Rust 学习笔记

目录

0 前言

在 Rust 中, 使用 `//` 进行单行注释, 使用 `/*...*/` 进行多行注释. 其中 `//` 将注释到行尾, 而 `/*...*/` 则从 `/*` 开始到 `*/` 结束, 可以跨越多行.

> Rust 官方网站
> Rust 开发环境搭建
> Rust 官方文档 | Rust 文档中文译本
> Rust 在线练习
> Rust 语言参考
> 通过例子学习 Rust (Rust By Example)
> Rust 小练习(Rustlings)
> Rust 语言标准库
> Rust 中文 Wiki
> Rust 在哔哩哔哩上也有视频教程, 推荐此视频或此视频

1 变量与常见数据类型

1. 在 Rust 中, 使用 `let` 关键字声明变量.
2. Rust 支持类型推导, 但也可以显式指定变量类型: `let x: i32 = 5; //显式指定 x 的类型为 i32.`
3. 变量命名使用蛇形命名法, 而枚举与结构提使用帕斯卡命名法; 若变量没有使用则前置下划线以消除警告.
4. 强制类型转换(Casting a value to a different type): `let a = 3.1; let b = a as i32;`
5. 打印变量:

```
    println!("val: {}", x);
    println!("val: {x}");
```

1.1 Rust 的变量默认不可变

不可变性是 Rust 实现其可靠性与安全性目标的关键.

1.1.1 使用 `mut` 关键字使变量变得可变

```
fn main() {
    let mut y = 5; //可变变量
    y = 20; //合法修改
}
```

1.1.2 遮蔽(Shadow)

Rust 允许隐藏一个变量, 这意味着可以声明一个与现有变量同名的新变量, 并隐藏前一个变量.

- 可以改变值
- 可以改变类型
- 可以改变可变性

```
fn variable() {
    //不可变与命名
    let count_a = 15; //自动推导变量类型为 i32
    let count_b: i16 = 15; //显式指定变量类型为 i16
    /*count_a = 2;*/ //尝试改变不可变变量的值,会报错
    let mut count_c: i32 = 2147483647; //声明可变
    count_c = -2147483648;
    //遮蔽(Shadow)
    let x = 1;
    {
        let x: bool = false; //覆盖x(数值)
        {
            let x = "Shadowing"; //遮蔽x(布尔值)
            println!("{}", x); //打印以验证
        } //x(字符串)作用域结束(释放)
        println!("{}", x); //打印以验证
    } //x(布尔值)作用域结束(释放)
    println!("{}", x); //打印以验证
    let mut y = 4;
}
```

1.2 常量(const)与静态变量(static)

1.2.1 常量(const)

- 常量使用 `const` 关键字声明, 必须指定类型和值.
- 常量的值直接嵌到底层代码中, 而不是简单的字符替换.
- 常量名必须全部大写, 使用下划线分隔单词.
- 常量的声明只在作用域内有效.

1.2.2 静态变量(static)

- 与常量(const)不同, 静态变量实在运行时分配内存的.
- 并非不可变, 可以使用 `unsafe` 关键字修改静态变量(前提是使用 `mut` 声明其可变).
- 静态变量的生命周期整个程序的运行时间.

示例

```
static DAY_IN_SOLAR_YEAR_APPROXIMATE: f64 = 365.2422; //静态变量
static  mut DAY_IN_YAER: u16 = 365; //可变静态变量

fn c2_const_and_static() {
    const SECONDS_IN_WEEK: u32 = 60 * 60 * 24; //使用`const`关键字声明常量
    /* SECONDS_IN_WEEK = 5; */ //尝试改变常量的值,会报错
    const SECONDS_IN_DAY: u32 = 86_400; //常量(另一种写法)
    {
        const A: u8 = 255;
    }
    /*println!("{A}");*/ //尝试打印不在作用域内的常量A,会报错
    const SECONDS_IN_SOLAR_YEAR: u64 = 60 * 60 * 24 * 365 + 60 * 60 * 5 + 60 * 48 + 46;

    /*unsafe {
        DAY_IN_YAER = 366;
        println!("{DAY_IN_YAER}"); //打印可变静态变量
```

```
    }*/ //使用`unsafe`修改可变静态变量

    println!("One week has {SECONDS_IN_WEEK} seconds |\
    One day has {SECONDS_IN_DAY} seconds |\
    One solar year has {SECONDS_IN_SOLAR_YEAR} seconds");

    println!("One solar year has {DAY_IN_SOLAR_YEAR_APPROXIMATE} days");
}
```

1.3 数据类型

1.3.1 整数类型(整型)

1. 整数类型有符号与无符号两种, 分别使用 `i` 和 `u` 前缀.
   - 有符号整数: `i8` , `i16` , `i32` , `i64` , `i128` , `isize`
   - 无符号整数: `u8` , `u16` , `u32` , `u64` , `u128` , `usize`
2. 有符号整数不支持负数, 无符号整数支持负数.
   - 对于有符号整数 `i<x>` , 其支持范围为$[-2^{x-1}, 2^{x-1}-1]$.
   - 对于无符号整数 `u<x>` , 其支持范围为$[0, 2^{x}-1]$.
3. 整数类型默认为 `i32` .
4. 整数类型的大小与平台无关, 但 `isize` 与 `usize` 的大小与平台相关.

