腾讯课堂 零声教育 C/C++Linux服务器开发/高级架构师【零声教育】-学习视频教程-腾讯课堂 (qq.com)
https://rustwiki.org/

1 String和&str

String库地址：https://rustwiki.org/zh-CN/std/string/struct.String.html

1.1 基本概念

1.1.1 String （堆里）

Rust中的String是一个dynamic heap string type，也就是说它是存储在堆上面的，当你的字符串需要动态修改的时候应该使用String
创建一个空字符串：

> **let s = String::new();**

调用String的from函数根据字符串字面量来创建一个String实例：

> let s = String::from("Darren");

这里的双冒号（::）运算符允许我们调用置于String命名空间下面的特定函数（比如这里的new、from）。

1.1.2 &str（静态存储区、栈、 执行堆）

&str是一个**不可变的固定长度的字符串**，并且创建后无法再为其追加内容或更改内内容，可以存储在任意地方，

- 静态存储区： 有代表性的是字符串 字面量， &『static str 类型 的字符串被直接存储到已编译的可执行文件中， 随着程序一起加载启动 。
- 堆分配： 如果&str 类型 的字符串是通过堆 String 类型 的字符串**取切片生成 的**，则存储在堆上 。因为 String 类型 的字符串是堆分配的， &str 只 不过是其在堆上的切片 。
- 栈分配： 比如使用 str::from_utf8 方法， 就可以将栈分配的[u8; N] 数组转换为一个&str 字符串 ，举个例子

范例：ex1_1.rs

```
fn print_type_of<T>(_: &T) {
    println!("Type is: {}", std::any::type_name::<T>())
}
fn main() {
    let Str1: String = String::from("hello Str1");
    let str1: &str = "hello str1";
    let str2 = &Str1;
    print_type_of(&Str1); // Type is: alloc::string::String
    print_type_of(&str1); // Type is: &str
    print_type_of(&str2); // Type is: &alloc::string::String

    let i = 10;
    print_type_of(&i);
}
```

其中String和&str是可以相互转换的

1.2 String和&str互转

1.2.1 &str转String

ex1_2_1.rs

```
let str1: &str = "hello str1";
let cover_String1 = str1.to_string();
print_type_of(&cover_String1); // Type is: alloc::string::String

let cover_String2 = String::from(str1);
print_type_of(&cover_String2); // Type is: alloc::string::String

let cover_String3 = str1.to_owned();
print_type_of(&cover_String3); // Type is: alloc::string::String
```

1.2.2 String转&str

```
fn print_type_of<T>(_: &T) {
    println!("Type is: {}", std::any::type_name::<T>())
}
fn main() {
    let str1 = String::from("rust");
    let cover_str1 = &String::from(str1);
    print_type_of(&cover_str1); // Type is: &alloc::string::String

    let cover_str2_temp = String::from("hello");
    let cover_str2 = cover_str2_temp.as_str();
    print_type_of(&cover_str2); // Type is: &str
}
```

1.2.3 len()和capacity()

同时String和&str还有一个区别就是String有len()和capacity()，但str只有一个len()。

```
fn print_type_of<T>(_: &T) {
    println!("Type is: {}", std::any::type_name::<T>())
}
fn main() {
    let mut Str1: String = String::from("hello Str1");
    let str1: &str = "hello str1";
    let str2 = &Str1;

    println!("{}", Str1.len()); // 10
    println!("{}", Str1.capacity()); // 10
    println!("{}", str1.len()); // 10
    // println!("{}", str1.capacity()); // no method named `capacity` found for
reference `&str` in the current scope
}
```

1.3 字符串中的len和capacity的区别

范例ex1_3

```
fn main() {
    let mut story = String::new();
    let mut len = story.len();
    let mut capacity = story.capacity();
    println!("{}, {}", capacity, len);
    for _ in 0..5 {
        story.push_str("hello");
        len = story.len();
        capacity = story.capacity();
        println!("capacity is {} , len is {}", capacity, len);
    }
}
```

结论:

- len : 获取当前字符串对象的长度，随字符串变化而变化
- capacity : 获取当前字符串对象容器的长度，初始化*2 ， 默认长度是8

1.4 字符串索引

在其它语言中，使用索引的方式访问字符串的某个字符或者子串是很正常的行为，但是在 Rust 中就会报错:

```
let s1 = String::from("hello");
let h = s1[0];
```

该代码会产生如下错误:

```
3 |     let h = s1[0];
|             ^^^^^ `String` cannot be indexed by `{integer}`
|
= help: the trait `Index<{integer}>` is not implemented for `String`
```

1.4.1 深入字符串内部

字符串的底层的数据存储格式实际上是[u8]，一个字节数组。对于 let hello = String::from("Hola"); 这行代码来说，hello 的长度是 4 个字节，因为 "hola" 中的每个字母在 UTF-8 编码中仅占用 1 个字节，但是对于下面的代码呢?
ex1_4.rs

```
let hello = String::from("中国人");
```

如果问你该字符串多长，你可能会说 3，但是实际上是 9 个字节的长度，因为大部分常用汉字在 UTF-8 中的长度是 3 个字节，因此这种情况下对 hello 进行索引，访问 &hello[0] 没有任何意义，因为你取不到 中 这个字符，而是取到了这个字符三个字节中的第一个字节，这是一个非常奇怪而且难以理解的返回值。

1.4.2 字符串的不同表现形式

现在看一下用梵文写的字符串 “नमस्ते”, 它底层的字节数组如下形式：
[224, 164, 168, 224, 164, 174, 224, 164, 184, 224, 165, 141, 224, 164, 164, 224, 165, 135]
长度是 18 个字节，这也是计算机最终存储该字符串的形式。如果从字符的形式去看，则是：
['न', 'म', 'स', '्', 'त', 'े']
但是这种形式下，第四和六两个字母根本就不存在，没有任何意义，接着再从字母串的形式去看：
["न", "म", "स्", "ते"]
所以，可以看出来 Rust 提供了不同的字符串展现方式，这样程序可以挑选自己想要的方式去使用，而无需去管字符串从人类语言角度看长什么样。
还有一个原因导致了 **Rust 不允许去索引字符串**：因为索引操作，我们总是期望它的性能表现是 O(1)，然而对于 String 类型来说，无法保证这一点，因为 Rust 可能需要从 0 开始去遍历字符串来定位合法的字符。

1.5 String增删改查

由于 String 是可变字符串，下面介绍 Rust 字符串的修改，添加，删除等常用方法。

范例ex1_5.rs

1.5.1 追加 (Push)

在字符串尾部可以使用 push() 方法追加字符 char，也可以使用 push_str() 方法追加字符串字面量。这两个方法都是**在原有的字符串上追加，并不会返回新的字符串。**由于字符串追加操作要修改原来的字符串，则该字符串必须是可变的，即**字符串变量必须由 mut 关键字修饰。**

```
fn main() {
    let mut s = String::from("Hello ");
    s.push('r');
    println!("追加字符 push() -> {}", s);

    s.push_str("ust!");
    println!("追加字符串 push_str() -> {}", s);
}
```

