**تهیه کننده:**

زهرا دردخوار

نرگس توکلی پور

مریم رضایی زاده

کبری رحمی زاده

برنامه نویسی سیستمی

استاد: آقای احمد زاده

1. کتابخانه های Machine Learning در زبان Rust را نام ببرید؟ یک مثال ساده بنویسید؟

کتابخانه‌های مهم Machine Learning در Rust:

Linfa:

یکی از جامع‌ترین کتابخانه‌های یادگیری ماشین در Rust.

الگوریتم‌های متنوعی رو برای دسته‌بندی، خوشه‌بندی، رگرسیون و کاهش ابعاد ارائه میده.

دارای APIهای سطح بالا و راحتی هست.

برای یادگیری ماشین کلاسیک مناسبه.

Rust-ML:

مجموعه‌ای از کتابخانه‌های یادگیری ماشین که بر اساس اصول طراحی Rust ساخته شدن.

شامل کتابخانه‌هایی برای جبر خطی، شبکه‌های عصبی و پردازش سیگنال هست.

API سطح پایین‌تری داره و برای کاربردهای تخصصی مناسبه.

SmartCore:

یه کتابخانه یادگیری ماشین دیگه که الگوریتم‌های مختلفی رو ارائه میده.

دارای یک API تمیز و مدولاره.

تمرکز اصلی اون روی الگوریتم‌های دسته‌بندی و رگرسیون هست.

ndarray:

یه کتابخانه جبر خطی پایه که آرایه‌های چند بعدی رو در Rust فراهم می‌کنه.

برای کار با داده‌های عددی و ماتریسی مناسبه.

خیلی از کتابخانه‌های یادگیری ماشین در Rust از این کتابخانه استفاده می‌کنن.

tch-r:

باندینگ (binding) زبان Rust برای کتابخانه PyTorch (یکی از پرکاربردترین کتابخانه‌های یادگیری عمیق در Python).

امکان استفاده از مدل‌های PyTorch در Rust رو فراهم میکنه.

برای یادگیری عمیق مناسبه.

TensorFlow-rs:

باندینگ زبان Rust برای کتابخانه TensorFlow (یکی دیگه از کتابخانه‌های پرکاربرد یادگیری عمیق در Python).

امکان استفاده از مدل‌های TensorFlow در Rust رو فراهم میکنه.

برای یادگیری عمیق مناسبه.

یک مثال ساده با استفاده از Linfa:

در این مثال، یک مدل رگرسیون خطی ساده رو با استفاده از کتابخانه linfa آموزش میدیم.

rust

use linfa::prelude::\*;

use linfa\_linear::LinearRegression;

use ndarray::array;

fn main() {

// 1. ایجاد داده‌های آموزشی

let x = array![[1.0], [2.0], [3.0], [4.0], [5.0]]; // ورودی

let y = array![2.0, 4.0, 5.0, 4.0, 5.0]; // خروجی متناظر

// 2. ایجاد مدل رگرسیون خطی

let model = LinearRegression::default();

// 3. آموزش مدل با استفاده از داده‌ها

let fitted\_model = model.fit(&x, &y).unwrap();

// 4. پیش‌بینی مقدار برای یک داده جدید

let x\_test = array![[6.0]]; // داده تست

let prediction = fitted\_model.predict(&x\_test);

println!("Prediction for x = 6.0: {:?}", prediction);

}

1. برنامه نویسی Multi-Threading در زبان Rust را با ذکر یک مثال ساده توضیح دهید؟

Multi-Threading به معنای اجرای همزمان چندین نخ (thread) در یک برنامه است. هر نخ یک مسیر اجرایی مستقل است که می تواند کد را به طور موازی اجرا کند. این کار می تواند باعث افزایش سرعت و کارایی برنامه شود، به خصوص در مواردی که وظایف مختلف را می توان به طور همزمان انجام داد.

چرا Multi-Threading در Rust؟

ایمنی حافظه: Rust به دلیل سیستم مالکیت و قرض گرفتن (Ownership & Borrowing) منحصر به فردش، از خطاهای دسترسی همزمان به حافظه جلوگیری می کند، که این موضوع در برنامه نویسی چند نخی خیلی مهم است.

کارایی: Rust به زبان های سطح پایین نزدیک است و به شما کنترل کاملی روی مدیریت حافظه و نخ ها می دهد که منجر به عملکرد بهتری می شود.

انتزاع بالا: Rust انتزاع های خوبی برای کار با نخ ها فراهم می کند که برنامه نویسی چند نخی را راحت تر می کند.

مثال ساده Multi-Threading در Rust

در این مثال، دو نخ را ایجاد می کنیم که هر کدام یک شمارنده را افزایش می دهند و مقدارش را چاپ می کنند:

rust

use std::thread;

use std::time::Duration;

fn main() {

// ایجاد یک نخ جدید

let thread1 = thread::spawn(|| {

for i in 1..=5 {

println!("Thread 1: Count = {}", i);

thread::sleep(Duration::from\_millis(500));

}

});

// ایجاد یک نخ جدید دیگر

let thread2 = thread::spawn(|| {

for i in 10..=14 {

println!("Thread 2: Count = {}", i);

thread::sleep(Duration::from\_millis(700));

}

});

// صبر کردن برای اتمام نخ ها

thread1.join().unwrap();

thread2.join().unwrap();

println!("Main thread: All threads finished.");

}

1. برنامه نویسی Parallel Programming در زبان Rust را با ذکر یک مثال ساده توضیح دهید ؟

برنامه‌نویسی موازی (Parallel Programming) به معنای اجرای همزمان چندین کار است تا از منابع سخت‌افزاری به بهترین شکل ممکن استفاده شود. زبان Rust با ایمنی حافظه و کارایی بالای خود، گزینه‌ای عالی برای پیاده‌سازی برنامه‌نویسی موازی است.

چرا Rust برای برنامه‌نویسی موازی؟

ایمنی حافظه: Rust از سیستم مالکیت خود برای جلوگیری از مشکلات متداول مانند دسترسی همزمان و شرایط رقابتی استفاده می‌کند.

کارایی: کد Rust به زبان‌های سطح پایین نزدیک است و می‌تواند به خوبی از چند هسته‌ای بودن پردازنده‌ها بهره‌برداری کند.

ابزارهای مدرن: Rust ابزارهایی مانند Rayon دارد که برنامه‌نویسی موازی را ساده می‌کند.

1. LazyLoading چیست؟ با ذکر مثال در زبان Rust توضیح دهید ؟

Lazy Loading یک الگوی برنامه‌نویسی است که در آن تنها زمانی که یک منبع یا داده واقعا نیاز است، آن را بارگذاری می‌کنیم. این روش به ما کمک می‌کند تا زمان بارگذاری و مصرف حافظه را بهینه کنیم، به ویژه در مواقعی که داده‌ها یا منابع به طور دائم مورد استفاده قرار نمی‌گیرند.

مزایای Lazy Loading

کاهش زمان بارگذاری: بجای بارگذاری همه منابع در ابتدا، تنها منابع مورد نیاز بارگذاری می‌شوند.

استفاده بهینه از حافظه: حافظه تنها برای داده‌های فعال مصرف می‌شود.

