﴿_ بِسمِ اللهِ الرَّحمنِ الرَّحیم _﴾

نام پروژه: برنامه نویسی سیستم

استاد محترم:محمد احمدزاده

تهیه کننده:مرضیه تاجایی

پاییز۱۴۰۳

1. کتابخانه های Machine Larning در زبان راست را نام ببرید؟یک مثال ساده بنویسید.

زبان برنامه‌نویسی Rust به دلیل ویژگی‌های منحصر به فرد خود مانند ایمنی حافظه و عملکرد بالا، به آرامی در حال gaining traction در حوزه یادگیری ماشین (Machine Learning) است. چند کتابخانه معتبر در زمینه یادگیری ماشین در Rust عبارتند از:

1. **ndarray**: این کتابخانه برای عملیات‌های عددی و کار با آرایه‌ها طراحی شده است و می‌تواند در زمینه یادگیری ماشین مورد استفاده قرار گیرد.

2. **tch-rs**: این یک بسته Rust برای استفاده از PyTorch در Rust است. این کتابخانه به شما اجازه می‌دهد تا از قابلیت‌های یادگیری عمیق PyTorch استفاده کنید.

3. **rustlearn**: این کتابخانه به عنوان یک کتابخانه یادگیری ماشین ساده و مدرن طراحی شده است و مجموعه‌ای از الگوریتم‌های یادگیری ماشین را ارائه می‌دهد.

4. **linfa**: این یک کتابخانه‌ی یادگیری ماشین است که سعی دارد یک مجموعه از الگوریتم‌های یادگیری ماشین را مشابه scikit-learn در زبان Python فراهم سازد.

5. **SmartCore**: یک کتابخانه یادگیری ماشین که شامل الگوریتم‌های مختلف و ابزارهای مربوط به آن‌ها می‌باشد.

مثال ساده با استفاده از کتابخانه ``ndarray

در این مثال، ما یک مدل ساده رگرسیون خطی را با استفاده از Rust و کتابخانه `ndarray` پیاده‌سازی خواهیم کرد.

1. **نصب بسته‌های مورد نیاز**:

ابتدا در فایل `Cargo.toml` خود، وابستگی‌های زیر را اضافه کنید:

```toml
[dependencies]
Ndarray = "0.15.3"
```

```
"Ndarray-rand = "0.14.0
```

2. **کد رگرسیون خطی ساده**:

```rust
;Extern crate ndarray
;Extern crate ndarray_rand
;Use ndarray::{Array, Array2}
;Use ndarray_rand::RandomExt
;Use ndarray_rand::rand_distr::Uniform
} ()Fn main
// تولید داده‌های آموزشی (x: ویژگی‌ها، y: برچسب‌ها)
داده ورودی Let x = Array::random((100, 1), Uniform::new(0., 10.)); // 100
Let y = &x * 3.0 + Array::random(100, Uniform::new(0., 1.)); // y = 3x + noise
// تولید وزن اولیه (W) و تعصب (b)
;Let mut w = Array::zeros(1)
;Let mut b = 0.0
;Let learning_rate = 0.01
// تعداد دوره (Epochs) برای آموزش
} For _ in 0..1000
    // پیش‌بینی (Forward Pass)
    ;Let y_pred = &x.dot(&w) + b
    // محاسبه خطای (Loss)
    ;()Let loss = (&y - &y_pred).mapv(|v| v.powi(2)).mean().unwrap
    // محاسبه گرادیان
    ;Let dw = -2.0 * x.t().dot(&(y - &y_pred)) / y.len() as f64
    ;Let db = -2.0 * (y - &y_pred).sum() / y.len() as f64
    // به‌روزرسانی وزن‌ها و تعصب
    ;W -= &(dw * learning_rate)
```

```
    B -= db * learning_rate;

    // چاپ خطا در هر 100 دوره
    if _ % 100 == 0 {
        println!("Epoch: {}, Loss: {}", _, loss);
    }
}

println!("Weight: {:?}", w);
println!("Bias: {}", b);
}
```

توضیحات کد:

- ما ابتدا داده‌های تصادفی تولید می‌کنیم که به صورت خطی به هم مرتبط هستند: $y = 3x + \text{noise}$.
- سپس وزن‌ها و تعصب‌ها را با مقدار اولیه صفر مقداردهی می‌کنیم.
- در حلقه اصلی، ما پیش‌بینی‌ها را انجام داده و خطای مربوطه را محاسبه می‌کنیم.
- بر اساس خطا، گرادیان‌ها را محاسبه کرده و وزن‌ها و تعصب‌ها را به‌روزرسانی می‌کنیم.
- در نهایت، نتایج مدل نهایی (وزن و تعصب) را چاپ می‌کنیم.

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

برنامه‌نویسی موازی ((Parallel Programmingبه معنای اجرای همزمان چندین محاسبه یا پروسه به منظور افزایش کارایی و کاهش زمان اجرای برنامه است. زبان Rustبه دلیل ویژگی‌های منحصر به فردی مثل ایمنی حافظه و عدم وجود تنش‌های داده‌ای می‌تواند یک گزینه عالی برای برنامه‌نویسی موازی باشد.

نکات کلیدی درباره برنامه‌نویسی موازی در :Rust

1. **مدیریت حافظه**: Rustبا استفاده از سیستم قرض‌دهی ((Borrowingو مالکیت ((Ownership می‌تواند از بروز مشکلات معمول در برنامه‌نویسی موازی مانند شرایط رقابتی ((Race Conditions جلوگیری کند.

2. **کتابخانه‌های مفید**: Rustهمچنین کتابخانه‌های بیشتری را برای راحت‌تر کردن برنامه‌نویسی موازی در اختیار دارد مثل `std::threadبرای مدیریت تردها و `rayonبرای پردازش موازی داده‌ها.

