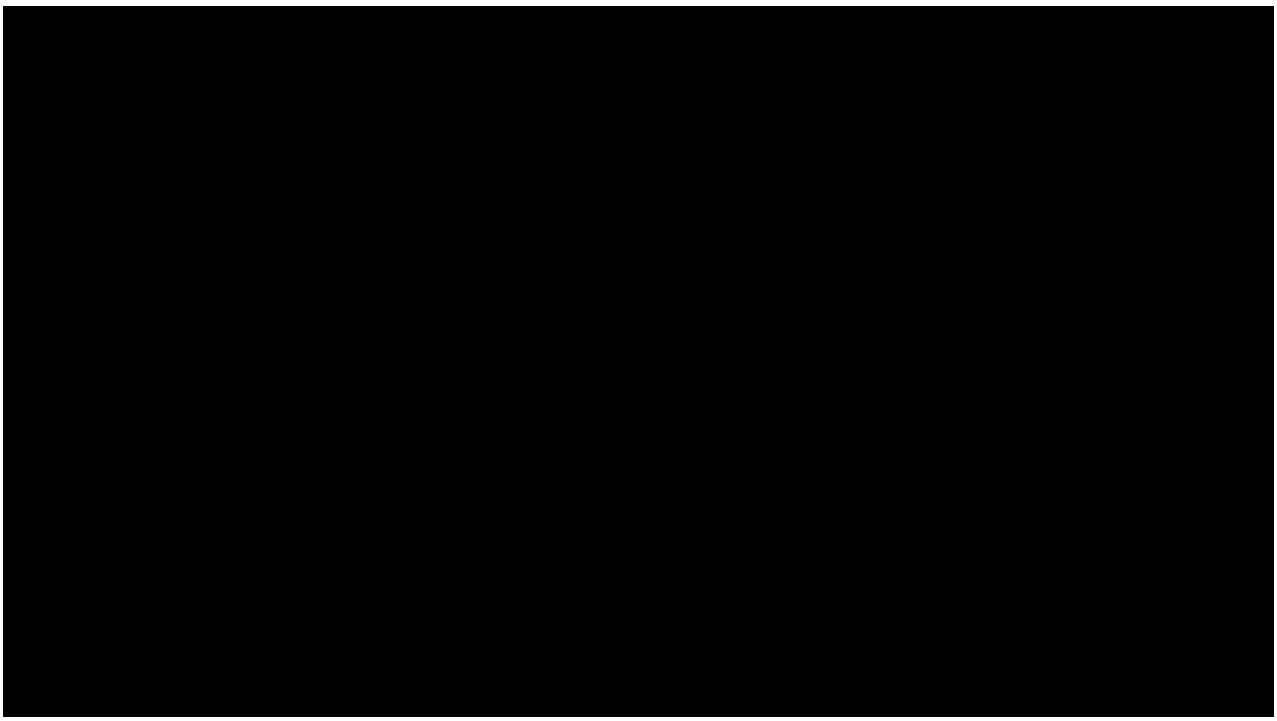
02｜串讲：编程开发中，那些你需要掌握的基本概念

time.geekbang.org/column/article/410038

陈天 2021-08-22



00:00

1.0x
讲述：陈天大小：16.68M时长：18:12

你好，我是陈天。

上一讲我们了解了内存的基本运作方式，简单回顾一下：栈上存放的数据是静态的，固定大小，固定生命周期；堆上存放的数据是动态的，不固定大小，不固定生命周期。

今天我们来继续梳理，编程开发中经常接触到的其它基本概念。需要掌握的小概念点比较多，为了方便你学习，我把它们分为四大类来讲解：数据（值和类型、指针和引用）、代码（函数、方法、闭包、接口和虚表）、运行方式（并发并行、同步异步和 Promise / async / await ），以及编程范式（泛型编程）。

希望通过重温这些概念，你能够夯实软件开发领域的基础知识，这对你后续理解 Rust 里面的很多难点至关重要，比如所有权、动态分派、并发处理等。

好了，废话不多说，我们马上开始。

数据

数据是程序操作的对象，不进行数据处理的程序是没有意义的，我们先来重温和数据有关的概念，包括值和类型、指针和引用。

值和类型

严谨地说，类型是对值的区分，它包含了值在内存中的长度、对齐以及值可以进行的操作等信息。一个值是符合一个特定类型的数据的某个实体。比如 64u8，它是 u8 类型，对应一个字节大小、取值范围在 0 ~ 255 的某个整数实体，这个实体是 64。

值以类型规定的表达方式（representation）被存储成一组字节流进行访问。比如 64，存储在内存中的表现形式是 0x40，或者 0b 0100 0000。