1.5.2 插入 (Insert)

可以使用 insert() 方法插入单个字符 char，也可以使用 insert_str() 方法插入字符串字面量，与 push() 方法不同，这俩方法需要传入两个参数，第一个参数是字符（串）插入插入位置的索引，第二个参数是要插入的字符（串），索引从 0 开始计数，如果越界则会发生错误。由于字符串插入操作要**修改原来的字符串**，则该字符串必须是可变的，即**字符串变量必须由 mut 关键字修饰。**

```
fn main() {
    let mut s = String::from("Hello rust!");
    s.insert(5, ',');
    println!("插入字符 insert() -> {}", s);
    s.insert_str(6, " I like");
    println!("插入字符串 insert_str() -> {}", s);
}
```

1.5.3 替换 (Replace)

如果想要把字符串中的某个字符串替换成其它的字符串，那可以使用 replace() 方法。与替换有关的方法有三个。

1、replace

该方法可适用于 String 和 &str 类型。replace() 方法接收两个参数，第一个参数是要被替换的字符串，第二个参数是新的字符串。该方法会替换所有匹配到的字符串。**该方法是返回一个新的字符串，而不是操作原来的字符串。**

```
fn main() {
    let string_replace = String::from("I like rust. Learning rust is my
favorite!");
    let new_string_replace = string_replace.replace("rust", "RUST");
    dbg!(new_string_replace);
}
```

2、replacen

该方法可适用于 String 和 &str 类型。replacen() 方法接收三个参数，前两个参数与 replace() 方法一样，第三个参数则表示替换的个数。**该方法是返回一个新的字符串，而不是操作原来的字符串。**

```
fn main() {
    let string_replace = "I like rust. Learning rust is my favorite!";
    let new_string_replacen = string_replace.replacen("rust", "RUST", 1);
    dbg!(new_string_replacen);
}
```

3、replace_range

该方法仅适用于 String 类型。replace_range 接收两个参数，第一个参数是要替换字符串的范围（Range），第二个参数是新的字符串。**该方法是直接操作原来的字符串，不会返回新的字符串。该方法需要使用 mut 关键字修饰。**

```
fn main() {
    let mut string_replace_range = String::from("I like rust!");
    string_replace_range.replace_range(7..8, "R");
    dbg!(string_replace_range);
}
```

1.5.4 删除 (Delete)

与字符串删除相关的方法有 4 个，他们分别是 pop()，remove()，truncate()，clear()。这四个方法仅适用于 String 类型。

1、 pop —— 删除并返回字符串的最后一个字符

该方法是直接操作原来的字符串。但是存在返回值，其返回值是一个 Option 类型，如果字符串为空，则返回 None。

```rust
fn main() {
    let mut string_pop = String::from("rust pop 中文!");
    let p1 = string_pop.pop();
    let p2 = string_pop.pop();
    dbg!(p1);
    dbg!(p2);
    dbg!(string_pop);
}
```

2、 remove —— 删除并返回字符串中指定位置的字符
该方法是直接操作原来的字符串。但是存在返回值，其返回值是删除位置的字符串，只接收一个参数，表示该字符起始索引位置。remove() 方法是按照字节来处理字符串的，如果参数所给的位置不是合法的字符边界，则会发生错误。

```rust
fn main() {
    let mut string_remove = String::from("测试remove方法");
    println!(
        "string_remove 占 {} 个字节",
        std::mem::size_of_val(string_remove.as_str())
    );
    // 删除第一个汉字
    string_remove.remove(0);
    // 下面代码会发生错误
    // string_remove.remove(1);
    // 直接删除第二个汉字
    // string_remove.remove(3);
    dbg!(string_remove);
}
```

3、truncate —— 删除字符串中从指定位置开始到结尾的全部字符
该方法是直接操作原来的字符串。无返回值。该方法 truncate() 方法是按照字节来处理字符串的，如果参数所给的位置不是合法的字符边界，则会发生错误。

```rust
fn main() {
    let mut string_truncate = String::from("测试truncate");
    string_truncate.truncate(3);
    dbg!(string_truncate);
}
```

4、clear —— 清空字符串
该方法是直接操作原来的字符串。调用后，删除字符串中的所有字符，相当于 truncate() 方法参数为 0 的时候。

```rust
fn main() {
    let mut string_clear = String::from("string clear");
    string_clear.clear();
    dbg!(string_clear);
}
```

1.5.5 连接 (Catenate)

1、使用 + 或者 += 连接字符串
使用 + 或者 += 连接字符串，要求右边的参数必须为字符串的切片引用（Slice)类型。其实当调用 + 的操作符时，相当于调用了 std::string 标准库中的 add() 方法，这里 add() 方法的第二个参数是一个引用的类型。因此我们在使用 +， 必须传递切片引用类型。不能直接传递 String 类型。**+ 和 += 都是返回一个新的字符串。所以变量声明可以不需要 mut 关键字修饰。**

```
fn main() {
    let string_append = String::from("hello ");
    let string_rust = String::from("rust");
    // &string_rust会自动解引用为&str
    let result = string_append + &string_rust;
    let mut result = result + "!";
    result += "!!!";

    println!("连接字符串 + -> {}", result);
}
```

self 是 String 类型的字符串 s1，该函数说明，只能将 &str 类型的字符串切片添加到 String 类型的 s1 上，然后返回一个新的 String 类型，所以 let s3 = s1 + &s2; 就很好解释了，将 String 类型的 s1 与 &str 类型的 s2 进行相加，最终得到 String 类型的 s3。
由此可推，以下代码也是合法的：

```
#![allow(unused)]
fn main() {
let s1 = String::from("tic");
let s2 = String::from("tac");
let s3 = String::from("toe");

// String = String + &str + &str + &str + &str
let s = s1 + "-" + &s2 + "-" + &s3;
}
```

String + &str返回一个 String，然后再继续跟一个 &str 进行 + 操作，返回一个 String 类型，不断循环，最终生成一个 s，也是 String 类型。
s1 这个变量通过调用 add() 方法后，所有权被转移到 add() 方法里面， add() 方法调用后就被释放了，同时 s1 也被释放了。再使用 s1 就会发生错误。这里涉及到所有权转移（Move）的相关知识。
2、使用 format! 连接字符串
format! 这种方式适用于 String 和 &str 。format! 的用法与 print! 的用法类似，详见格式化输出。

```
fn main() {
    let s1 = "hello";
    let s2 = String::from("rust");
    let s = format!("{} {}!", s1, s2);
    println!("{}", s);
}
```