مثال ساده:

در این مثال، می‌خواهیم یک آرایه از اعداد را جمع کنیم، اما این کار را با استفاده از چندین ترد ((Thread انجام خواهیم داد. ابتدا آرایه را به تعدادی بخش تقسیم می‌کنیم و هر بخش را در یک ترد پردازش می‌کنیم.

```rust
Use std::thread;

Fn main() {
    // ایجاد یک آرایه از اعداد
Let numbers = vec![1, 2, 3, 4, 5, 6, 7, 8, 9, 10];

Let mut handles = vec![];
    // تقسیم آرایه به ۲ بخش برای تردها و جمع‌زدن هر بخش
    Let mid = numbers.len() / 2;
    // ترد اول
Let numbers_first_half = &numbers[0..mid];

Let handle1 = thread::spawn(move || {
// Numbers_first_half.iter().sum::<i32>() جمع کردن مقادیر بخش اول
});
    // ترد دوم
Let numbers_second_half = &numbers[mid..];

Let handle2 = thread::spawn(move || {
```

```
// Numbers_second_half.iter().sum::<i32>() جمع کردن مقادیر بخش دوم
});
// جمع نتیجه
Let sum1 = handle1.join().unwrap(); // دریافت نتیجه از ترد اول
Let sum2 = handle2.join().unwrap(); // دریافت نتیجه از ترد دوم

Let total_sum = sum1 + sum2; // جمع کل
Println!("The total sum is: {}", total_sum); // نمایش نتیجه
}
```

توضیحات کد:

- **ایجاد آرایه**: یک آرایه از اعداد صحیح تعریف شده است.
- **تقسیم آرایه**: آرایه به دو بخش تقسیم می‌شود. تعداد تردها به تعداد بخش‌ها بستگی دارد.
- **spawn**: با استفاده از `thread::spawn` یک ترد جدید ایجاد می‌شود. هر ترد جمع اعداد مربوط به بخش خود را محاسبه می‌کند و نتیجه را برمی‌گرداند.
- **join**: `handle1.join()` و `handle2.join()` برای دریافت نتایج تردها استفاده می‌شود. این متدها بلوکه می‌کنند تا تردهای فرزند به اتمام برسند.
- **جمع نهایی**: در نهایت، نتایج جمع تردها با هم جمع می‌شوند و چاپ می‌شوند.

نکات دیگر:

- **ایمنی**: Rust به طور خودکار ایمنی حافظه را در تردها بررسی می‌کند و از دسترسی همزمان به داده‌های مشترک جلوگیری می‌کند.
- **سادەسازی با rayon**: اگر بخواهیم از کتابخانه `rayon` استفاده کنیم، می‌توانیم با یک خط کد، پردازش موازی را بسیار راحت‌تر کنیم:

```rust
Use rayon::prelude::*;
Fn main() {
Let numbers = vec![1, 2, 3, 4, 5, 6, 7, 8, 9, 10];
Let total_sum: i32 = numbers.par_iter().sum(); // استفاده از rayon برای جمع‌زنی موازی
Println!("The total sum is: {}", total_sum);
}
```

```

با `rayon`, فقط کافی است تا از `par_iter()` به جای `iter()` استفاده کنیم و کل فرآیند به صورت موازی انجام می‌شود.

این مثال‌ها به خوبی نشان می‌دهد که چگونه می‌توان با استفاده از Rust و ویژگی‌های آن، برنامه‌نویسی موازی را پیاده‌سازی کرد.

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

در زبان Rust، می‌توان یک درخت جستجوی دودویی (Binary Search Tree یا BST) را با استفاده از ساختارها (structs) و متدها (methods) پیاده‌سازی کرد. درخت جستجوی دودویی یک ساختار داده‌ای است که در آن هر گره دارای یک مقدار (یا کلید) است و دو زیر درخت (چپ و راست) دارد. برای هر گره، کلیدهای موجود در زیر درخت چپ کوچک‌تر از کلید گره و کلیدهای موجود در زیر درخت راست بزرگ‌تر از کلید گره هستند.

در ادامه یک پیاده‌سازی ساده از BST به زبان Rust ارائه می‌دهم:

```rust
#[derive(Debug)]
Struct Node {
    Key: i32,
    Left: Option<Box<Node>>,
    Right: Option<Box<Node>>,
}

#[derive(Debug)]
Struct BinarySearchTree {
    Root: Option<Box<Node>>,
}

Impl BinarySearchTree {
    Fn new() -> Self {
```
```

```
                                        BinarySearchTree { root: None }
                                                                    {

                                                } Fn insert(&mut self, key: i32)
                                    ;Self.root = Self::insert_node(self.root.take(), key)
                                                                    {

} <<Fn insert_node(node: Option<Box<Node>>, key: I32) -> Option<Box<Node
                                                        } Match node
                                                    } <= Some(mut n)
                                                     } If key < n.key
                                        ;n.left = Self::Insert_node(n.left, key)
                                                    } else if key > n.key {
                                    ;n.right = Self::Insert_node(n.right, key)
                                                                {
                                                Some(n)
                                                            {
                ,None => Some(Box::new(Node { key, left: None, right: None }))
                                                                {
                                                                    {

                                                } Fn search(&self, key: i32) -> bool
                                                Self::search_node(&self.root, key)
                                                                    {

                                    } Fn search_node(node: &Option<Box<Node>>, key: I32) -> bool
                                                        } Match node
                                                    } <= Some(n)
```

```
                                                    } If key == n.key
                                                                True
                                                    } else if key < n.key {
                                            Self::search_node(&n.left, key)
                                                    } else {
                                           Self::search_node(&n.right, key)
                                                                {
                                                                   {
                                                  ,None => false
                                                                      {
                                                                         {

                                                } <Fn in_order_traversal(&self) -> Vec<i32
                                                        ;()Let mut result = Vec::new
                                                ;Self::in_order_node(&self.root, &mut result)
                                                                Result
                                                                      {

} Fn in_order_node(node: &Option<Box<Node>>, result: &mut Vec<i32>)
                                                    } If let Some(n) = node
                                                ;Self::in_order_node(&n.left, result)
                                                        ;Result.push(n.key)
                                                ;Self::in_order_node(&n.right, result)
                                                                      {
                                                                         {
                                                                            {

                                                                } ()Fn main
```

```
;()Let mut bst = BinarySearchTree::new

;Bst.insert(5)

;Bst.insert(3)

;Bst.insert(7)

;Bst.insert(2)

;Bst.insert(4)

;Bst.insert(6)

;Bst.insert(8)

;Println!("In-order traversal: {:?}", bst.In_order_traversal())

;Let search_key = 4

} If bst.search(search_key)

    ;Println!("Key {} found In the tree.", search_key)

} else {

    ;Println!("Key {} not found in the tree.", search_key)

{

{
```

توضیحات:

1. **ساختار Node**:

- گره درخت که دو زیر گره (چپ و راست) و یک کلید (key) دارد.

