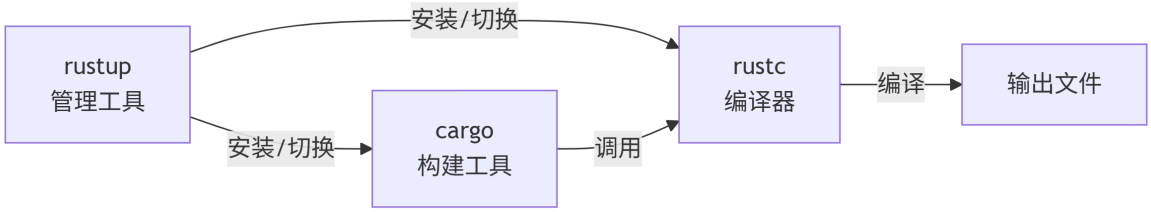
Rust介绍（上）

Rust工具链的安装与使用

我们使用`rustup`来管理rust工具链（安装多个版本的rust、设置默认版本等），使用`cargo`管理和构建rust项目（依赖管理、编译运行测试构建）。



- **Linux/MacOS**：执行以下命令下载安装

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

- **Windows**：访问网页Install Rust下载安装

安装完毕后执行`rustup --version`以及`cargo --version`确定安装成功。

> Rust Analyzer支持对代码实时检查，提供自动补全、定义跳转等功能，可以在VSCode中下载rust analyzer插件进行使用。

利用`cargo`创建我们的第一个项目`helloworld`：

```
cargo new helloworld
cd helloworld
```

运行该程序：

```
cargo run
```

```
calvin@calvin-lab:~/Desktop/helloworld$ cargo run
   Compiling helloworld v0.1.0 (/home/calvin/Desktop/helloworld)
    Finished `dev` profile [unoptimized + debuginfo] target(s) in 0.21s
     Running `target/debug/helloworld`
Hello, world!
```

cargo默认以`dev`模式编译，对于正式构建（例如使用`cargo build`等），可以加上`--release`参数在发布模式下编译以获得更好的性能。

练习1：使用`cargo build`在**release**模式下构建可执行文件并执行（可执行文件生成在`./target/release/`）。

常用类型与程序结构

Rust使用`mut`**关键字**来规定变量/借用的可变性，声明mut变量表示我们允许后续对变量进行修改，否则这个对象只能用于读取。

```
let a: i32 = 0;
a = 1; // error! a is immutable

let mut b: i32 = 0;
b = 1;
```

> mut更多用于修饰借用，类似读写锁互斥的思路，一个对象可以同时有多个不可变借用，但可变借用却必须独占变量。

常见的数据类型有如下几类，很多时候我们不需要显示地指定数据类型，因为Rust编译器会帮我们自动推导类型，但是`mut`依旧需要标注。

- i8, i16, i32, i64, i128, isize：有符号整数，isize表示大小取决于系统
- u8, ..., usize：无符号整数
- f32, f64：浮点数
- char：**utf-8字符**，实际可能占多个字节
- (T1, ...)：组，使用`.0`，`.1`来访问元素

  ```
  let mut tuple: (i32, i32, i32) = (1, 2, 3);
  tuple.0 = 4;
  ```

- [T; len]：数组，使用下标访问，超出数组长度会报错(编译期/运行期)

  ```
  let mut array: [i32; 3] = [1, 2, 3];
  array[3] = 4; //error, out of length!
  ```

- Vec：变长数组，使用下标[...]访问，运行时进行边界检查。文档：Vec in std::vec - Rust

  ```
  let mut a = Vec::new();
  a.push(1);
  a.push(2); //这里自动推导出a的类型是Vec<i32>

  let mut b = vec![1, 2]; //这里自动推导出b的类型是Vec<i32>
  println!("{}", b[0]);
  ```

- String：utf-8字符串。文档：String in std::string - Rust

  ```
  let mut a = String::from("这是一个字符串！");

  a.push_str("这是添加到末尾的部分！");
  ```