سریع‌تر شدن عملکرد: در برخی موارد، با بارگذاری تنبل، می‌توانیم کارایی برنامه را بهبود بخشیم.

مثال در زبان Rust

برای پیاده‌سازی Lazy Loading در Rust، می‌توانیم از ابزارهایی مانند std::sync::Mutex و std::sync::Once استفاده کنیم. در اینجا یک مثال ساده از Lazy Loading یک منبع را خواهیم داشت.

کد مثال

در این مثال، یک struct داریم که فقط هنگام اولین درخواست، یک محاسبه گران‌قیمت را انجام می‌دهد و آن نتیجه را ذخیره می‌کند.

rust

use std::sync::{Once, ONCE\_INIT};

struct HeavyComputation {

value: i32,

}

impl HeavyComputation {

fn new() -> HeavyComputation {

// شبیه‌سازی محاسبه گران‌قیمت

println!("Performing heavy computation...");

HeavyComputation { value: 42 }

}

}

struct LazyLoader {

computation: Option<HeavyComputation>,

init: Once,

}

impl LazyLoader {

fn new() -> LazyLoader {

LazyLoader {

computation: None,

init: ONCE\_INIT,

}

}

fn compute(&mut self) -> &HeavyComputation {

self.init.call\_once(|| {

self.computation = Some(HeavyComputation::new());

});

self.computation.as\_ref().expect("Computation should be initialized")

}

}

fn main() {

let mut loader = LazyLoader::new();

// محاسبه فقط در اولین درخواست

let computation = loader.compute();

println!("The answer is: {}", computation.value);

// بار دیگر که فراخوانی شود، محاسبه انجام نمی‌شود

let computation\_again = loader.compute();

println!("The answer is still: {}", computation\_again.value);

}

1. ساختمان داده Binary Search Tree را در زبان Rust پیاده سازی نمایید؟

پیاده‌سازی Binary Search Tree (BST) در زبان Rust می‌تواند به درک بهتر ساختمان داده‌ها و مدیریت حافظه با استفاده از سه‌گانه‌ی مالکیت، وام‌دار بودن و توانایی Rust در مدیریت حافظه کمک کند. در اینجا یک پیاده‌سازی ساده از BST خواهیم داشت که شامل عملیات‌هایی مانند درج (insert)، جستجو (search)، و انتقال پیش‌سفارشی (pre-order traversal) می‌شود.

کد مثال برای Binary Search Tree در Rust

rust

// تعریف ساختار Node

#[derive(Debug)]

struct Node {

value: i32,

left: Option<Box<Node>>,

right: Option<Box<Node>>,

}

// پیاده‌سازی برای Node

impl Node {

fn new(value: i32) -> Node {

Node {

value,

left: None,

right: None,

}

}

}

// تعریف ساختار BinarySearchTree

#[derive(Debug)]

struct BinarySearchTree {

root: Option<Box<Node>>,

}

// پیاده‌سازی برای BinarySearchTree

impl BinarySearchTree {

fn new() -> BinarySearchTree {

BinarySearchTree { root: None }

}

fn insert(&mut self, value: i32) {

self.root = Some(Self::insert\_node(self.root.take(), value));

}

fn insert\_node(node: Option<Box<Node>>, value: i32) -> Box<Node> {

match node {

Some(mut n) => {

if value < n.value {

n.left = Some(Self::insert\_node(n.left.take(), value));

} else {

n.right = Some(Self::insert\_node(n.right.take(), value));

}

n

}

None => Box::new(Node::new(value)),

}

}

fn search(&self, value: i32) -> bool {

Self::search\_node(&self.root, value)

}

fn search\_node(node: &Option<Box<Node>>, value: i32) -> bool {

match node {

Some(n) => {

if value == n.value {

true

} else if value < n.value {

Self::search\_node(&n.left, value)

} else {

Self::search\_node(&n.right, value)

}

}

None => false,

}

}

fn pre\_order\_traversal(&self) {

Self::pre\_order(&self.root);

}

fn pre\_order(node: &Option<Box<Node>>) {

if let Some(n) = node {

print!("{} ", n.value);

Self::pre\_order(&n.left);

Self::pre\_order(&n.right);

}

}

}

fn main() {

let mut bst = BinarySearchTree::new();

// درج داده‌ها

bst.insert(5);

bst.insert(3);

bst.insert(7);

bst.insert(2);

bst.insert(4);

bst.insert(6);

bst.insert(8);

// اجرای جستجو

println!("Searching for 4: {}", bst.search(4)); // باید true باشد

println!("Searching for 10: {}", bst.search(10)); // باید false باشد

// اجرای پیش‌سفارش

println!("Pre-order traversal:");

bst.pre\_order\_traversal(); // باید 5 3 2 4 7 6 8 را چاپ کند

}

1. ساختمان داده AVL Tree را در زبان Rust پیاده سازی نمایید؟

پیاده‌سازی AVL Tree در زبان Rust می‌تواند به ما کمک کند تا با یک درخت جستجوی دودویی که از تعادل خود پشتیبانی می‌کند آشنا شویم. AVL Tree نوعی درخت جستجوی دودویی است که در آن ارتفاع دو زیر درخت فرزند هر نود می‌تواند تنها یک واحد تفاوت داشته باشد، که این موضوع منجر به تعادل بهینه جستجو، درج و حذف می‌شود.

ساختار داده AVL Tree در Rust

در اینجا یک پیاده‌سازی ساده از AVL Tree را ارائه می‌دهیم که شامل عملیات‌هایی مانند درج (insert)، جستجو (search)، و پیش‌سفارش (pre-order traversal) می‌شود.

کد مثال:

rust

#[derive(Debug)]

struct Node {

value: i32,

height: i32,

left: Option<Box<Node>>,

right: Option<Box<Node>>,

}

impl Node {

fn new(value: i32) -> Node {

Node {

value,

height: 1, // ارتفاع نود تازه ایجاد شده

left: None,

right: None,

}

}

}

struct AVLTree {

root: Option<Box<Node>>,

}

impl AVLTree {

fn new() -> AVLTree {

AVLTree { root: None }

}

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

match node {

Some(n) => n.height,

None => 0,

}

}

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

match node {

Some(n) => {

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

}

None => 0,

}

}

fn rotate\_right(y: Option<Box<Node>>) -> Option<Box<Node>> {

let mut x = y.as\_ref()?.left.take()?;

let t2 = x.right.take();

x.right = Some(y.unwrap());

x.right.as\_mut()?.left = t2;

x.right.as\_mut().map(|n| n.height = Self::update\_height(n));

Some(x)

}

fn rotate\_left(x: Option<Box<Node>>) -> Option<Box<Node>> {

let mut y = x.as\_ref()?.right.take()?;

let t2 = y.left.take();

y.left = Some(x.unwrap());

y.left.as\_mut()?.right = t2;

y.left.as\_mut().map(|n| n.height = Self::update\_height(n));

Some(y)

}

fn update\_height(node: &Box<Node>) -> i32 {

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

}

fn insert(&mut self, value: i32) {

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

}

fn insert\_node(node: Option<Box<Node>>, value: i32) -> Option<Box<Node>> {

let mut node = match node {

Some(n) => {

if value < n.value {

n.left = Self::insert\_node(n.left, value);

} else {

n.right = Self::insert\_node(n.right, value);

