Rust

1. 编译与运行

- 创建项目：`cargo new`
- 直接运行：`cargo run`，如要传入命令行参数 `cargo run -- <arg1> <arg2>`
  - 可以定义多个入口 bin，在 `Cargo.toml` 文件中定义

```
[[bin]]
name = "bin1"
path = "src/cli/bin1.rs"

[[bin]]
name = "bin2"
path = "src/cli/bin2.rs"

# 这样可以 cargo run bin1/bin2
```

- 编译：`cargo build`，可执行文件在 `./target/debug/xxx`，默认是 debug 模式，代码编译很快但是运行速度很慢；release 模式编译 `cargo build --release`，运行是 `./target/release/xxx`
  - Debug 模式会占用更多的硬盘，如果空间不大可以使用 release，同时为了避免爆内存可以设置 `cargo build -j 4` 以 4 线程模式进行编译
- 当项目变大之后快速检查编译是否通过：`cargo check`
- 添加包：`cargo add`，删除包：`cargo remove`
  - 一般会在最外层 `Cargo.toml` 设置依赖的版本和信息（用 workspace），然后里面依赖和信息继承 workspace

```
# 最外层
[workspace]
members = [
    "account-decoder",
    "accounts-bench",
    ...
]

```

```toml
exclude = [
    "programs/sbf",
]

# This prevents a Travis CI error when building for Windows.
resolver = "2"

[workspace.package]
version = "1.17.0"
authors = ["Solana Labs Maintainers <maintainers@solanalabs.com>"]
repository = "https://github.com/solana-labs/solana"
homepage = "https://solanalabs.com/"
license = "Apache-2.0"
edition = "2021"

[workspace.dependencies]
aes-gcm-siv = "0.10.3"
ahash = "0.8.3"

# 里层
[package]
name = "solana-cli"
description = "Blockchain, Rebuilt for Scale"
documentation = "https://docs.rs/solana-cli"
version = { workspace = true }
authors = { workspace = true }
repository = { workspace = true }
homepage = { workspace = true }
license = { workspace = true }
edition = { workspace = true }

[dependencies]
bincode = { workspace = true }
bs58 = { workspace = true }
...

[dev-dependencies]
solana-streamer = { workspace = true }
solana-test-validator = { workspace = true }
tempfile = { workspace = true }

[[bin]]
name = "solana"
path = "src/main.rs"

[package.metadata.docs.rs]
targets = ["x86_64-unknown-linux-gnu"]
```

- `Cargo.toml` 是项目数据描述文件
- `Cargo.lock` ：如果是可运行程序就要上传，如果是依赖库项目就不要上传
- 换源：https://course.rs/first-try/slowly-downloading.html

2. 基础

2.1 入门代码

```rust
// Rust 程序入口函数，跟其它语言一样，都是 main，该函数目前无返回值
fn main() {
    // 使用let来声明变量，进行绑定，a是不可变的
    // 此处没有指定a的类型，编译器会默认根据a的值为a推断类型：i32，有符号32位整数
    // 语句的末尾必须以分号结尾
    let a = 10;
    // 主动指定b的类型为i32
    let b: i32 = 20;
    // 这里有两点值得注意：
    // 1. 可以在数值中带上类型:30i32表示数值是30，类型是i32
    // 2. c是可变的，mut是mutable的缩写
    let mut c = 30i32;
    // 还能在数值和类型中间添加一个下划线，让可读性更好
    let d = 30_i32;
    // 跟其它语言一样，可以使用一个函数的返回值来作为另一个函数的参数
    let e = add(add(a, b), add(c, d));

    // println!是宏调用，看起来像是函数但是它返回的是宏定义的代码块
    // 该函数将指定的格式化字符串输出到标准输出中(控制台)
    // {}是占位符，在具体执行过程中，会把e的值代入进来
    println!("( a + b ) + ( c + d ) = {}", e);
}

// 定义一个函数，输入两个i32类型的32位有符号整数，返回它们的和
fn add(i: i32, j: i32) -> i32 {
    // 返回相加值，这里可以省略return
    i + j
}
```

2.2 变量绑定与解构

- 不可变变量 `let x = 5;` 可变变量 `let mut x = 5;`
- 使用下划线开头忽略未使用的变量 `let _x = 5;`

- 变量解构

```rust
fn main() {
    let (a, mut b): (bool,bool) = (true, false);
    // a = true,不可变; b = false, 可变
    println!("a = {:?}, b = {:?}", a, b);

    b = true;
    assert_eq!(a, b);
}
```

```rust
struct Struct {
    e: i32
}

fn main() {
    let (a, b, c, d, e);

    (a, b) = (1, 2);
    // _ 代表匹配一个值，但是我们不关心具体的值是什么，因此没有使用一个变量名而是使用了 _
    [c, .., d, _] = [1, 2, 3, 4, 5];
    Struct { e, .. } = Struct { e: 5 };

    assert_eq!([1, 2, 1, 4, 5], [a, b, c, d, e]);
}
```

- 常量：`const MAX_POINTS: u32 = 100_000;`
- 变量遮蔽：Rust 允许声明相同的变量名（类型不同也可以），在后面声明的变量会遮蔽掉前面声明的

```rust
fn main() {
    let x = 5;
    // 在main函数的作用域内对之前的x进行遮蔽
    let x = x + 1;

    {
        // 在当前的花括号作用域内，对之前的x进行遮蔽
        let x = x * 2;
        println!("The value of x in the inner scope is: {}", x);
    }