Rust中使用`fn`关键字声明函数，参数列表的声明方式和变量相同，使用`->`标记指定函数的返回值类型。

```rust
fn echo(mut val: i32) { //不需要返回值
    println!("The value is: {val}"); //以语句结尾，类型是()
}

fn plus_v1(a: i32, b: i32) -> i32 {
    return a + b; //通过return语句返回
}

fn plus_v2(a: i32, b: i32) -> i32 {
    a + b //直接以值返回，类型为i32
}
```

Rust 支持**以表达式作为函数的结尾**，这样可以简洁地返回表达式的值。在使用这一特性时，需要明确区分两个基本概念：表达式（Expression）与语句（Statement）。**通常可以通过是否以分号结尾来区分表达式与语句。**`{}`本身就是一个表达式，其值取括号内最后一个表达式的值。而语句我们一般认为它的类型是`()`。

```rust
let a: i32 = 0; //语句
let c: i32 = {
    let b: i32 = 0; //语句
    a + b //表达式，值为a + b，类型为i32
}; //使用{}表达式给c赋值
```

类似的，if-else结构也是表达式，其最终类型来自分支综合的结果，编译器会帮你做类型推导。

```rust
fn foo(a: i32) -> i32 {
    if a > 0 {
        a + 1
    } // 分支1：块表达式，值为a + 1，类型为i32
    else {
        a - 1
    } // 分支2：块表达式，值为a - 1，类型为i32
}

fn err_foo(a: i32) -> i32 {
    //ERROR! 需要两个分支类型匹配
    if a > 0 {
        a + 1
    } // 分支1：块表达式，值为a + 1，类型为i32
    else {
        println!("a < 0!");
    } // 分支2：块表达式，类型为()
}
```

下面再介绍两种简单的循环形式

```rust
while ... {
    ...;
}

loop {
    break ...; //break，并且以该值作为loop表达式的值
}
```

> 多重loop嵌套可以用 '... 标记快速break到外层，有兴趣可以自行查阅
>
> for循环我们后面和迭代器一起介绍

练习2：编写函数`fib(n: i32)`，当`n >= 0`时返回斐波那契数列中第`n`个数，否则返回`0`。例如f(-1) = 0，f(0) = 1，f(1) = 1，f(2)= 2，f(3)=3。你可以使用**循环**或**递归**来实现该函数。

参考代码：

```
fn fib_recursive(n: i32) -> i32 {
    if n < 0 {
        0
    }
    else if n <= 1 {
        1
    }
    else {
        fib_recursive(n - 1) + fib_recursive(n - 2)
    }
}

fn fib_loop(n: i32) -> i32 {
    if n < 0 {
        0
    }
    else {
        let mut a = 1;
        let mut b = 1;
        let mut i = 2;
        while i <= n {
            (a, b) = (b, a + b);
            i += 1;
        }
        b
    }
}
```