- از نوع `Option<Box<Node>>` برای زیر گره‌ها استفاده شده است تا بتوانیم پوینترهای خالی را مدیریت کنیم.

2. **ساختار BinarySearchTree**:

- این ساختار شامل یک ریشه (root) است که یک `Option<Box<Node>>` است.

3. **متدهای اصلی**:

- `new()`: برای ایجاد یک درخت خالی.

- `insert()`: برای افزودن یک کلید به درخت.

- `search()`: برای جستجوی یک کلید در درخت.

- `in_order_traversal()`: برای پیمایش درخت به ترتیب افزایشی (in-order).

4. **پیمایش in-order**:

- در این پیمایش ابتدا زیر درخت چپ، سپس خود گره و سپس زیر درخت راست بازدید می‌شود.

5. **متد main**:

- در این قسمت یک درخت جستجوی دودویی ایجاد شده و تعدادی کلید به آن اضافه می‌شود. سپس درخت پیمایش می‌شود و و همچنین جستجوی یک کلید انجام می‌شود.

این پیاده‌سازی به‌طور کلی پایه‌ای است و می‌توان آن را با اضافه کردن ویژگی‌هایی مانند حذف گره و مدیریت تکراری بودن کلیدها گسترش داد.

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

برای پیاده‌سازی یک Max Heap در زبان Rust، ابتدا باید ساختار داده‌ای برای نمایندگی درخت و همچنین توابعی برای عملیات‌های مختلف مانند درج، حذف و نمایش درخت ایجاد کنیم. در زیر یک پیاده‌سازی ساده از Max Heap در Rust ارائه می‌شود:

```
Rust

[derive(Debug)]#

} Struct MaxHeap

,<Data: Vec<i32

{


} Impl MaxHeap

} Fn new() -> Self

MaxHeap { data: Vec::new() }

{


} Fn parent(&self, Index: usize) -> usize

(index – 1) / 2

{


} Fn left_child(&self, index: usize) -> usize
```

```
                                        index + 1 * 2
                                                {

                                    } Fn right_child(&self, index: usize) -> usize
                                        index + 2 * 2
                                                {

                                } Fn insert(&mut self, value: I32)
                                        ;Self.data.push(value)
                                ;Self.bubble_up(self.data.len() - 1)
                                                {

                                } Fn bubble_up(&mut self, index: usize)
                                        ;Let mut current_index = Index
                                        } While current_index > 0
                                ;Let parent_index = self.parent(current_Index)
                        } If self.data[current_index] > self.data[parent_Index]
                                ;Self.data.swap(current_index, parent_index)
                                        ;Current_index = parent_index
                                                } else {
                                                ;Break
                                                        {
                                                            {
                                                                {

                                    } <Fn extract_max(&mut self) -> Option<i32
                                                } ()If self.data.is_empty
                                                        ;Return None
```

```
                                                                {
                                                    ;Let max_value = self.data[0]
                                                ;()Let last_value = self.data.pop().unwrap
                                                        } ()If !self.data.is_empty
                                                    ;Self.data[0] = last_value
                                                        ;Self.bubble_down(0)
                                                                {
                                                            Some(max_value)
                                                                    {

                                                } Fn bubble_down(&mut self, Index: usize)
                                                        ;Let mut current_index = Index
                                                        ;()Let length = self.data.len
                                                                } Loop
                                                ;Let left_index = self.left_child(current_Index)
                                                ;Let right_index = self.right_child(current_Index)
                                                        ;Let mut largest_index = current_index

    } If left_index < length && self.data[left_Index] > self.data[largest_Index]
                                                            ;Largest_index = left_index
                                                                    {
} If right_index < length && self.data[right_Index] > self.data[largest_Index]
                                                            ;Largest_index = right_index
                                                                    {

                                                } If largest_index == current_Index
                                                                ;Break
                                                                    {
```

```
;Self.data.swap(current_index, largest_Index)
                ;Current_index = largest_index
                                        {
                                          {

                } <Fn peek(&self) -> Option<&i32
                                Self.data.get(0)
                                        {

                            } Fn is_empty(&self) -> bool
                                ()Self.data.is_empty
                                        {

                                } Fn size(&self) -> usize
                                    ()Self.data.len
                                        {
                                           {

                                                } ()Fn main
                            ;()Let mut heap = MaxHeap::new
                                        ;Heap.insert(10)
                                        ;Heap.insert(20)
                                         ;Heap.insert(5)
                                        ;Heap.insert(30)
                                        ;Heap.insert(15)
                        ;Println!("Max Heap: {:?}", heap)
                ;Println!("Max value: {:?}", heap.extract_max())
```

```
;Println!("Heap after extracting max: {:?}", heap)
}
```

توضیحات کد:

ساختار MaxHeap: یک ساختار برای نمایندگی درخت ماکس هیپ ایجاد کردیم که یک وکتور از اعداد صحیح (i32) را نگهداری می‌کند.

عملیات‌های اصلی:

Insert: برای اضافه کردن یک عنصر به هیپ و حفظ خاصیت ماکس هیپ.

Extract_max: برای حذف و بازگرداندن بزرگ‌ترین عنصر از هیپ.

Peek: برای مشاهده بزرگ‌ترین عنصر بدون حذف آن.

Is_empty و size: برای بررسی خالی بودن هیپ و اندازه آن.