```

```
12      println!("The value of x is: {}", x);
13  }
```

2.3 基本类型

- https://course.rs/basic/base-type/index.html
- 类型
  - 基本类型
    - 数值类型: 有符号整数 (`i8` , `i16` , `i32` , `i64` , `isize`)、 无符号整数 (`u8` , `u16` , `u32` , `u64` , `usize`) 、浮点数 (`f32` , `f64`)、以及有理数、复数
    - 字符串：字符串字面量和字符串切片 `&str`
    - 布尔类型： `true` 和 `false`
    - 字符类型: 表示单个 Unicode 字符，存储为 4 个字节
    - 单元类型: 即 `()` ，其唯一的值也是 `()`
  - 复合类型
- Rust 是静态类型语言，编译器必须在编译期知道所有变量的类型
- 序列

```
1  for i in 1..=5 {
2      println!("{}",i);
3  }
4
5  for i in 'a'..='z' {
6      println!("{}",i);
7  }
```

- 单元类型 `()` ，如 `fn main()` 返回值就是 `()`
- 表达式与语句
  - 函数是表达式，如果没有返回值，就返回 `()` ；语句也返回 `()`
- 函数

```
1  fn add(i: i32, j: i32) -> i32 {
2      i + j
3  }
```

- 如果函数永不返回（例如无限循环），可以将返回值设置成 `!`

```
fn forever() -> ! {
  loop {
    //...
  };
}
```

2.4 所有权和借用

2.4.1 所有权

- 所有权原则
  - Rust 中每一个值都被一个变量所拥有，该变量被称为值的所有者
  - 一个值同时只能被一个变量所拥有，或者说一个值只能拥有一个所有者
  - 当所有者(变量)离开作用域范围时，这个值将被丢弃(drop)
- 字符串
  - 字符串字面量类型 `&str`，硬编码到程序代码里，不可变
  - 字符串 `String` 是需要动态分配到堆上可以动态伸缩的
  - 使用例子

```
let mut s = String::from("hello");

s.push_str(", world!"); // push_str() 在字符串后追加字面值

println!("{}", s); // 将打印 `hello, world!`
```

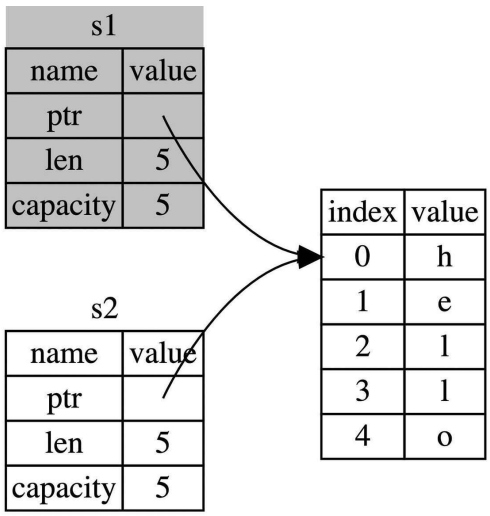
- 转移所有权
  - 栈上：自动拷贝，这种操作是 **浅拷贝**，因为两个变量都可以使用，在栈上都有自己的空间，基本类型都可以浅拷贝，最特别的是不可变引用（`&T`）也是可以浅拷贝的，但是可变引用（`&mut T`）不行。可以浅拷贝的类型都有一个叫做 `Copy` 的特征

```
let x = 5;
let y = x;

// 以下操作是可以的
fn main() {
```

```
    let x: &str = "hello, world";
    let y = x; // 对不可变引用进行浅拷贝
    println!("{},{}",x,y);
}
```

- 堆上：以下操作只会自动拷贝栈上的部分（**堆指针、字符串长度、字符串容量，在64位电脑上是24字节**），又因为一个值只能有一个 owner（否则会出现离开作用域二次释放的问题），因此这里 `s1` 所有权转移给了 `s2`， `s1` 失效了。这种操作叫 **移动**

```
let s1 = String::from("hello");
let s2 = s1;
```



- **深拷贝**：rust 不会自动深拷贝，需要调用特定函数

```
let s1 = String::from("hello");
let s2 = s1.clone();

println!("s1 = {}, s2 = {}", s1, s2);
```

- 函数传值会发生移动或者浅拷贝

```
fn main() {
    let s = String::from("hello");  // s 进入作用域

    takes_ownership(s);             // s 的值移动到函数里 ...
                                    // ... 所以到这里不再有效

    let x = 5;                      // x 进入作用域
```

```
8
9       makes_copy(x);                  // x 应该移动函数里，
10                                      // 但 i32 是 Copy 的，所以在后面可继续使用 x
11
12  } // 这里，x 先移出了作用域，然后是 s。但因为 s 的值已被移走，
13    // 所以不会有特殊操作
14
15  fn takes_ownership(some_string: String) { // some_string 进入作用域
16      println!("{}", some_string);
17  } // 这里，some_string 移出作用域并调用 `drop` 方法。占用的内存被释放
18
19  fn makes_copy(some_integer: i32) { // some_integer 进入作用域
20      println!("{}", some_integer);
21  } // 这里，some_integer 移出作用域。不会有特殊操作
```