所有权与借用

常见内存管理方式：

- 手动分配释放（C风格）：容易出错
- 垃圾回收GC（Java等）：有性能损耗

Rust中所有权机制有三个规则：

1. 每个值都有一个所有者。
2. 同一时刻，每个值只能有一个所有者。
3. 当**所有者**离开作用域，值将被自动清理。

这些原则解决了C/C++中令人头疼的**double free**问题，**资源释放的责任被明确指派给资源所有者**，而所有者由编译器保证唯一。

对于拥有所有权的变量，赋值行为实际上是**移动**行为，所有权从赋值号右边移动到左边，此后右边的变量**失效**。

如果我们仍想保留右边的变量，那么必须调用`clone()`方法使用**克隆**对对象进行**深拷贝**。此后我们将会在内存中拥有数据相同但所有权各自管理的两份资源。

```
fn consume(s: String) {//函数要求接受一个拥有所有权的字符串对象作为参数
    //直接返回，s对应的资源会被销毁
}

fn consume_and_return(s: String) -> String { //函数要求接受一个拥有所有权的字符串对象作为参数，同时返回一个具有所有权的字符串对象
    s
}

let s1 = String::from("hello"); //这里s1是这个字符串对象的唯一所有者
let s2 = s1; //s1的所有权“移动”到了s2，此后s1失效。
let mut s3 = s2.clone(); //对s2进行深拷贝，s2，s3均有效

consume(s2); //s2的所有权“移动”到了函数参数中，此后s2失效，且最终字符串在函数内被销毁。
let s4 = consume_and_return(s3); //s3的所有权“移动”到函数中，又被返回出来“移动”到s4

let s5 = s2.clone(); //error! s2的所有权已经被移动走了，再使用这个变量是非法的。
```

熟悉了所有权的基本理念，我们再看一个似乎与上面代码矛盾的示例：

```
fn consume(i: i32) {
}

let a = 1;
consume(a);
println!("{a}"); //correct!
```

这段代码不会报错，因为对于像i32、bool等简单类型，他们在进行赋值操作时并不执行所有权的移动语义，而是在栈上**拷贝Copy出一个副本**。因此无论是传参还是赋值都不会影响原变量的所有权。

因此我们更关注Vec，String，以及自定义的结构体等复杂类型实例的所有权问题，简单类型则不用考虑。

> 本质上是这些类型实现了Copy trait，如果我们为自定义类型一样实现Copy trait，它一样可以在赋值时直接拷贝出副本。

所有权的机制虽然很棒，但是也带来一个问题：当程序复杂起来后，如果要用一堆中间变量来转移来转移去，那会使程序很臃肿，同时造成很大的不便。

Rust提供严格的**借用（Borrowing）**机制作为补充。借用可以在精确控制可变性的前提下，灵活共享数据。

```
fn length(s: &String) -> usize {
    s.len()
}

let s = String::from("hello");
let len = length(&s); //使用&创建一个s的借用
println!("{}", &s); //s依旧有效，依然可以建立借用
```

与变量的可变性一样，借用的可变性也用`mut`关键字控制。任意时刻只能存在一个可变借用或若干个不可变借用（以免存在数据竞争），且可变借用只能从可变变量创建。举个例子：

```
let mut s = String::from("hello");
let b = &mut s; //创建一个s的可变借用
println!("{}", &s); // error! 此前已有对s的可变借用b存在，且至少有效到下一行，因此这里不可以再创建不可变借用。
b.push_str(" world");
```

而如果我们调换上面两行的顺序，那么程序就可以正常编译。因为编译器检测到可变借用`b`之后不再使用，因此在任意时间都满足借用互斥。

用图来表示就是：

我们可以使用`*`运算符解引用获得**值**（实现了Copy trait，例如i32等原始类型），或者直接在借用上使用`.`运算符调用方法。

```
let v = vec![1, 2, 3];
let v_ref = &v;

let x = *v_ref; // error! Vec不能copy，直接解引用是移动操作，正确做法是v_ref.clone()

let val_ref = &v[0];
let y = *val_ref; //&i32可以直接通过*解引用

let length = v_ref.len(); //直接使用.访问Vec的方法
```

事实上，Rust通过Deref trait来控制解引用的行为，无论多少重引用，例如`&&Vec`这种双重引用，只需要一次解引用就可以正常执行了。

如果我们要从引用拷贝资源，建立一个拥有所有权的对象，那一样可以调用`clone()`方法。

```
let v = vec![1, 2, 3]
let v_ref = &v;
let u = v.clone(); //或调用to_owned()，效果相同，拷贝并获得一个新的Vec<i32>对象
```

练习3：编写两个版本的函数`append_suffix`，版本一接受`Vec<String>`作为参数，对其中所有的字符串添加后缀"_"后返回Vec。版本二接受`Vec<String>`的借用作为参数，做一样的修改。

