



Rust 入门导学（1）

Mike Tang
mike@cdot.network
2020-8

本节课目的



1. 了解 Rust 语言的主要特点
2. 学习 Rust 语言的基本类型和基本运算符
3. 学习 Rust 语言的控制结构
4. 了解 Rust 语言的字符串
5. 学习 Rust 语言的枚举与模式匹配
6. 学习 Rust 语言的 Result 与 Option
7. 学习 Rust 语言的错误处理

Rust 语言的主要特点



- 高性能
- 内存安全
- 无忧并发（程序的开发）

高性能



- 与 C/C++ 一个级别的运行速度
- 方法抉择：
 - Zero Cost Abstract 零开销抽象
 - 无 GC 的自动内存管理 RAII
 - 可做到 C ABI 一致的设计

内存安全



- 使用 Rust（非 unsafe 部分）写出来的代码，保证内存安全
- 方法抉择：
 - Ownership, move 语义
 - Borrowchecker
 - 强类型系统
 - 无空值（Null, nil 等）设计

无忧并发



- 使用 Rust 进行多线程以及多任务并发代码开发，不会出现数据竞争和临界值破坏
- 方法抉择：
 - 对并发进行了抽象 Sync, Send
 - 融入类型系统
 - 基于 Ownership, Borrowchecker 实现，完美的融合性

Rust 代码长啥样



```
fn main() {  
    let mut x = 5;  
    println!("The value of x is: {}", x);  
    x = 6;  
    println!("The value of x is: {}", x);  
}
```

都说 Rust 难学，其实也很简单嘛～

Rust 语法和基础数据类型



- 完整的语法参考: <https://doc.rust-lang.org/stable/reference/introduction.html>
- 官方 Rust 教程书: <https://doc.rust-lang.org/book/>
- 初学者感受
 - 类型名标在变量名的后面, 中间用冒号隔开。比如: `x: usize`
 - if 条件变量上没有括号, 并且花括号不可省。比如: `if a>0 {println!("x")}`
 - 定义变量要用 `let` 这个东东, 其它语言中大部分都不用的
 - 变量是否可修改, 通过 `mut` 这个 keyword 来修饰
 - 最普通的打印语句后面竟然有个 `!` 号。比如: `println!("Hello world!");`
 - 函数或块的最后一个语句 (表达式) 可以不同分号, 就表示返回这个表达式的值
 -

基础数据类型 - 整数、浮点数



```
fn main() {  
    let x = 2.0; // f64  
  
    let y: f32 = 3.0; // f32  
}
```

基础数据类型 - 布尔型



```
fn main() {  
    let t = true;  
  
    let f: bool = false;  
}
```

基础数据类型 - 字符 Char



```
fn main() {  
    let c = 'z';  
    let z = 'Z';  
    let heart_eyed_cat = '😻';  
}
```

基础数据类型 - 元组 Tuple



```
fn main() {  
    let tup = (500, 6.4, 1);  
  
    let (x, y, z) = tup;  
  
    println!("The value of y is: {}", y);  
}
```

基础数据类型 - 数组 Array



```
let a: [i32; 5] = [1, 2, 3, 4, 5];
```

```
fn main() {  
    let a = [1, 2, 3, 4, 5];  
  
    let first = a[0];  
    let second = a[1];  
}
```

控制结构 - 函数代码块



```
fn main() {  
    another_function(5);  
}  
  
fn another_function(x: i32) {  
    println!("The value of x is: {}", x);  
}
```

控制结构 - If 表达式



```
fn main() {  
    let number = 3;  
  
    if number < 5 {  
        println!("condition was true");  
    } else {  
        println!("condition was false");  
    }  
}
```

控制结构 - 无条件循环（死循环）



```
fn main() {  
    let mut counter = 0;  
  
    let result = loop {  
        counter += 1;  
  
        if counter == 10 {  
            break counter * 2;  
        }  
    };  
  
    println!("The result is {}", result);  
}
```


控制结构 - While 循环



```
fn main() {  
    let mut number = 3;  
  
    while number != 0 {  
        println!("{}", number);  
  
        number -= 1;  
    }  
  
    println!("LIFTOFF!!!");  
}
```

控制结构 - For 迭代



```
fn main() {  
    let a = [10, 20, 30, 40, 50];  
  
    for element in a.iter() {  
        println!("the value is: {}", element);  
    }  
}
```

Cargo 时间



- Cargo 包管理体系介绍
- 此处留空，我们在终端里面操作一下 Rust 项目创建，编写，编译，运行的整个流程。



好像都挺简单，没有什么特别的。

咦，好像还没介绍字符串。

字符串： &str



// 字符串面值

```
let hello = "Hello, world!";
```

// 附带显式类型标识

```
let hello: &'static str = "Hello, world!";
```

字符串： String



```
// 创建一个空的字符串
let mut s = String::new();
// 从 `&str` 类型转化成 `String` 类型
let mut hello = String::from("Hello, ");
// 压入字符和压入字符串切片
hello.push('w');
hello.push_str("orld!");

// 弹出字符。
let mut s = String::from("foo");
assert_eq!(s.pop(), Some('o'));
assert_eq!(s.pop(), Some('o'));
assert_eq!(s.pop(), Some('f'));
assert_eq!(s.pop(), None);
```

