاری تصویر: با استفاده از Image.open() تصویر را بارگذاری می‌کنیم.

تبدیل به سیاه و سفید: با استفاده از متد convert('L') تصویر به حالت سیاه و سفید تبدیل می‌شود.

ذخیره تصویر: تصویر جدید با استفاده از save() ذخیره می‌شود.

اگر از OpenCV استفاده می‌کنید:

در اینجا یک مثال مشابه با استفاده از OpenCV آورده شده است:

Python

Import cv2

# بارگذاری تصویر

Image\_path = 'path/to/your/Image.jpg' # مسیر تصویر خود را وارد کنید

Image = cv2.imread(image\_path)

# تبدیل تصویر به سیاه و سفید

Bw\_image = cv2.cvtColor(image, cv2.COLOR\_BGR2GRAY)

# ذخیره تصویر جدید

Cv2.imwrite('path/to/save/black\_and\_white\_Image.jpg', bw\_image) # مسیر ذخیره تصویر سیاه و سفید

نکات مهم:

مسیر تصویر: حتماً مسیر صحیح تصویر را وارد کنید.

فرمت تصویر: می‌توانید فرمت تصویر ذخیره شده را تغییر دهید (مانند PNG، JPEG و غیره) با تغییر پسوند در نام فایل ذخیره.

این برنامه‌ها یک پردازش ساده بر روی تصویر انجام می‌دهند، اما می‌توانید با افزودن مراحل دیگر، پردازش‌های پیچیده‌تری را نیز انجام دهید.

۱۰.برنامه ای برای انجام یک پروژه ای ساده بر روی یک Video بنویسید؟

برای انجام یک پروژه ساده بر روی یک ویدیو، می‌توانیم یک برنامه را طراحی کنیم که شامل مراحل زیر باشد. به عنوان مثال، فرض کنید پروژه ما شامل ویرایش یک ویدیو ساده است که شامل برش، افزودن متن و فیلترهای ساده می‌شود. ما می‌توانیم از زبان برنامه‌نویسی Python و کتابخانه‌های مرتبط مانند OpenCV و MoviePy استفاده کنیم. در ادامه، مراحل پروژه را توضیح می‌دهم:

مراحل پروژه ویرایش ویدیو

1. نصب کتابخانه‌های مورد نیاز

برای شروع، ابتدا باید کتابخانه‌های مورد نیاز را نصب کنید. می‌توانید از pip برای نصب استفاده کنید:

Bash

Pip install moviepy opencv-python

1. بارگذاری ویدیو

ابتدا باید ویدیویی که می‌خواهید ویرایش کنید را بارگذاری کنید. می‌توانید از MoviePy برای این کار استفاده کنید.

Python

From moviepy.editor import VideoFileClip

# بارگذاری ویدیو

Video = VideoFileClip("input\_video.mp4")

1. برش ویدیو

می‌توانید بخشی از ویدیو را برش دهید. به عنوان مثال، اگر بخواهید ویدیو را از ثانیه 10 تا 20 برش دهید:

Python

# برش ویدیو از ثانیه 10 تا 20

Video\_cropped = video.subclip(10, 20)

1. افزودن متن

می‌توانید متنی را به ویدیو اضافه کنید. برای مثال، یک متن در وسط ویدیو:

Python

From moviepy.editor import TextClip

# ایجاد متن

Text = TextClip("سلام، این یک ویدیو است!", fontsize=70, color='white')

Text = text.set\_position('center').set\_duration(video\_cropped.duration)

# ترکیب ویدیو و متن

Video\_with\_text = video\_cropped.set\_duration(video\_cropped.duration).fx(vfx.composite, text)

1. افزودن فیلتر

می‌توانید فیلترهای ساده‌ای به ویدیو اضافه کنید. به عنوان مثال، می‌توانید ویدیو را به سیاه و سفید تبدیل کنید:

Python

Import cv2

Import numpy as np

Def black\_and\_white(frame):

Return cv2.cvtColor(frame, cv2.COLOR\_BGR2GRAY)

Video\_bw = video\_with\_text.fl\_image(black\_and\_white)

1. ذخیره ویدیو

در نهایت، ویدیو ویرایش شده را ذخیره کنید:

Python

Video\_bw.write\_videofile("output\_video.mp4", codec='libx264')

۱۱.برنامه ای برای خواندن و نوشتن یک فایل CSV ساده بنویسید؟

برای خواندن و نوشتن یک فایل CSV ساده در زبان برنامه‌نویسی Python، می‌توانید از ماژول csv استفاده کنید. در اینجا یک برنامه ساده ارائه می‌شود که نحوه خواندن و نوشتن فایل CSV را نشان می‌دهد.

مراحل برنامه:

نوشتن داده‌ها به یک فایل CSV: ابتدا یک فایل CSV ایجاد می‌کنیم و داده‌هایی را به آن می‌نویسیم.

خواندن داده‌ها از فایل CSV: سپس داده‌ها را از فایل CSV می‌خوانیم و نمایش می‌دهیم.

