بسم الله الرحمن الرحیم

*****

پروژه درس برنامه نویسی سیستمی

استاد مربوطه : استاد احمد زاده

*****

دانشجو زهرا کریمی اصل

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

ndarray: این کتابخانه برای محاسبات عددی و کار با آرایه‌های چندبعدی استفاده می‌شود و پایه‌ای برای بسیاری از الگوریتم‌های یادگیری ماشین است.

linfa: این کتابخانه یک چارچوب یادگیری ماشین برای Rust است که شامل الگوریتم‌های مختلف یادگیری نظارت شده و غیرنظارت شده می‌باشد.

tch-rs: این کتابخانه یک رابط برای کتابخانه PyTorch است و به شما امکان می‌دهد از قابلیت‌های PyTorch در Rust استفاده کنید.

rustlearn: این کتابخانه یک کتابخانه یادگیری ماشین ساده و کاربرپسند است که شامل الگوریتم‌های مختلف یادگیری ماشین می‌باشد.

smartcore: این کتابخانه شامل الگوریتم‌های مختلف یادگیری ماشین و ابزارهای تحلیلی است.

مثال ساده با استفاده از کتابخانه ndarray

```
rust

;use ndarray::{Array, Array2, Array1}

;use ndarray_stats::correlation::Pearson

} ()fn main

(y) و خروجی (X) داده‌های ورودی //

;()let x = Array2::from_shape_vec((5, 1), vec![1.0, 2.0, 3.0, 4.0, 5.0]).unwrap

;let y = Array1::from_vec(vec![2.0, 3.0, 5.0, 7.0, 11.0])

محاسبه میانگین //

;()let x_mean = x.mean().unwrap

;()let y_mean = y.mean().unwrap

محاسبه ضریب همبستگی //

;()let covariance = ((&x - x_mean) * (&y - y_mean)).mean().unwrap

;()let x_variance = (&x - x_mean).mapv(|v| v * v).mean().unwrap


(intercept) محاسبه شیب (slope) و عرض از مبدأ //

;let slope = covariance / x_variance

;let intercept = y_mean - slope * x_mean
```

;(y = {} * x + {}", slope, intercept :مدل رگرسیون خطی")!println

{

2) برنامه نویسی Parallel Programming در زبان Rustرا با ذکر یک مثال ساده توضیح دهید؟ برنامه‌نویسی موازی (Parallel Programming) به معنای اجرای همزمان چندین بخش از یک برنامه است تا کارایی و سرعت اجرای آن افزایش یابد. زبان Rust به دلیل ویژگی‌های خاص خود مانند ایمنی حافظه و مدیریت همزمانی (Concurrency) به یکی از گزینه‌های محبوب برای برنامه‌نویسی موازی تبدیل شده است.

مفهوم برنامه‌نویسی موازی در Rust

Rust از مفهوم "رشته‌ها" (Threads) برای پیاده‌سازی برنامه‌نویسی موازی استفاده می‌کند. هر رشته می‌تواند به‌طور مستقل از سایر رشته‌ها اجرا شود و می‌تواند داده‌ها را به اشتراک بگذارد. Rust با استفاده از سیستم مالکیت (Ownership) و وام‌دهی (Borrowing) خود، از بروز مشکلاتی مانند شرایط رقابتی (Race Conditions) جلوگیری می‌کند.

مثال ساده

در این مثال، ما یک برنامه ساده می‌نویسیم که از چندین رشته برای محاسبه مجموع اعداد از ۱ تا ۱۰۰۰ استفاده می‌کند. ما این کار را با تقسیم کار بین چندین رشته انجام می‌دهیم.

rust

;use std::thread

} ()fn main

;[]!let mut handles = vec

// تقسیم کار به ۴ رشته

} for i in 0..4

} || let handle = thread::spawn(move

// شروع هر رشته ;let start = i * 250 + 1

// پایان هر رشته ;let end = (i + 1) * 250

// محاسبه مجموع ;()let sum: u32 = (start..=end).sum

sum

;({

;handles.push(handle)

{

;let mut total_sum = 0

// جمع‌آوری نتایج از رشته‌ها

```
} for handle in handles
انتظار برای اتمام رشته و جمع کردن نتایج // ;()total_sum += handle.join().unwrap
{
;println!("Total sum is: {}", total_sum)
{
```

3) Lazy Loading چیست؟ با ذکر مثال در زبان Rust توضیح دهید؟ یک الگوی طراحی است که در آن یک شیء یا منبع تنها زمانی بارگذاری می‌شود که به آن نیاز است، به جای بارگذاری آن در زمان آغاز برنامه یا در زمان شروع یک فرایند. این الگو می‌تواند به بهینه‌سازی مصرف حافظه و زمان بارگذاری کمک کند و در مواردی که بارگذاری اولیه منابع زمان‌بر است، مفید باشد.

مثال در زبان Rust

در زبان Rust، می‌توانیم از ویژگی‌های مالکیت و مدیریت حافظه استفاده کنیم تا یک پیاده‌سازی ساده از Lazy Loading را ایجاد کنیم. در این مثال، ما یک ساختار ساده برای بارگذاری یک داده سنگین (مانند یک تصویر یا یک فایل) را پیاده‌سازی می‌کنیم. ما از Option برای نگهداری داده‌ها استفاده می‌کنیم تا بتوانیم مشخص کنیم که آیا داده بارگذاری شده است یا خیر.

```
rust
;use std::sync::{Arc, Mutex}
} struct LazyLoader
,<<<data: Arc<Mutex<Option<String
{
} impl LazyLoader
} fn new() -> Self
} LazyLoader
,data: Arc::new(Mutex::new(None))
{
{
} fn load(&self)
;()let mut data_lock = self.data.lock().unwrap
} ()if data_lock.is_none
(شبیه‌سازی با یک رشته) بارگذاری داده //
;println!("Loading data...")
;data_lock = Some("Heavy Data Loaded".to_string())*
{
```

```
{
} <fn get_data(&self) -> Option<String
;()let data_lock = self.data.lock().unwrap
()data_lock.clone
{
{
} ()fn main
;()let lazy_loader = LazyLoader::new
// در اینجا داده بارگذاری نمی‌شود
;println!("Data not loaded yet.")
// بارگذاری داده
;()lazy_loader.load
// دریافت داده
} ()if let Some(data) = lazy_loader.get_data
;println!("Data: {}", data)
{
{
```

توضیحات کد

ساختار LazyLoader: این ساختار شامل یک فیلد data است که به صورت Arc<Mutex<Option<String>>> تعریف شده است. Arc برای به اشتراک‌گذاری ایمن داده‌ها بین چندین ترد و Mutex برای قفل کردن داده‌ها در هنگام دسترسی استفاده می‌شود.

متد load: این متد چک می‌کند که آیا داده‌ها قبلاً بارگذاری شده‌اند یا خیر. اگر نه، داده‌ها بارگذاری می‌شوند (در اینجا به صورت شبیه‌سازی با یک رشته).

متد get_data: این متد داده‌ها را برمی‌گرداند. اگر داده‌ها بارگذاری نشده باشند، None برمی‌گرداند.

تابع main: در اینجا، ما یک نمونه از LazyLoader ایجاد می‌کنیم، داده‌ها را بارگذاری نمی‌کنیم و سپس بارگذاری داده‌ها و دریافت آن‌ها را انجام می‌دهیم.

4) ساختمان داده Binary Search Tree در زبان Rust پیاده سازی نمایید ؟ پیاده‌سازی یک درخت جستجوی دودویی (Binary Search Tree یا BST) در زبان Rust می‌تواند شامل مراحل زیر باشد:

تعریف ساختار داده: ابتدا باید یک ساختار برای نودهای درخت تعریف کنیم.

پیاده‌سازی توابع اصلی: توابعی برای افزودن، جستجو و حذف نودها در درخت.

توابع کمکی: توابعی برای پیمایش درخت و نمایش آن.

