Traits

- Trait是定义了很多数据结构通用的behaviors， 比如有很多不同的struct，但他们都想实现某一个相同的功能。我们可以通过先定义这个功能，然后让这些struct都去impl这个功能。这样讲有点抽象，，先看个例子：

```
pub trait Summary {
    fn summarize(&self) -> String;
}

pub struct NewsArticle{
    pub headline: String,
    pub location: String,
    pub author: String,
    pub content: String,
}

impl Summary for NewsArticle{
    fn summarize(&self) -> String{
        format!("{}, by {} ({})", self.headline, self.author, self.location)
    }
}

pub struct Tweet{
    pub username: String,
    pub content: String,
    pub reply: bool,
    pub retweet: bool,
}

impl Summary for Tweet {
    fn summarize(&self) -> String {
        format!("{}, {}", self.username, self.content)
    }
}
```

  - 在一开始我们定义了这个trait：

```
pub trait Summary {
    fn summarize(&self) -> String;
}
```

  - 它只包含trait名，函数名，输入是什么，以及输出的数据类型。这为下面的impl提供了一定的标准。
  - 接着下面的impl就是根据这个标准来实现各自的功能

Default implementation

- 我们可以在定义trait的时候就实现一些默认的功能：

```
impl Summary for NewsArticle{
    fn summarize(&self) -> String{
        format!("{}, by {} ({})", self.headline, self.author, self.location)
    }
}
```

- 这样，就算我们不为一个stuct去impl这个strait，我们也可以使用这个默认的功能：

```
impl Summary for NewsArticle{
    // 这里面是空的
}
fn main() {
    let article = NewsArticle{
        headline: String::from("This is a NEWS"),
        location: String::from("shenzhen"),
        author: String::from("qpr"),
        content: String::from("nothing"),
    };

    println!("new article available! {}", article.summarize());
}
```

输出：

```
new article available! (Read more...)
```

- 我们在impl的时候可以overwrite这个default的功能：

```
impl Summary for NewsArticle{
    fn summarize(&self) -> String{
        format!("{}, by {} ({})", self.headline, self.author, self.location)
    }
}
fn main() {
    let article = NewsArticle{
        headline: String::from("This is a NEWS"),
        location: String::from("shenzhen"),
        author: String::from("qpr"),
        content: String::from("nothing"),
    };

    println!("new article available! {}", article.summarize());
}
```

输出：

```
new article available! This is a NEWS, by qpr (shenzhen)
```

用trait处理不同数据类型的输入

- 直接看下面的例子：

```
pub fn notify(item: &impl Summary){
    println!("Breaking news! {}", item.summarize());
}

fn main() {
    let article = NewsArticle{
        headline: String::from("This is a NEWS"),
        location: String::from("shenzhen"),
        author: String::from("qpr"),
        content: String::from("nothing"),
    };

    println!("new article available! {}", article.summarize());

    notify(&article);
}
```

  - `pub fn notify(item: &impl Summary)`：指明了item的数据类型，可以输入任何impl了Summary这个trait的输入！

- 上面的写法还可以等价为：

```
pub fn notify<T: Summary>(item: &T){
    println!("Breaking news! {}", item.summarize());
}
```

  - 这样的写法可以让在参数多的时候，简化一些代码：

```
pub fn notify(item1: &impl Summary, item2: &impl Summary) {}

pub fn notify<T: Summary>(item1: &T, item2: &T) {}
```

- 我们甚至可以允许一个函数接受实现了更多trait的数据类型(通过`+`)：

```
pub fn notify(item: &(impl Summary + Display)) {}

pub fn notify<T: Summary + Display>(item: &T) {}
```

- 但是当trait数量和参数的数量比较多的时候，还是比较冗长，我们可以用where来做进一步的简化：

```
fn some_function<T: Display + Clone, U: Clone + Debug>(t: &T, u: &U) -> i32
{}

fn some_function<T, U>(t: &T, u: &U) -> i32 {
    where T: Display + Clone,
          U: Clone + Debug,
}
```

- 甚至可以讲`impl Summary`放在return type的位置上，用来说明这个函数要返回的是一个实现过这个trait的数据类型：

```
fn return_summarizable() -> impl Summary{
    Tweet {
        username: String::from("horse_ebooks"),
        content: String::from("nothing at all"),
        reply: false,
        retweet: false,
    }
}
```

  - 这样，我们的return type也可以handle不同的数据类型了！！

Fixing the largest Function with Trait Bounds

- 我们回到上一小小节的例子，当我们想要用`>`比较两个generics types的时候，它会报错：

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

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

报错：

```
error[E0369]: binary operation `>` cannot be applied to type `T`
  --> src/main.rs:63:17
   |
63 |         if item > largest{
   |            ---- ^ ------- T
   |            |
   |            T
   |
help: consider restricting type parameter `T`
   |
59 | fn largest<T: std::cmp::PartialOrd>(list: &[T])-> T {
```

  - 这是因为`>`是一个method on the standard libary trait `std::cmp::PartialOrd`. 如果不把这个trait bound加上的话，T是不能用这个method的。所以：

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

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

- 可这次报错变成了：

```
error[E0508]: cannot move out of type `[T]`, a non-copy slice
  --> src/main.rs:60:23
   |
60 |     let mut largest = list[0];
   |                       ^^^^^^^
   |                       |
   |                       cannot move out of here
   |                       move occurs because `list[_]` has type `T`, which
does not implement the `Copy` trait
   |                       help: consider borrowing here: `&list[0]`

error[E0507]: cannot move out of a shared reference
```

- 这是因为我们的这句`let mut largest = list[0];`默认了list里面的值是实现了Copy trait的。但实际上我们用了泛化的T后，T就有可能是那些没有实现copy trait的数据类型，所以我们需要再加上Copy这个bound, 来限制输入的T实现了Copy trait（比如我们希望的char和i32）

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

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