rust&&netcore

- 安装rust
- 基础语法
- 自定义包
- 创建NETCORE新项目
- 速查知识库
- 常用库索引
- 实现RESTful API服务

安装rust

安装

```
# windows:

https://static.rust-lang.org/rustup/dist/x86_64-pc-windows-msvc/rustup-init.exe

根据提示，安装msvc和windows sdk
```

```
# linux:

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

# 安装完成后需要刷新环境变量
source "$HOME/.cargo/env"
```

rustc使--version 验证安装成功，

```
# 查看版本
rustc --version

# 显示安装信息
rustup show

# 更新
rustup update

# 显示本地帮助手册
rustup doc


# 卸载
```

```
rustup self uninstall
```

开发工具与插件

vscode, rust-analyzer, rust-lang.org 这个官网请看清楚作者为

编写hello,world

```
# vi main.rs

fn main() {
    println!("Hello, world!");
}

# 使用rustc编译源文件
rustc main.rs
```

使用cargo构建和包管理器

```
# 显示管理版本
cargo --version

# 创建一个新项目
cargo new hello_cargo

# 编译, 编译debug版本
cargo build

# 编译, 编译release版本
cargo build --release

# 编译并运行
cargo run

# 编译检查
```

```
cargo check
```

基础语法

项目结构和项目文件定义

```
/
/Cargo.toml                   # 项目文件定义


/src                          # 源文件目录
├--/main.rs                   # main
└--


/target                       # 编译输出目录
├--/debug                     # debug目录
└--/release                   # release目录
```

```
[package]
name = "project_name"
version = "0.1.0"
edition = "2023"



[dependencies]
```

基础语法

```


# 定义不可变变量
let x = 5;


# 定义可变变量
let mut x = 5;
x=10;


# 变量覆盖
let x = 9;
```

```
fn function_name(){
}

// 带返回值函数
// 使用 ->说明返回类型
fn fun_name() -> u32 {
    // 最后一句指令为返回值
    // 55
    // 或者使用 return
    // return 55;
}
```

定义结构

```
struct Employ {
    name: String,
    age: u32,
    sex: String,
}

fn main() {
    let mut em: Employ = Employ {
        name: String::from("董列涛"),
        age: 30,
        sex: String::from("a"),
    };
    em.name = String::from("donglietao");
    println!("{}", em.name);
}
```

结构作为函数返回值

```
fn main() {
    let em = get_employ();
  // 赋值更新
    let em2 = Employ { ..em };
```

```
    println!("{}", em.name);
    println!("{}", em2.name);


  // 打印调试信息
   println!("{:#?}", em2);
}

fn get_employ() -> Employ {
    let mut em: Employ = Employ {
        name: String::from("董列涛"),
        age: 30,
        sex: String::from("a"),
    };
    em.name = String::from("donglietao");
    return em;
}

 // 支持打印调试信息
#[derive(Debug)]
struct Employ {
    name: String,
    age: u32,
    sex: String,
}
```

自定义对象

```
fn main() {
    let em = Employ::new(String::from("董列涛"), 33, String::from("1"));
    println!("{:#?}", em.to_string());
}

// 定义数据结构
#[derive(Debug)]
struct Employ {
    name: String,
    age: u32,
    sex: String,
}
```

```
// 使用impl关键字
impl Employ {
    // 并没有构造函数一说
    fn new(name: String, age: u32, sex: String) -> Employ {
        return Employ {
            name: name,
            age: age,
            sex: sex,
        };
    }

    // 定义成员方法
    // 第一个参数self为实例本身
    fn to_string(self) -> String {
        return self.name;
    }
}
```

定义枚举

```
enum Sex {
    Main = 0,
    Woman = 1,
}
```

自定义包

概述

在rust中, 自定义包有4个概念

- package:包, 既一个项目成为一个包
- crate 包的树形结构
- modules和use 模块和作用域私有性
- path 模块的路径

package 和 crate

cargo new project crate **src/main.rs** (二进制包: binary) **src/lib.rs**(库包: library) **package**

申明module

```
/Cargo.toml
/src/
/src/main.rs
/src/ffmpeg_encoder/mod.rs
/src/ffmpeg_encoder/book.rs
/src/ffmpeg_encoder/people.rs
```

crate **src/main.rs** **src/lib.rs**

```
# src/main.rs

// 引入模块
use crate::ffmpeg_encoder::book::Book;
use crate::ffmpeg_encoder::people::Employ;
use crate::ffmpeg_encoder::people::Sex;
use crate::ffmpeg_encoder::Shop;

// 定义oo module
pub mod ffmpeg_encoder;
```

```
fn main() {


    println!("{}", Shop::to_string());


    println!("{}", Book::new().to_string());


    let em = Employ::new(String::from("董列涛"), 33, Sex::Woman);
    println!("{}", em.to_string());
}
```

```
# 申明子模块，rust根据约定采用固定的文件命令方式，建议采用第一种，可读性更强
# src/ffmpeg_encoder/mod.rs
# src/ffmpeg_encoder.rs



pub mod book;
pub mod people;

pub struct Shop {}

impl Shop {
    pub fn to_string() -> String {
        return String::from("Shop to_string");
    }
}
```

```
# src/ffmpeg_encoder/book.rs

pub struct Book;

impl Book {
    pub(crate) fn new() -> Book {
        Book {}
    }

    pub(crate) fn to_string(&self) -> String {
        return String::from("Book to_string");
```

```
    }
}
```

```
# 编写代码文件
# src/ffmpeg_encoder/people.rs


#[derive(Debug)]
pub struct Employ {
    name: String,
    age: u32,
    sex: Sex,
}

impl Employ {
    pub fn new(name: String, age: u32, sex: Sex) -> Employ {
        return Employ {
            name: name,
            age: age,
            sex: sex,
        };
    }
    // 第一个参数self为实例本身
    pub fn to_string(&self) -> String {
        return self.name.to_string();
    }
}
#[derive(Debug)]
pub enum Sex {
    Main = 0,
    Woman = 1,
}
```