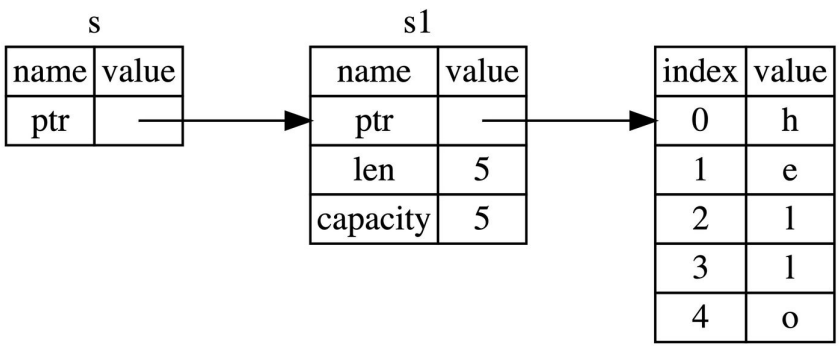
- 函数返回值会发生移动或者浅拷贝

```
1   fn main() {
2       let s1 = gives_ownership();         // gives_ownership 将返回值
3                                           // 移给 s1
4
5       let s2 = String::from("hello");     // s2 进入作用域
6
7       let s3 = takes_and_gives_back(s2);  // s2 被移动到
8                                           // takes_and_gives_back 中,
9                                           // 它也将返回值移给 s3
10  } // 这里，s3 移出作用域并被丢弃。s2 也移出作用域，但已被移走，
11    // 所以什么也不会发生。s1 移出作用域并被丢弃
12
13  fn gives_ownership() -> String {             // gives_ownership 将返回值移动给
14                                              // 调用它的函数
15
16      let some_string = String::from("hello"); // some_string 进入作用域.
17
18      some_string                             // 返回 some_string 并移出给调用的
    函数
19  }
20
21  // takes_and_gives_back 将传入字符串并返回该值
22  fn takes_and_gives_back(a_string: String) -> String { // a_string 进入作用域
23
24      a_string  // 返回 a_string 并移出给调用的函数
25  }
```

2.4.2 引用与解引用

- 常规引用（不可变引用）是一个指针类型，指向对象存储的内存地址

```rust
fn main() {
    let x = 5;
    let y = &x;

    assert_eq!(5, x);
    assert_eq!(5, *y);
}

fn main() {
    let s1 = String::from("hello");

    let len = calculate_length(&s1);

    println!("The length of '{}' is {}.", s1, len);
}

fn calculate_length(s: &String) -> usize {
    s.len() // 没有获得所有权，离开作用域的时候，指向的值不会被丢弃
}
```



- 可变引用：**同一作用域，特定数据只能有一个可变引用**

```rust
// 第一种情况
let mut s = String::from("hello");

let r1 = &mut s;
let r2 = &mut s;

```

```rust
println!("{}, {}", r1, r2); // 出错的原因在于，第一个可变借用 r1 必须要持续到最后一次使用的位置 println!，在 r1 创建和最后一次使用之间，我们又尝试创建第二个可变借用 r2。

// 第二种情况
let mut s = String::from("hello");

{
    let r1 = &mut s;

} // r1 在这里离开了作用域，所以我们完全可以创建一个新的引用

let r2 = &mut s;

// 第三种情况
let mut s = String::from("hello");

let r1 = &s; // 没问题
let r2 = &s; // 没问题
let r3 = &mut s; // 大问题，可变引用与不可变引用不能在一个作用域中同时存在，这里的作用域指引用的作用域，不是变量的作用域

println!("{}, {}, and {}", r1, r2, r3);
// 注意，引用的作用域 s 从创建开始，一直持续到它最后一次使用的地方，这个跟变量的作用域有所不同，变量的作用域从创建持续到某一个花括号 }

// 第四种情况
fn main() {
   let mut s = String::from("hello");

    let r1 = &s;
    let r2 = &s;
    println!("{} and {}", r1, r2);
    // 新编译器中，r1,r2作用域在这里结束

    let r3 = &mut s;
    println!("{}", r3);
} // 老编译器中，r1、r2、r3作用域在这里结束
  // 新编译器中，r3作用域在这里结束，这种技术叫 Non-Lexical Lifetimes(NLL)
```

- Rust 不允许出现悬垂指针，即引用对应的对象已经被释放掉了

```rust
fn main() {
    let reference_to_nothing = dangle();
}
```

```rust

fn dangle() -> &String { // dangle 返回一个字符串的引用

    let s = String::from("hello"); // s 是一个新字符串

    &s // 返回字符串 s 的引用
} // 这里 s 离开作用域并被丢弃。其内存被释放。
  // 危险!
```

- 总结
  - 同一时刻，你只能拥有要么一个可变引用, 要么任意多个不可变引用
  - 引用必须总是有效的

2.5 复合类型

2.5.1 字符串与切片

- 切片
  - 切片是集合的一部分，字符串切片类型 `&str`，数组切片类型 `&[i32]`
  - 一个汉字是 3 个字符，至少需要取三个， `&s[0..3]`
- 字符串字面量是不可变引用 `&str`
- `String` 和 `&str` 的转换

```rust
// String -> &str
let s = String::from("hello,world!");
say_hello(&s);
say_hello(&s[..]);
say_hello(s.as_str());

// &str -> String
String::from("hello,world")
"hello,world".to_string()
```

