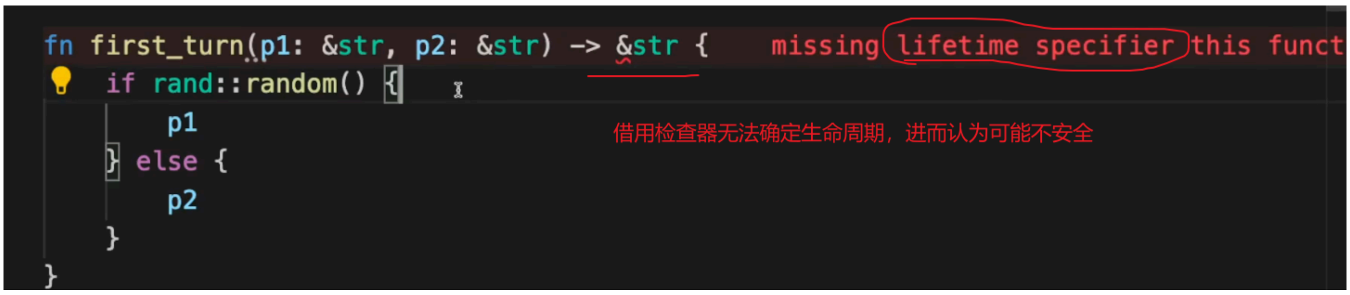
生命周期

- Concrete lifetimes

  > 一个值在特定内存中生存期，他从创建时开始或者移动到一个特定具体的memory location，并在该值被删除或移出特定内存位置时结束
  >
  > - 值本身的生命周期：一般是作用域，如`{}`
  > - 引用的生命周期：要求被引用的值的生命周期不能小于引用的生命周期，否则发生悬空引用

- generic lifetimes

  > 生命周期说明符（lifetime specifier），也称为泛型生命周期注释(generic lifetime annotations): 用于描述生民周期之间关系的一种方式，标注帮助借用检查器在后续代码中检查问题，其实还用什么样的生命周期跟实际的目的有关。
  >
  > - `'a`：定义了一个名叫 a 生命周期，名称可以任意，但公约一般是 tick符号`'` + 单个小写字母

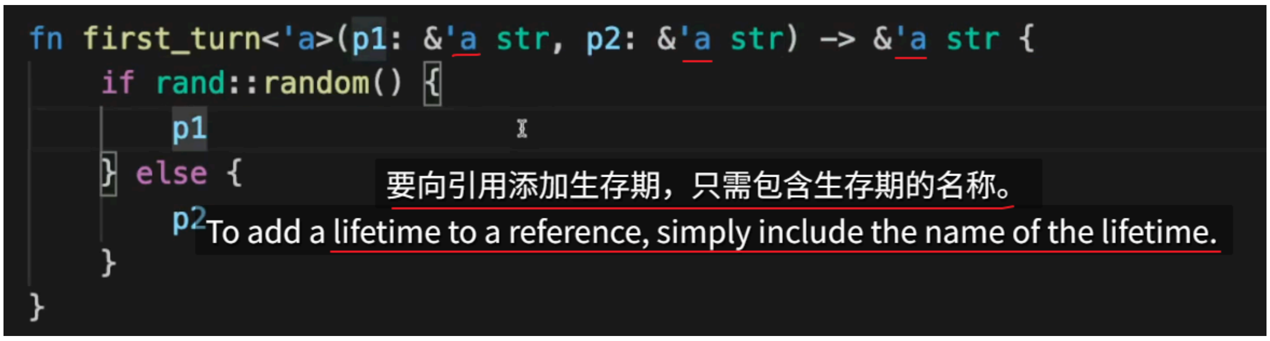


```
fn first_turn(p1: &str, p2: &str) -> &str {    missing lifetime specifier this funct
    if rand::random() {
        p1
    } else {
        p2
    }
}
```

