介绍了定义域，并没有介绍定义方法的方法。这是因为Rust的方法定义更像是动态语言，并不需要写在结构体定义时。

**为**结构体定义方法

我们来复习下上节介绍的结构体的例子：

**struct Complex {**

**real : i32,**

**imagine : i32**

**}**

**let mut c1 = Complex{real :0, imagine: 1};**

**println!("{}+{}i",c1.real,c1.imagine);**

下面我们想给它增加一个计算加法的方法，我们不用修改Complex的定义，而是新增一段impl Complex就可以了：

**impl Complex{**

**fn add(&mut self, c2 : Complex) {**

**self.real += c2.real;**

**self.imagine += c2.imagine;**

**}**

**}**

与定义在结构体或者类中的方法不同，在impl中定义的方法需要显示指定self。

调用的方法就如大家想像的那样：

**let c3 = Complex{real: 10, imagine: 0};**

**c1.add(c3);**

**println!("{}+{}i",c1.real,c1.imagine);**

好，我们再实现一个减法，不需要改动加法的部分，新增即可：

**impl Complex{**

**fn sub(&mut self, c2 : Complex) {**

**self.real -= c2.real;**

**self.imagine -= c2.imagine;**

**}**

**}**

**let c4 = Complex{real: 5, imagine: -1};**

**c1.sub(c4);**

**println!("{}+{}i",c1.real,c1.imagine);**

**trait**

有了方法定义之后，我们自然而然的想法就是需要一个类似于Java中接口的东西。根据依赖倒置原则，我们应该面向接口编程而非面向实现。在Rust语言中，我们使用trait来实现类似于接口的功能。

trait类似接口，就是一些函数声明的组合，我们来将前面的add和sub实现成一个trait:

**trait AddSub<T> {**

**fn add(&mut self, c2: T);**

**fn sub(&mut self, c2: T);**

**}**

<T>是泛型，大家在C++和Java中已经比较熟悉了。

实现trait仍然使用上面学习的impl语句，不过要加上"for 结构体名"

**impl AddSub<Complex> for Complex{**

**fn add(&mut self, c2 : Complex) {**

**self.real += c2.real;**

**self.imagine += c2.imagine;**

**}**

**fn sub(&mut self, c2 : Complex) {**

**self.real -= c2.real;**

**self.imagine -= c2.imagine;**

**}**

**}**

虽然实现变成了实现trait，但是对我们之前写的调用语句完全没有影响。

trait是可以继承的，比如我们想在AddSub接口基础上增加一个mul方法，我们可以用":"来继承AddSub：

trait AddSubMul<T> : AddSub<T> {

fn mul(&mut self, c2: T);

}

**Display和Debug trait**

学习了trait之后我们就可以面向接口编程了。  
首先我们从println!说起，我们之前打印变量值的时候，使用的“{}”格式，其实调用的是对象对Display trait的实现；而"{:?}"是在调用Debug trait的实现。

比如我们如果这么写：

println!("c1={}",c1);

直接导致编译失败：

`Complex` doesn't implement `Display` (required by {})

这个Display的定义如下：

pub trait Display {

fn fmt(&self, f: &mut Formatter<'\_>) -> Result<(), Error>;

}

我们来实现下这个trait，也就是这个fmt方法就好了：

impl std::fmt::Display for Complex {

fn fmt(&self, f: &mut std::fmt::Formatter<'\_>) -> std::fmt::Result {

write!(f, "({}+{}i)", self.real, self.imagine)

}

}

同样，对于{:?}，需要实现Debug trait。定义如下：

pub trait Debug {

fn fmt(&self, f: &mut Formatter<'\_>) -> Result<(), Error>;

}

细心的同学发现，这跟Display的fmt定义一模一样。

所以，我们的实现方法跟Display一样，改个名就可以：

impl std::fmt::Debug for Complex {

fn fmt(&self, f: &mut std::fmt::Formatter<'\_>) -> std::fmt::Result {

write!(f, "{}+{}i", self.real, self.imagine)

}

}

不过，如果每个结构体都去实现Debug trait的话效率是不高。如果它的全部字段都实现了Debug trait，那么我们就可以偷个懒，让Rust帮我们自动实现它，方法是写一个#[derive(Debug)]：

#[derive(Debug)]

struct Complex {

real : i32,

imagine : i32

}

println!("c1={:?}",c1);

输出结果如下：

c1=Complex { real: 5, imagine: 2 }

**小结**

这一节我们学习了用impl给结构体实现方法，方法组合成trait，trait可以继承，以及Display和Debug这两个最常用的trait。