这里你要注意，值是无法脱离具体的类型讨论的。同样是内存中的一个字节 0x40，如果其类型是 ASCII char，那么其含义就不是 64，而是 @ 符号。

不管是强类型的语言还是弱类型的语言，语言内部都有其类型的具体表述。一般而言，编程语言的类型可以分为原生类型和组合类型两大类。

原生类型（primitive type）是编程语言提供的最基础的数据类型。比如字符、整数、浮点数、布尔值、数组（array）、元组（tuple）、指针、引用、函数、闭包等。所有原生类型的大小都是固定的，因此它们可以被分配到栈上。

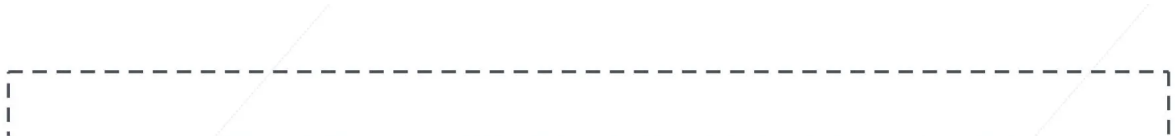
组合类型（composite type）或者说复合类型，是指由一组原生类型和其它类型组合而成的类型。组合类型也可以细分为两类：

结构体（structure type）：多个类型组合在一起共同表达一个值的复杂数据结构。比如 Person 结构体，内部包含 name、age、email 等信息。用代数数据类型（algebraic data type）的说法，结构体是 product type。

标签联合（tagged union）：也叫不相交并集（disjoint union），可以存储一组不同但固定的类型中的某个类型的对象，具体是哪个类型由其标签决定。比如 Haskell 里的 Maybe 类型，或者 Swift 中的 Optional 就是标签联合。用代数数据类型的说法，标签联合是 sum type。

另外不少语言不支持标签联合，只取其标签部分，提供了枚举类型（enumerate）。枚举是标签联合的子类型，但功能比较弱，无法表达复杂的结构。

看定义可能不是太好理解，你可以看这张图：



指针和引用

在内存中，一个值被存储到内存中的某个位置，这个位置对应一个内存地址。而指针是一个持有内存地址的值，可以通过解引用（deference）来访问它指向的内存地址，理论上可以解引用到任意数据类型。

引用（reference）和指针非常类似，不同的是，引用的解引用访问是受限的，它只能解引用到它引用数据的类型，不能用作它用。比如，指向 42u8 这个值的一个引用，它解引用的时候只能使用 u8 数据类型。

所以，指针的使用限制更少，但也会带来更多的危害。如果没有用正确的类型解引用一个指针，那么会引发各种各样的内存问题，造成系统崩溃或者潜在的安全漏洞。

刚刚讲过，指针和引用是原生类型，它们可以分配在栈上。

根据指向数据的不同，某些引用除了需要一个指针指向内存地址之外，还需要内存地址的长度和其它信息。

如上一讲提到的指向 “hello world” 字符串的指针，还包含字符串长度和字符串的容量，一共使用了 3 个 word，在 64 位 CPU 下占用 24 个字节，这样比正常指针携带更多信息的指针，我们称之为胖指针（fat pointer）。很多数据结构的引用，内部都是由胖指针实现的。

代码

数据是程序操作的对象，而代码是程序运行的主体，也是我们开发者把物理世界中的需求转换成数字世界中逻辑的载体。我们会讨论函数和闭包、接口和虚表。

函数、方法和闭包

函数是编程语言的基本要素，它是对完成某个功能的一组相关语句和表达式的封装。函数也是对代码中重复行为的抽象。在现代编程语言中，函数往往是一等公民，这意味着函数可以作为参数传递，或者作为返回值返回，也可以作为复合类型中的一个组成部分。

在面向对象的编程语言中，在类或者对象中定义的函数，被称为方法（method）。方法往往和对象的指针发生关系，比如 Python 对象的 self 引用，或者 Java 对象的 this 引用。

而闭包是将函数，或者说代码和其环境一起存储的一种数据结构。闭包引用的上下文中的自由变量，会被捕获到闭包的结构中，成为闭包类型的一部分。

一般来说，如果一门编程语言，其函数是一等公民，那么它必然会支持闭包（closure），因为函数作为返回值往往需要返回一个闭包。

你可以看这张图辅助理解，图中展示了一个闭包对上下文环境的捕获。可以 在这里 运行这段代码：

接口和虚表

接口是一个软件系统开发的核心部分，它反映了系统的设计者对系统的抽象理解。作为一个抽象层，接口将使用方和实现方隔离开来，使两者不直接有依赖关系，大大提高了复用性和扩展性。

很多编程语言都有接口的概念，允许开发者面向接口设计，比如 Java 的 interface、Elixir 的 behaviour、Swift 的 protocol 和 Rust 的 trait。

比如说，在 HTTP 中，Request/Response 的服务处理模型其实就是一个典型的接口，我们只需要按照服务接口定义出不同输入下，从 Request 到 Response 具体该如何映射，通过这个接口，系统就可以在合适的场景下，把符合要求的 Request 分派给我们的服务。