}

Some(n)

}

None => {

return Some(Box::new(Node::new(value)));

}

};

// بروزرسانی ارتفاع

node.as\_mut().map(|n| n.height = Self::update\_height(n));

// تعادل درخت و چرخش

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

if balance > 1 {

if value < node.as\_ref()?.left.as\_ref()?.value {

return Self::rotate\_right(node);

} else {

node.as\_mut().map(|n| n.left = Self::rotate\_left(n.left.take()));

return Self::rotate\_right(node);

}

}

if balance < -1 {

if value > node.as\_ref()?.right.as\_ref()?.value {

return Self::rotate\_left(node);

} else {

node.as\_mut().map(|n| n.right = Self::rotate\_right(n.right.take()));

return Self::rotate\_left(node);

}

}

node

}

fn search(&self, value: i32) -> bool {

Self::search\_node(&self.root, value)

}

fn search\_node(node: &Option<Box<Node>>, value: i32) -> bool {

match node {

Some(n) => {

if value == n.value {

true

} else if value < n.value {

Self::search\_node(&n.left, value)

} else {

Self::search\_node(&n.right, value)

}

}

None => false,

}

}

fn pre\_order\_traversal(&self) {

Self::pre\_order(&self.root);

}

fn pre\_order(node: &Option<Box<Node>>) {

if let Some(n) = node {

print!("{} ", n.value);

Self::pre\_order(&n.left);

Self::pre\_order(&n.right);

}

}

}

fn main() {

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

// درج مقادیر

avl\_tree.insert(10);

avl\_tree.insert(20);

avl\_tree.insert(30);

avl\_tree.insert(25);

avl\_tree.insert(5);

// جستجو

println!("Searching for 25: {}", avl\_tree.search(25)); // باید true باشد

println!("Searching for 50: {}", avl\_tree.search(50)); // باید false باشد

// پیش‌سفارش

println!("Pre-order traversal:");

avl\_tree.pre\_order\_traversal(); // باید اعداد را به صورت پیش‌سفارش چاپ کند

}

1. ساختمان داده Max heap-Tree را در زبان Rust پیاده سازی نمایید ؟

در اینجا یک پیاده‌سازی از ساختمان داده Max-Heap (درخت Max-Heap) به زبان برنامه‌نویسی Rust ارائه می‌شود. در این برنامه، از یک وکتور (Vec) برای مدیریت داده‌ها استفاده شده و عملیات استاندارد مانند insert، delete، و peek پیاده‌سازی شده‌اند.

پیاده‌سازی Max-Heap در Rust

rust

use std::cmp::Ordering;

pub struct MaxHeap {

data: Vec<i32>, // داده‌ها در قالب یک وکتور نگهداری می‌شوند

}

impl MaxHeap {

// ساخت یک MaxHeap جدید (خالی)

pub fn new() -> Self {

MaxHeap { data: Vec::new() }

}

// افزودن یک مقدار جدید به Heap

pub fn insert(&mut self, value: i32) {

self.data.push(value); // افزودن مقدار به انتهای وکتور

self.bubble\_up(); // مرتب‌سازی به بالا برای اطمینان از ساختار Max-Heap

}

// حذف و بازگرداندن المنت ماکزیمم (ریشه)

pub fn extract\_max(&mut self) -> Option<i32> {

if self.data.is\_empty() {

return None; // اگر هیپ خالی باشد، None برمی‌گردد

}

let max = self.data[0]; // مقدار ریشه را ذخیره می‌کنیم

let last = self.data.pop().unwrap(); // آخرین عنصر را حذف و ذخیره می‌کنیم

if !self.data.is\_empty() {

self.data[0] = last; // جایگزینی ریشه با آخرین عنصر

self.bubble\_down(); // مرتب‌سازی به پایین

}

Some(max) // مقدار ماکزیمم را بازگردانید

}

// مشاهده (بدون حذف) مقدار ماکزیمم

pub fn peek(&self) -> Option<i32> {

self.data.get(0).cloned() // بازگرداندن کپی از ریشه درصورت وجود

}

// تعداد عناصر در هیپ

pub fn len(&self) -> usize {

self.data.len()

}

// چک کردن اینکه آیا هیپ خالی است

pub fn is\_empty(&self) -> bool {

self.data.is\_empty()

}

// متد برای مرتب‌سازی به بالا (bubble up)

fn bubble\_up(&mut self) {

let mut index = self.data.len() - 1; // آخرین عنصر

while index > 0 {

let parent = (index - 1) / 2; // محاسبه شاخص پدر

if self.data[index] <= self.data[parent] {

break; // اگر شرط Max-Heap برقرار است، خاتمه دهید

}

self.data.swap(index, parent); // جابجایی با والد

index = parent; // حرکت به شاخص والد

}

}

// متد برای مرتب‌سازی به پایین (bubble down)

fn bubble\_down(&mut self) {

let mut index = 0; // شروع از ریشه

let len = self.data.len();

loop {

let left = 2 \* index + 1; // شاخص فرزند چپ

let right = 2 \* index + 2; // شاخص فرزند راست

let mut largest = index; // فرض می‌کنیم فعلاً بزرگترین همان ریشه است

if left < len && self.data[left] > self.data[largest] {

largest = left; // اگر فرزند چپ بزرگتر است، جایگزین کنیم

}

if right < len && self.data[right] > self.data[largest] {

largest = right; // اگر فرزند راست بزرگتر است، جایگزین کنیم

}

if largest == index {

break; // اگر هیچ تغییری لازم نیست، خاتمه دهید

}

self.data.swap(index, largest); // جابجایی با فرزند بزرگتر

index = largest; // حرکت به پایین به شاخص جدید

}

}

}

// مثال استفاده از MaxHeap

fn main() {

let mut max\_heap = MaxHeap::new();

max\_heap.insert(10);

max\_heap.insert(20);

max\_heap.insert(5);

max\_heap.insert(30);

println!("Max: {:?}", max\_heap.peek()); // نمایش مقدار ماکزیمم (30)

while !max\_heap.is\_empty() {

println!("Extracted Max: {:?}", max\_heap.extract\_max());

}

}

1. یک سرویس RESTFULAPI در جهت پردازش درخواست های JSON بنویسید؟

اصول کلی RESTful API قبل از پرداختن به جزئیات پیاده سازی، یادآوری اصول اولیه RESTful API ها ضروری است:

منابع: REST بر مبنای منابع کار می کند. هر منبع با یک URL یکتا شناسایی می شود (به عنوان مثال، /users، /products).

متد های HTTP: عملیات بر روی منابع با متدهای HTTP تعریف می شود:

GET: برای بازیابی یک منبع

POST: برای ایجاد یک منبع جدید

PUT: برای بروزرسانی یک منبع موجود

DELETE: برای حذف یک منبع

PATCH: برای بروزرسانی جزئی یک منبع

فرمت داده: معمولا از JSON برای تبادل داده ها استفاده می شود.

کدهای وضعیت HTTP: برای نشان دادن نتیجه درخواست، کدهای وضعیت استاندارد استفاده می شود (به عنوان مثال، 200 OK, 201 Created, 400 Bad Request, 404 Not Found).