借用检查器无法确定生命周期，进而认为可能不安全

- 向引用添加生命周期：只需加上生命周期名称即可

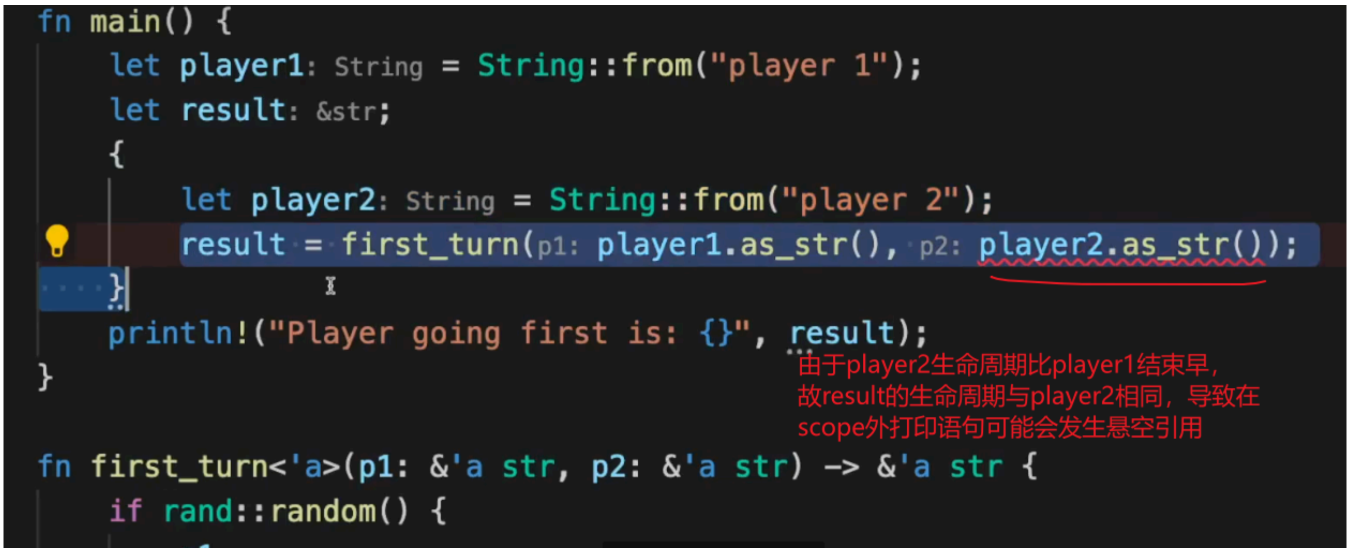
  > 注意：`'a'` 并不是concrete lifetime, 他仅仅描述了生命周期之间的关系：
  >
  > p1 , p2 及 返回值具有相同生命周期，即返回值的生命周期与p1, p2中最短的那个相同



```
fn first_turn<'a>(p1: &'a str, p2: &'a str) -> &'a str {
    if rand::random() {
        p1
    } else {
        p2
    }
}
```

