Struct

定义与实例化一个Struct

- 首先，很直观的定义一个struct：

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

- 在实例化一个struct的时候，我们可以设置它为mutable的，但这个mutable是对于整个struct的，而不能仅针对其中一个元素

```
fn main() {
    let mut user1 = User{
        email: String::from("119010249@link.cuhk.edu.cn"),
        username: String::from("qpr"),
        active: true,
        sign_in_count: 1,
    };

    user1.email = String::from("1905873179@qq.com");
}
```

- 我们也可以用一个函数，来简化实例化的输入：

```
fn build_user(email: String, username: String) -> User{
    User{
        email,
        username,
        active: true,
        sign_in_count: 1,
    }
}
```

- 我们也可以用另一个struct的值来实例化一个struct：

```
let user2 = User{
    email: String::from("11111111@qq.com"),
    ..user1
};
```

    - 这里很神奇地做到了，user2有自己地email，但其他field的值完全和user1相同。
    - 但！非常重要的是，这里相当与是copy了user1的除了email之外的值。对于username这个属性，它是一个&String, 这意味着ownship发生了转换。所以这样赋值以后，user1就不能invalid了。

    - 除非，user2的email和username这两个属性都用的自己的值。active和sign_in_count这两个不用担心ownship转换,因为他们都是u64或bool

---

Tuple Struct

- Tuple Struct与Tuple不同的是：tuple是一个变量，而Tuple Struct是一个Struct，其是可以声明+实例化的。

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

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

Unit-Like Structs

- 甚至可以定义并实例化没有任何fields的struct（可能之后会有奇用）：

```
struct AlwaysEqual;

let subject = AlwaysEqual;
```

一个应用Struct的例子

- 其实应用Struct很多情况下是为了代码的可读性。比如，如果不用struct而用一个tuple来表示一个数据结构，我们就必须提前知道tuple的每个index代表的什么，这样非常不直观。但struct就可以避免这个问题。
- 一个简单的例子：

```
struct Rectangle{
    width: u32,
    height: u32,
}

fn main() {
    let rect1 = Rectangle{
        width: 30,
        height: 50,
    };

    println!(
        "The area of the rectangle is {} square pixels.",
        area(&rect1)
    );
}

fn area(rectangle: &Rectangle) -> u32{
    rectangle.width * rectangle.height
}
```

    - 注意这里的`area(&rect1)`。我们只想borrow这个rect1，所以这样用。

Struct debug

- 我们是否可以尝试打印出来一个struct呢？看下面的代码：

```
fn main() {
    let rect1 = Rectangle{
        width: 30,
        height: 50,
    };

    println!("rect1 is {}", rect1);
}
```

会产生以下的报错：

```
error[E0277]: `Rectangle` doesn't implement `std::fmt::Display`
  --> src/main.rs:17:29
   |
17 |     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` (in
Nightly builds, run with -Z macro-backtrace for more info)
```

  - 这里，我们可以关注报错信息中几个有趣的点：
    - `= help: the trait std::fmt::Displayis not implemented for Rectangle`：实际上我们用println!这个macro的时候，它的打印是以`Display`为格式的。对于primitive type，它们都implement了这个格式。但对于自己定义的struct而言并没有。所以我们需要使用其他的打印方法
    - `= note: in format strings you may be able to use {:?} (or {:#?} for pretty-print) instead`：报错信息给了我们一个其他的解决方案：`{:?}`。我们尝试使用这个
- 将打印部分改成：

```
println!("rect1 is {:?}", rect1);
```

报错信息如下：

```
= help: the trait `Debug` is not implemented for `Rectangle`
   = note: add `#[derive(Debug)]` to `Rectangle` or manually `impl Debug for
Rectangle`
```

  - Rust提示`Debug`这个trait并没有在Rectangle这个struct中实现（什么是`strait`应该以后会讲）。所以如果要使用`{:?}`的话，我们要做的是给Rectangle加上Debug这个strait。
- 加上`#[derive(Debug)]`：

```
#[derive(Debug)]
struct Rectangle{
```

```
    width: u32,
    height: u32,
}

fn main() {
    let rect1 = Rectangle{
        width: 30,
        height: 50,
    };

    println!("rect1 is {:?}", rect1);
}
```

- 成功打印：

```
rect1 is Rectangle { width: 30, height: 50 }
```

- 另外，我们还可以尝试下之前报错信息里提到的`{:#?}`：

```
println!("rect1 is {:#?}", rect1);
```

打印结果为：

```
rect1 is Rectangle {
    width: 30,
    height: 50,
}
```

- 这个格式上就更加舒服了。

---

dgb!

