Defining Enum

- Enum是一种新的数据结构，它和struct还是有一些不同点的，接下来将通过举例来说明Enum这个数据类型的特点
- 我们把Enum下的变量称为variant，它是可以不和任何数据类型associated的：

```
enum IpAddrKind{
    V4,
    V6,
}

fn main() {
    let four = IpAddrKind::V4;
    let six = IpAddrKind::V6;
}
```

- 可以看到，每个IpAddrKind的实例只能是其中的一个variant，这和struct是不同的
- 当然，也可以和某些数据类型associated, 比如，下面例子里variants规定是String类型的：

```
enum IpAddrKind{
    V4(String),
    V6(String),
}

fn main() {
    let four = IpAddrKind::V4(String::from("127.0.0.1"));
    let six = IpAddrKind::V6(String::from("::1"));
}
```

  - 这样，我们就可以给enum的每个variant附上数据。而这个例子也相当于是自动得到了一个constructor function
- 另外，和struct不同的是，Enum下的variant可以拥有不同的types和amounts of associated data:
- 并且！variant中可以是任意的数据类型，比如strings, structs, 或者另一个enum，可以看下面这个例子：

```
enum Message{
    Quit,
    Move {x: i32, y: i32},
    Write(String),
    ChangeColor(i32, i32, i32),
}

fn main() {

    let msg1 = Message::Quit;

    let msg2 = Message::Move{
        x: 1,
        y: -1,
    };
```

```
    let msg3 = Message::Write(String::from("localhost"));

    let msg4 = Message::ChangeColor(100, 101, 99);
}
```

- 可以是像Quit这样，with no data associated with it at all
- Move：有个像struct一样的named field
- Write：包含了一个string
- ChangeColoar: 包含了三个i32
- 我们可以看到，如果单纯用`struct`来表示Message这个类型的话，就需要分别写四个独立的struct。可见enum类型还是非常强大的。

- 和struct一样的是，我们也可以为enum实现方法：

```
#[derive(Debug)]
enum Message{
    Quit,
    Move {x: i32, y: i32},
    Write(String),
    ChangeColor(i32, i32, i32),
}

impl Message{
    fn call(&self){
        println!("The message reads {:#?}", self);
    }
}

fn main() {

    let msg1 = Message::Quit;

    let msg2 = Message::Move{
        x: 1,
        y: -1,
    };

    let msg3 = Message::Write(String::from("localhost"));

    let msg4 = Message::ChangeColor(100, 101, 99);

    msg1.call();
    msg2.call();
    msg3.call();
    msg4.call();
}
```

输出：

```
The message reads Quit
The message reads Move {
    x: 1,
    y: -1,
}
The message reads Write(
    "localhost",
)
The message reads ChangeColor(
    100,
    101,
    99,
)
```

Option

- 在C++等很多其他语言中，都有Null这个概念，但当我们对Null空值使用not-null值的操作时，就会发生error。但Null可以表示一个值是absent的，这个功能是十分useful的。而Rust作为一个安全的语言，是不允许Null出现的。其取代Null出现的更安全的功能，就是Option

```
// Option在standard lib中的实现
enum Option<T>{
    None,
    Some(T),
}
```

- 因为Option太常用了，它被包含在了preclude中，这说明我们使用的时候可以省去`Option<T>::`这个前缀。但在None的情况下是不可以省去的：

```
let some_number = Some(5);
let some_string = Some("a string");
let absent_var: Option<i32> = None;
```

  - 这里，定义一个变量是None的时候，我们是不能省去前缀的。如果改成`let absent_var = None;`，它会报错：

```
let absent_var = None;
   |         ----------   ^^^^ cannot infer type for type parameter `T`
declared on the enum `Option
```

  - 那是因为rust的compiler是只根据一个None就推断出其Option的Some中包含的数据类型。但像`let some_number = Some(5);`这种它就可以推断出是i32类型。

- 那么为什么这种设计会比直接用一个Null更加安全呢？这是因为`Option<T>`和`T`不是一种数据类型，`T`是被Option包裹起来的。看下面的例子：

```
let x: i8 = 5;
let y: Option<i8> = Some(5);

let sum = x + y;
```

  - 会编译报错，因为不是一个数据类型

    - 这意味着，当程序员想说明一个变量可能为空的时候，其必须将值包裹在Option下。而Option又不能和其他的值直接操作。这就需要一个从Option中取出值得进行额外操作。这步操作就需要对值是否为None进行处理 ----> In order to have a value that can be possibly null, you must explicitly opt in by making the type of that value Option . Then, when you use that value, you are required to explicitly handle the case when the calue is null.
- 那么，如何取出Some中的值呢？这需要用到match，将在之后的章节讲到。