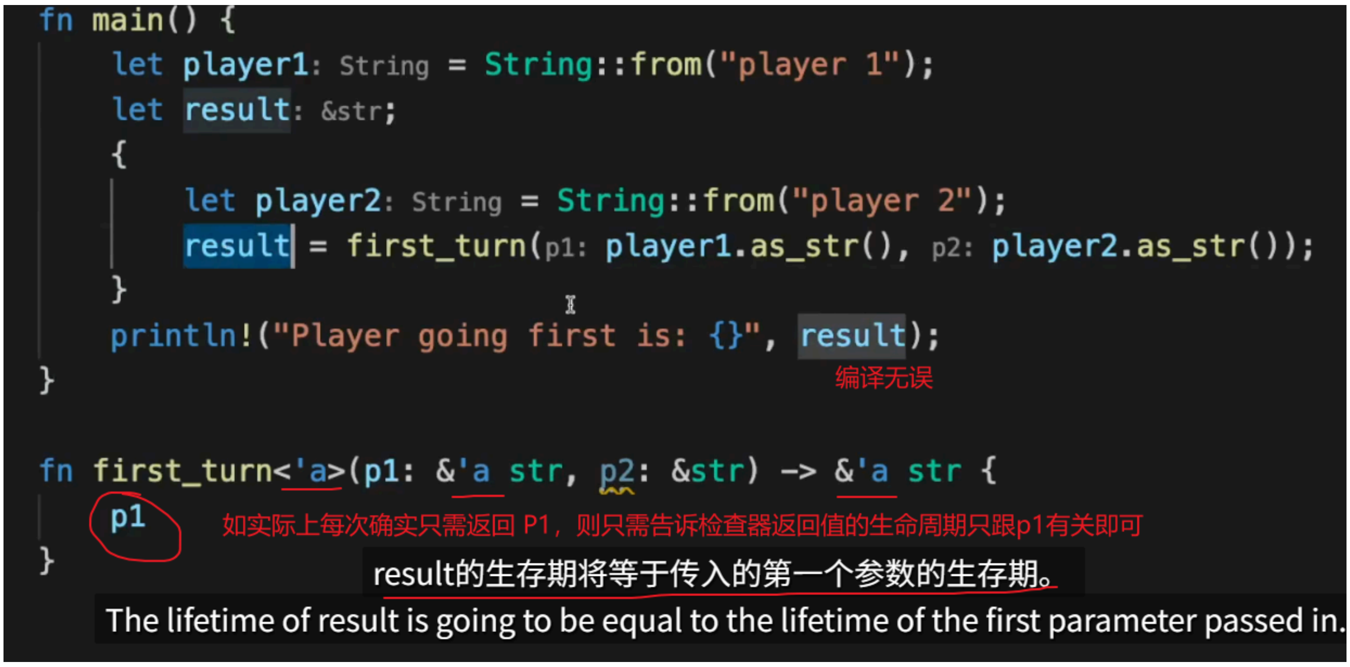
要向引用添加生存期，只需包含生存期的名称。
To add a lifetime to a reference, simply include the name of the lifetime.

- 生命周期不同情况



```
fn main() {
    let player1: String = String::from("player 1");
    let result: &str;
    {
        let player2: String = String::from("player 2");
        result = first_turn(p1: player1.as_str(), p2: player2.as_str());
    }
    println!("Player going first is: {}", result);
}

fn first_turn<'a>(p1: &'a str, p2: &'a str) -> &'a str {
    if rand::random() {
```

由于player2生命周期比player1结束早，故result的生命周期与player2相同，导致在scope外打印语句可能会发生悬空引用



```
fn main() {
    let player1: String = String::from("player 1");
    let result: &str;
    {
        let player2: String = String::from("player 2");
        result = first_turn(p1: player1.as_str(), p2: player2.as_str());
    }
    println!("Player going first is: {}", result);
}

fn first_turn<'a>(p1: &'a str, p2: &str) -> &'a str {
    p1
}
```