**فایل toml:**

content\_copytoml

**[package]**

name = "json\_api"

version = "0.1.0"

edition = "2021"

**[dependencies]**

actix-web = "4"

serde = { version = "1", features = ["derive"] }

serde\_json = "1"

log = "0.4"

env\_logger = "0.10"

**فایل src/main.rs:**

rust

use actix\_web::{web, App, HttpResponse, HttpServer, Responder, Result, middleware::Logger};

use serde::{Deserialize, Serialize};

use std::sync::Mutex;

#[derive(Serialize, Deserialize, Debug, Clone)]

struct RequestData {

name: String,

value: i32,

}

#[derive(Serialize, Deserialize, Debug)]

struct ResponseData {

message: String,

received: RequestData,

}

struct AppState {

data: Mutex<Vec<RequestData>>,

}

async fn post\_data(req: web::Json<RequestData>, state: web::Data<AppState>) -> Result<HttpResponse> {

let request\_data = req.into\_inner();

let mut data = state.data.lock().unwrap();

data.push(request\_data.clone());

let response = ResponseData {

message: String::from("Data received successfully!"),

received: request\_data,

};

Ok(HttpResponse::Created().json(response))

}

async fn get\_data(state: web::Data<AppState>) -> impl Responder {

let data = state.data.lock().unwrap();

HttpResponse::Ok().json(data.clone())

}

async fn index() -> impl Responder {

HttpResponse::Ok().body("Hello, World!")

}

#[actix\_web::main]

async fn main() -> std::io::Result<()> {

env\_logger::init\_from\_env(env\_logger::Env::new().default\_filter\_or("info")); //تنظیم سطح لاگ

let app\_state = web::Data::new(AppState { data: Mutex::new(Vec::new()) });

HttpServer::new(move || {

App::new()

.app\_data(app\_state.clone())

.wrap(Logger::default()) // اضافه کردن middleware برای ثبت لاگ

.route("/", web::get().to(index))

.route("/data", web::post().to(post\_data))

.route("/data", web::get().to(get\_data))

})

.bind("127.0.0.1:8080")?

.run()

.await

}

1. یک سرویس ساده جهت پردازش درخواست های مبتنی بر پروتکل gPRC بنویسید؟

تعریف پروتکل با استفاده از فایل protobuf: یک فایل به نام simple\_service.proto ایجاد کنید و مشخصات سرویس gRPC را در آن تعریف کنید.

protobuf

syntax = "proto3";

package simple\_service;

service SimpleService {

rpc SendMessage(MessageRequest) returns (MessageResponse);

}

message MessageRequest {

string text = 1;

}

message MessageResponse {

string reply = 1;

}

توضیحات:

syntax = "proto3": مشخص می‌کند که از نسخه proto3 زبان protobuf استفاده می‌کنیم.

package simple\_service: نام پکیج را مشخص می‌کند.

service SimpleService: نام سرویس را مشخص می‌کند.

rpc SendMessage: یک متد SendMessage با یک درخواست MessageRequest و یک پاسخ MessageResponse را مشخص می‌کند.

message MessageRequest: ساختار پیام درخواست را با یک فیلد text از نوع رشته تعریف می‌کند.

message MessageResponse: ساختار پیام پاسخ را با یک فیلد reply از نوع رشته تعریف می‌کند.

تولید کد Python از فایل protobuf: از دستور زیر برای تولید کدهای Python برای سرویس و کلاینت استفاده کنید:

bash

python -m grpc.tools.protoc -I. --python\_out=. --grpc\_python\_out=. simple\_service.proto

این دستور دو فایل simple\_service\_pb2.py و simple\_service\_pb2\_grpc.py ایجاد می‌کند که شامل کدهای تولیدشده از فایل protobuf هستند.

پیاده‌سازی سرور gRPC: یک فایل به نام simple\_server.py ایجاد کنید و کد زیر را در آن قرار دهید: python

import grpc

from concurrent import futures

import time

import simple\_service\_pb2

import simple\_service\_pb2\_grpc

class SimpleService(simple\_service\_pb2\_grpc.SimpleServiceServicer):

def SendMessage(self, request, context):

reply\_text = f"Server received: {request.text}"

return simple\_service\_pb2.MessageResponse(reply=reply\_text)

def serve():

server = grpc.server(futures.ThreadPoolExecutor(max\_workers=10))

simple\_service\_pb2\_grpc.add\_SimpleServiceServicer\_to\_server(SimpleService(), server)

server.add\_insecure\_port("[::]:50051")

server.start()

print("gRPC server is running on port 50051")

try:

while True:

time.sleep(86400) # Server keeps running until stopped manually

except KeyboardInterrupt:

server.stop(0)

if \_\_name\_\_ == '\_\_main\_\_':

serve()

توضیحات:

کتابخانه‌های grpc، concurrent.futures، time، simple\_service\_pb2 و simple\_service\_pb2\_grpc را import می‌کنیم.

کلاسی به نام SimpleService ایجاد می‌کنیم که از simple\_service\_pb2\_grpc.SimpleServiceServicer مشتق شده و متد SendMessage را پیاده‌سازی می‌کند.

در متد SendMessage، یک پیام از کلاینت دریافت کرده و با پیشوند "Server received: " برمی‌گردانیم.

تابع serve سرور gRPC را راه‌اندازی و شروع می‌کند و منتظر درخواست‌ها می‌ماند.

با فشار دادن Ctrl+C سرور متوقف می‌شود.

پیاده‌سازی کلاینت gRPC: یک فایل به نام simple\_client.py ایجاد کنید و کد زیر را در آن قرار دهید:

python

import grpc

import simple\_service\_pb2

import simple\_service\_pb2\_grpc

def run():

channel = grpc.insecure\_channel("localhost:50051")

stub = simple\_service\_pb2\_grpc.SimpleServiceStub(channel)

request = simple\_service\_pb2.MessageRequest(text="Hello from the client!")

response = stub.SendMessage(request)

print(f"Response received: {response.reply}")

if \_\_name\_\_ == "\_\_main\_\_":

run()

توضیحات:

کتابخانه‌های grpc، simple\_service\_pb2 و simple\_service\_pb2\_grpc را import می‌کنیم.

در تابع run، یک کانال gRPC به سرور ایجاد می‌کنیم.

یک stub برای فراخوانی متدهای سرویس ایجاد می‌کنیم.

یک درخواست MessageRequest با پیام “Hello from the client!” ایجاد می‌کنیم.

متد SendMessage را فراخوانی و پاسخ را چاپ می‌کنیم.

اجرا:

اجرای سرور: ابتدا سرور را اجرا کنید:

bash

python simple\_server.py

سرور پیام gRPC server is running on port 50051 را نمایش می‌دهد.

اجرای کلاینت: سپس کلاینت را اجرا کنید:

bash

python simple\_client.py

کلاینت باید پیام Response received: Server received: Hello from the client! را نمایش دهد.

1. یک سرویس جهت پردازش درخواست های مبتنی بر Web Assembly بنویسید؟

این سرویس به شما اجازه می‌دهد تا توابع WebAssembly را از طریق درخواست‌های HTTP فراخوانی کنید.