در زیر یک پیاده‌سازی ساده از BST در Rust آورده شده است:

```
rust
;use std::cmp::Ord
;use std::boxed::Box
;use std::option::Option

[derive(Debug)]#
} <pub struct Node<T
,value: T
,<<<left: Option<Box<Node<T
,<<<right: Option<Box<Node<T
{

} <impl<T: Ord> Node<T
} pub fn new(value: T) -> Self
} Node
,value
,left: None
,right: None
{
{

} pub fn insert(&mut self, value: T)
} if value < self.value
} if let Some(ref mut left) = self.left
;left.insert(value)
} else {
;self.left = Some(Box::new(Node::new(value)))
```

```
{
} else if value > self.value {
} if let Some(ref mut right) = self.right
;right.insert(value)
} else {
;self.right = Some(Box::new(Node::new(value)))
{
{
{

} pub fn contains(&self, value: T) -> bool
} if value < self.value
self.left.as_ref().map_or(false, |left| left.contains(value))
} else if value > self.value {
self.right.as_ref().map_or(false, |right| right.contains(value))
} else {
true
{
{

} pub fn in_order(&self)
} if let Some(ref left) = self.left
;()left.in_order
{
;println!("{}", self.value)
} if let Some(ref right) = self.right
;()right.in_order
{
{
```

```
{


} <pub struct BinarySearchTree<T
,<<root: Option<Node<T
{


} <impl<T: Ord> BinarySearchTree<T
} pub fn new() -> Self
BinarySearchTree { root: None }
{


} pub fn insert(&mut self, value: T)
} match self.root
,Some(ref mut node) => node.insert(value)
,None => self.root = Some(Node::new(value))
{
{


} pub fn contains(&self, value: T) -> bool
} match &self.root
,Some(node) => node.contains(value)
,None => false
{
{


} pub fn in_order(&self)
} if let Some(ref node) = self.root
;()node.in_order
{
```

```
{
{


} ()fn main
;()let mut bst = BinarySearchTree::new
;(5)bst.insert
;(3)bst.insert
;(7)bst.insert
;(2)bst.insert
;(4)bst.insert
;(6)bst.insert
;(8)bst.insert


;println!("In-order traversal of the BST:")
;()bst.in_order


;let search_value = 4
;search_value, bst.contains(search_value)) ,"{} ?{} println!("Does the BST contain
{
```

5) ساختمان داده AVL Tree را در زبان Rust پیاده سازی نمایید؟در اینجا یک پیاده‌سازی ساده از درخت AVL (درخت جستجوی متعادل) در زبان Rust ارائه می‌شود. درخت AVL نوعی درخت جستجوی دودویی است که در آن ارتفاع دو زیر درخت هر گره می‌تواند حداکثر یک واحد تفاوت داشته باشد. این ویژگی باعث می‌شود که درخت همیشه متعادل باقی بماند و عملیات جستجو، درج و حذف با زمان اجرای O(log n) انجام شود.

پیاده‌سازی درخت AVL در Rust

```
rust
[derive(Debug)]#
} <pub struct AVLNode<T
```

```
,pub value: T
,pub height: isize
,<<<pub left: Option<Box<AVLNode<T
,<<<pub right: Option<Box<AVLNode<T
{

} <impl<T: Ord> AVLNode<T
} pub fn new(value: T) -> Self
} AVLNode
,value
,height: 1
,left: None
,right: None
{
{

} pub fn height(node: &Option<Box<AVLNode<T>>>) -> isize
} match node
,Some(n) => n.height
,None => 0
{
{

} pub fn balance_factor(node: &Option<Box<AVLNode<T>>>) -> isize
} match node
,Some(n) => Self::height(&n.left) - Self::height(&n.right)
,None => 0
{
{
```

```
} <<pub fn rotate_right(y: Box<AVLNode<T>>) -> Box<AVLNode<T

;()let x = y.left.unwrap

;let t2 = x.right


} let mut new_root = AVLNode

,value: x.value

,height: 0

,left: Some(x.left)

,right: Some(Box::new(y))

;{


;new_root.right.as_mut().unwrap().left = t2

new_root.right.as_mut().unwrap().height = 1 +
std::cmp::max(Self::height(&new_root.right.as_ref().unwrap().left),
;Self::height(&new_root.right.as_ref().unwrap().right))

;new_root.height = 1 + std::cmp::max(Self::height(&new_root.left), Self::height(&new_root.right))


Box::new(new_root)

{


} <<pub fn rotate_left(x: Box<AVLNode<T>>) -> Box<AVLNode<T

;()let y = x.right.unwrap

;let t2 = y.left


} let mut new_root = AVLNode

,value: y.value

,height: 0

,left: Some(Box::new(x))

,right: t2
```

```
;{


;new_root.left.as_mut().unwrap().right = t2

new_root.left.as_mut().unwrap().height = 1 +
std::cmp::max(Self::height(&new_root.left.as_ref().unwrap().left),
;Self::height(&new_root.left.as_ref().unwrap().right))

;new_root.height = 1 + std::cmp::max(Self::height(&new_root.left), Self::height(&new_root.right))


Box::new(new_root)

{


} <<pub fn insert(node: Option<Box<AVLNode<T>>>, value: T) -> Box<AVLNode<T

} let mut node = match node

} <= Some(n)

} if value < n.value

;let left_child = Self::insert(n.left, value)

;let mut n = *n

;n.left = Some(left_child)

;n.height = 1 + std::cmp::max(Self::height(&n.left), Self::height(&n.right))

Self::balance(n)

} else if value > n.value {

;let right_child = Self::insert(n.right, value)

;let mut n = *n

;n.right = Some(right_child)

;n.height = 1 + std::cmp::max(Self::height(&n.left), Self::height(&n.right))

Self::balance(n)

} else {

Duplicate values are not allowed in AVL Tree //

n

{
```

```
{
,None => Box::new(AVLNode::new(value))
;{

node
{

} <<pub fn balance(mut node: AVLNode<T>) -> Box<AVLNode<T
;let balance = Self::balance_factor(&Some(Box::new(node)))

Left Left Case //
} if balance > 1 && Self::balance_factor(&node.left) >= 0
;return Self::rotate_right(Box::new(node))
{

Right Right Case //
} if balance < -1 && Self::balance_factor(&node.right) <= 0
;return Self::rotate_left(Box::new(node))

Left Right Case //
} if balance > 1 && Self::balance_factor(&node.left) < 0
;node.left = Some(Self::rotate_left(node.left.unwrap()))
;return Self::rotate_right(Box::new(node))
{

Right Left Case //
} if balance < -1 && Self::balance_factor(&node.right) > 0
;node.right = Some(Self::rotate_right(node.right.unwrap()))
;return Self::rotate_left(Box::new(node))
```

```
{

Box::new(node)
{
{

} <pub struct AVLTree<T
,<<<root: Option<Box<AVLNode<T
{

} <impl<T: Ord> AVLTree<T
} pub fn new() -> Self
AVLTree { root: None }
{

} pub fn insert(&mut self, value: T)
;self.root = Some(AVLNode::insert(self.root.take(), value))
{

} pub fn in_order_traversal(&self)
;self.in_order_helper(&self.root)
{

} fn in_order_helper(&self, node: &Option<Box<AVLNode<T>>>)
} if let Some(n) = node
;self.in_order_helper(&n.left)
;println!("{}", n.value)
;self.in_order_helper(&n.right)
{
```

```
{
{


} ()fn main
;()let mut tree = AVLTree::new
;(10)tree.insert
;(20)tree.insert
;(30)tree.insert
;(40)tree.insert
;(50)tree.insert
;(25)tree.insert


;println!("In-order traversal of the AVL tree:")
;()tree.in_order_traversal
{
```

توضیحات

ساختار AVLNode: این ساختار نمایانگر هر گره در درخت AVL است و شامل مقدار، ارتفاع و اشارەگرهایی به فرزندان چپ و راست است.

عملیات چرخش: دو تابع rotate_left و rotate_right برای چرخش‌های چپ و راست درخت در هنگام عدم تعادل استفاده می‌شوند.

عملیات درج: تابع insert برای درج یک مقدار جدید به درخت استفاده می‌شود و در صورت نیاز تعادل درخت را برقرار می‌کند.

عملیات پیمایش: تابع in_order_traversal برای پیمایش درخت به صورت مرتب (In-order) و چاپ مقادیر استفاده می‌شود.

تابع main: در این تابع، یک درخت AVL ایجاد شده و مقادیری به آن اضافه می‌شود و سپس پیمایش درخت انجام می‌شود.

این پیاده‌سازی یک درخت AVL ساده است و می‌تواند به عنوان پایه‌ای برای توسعه بیشتر و افزودن ویژگی‌های دیگر مورد استفاده قرار گیرد

6) ساختمان داده Max heap- Tree را در زبان Rust پیاده سازی نمایید؟پیاده‌سازی یک Max Heap (درخت ماکس) در زبان Rust شامل ایجاد یک ساختار داده‌ای است که به شما اجازه می‌دهد تا عناصر را به صورت مرتب ذخیره کنید و به راحتی بزرگترین عنصر را استخراج کنید. در زیر، یک پیاده‌سازی ساده از Max Heap در Rust ارائه می‌شود.

مراحل پیاده‌سازی

تعریف ساختار MaxHeap: ابتدا یک ساختار برای MaxHeap تعریف می‌کنیم که شامل یک وکتور برای ذخیره عناصر است.

عملیات پایه: شامل افزودن عنصر، حذف بزرگترین عنصر و ساختن درخت از یک آرایه.

تابع‌های کمکی: برای حفظ خاصیت Max Heap (یعنی هر گره بزرگتر یا برابر با فرزندانش باشد) نیاز به توابعی برای بالا بردن و پایین آوردن عناصر داریم.