- 字符串不能取索引 `s[0]` 是不行的
- 字符串相关添加、删除、连接的操作：https://course.rs/basic/compound-type/string-slice.html

2.5.2 元组

- 模式匹配

```
```

```rust
fn main() {
    let tup = (500, 6.4, 1);

    let (x, y, z) = tup;

    println!("The value of y is: {}", y);
}
```

- 用 `.` 来访问

```rust
fn main() {
    let x: (i32, f64, u8) = (500, 6.4, 1);

    let five_hundred = x.0;

    let six_point_four = x.1;

    let one = x.2;
}
```

2.5.3 结构体

- 结构体定义、创建例子

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

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

// 简化创建
fn build_user(email: String, username: String) -> User {
    User {
```

```
        email,
        username,
        active: true,
        sign_in_count: 1,
    }
}

// 赋值
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");

// 更新
let user2 = User {
    email: String::from("another@example.com"),
    ..user1
};

// 所有权转移
let user1 = User {
    email: String::from("someone@example.com"),
    username: String::from("someusername123"),
    active: true,
    sign_in_count: 1,
};
let user2 = User {
    active: user1.active,
    username: user1.username,
    email: String::from("another@example.com"),
    sign_in_count: user1.sign_in_count,
};
println!("{}", user1.active); // active 会浅拷贝，但是 username 发生了移动，所以 user1 用不了但是 user1.active 可以用
// 下面这行会报错
println!("{:?}", user1);
```

- 元组结构体

```
struct Color(i32, i32, i32);
struct Point(i32, i32, i32);
```

```
3
4 let black = Color(0, 0, 0);
5 let origin = Point(0, 0, 0);
```

- 单元结构体，不关心属性只关心行为

```
1 struct AlwaysEqual;
2
3 let subject = AlwaysEqual;
4
5 // 我们不关心 AlwaysEqual 的字段数据，只关心它的行为，因此将它声明为单元结构体，然后再为它实现某个特征
6 impl SomeTrait for AlwaysEqual {
7
8 }
```

- 打印结构体

```
1 #[derive(Debug)]
2 struct Rectangle {
3     width: u32,
4     height: u32,
5 }
6
7 fn main() {
8     let rect1 = Rectangle {
9         width: 30,
10        height: 50,
11    };
12
13    println!("rect1 is {:?}", rect1); // {:#?} 效果更好，使用 {} 需要实现 Display 特性，使用 {:?} 需要实现 Debug 特性
14 }
```

2.5.4 枚举

- 基本使用

```
1 enum PokerSuit {
2   Clubs,
3   Spades,
```

```
    Diamonds,
    Hearts,
}

fn main() {
    let heart = PokerSuit::Hearts;
    let diamond = PokerSuit::Diamonds;

    print_suit(heart);
    print_suit(diamond);
}

fn print_suit(card: PokerSuit) {
    // 需要在定义 enum PokerSuit 的上面添加上 #[derive(Debug)]，否则会报 card 没有实现 Debug
    println!("{:?}",card);
}

enum PokerCard {
    Clubs(u8),
    Spades(u8),
    Diamonds(char),
    Hearts(char),
}

fn main() {
   let c1 = PokerCard::Spades(5);
   let c2 = PokerCard::Diamonds('A');
}
```

- `Option` 枚举类型

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

let some_number = Some(5);
let some_string = Some("a string");

let absent_number: Option<i32> = None;

fn plus_one(x: Option<i32>) -> Option<i32> {
    match x {
        None => None,
```

```
        Some(i) => Some(i + 1),
    }
}

let five = Some(5);
let six = plus_one(five);
let none = plus_one(None);
```

2.5.5 数组

- 固定长度 array，不固定长度 vector
- 固定长度

```
let a: [i32; 5] = [1, 2, 3, 4, 5]; // 类型
let a = [3; 5]; // 多个相同值组成的 array
let array = [String::from("rust is good!"); 8]; // 报错，不支持 Copy 特性
let array: [String; 8] = std::array::from_fn(|_i| String::from("rust is good!")); // 正确
```

2.6 流程控制

- For

```
for item in &container { // 一般使用引用，否则控制权就到了 for 语句块中，后续无法使用了
  // ...
}

for item in &mut collection { // 想修改加 mut
  // ...
}

fn main() {
    let a = [4, 3, 2, 1];
    // `.iter()` 方法把 `a` 数组变成一个迭代器
    for (i, v) in a.iter().enumerate() { // 获取索引
        println!("第{}个元素是{}", i + 1, v);
    }
}
```

使用方法	等价使用方式	
for item in collection	for item in IntoIterator::into_iter(collection)	转移
for item in &collection	for item in collection.iter()	不可
for item in &mut collection	for item in collection.iter_mut()	可变

- Loop

```rust
fn main() {
    let mut counter = 0;

    let result = loop {
        counter += 1;

        if counter == 10 {
            break counter * 2;
        }
    };

    println!("The result is {}", result);
}
```