面向接口的设计是软件开发中的重要能力，而 Rust 尤其重视接口的能力。在后续讲到 Trait 的章节，我们会详细介绍如何用 Trait 来进行接口设计。

当我们在运行期使用接口来引用具体类型的时候，代码就具备了运行时多态的能力。但是，在运行时，一旦使用了关于接口的引用，变量原本的类型被抹去，我们无法单纯从一个指针分析出这个引用具备什么样的能力。

因此，在生成这个引用的时候，我们需要构建胖指针，除了指向数据本身外，还需要指向一张涵盖了这个接口所支持方法的列表。这个列表，就是我们熟知的虚表（virtual table）。

下图展示了一个 Vec 数据在运行期被抹去类型，生成一个指向 Write 接口引用的过程：

由于虚表记录了数据能够执行的接口，所以在运行期，我们想对一个接口有不同实现，可以根据上下文动态分派。

比如我想为一个编辑器的 Formatter 接口实现不同语言的格式化工具。我们可以在编辑器加载时，把所有支持的语言和其格式化工具放入一个哈希表中，哈希表的 key 为语言类型，value 为每种格式化工具 Formatter 接口的引用。这样，当用户在编辑器打开某个文件的时候，我们可以根据文件类型，找到对应 Formatter 的引用，来进行格式化操作。

运行方式

程序在加载后，代码以何种方式运行，往往决定着程序的执行效率。所以我们接下来讨论并发、并行、同步、异步以及异步中的几个重要概念 Promise/async/await。

并发（concurrency）与并行（parallel）

并发和并行是软件开发中经常遇到的概念。

并发是同时与多件事情打交道的能力，比如系统可以在任务 1 做到一定程度后，保存该任务的上下文，挂起并切换到任务 2，然后过段时间再切换回任务 1。

并行是同时处理多件事情的手段。也就是说，任务 1 和任务 2 可以在同一个时间片下工作，无需上下文切换。下图很好地阐释了二者的区别：

并发是一种能力，而并行是一种手段。当我们的系统拥有了并发的能力后，代码如果跑在多个 CPU core 上，就可以并行运行。所以我们平时都谈论高并发处理，而不会说高并行处理。

很多拥有高并发处理能力的编程语言，会在用户程序中嵌入一个 M:N 的调度器，把 M 个并发任务，合理地分配在 N 个 CPU core 上并行运行，让程序的吞吐量达到最大。

同步和异步

同步是指一个任务开始执行后，后续的操作会阻塞，直到这个任务结束。在软件中，我们大部分的代码都是同步操作，比如 CPU，只有流水线中的前一条指令执行完成，才会执行下一条指令。一个函数 A 先后调用函数 B 和 C，也会执行完 B 之后才执行 C。

同步执行保证了代码的因果关系（causality），是程序正确性的保证。

然而在遭遇 I/O 处理时，高效 CPU 指令和低效 I/O 之间的巨大鸿沟，成为了软件的性能杀手。下图对比了 CPU、内存、I/O 设备、和网络的延迟：
我们可以看到和内存访问相比，I/O 操作的访问速度低了两个数量级，一旦遇到 I/O 操作，CPU 就只能闲置来等待 I/O 设备运行完毕。因此，操作系统为应用程序提供了异步 I/O，让应用可以在当前 I/O 处理完毕之前，将 CPU 时间用作其它任务的处理。

所以，异步是指一个任务开始执行后，与它没有因果关系的其它任务可以正常执行，不必等待前一个任务结束。

在异步操作里，异步处理完成后的结果，一般用 Promise 来保存，它是一个对象，用来描述在未来的某个时刻才能获得的结果的值，一般存在三个状态；

初始状态，Promise 还未运行；

等待（pending）状态，Promise 已运行，但还未结束；

结束状态， Promise 成功解析出一个值，或者执行失败。

如果你对 Promise 这个词不太熟悉，在很多支持异步的语言中，Promise 也叫 Future / Delay / Deferred 等。除了这个词以外，我们也经常看到 async/await 这对关键字。

一般而言，async 定义了一个可以并发执行的任务，而 await 则触发这个任务并发执行。大多数语言中，async/await 是一个语法糖（syntactic sugar），它使用状态机将 Promise 包装起来，让异步调用的使用感觉和同步调用非常类似，也让代码更容易阅读。

编程范式

为了在不断迭代时，更好地维护代码，我们还会引入各种各样的编程范式，来提升代码的质量。所以最后来谈谈泛型编程。

如果你来自于弱类型语言，如 C / Python / JavaScript，那泛型编程是你需要重点掌握的概念和技能。泛型编程包含两个层面，数据结构的泛型和使用泛型结构代码的泛型化。

数据结构的泛型

首先是数据结构的泛型，它也往往被称为参数化类型或者参数多态，比如下面这个数据结构：

```
struct Connection<S> {

io: S,

state: State,

}
```

