Match Control Flow

- Match是一个control flow structure, 允许用户去将一个变量的值跟一系列的pattern进行比较，然后基于matched的pattern执行对应的代码。我们可以看到下面这个例子：

```
enum Coin{
    Penny,
    Nickel,
    Dime,
    Quarter,
}

fn main() {
    let p = Coin::Penny;
    dbg!(value_in_cents(p));
}
fn value_in_cents(coin: Coin) -> u8{
    match coin{
        Coin::Penny => 1,
        Coin::Nickel => 5,
        Coin::Dime => 10,
        Coin::Quarter => 25,
    }
}
```

  - match的结构是由很多的arms组成的。在每个arms后面都有一个expression，将会返回值。
  - 但要注意的是，这个match中的patterns必须要涵盖所有可能的情况，不然是很不安全的。以下的实例也会编译报错：

```
fn value_in_cents(coin: Coin) -> u8{
    match coin{
        Coin::Penny => 1,
        Coin::Nickel => 5,
        Coin::Dime => 10,
        // Coin::Quarter => 25,
    }
}
```

- 同时，arm后面的expression是可以执行很多东西的，比如：

```
fn value_in_cents(coin: Coin) -> u8{
    match coin{
        Coin::Penny => {
            println!("lucky penney!");
            1
        },
        Coin::Nickel => 5,
        Coin::Dime => 10,
        Coin::Quarter => 25,
    }
}
```

- 如果enum中的variants associate了data呢？match是可以从中将data取出来的：

```
#[derive(Debug)]
enum UsState {
    Alabama,
    Alaska,
    // etc
}

enum Coin{
    Penny,
    Nickel,
    Dime,
    Quarter(UsState),
}

fn main() {
    let p = Coin::Quarter(UsState::Alaska);
    dbg!(value_in_cents(p));
}

fn value_in_cents(coin: Coin) -> u8{
    match coin{
        Coin::Penny => {
            println!("lucky penney!");
            1
        },
        Coin::Nickel => 5,
        Coin::Dime => 10,
        Coin::Quarter(state) => {
            println!("State quarter from {:#?}!", state);
            25
        },
    }
}
```

  - 这样，我们就可以将data从enums中提出出来（我们可以用这个提取出Option中的T）

Matching with Option T

- 我们可以用match，将Option里的data取出来，进行运算，并处理None的情况：

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

fn main() {

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

- Match是Exhaustive的，即我们上面所说的，在match的patterns中需要包含enums的所有情况。比如以下也会编译错误：

```
fn plus_one(x: Option<i32>) -> Option<i32>{
    match x{
        Some(i) => Some(i+1),
    }
}
```

  - 这强迫程序员必须考虑None时候的情况，从而确保程序的安全！
- 当然，我们也可以用`other`来达到match的exhaustive要求，比如下面的match并不会报错：

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

  - 这样的话，如果pattern match了的话，other其实是会被bind上value的，compiler会提醒unused variable. 为了避免这个warning，我们可以选择用`_`代替`other`, 这样值就不会被绑定上去。

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

- 并且，arm后面的expression也可以是空的，代表并不进行任何操作：

```
fn main() {
    let dice_roll = 9;
    match dice_roll{
        3 => add_fancy_hat(),
        7 => remove_fancy_hat(),
        _ => ()
    }
}

fn add_fancy_hat(){}
fn remove_fancy_hat(){}
```