تابع‌های کمکی:

Parent, left_child, right_child: برای محاسبه اندیس والد و فرزندان.

Bubble_up و bubble_down: برای حفظ خاصیت ماکس هیپ پس از درج و حذف.

استفاده:

در تابع main، چندین عدد به هیپ اضافه شده و عملیات استخراج انجام می‌شود. می‌توانید این کد را در محیط Rust خود اجرا کنید و نتایج را مشاهده کنید.

۵.یک سرویس ساده جهت پردازش درخواست های مبتنی بر پروتکل gRPC بنویسید؟

برای ایجاد یک سرویس ساده مبتنی بر پروتکل gRPC، ابتدا باید با مفاهیم پایه gRPC آشنا شوید. gRPC یک چارچوب ارتباطی است که توسط گوگل توسعه یافته و برای ارتباط بین سرویس‌ها استفاده می‌شود. این پروتکل از HTTP/2 برای انتقال داده‌ها و پروتکل‌های protobuf برای تعریف ساختار داده‌ها استفاده می‌کند.

مراحل ایجاد یک سرویس gRPC ساده:

نصب وابستگی‌ها:

ابتدا باید gRPC و protobuf را نصب کنید. اگر از زبان برنامه‌نویسی Python استفاده می‌کنید، می‌توانید از pip استفاده کنید:

Bash

```
Pip install grpcio grpcio-tools
```

تعریف پروتکل:

یک فایل با پسوند .proto ایجاد کنید. این فایل شامل تعریف خدمات و پیام‌ها است. به عنوان مثال، فایل example.proto را ایجاد کنید:

Proto

```
;"Syntax = "proto3

;Package example

} Service Greeter {

;Rpc SayHello (HelloRequest) returns (HelloResponse)

{

} Message HelloRequest {

;String name = 1

{

} Message HelloResponse {

;String message = 1

{
```

تولید کدهای gRPC:

با استفاده از ابزار grpcio-tools، کدهای لازم را از فایل proto تولید کنید:

Bash

```
Python -m grpc_tools.protoc -I. –python_out=. –grpc_python_out=. Example.proto
```

نوشتن سرور gRPC:

در این مرحله، شما باید یک سرور gRPC بنویسید که خدمات را پیاده‌سازی کند. یک فایل server.py ایجاد کنید:

Python

```
Import grpc

From concurrent import futures

Import time

Import example_pb2

Import example_pb2_grpc

:Class Greeter(example_pb2_grpc.GreeterServicer)
```

```
:Def SayHello(self, request, context)
Return example_pb2.HelloResponse(message='Hello, ' + request.name)
:()Def serve
    Server = grpc.server(futures.ThreadPoolExecutor(max_workers=10))
    Example_pb2_grpc.add_GreeterServicer_to_server(Greeter(), server)
    Server.add_insecure_port('[::]:50051')
    ()Server.start
    Print("Server is running on port 50051...")
    :Try
        :While True
            Time.sleep(86400)  # Keep the server running
    :Except KeyboardInterrupt
        Server.stop(0)
:'__If __name__ == '__main
    ()Serve
```

نوشتن کلاینت gRPC:

حالا یک کلاینت برای تست سرویس بنویسید. یک فایل client.py ایجاد کنید:

```
Python
Import grpc
Import example_pb2
Import example_pb2_grpc
:()Def run
    :With grpc.insecure_channel('localhost:50051') as channel
        Stub = example_pb2_grpc.GreeterStub(channel)
Response = stub.SayHello(example_pb2.HelloRequest(name='World'))
        Print("Greeter client received: " + response.message)
:'__If __name__ == '__main
    ()Run
```

اجرای سرور و کلاینت:

در یک ترمینال، سرور را اجرا کنید:

Bash

Python server.py

و در ترمینال دیگر، کلاینت را اجرا کنید:

Bash

Python client.py

۶. Socket programmin در زبان rust را ب همراه یک مثال بیان کنید؟

برنامه‌نویسی سوکت در زبان Rust به شما این امکان را می‌دهد که به راحتی با شبکه‌ها ارتباط برقرار کنید. Rust یک زبان امن و کارآمد است که برای توسعه نرم‌افزارهای شبکه‌ای بسیار مناسب است. در اینجا یک راهنمای ساده برای ایجاد یک سرور و کلاینت با استفاده از سوکت‌ها در Rust آورده شده است.

مراحل ایجاد یک برنامه سوکت در Rust

نصب Rust: ابتدا باید Rust را روی سیستم خود نصب کنید. می‌توانید از rustup استفاده کنید.

ایجاد یک پروژه جدید: با استفاده از دستور زیر یک پروژه جدید ایجاد کنید:

Bash

Cargo new socket_example

Cd socket_example

اضافه کردن وابستگی‌ها: برای کار با سوکت‌ها، شما نیاز به tokio crate و tokio-util دارید. این وابستگی‌ها را به فایل Cargo.toml اضافه کنید:

Toml

[dependencies]

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

نوشتن کد سرور: در فایل src/main.rs، کد زیر را برای سرور بنویسید:

```
Rust

;Use tokio::net::{TcpListener, TcpStream}
;Use tokio::io::{AsyncBufReadExt, AsyncWriteExt, BufReader}

[tokio::main]#
} ()Async fn main
;()Let listener = TcpListener::bind("127.0.0.1:8080").await.unwrap
;Println!("Server listening on 127.0.0.1:8080")

} Loop
;()Let (socket, _) = listener.accept().await.unwrap
;Tokio::spawn(handle_connection(socket))
{
{

} Async fn handle_connection(socket: TcpStream)
;Let mut reader = BufReader::new(&socket)
;()Let mut buffer = String::new

} While reader.read_line(&mut buffer).await.unwrap() > 0
;Println!("Received: {}", buffer.trim())
;()Buffer.clear
;"Let response = "Message received\n
;()Socket.write_all(response.as_bytes()).await.unwrap
{
{
```

نوشتن کد کلاینت: در یک فایل جدید به نام client.rs، کد زیر را بنویسید:

Rust

```
;Use tokio::net::TcpStream
;Use tokio::io::{AsyncWriteExt}

#[tokio::main]
} ()Async fn main {
;()Let mut stream = TcpStream::connect("127.0.0.1:8080").await.unwrap
;"Let message = "Hello, server!\n

;()Stream.write_all(message.as_bytes()).await.unwrap
;Println!("Message sent to the server")
{
```

اجرای سرور و کلاینت: ابتدا سرور را اجرا کنید:

Bash

```
Cargo run
```

سپس در یک ترمینال جدید، کلاینت را اجرا کنید:

Bash

```
Cargo run –bin client
```

توضیحات کد

سرور:

از TcpListener برای گوش دادن به اتصالات ورودی استفاده می‌شود.