它有一个参数 S，其内部的域 io 的类型是 S，S 具体的类型只有在使用 Connection 的上下文中才得到绑定。

你可以把参数化数据结构理解成一个产生类型的函数，在“调用”时，它接受若干个使用了具体类型的参数，返回携带这些类型的类型。比如我们为 S 提供 TcpStream 这个类型，那么就产生 Connection这个类型，其中 io 的类型是 TcpStream。

这里你可能会疑惑，如果 S 可以是任意类型，那我们怎么知道 S 有什么行为？如果我们要调用 io.send() 发送数据，编译器怎么知道 S 包含这个方法？

这是个好问题，我们需要用接口对 S 进行约束。所以我们经常看到，支持泛型编程的语言，会提供强大的接口编程能力，在后续的课程中在讲 Rust 的 trait 时，我会再详细探讨这个问题。

数据结构的泛型是一种高级抽象，就像我们人类用数字抽象具体事物的数量，又发明了代数来进一步抽象具体的数字一样。它带来的好处是我们可以延迟绑定，让数据结构的通用性更强，适用场合更广阔；也大大减少了代码的重复，提高了可维护性。

代码的泛型化

泛型编程的另一个层面是使用泛型结构后代码的泛型化。当我们使用泛型结构编写代码时，相关的代码也需要额外的抽象。

这里用我们熟悉的二分查找的例子解释会比较清楚：

左边用 C 撰写的二分查找，标记的几处操作隐含着和 int[] 有关，所以如果对不同的数据类型做二分查找，实现也要跟着改变。右边 C++ 的实现，对这些地方做了抽象，让我们可以用同一套代码二分查找迭代器（iterator）的数据类型。

同样的，这样的代码可以在更广阔的场合使用，更简洁容易维护。

小结

今天我们讨论了四大类基础概念：数据、代码、运行方式和编程范式。

值无法离开类型单独讨论，类型一般分为原生类型和组合类型。指针和引用都指向值的内存地址，只不过二者在解引用时的行为不一样。引用只能解引用到原来的数据类型，而指针没有这个限制，然而，不受约束的指针解引用，会带来内存安全方面的问题。

函数是代码中重复行为的抽象，方法是对象内部定义的函数，而闭包是一种特殊的函数，它会捕获函数体内使用到的上下文中的自由变量，作为闭包成员的一部分。

而接口将调用者和实现者隔离开，大大促进了代码的复用和扩展。面向接口编程可以让系统变得灵活，当使用接口去引用具体的类型时，我们就需要虚表来辅助运行时代码的执行。有了虚表，我们可以很方便地进行动态分派，它是运行时多态的基础。

在代码的运行方式中，并发是并行的基础，是同时与多个任务打交道的能力；并行是并发的体现，是同时处理多个任务的手段。同步阻塞后续操作，异步允许后续操作。被广泛用于异步操作的 Promise 代表未来某个时刻会得到的结果，async/await 是 Promise 的封装，一般用状态机来实现。

泛型编程通过参数化让数据结构像函数一样延迟绑定，提升其通用性，类型的参数可以用接口约束，使类型满足一定的行为，同时，在使用泛型结构时，我们的代码也需要更高的抽象度。

这些基础概念，这对于后续理解 Rust 的很多概念至关重要。如果你对某些概念还是有些模糊，务必留言，我们可以进一步讨论。

思考题

（现在我们还没有讲到 Rust 的具体语法，所以你可以用自己平时常用的语言来思考这几道题，巩固你对基本概念的理解）

1. 有一个指向某个函数的指针，如果将其解引用成一个列表，然后往列表中插入一个元素，请问会发生什么？（对比不同语言，看看这种操作是否允许，如果允许会发生什么）

2. 要构造一个数据结构 Shape，可以是 Rectangle、 Circle 或是 Triangle，这三种结构见如下代码。请问 Shape 类型该用什么数据结构实现？怎么实现？

```
struct Rectangle {

a: f64,

b: f64,

}

struct Circle {

r: f64,

}

struct Triangle {

a: f64,

b: f64,

c: f64,

}
```

3. 对于上面的三种结构，如果我们要定义一个接口，可以计算周长和面积，怎么计算？

欢迎在留言区分享你的思考。今天是我们打卡学习的第二讲，如果你觉得有收获，也欢迎你分享给你身边的朋友，邀 TA 一起讨论。

参考资料

Latency numbers every programmer should know，对比了 CPU、内存、I/O 设备、和网络的延迟

20人觉得很赞给文章提建议

良师益友

Command + Enter 发表

0/2000字

提交留言

精选留言(40)

- Geek_5b5ca4

  置顶
  Python 是强类型动态语言

  作者回复: 这里我写的不太严谨。强类型和弱类型的定义一直不太明确，wikipedia 上也没有一个标准的说法。
  https://en.wikipedia.org/wiki/Strong_and_weak_typing。我一般是看类型在调用时是否会发生隐式转换，所以说 python 是弱类型。不过 wikipedia 在介绍 python 时确实说它是 strongly typed：
  https://en.wikipedia.org/wiki/Python_(programming_language)。