表1.1 不同整数类型支持的范围

类型	范围	说明
i8	$[-2^{7}, 2^{7}-1]$	8 位有符号整数, 支持负数
i16	$[-2^{15}, 2^{15}-1]$	16 位有符号整数, 支持负数
i32	$[-2^{31}, 2^{31}-1]$	32 位有符号整数, 支持负数
i64	$[-2^{63}, 2^{63}-1]$	64 位有符号整数, 支持负数
i128	$[-2^{127}, 2^{127}-1]$	128 位有符号整数, 支持负数
u8	$[0, 2^{8}-1]$	8 位无符号整数, 不支持负数
u16	$[0, 2^{16}-1]$	16 位无符号整数, 不支持负数
u32	$[0, 2^{32}-1]$	32 位无符号整数, 不支持负数
u64	$[0, 2^{64}-1]$	64 位无符号整数, 不支持负数
u128	$[0, 2^{128}-1]$	128 位无符号整数, 不支持负数

- 可以使用任何一种形式编写数字字面值，同时也允许使用 `_` 做为分隔符以方便读数，例如 `1_000` ， 它的值与你指定的 1000 相同。

表1.2 整数类型的字面值

字面值	示例	实际数值
十进制	`1_000`	1000

字面值	示例	实际数值
十六进制	`0xE3`	227
八进制	`0o77`	63
二进制	`0b1111_0000`	240

> *整数溢出*
> 整数溢出是指在计算中，结果超出了整数类型的表示范围. 例如: 在使用 `u8` 类型时, 计算 `255 + 1` 会导致溢出.
> 在 Rust 中, 整数溢出会导致程序 *panic*, 这是一种运行时错误. 使用 `--release` flag 在 release 模式中构建时， Rust 不会检测会导致 panic 的整型溢出. 相反发生整型溢出时，Rust 会进行一种被称为二进制补码 wrapping 的操作。 简而言之，比此类型能容纳最大值还大的值会回绕到最小值, 值 256 变成 0，值 257 变成 1，依此类推. 程序不会 panic 不过变量可能也不会是你所期望的值.依赖整型溢出 wrapping 的行为被认为是一种错误。 为了显式地处理溢出的可能性，可以使用这几类标准库提供的原始数字类型方法：
>
> - 所有模式下都可以使用 wrapping_* 方法进行 wrapping，如 wrapping_add
> - 如果 checked_* 方法出现溢出，则返回 None值
> - 用 overflowing_* 方法返回值和一个布尔值，表示是否出现溢出
> - 用 saturating_* 方法在值的最小值或最大值处进行饱和处理

1.3.2 浮点数类型(浮点型)

Rust 支持两种浮点数类型: `f32` 与 `f64` , 默认为 `f64` ， `f64` 的精度更高.

1.3.3 布尔类型(bool)

布尔类型只有两个值: `true` 与 `false` , 其字面值为 `true` 与 `false` . 通常在 `if` 语句中使用.

1.3.4 字符类型(char)

Rust 的字符类型是 `char` , 它表示单个 Unicode 字符, 而不是单个字节. `char` 类型的大小为 4 字节,

```
//示例：数据类型
fn data_type() {
    let a = 5; //整数，默认i32
    let b: i32 = -5; //整数，显式指定类型为i32
    let c: u32 = 5; //整数，显式指定类型为u32，仅支持自然数
    let d = 1.14; //浮点数，默认f64
    let e: f32 = 3.14; //浮点数，显式指定类型为f32
    let t = true; //布尔值
    let f: bool = false; //布尔值
    let character = '😈'; //字符类型，4字节
}
```

1.3.5 数值运算

Rust 中的所有数字类型都支持基本数学运算：加法、减法、乘法、除法和取余. 整数除法会 向零舍入到最接近的整数.

```
fn calculate() {
    let a: i32 = 5;
    let b: u64 = 455432378;
```

```
    let c: f64 = 3.14;
    let d: f32 = 2.5;
    let e: i32 = 10;
    let f: i32 = 17;
    let add = a + b; //加法
    let sub = b - d; //减法
    let mul = d * c; //乘法
    let div = e / a; //除法
    let rem = e % a; //取余(能整除, 结果为0)
    let rems = f % a; //取余(不能整除)
}
```

1.4 复合类型

复合类型(Compound types)可以将多个值组合成一个类型. 复合类型有两种: 元组(tuple)与数组(array).