وقتی یک اتصال جدید برقرار می‌شود، تابع handle_connection به صورت همزمان برای پردازش آن اتصال فراخوانی می‌شود.

سرور پیام‌های دریافتی را چاپ کرده و یک پاسخ ساده به کلاینت ارسال می‌کند.

کلاینت:

از TcpStream برای اتصال به سرور استفاده می‌شود.

یک پیام به سرور ارسال می‌کند و سپس به ترمینال برمی‌گردد.

نتیجه‌گیری

با این روش ساده، شما می‌توانید یک سرور و کلاینت پایه‌ای با استفاده از سوکت‌ها در Rust ایجاد کنید. این مثال می‌تواند به عنوان یک نقطه شروع برای پروژه‌های پیچیده‌تر شبکه‌ای در Rust استفاده شود.

۷.با استفاده از ORM در زبان rust برنامه ای بنویسید ک عملیات CRUD را بر روی یک پایگاه داده انجام دهد؟

برای نوشتن یک برنامه ساده در زبان Rust که از ORM (Object-Relational Mapping) برای انجام عملیات CRUD (Create, Read, Update, Delete) بر روی یک پایگاه داده استفاده کند، می‌توانیم از کتابخانه Diesel استفاده کنیم. Diesel یکی از محبوب‌ترین ORMها در Rust است و از SQLite، PostgreSQL و MySQL پشتیبانی می‌کند.

در اینجا یک مثال ساده از نحوه استفاده از Diesel برای انجام عملیات CRUD بر روی یک پایگاه داده SQLite ارائه می‌دهم.

مراحل ایجاد برنامه

ایجاد پروژه جدید:

ابتدا یک پروژه جدید با استفاده از Cargo ایجاد کنید:

Bash

```
Cargo new rust_crud_example

Cd rust_crud_example
```

اضافه کردن وابستگی‌ها:

در فایل Cargo.toml، وابستگی‌های Diesel و SQLite را اضافه کنید:

Toml

```
[dependencies]

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

"Dotenv = "0.15

ایجاد پایگاه داده:

با استفاده از Diesel CLI، پایگاه داده SQLite را ایجاد کنید. ابتدا Diesel CLI را نصب کنید:

Bash

```
Cargo install diesel_cli –no-default-features –features sqlite
```

سپس یک پایگاه داده جدید ایجاد کنید:

Bash

```
Diesel setup
```

تعریف مدل و جدول:

یک فایل جدید به نام schema.rs در پوشه src ایجاد کنید و جدول و مدل را تعریف کنید:

Rust

```
// src/schema.rs

table! {

    users (id) {

        id -> Integer,

        name -> Text,

        age -> Integer,

    }

}
```

و مدل کاربر را در main.rs تعریف کنید:

Rust

```
// src/main.rs

#[macro_use]

extern crate diesel;

mod schema;

use diesel::prelude::*;

use std::env;

#[derive(Queryable, Insertable)]
```

```
#["table_name = "users]
} Struct User
    ,Id: i32
    ,Name: String
    ,Age: i32
{

// تنظیمات پایگاه داده
} Fn establish_connection() -> SqliteConnection {
    ;"Let database_url = "db.sqlite // نام پایگاه داده
    SqliteConnection::establish(&database_url)
        .expect(&format!("Error connecting to {}", database_url))
}
```

عملیات CRUD:

حالا می‌توانیم توابعی برای عملیات CRUD ایجاد کنیم:

```
Rust
// src/main.rs
;*::Use diesel::prelude
;*::Use schema::users::dsl

} Fn create_user(conn: &SqliteConnection, name: &str, age: i32)
    } Let new_user = User
        Id: 0, // Id به صورت خودکار افزایش می‌یابد
        ,()Name: name.to_string
        ,Age
    ;{
    Diesel::insert_into(users)
        .values(&new_user)
        .execute(conn)
```

```
        .expect("Error inserting new user");
}

fn read_users(conn: &SqliteConnection) -> Vec<User> {
    users.load::<User>(conn).expect("Error loading users")
}

fn update_user(conn: &SqliteConnection, user_id: i32, new_name: &str) {
    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) {
    diesel::delete(users.find(user_id))
        .execute(conn)
        .expect("Error deleting user");
}
```

استفاده از توابع CRUD:

در نهایت، توابع CRUD را در تابع main فراخوانی کنید:

Rust

```
fn main() {
    let connection = establish_connection();

    // ایجاد کاربر
    create_user(&connection, "Alice", 30);
    create_user(&connection, "Bob", 25);

    // خواندن کاربران
    let users = read_users(&connection);
    for user in users {
        println!("ID: {}, Name: {}, Age: {}", user.id, user.name, user.age);
```