  但如果按照类型是否会隐式转换，Rust 是强类型，Python 和 C 是弱类型：

  ```rust
  fn main() {
    let a = 42u8;
    let b = 42.0f64;



    // 不会做隐式转换
    println!("a+b = {}", a+b);
  }
  ```

  Python 会做类型的隐式转换：

  ```python
  def add_numbers(a, b):
    return a + b

  if __name__ == '__main__':
    a = 42
    b = 42.0
    print(add_numbers(a,b))
  ```

  C 也会：

  ```C
  #include <stdio.h>

  int add_numbers(int a, int b) {
    int result = a + b;
    return result;
  }
  ```

```
int main() {
  char c = "42";
  int n = 42;

  // 为什么说 C 是 weakly typed
  int result = add_numbers(c, n);
  printf("%d\n", result);

  return 0;
}
```

2021-08-24

1

21

- 无名氏

置顶
虚表没有理解，虚表会存储在哪里

作者回复: 虚表相当于在运行时生成的一个涵盖了一系列函数指针的数据结构。有时候对于不同类型但是满足相同接口的数据，我们希望可以抹去他们的原始类型，让它们有相同的接口类型，以便于统一处理，这样更加灵活，但此时需要为每个数据构造他们各自对接口实现的虚表，这样可以依旧调用到属于该类型的实现。

虚表一般存储在堆上。Rust 下也有虚表的栈实现：
https://github.com/archshift/dynstack

2021-08-23

1

6

- 

Christian

1. 函数放在代码段中，通常是只读的，往只读段写入数据会触发保护异常。

2. 使用 enum：
```rust
enum Shape {
   Rectangle(Rectangle),
   Circle(Circle),
   Triangle(Triangle),
}
```

3. 定义一个 trait 并为三种结构都实现该 trait：
```rust
trait SomeInterface {
```

```
    fn area(&self) -> f64;
    fn circumference(&self) -> f64;
}

impl Rectangle for SomeInterface {
    fn area(&self) -> f64 {
        ...
    }

    fn circumference(&self) -> f64 {
        ...
    }
}

impl Circle for SomeInterface {
    ...
}

impl Triangle for SomeInterface {
    ...
}
```

作者回复: 正确！非常棒！

2021-08-23

4

43

- 

🔥神山｜雷神山

1. 有一个指向某个函数的指针，如果将其解引用成一个列表，然后往列表中插入一个元素，请问会发生什么？
对于强类型语言（如：rust），无法解引用成一个列表，rust会提示类型不匹配错误。
对于弱类型语言(如：python, javascript)，解引用成一个列表后，可以正常插入元素。
```javascript
let fn = function (title) { console.log(title); }
   fn("test");
   fn = [1, 2, 3, 4, 5, 6, 7, 8];
   fn.push(9);
   console.log(fn);
```
```rust
let func = |x| x;
   func(1);
   func = vec![1, 2, 3];
   println!(func);
```
2. 要构造一个数据结构 Shape，可以是 Rectangle、 Circle 或是 Triangle，这三种结构见如下代码。请问 Shape 类型该用什么数据结构实现？怎么实现？
```rust
use std::f64::consts;
trait Calculator {
   fn perimeter(&self) -> f64;
   fn area(&self) -> f64;
}

struct Rectangle {
   a: f64,
   b: f64,
}

struct Circle {
   r: f64,
}
```

```
struct Triangle {
    a: f64,
    b: f64,
    c: f64,
}

#[derive(Debug)]
enum EShape {
    Rectangle(f64, f64),
    Circle(f64),
    Triangle(f64, f64, f64),
}

#[derive(Debug)]
struct Shape {
    shape: EShape,
}
impl Shape {
    fn new(shape: EShape) -> Shape {
        Shape { shape }
    }
}

impl Calculator for Shape {
    fn perimeter(&self) -> f64 {
        match self.shape {
            EShape::Rectangle(a, b) => (a + b) * 2.0,
            EShape::Circle(r) => 2.0 * consts::PI * r,
            EShape::Triangle(a, b, c) => a + b + c,
        }
    }
    fn area(&self) -> f64 {
        match self.shape {
            EShape::Rectangle(a, b) => a * b,
            EShape::Circle(r) => consts::PI * r * r,
            EShape::Triangle(a, b, c) => {
                let p = (a + b + c) / 2.0;
                (p * (p - a) * (p - b) * (p - c)).sqrt()
            }
        }
    }
}

fn main() {
    let shape = Shape::new(EShape::Triangle(3.0, 4.0, 5.0));
    println!("shape:{:#?}", shape);
```

```
    println!("perimeter: {}", shape.perimeter());
    println!("area: {}", shape.area());
}
```

