第二课: 介绍Rust类型系统

类型大小、类型推导、泛型、Trait、类型转换

苏林

今天公开课内容

1、类型大小　类型在内存中对齐、布局

2、类型推导

3、泛型

4、Trait

5、类型转换

Rust是一门《显式静态强类型的类型安全语言》

1、显式
=> 是因为它的类型推导在某些时候需要显示指定

2、静态
=> 表明它在编译期进行类型检查

3、强类型
=> 表明它不允许类型自动隐式转换，不同类型无法进行计算

4、类型安全
=> 表明它保证运行时的内存安全

类型大小

思考: 为什么我们要知道类型的大小?
=> 编译期为变量/值分配地址 ->
Rust内存首先由编译器来分配, Rust代码编译为LLVM IR中间语言. 其中携带了内存分配的信息.

可确定大小类型

```rust
fn main() {
    println!("{}", std::mem::size_of::<bool>());
    println!("{}", std::mem::size_of::<u8>());
    println!("{}", std::mem::size_of::<u16>());
    println!("{}", std::mem::size_of::<u32>());
    println!("{}", std::mem::size_of::<i8>());
    println!("{}", std::mem::size_of::<i16>());
    println!("{}", std::mem::size_of::<i32>());
    println!("{}", std::mem::size_of::<f32>());
    println!("{}", std::mem::size_of::<char>());
}
```

动态大小类型

```rust
fn main() {

    let str : &str = "Hello World!";

    let ptr : *const u8 = str.as_ptr();

    let len : usize = str.len();

    println!("{:p}", ptr);

    println!("{:?}", len);

}
```

类型大小

动态大小类型

```rust
fn reset(mut arr: [u32]) {
    arr[0] = 5;
    arr[1] = 4;
    arr[2] = 3;
    arr[3] = 2;
    arr[4] = 1;

    println!("reset arr {:?}", arr);
}

fn main() {
    let arr: [u32] = [1, 2, 3, 4, 5];
    reset(arr);
    println!("reset arr {:?}", arr);
}
```

类型大小

零大小类型: -> 典型的特点: 可以提高性能

```rust
enum Void {}
struct Foo;
struct Baz {
    foo: Foo,
    qux: (),
    baz: [u8; 0],
}

fn main() {
    println!("{}", std::mem::size_of::<()>());
    println!("{}", std::mem::size_of::<Foo>());
    println!("{}", std::mem::size_of::<Baz>());
    println!("{}", std::mem::size_of::<Void>());
    println!("{}", std::mem::size_of::<[(); 10]>());
}
```

类型大小

零大小类型——优势？

```rust
fn main() {
    let v: Vec<()> = vec![(); 10];
    for i: () in v {
        println!("{:?}", i);
    }
}
```

类型推导

```rust
fn sum(a: u32, b: i32) -> u32 {
    a + (b as u32)
}

fn main() {
    let a : u32 = 1;
    let b : i32 = 2;
    sum(a, b);
    let elem : u8 = 5u8;
    let mut vec : Vec<u8> = Vec::new();
    vec.push( value: elem);
}
```

类型推导

```
1  fn main() {
2      let x : &str = "1";
3      println!("{:?}", x.parse().unwrap());
4  }
```

turbofish 操作符 ::<>

泛型

```rust
fn add_i32(a: i32, b: i32) -> i32 {
    a + b
}

fn add_i64(a: i64, b: i64) -> i64 {
    a + b
}

fn add_u32(a: u32, b: u32) -> u32 {
    a + b
}

fn add_u64(a: u64, b: u64) -> u64 {
    a + b
}
```

泛型

```rust
fn add<T>(a: T, b: T) -> T {
    a + b
}
```

单态化 零成本抽象的一种实现

Trait 为Rust提供了零成本抽象能力.

接口抽象和泛型约束

```rust
trait Shape {
    fn desc(&self) -> String;
}
struct Circle {}
struct Triangle {}

impl Shape for Circle {
    fn desc(&self) -> String {
        "this is a circle".to_string()
    }
}

impl Shape for Triangle {
    fn desc(&self) -> String {
        "this is a triangle".to_string()
    }
}

fn get_desc_from_shape<T: Shape>(shape: T) {
    println!("describe: {:?}", shape.desc());
}

```

Trait 为Rust提供了零成本抽象能力.

可以预先实现方法, 节省具体类型的实现工作

```rust
trait ShapePrinter {
    fn print_shape(&self) {
        println!("this is a type implemented ShapePrinter");
    }
}

impl ShapePrinter for Circle {}

fn print_shape<T: ShapePrinter>(shape: T) {
    shape.print_shape();
}

fn main() {
    let c = Circle{} ;
    print_shape( shape: c);
}
```

类型转换

基本类型转换

```rust
fn main() {
    let a :u32 = 1u32;
    let b :u64 = a as u64;
}
```

类型转换

无歧义完全限定语法

QA环节

加群一起交流Rust & Databend

Databend

Rust语言中文社区

知数堂