Defining Modules

因为The Rust Programming Language中这一块讲的很短，我在网上找到了一个非常棒的资料:https://blog.csdn.net/qq_41359051/article/details/108135856

- modules是用来组织一个crate中的代码的。我个人将其理解为文件夹，分类各种各样的功能。
- 举一个lib的例子,创建一个restaurant的package：

```
cargo new --lib restaurant
```

- 在里面，我们写餐厅的modules

```
mod front_of_house{
    mod hosting{
        fn add_to_waitlist(){}

        fn seat_at_table(){}

        fn give_two() -> i32{
            2
        }
    }

    mod serving{
        fn take_over(){}

        fn serve_order() {}

        fn take_payment() {}
    }
}
#[cfg(test)]
mod tests {
    #[test]
    fn it_works() {
        let result = 2 + 2;
        assert_eq!(result, 4);
    }
}
```

    - 这样，我们就可以比较好的组织我们的代码了：将整个餐厅抽象为一个crate， 然后在这个crate中将前厅抽象为一个`module`， 然后又将前厅的两个服务抽象为两个modules，后来再在这些modules中实现其服务的功能。
- 如果我们想用modules中定义的函数的话，就需要用到`path`。我们通过这个path将modules中的功能引入到scope中。（这就很像文件系统中的路径）比如，如果我们想用give_two()这个函数，就需要：

```
#[cfg(test)]
mod tests {
    #[test]
    fn it_works() {
        let result = crate::front_of_house::hosting::give_two();
        assert_eq!(result, 2);
    }
}
```

- 这样写其实还会报错，因为module是默认private的，这样用的话，就需要将用到的module和fn改成public性质：

```
mod front_of_house{
    pub mod hosting{
        fn add_to_waitlist(){}

        fn seat_at_table(){}

        pub fn give_two() -> i32{
            2
        }
    }

    mod serving{
        fn take_over(){}

        fn serve_order() {}

        fn take_payment() {}
    }
}
```

- 这里private和public的规则是：子module可以调用父module中的private content，但父module就不能调用子module中的private content。

相对路径

- 还是上面的例子，它用到的是从`root crate`开始的相对路径，这在很多时候是不方便的，我们也可以像下面的例子一样用绝对路径

```
mod front_of_house{
    pub mod hosting{
        fn add_to_waitlist(){}

        fn seat_at_table(){}

        pub fn give_two() -> i32{
            2
        }
    }

    mod serving{
        fn take_over(){}
```

```
        fn serve_order() {}

        fn take_payment() {}
    }
}
#[cfg(test)]
mod tests {
    #[test]
    fn it_works() {
        let result = super::front_of_house::hosting::give_two();
        assert_eq!(result, 2);
    }
}

pub fn hello(){
    let result = front_of_house::hosting::give_two();
}
```

- 在hello()函数中，因为这个函数就处在`root crate`当中，所以它的相对路径就是直接`front_of_house::hosting::give_two()`
- 在`it_work()`中，我们用的是`super::front_of_house::hosting::give_two()`，因为这个函数在tests这个module中，我们需要用super回到上级路径。

如何获取private 的内容呢?

- 我们知道在C++中，之所以使用public和private是想区分读写的权限，private只可读，那么我们如何在rust中如何读取private的内容呢?

```
mod back_of_house(){
    pub struct Breakfast{
        pub toast: String,
        seasonal_fruit: String,
    }

    impl Breakfast {
        impl Breakfast {
            pub fn summer() -> String{
                "peaches".to_string()
            }
        }
    }
}
```

- 在下面的例子中，虽然Breakfast这个struct是public的，但里面的data field是可以不共有的，比如seasonal_fruit就是private的。
- 想要获取它，可以用impl一个获取它的函数。

- 对于Enum而言，它比较特殊的是，enum中的variants是默认public（这样就不用再public中一个一个加public了）

```
mod back_of_house{
    pub struct Breakfast{
        pub toast: String,
        seasonal_fruit: String,
    }
```

```
    impl Breakfast {
        pub fn summer() -> String{
            "peaches".to_string()
        }
    }

    pub enum Appetizer{
        soup,
        Salad,
    }
}
```

Use

- 每次都要写一次path其实非常的冗长，我们可以在scope中引入`use`. 有点类似C++中的using namespace

```
#[cfg(test)]
mod tests {

    use crate::front_of_house::hosting;

    #[test]
    fn it_works() {
        let result = hosting::give_two();
        assert_eq!(result, 2);
    }
}
```

- 我们也可以use 相对路径

```
#[cfg(test)]
mod tests {

    // use crate::front_of_house::hosting;
    use super::front_of_house::hosting;

    #[test]
    fn it_works() {
        let result = hosting::give_two();
        assert_eq!(result, 2);
    }
}
```

As

- 我们是可以用as"重命名"一个path的。比如：

```
use std::io::Result as IoResult;
```

引用外部的package

- 这个在第二章用rand的时候有演示，其实就是在Cargo.toml中加入包的名字和版本号：

```
rand = "0.8.3"
```

- standard_library（std）也是我们package之外的crate，但因为std是shipped with rust的，我们不用在Cargo.toml中将它include进来。比如对于HashMap,我们可以直接：

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

我们use的写法可以更加简单

- 比如：

```
use std::{cmp::Ordering, io};
```

它等效于：

```
use std::cmp::Ordering;
use std::cmp::io;
```

- 比如：

```
use std::io::{self, Write};
```

它等效于：

```
use std::io;
use std::io::Write;
```

- 我们也可以把一个path下的所有public全部bring进来：

```
use std::collections::*;
```

将Modules分散在不同的文件中

- 之前我们所写的modules都是在一个文件里，这样很蠢，因为一个项目肯定不会只有一个文件。我们希望写多个文件，并且能用引用到。
- 首先，在创建/src/front_of_house，把host module放进去：

```
pub mod hosting{
    fn add_to_waitlist(){}

    fn seat_at_table(){}

    pub fn give_two() -> i32{
        2
    }
}
```

- 在lib中：

```
mod front_of_house;

#[cfg(test)]
mod tests {

    pub use crate::front_of_house::hosting;

    #[test]
    fn it_works() {
        let result = hosting::give_two();
        assert_eq!(result, 2);
    }
}
```

  - 这里`mod front_of_house;`是将文件名为front_of_house.rs的文件中的module给load进来（`;`告诉编译器，不是定义一个新的module，而是从这个文件中load）
  - 然后就可以正常用了
- 我们的文件也可以不在同一级目录下，比如，我们创建一个/src/back_of_house/cooking.rs文件，并写上：

```
pub fn give_100() -> i32{
    100
}
```

  - 每个文件/文件夹就相当于一个module了
- 然后创建一个和文件夹同名的/src/back_of_house.rs文件，写上：

```
pub mod cooking;
```

  - 把cooking.rs给load进来
- 然后在/src/lib.rs中：

```rust
mod back_of_house;

#[cfg(test)]
mod tests {
    pub use crate::back_of_house::cooking;

    #[test]
    fn cooking_works(){
        let result = cooking::give_100();
        assert_eq!(result, 100);
    }
}
```