کد نمونه:

Python

Import csv

# مرحله 1: نوشتن داده‌ها به یک فایل CSV

Def write\_to\_csv(filename):

# داده‌هایی که می‌خواهیم بنویسیم

Data = [

["نام", "سن", "شغل"],

["علی", 30, "مهندس"],

["زهرا", 25, "طراح"],

["مهدی", 35, "مدیر"]

]

# نوشتن داده‌ها به فایل CSV

With open(filename, mode='w', newline='', encoding='utf-8') as file:

Writer = csv.writer(file)

Writer.writerows(data)

# مرحله 2: خواندن داده‌ها از یک فایل CSV

Def read\_from\_csv(filename):

With open(filename, mode='r', encoding='utf-8') as file:

Reader = csv.reader(file)

For row in reader:

Print(row)

# نام فایل CSV

Filename = 'data.csv'

# نوشتن داده‌ها به فایل CSV

Write\_to\_csv(filename)

# خواندن داده‌ها از فایل CSV

Print("داده‌های خوانده شده از فایل CSV:")

Read\_from\_csv(filename)

توضیحات کد:

نوشتن به CSV:

تابع write\_to\_csv یک لیست از لیست‌ها (داده‌ها) را تعریف می‌کند.

با استفاده از csv.writer داده‌ها را به فایل CSV می‌نویسیم.

از newline='' استفاده می‌کنیم تا از ایجاد خطوط خالی اضافی جلوگیری کنیم.

خواندن از CSV:

تابع read\_from\_csv فایل CSV را باز می‌کند و با استفاده از csv.reader داده‌ها را می‌خواند.

هر ردیف از داده‌ها را با استفاده از print نمایش می‌دهیم.

نحوه استفاده:

کافی است کد را در یک فایل Python (مثلاً csv\_example.py) ذخیره کنید و آن را اجرا کنید. این کد یک فایل به نام data.csv ایجاد می‌کند و داده‌ها را در آن ذخیره می‌کند و سپس آن‌ها را می‌خواند و نمایش می‌دهد.

۱۲.مدل MVS را در قالب یک برنامه پیاده سازی کنید؟

مدل MVS (Model-View-Controller) یک الگوی طراحی نرم‌افزاری است که به تفکیک مسئولیت‌ها در یک برنامه کمک می‌کند. این مدل معمولاً در توسعه وب و برنامه‌های کاربردی استفاده می‌شود. در زیر یک پیاده‌سازی ساده از مدل MVS در یک برنامه وب با استفاده از زبان برنامه‌نویسی Python و فریم‌ورک Flask ارائه می‌شود.

مراحل پیاده‌سازی مدل MVS

نصب Flask: ابتدا باید Flask را نصب کنید. می‌توانید از pip برای نصب آن استفاده کنید:

Bash

Pip install Flask

ساختار پروژه: ساختار پروژه شما باید به شکل زیر باشد:

Stylus

My\_flask\_app/

├── app.py

├── models.py

├── views.py

└── templates/

└── index.html

مدل (models.py): در این فایل، می‌توانید مدل‌های داده‌ای خود را تعریف کنید. به عنوان مثال، یک مدل ساده برای یک کاربر:

Python

# models.py

Class User:

Def \_\_init\_\_(self, username, email):

Self.username = username

Self.email = email

نما (views.py): در این فایل، می‌توانید نماهای مختلف را تعریف کنید. به عنوان مثال، یک نمای ساده برای نمایش اطلاعات کاربر:

Python

# views.py

From flask import render\_template

From models import User

Def get\_user\_view():

User = User("JohnDoe", "john@example.com")

Return render\_template('index.html', user=user)

کنترلر (app.py): در این فایل، می‌توانید کنترلرها را تعریف کنید و برنامه Flask خود را راه‌اندازی کنید:

Python

# app.py

From flask import Flask

From views import get\_user\_view

App = Flask(\_\_name\_\_)

@app.route('/')

Def home():

Return get\_user\_view()

If \_\_name\_\_ == '\_\_main\_\_':

App.run(debug=True)

قالب (templates/index.html): در این فایل، می‌توانید قالب HTML خود را تعریف کنید:

Html

<!DOCTYPE html>

<html lang="en">

<head>

<meta charset="UTF-8">

<meta name="viewport" content="width=device-width, initial-scale=1.0">

<title>User Profile</title>

</head>

<body>

<h1>User Profile</h1>

<p>Username: {{ user.username }}</p>

<p>Email: {{ user.email }}</p>

</body>

</html>

توضیحات

مدل (Model): در اینجا، کلاس User مدل داده‌ای ما را نشان می‌دهد که شامل اطلاعات کاربر است.

نما (View): تابع get\_user\_view به عنوان نمای ما عمل می‌کند و اطلاعات کاربر را به قالب HTML ارسال می‌کند.

کنترلر (Controller): فایل app.py کنترلر است که مسیرها را تعریف می‌کند و به نماها دسترسی دارد.