2.7 模式匹配

- Match

```rust
match target {
    模式1 => 表达式1, // 还可以：模式 1 | 模式2, 还可以 1..=5
    模式2 => {
        语句1;
        语句2;
        表达式2
    },
    _ => 表达式3
} // 需要匹配所有情况

fn plus_one(x: Option<i32>) -> Option<i32> {
    match x {
        None => None,
        Some(i) => Some(i + 1),
    }
}
```

```
17
18 let five = Some(5);
19 let six = plus_one(five);
20 let none = plus_one(None);
```

- If let / While let：只需要匹配一个条件的时候使用

```
1 if let Some(3) = v {
2     println!("three");
3 }
4
5 while let Some(top) = stack.pop() {
6     println!("{}", top);
7 }
```

- `matches!` 宏

```
1 v.iter().filter(|x| x == MyEnum::Foo); // 这里不能等于
2 v.iter().filter(|x| matches!(x, MyEnum::Foo));
3
4 let foo = 'f';
5 assert!(matches!(foo, 'A'..='Z' | 'a'..='z'));
6
7 let bar = Some(4);
8 assert!(matches!(bar, Some(x) if x > 2));
```

- 绑定 @ 有点复杂，就先不管了 全模式列表 - Rust语言圣经(Rust Course)

2.8 方法

- 使用 `impl` 来定义方法

```
1 struct Circle {
2     x: f64,
3     y: f64,
4     radius: f64,
5 }
6
7 impl Circle {
8     // new是Circle的关联函数，因为它的第一个参数不是self，且new并不是关键字
9     // 这种方法往往用于初始化当前结构体的实例
```

```
    fn new(x: f64, y: f64, radius: f64) -> Circle {
        Circle {
            x: x,
            y: y,
            radius: radius,
        }
    }

    // Circle的方法，&self表示借用当前的Circle结构体
    fn area(&self) -> f64 {
        std::f64::consts::PI * (self.radius * self.radius)
    }
}
```

- self、&self 和 &mut self
  - `self` 表示 `Rectangle` 的所有权转移到该方法中，这种形式用的较少
  - `&self` 表示该方法对 `Rectangle` 的不可变借用
  - `&mut self` 表示可变借用
  - `Self` 表示的类型，小写的是实例
- 枚举也可以实现方法

2.9 泛型和特征

2.9.1 泛型

- 例子

```
fn add<T: std::ops::Add<Output = T>>(a:T, b:T) -> T { // T 要满足可相加
    a + b
}

fn largest<T: std::cmp::PartialOrd>(list: &[T]) -> T { // T 要满足可比较
    let mut largest = list[0];

    for &item in list.iter() {
        if item > largest {
            largest = item;
        }
    }

    largest
}
```

```
16
17 fn main() {
18     let number_list = vec![34, 50, 25, 100, 65];
19
20     let result = largest(&number_list);
21     println!("The largest number is {}", result);
22
23     let char_list = vec!['y', 'm', 'a', 'q'];
24
25     let result = largest(&char_list);
26     println!("The largest char is {}", result);
27 }
```

- 枚举中经典泛型

```
1 enum Option<T> {
2     Some(T),
3     None,
4 }
5
6 enum Result<T, E> {
7     Ok(T),
8     Err(E),
9 }
```

- 方法泛型

```
1 struct Point<T, U> {
2     x: T,
3     y: U,
4 }
5
6 impl<T, U> Point<T, U> {
7     fn mixup<V, W>(self, other: Point<V, W>) -> Point<T, W> {
8         Point {
9             x: self.x,
10            y: other.y,
11        }
12    }
13 }
14
15 impl Point<f32> { // 特定泛型下的方法
16     fn distance_from_origin(&self) -> f32 {
```

```
        (self.x.powi(2) + self.y.powi(2)).sqrt()
    }
}
```

- 数组泛型

```
// 使用数组切片
fn display_array<T: std::fmt::Debug>(arr: &[T]) {
    println!("{:?}", arr);
}
fn main() {
    let arr: [i32; 3] = [1, 2, 3];
    display_array(&arr);

    let arr: [i32;2] = [1,2];
    display_array(&arr);
}

// 直接使用数组
fn display_array<T: std::fmt::Debug, const N: usize>(arr: [T; N]) {
    println!("{:?}", arr);
}
fn main() {
    let arr: [i32; 3] = [1, 2, 3];
    display_array(arr);

    let arr: [i32; 2] = [1, 2];
    display_array(arr);
}
```

- 编译器会帮我们做单态化操作，即将泛型确定成实际使用的那些类型

2.9.2 特征

- 特征定义和实现的例子

```
pub trait Summary {
    fn summarize(&self) -> String;
}
pub struct Post {
    pub title: String, // 标题
    pub author: String, // 作者
    pub content: String, // 内容
```

```rust
}

impl Summary for Post {
    fn summarize(&self) -> String {
        format!("文章{}, 作者是{}", self.title, self.author)
    }
}

pub struct Weibo {
    pub username: String,
    pub content: String
}

impl Summary for Weibo {
    fn summarize(&self) -> String {
        format!("{}发表了微博{}", self.username, self.content)
    }
}
```