پیش‌نیازها:

Node.js: اگر Node.js را نصب نکرده‌اید، از وب‌سایت رسمی Node.js دانلود و نصب کنید.

npm: مدیر بسته Node.js که همراه با Node.js نصب می‌شود.

wasmer و سایر ماژول ها: برای کار با WebAssembly و ایجاد سرور. آنها را با استفاده از npm نصب کنید:

bash

npm install express body-parser @wasmer/wasi @wasmer/sdk

کد Wasm: شما به یک فایل .wasm نیاز دارید. اگر ندارید، می‌توانید از کامپایلرهایی مانند emscripten یا AssemblyScript استفاده کنید.

c

#include <stdio.h>

#include <stdlib.h>

int add(int a, int b) {

return a + b;

}

void greet(const char\* name){

printf("Hello %s\n", name);

}

int main() {

return 0;

}

کامپایل با emscripten:

content\_copybash

emcc example.c -o example.wasm -s WASM=1 -s SIDE\_MODULE=1

مراحل:

ساختار پروژه: یک دایرکتوری جدید ایجاد کنید و فایل‌های زیر را در آن قرار دهید:

wasm-service/

├── node\_modules/

├── package.json

├── package-lock.json

├── server.js

└── example.wasm

فایل server.js (سرور Node.js):

javascript

const express = require('express');

const bodyParser = require('body-parser');

const { init, WASI } = require('@wasmer/wasi');

const { WasmFs } = require('@wasmer/wasmfs');

const { WebAssembly } = require('@wasmer/sdk');

const fs = require('node:fs/promises');

const app = express();

const port = 8080;

// Enable parsing of JSON bodies

app.use(bodyParser.json());

async function loadWasm() {

const wasmBinary = await fs.readFile('./example.wasm');

const wasmModule = await WebAssembly.compile(wasmBinary);

return wasmModule;

}

async function createWasmInstance(wasmModule){

const wasmFs = new WasmFs();

const wasi = new WASI({

args: [],

env: {},

preopens: {

"/": "/",

},

fs: wasmFs.fs,

});

const instance = await WebAssembly.instantiate(wasmModule, {

wasi\_snapshot\_preview1: wasi.wasiImport

});

return {instance, wasi};

}

async function wasmCaller(wasmModule, methodName, params) {

const {instance, wasi} = await createWasmInstance(wasmModule);

wasi.start(instance);

const exports = instance.exports;

const method = exports[methodName];

if (!method) {

throw new Error(`Method ${methodName} not found in wasm module.`);

}

try {

if(typeof method === 'function'){

return method(...params);

}else{

return method;

}

} catch (error) {

console.error(error);

throw new Error(`Error during wasm method call.`)

}

}

let wasmModule;

loadWasm().then(module =>{ wasmModule = module});

app.post('/wasm/:method', async (req, res) => {

const methodName = req.params.method;

const params = req.body;

if(!wasmModule){

return res.status(500).json({ error: "Wasm module loading has failed."});

}

try {

const result = await wasmCaller(wasmModule, methodName, params);

res.json({ result });

} catch (error) {

res.status(500).json({ error: error.message });

}

});

app.get('/', (req, res) => {

res.send("WebAssembly service is running!");

});

app.listen(port, () => {

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

});

اجرا: ابتدا سرور را اجرا کنید:

bash

node server.js

سپس، می توانید با استفاده از curl یا Postman درخواست‌های POST به http://localhost:8080/wasm/<method-name> بفرستید.

مثال‌های تست:

GET request:

bash

curl http://localhost:8080

پاسخ باید: WebAssembly service is running! باشد.

فراخوانی متد add:

bash

curl -X POST -H "Content-Type: application/json" -d '[5, 10]' http://localhost:8080/wasm/add

پاسخ باید:

json

{"result":15}

فراخوانی متد greet:

bash

curl -X POST -H "Content-Type: application/json" -d '["Test user"]' http://localhost:8080/wasm/greet

پاسخ باید:

json

{

"result": null

}

به این دلیل که متد greet خروجی ندارد مقدار null برمی‌گرداند. (در کنسول سرور عبارت Hello Test user چاپ می شود.)

1. Socket Programming در زبان Rust را بهمراه یک مثال بیان کنید؟

برنامه‌نویسی سوکت (Socket Programming) در Rust به شما امکان می‌دهد تا برنامه‌هایی بسازید که می‌توانند از طریق شبکه با یکدیگر ارتباط برقرار کنند. در اینجا یک مثال ساده از یک سرور و کلاینت با استفاده از TCP در Rust آورده شده است:

مفاهیم کلیدی:

std::net: ماژول اصلی برای کار با سوکت‌ها در Rust. شامل ساختارهایی مانند TcpListener، TcpStream و غیره است.

TcpListener: برای ایجاد یک سوکت سرور که به درخواست‌های ورودی گوش می‌دهد.

TcpStream: برای برقراری ارتباط بین کلاینت و سرور. هر دو طرف از این ساختار برای ارسال و دریافت داده‌ها استفاده می‌کنند.

read و write: متدهایی برای خواندن و نوشتن داده‌ها از/به سوکت.

thread: برای مدیریت همزمانی و هندل کردن چندین اتصال به صورت همزمان در سرور.

مثال:

ابتدا، دو فایل با نام‌های server.rs و client.rs ایجاد کنید.

server.rs:

rust

use std::io::{Read, Write};

use std::net::{TcpListener, TcpStream};

use std::thread;

fn handle\_client(mut stream: TcpStream) {

let mut buffer = [0; 1024];

loop {

match stream.read(&mut buffer) {

Ok(0) => {

println!("Client disconnected.");

break; // Connection closed

}

Ok(bytes\_read) => {

let message = String::from\_utf8\_lossy(&buffer[0..bytes\_read]);

println!("Received: {}", message);

// Echo back to client

match stream.write(&buffer[0..bytes\_read]) {

Ok(\_) => {}

Err(e) => {

eprintln!("Error echoing back: {}", e);

break;

}

}

}

Err(e) => {

eprintln!("Error reading from client: {}", e);

break;

}

}

}

}

fn main() {

let listener = TcpListener::bind("127.0.0.1:8080").expect("Could not bind to address");

println!("Server listening on 127.0.0.1:8080");

for stream in listener.incoming() {

match stream {

Ok(stream) => {

println!("New client connected: {}", stream.peer\_addr().unwrap());

thread::spawn(move || {

handle\_client(stream);

});

}

Err(e) => {

eprintln!("Error accepting connection: {}", e);

}

}

}

}

توضیحات server.rs:

TcpListener::bind: یک سوکت سرور را روی آدرس مشخص شده ایجاد می‌کند.

listener.incoming(): یک Iterator برمی‌گرداند که اتصالات ورودی را دریافت می‌کند.

thread::spawn: یک نخ جدید برای مدیریت هر اتصال ایجاد می‌کند. این کار باعث می‌شود که سرور بتواند همزمان به چندین کلاینت پاسخ دهد.

handle\_client: تابعی است که در هر نخ اجرا می‌شود و مسئول خواندن داده‌ها از کلاینت و ارسال پاسخ به آن است.

stream.read: داده‌ها را از کلاینت می‌خواند.

stream.write: داده‌ها را به کلاینت ارسال می‌کند.

client.rs:

crust

use std::io::{self, Read, Write};

use std::net::TcpStream;

use std::str;

fn main() -> io::Result<()> {

let mut stream = TcpStream::connect("127.0.0.1:8080")?;

println!("Connected to server!");

loop {

let mut input = String::new();

print!("Enter message: ");

io::stdout().flush()?; // Flush to show prompt

io::stdin().read\_line(&mut input)?;

let trimmed\_input = input.trim();

if trimmed\_input == "quit" {

break;

}

stream.write\_all(trimmed\_input.as\_bytes())?;

let mut buffer = [0; 1024];

let bytes\_read = stream.read(&mut buffer)?;

println!("Received from server: {}", str::from\_utf8(&buffer[0..bytes\_read]).unwrap\_or("Invalid UTF-8"));

}

Ok(())

}

توضیحات client.rs:

TcpStream::connect: به سرور متصل می‌شود.

io::stdin().read\_line: ورودی کاربر را می‌خواند.

stream.write\_all: داده‌ها را به سرور ارسال می‌کند.

stream.read: پاسخ سرور را می‌خواند.