编译无误

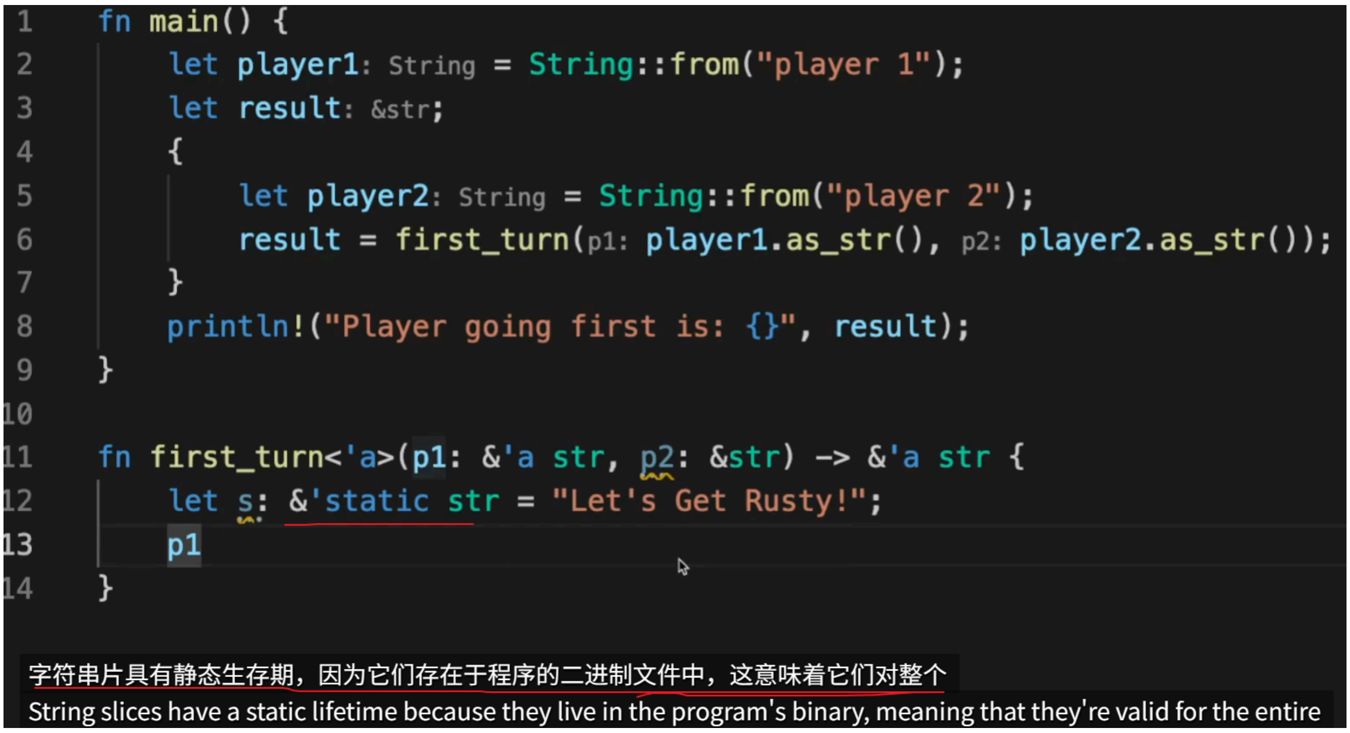
如实际上每次确实只需返回 P1，则只需告诉检查器返回值的生命周期只跟p1有关即可

result的生存期将等于传入的第一个参数的生存期。

The lifetime of result is going to be equal to the lifetime of the first parameter passed in.