- 元组可以包含不同类型的数据, 而数组只能包含相同类型的数据.
- 元组和数组的大小都是固定的, 不能动态改变.

1.4.1 数组(Array)

- 数组的大小是固定的, 只能由相同的数据构成.
- 对于不会改变元素个数的情况, 更适合使用数组.

定义一个数组

数组的元素写在方括号内, 用逗号分隔: `[1, 2, 3, 4]` , 也可以显式指定数组的类型与数量: `[i32; 4]` .

访问数组元素

对于任意数组 `<array>` , 可以使用下标访问元素, 下标从 0 开始: `<array>.[<index>]` , 其中 `<index>` 是下标, 即要访问数组元素的第几位(从0开始). 例如: `a.[1]` 会得到数组 `a` 的第二个元素.

获取数组长度

对于任意数组 `<array>` , 可以使用 `len()` 方法获取数组的长度: `<array>.len()` , 例如: `a.len()` 会得到数组 `a` 的长度.

```
// 示例: 数组
fn array() {
    let a = [1, 2, 3, 4, 5]; //声明数组, 使用的是最基本的方法
    let b: [i32/*指定类型为i32*/; 6 /*指定数组的长度为6*/] = [0, 2, 4, 6, 8, 10];
    let months = ["January", "February", "March", "April", "May", "June", "July",
        "August", "September", "November", "October", "December"];
    let a_first = a[0]; //访问数组a的第一个元素
    let b_second = b[1]; //访问数组b的第二个元素
    let length = months.len(); //获取数组months的长度
}
```

1.4.2 元组(Tuple)

元组类似于数组, 长度固定, 但可以包含不同类型的数据. 元组的数据用圆括号包围, 用逗号分隔: `(1, 2, 3)` , 也可以显式指定元组的类型与数量: `(i32, i32, i32)` . 没有任何数据的元组称作单元元组, 写作 `()` , 表示空值或空的返回类型. 如果表达式不返回任何其他值, 则会隐式返回单元值.

访问元组元素

对于任意元组 `<tuple>` , 可以使用下标访问元素, 下标从 0 开始: `<tuple>.<index>` , 其中 `<index>` 是下标, 即要访问元组元素的第几位(从0开始).

```
// 示例: 元组
fn tuple() {
    let a = (1, 2, 3); //声明元组
    let b: (u8, i32, f64) = (7 ,-9, 33.2676); //声明元组, 显式指定类型
    let c = (1, 2.0, "Hello"); //声明元组, 不同类型
    let a_first = a.0; //访问元组a的第一个元素
    let b_second = b.1; //访问元组b的第二个元素
    let c_third = c.2; //访问元组c的第三个元素
}
```

> 对于长度可变的多种数据, 应当使用Vector.(见后文)

2 所有权与 `String` 类型

2.1 所有权

> 1. 对于 C/C++ 等语言, 内存管理由程序员手动控制.
> 2. 而 Python、Java 等语言, 内存管理由垃圾回收器(Garbage Collector)自动控制.
> 3. Rust 采用了所有权(Ownership)机制, 使得内存管理更高效且安全.

> **C/C++的内存错误**
>
> - 内存泄漏(Memory Leak): 程序分配了内存但没有释放, 导致内存无法使用. > ``` int* ptr = new int[10]; //分配内存 //忘记释放内存 //delete ptr;
>
> - 悬垂指针(Dangling Pointer): 指向已释放内存的指针, 访问时会导致错误. > ``` int* ptr = new int[10]; //分配内存 delete ptr; //释放内存 //ptr 仍然指向已释放内存, 即悬垂指针
>
> - 重复释放: 多次释放同一块内存, 导致错误. > ``` int* ptr = new int[10]; //分配内存 delete ptr; //释放内存 delete ptr; //重复释放, 导致错误
>
> - 数组越界(Array Out of Bounds): 访问数组越界的元素, 导致错误. > ``` int arr[10]; arr[10] = 5; //访问越界, 导致错误

- 野指针(Wild Pointer): 指向未分配内存的指针, 访问时会导致错误. > ``` int* ptr; //未初始化指针 *ptr = 10; //访问未分配内存, 导致错误

- 使用已释放的内存(Use After Free): 访问已释放的内存, 导致错误. > ``` int* ptr = new int[10]; //分配内存 delete ptr; //释放内存 *ptr = 10; //访问已释放内存, 导致错误

Rust 最重要的特性之一是**所有权(Ownership)**, 它是 Rust 的内存管理机制. 所有权系统使 Rust 独树一帜, 使其在内存安全性和性能方面都表现出色. 编译器在编译时会根据一系列的规则进行检查. 如果违反了任何这些规则, 程序都不能编译. 在运行时, 所有权系统的任何功能都不会减慢程序.
所有权系统有三大规则:

1. 每个值都有一个所有者(Owner).
2. 每个值只能有一个所有者.
3. 当所有者离开作用域时, 值会被丢弃(Drop).