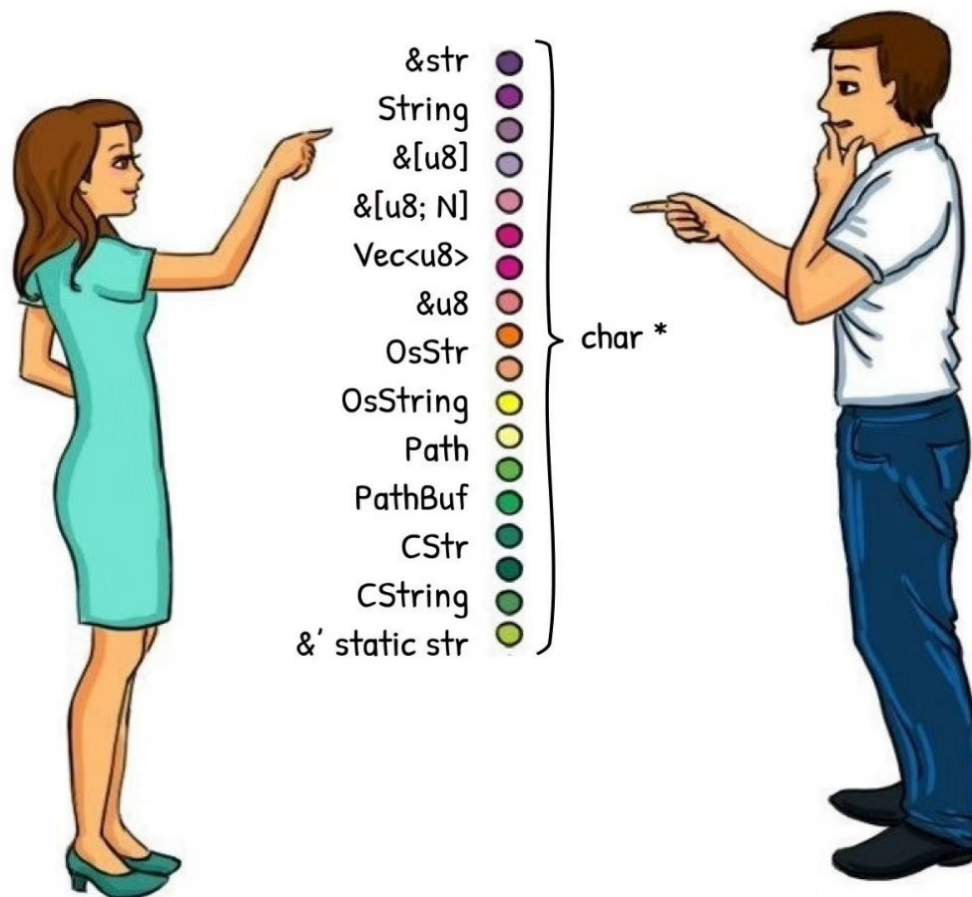
字符串：More



How We See Strings

Rust

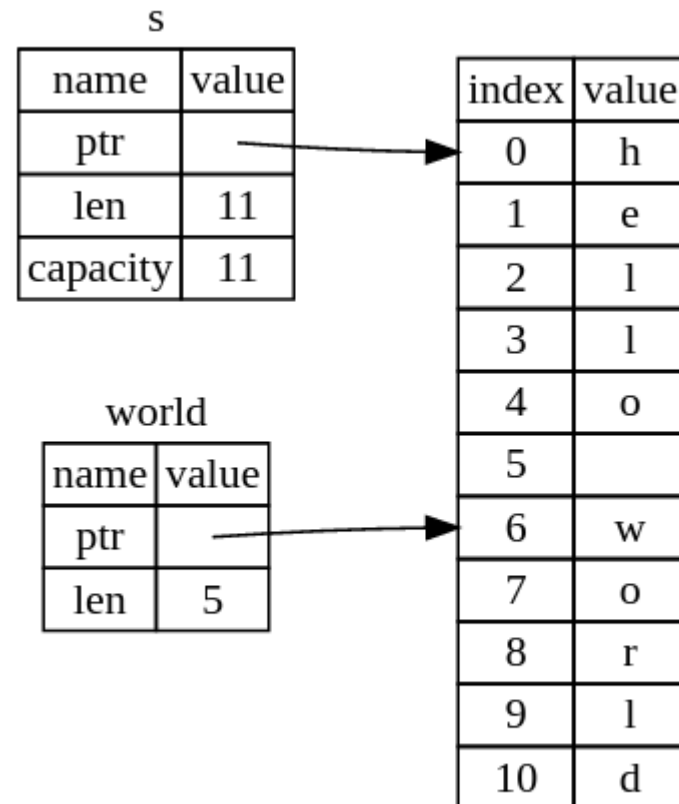
C



Slice



```
let s = String::from("hello world");  
  
let hello = &s[0..5];  
let world = &s[6..11];
```