1.6 更多String的用法

https://rustwiki.org/zh-CN/std/string/struct.String.html

https://course.rs/basic/compound-type/string-slice.html

2 slice（切片）类型

切片允许我们引用集合中部分连续的元素序列，而不是引用整个集合。例如，字符串切片就是一个子字符串，数组切片就是一个子数组。

2.1 切片

切片（Slice）是对数据值的部分**引用**，是一种不持有所有权的数据类型。String（是 Rust 标准公共库提供的一种数据类型） 和 str（是 Rust 核心语言类型） 都支持切片，切片的结果是 &str 类型的数据。

- 规则
  - ..y 等价于 0..y
  - x.. 等价于位置 x 到数据结束
  - .. 等价于位置 0 到结束
  - 被部分引用，禁止更改其值
- 实例

```
 //ex2_1.rs
 // String的切片
 fn test1(){
    let s = String::from("broadcast");

    let part1 = &s[0..5];
    let part2 = &s[5..9];

    println!("{}={}+{}", s, part1, part2);
 }
 //被部分引用，禁止更改其值
 fn test2() {
    let mut s = String::from("china");
    let slice = &s[0..3]; // 不包括3
    // s.push_str("yes!"); // 错误
    println!("slice = {}", slice);
}

// 有一个快速的办法可以将 String 转换成 &str：
fn test3(){
    let s1 = String::from("hello");
    let s2 = &s1[..];
    println!("s1:{}, s2:{}", s1, s2);
}

fn test4()
{
    let arr = [1, 3, 5, 7, 9];
    let part = &arr[0..3];
    for i in part.iter() {
        println!("{}", i);
    }
}
fn main () {
   test1();
   test2();
   test3();
```

```
    test4();
}
```

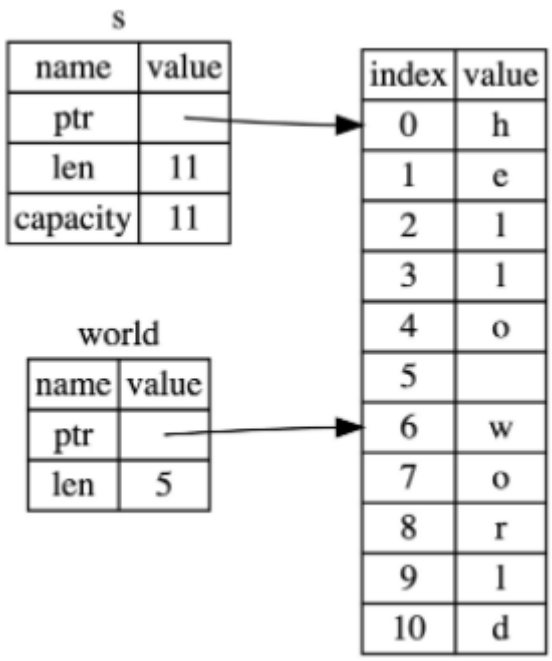
2.2 字符串切片内存布局

- 说明

字符串切片是指向字符串中一部分内容的引用。

- 形式: [开始索引..结束索引]
    - 开始索引就是切片起始位置的索引值
    - 结束索引是切片终止位置的下一个索引值
- 实例

```
//ex2_2.rs
fn main() {
    let mut s = String::from("hello world");
    let hello = &s[0..5];
    let word =  &s[6..11];
    println!("hello:{}, word:{}", hello, word);
}
```



2.3 多字节的字符中创建字符串切片问题

注意:

- 字符串切片的范围索引必须发生在有效的UTF-8字符边界内。
- 如果尝试从一个多字节的字符中创建字符串切片，程序会报错并退出。

实例

```rust
//ex2_3.rs

fn main() {
    let mut s = String::from("廖庆富");
    let liao = &s[0..3];    // 如果改成&s[0..2]
    let qing =  &s[3..];
    println!("liao:{}, qing:{}", liao, qing);
}
```

2.4 将字符串切片作为参数传递

- fn first_word(s: &string)-> &str {
- 有经验的Rust开发者会**采用&str作为参数类型**，因为这样就可以同时接收 string和 &str类型的参数。
- fn first_word(**s: &str**) -> &str {
    - 使用字符串切片，直接调用该函数
    - 使用String，可以创建一个完整的String切片来调用该函数
- 定义函数时使用字符串切片来代替字符串引用会使我们的API更加通用，且不会损失任何功能。

```rust
// ex2_4.rs
fn first_word(s: &str) -> &str {
    let bytes = s.as_bytes();

    for (i, &item) in bytes.iter().enumerate() {
        if item == b' ' {
            return &s[0..i];
        }
    }

    &s[..]
}

fn first_word_string(s: &String) -> &str {
    let bytes = s.as_bytes();

    for (i, &item) in bytes.iter().enumerate() {
        if item == b' ' {
            return &s[0..i];
        }
    }

    &s[..]
}

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

    // first_word works on slices of `String`s
    let word = first_word(&my_string[..]);

    let my_string_literal = "hello world";

    let word = first_word(&my_string_literal[..]);
```

```
    let word = first_word(my_string_literal);

    let word = first_word(my_string);
    let word = first_word(&my_string);
    let word = first_word_string(&my_string[..]);

    let my_string_literal = "hello world";

    let word = first_word_string(&my_string_literal[..]);
    let word = first_word_string(my_string_literal);
    let word = first_word_string(&my_string_literal);
}
```

3 vector

vector 是大小可变的数组。和 slice（切片）类似，它们的大小在编译时是未知的，但 它们可以随时扩大或缩小。一个 vector 使用 3 个词来表示：一个指向数据的指针，vector 的长度，还有它的容量。此容量指明了要为这个 vector 保留多少内存。vector 的长度 只要小于该容量，就可以随意增长；当需要超过这个阈值时，会给 vector 重新分配一段 更大的容量。

3.1 新建 vector 的两种方式

vector 是用泛型实现的，类似 C++ 的模板，需在尖括号里指明数据类型。不过大部分情况 Rust 都可以自动推导出来，无需程序员手动指明。

```
//  ex3_1.rs新建 vector 的两种方式
fn main() {
    //用宏新建，直接给了数据，Rust可以自行推算出类型
    let mut v = vec![1,2,3];
    //使用 new 函数，没有直接给数据，需要进行类型标注
    let mut vec: Vec<i32> = Vec::new();
}
```

3.2 更新 vector

对新建的 vector 可以使用 push 方法追加数据元素。可以使用 [index] 的方式修改已有的元素。

```
let mut vec: Vec<i32> = Vec::new();
vec.push(0);
vec.push(1);
vec.push(2);
vec[0] = 100;  // 支持索引的方式
```

3.3 读取 vector 的两种方式

1. 使用索引语法
2. 使用 get 方法

```
fn main() {
    let mut v = vec![1,2,3];

    v.push(3);
```

```
    v.push(4);
    v.push(5);
    for i in v.iter() {
        println!("{}", i);
    }

    v[0] = 999;
    for i in v.iter() {
        println!("{}", i);
    }

    let first = &v[0];  //1. 使用索引语法
    println!("index v[0] = {}", first);

    match v.get(0) {   //2. 使用 get 方法
        Some(value) => println!("get(0) = {}", value),
        None => println!("Error: No this value"), //如果请求的index不存在
    }

}
```