2.1.1 String 类型

- Rust 中的字符串有两种类型: `String` 与 `&str` .

`&str` 是字符串字面值, 是不可变的. 为此, Rust 提供了另一种字符串类型 `String` , 它是可变的. 因为 `String` 是可变的, 所以它在堆上分配内存. 可以通过 `String::from()` 方法创建一个 `String` 类型的字符串:

```
let s = String::from("Hello, world!");
```

可以通过 `push_str()` 方法向 `String` 类型的字符串添加内容:

```
fn string() {
    let mut s = String::from("Hello");
    s.push_str(", world!"); //添加内容
    println!("{s}"); //将会打印 "Hello, world!"
}
```

2.1.2 数据的移动(Move)

这是普通变量的例子:

```
let x = 5;
let y = x;
println!("x: {x}, y: {y}"); //打印 x 和 y
```

如上所示, 先给 `x` 赋值 `5` , 然后将 `x` 的值赋给 `y` . 这时, `x` 和 `y` 都是 `5` , 但它们是两个不同的变量, 互不影响, `x` 与 `y` 都是可用的.
但对于 `String` 类型来说, 情况就不同了:

```
//错误演示
let s1 = String::from("Hello");
let s2 = s1;
println!("s1: {s1}, s2: {s2}"); //打印 s1 和 s2
```

上面的代码会报错, 因为 `s1` 的所有权已经转移给了 `s2` , `s1` 不再有效. 这叫做 Rust 中的**移动(Move)**.

2.1.3 克隆(Cloning)

如果想要同时保留原数据与移动后的数据, 可以使用 `clone()` 方法.

```
fn main() {
    let s1 = String::from("Hello");
    let s2 = s1.clone(); //克隆
    println!("s1: {s1}, s2: {s2}"); //打印 s1 和 s2
}
```

上面的代码可以正常运行, 因为 `s1` 和 `s2` 都是有效的.

2.1.4 拷贝

对于简单的数据类型, Rust 会自动进行**拷贝**, 而不是移动.
可以进行拷贝的数据类型有:

- 所有整数类型
- 所有浮点数类型
- 布尔类型
- 字符类型 `char`
- 元组, 前提是其所有元素都符合以上可以进行拷贝的数据类型.
  比如说 `(i32, i32)` 可以进行拷贝, 但 `(i32, String)` 不可以进行拷贝.

2.1.5 所有权在函数及返回值中的作用

所有权与函数

```
fn main() {
    let s = String::from("hello");
    takes_ownership(s); //s的所有权转移到函数中
    //此时s失效
    let x = 5;
    copy(x); //x的值被拷贝到函数中
    //此时x仍然有效
} //此时, x与s都被释放, 不过s已经失效了.

fn takes_ownership(strings: String) { //s的所有权转移到函数中
    println!("s: {strings}");
}

fn copy(var: i32) {
    println!("var: {var}");
}
```

所有权与返回值

```
fn main() {
    let s1 = back();
    let s2 = String::from("Hello");
    let s3 = give_and_back(s2); //s2的所有权转移到函数中
}

fn back() -> String {
    let a_string = String::from("Hello");
    a_string //返回值
}

fn give_and_back(next_string: String) -> String {
    next_string
}
```

还能通过元组返回多个值:

```
fn main() {
    let s1 = String::from("Hello");
    let (s, length) = length(s1); //s的所有权转移到函数中
}

fn length(s: String) -> (String, usize) {
    let length = s.len();
    (s, length) //返回值
}
```

2.2 引用与借用

`&` 符号代表**引用(Reference)**, 它允许你使用一个值而不获取所有权.

```
fn main() {
    let s1 = String::from("Hello");
    let len = calculate_length(&s1/*对s1进行引用*/); //传递引用
}
fn calculate_length(s: &String) -> usize { //s是String类型的引用
    s.len() //返回字符串的长度
}
```