پیاده‌سازی

در زیر کد Rust برای Max Heap آورده شده است:

```
rust

} struct MaxHeap

,<data: Vec<i32

{


} impl MaxHeap

ایجاد یک MaxHeap جدید //

} fn new() -> Self

} ()MaxHeap { data: Vec::new

{


MaxHeap افزودن عنصر به //

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

;self.data.push(value)

;self.bubble_up(self.data.len() - 1)

{


MaxHeap حذف بزرگترین عنصر (ریشه) از //

} <fn extract_max(&mut self) -> Option<i32

} ()if self.data.is_empty

;return None

{

;[0]let max = self.data
```

```
;()let last = self.data.pop().unwrap
} ()if !self.data.is_empty
;self.data[0] = last
;(0)self.bubble_down
{
Some(max)
{
```

// بالا بردن عنصر جدید به موقعیت صحیح

```
} fn bubble_up(&mut self, index: usize)
;let mut idx = index
} while idx > 0
;let parent_idx = (idx - 1) / 2
} if self.data[idx] > self.data[parent_idx]
;self.data.swap(idx, parent_idx)
;idx = parent_idx
} else {
;break
{
{
{
```

// پایین آوردن عنصر به موقعیت صحیح

```
} fn bubble_down(&mut self, index: usize)
;let mut idx = index
;()let len = self.data.len
} loop
;let left_child = 2 * idx + 1
;let right_child = 2 * idx + 2
```

```
;let mut largest = idx


} if left_child < len && self.data[left_child] > self.data[largest]
;largest = left_child
{
} if right_child < len && self.data[right_child] > self.data[largest]
;largest = right_child
{
} if largest == idx
;break
{
;self.data.swap(idx, largest)
;idx = largest
{
{


MaxHeap نمایش عناصر //
} <fn peek(&self) -> Option<&i32
(0)self.data.get
{
{


} ()fn main
;()let mut heap = MaxHeap::new
;(10)heap.insert
;(20)heap.insert
;(5)heap.insert
;(30)heap.insert
```

باشد 30 باید // ;println!("Max: {:?}", heap.extract_max())

باشد 20 باید // ;println!("Next Max: {:?}", heap.peek())

{

توضیحات

ساختار MaxHeap: شامل یک وکتور data برای ذخیره عناصر است.

متد insert: عنصر جدید را به وکتور اضافه کرده و با استفاده از bubble_up موقعیت آن را اصلاح می‌کند.

متد extract_max: بزرگترین عنصر (ریشه) را حذف کرده و آخرین عنصر را به ریشه منتقل می‌کند و سپس با استفاده از bubble_down موقعیت آن را اصلاح می‌کند.

توابع bubble_up و bubble_down: برای حفظ خاصیت Max Heap استفاده می‌شوند.

متد peek: بزرگترین عنصر را بدون حذف آن برمی‌گرداند.

با این پیاده‌سازی، می‌توانید یک Max Heap ساده در Rust ایجاد کنید و از آن برای ذخیره و مدیریت داده‌ها استفاده کنید.

7) یک سرویس ساده جهت پردازش در خواست های مبتنی بر پروتکل gRPCبنویسید؟ برای ایجاد یک سرویس ساده با استفاده از پروتکل gRPC، مراحل زیر را دنبال می‌کنیم. ما از زبان برنامه‌نویسی Python و کتابخانه gRPC استفاده خواهیم کرد. این سرویس یک تابع ساده را پیاده‌سازی می‌کند که دو عدد را جمع می‌کند.

مراحل ایجاد سرویس gRPC

1. نصب کتابخانه‌های لازم

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

bash

pip install grpcio grpcio-tools

2. تعریف پروتکل gRPC

یک فایل با پسوند .proto ایجاد کنید. به عنوان مثال، calculator.proto:

protobuf

;"syntax = "proto3

;package calculator

// تعریف سرویس

} service Calculator

// تعریف متد جمع

;rpc Add (AddRequest) returns (AddResponse)

```
{
// تعریف پیام درخواست
} message AddRequest
;int32 a = 1
;int32 b = 2
{

// تعریف پیام پاسخ
} message AddResponse
;int32 result = 1
{
```

3. تولید کدهای gRPC

برای تولید کدهای Python از فایل .proto، از دستور زیر استفاده کنید:

bash

```
python -m grpc_tools.protoc -I. --python_out=. --grpc_python_out=. calculator.proto
```

این دستور دو فایل جدید به نام‌های calculator_pb2.py و calculator_pb2_grpc.py ایجاد می‌کند.

4. پیاده‌سازی سرویس

حالا یک فایل Python جدید ایجاد کنید، به عنوان مثال server.py و کد زیر را در آن قرار دهید:

```
python
import grpc
from concurrent import futures
import time
# وارد کردن کدهای تولید شده
import calculator_pb2
import calculator_pb2_grpc
# پیاده‌سازی سرویس
:class CalculatorServicer(calculator_pb2_grpc.CalculatorServicer)
:def Add(self, request, context)
```

```
result = request.a + request.b
return calculator_pb2.AddResponse(result=result)
:()def serve
server = grpc.server(futures.ThreadPoolExecutor(max_workers=10))
calculator_pb2_grpc.add_CalculatorServicer_to_server(CalculatorServicer(), server)
('50051:[::]')server.add_insecure_port
()server.start
print("Server is running on port 50051...")
:try
:while True
time.sleep(86400)  # Keep the server running
:except KeyboardInterrupt
(0)server.stop

:'if name == 'main
()serve
```

5. ایجاد کلاینت

حالا یک فایل دیگر به نام client.py ایجاد کنید و کد زیر را در آن قرار دهید:

```
python
import grpc
# وارد کردن کدهای تولید شده
import calculator_pb2
import calculator_pb2_grpc
:()def run
channel = grpc.insecure_channel('localhost:50051')
stub = calculator_pb2_grpc.CalculatorStub(channel)
# ایجاد درخواست
request = calculator_pb2.AddRequest(a=10, b=20)
# ارسال درخواست و دریافت پاسخ
```

```
response = stub.Add(request)

print("Result: ", response.result)

:'if name == 'main

()run
```

6. اجرای سرویس

برای اجرای سرویس، ابتدا سرور را راه‌اندازی کنید:

```
bash

python server.py
```

سپس در یک ترمینال جدید، کلاینت را اجرا کنید:

```
bash

python client.py
```

8) یک سرویس ساده جهت پردازش درخواست های مبتنی بر Web Assembly بنویسید؟ برای ایجاد یک سرویس ساده مبتنی بر WebAssembly (Wasm)، ابتدا باید چند مرحله کلیدی را دنبال کنیم. این مراحل شامل نوشتن کد WebAssembly، کامپایل آن، و سپس استفاده از آن در یک برنامه وب است. در اینجا یک راهنمای ساده برای ایجاد یک سرویس WebAssembly ارائه می‌شود.

مرحله 1: نوشتن کد WebAssembly

برای نوشتن کد WebAssembly، می‌توانیم از زبان‌هایی مانند Rust یا C استفاده کنیم. در اینجا یک مثال ساده با استفاده از Rust آورده شده است.

1.1 نصب ابزارهای لازم

ابتدا باید Rust و ابزارهای مربوط به WebAssembly را نصب کنید. این کار را با اجرای دستورات زیر در ترمینال انجام دهید:

```
bash

curl --proto '=https' --tlsv1.2 -sSf https://sh.rustup.rs | sh

source $HOME/.cargo/env

rustup target add wasm32-unknown-unknown
```

1.2 نوشتن کد

سپس یک پروژه جدید Rust ایجاد کنید:

bash

cargo new wasm_example --lib

cd wasm_example

در فایل Cargo.toml، وابستگی به wasm-bindgen را اضافه کنید:

toml

[lib]

crate-type = ["cdylib"]

[dependencies]

"wasm-bindgen = "0.2

حالا کد زیر را در فایل src/lib.rs قرار دهید:

rust

;*::use wasm_bindgen::prelude

[wasm_bindgen]#

} pub fn greet(name: &str) -> String

format!("Hello, {}!", name)

{

مرحله 2: کامپایل کد به WebAssembly

برای کامپایل کد Rust به WebAssembly، از دستور زیر استفاده کنید:

bash

wasm-pack build --target web

این دستور یک پوشه به نام pkg ایجاد می‌کند که شامل فایل‌های WebAssembly و JavaScript است.

مرحله 3: ایجاد یک برنامه وب

حالا یک برنامه وب ساده ایجاد می‌کنیم که از WebAssembly استفاده می‌کند.