尝试编译以下两段程序，会发现第 2 段程序无法编译，因为其同时拥有不可变引用和一个可变引用：

```
fn main() {
    let mut v = vec![1,2,3];

    let first = &v[0];
    println!("v[0] = {}", first);

    v.push(4);
}

fn main() {
    let mut v = vec![1,2,3];

    let first = &v[0];
    v.push(4);
    println!("v[0] = {}", first);
}
```

3.4 Vector 的遍历

两者的区别：

```
for index in &v
for index in v
```

```
//ex3_4.rs
fn main() {
    let mut v = vec![1,2,3];

    for index in &mut v { //可变引用，遍历时可以修改其元素
        *index += 1;      //对元素进行加一操作，  * 号是解引用
    }
```

```
    for index in &v {  //不可变引用，元素只读
        println!("&v{}", index);
    }

    for index in v {  // 遍历时一般不用 for index in v {} ，因为这种方式会取得 vector 的所有权。
        println!("v{}", index);
    }

    v.push(4);  // 是否会报错
}
```

3.5 使用枚举使 vector 存储更多类型的数据

vector 只能储存相同类型的值，这很不方便使用。还好 Rust 中有枚举可以帮助 vector 来存储不同类型的数据。

```
// vector 只能储存相同类型的值，这很不方便使用。还好 Rust 中有枚举可以帮助 vector 来存储不同类型的数据。
enum Data {
    Int(i32),
    Double(f64),
    Str(String),
}
fn main() {
    let mut vec: Vec<Data> = Vec::new();
    vec.push(Data::Int(99));
    vec.push(Data::Double(99.99));
    vec.push(Data::Str(String::from("Hello Rust")));
    for index in &vec {
        match index {
            Data::Int(value) => println!("{}", value),
            Data::Double(value) => println!("{}", value),
            Data::Str(value) => println!("{}", value),
        }
    }

    let vv = vec![
        Data::Int(99),
        Data::Double(99.99),
        Data::Str(String::from("Hello Rust"))
    ];
    for index in &vv {
        match index {
            Data::Int(value) => println!("{}", value),
            Data::Double(value) => println!("{}", value),
            Data::Str(value) => println!("{}", value),
        }
    }
}
```

3.6 更多的vector函数

https://doc.rust-lang.org/stable/std/vec/struct.Vec.html
https://rustwiki.org/zh-CN/std/vec/struct.Vec.html

```rust
fn main() {
    // 迭代器可以被收集到 vector 中
    let collected_iterator: Vec<i32> = (0..10).collect();
    println!("Collected (0..10) into: {:?}", collected_iterator);

    // `vec!` 宏可用来初始化一个 vector
    let mut xs = vec![1i32, 2, 3];
    println!("Initial vector: {:?}", xs);

    // 在 vector 的尾部插入一个新的元素
    println!("Push 4 into the vector");
    xs.push(4);
    println!("Vector: {:?}", xs);

    // 报错！不可变 vector 不可增长
    collected_iterator.push(0);
    // 改正 ^ 将此行注释掉

    // `len` 方法获得一个 vector 的当前大小
    println!("Vector size: {}", xs.len());

    // 下标使用中括号表示（从 0 开始）
    println!("Second element: {}", xs[1]);

    // `pop` 移除 vector 的最后一个元素并将它返回
    println!("Pop last element: {:?}", xs.pop());

    // 超出下标范围将抛出一个 panic
    println!("Fourth element: {}", xs[3]);
    // 改正 ^ 注释掉此行

    // 迭代一个 `Vector` 很容易
    println!("Contents of xs:");
    for x in xs.iter() {
        println!("> {}", x);
    }

    // 可以在迭代 `Vector` 的同时，使用独立变量（`i`）来记录迭代次数
    for (i, x) in xs.iter().enumerate() {
        println!("In position {} we have value {}", i, x);
    }

    // 多亏了 `iter_mut`，可变的 `Vector` 在迭代的同时，其中每个值都能被修改
    for x in xs.iter_mut() {
        *x *= 3;
    }
    println!("Updated vector: {:?}", xs);
}
```

4 占位标记

序号遗漏了，抱歉。

5 hashmap

- 可以使用 new 创建一个空的 HashMap，并使用 insert 增加元素。
- 对于像 i32 这样的实现了 Copy trait 的类型，其值可以拷贝进hashmap。对于像 String 这样拥有所有权的值，其值将被移动而hashmap 会成为这些值的所有者。
- 可以使用与 vector 类似的方式来遍历hashmap 中的每一个键值对，也就是 for 循环。
- 可以通过 get 方法并提供对应的键来从hashmap 中获取值， get 返回 Option 。
- 如果我们插入了一个键值对，接着用相同的键插入一个不同的值，与这个键相关联的旧值将被替换。即 insert 可以用来覆盖一个值。
- 用 entry(key) 方法检查是否有值，没有就用 or_insert(value) 插入。
- 可以获取 HashMap 中某个值的可变引用，并对其进行修改。

https://rustwiki.org/zh-CN/std/collections/hash_map/struct.HashMap.html 更多的范例

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

fn main() {
    let mut hm = HashMap::new();
    let key = 3;
    let value = String::from("C");

    hm.insert(1, String::from("A"));
    hm.insert(2, String::from("B"));
    hm.insert(key, value);

    println!("key = {}\n", key);
    //println!("{}", value); //错误，已经丢失了所有权

    //用循环遍历HashMap
    for iter in &hm {
        println!("{}:{}", iter.0, iter.1);
    }
    println!("");

    //用 get(key) 方法获取 value
    match hm.get(&2) {
        Some(val) => println!("2:{}\n", val),
        None => println!("No this value"),
    };

    //覆盖一个值
    hm.insert(2, String::from("b"));

    //只在map中没有对应值时才插入
    hm.entry(1).or_insert(String::from("a")); //不会插入
    hm.entry(4).or_insert(String::from("D")); //会插入

    //用循环遍历HashMap，模式匹配
    for (k,v) in &hm {
        println!("{}:{}", k, v);
    }
    println!("");

    //获取key=4对应的值的可变引用，并修改其值
    let str = hm.entry(4).or_insert(String::from("D"));
    *str = String::from("d");

    //用循环遍历HashMap，模式匹配
    for (k,v) in &hm {
```

```
        println!("{}:{}", k, v);
    }

    let mut map = HashMap::new();//这里没有声明散列表的泛型，是因为 Rust 的自动判断类型机制。

    map.insert("color", "red");
    map.insert("size", "10 m^2");
    map.insert("a", "aba");