此处通过 `&s1` 传递了 `s1` 的引用, 而不是 `s1` 的所有权.
普通引用函数的值是不可以被修改的:

```
fn main() {
    let mut s = String::from("Hello");

    change(&s); //传递不可变引用
}
fn change(s: &String) {
    s.push_str(", world!"); //错误, s是不可变引用
}
```

于是, 代码报错:

```
error[E0596]: cannot borrow `*s` as mutable, as it is behind a `&` reference
 --> src\main.rs:7:5
  |
7 |     s.push_str(", world!"); //错误, s是不可变引用
  |     ^ `s` is a `&` reference, so the data it refers to cannot be borrowed as mutable
  |
help: consider changing this to be a mutable reference
  |
6 | fn change(s: &mut String) {
  |               +++
```

2.2.1 可变引用

在上文我们提到, 引用是不可变的, 要使引用可变, 可以使用 `&mut` 来创建可变引用.

```
fn main() {
    let mut s = String::from("Hello");
    change(&mut s); //传递可变引用
}
fn change(s: &mut String) {
    s.push_str(", world!"); //合法, s是可变引用
}
```

可变引用的限制

不过, 可变引用只能够有一个, 也就是说, 在同一作用域内, 不能同时存在多个可变引用.

```
fn main() {
    let mut s = String::from("Hello");
    let s1 = &mut s;
    let s2 = &mut s; //错误, s1与s2都是对s的引用
    println!("{} | {}", s1, s2);
}
```

于是, 代码报错:

```
error[E0499]: cannot borrow `s` as mutable more than once at a time
 --> src\main.rs:4:14
  |
3 |     let s1 = &mut s;
  |              ------ first mutable borrow occurs here
4 |     let s2 = &mut s; //错误, s1与s2都是对s的引用
  |              ^^^^^^ second mutable borrow occurs here
5 |     println!("{} | {}", s1, s2);
  |                          -- first borrow later used here
```

- `error[E0499]` 告诉我们, 不能同时进行超过一次可变引用.
- 这样做避免了数据竞争(Data Race), 也就是两个线程同时访问同一数据, 可能导致数据不一致的问题. 解决方法也很简单, 只需要新建一个作用域即可.

```
fn main(){
    let mut s = String::from("Hello");
    {
```

```
        let s1 = &mut s; //可变引用
        println!("{}", s1);
    } //s1的作用域结束，可变引用被释放
    let s2 = &mut s; //合法，因为s1的作用域已经结束
    println!("{}", s2);
}
```

也不能同时存在可变引用与不可变引用:

```
fn main() {
    let mut s = String::from("HELLO");
    let s1 = &s; //不可变引用
    let s2 = &mut s; //错误，s1与s2都是对s的引用
    println!("{} | {}", s1, s2)
}
```

于是, 代码报错:

```
error[E0502]: cannot borrow `s` as mutable because it is also borrowed as immutable
 ⟶ src\main.rs:4:14
  |
3 |     let s1 = &s; //不可变引用
  |              -- immutable borrow occurs here
4 |     let s2 = &mut s; //错误，s1与s2都是对s的引用
  |              ^^^^^^ mutable borrow occurs here
5 |     println!("{} | {}", s1, s2)
  |                         -- immutable borrow later used here
```

- `error[E0502]` 告诉我们, 不能同时进行不可变引用与可变引用.
- 这仍然是为了避免数据竞争的问题.

2.2.2 悬垂引用(Dangling Reference)

在具有指针的语言中, 很容易通过释放内存时保留指向它的指针而错误地生成一个**悬垂指针 (dangling pointer)**, 所谓悬垂指针是其指向的内存可能已经被分配给其它持有者.
在 Rust 中, 类似于悬垂指针的引用被称为**悬垂引用(dangling reference)**. 为了避免悬垂引用的情况出现, 编译器需确保数据不会在其引用之前离开作用域. 以下是一个悬垂引用的例子:

```
fn main() {
    let reference_to_nothing = dangle();
}
fn dangle() -> &String {
    let s = String::from("hello");
    &s
}
```

于是, 代码报错:

```
error[E0106]: missing lifetime specifier
 ⟶ src\main.rs:4:16
  |
4 | fn dangle() -> &String {
  |                ^ expected named lifetime parameter
```

```
  |
  = help: this function's return type contains a borrowed value, but there is no value for it to be borrowed fr
help: consider using the `'static` lifetime, but this is uncommon unless you're returning a borrowed value from
  |
4 | fn dangle() -> &'static String {
  |                +++++++
help: instead, you are more likely to want to return an owned value
  |
4 - fn dangle() -> &String {
4 + fn dangle() -> String {
  |
```

- `error[E0106]` 告诉我们, 需要一个生命周期参数.
- `help` 提示我们, 这个函数的返回值包含一个借用的值, 但没有值可以借用.