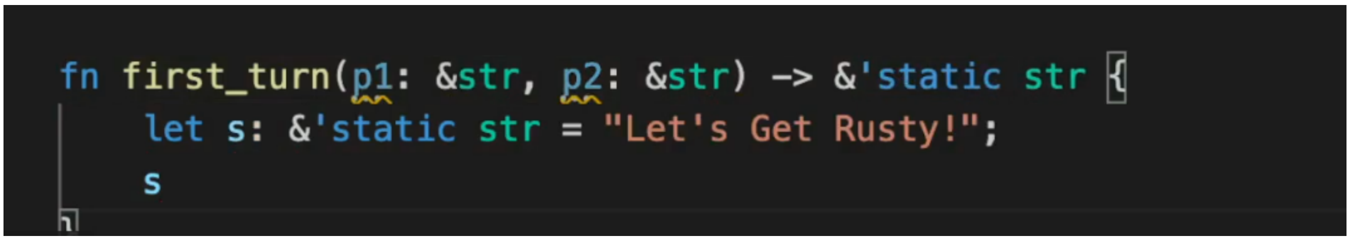
- 静态生命周期

> 如字符串切片



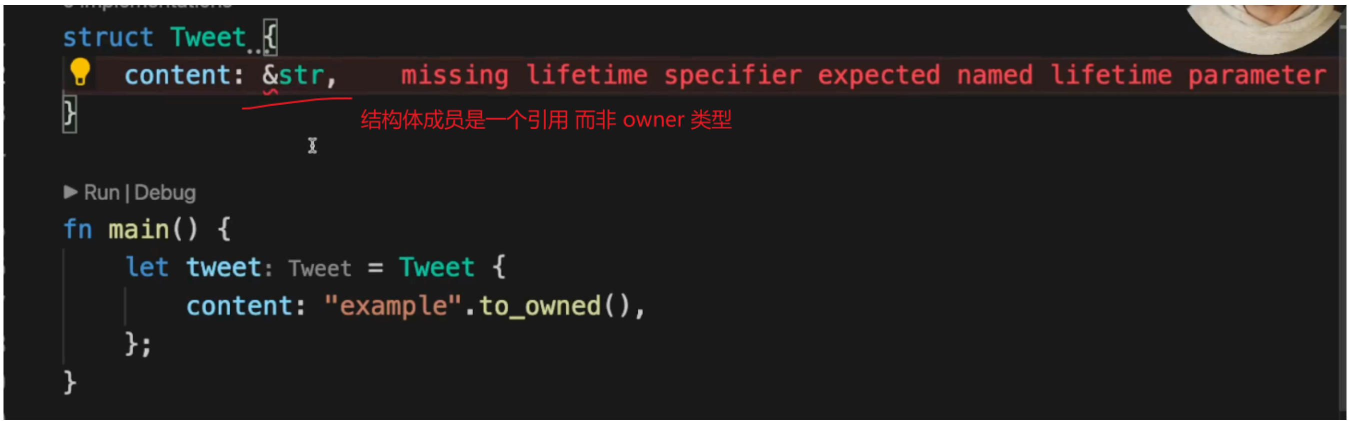
```rust
fn main() {
    let player1: String = String::from("player 1");
    let result: &str;
    {
        let player2: String = String::from("player 2");
        result = first_turn(p1: player1.as_str(), p2: player2.as_str());
    }
    println!("Player going first is: {}", result);
}

fn first_turn<'a>(p1: &'a str, p2: &str) -> &'a str {
    let s: &'static str = "Let's Get Rusty!";
    p1
}
```

字符串片具有静态生存期，因为它们存在于程序的二进制文件中，这意味着它们对整个
String slices have a static lifetime because they live in the program's binary, meaning that they're valid for the entire



```rust
fn first_turn(p1: &str, p2: &str) -> &'static str {
    let s: &'static str = "Let's Get Rusty!";
    s
}
```

- Struct 与生命周期省略



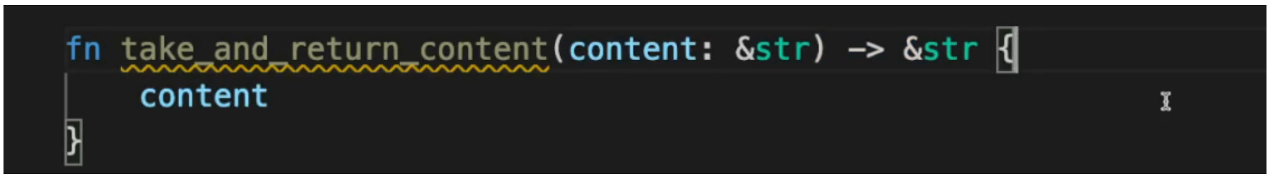
```rust
struct Tweet {
    content: &str,    missing lifetime specifier expected named lifetime parameter
}
```

结构体成员是一个引用 而非 owner 类型

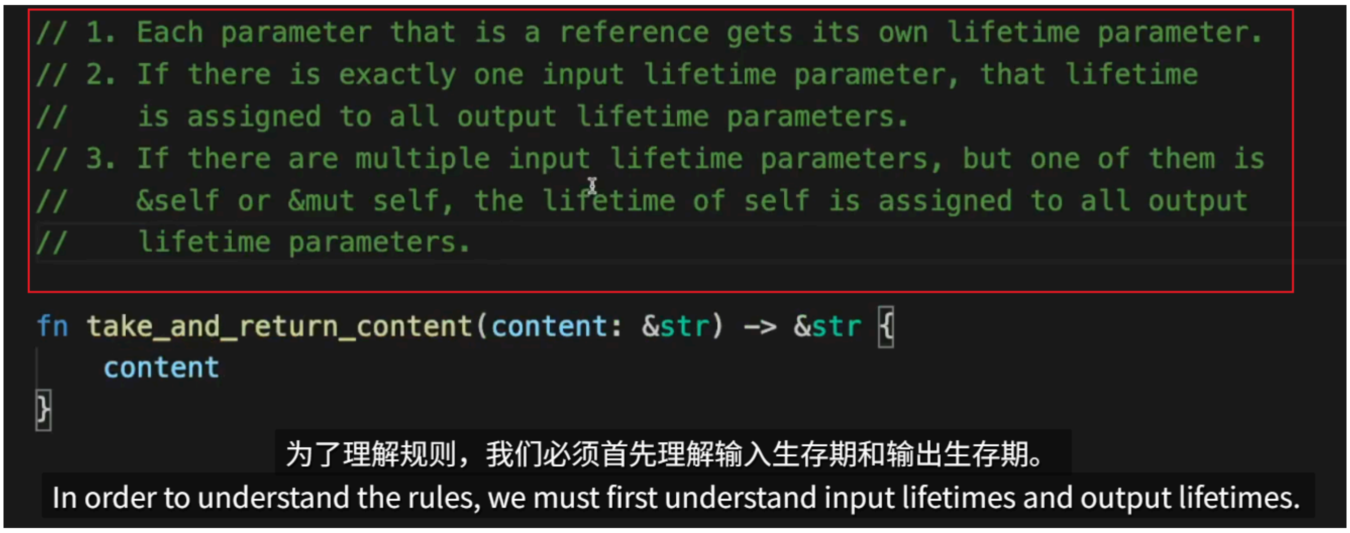
```rust
▶ Run | Debug
fn main() {
    let tweet: Tweet = Tweet {
        content: "example".to_owned(),
    };
}
```