    // 在已经确定有某个键的情况下如果想直接修改对应的值，有更快的办法：
    if let Some(x) = map.get_mut("a") {
        *x = "b";
    }
    //当使用 insert 方法添加新的键值对的时候，如果已经存在相同的键，会直接覆盖对应的值。如果你想"安全地插入"，就是在确认当前不存在某个键时才执行的插入动作
    map.entry("color").or_insert("red1");

    println!("映射");
    //映射表支持迭代器
    for p in map.iter() {
        println!("{:?}", p);
    }

    println!("{}", map.get("color2").unwrap()); // panicked at 'called `Option::unwrap()` on a `None` value' 在错误处理讲解
}
```

6 hashset

请把 HashSet 当成这样一个 HashMap：我们只关心其中的键而非值（HashSet 实际上只是对 HashMap<T, ()> 的封装）。
你可能会问："这有什么意义呢？我完全可以将键存储到一个 Vec 中呀。"
HashSet 的独特之处在于，它保证了不会出现重复的元素。这是任何 set 集合类型（set collection）遵循的规定。HashSet 只是它的一个实现。（参见：BTreeSet）
如果插入的值已经存在于 HashSet 中（也就是，新值等于已存在的值，并且拥有相同 的散列值），那么新值将会替换旧的值。
如果你不想要一样东西出现多于一次，或者你要判断一样东西是不是已经存在，这种做法 就很有用了。
不过集合（set）可以做更多的事。
集合（set）拥有 4 种基本操作（下面的调用全部都返回一个迭代器）：

- **union**（并集）：获得两个集合中的所有元素（不含重复值）。
- **difference**（差集）：获取属于第一个集合而不属于第二集合的所有元素。
- **intersection**（交集）：获取同时属于两个集合的所有元素。
- **symmetric_difference**（对称差）：获取所有只属于其中一个集合，而不同时属于 两个集合的所有元素。

在下面的例子中尝试使用这些操作。

```
use std::collections::HashSet;

fn main() {
    let mut a: HashSet<i32> = vec!(1i32, 2, 3).into_iter().collect();
    let mut b: HashSet<i32> = vec!(2i32, 3, 4).into_iter().collect();
    // a 1 2 3, b 2 3 4,  union a/b : 1 2 3 4
```

```
    assert!(a.insert(4));
    assert!(a.contains(&4));

    // 如果值已经存在，那么 `HashSet::insert()` 返回 false。
    assert!(b.insert(4), "Value 4 is already in set B!");
    // 改正 ^ 将此行注释掉。

    b.insert(5);

    // 若一个集合（collection）的元素类型实现了 `Debug`，那么该集合也就实现了 `Debug`。
    // 这通常将元素打印成这样的格式 `[elem1, elem2, ...]
    println!("A: {:?}", a);
    println!("B: {:?}", b);

    // 乱序打印 [1, 2, 3, 4, 5]。
    println!("Union: {:?}", a.union(&b).collect::<Vec<&i32>>());

    // 这将会打印出 [1]
    println!("Difference: {:?}", a.difference(&b).collect::<Vec<&i32>>());

    // 乱序打印 [2, 3, 4]。
    println!("Intersection: {:?}", a.intersection(&b).collect::<Vec<&i32>>());

    // 打印 [1, 5]
    println!("Symmetric Difference: {:?}",
             a.symmetric_difference(&b).collect::<Vec<&i32>>());
}
```

7 错误处理

Rust具有极强的安全性，其中一部分原因就是因为在大部分时候在编译就会提示错误，并处理。
可能发生的错误分为两类：

1. 可恢复的错误；
2. 不可恢复的错误。

很多其他语言直接用Exception这种方法把两种错误柔道一起，**但是Rust本身没有异常的概念**，所以Rust的错误处理势必要知道错误类型：

- 可恢复的错误： 例如文件未找到，可以再次尝试查找文件。**Rust利用Result<T,E> 这样的泛型**
- 不可恢复的错误： Bug, 例如访问的索引超出范围。**会引发panic! 宏终止执行程序**：
    - 程序打印一个错误信息
    - 展开（unwind) , 清理调用栈 (Stack)。清理调用栈有两种方式： 1.（默认）Rust沿着调用栈往回走清理所有数据 2.Rust直接退出程序，交给OS清理数据（这种方法生成的二进制文件会更小）

```
// 我们在Cargp.toml的profile.release 中将panic设置为abort
[package]
name = "Panic_"
version = "0.1.0"
edition = "2022"
# See more keys and their definitions at https://doc.rust-
lang.org/cargo/reference/manifest.html
[profile.release]
panic = "abort"
[dependencies]
```

- 退出程序(Quit)

既然panic是一个宏，那我们就可以使用

```
fn main() {
    println!("Hello, world!");
    panic!("Something Wrong");
}
// 什么panic，在那个文职全给我们打印出来了
Hello, world!
thread 'main' panicked at 'Something Wrong', src\main.rs:3:5
```

7.1 Result 枚举与可恢复的错误

当 panic 发生时，程序一定会停止，但并不是所有的错误发生时都需要程序完全停止。
例如，我们要打开一个文件，但是文件不存在，这个错误发生后，也许我们想要的不是终止程序，而是创建这个文件。
对于这个场景，我们就要用到 Result<T, E>。
Result<T, E> 这是一个枚举类型，其定义如下：

```
enum Result<T, E> {
    Ok(T),
    Err(E),
}
```

T 和 E 都是泛型参数。T 代表成功时返回的 Ok 成员中的数据的类型，而 E 代表失败时返回的 Err 成员中的错误的类型。

Result<T, E> 通常作为一个可能出错的方法的返回值类型。如果方法执行成功了，会把成功的结果放在 T 中，返回 Ok(T)，如果失败了，会把错误信息放在 E 中，返回 Err(E)。

7.1.1 Result<T, E>和 File::open()

比如标准库中的打开文件函数 File::open() 返回值类型是 Result<T, E>。这个函数如果执行成功了，T 的类型就是 std::fs::File，这是一个文件句柄；如果失败了，E 的类型是 std::io::Error。

范例 ex7_1.rs

```
use std::fs::File;
// 标准库中的定义，我们无需写
enum Result<T,E>{
    Ok(T),      // 操作成功返回Ok数据的类型。
```

```
    Err(E),     // 失败返回错误的类型
}
//我们依然可以使用match 进行处理
// 一个打开文件的例子。
fn main() {
    println!("Hello, world!");
    let p = File::open("hello.txt");
    match p{
        Ok(_) => {println!("读取成功");},
        Err(e) =>{
                println!("读取失败，具体错误如下");
                println!("{}",e);
                // 使用println!("{:?}",e); 获取更详细的错误信息。
        },
    }
}
Hello, world!
读取失败，具体错误如下
系统找不到指定的文件。 (os error 2)
```

7.1.2 使用 match 为不同的结果执行不同的操作

如果是文件不存在，我们希望文件不存则创建一个文件，如果是其他错误或者创建文件仍然失败，再执行 panic!。
ex7_1_2.rs

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

fn main() {
    let f = File::open("hello.txt");

    let f = match f {
        Ok(file) => file,
        Err(error) => match error.kind() {
            ErrorKind::NotFound => match File::create("hello.txt") {
                Ok(fc) => {println!("create file ok"); fc},
                Err(e) => panic!("Problem creating the file: {:?}", e),
            },
            other_error => panic!("Problem opening the file: {:?}",
other_error),
        },
    };
}
```