- 另外，我们也可以不用`println!`这个macro而使用`dbg!`这个macro(一听名字就很像是为调试而生的macro)。
  - 注意以下`println!`和`dbg!`的区别：
    - `println!`是将信息打印到standard output console stream(stdout)
    - `dgb!`则是将信息打印到standart error console stream(stderr)
- 用例：

```
fn main() {
    let scale = 2;
    let rect2 = Rectangle{
        width: dbg!(30 * scale),
        height: 50,
    };

    dbg!(rect2);
}
```

输出：

```
[src/main.rs:23] 30 * scale = 60
[src/main.rs:27] rect2 = Rectangle {
    width: 60,
    height: 50,
}
```

- 非常方便！`dbg!`还打印出了文件名和打印的位置在哪一行
- 注意，`width: dbg!(30 * scale)`. 可以正常用，因为`dbg!(30 * scale)`返回了30*scale这个expression的值的ownership
- 注意，要想让`dbg!(rect2)`正常工作，还是需要在Rectangle这个struct上方加上`#[derive(Debug)]`的

Method

- 我们也可以为struct实现method。method和function的相同在于，它们都是接受参数，然后return value。但method不同的点在于，methods are defined within the context of a struct(or an enum, a trait)。并且method获取参数的方法永远是self
- 看以下的代码：

```
#[derive(Debug)]
struct Rectangle{
    width: u32,
    height: u32,
}

impl Rectangle{
    fn area(&self) -> u32{
        self.width * self.height
    }
}

fn main() {
    let rect1 = Rectangle{
        width: 30,
        height: 50,
    };

    println!(
        "The area of rect1 is {} square pixels",
        rect1.area()
    );
}
```

  - 可以看到，rust中实现methods主要是通过`impl struct_name`然后里面再包裹`fn`的方式。并且里面`fn`取参的方法都是通过`self`
  - 另外，我们可以看到`fn`的参数：`fn area(&self) -> u32`：`&self`是`self: &Self`的简写。用上`&`说明我们是borrow这个ownship的。（绝大多数method中的函数都是需要borrow这个self的）
- 另外，method中的`fn`除了&self这个参数还能携带其他的参数。比如以下的例子：

```
impl Rectangle{
    fn can_hold(&self, another_rect: &Rectangle) -> bool{
```

```
        return self.width > another_rect.width && self.height >
another_rect.height;
    }
}

fn main() {
    let rect1 = Rectangle{
        width: 30,
        height: 50,
    };

    let rect2 = Rectangle{
        width: 10,
        height: 40,
    };

    let rect3 = Rectangle{
        width: 60,
        height: 45,
    };

    println!("Can rect1 hold rect2? {}", rect1.can_hold(&rect2));
    println!("Can rect1 hold rect3? {}", rect1.can_hold(&rect3));
}
```

- `can_hold`是比较一个Rectangle能否放得下另一个Rectangle

output:

```
Can rect1 hold rect2? true
Can rect1 hold rect3? false
```

Where is the -> Operator

- 我们知道，再C++中是有`->`这个operator的，它是`*`和`.`operator的结合。但Rust中并没有`->`，因为rust有一个特性叫**automatic referencing**. 它会自动给objects加上`&`, `&mut`, 或者`*`。所以以下的两个是等效的:

```
p1.distance(&p2);
(&p1).distance(&p2)
```

Association Function

- 所有再`impl`中实现得`fn`都称之为associated functions.
- 我们也可以实现不带`&self`的associated function。 比如下面的例子:

```
impl Rectangle{
    fn square(size: u32) -> Rectangle{
        Rectangle{
            width: size,
            height: size,
        }
    }
```

```
}

fn main() {

    let sq = Rectangle::square(3);
    dbg!(sq);
}
```

输出：

```
[src/main.rs:50] sq = Rectangle {
    width: 3,
    height: 3,
}
```

- 特别注意的是，对于这类不带`&self`的associated functions，我们调用的方法是：`let sq = Rectangle::sqaure(3)`。
  - `::`是跟namespace有关的。后面的章节会讲到