- 省略生命周期

> 函数的入参直接被返回情况



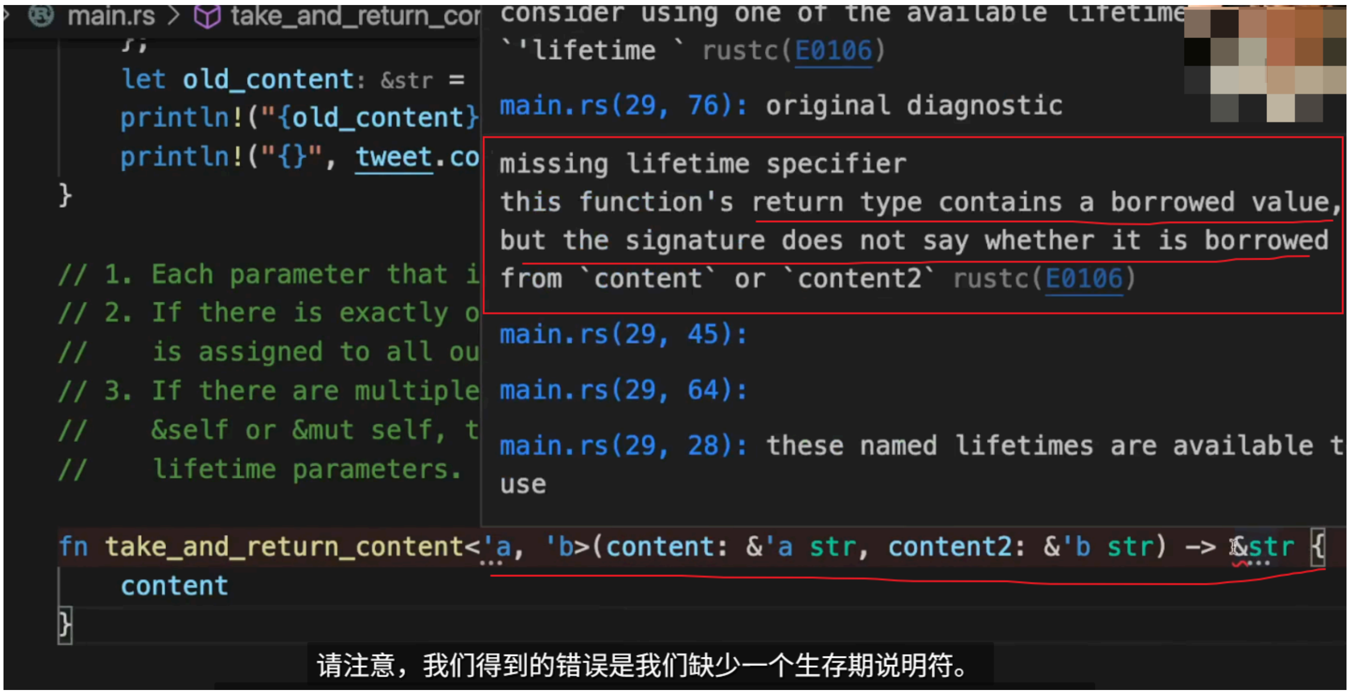
```rust
fn take_and_return_content(content: &str) -> &str {
    content
}
```



```rust
// 1. Each parameter that is a reference gets its own lifetime parameter.
// 2. If there is exactly one input lifetime parameter, that lifetime
//    is assigned to all output lifetime parameters.
// 3. If there are multiple input lifetime parameters, but one of them is
//    &self or &mut self, the lifetime of self is assigned to all output
//    lifetime parameters.

fn take_and_return_content(content: &str) -> &str {
    content
}
```