کامپایل و اجرا:

کامپایل:

bash

rustc server.rs

rustc client.rs

اجرا: ابتدا سرور را اجرا کنید:

bash

./server

سپس کلاینت را در یک ترمینال جداگانه اجرا کنید:

bash

./client

حالا، می‌توانید در ترمینال کلاینت پیام‌هایی را تایپ کنید و آنها را به سرور ارسال کنید. سرور پیام‌ها را دریافت کرده و دوباره به کلاینت ارسال می‌کند. برای بستن کلاینت، در ترمینال آن عبارت quit را تایپ کنید.

1. برنامه ای به زبان Rust بنویسید و عملیات CRUD را بر روی یک پایگاه داده انجام دهد؟

برای نوشتن یک برنامه به زبان Rust که عملیات SRUD (Search, Read, Update, Delete) را بر روی یک پایگاه داده انجام دهد، می‌توان از کتابخانه‌های محبوبی مانند Diesel یا SQLx استفاده کرد. در اینجا از SQLx استفاده می‌کنیم، زیرا این کتابخانه async-friendly است و با پایگاه داده‌های مختلفی مانند PostgreSQL، MySQL و SQLite سازگار است.

مراحل کلی:

نصب وابستگی‌ها

تنظیمات پایگاه داده

تعریف مدل‌ها

پیاده‌سازی عملیات SRUD

کد Rust برای عملیات SRUD

در این مثال از SQLite به عنوان پایگاه داده استفاده می‌کنیم.

1. افزودن وابستگی‌ها به فایل Cargo.toml

toml

**[dependencies]**

sqlx = { version = "0.7", features = ["sqlite", "runtime-tokio-native-tls"] }

tokio = { version = "1", features = ["full"] }

2 کد Rust برای عملیات SRUD

use sqlx::{sqlite::SqlitePool, Row};

use std::error::Error;

#[tokio::main]

async fn main() -> Result<(), Box<dyn Error>> {

// اتصال به پایگاه داده SQLite

let database\_url = "sqlite://example.db";

let pool = SqlitePool::connect(database\_url).await?;

// ایجاد جدول

sqlx::query(

"CREATE TABLE IF NOT EXISTS users (

id INTEGER PRIMARY KEY AUTOINCREMENT,

name TEXT NOT NULL,

email TEXT NOT NULL

)",

)

.execute(&pool)

.await?;

// عملیات Create

let name = "Ali";

let email = "ali@example.com";

sqlx::query("INSERT INTO users (name, email) VALUES (?, ?)")

.bind(name)

.bind(email)

.execute(&pool)

.await?;

println!("User created: {} - {}", name, email);

// عملیات Read

let rows = sqlx::query("SELECT id, name, email FROM users")

.fetch\_all(&pool)

.await?;

for row in rows {

let id: i64 = row.get("id");

let name: String = row.get("name");

let email: String = row.get("email");

println!("User: {} - {} - {}", id, name, email);

}

// عملیات Update

let updated\_name = "Ali Reza";

sqlx::query("UPDATE users SET name = ? WHERE email = ?")

.bind(updated\_name)

.bind(email)

.execute(&pool)

.await?;

println!("User updated: {} -> {}", name, updated\_name);

// عملیات Delete

sqlx::query("DELETE FROM users WHERE email = ?")

.bind(email)

.execute(&pool)

.await?;

println!("User deleted: {}", email);

Ok(())

}

توضیحات کد:

اتصال به پایگاه داده: از SqlitePool برای مدیریت اتصال به پایگاه داده استفاده شده است.

ایجاد جدول: اگر جدول users وجود نداشته باشد، آن را ایجاد می‌کنیم.

عملیات Create: یک کاربر جدید به جدول اضافه می‌شود.

عملیات Read: تمام کاربران موجود در جدول خوانده و چاپ می‌شوند.

عملیات Update: نام کاربر با ایمیل مشخص به‌روزرسانی می‌شود.

عملیات Delete: کاربر با ایمیل مشخص حذف می‌شود.

اجرای برنامه:

ابتدا وابستگی‌ها را با دستور زیر نصب کنید:

bash

cargo build

سپس برنامه را اجرا کنید:

bash

cargo run

1. با استفاده از یک ORM در زبان Rust برنامه ای بنویسید که عملیات CRUD را برروی یک پایگاه داده انجام دهد؟

برای نوشتن برنامه‌ای که عملیات CRUD را با استفاده از یک ORM در زبان Rust انجام دهد، از کتابخانه Diesel استفاده می‌کنیم. Diesel یک ORM تایپ-سیف و قدرتمند است که با پایگاه داده‌های مختلفی مانند SQLite، PostgreSQL و MySQL سازگار است.

مراحل کلی:

نصب وابستگی‌ها: افزودن Diesel و وابستگی‌های مرتبط به پروژه.

ایجاد فایل‌های مربوط به پایگاه داده: شامل migrations و schema.

تعریف مدل‌ها: ایجاد ساختارهای Rust برای نگاشت به جداول پایگاه داده.

پیاده‌سازی عملیات CRUD: نوشتن توابع برای ایجاد، خواندن، به‌روزرسانی و حذف رکوردها.

1. افزودن وابستگی‌ها به فایل Cargo.toml

toml

**[dependencies]**

diesel = { version = "2.1", features = ["sqlite", "r2d2"] }

dotenvy = "0.15"

2. ایجاد فایل .env

برای ذخیره URL پایگاه داده:

DATABASE\_URL=sqlite://./my\_database.db

3. ایجاد فایل src/schema.rs

startLine: 1

endLine: 10

4. ایجاد فایل src/models.rs

startLine: 1

endLine: 20

5. کد Rust برای عملیات CRUD در src/main.rs

startLine: 1

endLine: 80

6. ایجاد migration

ابتدا با استفاده از Diesel CLI یک migration ایجاد کنید:

bash

diesel setup

diesel migration generate create\_users

سپس فایل ایجاد شده را باز کرده و در فایل up.sql کد زیر را اضافه کنید:

sql

CREATE TABLE users (

id INTEGER PRIMARY KEY AUTOINCREMENT,

name TEXT NOT NULL,

email TEXT NOT NULL

);

و در فایل down.sql کد زیر را اضافه کنید:

sql

DROP TABLE users;

سپس migration را اجرا کنید:

content\_copybash

diesel migration run

توضیحات کد:

اتصال به پایگاه داده: از SqliteConnection برای مدیریت اتصال به پایگاه داده استفاده شده است.

schema.rs: شامل تعریف جداول پایگاه داده است.

models.rs: شامل ساختارهای Rust برای نگاشت به جداول پایگاه داده است.