```
{
// به‌روزرسانی کاربر
;Update_user(&connection, 1, "Alice Smith")
// حذف کاربر
;Delete_user(&connection, 2)
{
```

اجرای برنامه

برای اجرای برنامه، از دستور زیر استفاده کنید:

Bash

Cargo run

۸.مفهوم Regular Expression چیست؟در زبان rust با بیان یک مثال توضیح دهید؟

مفهوم Regular Expression (عبارات منظم)

عبارات منظم (Regular Expressions یا به اختصار Regex) الگوهایی هستند که برای جستجو و تطبیق الگوهای متنی در رشته‌ها استفاده می‌شوند. این ابزارها به شما امکان می‌دهند تا رشته‌های متنی را بر اساس الگوهای خاصی پیدا کنید، آن‌ها را تغییر دهید یا تجزیه و تحلیل کنید. عبارات منظم در بسیاری از زبان‌های برنامه‌نویسی و ابزارهای متنی مورد استفاده قرار می‌گیرند.

ساختار عبارات منظم

عبارات منظم شامل کاراکترها و نمادهای خاصی هستند که معنای خاصی دارند. به عنوان مثال:

. به معنای هر کاراکتر است.

- به معنای صفر یا بیشتر از کاراکتر قبلی است.

+ به معنای یک یا بیشتر از کاراکتر قبلی است.

? به معنای صفر یا یک بار از کاراکتر قبلی است.

[] برای تعیین یک مجموعه از کاراکترها استفاده می‌شود.

() برای گروه‌بندی استفاده می‌شود.

مثال در زبان Rust

در زبان Rust، برای کار با عبارات منظم می‌توانیم از کتابخانه regex استفاده کنیم. در زیر یک مثال ساده از نحوه استفاده از عبارات منظم در Rust آورده شده است:

Rust

```
;Use regex::Regex


} ()Fn main {
    // تعریف یک الگوی عبارات منظم
    ;()Let re = Regex::new(r"(\w+)@(\w+)\.(\w+)").unwrap;

    // یک رشته برای جستجو
    ;"لطفاً به آدرس email@example.com ایمیل بزنید." = Let text

    // جستجو در رشته
    } Match re.captures(text) {
        } <= Some(caps) {
            ;("آدرس ایمیل پیدا شد:")!Println
            ;(caps[1]& ,"{}" ,"کاربر: ")!Println
            ;(caps[2]& ,"{}" ,"دامنه: ")!Println
            ;(caps[3]& ,"{}" ,"پسوند: ")!Println
        },{
        ,("هیچ آدرسی پیدا نشد.")!None => println
    }
}
```

توضیحات مثال:

وارد کردن کتابخانه: با استفاده از use regex::Regex; کتابخانه regex را وارد می‌کنیم.

تعریف الگو: با Regex::new(r"(\w+)@(\w+)\.(\w+)") یک الگوی عبارات منظم تعریف می‌کنیم که به دنبال آدرس‌های ایمیل می‌گردد. این الگو شامل سه گروه است:

(\w+): کاراکترهای الفبا و عدد که نمایانگر نام کاربری است.

@: کاراکتر @ که در آدرس ایمیل وجود دارد.

(\w+): دامنه (مثل example).

\.: کاراکتر نقطه.

(\w+): پسوند (مثل com).

جستجو در رشته: با re.captures(text) در رشته جستجو می‌کنیم و در صورت پیدا شدن الگو، اطلاعات را چاپ می‌کنیم.

این مثال نشان می‌دهد که چگونه می‌توان از عبارات منظم برای استخراج اطلاعات خاص از رشته‌ها در زبان Rust استفاده کرد.

۹.عملکرد کتابخانه windwos و native-windwos-gui در زبان rust چیست؟با ذکر یک مثال توضیح دهید.

کتابخانه‌های windows و native-windows-gui در زبان Rust به توسعه‌دهندگان این امکان را می‌دهند که برنامه‌های کاربردی ویندوزی بسازند. هر کدام از این کتابخانه‌ها ویژگی‌های خاص خود را دارند و برای مقاصد مختلف مناسب‌اند.

1. کتابخانه windows

کتابخانه windows یک رابط کاربری برای APIهای ویندوز است که به شما اجازه می‌دهد تا از قابلیت‌های مختلف ویندوز مانند WinRT، COM و APIهای سنتی ویندوز استفاده کنید. این کتابخانه به شما این امکان را می‌دهد که به طور مستقیم با APIهای ویندوز کار کنید و قابلیت‌های پیچیده‌تری را پیاده‌سازی کنید.

2. کتابخانه native-windows-gui

کتابخانه native-windows-gui یک لایه بالاتر از APIهای ویندوز است که طراحی شده تا توسعه برنامه‌های GUI را آسان‌تر کند. این کتابخانه به شما اجازه می‌دهد تا به راحتی پنجره‌ها، دکمه‌ها و دیگر عناصر GUI را ایجاد کنید بدون اینکه نیاز به کار مستقیم با APIهای پیچیده ویندوز داشته باشید.

مثال

در اینجا یک مثال ساده با استفاده از native-windows-gui آورده شده است که یک پنجره با یک دکمه ایجاد می‌کند:

Rust

```
;Use native_windows_gui as nwg

} ()Fn main
    // راه‌اندازی کتابخانه
    ;Nwg::init().expect("Failed to initialize Native Windows GUI")

    // ایجاد یک پنجره
    ()Let main_window = nwg::Window::builder
        ("مثال پنجره")title.
        size((300, 200)).
        ()build.
    ;expect("Failed to build main window").

    // ایجاد یک دکمه
    ()Let button = nwg::Button::builder
        ("کلیک کن")text.
        parent(&main_window).
        ()build.
    ;expect("Failed to build button").