- 特征作为函数参数

```rust
// 单个约束
pub fn notify(item: &impl Summary) { // 语法糖
    println!("Breaking news! {}", item.summarize());
}

// 等价于
pub fn notify<T: Summary>(item: &T) {
    println!("Breaking news! {}", item.summarize());
}

// 多重约束
pub fn notify(item: &(impl Summary + Display)) {}
pub fn notify<T: Summary + Display>(item: &T) {}
// 用 where 方便一些
fn some_function<T, U>(t: &T, u: &U) -> i32
    where T: Display + Clone,
          U: Clone + Debug
{}

// 上一节中的例子
fn largest<T: PartialOrd + Copy>(list: &[T]) -> T {
    let mut largest = list[0];

    for &item in list.iter() {
```

```
        if item > largest {
            largest = item;
        }
    }

    largest
}

fn main() {
    let number_list = vec![34, 50, 25, 100, 65];

    let result = largest(&number_list);
    println!("The largest number is {}", result);

    let char_list = vec!['y', 'm', 'a', 'q'];

    let result = largest(&char_list);
    println!("The largest char is {}", result);
}
```

- 特征作为返回值

```
fn returns_summarizable() -> impl Summary {
    Weibo {
        username: String::from("sunface"),
        content: String::from(
            "m1 max太厉害了，电脑再也不会卡",
        )
    }
}
```

2.9.3 特征对象

- 一个无法通过编译的代码：尽管 `Post` 和 `Weibo` 都实现了 `Summary` 但是不能返回不同的类型，我理解是这里 `impl Summary` 在编译的时候还是会换成 `Post/Weibo` 导致不能返回两种类型

```
fn returns_summarizable(switch: bool) -> impl Summary {
    if switch {
        Post {
            // ...
        }
    } else {
```

```
        Weibo {
            // ...
        }
    }
}

```

- 特征对象：动态分发

```
trait Draw {
    fn draw(&self) -> String;
}

impl Draw for u8 {
    fn draw(&self) -> String {
        format!("u8: {}", *self)
    }
}

impl Draw for f64 {
    fn draw(&self) -> String {
        format!("f64: {}", *self)
    }
}

// 若 T 实现了 Draw 特征， 则调用该函数时传入的 Box<T> 可以被隐式转换成函数参数签名中的 Box<dyn Draw>
fn draw1(x: Box<dyn Draw>) {
    // 由于实现了 Deref 特征，Box 智能指针会自动解引用为它所包裹的值，然后调用该值对应的类型上定义的 `draw` 方法
    x.draw();
}

fn draw2(x: &dyn Draw) {
    x.draw();
}

fn main() {
    let x = 1.1f64;
    // do_something(&x);
    let y = 8u8;

    // x 和 y 的类型 T 都实现了 `Draw` 特征，因为 Box<T> 可以在函数调用时隐式地被转换为特征对象 Box<dyn Draw>
    // 基于 x 的值创建一个 Box<f64> 类型的智能指针，指针指向的数据被放置在了堆上
```

```
    draw1(Box::new(x));
    // 基于 y 的值创建一个 Box<u8> 类型的智能指针
    draw1(Box::new(y));
    draw2(&x);
    draw2(&y);
}

// 关键在此：这个vector里面可以放不同类型，如果是 Vec<T> 那就只能一个类型了
pub struct Screen {
    pub components: Vec<Box<dyn Draw>>,
}

```

2.9.4 特征进阶 TODO

- https://course.rs/basic/trait/advance-trait.html

2.10 集合类型

2.10.1 Vector

- 创建 Vector

```
let v: Vec<i32> = Vec::new();

let mut v = Vec::new();
v.push(1);

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

- 读取元素

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

let third: &i32 = &v[2];
println!("第三个元素是 {}", third);

match v.get(2) {
    Some(third) => println!("第三个元素是 {third}"),
    None => println!("去你的第三个元素，根本没有！"),
}

// 注意下面这个例子
```

```rust
let mut v = vec![1, 2, 3, 4, 5];

let first = &v[0]; // 不可变引用

v.push(6); // 可变引用

println!("The first element is: {first}"); // 如果不可变引用这里没有用到就可以，但是这里用到了就不行
```

- 遍历

```rust
let v = vec![1, 2, 3];
for i in &v {
    println!("{i}");
}

let mut v = vec![1, 2, 3];
for i in &mut v {
    *i += 10
}

```

- 存储不同类型的元素（静态）

```rust
#[derive(Debug)]
enum IpAddr {
    V4(String),
    V6(String)
}
fn main() {
    let v = vec![
        IpAddr::V4("127.0.0.1".to_string()),
        IpAddr::V6("::1".to_string())
    ];

    for ip in v {
        show_addr(ip)
    }
}

fn show_addr(ip: IpAddr) {
    println!("{:?}",ip);
}
```

- 存储不同类型的元素（动态），更常用

```rust
trait IpAddr {
    fn display(&self);
}

struct V4(String);
impl IpAddr for V4 {
    fn display(&self) {
        println!("ipv4: {:?}",self.0)
    }
}
struct V6(String);
impl IpAddr for V6 {
    fn display(&self) {
        println!("ipv6: {:?}",self.0)
    }
}

fn main() {
    let v: Vec<Box<dyn IpAddr>> = vec![
        Box::new(V4("127.0.0.1".to_string())),
        Box::new(V6("::1".to_string())),
    ];

    for ip in v {
        ip.display();
    }
}
```

