Generics_Triats_Lifetime

- 在前面我们遇到的Result<T, E>, Option, 和Hash<K, V>都是用到了Generics，它允许我们输入泛化的数据类型而不必输入准确的数据类型到一个函数中，这大大减少了code duplication.
- 我们先讲讲一般情况下我们是如何减少code的duplication的。我们一般是利用函数，比如下面的找出一个vec中最大值的方法：

```
fn main() {
    let number_list = vec![34, 50, 25, 100, 65];

    let mut largest = number_list[0];

    for number in number_list {
        if number > largest {
            largest = number;
        }
    }

    println!("The largest number is {}", largest);
}
```

- 我们可以把它精简为：

```
fn main() {
    let number_list = vec![34, 50, 25, 100, 65];

    let result = largest(&number_list);

    println!("The largest number is {}", result);
}

fn largest(list: &[i32])-> i32 {
    let mut largest = list[0];

    for &item in list {
        if item > largest{
            largest = item;
        }
    }
    largest
}
```

    - 这样我们关于求一个Vec中最大值的函数就可以重复使用。
    - 这里的`&item`并不是在reference一个i32, 而是将for返回的&i32类型给destructure为i32. 后面第18章会讲到。
- 但是现在函数中能输入的还是i32， 如何让它handle更多的数据类型呢?

Generics type

- 我们可以用以下的形式来实现一个函数handle不同的数据类型：

```
fn largest<T>(list: &[T])-> T
```

- 于是上面的函数可以改成：

```
fn largest<T>(list: &[T])-> T {
    let mut largest = list[0];

    for &item in list {
        if item > largest{
            largest = item;
        }
    }
    largest
}
```

  - 但实际上，这段代码会报编译错误：

```
error[E0369]: binary operation `>` cannot be applied to type `T`
  --> src/main.rs:13:17
   |
13 |         if item > largest{
   |            ---- ^ ------- T
   |            |
   |            T
   |
help: consider restricting type parameter `T`
   |
9  | fn largest<T: std::cmp::PartialOrd>(list: &[T])-> T {
   |             ++++++++++++++++++++++
```

  - 这个报错跟trait有关，将会在下一小节讲到。

在Struct中用上generics

- 直接看下面的例子：

```
struct Point<T>{
    x: T,
    y: T,
}

fn main() {
    let integer = Point{x: 5, y: 10};
    let charac = Point{x: 'a', y: 'b'};
}
```

  - 但在这个例子下，x, y必须是同一数据类型，如果想让x, y是不同的数据类型，我们可以依赖于：

```
struct weird_point<T, U>{
    x: T,
    y: U,
}

fn main() {
    let strange = weird_point{x: 5, y: 'b'};
}
```

在Enums中用上Generics

- 看完上面的例子后，之前遇到的Option，Result<T, E>就很符合直觉了。

```
enum Option<T>{
    Some(T),
    None,
}
enum Result<T, E>{
    Some(T),
    Err(E),
}
```

在impl中用上Generics

- 直接看下面的例子：

```
#[derive(Debug)]
struct weird_point<X1, Y1>{
    x: X1,
    y: Y1,
}

impl<X1, Y1> weird_point<X1, Y1>{
    fn mixup<X2, Y2>(self, other: weird_point<X2, Y2>) -> weird_point<X1,
Y2>{
        weird_point {
            x: self.x,
            y: other.y,
        }
    }
}

fn main() {
    let int_p = weird_point{x: 5, y: 1};
    let char_p = weird_point{x: 'a', y: 'b'};

    let stange = int_p.mixup(char_p);

    println!("the mix point is {:?}", stange);
}
```

Compiler是如何处理Generics的

- 在实际中，用gerenics并不会造成runtime时候有更多的开销，这和Rust的compiler实现generics的方法有关。Rust用到了monomorphization来减小generics的开销：
- 比如Option在编译的时候会根据输入的数据类型变成：

```
enum Option_i32{
    Some(i32),
    None,
}
enum Option_f64{
    Some(f64),
    None,
}

fn main(){
    let integer = Option_i32::Some(5);
    let float = OPtion_f64::Some(5.0);
}
```