在这段代码中，就可以实现我们想要的效果：

- 如果文件存在且正确打开了，就返回文件的句柄；
- 如果是除了文件不存在的其他错误，执行 panic!；
- 如果是文件不存在的错误，就新建一个文件。
    - 如果新建文件成功了，返回新建文件的句柄；
    - 如果新建文件遇到错误，执行 panic!。

7.1.3 使用闭包简化代码

下面这个不使用 match 的写法会更简单更清晰：

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

fn main() {
    let f = File::open("hello.txt").unwrap_or_else(|error| {
        if error.kind() == ErrorKind::NotFound {
            File::create("hello.txt").unwrap_or_else(|error| {
                panic!("Problem creating the file: {:?}", error);
            })
        } else {
            panic!("Problem opening the file: {:?}", error);
        }
    });
}
```

用闭包简化这段代码并不是最好的选择，这里只是提供一个思路，也许以后用得到。 闭包在我们后续的课时里面继续讲解。

7.1.4 失败时 panic 的简写: unwarp() 和 expect()

Result<T, E> 类型定义了很多辅助方法来处理各种情况，unwrap() 和 expect() 便是其中之二，
unwrap() 不带参数，expect()带参数。
unwrap() 和 expect() 用起来就和使用了 match 的类似。

- 如果 Result<T, E> 值是 Ok，那么 unwarp() 和 expect() 就会返回 T；
- 如果 Result<T, E> 值是 Err，那么 unwarp() 和 expect() 就会执行 panic!。

unwrap() 和 expect() 的区别是，expect() 可以自己指定 panic! 中的信息，**而 unwrap() 不可以。**

unwrap() 的示例代码：

```
use std::fs::File;

fn main() {
    let f = File::open("hello.txt").unwrap();
}
```

如果文件不存在：thread 'main' panicked at '**called** `Result::unwrap()` on an `Err` value: Os { code: 2, kind: NotFound, message: "No such file or directory" }', ex7_1_4unwrap.rs:4:37
note: run with `RUST_BACKTRACE=1` environment variable to display a backtrace

expect() 的示例代码：

```
use std::fs::File;

fn main() {
    let f = File::open("hello.txt").expect("Failed to open hello.txt");
}
```

如果文件不存在：thread 'main' panicked at '**Failed to open hello.txt:** Os { code: 2, kind: NotFound, message: "No such file or directory" }', ex7_1_4expect.rs:4:37
note: run with `RUST_BACKTRACE=1` environment variable to display a backtrace

使用了 expect() 的错误信息以我们指定的文本 “Failed to open hello.txt” 开始，我们将会更容易找到代码中的错误信息来自何处。如果在多处使用 unwrap，则需要花更多的时间来分析到底是哪一个 unwrap 造成了 panic，因为所有的 unwrap 调用都打印相同的信息。

7.2 传播错误

有时候我们封装了rust函数，通过提供API的方式供其他做业务的同事调用，在一些场景下，在出错位置可能无法判断是应该解决错误，还是 panic!，这种情况下，我们可以在代码中将此错误向上传递，把错误的处理交给调用者。这个叫做**传播错误。**

7.2.1 如何传播错误

ex7_2_1.rs 这段代码从文件中读取信息

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

fn read_username_from_file() -> Result<String, io::Error> {
    let f = File::open("hello.txt");

    let mut f = match f {
        Ok(file) => file,
        Err(e) => return Err(e),
    };

    let mut s = String::new();

    match f.read_to_string(&mut s) {
        Ok(_) => Ok(s),
        Err(e) => Err(e),
    }
}
```

上面的代码中，有两个 match 表达式， 第一个表达式是打开文件的分支，第二个表达式是读取内容的分支。

- 在第一个 match 表达式中，可以看到，如果打开文件成功了，就会返回句柄给 f，如果失败了，则显式调用 return，将错误返回给调用者，中止执行函数。
- 在第二个 match 表达式中也是类似的，如果读取内容成功了，就返回读取到的内容给调用者，否则返回错误给调用者。由于这个 match 是整个函数最后的表达式，所以无需显式调用 return。

7.2.2 传播错误的简写：？运算符

Rust 中提供了一个运算符 ? 用以简化传播错误的实现。
? 与 match 表达式做的事还是有些不同的：

- ? 运算符所使用的错误值被传递给了 from 函数，它定义于标准库的 From trait 中，**其用来将错误从一种类型转换为另一种类型。**
- 当 ? 运算符调用 from 函数时，收到的错误类型被转换为由当前函数返回类型所指定的错误类型。这在当函数返回单个错误类型来代表所有可能失败的方式时很有用，即使其可能会因很多种原因失

败。只要每一个错误类型都实现了 from 函数来定义如何将自身转换为返回的错误类型，? 运算符会自动处理这些转换。

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

fn read_username_from_file() -> Result<String, io::Error> {
    let mut f = File::open("hello.txt")?;
    let mut s = String::new();
    f.read_to_string(&mut s)?;
    Ok(s)
}
```

在第 6 行末尾，加了一个** ?**，这个的作用是，如果前面表达式（这个表达式返回值类型是 Result<T, E>）的值是 OK，? 就会返回其中的 T，如果前面的表达式的值是 Err，那么 ? 就会将其返回给调用者，这个返回类似执行了 return，会中止函数，返回错误给调用者。

第 8 行同理，如果读取内容成功了，**?** 会返回 Ok() （*read_to_string()* 执行成功的话里面的 *T* 就是 _），如果出错了，就会返回错误给调用者，中止函数。我们这里没有用变量来接收 *Ok()*，因为这里什么也没有，读取的内容是存在 s 里面的，所以函数最后返回 Ok(s)。

上面的代码还可以进一步简化，功能完全一样，使用链式调用的方法：

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

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)
}
```

ex7_2项目

```
// read_file.rs
pub mod file_control{
    use std::error::Error;
    use std::fs::File;
    use std::io;

    pub fn readfile(path:String) -> Result<File, io::Error> {
        return File::open(path);
    }
    pub  fn createfile(name:String) -> Result<File, io::Error> {
        return File::create(name);
    }
}

// main.rs
use std::fs::File;
mod read_file;
use read_file::file_control as file_ctrl;
```

```
fn main() {
    let find_file = file_ctrl::readfile("Good_Morning.txt".to_string());
    if find_file.is_err(){
        let make_file = file_ctrl::createfile("Good_Morning.txt".to_string());
        if make_file.is_err(){
            panic!("Something wrong with writing");
        }else{
            println!("File writing is ok {:?}",make_file);
        }
        let find_file = file_ctrl::readfile("Good_Morning.txt".to_string());
        if find_file.is_err() {
            panic!("Something wrong with reading");
        }else{
            println!("File reading is ok {:?}",find_file);
        }
    }
}
```