参考代码：

```
fn append_suffix_v1(mut strings: Vec<String>) -> Vec<String> {
    let mut i = 0;
    while i < strings.len() {
        strings[i].push_str("_");
```

```
        i += 1;
    }
    strings
}

fn append_suffix_v2(strings: &mut Vec<String>) {
    let mut i = 0;
    while i < strings.len() {
        strings[i].push_str("_");
        i += 1;
    }
}
```

迭代器与闭包

Rust中`for`循环一般与迭代器搭配使用。举个例子：

```
for i in 1..10 {
    println!("{}", i);
}
```

这里`1..10`实际上展开后是是range(1, 10)，而`for`的作用就是从迭代器中取出元素并进行遍历。有三种常用方法为我们提供迭代器，实际上这些方法的行为都由具体trait的实现决定，这里我们只介绍惯用的实现：

- iter：迭代每个元素的不可变引用。
- iter_mut：迭代每个元素的可变引用。
- into_iter：迭代出来的每个元素都具有所有权【从原集合转移到迭代出的对象】。

```
let mut a: Vec<Vec<i32>> = vec![vec![1, 2], vec![3, 4]];
for i in a.iter() {} //i的类型：&Vec<i32>
for i in a.iter_mut() {} //i的类型：&mut Vec<i32>
for i in a.into_iter() {} //i的类型：Vec<i32>

let b = a.into_iter(); //error! a的所有权已经被上次into_iter消费了，因此a已经失效，不能再调用函数
```

迭代器具有迭代器适配器和消费者适配器两类方法，迭代器适配器在迭代器的基础上加装操作并产生新的迭代器，而消费者适配器消耗**迭代器的所有权**并实际承担之前的一切计算。

常见的迭代器适配器包括：

- map()：映射
- filter()：过滤
- ……

而消费者适配器则包括：

- sum()：求和
- collect()：收集元素构建一个新的数据结构，**类型由上下文推导或显式指定。**
- ……

在使用迭代器适配器时，我们一般使用函数闭包来操作迭代器元素。它的形式类似

```
let x = 1
let sum = |y| { // |y|表示它接受一个参数y，这里编译器推导出y的类型是i32，闭包返回值类型为i32，你也可以自己显式指定
    x + y // x捕获自作用域中x的*值*
};
```

如果我们需要捕获作用域中变量获得其可变借用，或者需要获得其所有权，那可以用如下格式：

```
let mut a = vec![1, 2, 3];
let mut append = |v: i32| {
    a.push(v) //捕获a的可变借用
};
let sum = move || -> i32 {
    a.into_iter().sum() //捕获a变量（具有所有权）
};
```

前面介绍的迭代器适配器大多都会接受函数闭包作为参数，这提供了极大的自由度。例如

```
let mut a = vec![1, 2, 3];
let b: Vec<i32> = a.iter().map(|i| {
    *i + 1
}).collect();

assert!(a.len() == b.len()); //a和b元素个数相同，且a仍有效
```

这里闭包`|i| { i + 1 }`就是map的参数，它表示对`iter()`中取出的每个`&i32`类型的元素先解引用（这里触发**Copy**）然后加1并返回。因此最终`collect()`创建出了一个**新**的Vec对象，每个元素是原集合元素加1的结果。

`map`的后面还可以链条式地继续加迭代器适配器，这种函数式编程的表达能力正是 Rust 迭代器的强大之处。

```
let numbers = vec![1, 2, 3, 4, 5];

let result: Vec<i32> = numbers
    .iter()
    .map(|x| x * 2)           // [2, 4, 6, 8, 10]
    .filter(|x| x > &5)       // [6, 8, 10]
    .collect();
```

练习4：编写函数`analyze_score()`，接受`&Vec<(String, i32)>`作为参数，返回优秀学生(score >= 85)的平均分。

参考代码：

```rust
fn analyze_score(scores: &Vec<(String, i32)>) -> f32 {
    let excellent_scores: Vec<i32> = scores
        .iter()
        .map(|student| student.1)
        .filter(|score| *score >= 85)
        .collect();
    let sum: i32 = excellent_scores.iter().sum();
    sum as f32 / excellent_scores.len() as f32
}
```