- 其他内容：https://course.rs/basic/collections/vector.html

2.10.2 KV 存储 HashMap

- 创建

```rust
use std::collections::HashMap;

// 创建一个HashMap，用于存储宝石种类和对应的数量
let mut my_gems = HashMap::new();

// 将宝石类型和对应的数量写入表中
my_gems.insert("红宝石", 1);
```

```
8 my_gems.insert("蓝宝石", 2);
9 my_gems.insert("河边捡的误以为是宝石的破石头", 18);
```

2.11 认识生命周期

- 对于函数，它的返回值是一个引用类型
  - 那么该引用只有两种情况：
    - 从参数获取
    - 从函数体内部新创建的变量获取
  - 如果是后者，就会出现悬垂引用，最终被编译器拒绝，因此只剩一种情况：返回值的引用是获取自参数，这就意味着参数和返回值的生命周期是一样的。

```
1  fn longest<'a>(x: &'a str, y: &'a str) -> &'a str {
2      if x.len() > y.len() {
3          x
4      } else {
5          y
6      }
7  }
8
9  fn longest<'a>(x: &'a str, y: &str) -> &'a str {
10     x
11 }
12
13 fn longest<'a>(_x: &str, _y: &str) -> String {
14     String::from("really long string")
15 }
```

- 对于结构
  - 例子

```
1  struct ImportantExcerpt<'a> {
2      part: &'a str, // 如果使用引用作为成员变量，最好加上生命周期
3  }
4
5  fn main() {
6      let novel = String::from("Call me Ishmael. Some years ago...");
7      let first_sentence = novel.split('.').next().expect("Could not find
   a '.'");
8      let i = ImportantExcerpt {
```

```
 9          part: first_sentence,
10      };
11  }
```

- 默认规则
  a. **每一个引用参数都会获得独自的生命周期**
     i. 例如一个引用参数的函数就有一个生命周期标注: `fn foo<'a>(x: &'a i32)`，两个引用参数的有两个生命周期标注: `fn foo<'a, 'b>(x: &'a i32, y: &'b i32)`，依此类推。
  b. **若只有一个输入生命周期(函数参数中只有一个引用类型)，那么该生命周期会被赋给所有的输出生命周期**，也就是所有返回值的生命周期都等于该输入生命周期
     i. 例如函数 `fn foo(x: &i32) -> &i32`，`x` 参数的生命周期会被自动赋给返回值 `&i32`，因此该函数等同于 `fn foo<'a>(x: &'a i32) -> &'a i32`
  c. **若存在多个输入生命周期，且其中一个是 `&self` 或 `&mut self`，则 `&self` 的生命周期被赋给所有的输出生命周期**
     i. 拥有 `&self` 形式的参数，说明该函数是一个 `方法`，该规则让方法的使用便利度大幅提升。

例子 1

```
1 fn first_word(s: &str) -> &str { // 实际项目中的手写代码
```

首先，我们手写的代码如上所示时，编译器会先应用第一条规则，为每个参数标注一个生命周期：

```
1 fn first_word<'a(s: &'a str) -> &str { // 编译器自动为参数添加生命周期
```

此时，第二条规则就可以进行应用，因为函数只有一个输入生命周期，因此该生命周期会被赋予所有的输出生命周期：

```
1 fn first_word<'a(s: &'a str) -> &'a str { // 编译器自动为返回值添加生命周期
```

此时，编译器为函数签名中的所有引用都自动添加了具体的生命周期，因此编译通过，且用户无需手动去标注生命周期，只要按照 `fn first_word(s: &str) -> &str {` 的形式写代码即可。

例子 2 再来看一个例子：

```rust
fn longest(x: &str, y: &str) -> &str { // 实际项目中的手写代码
```

首先，编译器会应用第一条规则，为每个参数都标注生命周期：

```rust
fn longest<'a, 'b(x: &'a str, y: &'b str) -> &str {
```

但是此时，第二条规则却无法被使用，因为输入生命周期有两个，第三条规则也不符合，因为它是函数，不是方法，因此没有 `&self` 参数。在套用所有规则后，编译器依然无法为返回值标注合适的生命周期，因此，编译器就会报错，提示我们需要手动标注生命周期：

```
error[E0106]: missing lifetime specifier
 --> src/main.rs:1:47
  |
1 | fn longest<'a, 'b>(x: &'a str, y: &'b str) -> &str {
  |                       -------     -------     ^ expected named lifetime
parameter
  |
  = help: this function's return type contains a borrowed value, but the
signature does not say whether it is borrowed from
x
 or
y
note: these named lifetimes are available to use
 --> src/main.rs:1:12
  |
1 | fn longest<'a, 'b>(x: &'a str, y: &'b str) -> &str {
  |            ^^  ^^
help: consider using one of the available lifetimes here
  |
1 | fn longest<'a, 'b>(x: &'a str, y: &'b str) -> &'lifetime str {
  |                                                +++++++++
```

不得不说，Rust 编译器真的很强大，还贴心的给我们提示了该如何修改，虽然……好像……。它的提示貌似不太准确。这里我们更希望参数和返回值都是 `'a` 生命周期。