3.1 ایجاد فایل HTML

یک فایل HTML به نام index.html ایجاد کنید و کد زیر را در آن قرار دهید:

```
html
<DOCTYPE html!>
<"html lang="en>
<head>
<"meta charset="UTF-8>
<"meta name="viewport" content="width=device-width, initial-scale=1.0>
<title>WebAssembly Example</title>
<head/>
<body>
<h1>WebAssembly Example</h1>
</ "input id="name" type="text" placeholder="Enter your name>
<button id="greetBtn">Greet</button>
<p id="greeting"></p>

<"script type="module>
'import init, { greet } from './pkg/wasm_example.js ;

} ()async function run
await init () ;
document.getElementById('greetBtn').onclick  <= () =
}
const name = document.getElementById('name').value ;
const greeting = greet(name);
```

```
document.getElementById('greeting').innerText = greeting;
{
{

()run ;
<script/>
<body/>
<html/>
```

مرحله 4: راه‌اندازی سرور محلی

برای مشاهده برنامه، می‌توانید از یک سرور محلی استفاده کنید. به عنوان مثال، می‌توانید از http-server استفاده کنید. ابتدا آن را نصب کنید:

```
bash
npm install -g http-server
```

سپس در پوشه پروژه، دستور زیر را اجرا کنید:

```
bash
. http-server
```

9) Socket Programming در زبان Rust را بهمراه یک مثال بیان کنید؟ برنامه‌نویسی سوکت (Socket Programming) در زبان Rust به شما این امکان را می‌دهد که برنامه‌هایی بسازید که بتوانند با شبکه‌ها ارتباط برقرار کنند. در زیر، یک توضیح کلی از نحوه کار با سوکت‌ها در Rust به همراه یک مثال ساده آورده شده است.

نصب کتابخانه‌های مورد نیاز

برای شروع، شما نیاز به کتابخانه tokio برای برنامه‌نویسی غیرهمزمان و tokio-tungstenite برای کار با وب‌سوکت‌ها دارید. برای این کار، می‌توانید این کتابخانه‌ها را به فایل Cargo.toml خود اضافه کنید:

```
toml
[dependencies]
tokio = { version = "1", features = ["full"] }
"tokio-tungstenite = "0.16
```

مثال: سرور و کلاینت ساده

در این مثال، یک سرور ساده و یک کلاینت خواهیم ساخت که با هم ارتباط برقرار می‌کنند.

سرور

ابتدا، سرور را پیاده‌سازی می‌کنیم:

```
rust
;use tokio::net::TcpListener
;use tokio::io::{AsyncReadExt, AsyncWriteExt}

[tokio::main]#
} <()>async fn main() -> std::io::Result
;?let listener = TcpListener::bind("127.0.0.1:8080").await
;println!("Server running on 127.0.0.1:8080")

} loop
;?let (mut socket, _) = listener.accept().await
} tokio::spawn(async move
;[1024 ;0] = let mut buffer
} match socket.read(&mut buffer).await
اتصال بسته شده // ,Ok(0) => return
} <= Ok(n)
;println!("Received: {}", String::from_utf8_lossy(&buffer[..n]))
;()socket.write_all(b"Hello from server!").await.unwrap
{
,Err(e) => eprintln!("Failed to read from socket; err = {:?}", e)
{
;({
{
{
```

کلاینت

حالا کلاینت را پیاده‌سازی می‌کنیم:

```
rust

;use tokio::net::TcpStream

;use tokio::io::{AsyncReadExt, AsyncWriteExt}


[tokio::main]#

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

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

;?stream.write_all(b"Hello from client!").await


;[1024 ;0] = let mut buffer

;?let n = stream.read(&mut buffer).await

;println!("Received: {}", String::from_utf8_lossy(&buffer[..n]))


(())Ok

{
```

توضیحات

سرور:

از TcpListener برای گوش دادن به اتصالات ورودی استفاده می‌کند.

با استفاده از accept، اتصالات جدید را می‌پذیرد و آن‌ها را به صورت غیرهمزمان مدیریت می‌کند.

پس از دریافت داده‌ها، پاسخ می‌دهد.

کلاینت:

با استفاده از TcpStream به سرور متصل می‌شود.

داده‌ها را به سرور ارسال می‌کند و سپس پاسخ را دریافت می‌کند.

اجرای برنامه

برای اجرای این برنامه، ابتدا سرور را اجرا کنید و سپس کلاینت را در یک ترمینال دیگر اجرا کنید. شما باید پیام‌های ارسال شده و دریافت شده را در ترمینال‌ها مشاهده کنید.

10) برنامه ای به زبان Rust بنویسید که عملیات CRUD را بر روی یک پایگاه انجام دهد؟برای ایجاد یک برنامه ساده به زبان Rust که عملیات CRUD (ایجاد، خواندن، به‌روزرسانی و حذف) را بر روی یک پایگاه داده انجام دهد، می‌توانیم از کتابخانه‌هایی مانند diesel برای کار با پایگاه داده و sqlite به عنوان پایگاه داده استفاده کنیم. در اینجا یک نمونه ساده از چنین برنامه‌ای را ارائه می‌دهم.

مراحل ایجاد برنامه

ایجاد پروژه جدید:

ابتدا با استفاده از Cargo، یک پروژه جدید ایجاد کنید:

bash

cargo new rust_crud

cd rust_crud

اضافه کردن وابستگی‌ها:

در فایل Cargo.toml، وابستگی‌های لازم را اضافه کنید:

toml

[dependencies]

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

"dotenv = "0.15

ایجاد پایگاه داده:

یک فایل SQLite برای پایگاه داده ایجاد کنید. می‌توانید این کار را با استفاده از sqlite3 انجام دهید:

bash

sqlite3 my_database.db

سپس جدول مورد نظر را ایجاد کنید:

sql

) CREATE TABLE users

,id INTEGER PRIMARY KEY AUTOINCREMENT

,name TEXT NOT NULL

```
age INTEGER NOT NULL
);
```

ایجاد فایل .env:

یک فایل .env در ریشه پروژه ایجاد کنید و آدرس پایگاه داده را در آن قرار دهید:

```
ini
DATABASE_URL=my_database.db
```

نوشتن کد Rust:

حالا می‌توانید کد CRUD را در فایل src/main.rs بنویسید:

```
rust
#[macro_use]
extern crate diesel;
extern crate dotenv;

use diesel::prelude::*;
use dotenv::dotenv;
use std::env;

pub mod schema {
table! {
users (id) {
id -> Integer,
name -> Text,
age -> Integer,
}
}
}

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

```
["table_name = "users]#
} struct User
,id: i32
,name: String
,age: i32
{

} 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, age: i32)
;let new_user = User { id: 0, name: name.to_string(), age }
diesel::insert_into(schema::users::table)
values(&new_user).
execute(conn).
;expect("Error saving new user").
{

} fn read_users(conn: &SqliteConnection)
;*::use schema::users::dsl
;let results = users.load::<User>(conn).expect("Error loading users")

} for user in results
;println!("{:?}", user)
{
{
```

```
} fn update_user(conn: &SqliteConnection, user_id: i32, new_name: &str, new_age: i32)
;use schema::users::dsl::{users, name, age}

diesel::update(users.find(user_id))
set((name.eq(new_name), age.eq(new_age))).
execute(conn).
;expect("Error updating user").
{

} fn delete_user(conn: &SqliteConnection, user_id: i32)
;use schema::users::dsl::users

diesel::delete(users.find(user_id))
execute(conn).
;expect("Error deleting user").
{

} ()fn main
;()let connection = establish_connection

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

// خواندن کاربران
;println!("Current users:")
;read_users(&connection)
```

```
// به‌روزرسانی یک کاربر
update_user(&connection, 1, "Alice Smith", 31);

// حذف یک کاربر
delete_user(&connection, 2);

// خواندن کاربران پس از به‌روزرسانی و حذف
println!("Users after update and delete:");
read_users(&connection);
}
```

توضیحات کد

تعریف جدول: با استفاده از ماکرو table!، جدول users تعریف شده است.

مدل User: ساختار User برای نگهداری اطلاعات کاربران تعریف شده است.

عملیات CRUD: توابع create_user, read_users, update_user, و delete_user برای انجام عملیات CRUD نوشته شده‌اند.

تابع main: در این تابع، ارتباط با پایگاه داده برقرار می‌شود و عملیات CRUD انجام می‌شود.

اجرای برنامه

برای اجرای برنامه، از دستور زیر استفاده کنید:

bash

```
cargo run
```

این برنامه باید عملیات CRUD را بر روی پایگاه داده SQLite انجام دهد و نتایج را در کنسول نمایش دهد.

11) با استفاده از یک ORM در زبان Rust برنامه‌ای بنویسید که عملیات CRUD را بر روی یک پایگاه داده انجام دهد؟ برای ایجاد یک برنامه CRUD (ایجاد، خواندن، به‌روزرسانی و حذف) در زبان Rust با استفاده از یک ORM (Object-Relational Mapping)، می‌توانیم از کتابخانه diesel استفاده کنیم. diesel یکی از محبوب‌ترین ORMها در Rust است و به ما امکان می‌دهد به راحتی با پایگاه‌های داده کار کنیم.

مراحل ایجاد برنامه CRUD با استفاده از Diesel

1. نصب وابستگی‌ها

ابتدا باید diesel و dotenv را به پروژه خود اضافه کنیم. فایل Cargo.toml شما باید به شکل زیر باشد:

toml

[package]

"name = "crud_example

"version = "0.1.0

"edition = "2021

[dependencies]

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

"dotenv = "0.15

bash

diesel setup

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

;(

و فایل down.sql را به شکل زیر ویرایش کنید:

sql

;DROP TABLE users

bash

diesel migration run

4. ایجاد مدل

در فایل src/models.rs، مدل User را تعریف کنید:

rust

```
;use diesel::{Queryable, Insertable}
;use serde::{Serialize, Deserialize}

[derive(Queryable, Serialize, Deserialize)]#
} pub struct User
,pub id: i32
,pub name: String
,pub email: String
{

[derive(Insertable, Serialize, Deserialize)]#
["table_name = "users]#
} <pub struct NewUser<'a
,pub name: &'a str
,pub email: &'a str
{
```

در فایل src/main.rs، عملیات CRUD را پیاده‌سازی کنید:

rust

```
[macro_use]#
;extern crate diesel
;extern crate dotenv

;*::use diesel::prelude
;use dotenv::dotenv
;use std::env
```

```
;mod models
;use models::{User, NewUser}

} 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 = NewUser { name, email }

diesel::insert_into(users::table)
values(&new_user).
execute(conn).
;expect("Error inserting new user").