3. 对于上面的三种结构，如果我们要定义一个接口，可以计算周长和面积，怎么计算?
只需要将上述代码中的Rectangle，Circle，Triangle 三个结构体分别实现Calculator trait即可。
作者回复: 非常棒！

2021-08-24

3

11

- 太子长琴

  第一个题意不是特别理解
  后面的倾向于用 match，每个 struct 重复 impl 看着就觉得烦

```
pub enum ShapeEnum {
    Rectangle(f64, f64),
    Circle(f64),
    Triangle(f64, f64, f64),
}

#[derive(Debug)]
pub struct Shape {
    pub shape: ShapeEnum,
}

impl Shape {
    pub fn new(shape: ShapeEnum) -> Shape {
        Shape { shape: shape }
    }
}

pub trait Calculate {
    fn perimeter(&self) -> f64;
    fn area(&self) -> f64;
}

impl Calculate for Shape {
    fn perimeter(&self) -> f64 {
        match self.shape {
            ShapeEnum::Rectangle(a, b) => 2.0 * (a + b),
            ShapeEnum::Circle(r) => 2.0 * 3.14 * r,
            ShapeEnum::Triangle(a, b, c) => a + b + c,
        }
    }
    fn area(&self) -> f64 {
        match self.shape {
            ShapeEnum::Rectangle(a, b) => a * b,
            ShapeEnum::Circle(r) => 3.14 * r * r,
            ShapeEnum::Triangle(a, b, c) => {
                let p = (a + b + c) / 2.0;
                (p * (p - a) * (p - b) * (p - c)).sqrt()
            }
        }
```

```
    }
}
```

作者回复: 1 的题意可以看这个代码:

```C
#include "stdio.h"

void hello() {
    printf("Hello world!\n");
}

int main() {
    char buf[1024];
    void (* p)() = &hello;
    (*p)();
    int *p1 = (int *) p;
    p1[1] = 0xdeadbeef;
}
```

在 C 语言中，这种操作是允许的，但会造成内存访问越界，导致崩溃；在 Rust 中，编译器会拒绝这样的操作。

2/3 正确！

2021-08-24

1

5

- 徐洲更

这一篇是对编程语言的高度抽象呀，这一篇的知识完全可以应用到任何一门编程语言上。

作者回复: 嗯，前两篇都是每个程序员最好掌握的基础知识。

2021-08-24

4

- Scott

  我想问问B站油管视频里那个vscode直接在出错的那一行显示错误信息怎么弄的，我装了ra，但是出错信息还是显示在底部。

  作者回复: 对是 error lens。感谢 coer 的解答。https://marketplace.visualstudio.com/items?itemName=usernamehw.errorlens

  2021-08-23

  1

  4

- 虾

  请问一个关于虚表的问题。虚表是每个类有一份，还是每个对象有一份，还是每个胖指针有一份?

  作者回复: 好问题。这个在讲 trait 的那一课有讲到。虚表是每个 impl TraitA for TypeB {} 时就会编译出一份。比如 String 的 Debug 实现, String 的 Display 实现各有一份虚表，它们在编译时就生成并放在了二进制文件中（大多是 RODATA 段中）。

  所以虚表是每个 (Trait, Type) 一份。并且在编译时就生成好了。

  如果你感兴趣，可以在 playground 里运行这段代码（这是后面讲 trait 时使用的代码）：https://play.rust-lang.org/?version=stable&mode=debug&edition=2018&gist=89311eb50772982723a39b23874b20d6。限于篇幅，代码就不贴了。

  2021-09-04

  3

- Jason

  作为一个只写过脚本语言的前端，看虚表那部分基本上等于是在看天书，太菜了呜呜呜

  作者回复: 虚表你可以理解成一张指向若干个函数地址的表。这样在运行时，可以通过这张表找出要执行的函数，进而执行。比如 w.write()，如果 w 是一个类型被抹去的引用，指向了一块地址，此时 w 如何能够执行到 write() 方法？只能通过运行时构造的虚表来解决。

  2021-08-30

  1

  3

- kimi

  1. C 语言是可以把一个函数指针解引用为一个列表的，往列表中插入一个元素会报错，这是因为函数是代码，在内存中通常会存放在只读区域，对这部分内存写会报错；
  2. 定义一个 Shape 接口；
  3. 在 Shape 接口中定义一个计算周长和面积的方案；

  还不会写 Rust 代码，只能用 Java 的思路来考虑，希望学完本课程能用 Rust 的思路来思考和编程；
  作者回复: 正确！

  2021-08-23

  3

- 你好梦梦 

我用简单c++代码描述一下老师这个问题，不知道是不是这个意思：
typedef void (*funptr)(int,int);
void funtest(int,int){}
funptr p = funtest;
int* parr = (int*)p; //转成数组指针
parr[0] = 1; parr[1]=2
如果是这样的话，parr[0]、parr[1]会引发未知错误，因为他会覆盖原有内存区域的值。
对于Java、C#这些语言，在他们提供了Lambda、委托这种函数，其实是无法做强制转换的，
Rust还在学习，但我感觉应该不行，如果行的话，他就不强调自己安全啦。