    // تعریف رفتار دکمه
    } |_| Nwg::bind_event_handler(&button, nwg::Event::OnClick, move {
        ;("دکمه کلیک شد!")Println!
    ;expect("Failed to bind event handler").({

    // نمایش پنجره
```

```
    Nwg::dispatch_thread_events();
}
```

توضیحات کد:

راه‌اندازی: با nwg::init() کتابخانه را راه‌اندازی می‌کنیم.

ایجاد پنجره: با استفاده از Window::builder() یک پنجره جدید ایجاد می‌کنیم.

ایجاد دکمه: با Button::builder() یک دکمه به پنجره اضافه می‌کنیم.

تعریف رفتار دکمه: با استفاده از bind_event_handler یک رویداد برای دکمه تعریف می‌کنیم که وقتی دکمه کلیک شد، یک پیام را در کنسول چاپ می‌کند.

نمایش پنجره: با dispatch_thread_events() رویدادها را مدیریت می‌کنیم و پنجره را نمایش می‌دهیم.

این کد یک مثال ساده از نحوه استفاده از کتابخانه native-windows-gui برای ایجاد یک برنامه GUI در Rust است.

۱۰.برنامه ای برای انجام یک پردازش ساده بر روی یک Audio بنویسید؟

برای انجام یک پردازش ساده بر روی یک فایل صوتی، می‌توانیم از زبان برنامه‌نویسی پایتون و کتابخانه‌هایی مانند pydub و numpy استفاده کنیم. در اینجا یک برنامه ساده برای کاهش حجم صدا (normalize) یک فایل صوتی آورده شده است. این برنامه به شما اجازه می‌دهد تا یک فایل صوتی را بارگذاری کرده و آن را نرمال‌سازی کنید.

مراحل انجام کار:

نصب کتابخانه‌های مورد نیاز:

ابتدا باید کتابخانه‌های pydub و ffmpeg را نصب کنید. می‌توانید این کار را با استفاده از pip انجام دهید:

Bash

```
Pip install pydub
```

همچنین، برای استفاده از pydub به ffmpeg نیاز دارید. می‌توانید آن را از سایت ffmpeg دانلود و نصب کنید.

نوشتن برنامه:

در اینجا یک کد ساده برای نرمال‌سازی یک فایل صوتی آورده شده است:

```
Python

From pydub import AudioSegment

# تابعی برای نرمال‌سازی صدا

:Def normalize_audio(file_path, output_path)

    # بارگذاری فایل صوتی

    Audio = AudioSegment.from_file(file_path)

    # نرمال‌سازی صدا

    ()Normalized_audio = audio.normalize

    # ذخیره فایل نرمال‌شده

    Normalized_audio.export(output_path, format="mp3")

# مسیر فایل ورودی و خروجی

"Input_file = "input_audio.mp3  # نام فایل صوتی ورودی

"Output_file = "normalized_audio.mp3  # نام فایل صوتی نرمال‌شده

# فراخوانی تابع نرمال‌سازی

Normalize_audio(input_file, output_file)

Print("فایل صوتی نرمال‌شده با موفقیت ذخیره شد.")
```

توضیحات کد:

بارگذاری فایل صوتی: با استفاده از AudioSegment.from_file() فایل صوتی بارگذاری می‌شود.

نرمال‌سازی صدا: متد normalize() به طور خودکار سطح صدا را تنظیم می‌کند تا به حداکثر سطح ممکن برسد بدون اینکه موجب ایجاد اعوجاج شود.

ذخیره فایل نرمال‌شده: با استفاده از export() فایل نرمال‌شده در فرمت مورد نظر (در اینجا MP3) ذخیره می‌شود.

نکات اضافی:

اطمینان حاصل کنید که فایل ورودی (input_audio.mp3) در مسیر صحیح قرار دارد.

می‌توانید فرمت فایل خروجی را به دلخواه تغییر دهید (مثل WAV یا FLAC) با تغییر پارامتر format در تابع export().

۱۱.برنامه ای برای دانلود یک فایل از Internet بنویسید؟

برای نوشتن یک برنامه ساده برای دانلود یک فایل از اینترنت، می‌توانید از زبان‌های برنامه‌نویسی مختلفی استفاده کنید. در اینجا یک مثال با استفاده از زبان Python و کتابخانه requests ارائه می‌شود. این کتابخانه به شما اجازه می‌دهد تا به راحتی درخواست‌های HTTP را ارسال کنید و فایل‌ها را دانلود کنید.

مراحل کار:

نصب کتابخانه requests:

اگر هنوز این کتابخانه را نصب نکرده‌اید، می‌توانید با استفاده از pip آن را نصب کنید:

Bash

```
Pip install requests
```

نوشتن کد برای دانلود فایل:

در اینجا یک کد ساده برای دانلود یک فایل از اینترنت آورده شده است:

Python

```
Import requests

:Def download_file(url, destination)
    :Try
        # ارسال درخواست GET به URL
        Response = requests.get(url, stream=True)
        Response.raise_for_status()  # بررسی وضعیت پاسخ
```

```
# نوشتن داده‌ها به فایل
With open(destination, 'wb') as file:
    For chunk in response.iter_content(chunk_size=8192):
        File.write(chunk)

    Print(f"فایل با موفقیت دانلود شد: {destination}")
except requests.exceptions.RequestException as e:
    Print(f"خطا در دانلود فایل: {e}")

# مثال استفاده
File_url = 'https://example.com/path/to/your/file.txt'  # URL فایل
Destination_path = 'file.txt'  # مسیر ذخیره فایل
Download_file(file_url, destination_path)
```

توضیحات کد:

وارد کردن کتابخانه: ابتدا کتابخانه requests را وارد می‌کنیم.

تابع download_file: این تابع دو ورودی می‌گیرد: url (آدرس فایل) و destination (مسیر ذخیره فایل).

ارسال درخواست GET: با استفاده از requests.get به URL درخواست می‌فرستیم و داده‌ها را به صورت جریانی (stream) دریافت می‌کنیم.

نوشتن داده‌ها به فایل: با استفاده از یک حلقه، داده‌ها را به صورت بخش به بخش (chunk) به فایل نوشته می‌شود.

مدیریت خطا: در صورت بروز خطا در دانلود، پیام خطا چاپ می‌شود.

نکات:

اطمینان حاصل کنید که URL صحیح است و به یک فایل قابل دانلود اشاره می‌کند.

می‌توانید اندازه chunk_size را تغییر دهید تا کنترل بیشتری بر روی اندازه بخش‌های دانلود شده داشته باشید.

برای دانلود فایل‌های بزرگ، استفاده از stream=True به شما کمک می‌کند تا از مصرف حافظه بیش از حد جلوگیری کنید.

۱۲.برنامه ای برای خواندن و نوشتن یک فایل Excel ساده بنویسید؟

برای خواندن و نوشتن یک فایل Excel ساده در زبان برنامه‌نویسی پایتون، می‌توانید از کتابخانه pandas و openpyxl استفاده کنید. این کتابخانه‌ها ابزارهای قدرتمندی برای کار با داده‌ها و فایل‌های Excel هستند. در زیر یک مثال ساده برای خواندن و نوشتن یک فایل Excel آورده شده است.

مراحل نصب کتابخانه‌ها

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

Bash