عملیات CRUD:

Create: افزودن یک کاربر جدید به جدول.

Read: خواندن تمام کاربران موجود در جدول.

Update: به‌روزرسانی نام کاربر با id مشخص.

Delete: حذف کاربر با id مشخص.

اجرای برنامه:

ابتدا وابستگی‌ها را نصب کنید:

bash

cargo build

سپس برنامه را اجرا کنید:

bash

cargo run

نکات مهم:

Diesel CLI را نصب کنید:

bash

cargo install diesel\_cli

برای استفاده از پایگاه داده‌های دیگر (مانند PostgreSQL یا MySQL)، تنظیمات اتصال و وابستگی‌ها را تغییر دهید.

Diesel استفاده کنید قابلیت‌های پیشرفته‌تری مانند روابط بین جداول و تراکنش‌ها دارد که می‌توانید در پروژه‌های بزرگ‌تر از آن‌ها.

1. در کتابخانه های Parsing در زبان Rust را نام ببرید؟ و عملکرد یک Parser را در قالب یک مثال توضیح دهید؟

در زبان Rust، کتابخانه‌های متعددی برای Parsing وجود دارند که هرکدام برای کاربردهای خاصی طراحی شده‌اند. برخی از محبوب‌ترین کتابخانه‌های Parsing عبارتند از:

Nom:

یک کتابخانه قدرتمند و سریع برای Parsing داده‌های باینری و متنی.

از رویکرد Parser Combinator استفاده می‌کند.

مناسب برای Parsing‌های پیچیده و کارایی بالا.

Pest:

یک کتابخانه ساده و کاربرپسند برای Parsing متن.

از Grammar Definitions استفاده می‌کند که در فایل‌های .pest تعریف می‌شوند.

مناسب برای Parsing زبان‌های برنامه‌نویسی یا فایل‌های متنی ساختاریافته.

Combine:

یک کتابخانه دیگر مبتنی بر Parser Combinator.

انعطاف‌پذیر و مناسب برای Parsing‌های پیچیده.

Chumsky:

یک کتابخانه جدیدتر و مدرن برای Parsing.

از رویکرد Parser Combinator استفاده می‌کند و API ساده‌ای دارد.

Logos:

برای Lexing (تجزیه اولیه متن به توکن‌ها) طراحی شده است.

سریع و مناسب برای تولید Lexer.

عملکرد یک Parser: مثال با استفاده از Nom

در اینجا یک مثال ساده از Parsing یک رشته با استفاده از کتابخانه Nom آورده شده است. فرض کنید می‌خواهیم یک رشته مانند 42, 56, 78 را به اعداد صحیح تجزیه کنیم.

افزودن وابستگی‌ها به Cargo.toml

toml

**[dependencies]**

nom = "7"

کد Rust برای Parsing

rust

use nom::{

bytes::complete::tag,

character::complete::{char, digit1},

combinator::map\_res,

multi::separated\_list0,

IResult,

};

// یک تابع برای تبدیل رشته به عدد صحیح

fn parse\_number(input: &str) -> IResult<&str, i32> {

map\_res(digit1, |digit\_str: &str| digit\_str.parse::<i32>())(input)

}

// یک تابع برای Parsing لیستی از اعداد جداشده با کاما

fn parse\_numbers(input: &str) -> IResult<&str, Vec<i32>> {

separated\_list0(char(','), parse\_number)(input)

}

fn main() {

let input = "42,56,78";

match parse\_numbers(input) {

Ok((remaining, numbers)) => {

println!("Parsed numbers: {:?}", numbers);

println!("Remaining input: {}", remaining);

}

Err(err) => {

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

}

}

}

1. مفهوم Regular Exprssion چیست؟ در زبان Rust با بیان یک مثال توضیح دهید؟

Regular Expression یا به اختصار Regex، یک الگوی متنی است که برای جستجو، تطبیق و دستکاری رشته‌ها استفاده می‌شود. این ابزار قدرتمند در بسیاری از زبان‌های برنامه‌نویسی برای کار با متن به کار می‌رود. با استفاده از Regex می‌توان الگوهایی مانند ایمیل، شماره تلفن، کد پستی و غیره را شناسایی کرد.

در زبان Rust، کتابخانه‌ای به نام regex برای کار با Regular Expressions وجود دارد که امکانات متنوعی برای تطبیق و جستجوی الگوها فراهم می‌کند.

1. عملکرد کتابخانه SysInfo در زبان Rust چیست؟ با ذکر یک مثال ساده توضیح دهید.

کتابخانه SysInfo در زبان Rust ابزاری برای دسترسی به اطلاعات سیستم‌عامل و وضعیت سیستم است. با استفاده از این کتابخانه می‌توانید به اطلاعات متنوعی از قبیل:

مشخصات پردازنده (CPU)

اطلاعات حافظه (RAM)

جزئیات سیستم فایل

وضعیت پردازش‌ها

اطلاعات شبکه

دست پیدا کنید. این کتابخانه برای نظارت بر سیستم و کار با متا‌داده‌های مربوط به آن بسیار مفید است.

مثال: گرفتن اطلاعات سیستم

در این مثال، ما می‌خواهیم مشخصات پردازنده و اطلاعات حافظه را از سیستم خود بگیریم.

کد Rust

rust

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

fn main() {

// ایجاد یک شیء جدید از کلاس System

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

// به‌روزرسانی اطلاعات سیستم

system.refresh\_all();

// گرفتن و چاپ نام پردازنده‌ها

for processor in system.get\_processors() {

println!("Processor: {}", processor.get\_name());

}

// گرفتن و چاپ اطلاعات حافظه

let total\_memory = system.get\_total\_memory(); // در کیلوبایت

let free\_memory = system.get\_free\_memory(); // در کیلوبایت

println!("Total Memory: {} MB", total\_memory / 1024);

println!("Free Memory: {} MB", free\_memory / 1024);

}

1. عملکرد کتابخانه native-windows-gui وwindows در زبان Rust چیست؟ با ذکر مثال ساده توضیح دهید.

در زبان Rust، کتابخانه‌های native-windows-gui و windows برای توسعه و ساخت رابط‌های کاربری گرافیکی (GUI) بر روی سیستم‌عامل ویندوز طراحی شده‌اند. هدف این کتابخانه‌ها آسان‌تر کردن توسعه برنامه‌های ویندوزی با استفاده از زبان Rust است.

1. native-windows-gui

کتابخانه native-windows-gui به توسعه‌دهندگان اجازه می‌دهد که به سادگی رابط کاربری ویندوز را ایجاد و مدیریت کنند. این کتابخانه از Win32 API استفاده می‌کند و مناسب برای برنامه‌های دسکتاپ ساده است.

2. windows

کتابخانه windows که به‌تازگی معرفی شده، راهی جامع‌تر برای کار با Win32 API در Rust است و اجازه می‌دهد که به قسمت‌های بیشتری از API ویندوز دسترسی داشته باشید. این کتابخانه امکانات بیشتری نسبت به native-windows-gui ارائه می‌دهد و به همین دلیل کنترلی بیشتر و امکانات شناخته‌شده‌تری را برای کاربران فراهم می‌آورد.