报错原因是因为 `s` 的作用域在 `dangle()` 函数结束时就结束了, 而我们仍然尝试返回 `s` 的引用. 这将导致悬垂引用, 因为 `s` 已经被释放了. 解决方法为直接返回 `s` 的值, 而不是引用:

```rust
fn main() {
    let reference_to_nothing = dangle();
}
fn dangle() -> &String {
    let s = String::from("hello");
    s
}
```

代码正常运行.

总而言之, 引用必须满足以下条件:

1. 在同一时刻内:
    i. **只能**有一个可变引用.
    或
    ii. **只能**有多个不可变引用.
2. 引用必须始终有效.

2.3 Slice

Slice 是对数组或字符串的一部分的引用, 它允许你访问数组或字符串的一部分而不获取所有权. 可以通过 `&<String>[<start_index>..<end_index>]` 的方式创建一个 Slice. 其中 `<String>` 是需要进行引用的字符串或数组, `<start_index>` 是 Slice 的起始索引, `<end_index>` 是 Slice 的结束索引(从0开始).

2.3.1 字符串 Slice

例如, 要把 `Hello, world!` 中的2个单词分别提取出来, 可以使用以下代码:

```rust
fn main() {
    let s = String::from("Hello, world!");
    let first_word = &s[0..5]; //提取前5个字符
    let second_word = &s[7..12]; //提取后5个字符
    println!("{slice} | {slice2}");
}
```

以下的方法适用于寻找任意单词:

```
fn first_word(s: &String) -> &str {
    let bytes = s.as_bytes();
    for (i, &item) in bytes.iter().enumerate() {
        if item == b' ' {
            return &s[0..i]; //返回第一个单词
        }
    }
    &s[..] //返回整个字符串
}
```

2.3.2 数组 Slice

实际上, `slice` 还支持数组. 例如:

```
let a = [1, 2, 3, 4, 5];
let slice = &a[0..3]; //提取前3个元素
assert_eq!(&[1, 2, 3], slice); //验证
```

3 结构体与枚举

3.1 结构体(Struct)

结构体(Struct)是 Rust 中一种自定义数据类型, 用于将多个相关的数据组合在一起.

3.1.1 定义结构体并实例化

结构体的定义使用 `struct` 关键字, 并提供名字. 接着, 使用 `{}` 定义每一部分的数据及类型, 我们称其为字段(Field).其格式如下:

```
struct <StructName> {
    <Name1>: <Type1>,
    <Name2>: <Type2>,
    ...
}
```

定义结构体之后, 需要对结构体进行实例化. 对结构体进行实例化, 需要以结构体名字开头, 然后用 `{}` 包围每一部分的数据, 并在其中以 `<key>: <value>` 的格式定义每一部分的数据.其格式如下:

```
<StructName> {
    <key1>: <value1>,
    <key2>: <value2>,
    ...
}
```

定义并实例化一个结构体的例子:

```
// 示例：结构体
struct User {
    username: String,
    userid: u32,
    email: String,
    sign_up_date: String,
    active: bool,
} //定义结构体
fn main() {
    let user1 = User {
        username: "UserA".to_string(),
        userid: 1,
        email: "someone@example.com".to_string(),
        sign_up_date: "2023-10-01".to_string(),
        active: true,
    }; //实例化结构体
}
```

获取结构体字段的值，需要使用 `.` 符号: `<name>.<field>` ，其中 `<name>` 是结构体的名字，`<field>` 是字段的名字.例:

```
// 示例：结构体
struct User {
    username: String,
    userid: u32,
    email: String,
    sign_up_date: String,
    active: bool,
} //定义结构体
fn main() {
    let user1 = User {
        username: "UserA".to_string(),
        userid: 1,
        email: "someone@example.com".to_string(),
        sign_up_date: "2023-10-01".to_string(),
        active: true,
    }; //实例化结构体
    user1.userid = 4; //获取结构体字段的值
}
```

我们还可以用函数来实例化结构体，并隐式返回实例,例如:

```
struct User {
    username: String,
    userid: u32,
    email: String,
    sign_up_date: String,
    active: bool,
} //定义结构体
fn set_user(
    name: String,
    id: u32,
    email: String,
    date: String,
)
    -> User {
     User {
        username: name,
        userid: id,
        email: email,
        sign_up_date: date,
```

```
        active: true,
    }
}
```

这样我们就可以通过函数来实例化结构体了.