```
Pip install pandas openpyxl
```

کد برای خواندن و نوشتن فایل Excel

در این مثال، ما یک فایل Excel ایجاد می‌کنیم، داده‌هایی به آن اضافه می‌کنیم و سپس این داده‌ها را می‌خوانیم.

Python

```
Import pandas as pd

# ایجاد یک DataFrame ساده
} = Data
    'نام': ['علی', 'زهرا', 'محمد'],
    'سن': [25, 30, 22],
    'شغل': ['مهندس', 'دکتر', 'هنرمند']
{

Df = pd.DataFrame(data)

# نوشتن DataFrame به یک فایل Excel
'File_name = 'data.xlsx
Df.to_excel(file_name, index=False)
```

Print(f'فایل {file_name} با موفقیت ایجاد شد.')

خواندن داده‌ها از فایل Excel

Df_read = pd.read_excel(file_name)

Print('داده‌های خوانده شده از فایل Excel:')

Print(df_read)

توضیحات کد

وارد کردن کتابخانه‌ها: ابتدا کتابخانه pandas را وارد می‌کنیم.

ایجاد DataFrame: داده‌ها را در یک دیکشنری تعریف کرده و سپس آن را به یک DataFrame تبدیل می‌کنیم.

نوشتن به فایل Excel: با استفاده از متد DataFrame ،to_excel را به یک فایل Excel می‌نویسیم. Index=False به این معنی است که شماره‌گذاری ردیف‌ها در فایل ذخیره نمی‌شود.

خواندن از فایل Excel: با استفاده از متد read_excel، داده‌ها را از فایل Excel خوانده و در یک DataFrame جدید ذخیره می‌کنیم.

چاپ داده‌ها: در نهایت، داده‌های خوانده شده را چاپ می‌کنیم.

۱۳.معماری Claen Architecture رادر قالب یک برنامه پیاده سازی کنید؟

معماری Clean Architecture (معماری تمیز) یک الگوی طراحی نرم‌افزاری است که هدف آن ایجاد سیستم‌هایی با قابلیت نگهداری و توسعه آسان‌تر است. این معماری بر اساس اصول SOLID و جداسازی نگرانی‌ها (Separation of Concerns) بنا شده و به توسعه‌دهندگان این امکان را می‌دهد که اجزای مختلف سیستم را به‌طور مستقل مدیریت کنند.

برنامه‌ریزی پیاده‌سازی Clean Architecture

1. تعریف لایه‌ها

معماری Clean به طور کلی شامل چهار لایه اصلی است:

لایه Entities (موجودیت‌ها):

شامل قوانین کسب‌وکار و موجودیت‌های اصلی سیستم است.

این لایه مستقل از سایر لایه‌ها است.

لایه Use Cases (موارد استفاده):

شامل منطق کسب‌وکار و موارد استفاده (Use Cases) است.

این لایه تعاملات بین موجودیت‌ها و سیستم را مدیریت می‌کند.

لایه Interface Adapters (مبدل‌های رابط):

شامل مبدل‌هایی است که داده‌ها را از فرمت‌های مختلف (مثل API، پایگاه داده، UI) به فرمت‌های قابل استفاده برای لایه‌های داخلی تبدیل می‌کند.

این لایه شامل کنترلرها و نمایندگان است.

لایه Frameworks and Drivers (فریم‌ورک‌ها و درایورها):

شامل فریم‌ورک‌ها و ابزارهای خارجی است که در پروژه استفاده می‌شوند.

این لایه به لایه‌های داخلی وابسته است و نباید بر روی آن‌ها تأثیر بگذارد.

2. ساختار پروژه

یک ساختار پیشنهادی برای پروژه می‌تواند به شکل زیر باشد:

Bash

```
project-root/
    src/
        entities/
        usecases/
        adapters/
            controllers/
            presenters/
        frameworks/
            database/
            api/
```

3. تعریف موجودیت‌ها

در لایه موجودیت‌ها، موجودیت‌های اصلی سیستم را تعریف کنید. به عنوان مثال، اگر یک سیستم مدیریت کتابخانه دارید، ممکن است موجودیت‌هایی مانند Book و User داشته باشید.

Python

```
entities/book.py #

:Class Book

    :Def __init__(self, title, author)

        Self.title = title

        Self.author = author
```

4. تعریف موارد استفاده

در لایه موارد استفاده، منطق کسب‌وکار را پیاده‌سازی کنید. به عنوان مثال، یک مورد استفاده برای اضافه کردن کتاب به کتابخانه.

Python

```
usecases/add_book.py #

:Class AddBook

    :Def __init__(self, book_repository)

        Self.book_repository = book_repository

    :Def execute(self, book)

        Self.book_repository.add(book)
```

5. تعریف مبدل‌های رابط

در این لایه، کنترلرها و مبدل‌ها را پیاده‌سازی کنید.

Python

```
adapters/controllers/book_controller.py #

:Class BookController

    :Def __init__(self, add_book_use_case)

        Self.add_book_use_case = add_book_use_case

    :Def add_book(self, title, author)

        Book = Book(title, author)

        Self.add_book_use_case.execute(book)
```

6. تعریف فریم‌ورک‌ها و درایورها

در این لایه، می‌توانید از فریم‌ورک‌های مختلف برای پیاده‌سازی پایگاه داده یا API استفاده کنید.

Python

```
# frameworks/database/book_repository.py

Class BookRepository:

    Def add(self, book):

        # کد برای اضافه کردن کتاب به پایگاه داده

        Pass
```

7. تست و پیاده‌سازی

پایان