7.2.3 ? 使用场景

在main函数

```
use std::fs::File;

fn main() {
    let f = File::open("hello.txt")?;
}
```

main() 现在的返回值是 ()，这不符合使用 ? 的条件。
? 运算符只能用于返回值类型为 **Result** 或 Option 或其他实现了 FromResidual 的函数。

当你期望在不返回 Result 的函数中调用其他返回 Result 的函数时使用 ? 的话，有两种方法解决这个问题：

1. 将函数返回值类型修改为 Result<T, E>，如果没有其它限制阻止你这么做的话；
2. 通过合适的方法（例如使用 match 或另一个 Result）来处理 Result<T, E>。

这个后续用到再展开讨论。

7.3 panic! 宏与不可恢复的错误

```
fn main() {
    let v = vec![1, 2, 3];

    v[99];
}
```

在 main 中触发了 panic，是数组越界的问题，现在我们再试试设置环境变量，RUST_BACKTRACE=1 cargo run

虽然我们自己编写的代码中没有调用 panic!，但是在上面的调用栈信息中还是可以看到触发了 panic，这也就是说明 panic 不一定发生在我们编写的代码中。
上面的调用栈信息中，第 1 条，也就是栈顶的那一条，是实际真正触发了 panic! 的位置，当然这个 panic! 不是我们写的。

7.4 Panic使用时机

那既然有了Result, 是不是就不需要Panic了呢？答案是否定的。panic的使用时机与规则如下：

- 定义一个可能失败的函数时，优先考虑 Result
- 编写示例，原形代码，测试可以使用panic (unwrap, except)
- 当代码最终可能处于损坏状态的时候（某些假设，保证，约定或不可变性被打破），最好使用Panic。eg. 传入无意义的参数值， 调用外部不可控代码。

8 rust泛型

C/C++、**Rust都是强类型语言**，在对数据进行处理时，必须明确数据的数据类型。但是很多时候，比如链表这种数据结构，我们可以是整型数据的链表，也可以是其他类型，我们可能就会写出重复的代码，仅仅是数据类型不同而已。还有比如说排序，算法里处理数据，也是同样的道理。
Rust标准库中定义了很多泛型类型，包括Option,Vec, HashMap<K,V>及Box。

8.1 函数中的泛型

必须在 impl 后面声明 T ，这样就可以在 Point 上实现的方法中使用它了。在**impl 之后声明泛型 T** ，这样 Rust 就知道 Point 的尖括号中的类型是泛型而不是具体类型。

```
struct Point<T> {
  x: T,
  y: T,
}
impl<T> Point<T> {
  fn x(&self) -> &T {
    &self.x
  }
}
fn main() {
  let p = Point { x: 5, y: 10 };
  println!("p.x = {}", p.x());
}
```

8.1.1 不使用泛型的例子

ex8_1_1.rs

```
//不使用泛型
//针对于整型数据
fn findmax_int(list : &[i32]) -> i32 {
    let mut max_int = list[0];
    for &i in list.iter() {
        if i > max_int {
            max_int = i;
        }
    }
    max_int
}

//针对于char数据
```

```
fn findmax_char(list : &[char]) -> char {
    let mut max_char = list[0];
    for &i in list.iter() {
        if i > max_char {
            max_char = i;
        }
    }
    max_char
}

fn main() {
    let v_int = vec![2, 4, 1, 5, 7, 3];
    println!("max_int: {}", findmax_int(&v_int));
    let v_char = vec!['A', 'C', 'G', 'B', 'F'];
    println!("max_char: {}", findmax_char(&v_char));
}
```

8.1.2 使用泛型的例子

涉及的trait在第9章节讲解。

```
// 两个操作需要类型具有Copy语义。因此加上对类型的Copy语义要求
// PartialOrd/Copy都属于trait
fn find_max<T : PartialOrd + Copy> (list : &[T]) -> T {
    let mut max = list[0];
    for &i in list.iter() {
        if i > max {
            max = i;
        }
    }
    max
}

fn main() {
    let v_int = vec![2, 4, 1, 5, 7, 3];
    println!("max_int: {}", find_max(&v_int));
    let v_char = vec!['A', 'C', 'G', 'B', 'F'];
    println!("max_char: {}", find_max(&v_char));
}
```

8.2 数据结构中的泛型

Option泛型

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

Option 是一个拥有泛型 T 的枚举，它有两个成员：Some ，它存放了一个类型 T 的值，和不存在任何值的 None 。
枚举也可以拥有多个泛型类型：

```
enum Result<T, E> {
  Ok(T),
  Err(E),
}
```

Result 枚举有两个泛型类型， T 和 E 。Result 有两个成员：Ok ，它存放一个类型 T的值，而 Err 则存放一个类型 E 的值。这个定义使得 Result 枚举能很方便的表达任何可能成功（返回 T 类型的值）也可能失败（返回 E 类型的值）的操作。

结构体：

```
struct A<T> {
 data1 : T,
 data2 : T,
 data3 : i32
}
```

声明了泛型就必须使用：

```
struct A<T> {
 data : i32
}
```

在结构体方法中泛型实例ex8_2_1.rs：

```
struct A<T> {
    x : T,
    y : T
}

impl <T> A<T> {
    fn get_x(&self) -> &T {
        &self.x
    }

    fn get_y(&self) -> &T {
        &self.y
    }
}


fn main() {
    let a = A { x : 1, y : 2};
    println!("a.x: {}, a.y: {}",a.get_x(), a.get_y());
    let b = A { x : 'a', y : 's'};
    println!("b.x: {}, b.y: {}",b.get_x(), b.get_y());
}
```

ex8_2_2.rs

```
struct A<, S> {
    x : T,
```

```
    y : S
}

impl <T, S> A<T, S> {
    //通过两个不同的A来创建一个新的A
    fn crete_new_a<M, N>(self, other : A<M, N>) -> A<T, N> {
        A {
            x : self.x,
            y : other.y
        }
    }
}

fn main() {
    let a1 = A { x : 's', y : 2.2 };
    let a2 = A { x : 3.5, y : 5 };
    let a3 = a1.crete_new_a(a2);
    println!("a3.x: {}, a3.y: {}", a3.x, a3.y);
}
```

8.3 泛型代码的性能

- Rust 实现了泛型，使得使用泛型类型参数的代码相比使用具体类型并没有任何速度上的损失。
- Rust 通过在编译时进行泛型代码的 单态化（monomorphization）来保证效率。单态化是一个通过填充编译时使用的具体类型，将通用代码转换为特定代码的过程。编译器寻找所有泛型代码被调用的位置并使用泛型代码针对具体类型生成代码。
- 我们可以使用泛型来编写不重复的代码，而 Rust 将会为每一个实例编译其特定类型的代码。这意味着在使用**泛型时没有运行时开销**；当代码运行，它的执行效率就跟好像手写每个具体定义的重复代码一样。这正是 Rust 泛型在运行时极其高效的原因。