为了理解规则，我们必须首先理解输入生存期和输出生存期。
In order to understand the rules, we must first understand input lifetimes and output lifetimes.

是为什么当我们只有一个引用作为输入参数时，我们不需要显式地注释生存期。

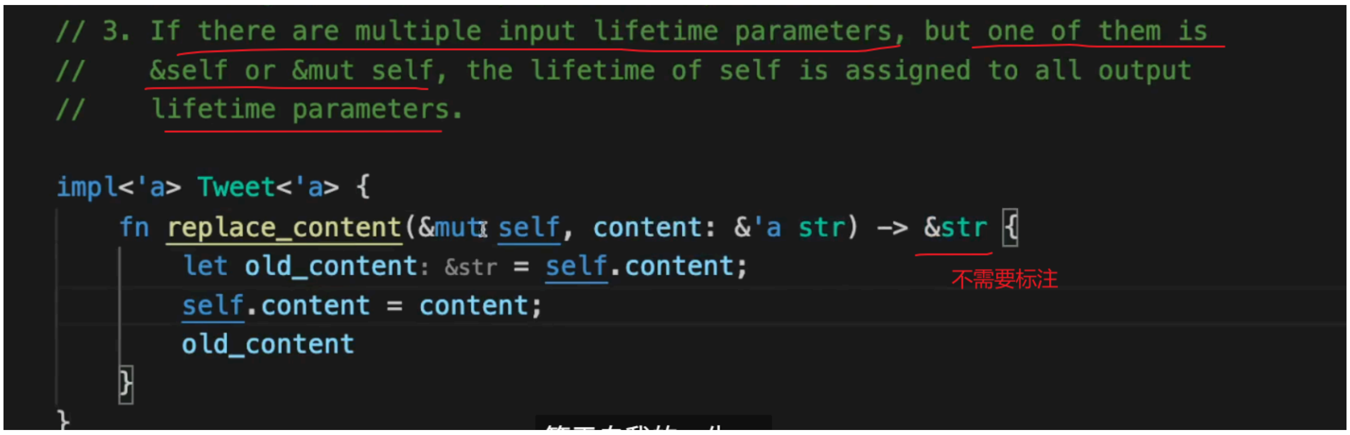


main.rs > take_and_return_con

```rust
    };
    let old_content: &str =
    println!("{old_content}
    println!("{}", tweet.co
}

// 1. Each parameter that i
// 2. If there is exactly o
//    is assigned to all ou
// 3. If there are multiple
//    &self or &mut self, t
//    lifetime parameters.

fn take_and_return_content<'a, 'b>(content: &'a str, content2: &'b str) -> &str {
    content
}
```

```
consider using one of the available lifetime
`'lifetime ` rustc(E0106)
main.rs(29, 76): original diagnostic
missing lifetime specifier
this function's return type contains a borrowed value,
but the signature does not say whether it is borrowed
from `content` or `content2` rustc(E0106)
main.rs(29, 45):
main.rs(29, 64):
main.rs(29, 28): these named lifetimes are available t
use
```

请注意，我们得到的错误是我们缺少一个生存期说明符。



```rust
// 3. If there are multiple input lifetime parameters, but one of them is
//    &self or &mut self, the lifetime of self is assigned to all output
//    lifetime parameters.

impl<'a> Tweet<'a> {
    fn replace_content(&mut self, content: &'a str) -> &str {
        let old_content: &str = self.content;
        self.content = content;
        old_content
    }
}
```

不需要标注