不过, 有些变量的名称与结构体字段的名称相同, 这时可以使用简写语法来简化代码:

```
struct TrainTicket {
    passenger_name: String,
    price: f32,
    from: String,
    arrive_station: String,
    date: String,
    train_number: String,
    carriage_number: u8,
    seat_number: String,
    need_hongkong_macau_pass: bool,
}
fn ticket(
    passenger_name: String,
    price: f32,
    from: String,
    arrive_station: String,
    date: String,
    train_number: String,
    carriage_number: u8,
    seat_number: String,
    pass: bool,
) -> TrainTicket {
    TrainTicket {
        passenger_name,
        price,
        from,
        arrive_station,
        date,
        train_number,
        carriage_number,
        seat_number,
        need_hongkong_macau_pass: pass,
    }
}
fn main() {
    let ticket1 = ticket(
        "Li Hua".to_string(),
        100.0,
        "Beijing West Railway Station".to_string(),
        "Hong Kong West Kowlon Station".to_string(),
        "2025-01-01".to_string(),
        "G79".to_string(),
        6,
        "3C".to_string(),
        true,
    ); //实例化结构体
    println!("Passenger: {}", ticket1.passenger_name);
    println!("Price: {}", ticket1.price);
}
```

这样, 当变量名与字段名重复的时候, 我们便不需要重复写变量名了.

3.1.2 结构体更新语法

结构体更新语法允许我们只修改一个旧的结构体的某些字段, 而不需要重新定义所有字段.
这是不使用结构体更新语法的示例:

```
struct User {
    username: String,
    userid: u32,
    email: String,
    sign_up_date: String,
    active: bool,
}
fn main() {
    let user1 = User {
        username: String::from("User1"),
        userid: 1,
        email: String::from("UserA@example.com"),
        sign_up_date: String::from("2023-10-01"),
        active: true,
    };
    let user2 = User {
        username: user1.username,
        userid: 2,
        email: String::from("another@example.com"),
        sign_up_date: user1.sign_up_date,
        active: user1.active,
    };
}
```

这是使用了结构体更新语法的示例:

```
struct User {
    username: String,
    userid: u32,
    email: String,
    sign_up_date: String,
    active: bool,
}
fn main() {
    let user1 = User {
        username: String::from("User1"),
        userid: 1,
        email: String::from("UserA@example.com"),
        sign_up_date: String::from("2023-10-01"),
        active: true,
    };
    let user2 = User {
        userid: 2,
        email: String::from("another@example.com"),
        ..user1 //使用结构体更新语法
    };
}
```

可以发现, 结构体更新语法节省了许多代码. 不过 `..user1` 必须放在最后.

需要注意, 结构体更新语法类似于移动, 故 `user1` 中的被结构体更新语法使用的字段便不能再使用了.

3.1.3 元组结构体

元组结构体(Tuple Struct) 类似于普通结构体, 但没有字段名, 只有字段类型.

定义元组结构体的方法与定义普通结构体的方法类似, 只不过没有字段名.

```
struct Color(u8, u8, u8); //定义元组结构体
struct Point(i32, i32, i32); //定义元组结构体
fn main() {
    let green = Color(0, 255, 0);
    let origin = Point(0, 0, 0);
}
```

元组结构体间的类型是不互通的, 即使它们拥有着相同的类型. 例如, 两个同时为 `(i32, i32, i32)` 的结构体 `A` 与 `B` 也不能互通. 如果尝试把 `A` 的参数传给 `B` , 那么将会报错. 元组结构体类似于元组, 可以被解构, 也可以通过索引单独访问.

3.1.4 类单元结构体

类单元结构体 (unit-like struct) 是没有任何字段的结构体, 也称为单元结构体, 与单元元组类似. 比如:

```
struct AlwaysEqual; //定义类单元结构体
fn main () {
    let subject = AlwaysEqual;
}
```

3.1.5 结构体的应用

使用结构体能够大大增加代码的可读性. 以下代码是不使用结构体计算长方体表面积与体积的示例:

> 长方体的表面积 $S = 2ab + 2ac + 2bc$.
> 长方体的体积 $V = abc$.
> 其中 a, b, c 分别为长方体的长、宽、高.

```
//计算长方体的体积与表面积
fn volume(a: f64, b: f64, c: f64) -> f64 {
    a * b * c //计算长方体体积
} //定义长方体体积计算函数

fn superficial_area(a: f64, b: f64, c:f64) -> f64 {
    2f64 * a * b + 2f64 * a * c + 2f64 * b * c //计算长方体表面积
} //定义长方体表面积计算函数

fn main() {
    //输入数据
    let length0 = 3;
    let width0 = 7;
    let height0 = 11;
    //将数据转换为f64
    let length = length0 as f64;
    let width = width0 as f64;
    let height = height0 as f64;
    //计算体积
    let volume = volume(length, width, height);
    //计算面积
    let superficial_area = superficial_area(length, width, height);
    //输出
    println!("Volume: {}", volume);
    println!("Superficial Area: {}", superficial_area);
```

```
}
```