()users::table.order(users::id.desc()).first(conn).unwrap
{

} <fn get_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
```

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

۱. nom

۲. pest

۳. serde

۴. combine

```
rust
use nom::{
    IResult,
    character::complete::digit1,
};

fn parse_number(input: &str) -> IResult<&str, i32> {
    let (remaining_input, digits) = digit1(input)?;
```

```
    let number: i32 = digits.parse().unwrap();
    Ok((remaining_input, number))
}

fn main() {
    let input = "1234abc";
    match parse_number(input) {
        Ok((remaining, number)) => {
            println!("Parsed number: {}, Remaining input: {}", number, remaining);
        },
        Err(err) => {
            println!("Error parsing input: {:?}", err);
        }
    }
}
```

توضیحات مثال:

برای پارس کردن استفاده می‌کنیم nom ما از توابع و ماژول‌های کتابخانه :nom کتابخانه.

این تابع یک رشته ورودی می‌گیرد و سعی می‌کند یک عدد صحیح را از آن استخراج کند :parse_number تابع.

برای پیدا کردن یک یا چند رقم استفاده می‌کنیم digit1 از.

تبدیل می‌کنیم i32 پس از پیدا کردن ارقام، آن‌ها را به نوع.

خروجی: اگر پارس موفقیت‌آمیز باشد، عدد و ورودی باقی‌مانده را چاپ می‌کنیم. در غیر این صورت، خطا را نمایش می‌دهیم.

این مثال نشان می‌دهد که چگونه می‌توان با استفاده از یک پارسر ساده در Rust، داده‌ها را تجزیه و تحلیل کرد.

13) مفهوم native-windows-gui و windwos در زبان Rust چیست؟ با ذکر یک مثال توضیح دهید؟ مفهوم Native Windows GUIدر زبان برنامه‌نویسی Rust به ایجاد رابط کاربری گرافیکی (GUI) برای سیستم‌عامل ویندوز اشاره دارد که به صورت بومی و با استفاده از API های اصلی ویندوز پیاده‌سازی می‌شود. این به این معناست که برنامه‌های نوشته شده با Rust می‌توانند از قابلیت‌ها و ویژگی‌های خاص ویندوز بهره‌برداری کنند و به طور طبیعی با سیستم‌عامل تعامل داشته باشند.

مفاهیم کلیدی:

Native: های ویندوز کار می‌کند و از کتابخانه‌های API به معنای بومی است و به این اشاره دارد که برنامه به طور مستقیم با شخص ثالث استفاده نمی‌کند.

Windows GUI: به رابط کاربری گرافیکی ویندوز اشاره دارد که شامل پنجره‌ها، دکمه‌ها، منوها و سایر عناصر گرافیکی است.

مثال ساده:

برای ایجاد یک برنامه ساده GUI در Rust مانند winapi یا windows-rs استفاده کرد. در اینجا یک مثال ساده با استفاده از windows-rs آورده شده است:

```
rust
use windows::{
    core::*,
    Win32::Foundation::*,
    Win32::Graphics::Gdi::*,
    Win32::UI::WindowsAndMessaging::*,
};

fn main() {
    // تعریف یک کلاس پنجره
    const CLASS_NAME: &str = "MyWindowClass";

    // ثبت کلاس پنجره
    let h_instance = unsafe { GetModuleHandleW(None).unwrap() };
    let wc = WNDCLASSW {
        hInstance: h_instance,
        lpszClassName: PCWSTR::from_raw(CLASS_NAME.encode_utf16().collect::<Vec<u16>>().as_ptr()),
        lpfnWndProc: Some(window_proc),
        ..Default::default()
    };

    unsafe { RegisterClassW(&wc) };

    // ایجاد پنجره
```

```
    let hwnd = unsafe {
        CreateWindowExW(
            Default::default(),
            PCWSTR::from_raw(CLASS_NAME.encode_utf16().collect::<Vec<u16>>().as_ptr()),
            PCWSTR::from_raw("Hello, Windows!".encode_utf16().collect::<Vec<u16>>().as_ptr()),
            WS_OVERLAPPEDWINDOW,
            CW_USEDEFAULT,
            CW_USEDEFAULT,
            300,
            200,
            None,
            None,
            h_instance,
            std::ptr::null_mut(),
        )
    };

    // نمایش پنجره
    unsafe { ShowWindow(hwnd, SW_SHOW) };

    // حلقه پیام
    let mut msg = MSG::default();
    while unsafe { GetMessageW(&mut msg, None, 0, 0) } != 0 {
        unsafe { TranslateMessage(&msg) };
        unsafe { DispatchMessageW(&msg) };
    }
}

// تابع پردازش پیام
```

```
extern "system" fn window_proc(hwnd: HWND, msg: u32, wparam: WPARAM, lparam: LPARAM) ->
LRESULT {
    match msg {
        WM_DESTROY => {
            unsafe { PostQuitMessage(0) };
            LRESULT(0)
        }
        _ => unsafe { DefWindowProcW(hwnd, msg, wparam, lparam) },
    }
}
```

توضیحات کد:

یک کلاس پنجره جدید ثبت می‌شود RegisterClassW ثبت کلاس پنجره: با استفاده از.

یک پنجره جدید ایجاد می‌شود CreateWindowExW ایجاد پنجره: با.

، برنامه به دریافت و پردازش پیام‌ها ادامه می‌دهدGetMessageW حلقه پیام: با استفاده از.

برای پردازش پیام‌های دریافتی از سیستم‌عامل تعریف شده است window_proc تابع پردازش پیام: تابع.

14) مفهوم Regular Expression چیست؟ در زبان Rust با بیا یک مثال توضیح دهید؟ مفهوم Regular Expression (عبارات منظم) یک ابزار قدرتمند برای جستجو و پردازش رشته‌ها در برنامه‌نویسی است. این عبارات به ما این امکان را می‌دهند که الگوهای خاصی را در یک رشته شناسایی کنیم، مانند آدرس‌های ایمیل، شماره تلفن‌ها، یا هر نوع داده‌ای که دارای الگوی مشخصی باشد.

در زبان Rust، برای کار با عبارات منظم می‌توان از کتابخانه regex استفاده کرد. این کتابخانه امکانات زیادی برای جستجو و تطبیق الگوها در رشته‌ها فراهم می‌کند.

مثال:

فرض کنید می‌خواهیم بررسی کنیم که آیا یک رشته، یک آدرس ایمیل معتبر است یا خیر. الگوی ساده‌ای که می‌توانیم برای آدرس‌های ایمیل در نظر بگیریم به صورت زیر است:

باید شامل یک یا چند کاراکتر قبل از علامت @ باشد.

سپس باید یک دامنه (مثلاً example.com) داشته باشد.

الگوی عبارات منظم برای این کار به صورت زیر خواهد بود:

inform7

```
${,2}[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]^
```

کد Rust:

در زیر یک مثال ساده از نحوه استفاده از عبارات منظم در Rust برای بررسی آدرس‌های ایمیل آورده شده است:

```
rust

;use regex::Regex


} ()fn main

;"${,2}let email_pattern = r"^[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]

;()let re = Regex::new(email_pattern).unwrap


;"let email = "example@example.com


} if re.is_match(email)