作者回复: 对。如果要覆盖的话，程序会崩溃，因为 TEXT 段不可泄，而且在这个过程中，可能会出现 aligment 错误。我也用 C 写了个例子：

```C
#include "stdio.h"

void hello() {
   printf("Hello world!\n");
}

int main() {
   char buf[1024];
   void (* p)() = &hello;
   (*p)();
   int *p1 = (int *) p;
```

```
    p1[1] = 0xdeadbeef;
}
```

2021-08-23

2

- return

  老师， 请教一下，
  强类型语言 和 弱类型语言的 定义是什么，区别是什么。

  作者回复: 可以看我之前的回复。强类型弱类型的定义并不算太明确，我接受的定义是类型是否会在使用过程中发生隐式转换，比如 char 如果和 int 相加被转换成 int，和 string 相加被转换成 string。这样的转换往往会超出开发者的预期，导致逻辑 bug。Rust 甚至不允许 u8 + u32 这样的操作，而是需要你使用显式的类型转换统一类型后再处理，所以是强类型。

  2021-08-23

  5

  2

- MILI

  1 没找到呀
  2.3.

```
fn main() {
    struct Rectangle { a: f64, b: f64 }
    struct Circle { r: f64 }
    struct Triangle { a: f64, b: f64, c: f64 }
    enum Shape {
        Rectangle,
        Circle,
        Triangle,
    }
    pub trait Graphics {
        fn perimeter(&self) -> f64;
        fn area(&self) -> f64;
    }
    impl Graphics for Rectangle {
        fn perimeter(&self) -> f64 {
            (self.a + self.b) * 2 as f64
        }

        fn area(&self) -> f64 {
            self.a * self.b
        }
    }
    impl Graphics for Circle {
        fn perimeter(&self) -> f64 {
            2 * 3.14 * self.r
        }

        fn area(&self) -> f64 {
            3.14 * self.r * self.r
        }
    }
    impl Graphics for Triangle {
        fn perimeter(&self) -> f64 {
            self.a + self.b + self.c
        }

        fn area(&self) -> f64 {
            let p = self.a + self.b + self.c;
            p * (p - self.a)(p - self.b)(p - self.c).sqrt()
        }
```

```
    }
}
```

作者回复: 1. 在 C 语言中，这种操作是允许的，但会造成内存访问越界，导致崩溃；在 Rust 中，编译器会拒绝这样的操作。

2/3: 非常棒。enum 的定义可以这样：

```rust
enum Shape {
    Rec(Rectangle),
    Cir(Circle),
    Tri(Triangle),
}
```

我也写了个参考版本，见 https://play.rust-lang.org/?version=stable&mode=debug&edition=2018&gist=9bde037aa71319fc88852cdfecc1ac35

2021-08-23

2

- TheLudlows 

习惯了写Java那种自动提示，Rust一碰到宏就没法提示就特别慌张，不知道其他人有没有这种感觉

作者回复: 宏的语法提示很难做，基本上支持宏的语言，自动提示都做得不好，尤其是像 tokio::select! 宏这样子里面还能有复杂的代码逻辑。

2021-11-05

1

- 

脑洞太大补不上

1. 有一个指向某个函数的指针，如果将其解引用成一个列表，然后往列表中插入一个元素，请问会发生什么？（对比不同语言，看看这种操作是否允许，如果允许会发生什么）

- C
  编译器允许把指针类型转换成列表, 这一步不会引起错误
  但在调用插入列表元素函数时应该会出错误, 因为函数指针是在只读段, 真的要去往内存里去写东西的时候, 大概率那个计算出来的被写地址也同样会落在只读段, 写的时候会报错..
  不过这如果不是一个函数指针而是别的什么指针, 有可能是不会报错的, 这就更恐怖了..