اجرای برنامه

برای اجرای برنامه، کافی است به دایرکتوری پروژه بروید و دستور زیر را اجرا کنید:

Bash

Python app.py

سپس می‌توانید با مراجعه به http://127.0.0.1:5000/ در مرورگر خود، خروجی برنامه را مشاهده کنید.

این یک پیاده‌سازی ساده از مدل MVS بود.

۱۳.اصول SOLID را در زبان rust پیاده سازی نمایید؟

اصول SOLID مجموعه‌ای از اصول طراحی نرم‌افزار هستند که به بهبود قابلیت نگهداری و توسعه‌پذیری کد کمک می‌کنند. این اصول عبارتند از:

Single Responsibility Principle (SRP): هر کلاس یا ماژول باید تنها یک مسئولیت داشته باشد.

Open/Closed Principle (OCP): کلاس‌ها باید برای گسترش باز و برای اصلاح بسته باشند.

Liskov Substitution Principle (LSP): اشیاء باید بتوانند با اشیاء دیگر که از کلاس‌های پایه مشتق شده‌اند، جایگزین شوند بدون اینکه رفتار برنامه تغییر کند.

Interface Segregation Principle (ISP): بهتر است چندین رابط خاص به جای یک رابط عمومی بزرگ داشته باشیم.

Dependency Inversion Principle (DIP): ماژول‌های سطح بالا نباید به ماژول‌های سطح پایین وابسته باشند. هر دو باید به انتزاعات وابسته باشند.

حالا بیایید ببینیم چگونه می‌توان این اصول را در زبان Rust پیاده‌سازی کرد.

1. Single Responsibility Principle (SRP)

در Rust، می‌توانید از ساختارها (structs) و توابع برای جدا کردن مسئولیت‌ها استفاده کنید. هر ساختار باید یک مسئولیت خاص داشته باشد.

Rust

Struct User {

Name: String,

Email: String,

}

Impl User {

Fn new(name: &str, email: &str) -> User {

User {

Name: name.to\_string(),

Email: email.to\_string(),

}

}

}

Struct UserService;

Impl UserService {

Fn create\_user(&self, name: &str, email: &str) -> User {

User::new(name, email)

}

}

1. Open/Closed Principle (OCP)

در Rust می‌توانید از traitها برای گسترش رفتارها استفاده کنید. این امکان را می‌دهد که کد را بدون تغییر در کلاس‌های موجود گسترش دهید.

Rust

Trait Shape {

Fn area(&self) -> f64;

}

Struct Circle {

Radius: f64,

}

Impl Shape for Circle {

Fn area(&self) -> f64 {

Std::f64::consts::PI \* self.radius \* self.radius

}

}

Struct Rectangle {

Width: f64,

Height: f64,

}

Impl Shape for Rectangle {

Fn area(&self) -> f64 {

Self.width \* self.height

}

}

1. Liskov Substitution Principle (LSP)

این اصل در Rust به این معنی است که می‌توانید از traitها استفاده کنید و اشیاء مختلف را به‌جای یکدیگر استفاده کنید.

Rust

Fn print\_area(shape: &dyn Shape) {

Println!("Area: {}", shape.area());

}

Let circle = Circle { radius: 5.0 };

Let rectangle = Rectangle { width: 4.0, height: 6.0 };

Print\_area(&circle);

Print\_area(&rectangle);

1. Interface Segregation Principle (ISP)

در Rust، می‌توانید از traitهای کوچک و خاص استفاده کنید تا از ایجاد رابط‌های بزرگ جلوگیری کنید.

Rust

Trait Readable {

Fn read(&self);

}

Trait Writable {

Fn write(&self);

}

Struct Document;

Impl Readable for Document {

Fn read(&self) {

Println!("Reading document");

}

}

Impl Writable for Document {

Fn write(&self) {

Println!("Writing document");

}

}

1. Dependency Inversion Principle (DIP)

در Rust، می‌توانید از traitها و dependency injection برای پیاده‌سازی این اصل استفاده کنید.

Rust

Trait Database {

Fn save(&self, user: &User);

}

Struct MySQLDatabase;

Impl Database for MySQLDatabase {

Fn save(&self, user: &User) {

Println!("Saving user to MySQL: {}", user.name);

}

}

Struct UserController<'a> {

Db: &'a dyn Database,

}

Impl<'a> UserController<'a> {

Fn new(db: &'a dyn Database) -> UserController<'a> {

UserController { db }

}

Fn create\_user(&self, user: User) {

Self.db.save(&user);

}

}

نتیجه‌گیری

با پیاده‌سازی اصول SOLID در Rust، می‌توانید کدی قابل نگهداری‌تر و قابل توسعه‌تر ایجاد کنید. این اصول به شما کمک می‌کنند تا کد خود را ساختارمندتر و منظم‌تر کنید و از مشکلات احتمالی در آینده جلوگیری کنید.

پایان.