;("آدرس ایمیل معتبر است.")!println

} else {

;("آدرس ایمیل نامعتبر است.")!println

{

{
```

توضیحات کد:

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

تعریف الگو: الگوی عبارات منظم را در متغیر email_pattern تعریف می‌کنیم.

ایجاد شی Regex: با استفاده از Regex::new یک شی از نوع Regex ایجاد می‌کنیم.

بررسی تطابق: با استفاده از متد is_match بررسی می‌کنیم که آیا رشته مورد نظر با الگوی تعریف شده مطابقت دارد یا خیر.

خروجی: بر اساس نتیجه بررسی، پیام مناسب چاپ می‌شود.

این مثال نشان می‌دهد که چگونه می‌توان از عبارات منظم در Rust برای پردازش و اعتبارسنجی داده‌ها استفاده کرد.

15) عملکرد کتابخانه SysInfo در زبان Rust چیست با ذکر مثال ساده توضیح دهید؟ کتابخانه sysinfo در زبان برنامه‌نویسی Rust برای جمع‌آوری اطلاعات سیستم مانند وضعیت پردازنده، حافظه، دیسک‌ها و پروسه‌ها طراحی شده است. این کتابخانه به شما امکان می‌دهد تا اطلاعات مفیدی درباره سیستم‌عامل و منابع سخت‌افزاری را به راحتی به دست آورید.

عملکرد کتابخانه sysinfo

کتابخانه sysinfo می‌تواند اطلاعات زیر را ارائه دهد:

اطلاعات پردازنده: تعداد هسته‌ها، بار پردازنده و ...

اطلاعات حافظه: مقدار حافظه‌ی آزاد و استفاده شده.

اطلاعات دیسک: فضای آزاد و استفاده شده در دیسک‌ها.

اطلاعات پروسه‌ها: لیست پروسه‌های در حال اجرا و اطلاعات مربوط به آن‌ها.

نصب کتابخانه

برای استفاده از sysinfo، ابتدا باید آن را به پروژه‌ی خود اضافه کنید. در فایل Cargo.toml، خط زیر را اضافه کنید:

toml

```
[dependencies]
"sysinfo = "0.24"  # اطمینان حاصل کنید که نسخه‌ی مناسب را انتخاب کرده‌اید
```

مثال ساده

در این مثال، ما اطلاعاتی درباره‌ی حافظه و پروسه‌های در حال اجرا را چاپ خواهیم کرد.

rust

```
use sysinfo::{System, SystemExt, ProcessExt};

fn main() {

    // ایجاد یک شیء از نوع System
    let mut system = System::new_all();

    // بارگذاری اطلاعات سیستم
    system.refresh_all();

    // چاپ اطلاعات حافظه
    println!("Total memory: {} KB", system.total_memory());
    println!("Used memory: {} KB", system.used_memory());
    println!("Free memory: {} KB", system.free_memory());

    // چاپ اطلاعات پروسه‌ها
```

```
;println!("\nProcesses:")
} ()for (pid, process) in system.processes
;println!("PID: {}, Name: {}", pid, process.name())
{
{
```

توضیحات کد

وارد کردن کتابخانه: با استفاده از {use sysinfo::{System, SystemExt, ProcessExt; ما ماژول‌های مورد نیاز را وارد می‌کنیم.

ایجاد شیء System: با let mut system = System::new_all()؛ یک شیء جدید از نوع System ایجاد می‌کنیم که می‌تواند اطلاعات سیستم را جمع‌آوری کند.

بارگذاری اطلاعات: با system.refresh_all()؛ اطلاعات سیستم را به‌روز می‌کنیم.

چاپ اطلاعات حافظه: با استفاده از متدهای total_memory(), used_memory(), و free_memory() اطلاعات مربوط به حافظه را چاپ می‌کنیم.

چاپ پروسه‌ها: با استفاده از حلقه for، به لیست پروسه‌ها دسترسی پیدا کرده و نام و شناسه (PID) هر پروسه را چاپ می‌کنیم.

16) برنامه ای برای انجام یک پردازش ساده بر روی یکImage بنویسید؟ برای انجام پردازش ساده بر روی تصاویر، می‌توان از کتابخانه‌های مختلفی در زبان‌های برنامه‌نویسی استفاده کرد. یکی از محبوب‌ترین کتابخانه‌ها برای پردازش تصویر در زبان Python، کتابخانه OpenCV است. در اینجا یک برنامه ساده با استفاده از OpenCV برای بارگذاری یک تصویر، تبدیل آن به مقیاس خاکستری و نمایش آن ارائه می‌شود.

مراحل انجام کار

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

bash

```
pip install opencv-python
```

نوشتن برنامه: در ادامه، یک برنامه ساده برای بارگذاری یک تصویر، تبدیل آن به مقیاس خاکستری و نمایش آن نوشته شده است.

python

```
import cv2
# بارگذاری تصویر
'image_path = 'path/to/your/image.jpg  # مسیر تصویر را اینجا وارد کنید
image = cv2.imread(image_path)
# بررسی اینکه آیا تصویر بارگذاری شده است یا نه
:if image is None
```

```
print("خطا: تصویر بارگذاری نشد.")
else:
    # تبدیل تصویر به مقیاس خاکستری
    gray_image = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)

    # نمایش تصویر اصلی
    cv2.imshow('Original Image', image)

    # نمایش تصویر خاکستری
    cv2.imshow('Gray Image', gray_image)

    # منتظر ماندن برای فشردن کلید
    cv2.waitKey(0)

    # بستن تمام پنجره‌ها
    cv2.destroyAllWindows()
```

توضیحات برنامه

بارگذاری تصویر: با استفاده از cv2.imread() تصویر بارگذاری می‌شود. مسیر تصویر باید به درستی مشخص شود.

بررسی بارگذاری تصویر: قبل از پردازش، برنامه بررسی می‌کند که آیا تصویر به درستی بارگذاری شده است یا خیر.

تبدیل به مقیاس خاکستری: با استفاده از cv2.cvtColor() تصویر به مقیاس خاکستری تبدیل می‌شود.

نمایش تصاویر: با استفاده از cv2.imshow()، تصویر اصلی و تصویر خاکستری نمایش داده می‌شوند.

مدت زمان نمایش: cv2.waitKey(0) باعث می‌شود برنامه منتظر فشردن یک کلید باشد و سپس با cv2.destroyAllWindows() تمام پنجره‌ها بسته می‌شوند.

17) برنامه ای برای انجام یک پردازش ساده بر روی یک Video بنویسید؟ برای انجام پردازش ساده بر روی یک ویدیو، می‌توانیم از زبان برنامه‌نویسی Python و کتابخانه OpenCV استفاده کنیم. OpenCV یک کتابخانه قدرتمند برای پردازش تصویر و ویدیو است. در اینجا یک برنامه ساده برای خواندن ویدیو، تبدیل آن به خاکستری و ذخیره کردن ویدیوی پردازش‌شده ارائه می‌شود.

مراحل انجام کار:

نصب کتابخانه‌های لازم: ابتدا باید کتابخانه OpenCV را نصب کنید. می‌توانید از pip استفاده کنید:

bash

pip install opencv-python

نوشتن کد: در ادامه، کد زیر را می‌توانید استفاده کنید:

python

import cv2

```
# نام فایل ویدیویی ورودی و خروجی
'input_video_path = 'input_video.mp4
'output_video_path = 'output_video.mp4

# ویدیو را باز کنید
cap = cv2.VideoCapture(input_video_path)

# بررسی کنید که ویدیو باز شده است یا نه
:()if not cap.isOpened
("خطا در باز کردن ویدیو")print
()exit

# دریافت اطلاعات ویدیو
# فریم در ثانیه  fps = cap.get(cv2.CAP_PROP_FPS)
# عرض فریم  width = int(cap.get(cv2.CAP_PROP_FRAME_WIDTH))
# ارتفاع فریم  height = int(cap.get(cv2.CAP_PROP_FRAME_HEIGHT))

# ایجاد یک ویدیو نویس برای ذخیره ویدیو پردازش‌شده
# کدک ویدیو  fourcc = cv2.VideoWriter_fourcc(*'mp4v')
out = cv2.VideoWriter(output_video_path, fourcc, fps, (width, height), False)

:while True
# خواندن فریم  ()ret, frame = cap.read
:if not ret
```

```
break  # اگر فریم وجود نداشت، حلقه را بشکنید

# تبدیل فریم به خاکستری
gray_frame = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)

# ذخیره فریم پردازش‌شده
out.write(gray_frame)

# نمایش فریم (اختیاری)
cv2.imshow('Video', gray_frame)

# برای خروج از نمایش ویدیو، کلید 'q' را فشار دهید
:if cv2.waitKey(1) & 0xFF == ord('q')
break

# آزاد کردن منابع
()cap.release
()out.release
()cv2.destroyAllWindows
```

توضیحات کد:

وارد کردن کتابخانه: ابتدا کتابخانه OpenCV را وارد می‌کنیم.

باز کردن ویدیو: با استفاده از cv2.VideoCapture ویدیوی ورودی را باز می‌کنیم.

بررسی باز شدن ویدیو: اگر ویدیو باز نشود، برنامه خطا می‌دهد و خارج می‌شود.

دریافت اطلاعات ویدیو: اطلاعاتی مانند فریم در ثانیه، عرض و ارتفاع فریم را دریافت می‌کنیم.

ایجاد ویدیو نویس: با استفاده از cv2.VideoWriter، یک ویدیو نویس برای ذخیره ویدیو پردازش‌شده ایجاد می‌کنیم.

پردازش فریم‌ها: در یک حلقه، هر فریم را خوانده و به خاکستری تبدیل می‌کنیم و سپس آن را ذخیره می‌کنیم.

نمایش فریم: فریم پردازش‌شده را در یک پنجره نمایش می‌دهیم.

آزاد کردن منابع: در پایان، منابع را آزاد می‌کنیم.

18) برنامه ای برای دانلود یک فایل از Internet بنویسید؟ رای دانلود یک فایل از اینترنت، می‌توانید از زبان‌های برنامه‌نویسی مختلفی استفاده کنید. در اینجا، یک مثال ساده با استفاده از زبان Python و کتابخانه requests آورده شده است. این برنامه به شما اجازه می‌دهد تا یک فایل را از یک URL مشخص دانلود کنید.

