Lifetimes

- Lifetimes是指如上图中reference是valid的时间。一般来说，Lifetime是implicit and inferred。但这样并不能让人安心。所以rust也提供了一套机制来让人确保一个reference在一定时间内一定是valid的。
- Lifetimes存在的目的很大一部分是为了防止悬空指针，比如：

```
fn main() {
    let r;
    {
        let x = 5;
        r = &x;
    }
    println!("r: {}", r);
}
```

  - 上面程序会报错的原因是r lives longer than x. 这就导致了悬空指针的情况。

Generic Lifetiems in Functions

- 我们考虑下面的一段程序：

```
fn main() {
    let string1 = String::from("abcd");
    let string2 = "xyz";

    let result = longest(&string1.as_str(), &string2);
    println!("the longest string is {}", result);
}

fn longest(x: &str, y: &str) -> &str {
    if x.len() > y.len() {
        x
    } else{
        y
    }
}
```

  报错信息：

```
error[E0106]: missing lifetime specifier
 --> src/main.rs:9:33
  |
9 | fn longest(x: &str, y: &str) -> &str {
  |               ----     ----     ^ expected named lifetime parameter
  |
  = help: this function's return type contains a borrowed value, but the
signature does not say whether it is borrowed from `x` or `y`
help: consider introducing a named lifetime parameter
  |
9 | fn longest<'a>(x: &'a str, y: &'a str) -> &'a str {
  |           ++++     ++          ++          ++

For more information about this error, try `rustc --explain E0106`.
```

- 产生上面错误的原因是：rust并不知道这个程序返回的是x还是y，也就是说rust compiler不能确定result的lifetime。万一返回的result的lifetime很长，那就有可能造成悬空指针的问题。所以rust需generic lifetime parameters去确定返回的值lifetime有多长。

Lifetime Annotation syntax

- Lifetime annotations并不改变一个reference的lifetime，而只是显式地描述多个references的lifetimes之间的关系。

```
&i32        // a reference
&'a i32     // a reference with an explicit lifetime
&'a mut i32  // a mutable reference with an explicit lifetime
```

- 我们对之前的报错进行修改：

```
fn longest<'a>(x: &'a str, y: &'a str) -> &'a str {
    if x.len() > y.len() {
        x
    } else{
        y
    }
}
```

  - 这段程序用`'a`描述了x, y, 和返回值之间lifetime的关系，用来表示这三个值需要有一样长的lifetime
  - 加上这个后，就可以正常编译了。
  - **注意，加上`'a`并不是改变了lifetime的长度，而是对这个函数做出了限制：限制输入的参数和输出的结果需要有这样的lifetimes的关系。**
- 如果我们输入的参数和输出的结果不满足这样的lifetime的关系呢?

```
fn main() {
    let string1 = String::from("abcd");

    let result;
    {
        let string2 = String::from("xyz");
        result = longest(string1.as_str(), string2.as_str());
    }
    println!("the longest string is {}", result);
}
```

报错为：

```
error[E0597]: `string2` does not live long enough
 --> src/main.rs:7:44
  |
7 |         result = longest(string1.as_str(), string2.as_str());
  |                                            ^^^^^^^^^^^^^^^^ borrowed
value does not live long enough
8 |     }
  |     - `string2` dropped here while still borrowed
9 |     println!("the longest string is {}", result);
  |                                          ------ borrow later used here
```

- 很容易看出来，上述的程序中，string2的reference在中间就invalid了，它并不满足函数要求的那样和string1的reference以及result有同样的lifetime，所以就会报错。

Lifetime annotation in Stuct

- 下面的例子：

```
struct ImportantExcept<'a> {
    part: &'a str,
}
```

- 这个要求了part这个reference的lifetime不能比instance还要长。

Lifetime annotation in Method Definition

- 比如下面的程序会报错：

```
struct ImportantExcept<'a> {
    part: &'a str,
}

impl ImportantExcept{

}
```

- 这是因为，有可能impl的method会输出跟part有关的reference。万一这个reference比ImportantExcept的instance的lifetime还要长，就会造成悬空指针的情况。所以我们需要改成：

```
impl<'a> ImportantExcept<'a>{

}
```

---

static lifetime

- static lifetime `'static`是要求reference必须有跟整个程序一样长的lifetime，比如下面的例子就一定有：

```
let s: &'static str = "I have a static lifetime";
```

- 但对于并没有整个程序那么长的lifetime的reference用`'static`的话，就会报错：

```
fn main() {
    {
        let x = String::from("I will die soon");

        let d: &'static str = x.as_str();
    }
    println!("hello world!");
}
```

Together with trait bound

- Lifetime实际上也算是一种generics，当又有strait的时候，我们可以写成：

```
fn longest_with_an_announcement<'a, T>(x: &'a str, y: &'a str, ann: T) ->
&'a str
    where T: Display
{
    println!("Announcement! {}", ann);
    if x.len() > y.len() {
        x
    } else{
        y
    }
}
```

只能说Rust的这个Generics的功能非常的强大！可以说是这个语言很大的魅力所在了！