创建NETCORE新项目

安装netcore

- runtime:https://dotnet.microsoft.com/zh-cn/download
- vscode扩展:https://marketplace.visualstudio.com/items?itemName=ms-dotnettools.csdevkit

创建项目

```
# 查看已经安装的模板
dotnet new list

# 创建新的项目
# dotnet new <template name> -o <project_name>
dotnet new web -o web-api
```

速查知识库

获取应用程序启动参数

```
    let args: Vec<String> = std::env::args().collect();

    for arg in args {
        println!("{}", arg);
    }
```

打印日志

```
# Cargo.toml
# 增加依赖



[dependencies]
log = "0.4.20"
simple_logger = "4.3.0"

# 特别注意，rust将日志接口与日志实现分开了，所以需要两个组件
```

```
# src/main.rs

use log::{debug, info, trace, warn};
use simple_logger::SimpleLogger;

fn main(){
    // 初始化日志组件
    SimpleLogger::new().init().unwrap();
    info!("application start");
}
```

JSON操作

```
# Cargo.toml
# 增加依赖
[dependencies]

serde = { version = "1.0", features = ["derive"] }
serde_json = "1.0.68"
```

```
# src/main.rs

use log::{debug, info, trace, warn};
use serde::{Deserialize, Serialize};
use serde_json::json;
use simple_logger::SimpleLogger;

#[derive(Serialize, Deserialize)]
struct Person {
    name: String,
    age: u32,
}

async fn root() -> impl IntoResponse {
    info!("entry root");

    // 多行文本与引号可以用 r## 处理
    // 构建一段JSON字符串
    let json_str = r#"{"name": "John Doe", "age": 30}"#;

    // 将JSON字符反序列化为对象
    let json_obj: Person = serde_json::from_str(json_str).unwrap();

    // 将对象序列化为JSON字符串
    let json_str = serde_json::to_string(&json_obj).unwrap();
    log::info!("{}", json_str);

    // 创建一个匿名的JSON对象
    let json_obj = json!({"name": "John Doe", "age": 30});

    // 将匿名JSON对象序列号为字符串
```

```
    let json_str = json_obj.to_string();
    log::info!("{}", json_str);

    // 将匿名JSON对象转换为对象
    let json_obj: Person = serde_json::from_value(json_obj).unwrap();
    log::info!("{}", json_obj.name);

    return (StatusCode::OK, Json(json_obj));
}
```

常用库索引

两个网站

- https://lib.rs/
- https://crates.io/

常用类库

```
[dependencies]
# web 框架
axum = "0.7.2"
# 一个高性能的IO库
tokio = { version = "1.35.1", features = ["full"] }
# http client
reqwest = "0.11.23"
# 日志
log = "0.4.20"
# 日志实现
simple_logger = "4.3.0"
# 序列化
serde = { version = "1.0", features = ["derive"] }
# json序列化
serde_json = "1.0.68"
```

实现RESTful API服务

项目依赖

```
[dependencies]
# 一个模块化的web应用程序框架
axum = "0.7.2"
# 一个事件驱动的非阻塞I/O库
tokio = { version = "1.35.1", features = ["full"] }
# http client
reqwest = "0.11.23"
# 日志抽象
log = "0.4.20"
# 日志实现
simple_logger = "4.3.0"
# 序列化
serde = { version = "1.0", features = ["derive"] }
# json序列化
serde_json = "1.0.68"
```

创建一个http server

```
# src/main.rs



// 引入模块
use axum::{
    http::StatusCode,
    response::{Html, IntoResponse, Json},
    routing::{get, post},
    Router,
};
use log::{debug, info, trace, warn};
use serde::{Deserialize, Serialize};
use serde_json::json;
use simple_logger::SimpleLogger;
```

```rust
#[tokio::main]
async fn main() {
    // 初始化日志组件
    SimpleLogger::new().init().unwrap();
    debug!("power by iovhm.com");
    warn!("©copyRight");
    trace!("2023年12月27日');
    info!("application start");

    let app = Router::new()
        // GET / 首页
        .route("/", get(root));

    let listener = tokio::net::TcpListener::bind("0.0.0.0:3000").await.unwrap();
    info!("server listen at 0.0.0.0:3000");

    axum::serve(listener, app).await.unwrap();
}

#[derive(Serialize, Deserialize)]
struct Person {
    name: String,
    age: u32,
}

async fn root() -> impl IntoResponse {
    info!("entry root");

    // 创建一个匿名的JSON对象
    let json_obj = json!({"name": "John Doe", "age": 30});

    return Json(json_obj);
}
```