体积与面积计算函数的参数过多, 使得代码可读性差. 并且, 当输入的数据为整数时, 如果使用函数, 需要引入中间变量使其变为 `f64` 浮点数. 这里改用元组:

```
fn volume(v: (f64, f64, f64)) -> f64 {
    v.0 * v.1 * v.2 //计算长方体体积
}

fn superficial_area(s: (f64, f64, f64)) -> f64 {
    2f64 * s.0 * s.1 + 2f64 * s.0 * s.2 + 2f64 * s.1 * s.2 //计算长方体表面积
}

fn main() {
    //输入数据
    let length = 3;
    let width = 5;
    let height = 19;
    //转换类型并传入元组
    let rect: (f64, f64, f64) = (length as f64, width as f64, height as f64);
    let volume = volume(rect);
    let superficial_area = superficial_area(rect);
    println!("Volume: {}", volume);
    println!("Superficial Area: {}", superficial_area);
}
```

当我们使用元组时, 容易发现: 使用元组索引而非名字很容易将各个数据弄混. 这时, 我们可以使用结构体来解决这个问题:

```
struct Rectangle {
    length: f64,
    width: f64,
    height: f64,
} //定义长方体结构体

fn volume(r: &Rectangle) -> f64 {
    r.length * r.width * r.height //计算长方体体积
}

fn superficial_area(r: &Rectangle) -> f64 {
    2f64 * r.length * r.width + 2f64 * r.length * r.height + 2f64 * r.width * r.height //计算长方体表面积
}

fn main() {
    let length = 4;
    let width = 11;
    let height = 9;
    let rect1 = Rectangle {
        length: length as f64,
        width: width as f64,
        height: height as f64,
    }; //实例化长方体结构体
    let volume = volume(&rect1);
    let superficial_area = superficial_area(&rect1);
    println!("Volume: {}", volume);
    println!("Superficial Area: {}", superficial_area);
}
```

使用结构体, 使我们的代码可读性大大提高, 可以省去许多不必要的注释.

假如我们需要进行调试, 直接打印结构体的值是非常方便的. 然而, 在一般情况下, 这**不可行**. 假如我们尝试打印结构体:

```
struct Rectangle {
    width: u32,
    height: u32,
}
fn main() {
    let rect1 = Rectangle {
        width: 30,
        height: 50,
    };
    println!("rect1 is {}", rect1);
}
```

于是, 代码报错:

```
error[E0277]: `Rectangle` doesn't implement `std::fmt::Display`
  --> src\main.rs:11:29
   |
11 |     println!("rect1 is {}", rect1);
   |                             ^^^^^ `Rectangle` cannot be formatted with the default formatter
   |
   = help: the trait `std::fmt::Display` is not implemented for `Rectangle`
   = note: in format strings you may be able to use `{:?}` (or {:#?} for pretty-print) instead
   = note: this error originates in the macro `$crate::format_args_nl` which comes from the expansion of t
```

- `error[E0277]` 告诉我们, `Rectangle` 没有实现 `std::fmt::Display` trait.
- 不过, 它提示我们, 可以使用 `{:?}` 来打印.

```
struct Rectangle {
    width: u32,
    height: u32,
}
fn main() {
    let rect1 = Rectangle {
        width: 30,
        height: 50,
    };
    println!("rect1 is {:?}", rect1);
}
```

修改后再试一次, 不过仍然无济于事...吗?

```
error[E0277]: `Rectangle` doesn't implement `Debug`
  --> src\main.rs:11:31
   |
11 |     println!("rect1 is {:?}", rect1);
   |                               ^^^^^ `Rectangle` cannot be formatted using `{:?}`
   |
   = help: the trait `Debug` is not implemented for `Rectangle`
   = note: add `#[derive(Debug)]` to `Rectangle` or manually `impl Debug for Rectangle`
   = note: this error originates in the macro `$crate::format_args_nl` which comes from the expansion of t
   help: consider annotating `Rectangle` with `#[derive(Debug)]`
   |
```

```
2  + #[derive(Debug)]
3  | struct Rectangle {
   |
```

尽管还是报错, 但信息有所变化.

- 编译器提示我们, 可以添加 `#[derive(Debug)]` 来解决. 再来一次:

```
#[derive(Debug)]
struct Rectangle {
    width: u32,
    height: u32,
}
fn main() {
    let rect1 = Rectangle {
        width: 30,
        height: 50,
    };
    println!("rect1 is {:?}", rect1);
}
```

这一次, 代码成功运行了!