复合类型 - 结构体



```
struct User {  
    username: String,  
    email: String,  
    sign_in_count: u64,  
    active: bool,  
}
```

结构体的初始化和字段更新



```
let mut user1 = User {  
    email: String::from("someone@example.com"),  
    username: String::from("someusername123"),  
    active: true,  
    sign_in_count: 1,  
};  
  
user1.email = String::from("anotheremail@example.com");
```

元组结构体（匿名字段结构体）



```
struct Color(i32, i32, i32);  
struct Point(i32, i32, i32);  
  
let black = Color(0, 0, 0);  
let origin = Point(0, 0, 0);
```

裸结构体



```
struct Point;
```

复合类型 - 枚举



```
enum IpAddrKind {  
    V4,  
    V6,  
}
```

```
let four = IpAddrKind::V4;  
let six = IpAddrKind::V6;
```

枚举的基本使用



```
enum IpAddrKind {  
    V4,  
    V6,  
}  
  
struct IpAddr {  
    kind: IpAddrKind,  
    address: String,  
}  
  
let home = IpAddr {  
    kind: IpAddrKind::V4,  
    address: String::from("127.0.0.1"),  
};  
  
let loopback = IpAddr {  
    kind: IpAddrKind::V6,  
    address: String::from("::1"),  
};
```

类 C 的枚举



```
// An attribute to hide warnings for unused code.
#![allow(dead_code)]

// enum with implicit discriminator (starts at 0)
enum Number {
    Zero,
    One,
    Two,
}

// enum with explicit discriminator
enum Color {
    Red = 0xff0000,
    Green = 0x00ff00,
    Blue = 0x0000ff,
}

fn main() {
    // `enums` can be cast as integers.
    println!("zero is {}", Number::Zero as i32);
    println!("one is {}", Number::One as i32);

    println!("roses are #{:06x}", Color::Red as i32);
    println!("violets are #{:06x}", Color::Blue as i32);
}
```

强大表现力的枚举



```
enum Message {  
    Quit,  
    Move { x: i32, y: i32 },  
    Write(String),  
    ChangeColor(i32, i32, i32),  
}
```


对等表示



```
struct QuitMessage; // unit struct
struct MoveMessage {
    x: i32,
    y: i32,
}
struct WriteMessage(String); // tuple struct
struct ChangeColorMessage(i32, i32, i32); // tuple struct
```

```
enum Message {
    Quit(QuitMessage),
    Move(MoveMessage),
    Write(WriteMessage),
    ChangeColor(ChangeColorMessage),
}
```

模式匹配 示例 1



```
enum Coin {  
    Penny,  
    Nickel,  
    Dime,  
    Quarter,  
}  
  
fn value_in_cents(coin: Coin) -> u8 {  
    match coin {  
        Coin::Penny => 1,  
        Coin::Nickel => 5,  
        Coin::Dime => 10,  
        Coin::Quarter => 25,  
    }  
}
```

模式匹配 示例 2



```
#[derive(Debug)] // so we
enum UsState {
    Alabama,
    Alaska,
    // --snip--
}
```

```
enum Coin {
    Penny,
    Nickel,
    Dime,
    Quarter(UsState),
}
```

```
fn value_in_cents(coin: Coin) -> u8 {
    match coin {
        Coin::Penny => 1,
        Coin::Nickel => 5,
        Coin::Dime => 10,
        Coin::Quarter(state) => {
            println!("State quarter from {:?}!", state);
            25
        }
    }
}
```

Result 与 Option



- Result 与 Option 其实就是两个特定的枚举
- 不特殊，因为就是普通的枚举
- 特殊，因为整个 std 标准库基于其打造。整个错误处理体系基于其打造

```
pub enum Result<T, E> {  
    Ok(T),  
    Err(E),  
}
```

```
pub enum Option<T> {  
    None,  
    Some(T),  
}
```

Rust 中无空值



- Option 代表一种通用的 空。其它语言中的空值往往用 NULL , 0 , nil 或类似的表达, 实际上还是处于一个维度之中。空值存在于变量取值范围之中。 Rust 的 Option 相当于加入了一个新的维度。于是, Rust 中无空值的概念。

Error Handling



- 基于 Result/Option + 模式匹配的错误处理方式
- 从形式上要求你必须做完整的错误处理，如果忘了做，编译器会警告你（像个好管家）
- 无 try-catch，要求对代码错误更精确仔细的处理
- ? 号，防守性编程
- 有了 Option，就不需要 null-pointer 了（Rust 中无空指针）
- 错误的传递和归纳整理，是一门艺术（专题）

错误处理示例



```
use std::fs::File;
use std::io::ErrorKind;

fn main() {
    let f = File::open("hello.txt");

    let f = match f {
        Ok(file) => file,
        Err(error) => match error.kind() {
            ErrorKind::NotFound => match File::create("hello.txt") {
                Ok(fc) => fc,
                Err(e) => panic!("Problem creating the file: {:?}", e),
            },
            other_error => {
                panic!("Problem opening the file: {:?}", other_error)
            }
        },
    };
}
```



Thank You

Q&A