9 trait

Trait是Rust中的核心，是实现Rust抽象功能的关键，在一些其他语言中，实现抽象功能的特性是接口，比如c++中的虚基类，Golang，Java中的interface，都可以来做接口的抽象，但在Rust中，Trait并不仅仅只是用作抽象，它对于类型系统有着很大的作用。

9.1 trait的定义和实现

ex9_1.rs

```
//定义trait
pub trait GetInfo {
    fn get_name(&self) -> &String;
    fn get_index(&self) -> i32;
}

//定义学生结构体
pub struct Student {
    pub name : String,
    pub index : i32,
    is_homework_completed : bool
}

pub struct Teacher {
    pub name : String,
```

```
    pub index : i32,
    pub sex : String
}

//实现trait
impl GetInfo for Student {
    fn get_name(&self) -> &String {
        &self.name
    }

    fn get_index(&self) -> i32 {
        self.index
    }
}

impl GetInfo for Teacher {
    fn get_name(&self) -> &String {
        &self.name
    }

    fn get_index(&self) -> i32 {
        self.index
    }
}

fn main() {
    let stu = Student { name: String::from("二狗"), index: 32 ,
is_homework_completed : false};
    println!("stu: {}, {}", stu.get_name(), stu.get_index());
    let t = Teacher { name: String::from("小芳"), index: 5 , sex :
String::from("male")};
    println!("t: {}, {}", t.get_name(), t.get_index());
}
```

在上面例子中，定义了一个trait，并分别为Student和Teacher实现了这个trait。

9.2 trait的应用举例

ex9_2.rs

```
//定义trait
pub trait GetInfo {
    fn get_name(&self) -> &String;
    fn get_index(&self) -> i32;
}

//定义学生结构体
pub struct Student {
    pub name : String,
    pub index : i32,
    Is_Homework_completed : bool
}

pub struct Teacher {
    pub name : String,
    pub index : i32,
    pub sex : String
```

```
}

//实现trait
impl GetInfo for Student {
    fn get_name(&self) -> &String {
        &self.name
    }

    fn get_index(&self) -> i32 {
        self.index
    }
}

//此处我们把Teacher的实现注视掉
// impl GetInfo for Teacher {
//     fn get_name(&self) -> &String {
//         &self.name
//     }

//     fn get_index(&self) -> i32 {
//         self.index
//     }
// }

fn Print_info(item : impl GetInfo) {
    println!("name: {}", item.get_name());
    println!("index: {}", item.get_index());
}

fn main() {
    let stu = Student { name: String::from("二狗"), index: 32 ,
Is_Homework_completed : false};
    Print_info(stu);
    let t = Teacher { name: String::from("小芳"), index: 5 , sex :
String::from("male")};
    Print_info(t); // 能否执行
}
```

9.3 trait bound

上面是通过impl trait进行约束。
还可以通过trait bound来进行约束。
什么是trait bound呢，在第8章节 **泛型**中的介绍的

```
fn find_max<T : PartialOrd + Copy> (list : &[T]) -> T {
    let mut max = list[0];
    for &i in list.iter() {
        if i > max {
            max = i;
        }
    }
    max
}
```

这里的<T : PartialOrd + Copy>就是trait bound。我们看到，我们绑定了两个trait。那么如果通过impl方式，实现方式如下：

```
pub fn haha(item1: impl trait1, item2: impl trait2) {
```

这里的item1和item2是两个不同的类型，那么如果我需要是相同的类型呢，那就只能trait bound才能做到。

```
fn find_max<T : PartialOrd + Copy> (list : &[T]) -> T {
```

9.4 trait的默认实现方式

trait是可以有默认实现的。

```
//定义学生结构体
pub struct Student {
    pub name : String,
    pub index : i32,
    is_homework_completed : bool
}

pub struct Teacher {
    pub name : String,
    pub index : i32,
    pub sex : String
}

trait Print_school {
    fn print_info(&self) {
        println!("默认trait");
        // println!("默认trait! name:{}, index:{}", self.name, self.index);
        // println!("默认trait! name:{}, index:", self.name);
    }
}

impl Print_school for Student {
}

impl Print_school for Teacher {
    fn print_info(&self) {
        println!("Teacher trait! name:{}, index:{}", self.name, self.index);
    }
}

fn main() {
    let stu = Student { name: String::from("小李"), index: 32 ,
is_homework_completed : false};
    let tea = Teacher {name : "隔壁老王".to_string(), index : 8, sex :
String::from("男")};
    stu.print_info();
    tea.print_info();
}
```

9.5 扩展方法

我们还可以通过trait为其他的类型添加成员方法，哪怕这个类型不是我们自己写的。
例如我们对i32添加一个方法：
ex9_5.rs

```
trait Triple {
    fn triple(&self) -> i32;
}

impl Triple for i32 {
    fn triple(&self) -> i32 {
        *self * 3
    }
}

fn main() {
    let a = 10;
    println!("res: {}", a.triple());
}
```

9.6 将 trait 作为返回值

用了 impl [trait] 语法，函数的返回值只要是一个实现了 Summary 的类型就可以

```
fn returns_summarizable() -> impl Summary {
    Tweet {
        username: String::from("horse_ebooks"),
        content: String::from("of course, as you probably already know,
people"),
        reply: false,
        retweet: false,
    }
}
```

9.7 更多trait的方式

9.7.1 通过 + 符号为参数类型指定多个 trait

假设我们希望参数 item 的类型是同时实现了 Summary 和 Display 两个 trait 的类型，那么两种语法的写法分别如下：

impl Summary for Student {

}

impl Display for Student {

}

```
// impl [trait]
pub fn notify(item: impl Summary + Display) {
```

```rust
// trait bound
pub fn notify<T: Summary + Display>(item: T) {
```

9.7.2 通过 where 关键字简化 trait bound

假设函数参数有多个，每个参数类型要求实现的 tarit 还不一样，那么像上面那样都写在函数声明里就显得太冗长

```rust
fn some_function<T: Display + Clone, U: Clone + Debug>(t: T, u: U) -> i32 {
```

T 是实现了 Display 和 Clone 的类型，U 是实现了 Clone 和 Debug 的类型。
使用 where 关键字的简化版写法，注意写法格式：

```rust
fn some_function<T, U>(t: T, u: U) -> i32
    where T: Display + Clone,
          U: Clone + Debug
{
```

参考

https://gukaifeng.cn/categories/%E7%BC%96%E7%A8%8B%E8%AF%AD%E8%A8%80%E5%9F%BA%E7%A1%80/

https://doc.rust-lang.org/stable/std/vec/

https://rust-book.junmajinlong.com/

https://rustwiki.org/