```
bash

pip install requests

python

import requests


:def download_file(url, filename)

:try

URL به GET ارسال درخواست #

response = requests.get(url, stream=True)

بررسی وضعیت پاسخ #  ()response.raise_for_status


نوشتن محتویات فایل در فایل محلی #

:with open(filename, 'wb') as file

:for chunk in response.iter_content(chunk_size=8192)

file.write(chunk)


("{filename} :فایل با موفقیت دانلود شد"f)print


:except requests.exceptions.RequestException as e

("{e} :خطا در دانلود فایل"f)print


:"if name == "main

فایل مورد نظر URL #

(" :فایل را وارد کنید URL لطفاً"(")url = input

نام فایل محلی برای ذخیره #

(" :نام فایل برای ذخیره را وارد کنید لطفاً"(")filename = input
```

download_file(url, filename)

توضیحات کد:

کتابخانه requests: این کتابخانه برای ارسال درخواست‌های HTTP استفاده می‌شود.

تابع download_file: این تابع دو ورودی می‌گیرد: url (آدرس فایل) و filename (نامی که فایل دانلود شده با آن ذخیره می‌شود).

درخواست GET: با استفاده از requests.get، یک درخواست به URL ارسال می‌شود و محتویات آن به صورت استریم دریافت می‌شود.

نوشتن فایل: محتویات فایل در یک فایل محلی با نام مشخص شده ذخیره می‌شود.

مدیریت خطا: اگر در حین دانلود خطایی پیش بیاید، پیام مناسبی نمایش داده می‌شود.

19) برنامه ای برای انجام یک پردازش ساده بر روی یکAudio بنویسید؟ برای انجام یک پردازش ساده بر روی یک فایل صوتی، می‌توانیم از کتابخانه‌های معروف پایتون مانند pydub و numpy استفاده کنیم. در اینجا یک برنامه ساده برای بارگذاری یک فایل صوتی، کاهش حجم آن و ذخیره‌ی فایل جدید ارائه می‌شود.

bash

pip install pydub

pip install numpy

کد برنامه

python

from pydub import AudioSegment

:def process_audio(input_file, output_file, reduce_db)

بارگذاری فایل صوتی #

audio = AudioSegment.from_file(input_file)

کاهش حجم صدا #

processed_audio = audio - reduce_db

ذخیره فایل جدید #

processed_audio.export(output_file, format="mp3")

:"if name == "main

نام فایل ورودی # "input_file = "input.mp3

نام فایل خروجی # "output_file = "output.mp3

reduce_db = 10 # مقدار کاهش دسیبل

process_audio(input_file, output_file, reduce_db)

print(f"فایل صوتی پردازش شده و به {output_file} ذخیره شد.")

توضیحات کد

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

کاهش حجم صدا: با کم کردن مقدار دسیبل (در اینجا 10 دسیبل) حجم صدا کاهش می‌یابد.

ذخیره فایل جدید: با استفاده از export() فایل پردازش شده ذخیره می‌شود.

20) برنامه ای برای خواندن ونوشتن یک فایل CSV بنویسید ؟ برای خواندن و نوشتن یک فایل CSV در زبان برنامه‌نویسی پایتون، می‌توانیم از کتابخانه‌ی داخلی csv استفاده کنیم. در ادامه یک برنامه ساده برای خواندن و نوشتن فایل CSV ارائه می‌دهم.

1. نصب پایتون

ابتدا مطمئن شوید که پایتون روی سیستم شما نصب شده است. می‌توانید با اجرای دستور زیر در ترمینال یا CMD بررسی کنید:

bash

python --version

2. نوشتن برنامه

الف) نوشتن داده‌ها به فایل CSV

python

import csv

داده‌هایی که می‌خواهیم در فایل CSV بنویسیم

data = [

['نام', 'سن', 'شغل'],

['علی', 30, 'برنامه‌نویس'],

['سارا', 25, 'طراح'],

['مهدی', 35, 'مدیر']

```
# نوشتن داده‌ها به فایل CSV
:with open('data.csv', mode='w', newline='', encoding='utf-8') as file
writer = csv.writer(file)
writer.writerows(data)

print("داده‌ها با موفقیت به فایل CSV نوشته شدند.")
```

ب) خواندن داده‌ها از فایل CSV