مثال: برنامه ساده با استفاده از native-windows-gui

در این مثال، ما یک پنجره ساده با یک دکمه ایجاد خواهیم کرد که با کلیک بر روی آن پیامی نشان داده می‌شود.

نصب وابستگی‌ها

برای استفاده از کتابخانه native-windows-gui، باید آن را به پروژه خود اضافه کنید. این کار را با افزودن موارد زیر به Cargo.toml انجام دهید:

toml

**[dependencies]**

native-windows-gui = "1.0"

کد Rust

rust

use native\_windows\_gui as nwg;

fn main() {

// راه‌اندازی رابط کاربری

nwg::init().expect("Failed to initialize Native Windows GUI");

// ایجاد پنجره اصلی

let mut window = nwg::Window::default();

window.set\_title("Hello World Window");

window.set\_size((400, 200));

window.set\_position((300, 300));

// ایجاد دکمه

let mut button = nwg::Button::default();

button.set\_text("Click me");

button.set\_parent(&window);

button.set\_position((150, 80));

// تعریف عملکرد دکمه برای کلیک

button.on\_click(move |\_btn| {

nwg::modal\_info("Message", "Hello, World!").expect("Failed to show message");

});

// نشان دادن پنجره

window.show();

nwg::dispatch\_thread\_events();

}

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

برای ایجاد یک برنامه ساده برای پردازش تصویر، می‌توانیم از کتابخانه‌های محبوبی مانند Pillow در پایتون استفاده کنیم. در اینجا یک مثال از یک برنامه ساده به شما ارائه می‌دهم که یک تصویر را بارگذاری می‌کند، آن را تبدیل به مقیاس خاکستری می‌کند و سپس ذخیره می‌کند.

مراحل نصب:

اول، اطمینان حاصل کنید که کتابخانه Pillow را نصب کرده‌اید. می‌توانید این کار را با استفاده از pip انجام دهید:

bash

pip install Pillow

کد برنامه:

python

from PIL import Image

# بارگذاری تصویر

image\_path = 'input\_image.jpg' # مسیر تصویر ورودی

output\_path = 'output\_image.jpg' # مسیر تصویر خروجی

# باز کردن تصویر

image = Image.open(image\_path)

# تبدیل به مقیاس خاکستری

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

# ذخیره تصویر پردازش شده

gray\_image.save(output\_path)

print("تصویر به مقیاس خاکستری تبدیل شد و در مسیر مشخص شده ذخیره گردید.")

توضیحات:

کتابخانه Pillow: این کتابخانه برای پردازش تصویر در پایتون استفاده می‌شود.

Image.open: برای باز کردن تصویر استفاده می‌شود.

convert(‘L’): تصویر را به مقیاس خاکستری تبدیل می‌کند.

save: تصویر پردازش شده را در مسیر مشخص شده ذخیره می‌کند.

1. برنامه ای برای انجام یک پردازش ساده برروی یک Audio بنویسید.

برای پردازش صوت در پایتون، می‌توانیم از کتابخانه pydub استفاده کنیم. این کتابخانه امکان کار با فایل‌های صوتی را فراهم می‌کند و به شما اجازه می‌دهد تا ویژگی‌های مختلف صوتی را تغییر دهید و پردازش‌های مختلفی انجام دهید.

در این مثال، ما یک فایل صوتی را بارگذاری می‌کنیم و آن را تبدیل به فرمت دیگری (به عنوان مثال، MP3) می‌کنیم.

مراحل نصب:

اول، اطمینان حاصل کنید که کتابخانه pydub و همچنین ffmpeg را نصب کرده‌اید. برای نصب pydub، از دستور زیر استفاده کنید:

bash

pip install pydub

برای نصب ffmpeg، بسته به سیستم عامل خود، روش‌های مختلفی وجود دارد. به عنوان مثال، می‌توانید از طریق brew در macOS استفاده کنید:

bash

brew install ffmpeg

یا بر روی ویندوز می‌توانید از [سایت رسمی ffmpeg](https://ffmpeg.org/download.html) دانلود کنید و سپس آن را به PATH اضافه نمایید.

کد برنامه:

python

from pydub import AudioSegment

# بارگذاری فایل صوتی

audio\_path = 'input\_audio.wav' # مسیر فایل صوتی ورودی

output\_path = 'output\_audio.mp3' # مسیر فایل صوتی خروجی

# بارگذاری فایل صوتی

audio = AudioSegment.from\_file(audio\_path)

# ذخیره فایل صوتی به فرمت جدید

audio.export(output\_path, format='mp3')

print("فایل صوتی به فرمت MP3 تبدیل شد و در مسیر مشخص شده ذخیره گردید.")

توضیحات:

کتابخانه pydub: برای پردازش فایل‌های صوتی استفاده می‌شود.

AudioSegment.from\_file: برای بارگذاری فایل صوتی استفاده می‌شود.

export: برای ذخیره کردن فایل صوتی به فرمت جدید (در اینجا MP3) استفاده می‌شود.

1. برنامه ای برای انجام یک پردازش ساده برروی یک Video بنویسید.

برای پردازش و ویرایش ویدیوها در پایتون، می‌توانیم از کتابخانه moviepy استفاده کنیم. این کتابخانه قابلیت‌های متنوعی برای پردازش ویدیوها، از جمله برش، ادغام، افزودن متن و صدا و … را ارائه می‌دهد.

در این مثال، ما یک ویدیو را بارگذاری می‌کنیم و بخش خاصی از آن را برش می‌زنیم و در یک فایل جدید ذخیره می‌کنیم.

مراحل نصب:

ابتدا اطمینان حاصل کنید که کتابخانه moviepy را نصب کرده‌اید. شما می‌توانید آن را با استفاده از دستور زیر نصب کنید:

bash

pip install moviepy

کد برنامه:

python

from moviepy.video.io.VideoFileClip import VideoFileClip

# بارگذاری ویدیو

video\_path = 'input\_video.mp4' # مسیر ویدیوی ورودی

output\_path = 'output\_video.mp4' # مسیر ویدیوی خروجی

# باز کردن ویدیو

video = VideoFileClip(video\_path)

# برش ویدیو (از ثانیه 10 تا ثانیه 20)

start\_time = 10 # ثانیه شروع

end\_time = 20 # ثانیه پایان

cut\_video = video.subclip(start\_time, end\_time)

# ذخیره ویدیو برش خورده

cut\_video.write\_videofile(output\_path, codec='libx264')

print("ویدیو برش داده شد و در مسیر مشخص شده ذخیره گردید.")

توضیحات:

کتابخانه moviepy: این کتابخانه برای پردازش ویدیوها و انجام کارهای مختلف بر روی آنها استفاده می‌شود.

VideoFileClip: برای بارگذاری ویدیو استفاده می‌شود.

subclip: این تابع برای برش ویدیو بین دو زمان مشخص (ثانیه 10 تا 20) استفاده می‌شود.

write\_videofile: این تابع برای ذخیره ویدیوی برش خورده در یک فایل جدید استفاده می‌شود.