- Impl 的生命周期

```rust
impl<'a: 'b, 'b> ImportantExcerpt<'a> {
    fn announce_and_return_part(&'a self, announcement: &'b str) -> &'b str {
```

```
        println!("Attention please: {}", announcement);
        self.part
    }
}
// 默认如果啥的不加，输出的生命周期就是 'a（根据第三条规则），如果我强行设置其为 'b，那么我要说明 'a 和 'b 的关系，这里 'a: 'b，说明 'a 比 'b 更长，也可以如下面写
impl<'a> ImportantExcerpt<'a> {
    fn announce_and_return_part<'b>(&'a self, announcement: &'b str) -> &'b str
    where
        'a: 'b,
    {
        println!("Attention please: {}", announcement);
        self.part
    }
}
```

- 实在不行可以用静态生命周期，表示全局存活：`let s: &'static str = "我没啥优点，就是活得久，嘿嘿";`

2.12 返回值和错误处理

- `Result<T, E>` 用于可恢复错误，`panic!` 用于不可恢复错误
- `unwrap` 对于 `Result<T, E>`：成功则返回值，失败则 `panic`，总之不进行任何错误处理
- `expect` 跟 `unwrap` 很像，也是遇到错误直接 `panic`，但是会带上自定义的错误提示信息，相当于重载了错误打印的函数
- `?` 如果结果是 `Ok(T)`，则把 `T` 赋值给 `f`，如果结果是 `Err(E)`，则返回该错误，所以 `?` 特别适合用来传播错误。

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

fn read_username_from_file() -> Result<String, io::Error> {
    let mut s = String::new();

    File::open("hello.txt")?.read_to_string(&mut s)?;

    Ok(s)
}
```

- `?` 操作符需要一个变量来承载正确的值，这个函数只会返回 `Some(&i32)` 或者 `None`，只有错误值能直接返回，正确的值不行，所以如果数组中存在 0 号元素，那么函数第二行使用 `?` 后的返回类型为 `&i32` 而不是 `Some(&i32)`。因此 `?` 只能用于以下形式：
  - `let v = xxx()?;`
  - `xxx()?.yyy()?;`

2.13 包和模块

- Package 是整个项目，分成二进制项目 binary 或者库项目 library，就是 `cargo new` 和 `cargo new --lib` 两种方法，前者可以运行后者不行，前者入口是 `src/main.rs` 后者是 `src/lib.rs`
- Package 中只能包含**一个**库(library)类型的包，但是可以包含**多个**二进制可执行类型的包。
- 包 Crate 是一个独立的可编译单元，它编译后会生成一个可执行文件或者一个库
- 典型架构

```
.
├── Cargo.toml
├── Cargo.lock
├── src
│   ├── main.rs
│   ├── lib.rs
│   └── bin
│       └── main1.rs
│       └── main2.rs
├── tests
│   └── some_integration_tests.rs
├── benches
│   └── simple_bench.rs
└── examples
    └── simple_example.rs
```

  - 唯一库包：`src/lib.rs`
  - 默认二进制包：`src/main.rs`，编译后生成的可执行文件与 `Package` 同名
  - 其余二进制包：`src/bin/main1.rs` 和 `src/bin/main2.rs`，它们会分别生成一个文件同名的二进制可执行文件
  - 集成测试文件：`tests` 目录下
  - 基准性能测试 `benchmark` 文件：`benches` 目录下
  - 项目示例：`examples` 目录下

- 模块 Module
  - mod的用户和相对绝对位置：https://course.rs/basic/crate-module/module.html
  - use的用法：https://course.rs/basic/crate-module/use.html

2.14 注释和文档

- https://course.rs/basic/comment.html

2.15 格式化输出

- 例子

```
println!("Hello");                 // => "Hello"
println!("Hello, {}!", "world");   // => "Hello, world!"
println!("The number is {}", 1);   // => "The number is 1"
println!("{:?}", (3, 4));          // => "(3, 4)"
println!("{value}", value=4);      // => "4"
println!("{} {}", 1, 2);           // => "1 2"
println!("{:04}", 42);             // => "0042" with leading zeros
```

- 常用函数
  - `print!` 将格式化文本输出到标准输出，不带换行符
  - `println!` 同上，但是在行的末尾添加换行符
  - `format!` 将格式化文本输出到 `String` 字符串
- 与 `{}` 类似，`{:?}` 也是占位符：
  - `{}` 适用于实现了 `std::fmt::Display` 特征的类型，用来以更优雅、更友好的方式格式化文本，例如展示给用户
  - `{:?}` 适用于实现了 `std::fmt::Debug` 特征的类型，用于调试场景，也可用 `{:#?}`
  - 其实两者的选择很简单，当你在写代码需要调试时，使用 `{:?}`，剩下的场景，选择 `{}`。
- 结构体要 `#[derive(Debug)]`
- 如要实现 Display

```
use std::fmt;
impl fmt::Display for Person {
    fn fmt(&self, f: &mut fmt::Formatter) -> fmt::Result {
        write!(
            f,
            "大佬在上，请受我一拜，小弟姓名{}，年芳{}，家里无田又无车，生活苦哈哈",
```

```
            self.name, self.age
        )
    }
}
```