```
python
import csv

# خواندن داده‌ها از فایل CSV
:with open('data.csv', mode='r', encoding='utf-8') as file
reader = csv.reader(file)
:for row in reader
print(row)
```

3. توضیحات برنامه

نوشتن به فایل CSV:

با استفاده از open فایل data.csv را در حالت نوشتن ('w') باز می‌کنیم.

از csv.writer برای ایجاد یک نویسنده CSV استفاده می‌کنیم.

با استفاده از writerows داده‌ها را به فایل می‌نویسیم.

خواندن از فایل CSV:

دوباره با استفاده از open فایل را در حالت خواندن ('r') باز می‌کنیم.

از csv.reader برای ایجاد یک خواننده CSV استفاده می‌کنیم.

با استفاده از یک حلقه for هر ردیف را خوانده و چاپ می‌کنیم.

22) برنامه ای برای خواندن و نوشتن یک فایل Excelساده بنویسید؟ برای خواندن و نوشتن یک فایل Excel ساده در زبان برنامه‌نویسی Python، می‌توانید از کتابخانه pandas و openpyxl استفاده کنید. این کتابخانه‌ها ابزارهای قدرتمندی برای کار با داده‌ها و فایل‌های Excel هستند.

bash

pip install pandas openpyxl

خواندن یک فایل Excel

برای خواندن یک فایل Excel، می‌توانید از کد زیر استفاده کنید:

python

import pandas as pd

خواندن فایل Excel

file_path = 'example.xlsx' # مسیر فایل Excel

df = pd.read_excel(file_path)

نمایش داده‌ها

print(df)

نوشتن به یک فایل Excel

برای نوشتن داده‌ها به یک فایل Excel، می‌توانید از کد زیر استفاده کنید:

python

import pandas as pd

ایجاد یک DataFrame جدید

data = {

'نام': ['علی', 'زهرا', 'مهدی'],

'سن': [25, 30, 22],

'شغل': ['برنامه‌نویس', 'طراح', 'مدیر']

}

df = pd.DataFrame(data)

نوشتن DataFrame به یک فایل Excel

'output_file_path = 'output.xlsx # مسیر فایل خروجی

df.to_excel(output_file_path, index=False)

print(f'داده‌ها با موفقیت در {output_file_path} ذخیره شدند.')

توضیحات کد

خواندن فایل: با استفاده از pd.read_excel()، فایل Excel خوانده می‌شود و داده‌ها به یک DataFrame تبدیل می‌شوند.

نوشتن فایل: با استفاده از df.to_excel()، داده‌ها به یک فایل Excel جدید نوشته می‌شوند. پارامتر index=False برای جلوگیری از نوشتن ایندکس‌ها در فایل خروجی استفاده می‌شود.

نکات

مطمئن شوید که مسیر فایل‌ها را به درستی تنظیم کرده‌اید.

فرمت فایل Excel باید .xlsx باشد.

23) مدل MVC را در قالب یک برنامه بیاده سازی کنید؟ مدل MVC (Model-View-Controller) یک الگوی طراحی نرم‌افزاری است که به جداسازی نگرانی‌ها در برنامه‌های کاربردی کمک می‌کند. این الگو به سه بخش اصلی تقسیم می‌شود:

مدل (Model): این بخش مسئول مدیریت داده‌ها و منطق تجاری برنامه است. مدل شامل کلاس‌ها و توابعی است که داده‌ها را مدیریت می‌کنند و به‌روزرسانی‌های لازم را انجام می‌دهند.

نما (View): این بخش مسئول نمایش داده‌ها به کاربر است. نما شامل رابط کاربری و نحوه نمایش اطلاعات است.

کنترلر (Controller): این بخش میان مدل و نما ارتباط برقرار می‌کند. کنترلر ورودی‌های کاربر را دریافت کرده و بر اساس آن‌ها مدل و نما را به‌روزرسانی می‌کند.

پیاده‌سازی MVC در یک برنامه ساده با استفاده از Python و Flask

در این مثال، یک برنامه وب ساده با استفاده از فریم‌ورک Flask پیاده‌سازی می‌کنیم که یک لیست از کارها (To-Do List) را مدیریت می‌کند.

1. نصب Flask

ابتدا، مطمئن شوید که Flask نصب شده است. می‌توانید با استفاده از pip آن را نصب کنید:

bashpip install Flask

2. ساختار پروژه

ساختار پروژه به صورت زیر خواهد بود:

vimmvc_example/ | ├── app.py ├── model.py ├── view.py └── controller.py

3. پیاده‌سازی مدل (model.py)

python# model.py class Task: def init(self, title): self.title = title self.completed = False class TaskModel: def init(self): self.tasks = [] def add_task(self, title): task = Task(title) self.tasks.append(task) def get_tasks(self): return self.tasks def complete_task(self, index): if 0 <= index < len(self.tasks): self.tasks[index].completed = True

4. پیاده‌سازی نما (view.py)

python# view.py from flask import render_template def render_task_list(tasks): return render_template('task_list.html', tasks=tasks)

5. پیاده‌سازی کنترلر (controller.py)

python# controller.py from flask import Flask, request, redirect, url_for from model import TaskModel from view import render_task_list app = Flask(name) task_model = TaskModel() @app.route('/') def index(): tasks = task_model.get_tasks() return render_task_list(tasks) @app.route('/add', methods=['POST']) def add_task(): title = request.form.get('title') task_model.add_task(title) return redirect(url_for('index')) @app.route('/complete/<int:index>') def complete_task(index): task_model.complete_task(index) return redirect(url_for('index'))

6. پیاده‌سازی فایل اصلی (app.py)

python# app.py from controller import app if name == 'main': app.run(debug=True)

7. ایجاد فایل HTML (task_list.html)

در پوشه‌ای به نام templates، فایل task_list.html را ایجاد کنید:

title> /><html<!DOCTYPE html> <html lang="fa"> <head> <meta charset="UTF-8"> <title لیست کارها
h1> <body> <></head>لیست باشد.

عنوان"=h1> <form action="/add" method="post"> <input type="text" name="title" placeholder/>کارها
button> </form> {% for task in tasks %} {{ /><اضافه کردن"required> <button type="submit "کار
{% endif %} {% if not task.completed در حال انجام {% else %} انجام شده task.title }} - {% if task.completed %}
a> {% endif %} {% endfor %} </body> />تکمیل<"%} <a href="/complete/{{ loop.index0 }}
<</html

24) معماری Clean Architecture را در قالب یک برنامه بیاده سازی کنید ؟ معماری Clean Architecture یکی از الگوهای طراحی نرم‌افزار است که به تفکیک مسئولیت‌ها و کاهش وابستگی‌ها در برنامه‌ها کمک می‌کند. این معماری معمولاً به چهار لایه اصلی تقسیم می‌شود:

لایه Entities: شامل مدل‌های کسب‌وکار و قوانین تجاری است.

لایه Use Cases: شامل منطق کسب‌وکار و موارد استفاده است.

لایه Interface Adapters: شامل مبدل‌ها و کنترلرها برای تبدیل داده‌ها به فرمت مناسب برای لایه‌های دیگر است.

لایه Frameworks & Drivers: شامل جزئیات پیاده‌سازی مانند پایگاه‌داده، UI و سایر فریم‌ورک‌ها است.

پیاده‌سازی یک برنامه ساده با Clean Architecture

برای مثال، بیایید یک برنامه ساده مدیریت وظایف (To-Do List) را پیاده‌سازی کنیم.

1. لایه Entities

python# entities/task.py class Task: def init(self, id: int, title: str, completed: bool = False): self.id = id self.title = title self.completed = completed def mark_completed(self): self.completed = True

2. لایه Use Cases

python# use_cases/task_use_case.py from entities.task import Task class TaskUseCase: def init(self, task_repository): self.task_repository = task_repository def create_task(self, title: str): task_id = len(self.task_repository.get_all()) + 1 # Simple ID generation task = Task(id=task_id, title=title) ()self.task_repository.save(task) def get_all_tasks(self): return self.task_repository.get_all

3. لایه Interface Adapters

python# adapters/task_repository.py from entities.task import Task class InMemoryTaskRepository: def init(self): self.tasks = [] def save(self, task: Task): self.tasks.append(task) def get_all(self): return self.tasks

4. لایه Frameworks & Drivers

python# main.py from use_cases.task_use_case import TaskUseCase from adapters.task_repository import InMemoryTaskRepository def main(): task_repository = InMemoryTaskRepository() task_use_case = TaskUseCase(task_repository) # Create tasks task_use_case.create_task("Buy groceries") task_use_case.create_task("Read a book") # Get all tasks tasks = task_use_case.get_all_tasks() for task in tasks: print(f"Task ID: {task.id}, Title: {task.title}, Completed: (){task.completed}") if name == "main": main

توضیحات

Entities: در اینجا ما یک کلاس Task داریم که نمایانگر یک وظیفه است و شامل ویژگی‌های آن و متدهایی برای تغییر وضعیت آن است.

Use Cases: TaskUseCase منطق کسب‌وکار را مدیریت می‌کند. این کلاس وظیفه ایجاد وظایف جدید و دریافت تمام وظایف را بر عهده دارد.

Interface Adapters: InMemoryTaskRepository یک پیاده‌سازی ساده از مخزن وظایف است که وظایف را در حافظه نگه می‌دارد.

Frameworks & Drivers: در main.py، ما از لایه‌های دیگر استفاده می‌کنیم تا برنامه را اجرا کنیم و وظایف را ایجاد و نمایش دهیم.

نتیجه‌گیری

این پیاده‌سازی ساده نشان‌دهنده اصول Clean Architecture است. با استفاده از این الگو، می‌توانیم به راحتی بخش‌های مختلف برنامه را تغییر دهیم یا گسترش دهیم بدون اینکه تأثیر زیادی بر روی سایر بخش‌ها داشته باشیم. به عنوان مثال، می‌توانیم InMemoryTaskRepository را با یک پایگاه‌داده واقعی جایگزین کنیم بدون اینکه نیازی به تغییر در لایه‌های دیگر داشته باشیم.

25) اصول SOLID را در زبان Rust بیاده سازی نمایید ؟ اصول SOLID مجموعه‌ای از پنج اصل طراحی شی‌گرا هستند که به بهبود طراحی نرم‌افزار و افزایش قابلیت نگهداری و توسعه آن کمک می‌کنند. این اصول به شرح زیر هستند:

Single Responsibility Principle (SRP): هر کلاس باید تنها یک مسئولیت داشته باشد. به عبارت دیگر، یک کلاس نباید بیش از یک دلیل برای تغییر داشته باشد.

Open/Closed Principle (OCP): کلاس‌ها باید برای گسترش باز و برای تغییر بسته باشند. این به این معناست که می‌توانیم رفتار یک کلاس را بدون تغییر کد آن کلاس گسترش دهیم.

Liskov Substitution Principle (LSP): اشیاء از یک زیرکلاس باید بتوانند جایگزین اشیاء از کلاس والد خود شوند بدون اینکه رفتار برنامه تغییر کند.

Interface Segregation Principle (ISP): بهتر است که چندین رابط خاص وجود داشته باشد تا یک رابط عمومی بزرگ. این به این معناست که یک کلاس نباید به متدهایی که استفاده نمی‌کند وابسته باشد.

Dependency Inversion Principle (DIP): وابستگی‌ها باید به انتزاع‌ها (interface یا abstract class) وابسته باشند، نه به کلاس‌های خاص.

پیاده‌سازی در Rust: با استفاده از traitها و پیاده‌سازی‌های مختلف می‌توانیم این اصل را رعایت کنیم.

```
rustfn print_area(shape: &dyn Shape) { println!("Area: {}", shape.area()); } let circle = Circle { radius: 5.0 ;}; let rectangle = Rectangle { width: 4.0, height: 3.0 }; print_area(&circle); print_area(&rectangle)
```

مشتری‌ها نباید به رابطهایی وابسته باشند که نیازی به آن‌ها ندارند.

پیاده‌سازی در Rust: با تعریف traitهای کوچک و خاص می‌توانیم این اصل را رعایت کنیم.

```
rusttrait Printer { fn print(&self); } trait Scanner { fn scan(&self); } struct MultiFunctionPrinter; impl Printer for MultiFunctionPrinter { fn print(&self) { println!("Printing..."); } } impl Scanner for MultiFunctionPrinter { fn scan(&self) { println!("Scanning..."); } }
```

ماژول‌های سطح بالا نباید به ماژول‌های سطح پایین وابسته باشند. هر دو باید به abstraction وابسته باشند.

پیاده‌سازی در Rust: می‌توانیم از traitها برای تعریف وابستگی‌ها استفاده کنیم.

```
rusttrait Database { fn connect(&self); } struct MySQLDatabase; impl Database for MySQLDatabase { fn connect(&self) { println!("Connected to MySQL Database"); } } struct App<T: Database> { db: T, } impl<T: ;()} } let app = App { db: MySQLDatabase }; app.run ;()Database> App<T> { fn run(&self) { self.db.connect
```