- Java / C#
  应该没有函数指针这个东西..
  不过假如是别的什么类型, 强制类型转换到列表这一步会在运行时报错

- Python / Javascript / Elixir
  不存在类型转换这一步, 直接调用列表插入元素时会在报运行时报错

- Rust
  目前还确定, 猜测应该也是类型转换时会在运行时报错

2, 3 就不重复了
作者回复: 👍 非常棒！

2021-10-19

1

- 

Marvichov 

virtual table check 神器 https://godbolt.org/

作者回复: 嗯，有意思，不过还是 gdb 更趁手。

2021-09-11

1

1

- 水口行舟入她心

半路出家的Java开发，来学一学面向未来的语言，同时在老师的带领下，夯实基础，做更优秀的开发者

作者回复: 加油！

2021-09-10

1

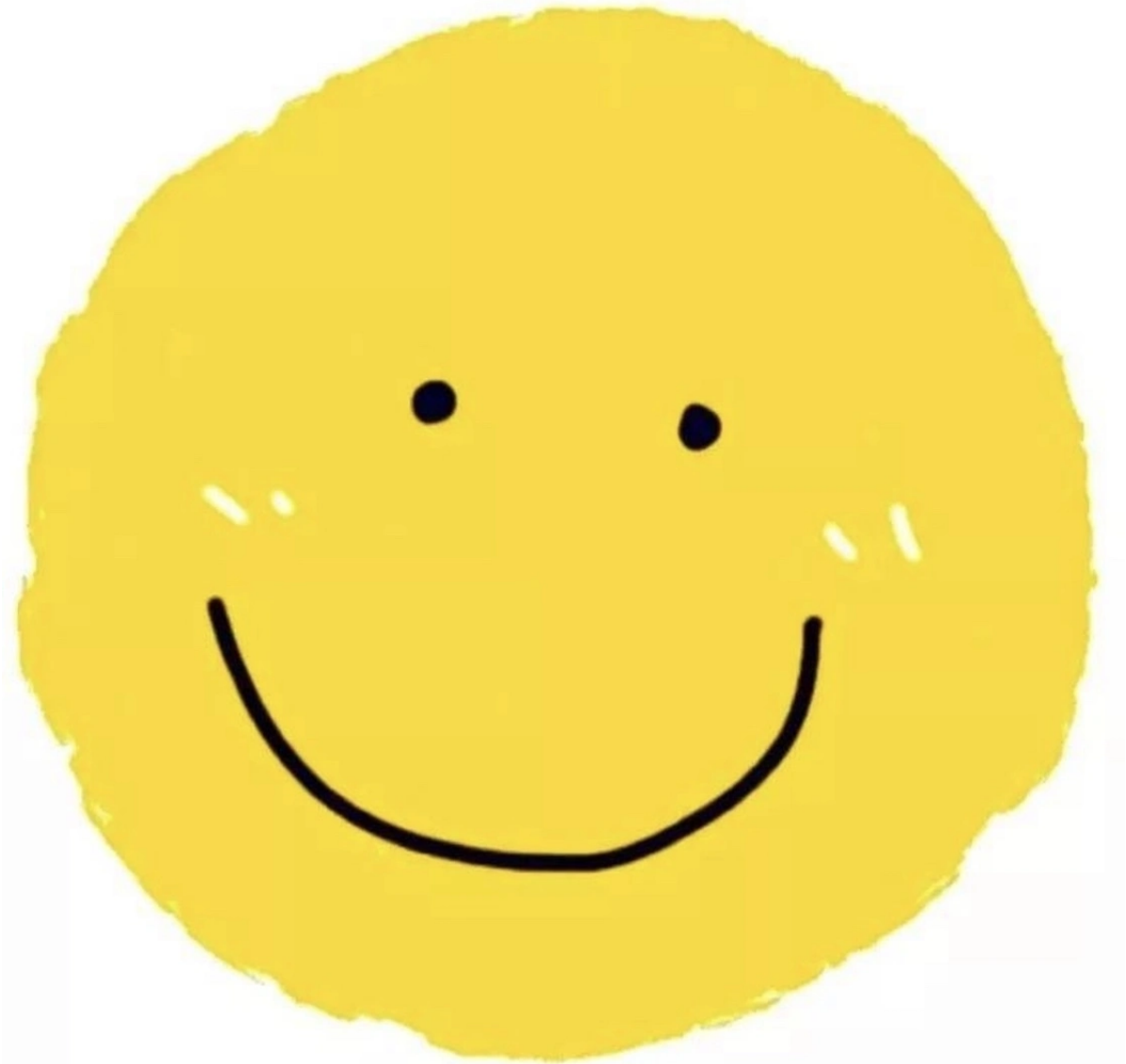
- Fan

  这是站在另外的角度看编程语言。

  作者回复: 嗯，我们先把一些基本概念扫一下，可以更好地学习后续的内容。

  2021-08-24

1

- 

只吃无籽西瓜

这一章的课后问题不会，希望有人能够解答。

编辑回复: 嗯如果想过之后还是有点不太明白，可以先在留言区看看各位同学的思路，也可以直接问下老师具体是哪里不太明白。

另外，每一讲的思考题，会挑重点普遍的问题，把参考思路都整理到后续加餐中的~

2021-08-23

2

1

- 小样

  “如果你来自于弱类型语言，如 C / Python / JavaScript”，c是弱类型语言？

  作者回复: C 是典型的弱类型。我们看代码，这里 char 被隐式转换为：

  ```C
  #include <stdio.h>

  int add_numbers(int a, int b) {
    int result = a + b;
    return result;
  }

  int main() {
    char c = "42";
    int n = 42;

    // 为什么说 C 是 weakly typed
    int result = add_numbers(c, n);
    printf("%d\n", result);

    return 0;
  }
  ```

  2021-08-23

  4

  1

收起评论