移给函数，函数的参数s只是text的一个不可变引用。

```
fn print_text(s: &str) {
    // s是不可变引用，不能被修改
    println!("text is: {}", s);
}

fn main() {
    let text: String = String::from("Hello CSDN");
    // 传递不可变引用给函数
    print_text(&text);
    // text仍然有效
    println!("{}", text);
}
```

注意：给一个变量指定不可变引用后，不能再转移变量的所有权。

```
fn main() {
    let str1 = String::from("CSDN");
    let str2: &String = &str1;
    // 编译错误：move out of `str1` occurs here
    let str3 = str1;
    println!("{}", str2);
}
```

在上面的示例代码中，str2是str1的不可变引用。但接下来，又将str1赋值给了str3，这就导致str1的所有权转移给了str3。Rust能检测到这种错误的情况，从而导致编译通不过。正确的代码应当是重新给str2指定不可变引用的对象，因为str1已经失效了，具体可参见下面的示例代码。

```
fn main() {
    let str1 = String::from 发生。当一个值的所有权离开作用域时，Rust会自
```

动清理该值所占用的内存空间。如果存在对该值的引用，由于Rust的借用规则，这些引用在所有者被销毁前不能存在，从而避免了悬垂引用的情况。

在下面的示例代码中，我们试图从函数内部返回一个局部变量的引用。这会导致编译错误，因为在函数执行完毕后，text这个局部变量会被销毁，返回的引用将是无效的（即：悬垂引用）。Rust的编译器会检查代码，以确保不存在悬垂引用。如果你尝试编写可能导致悬垂引用的代码，编译器会报错。这是Rust语言的一个重要特性，它允许程序员在编译时捕获这类错误，而不是等到运行时才出现错误。

```
fn test() -> &String {
    let text = String::from("Hello, CSDN");

    // 返回对局部变量的引用，该局部变量会在函数结束时被释放，故会编译报错
    return &text;
}

fn main() {
    let result = test();
    println!("{}", result);
}
```

总结

引用在Rust中非常重要，因为它是实现Rust所有权系统和内存安全性的关键部分。通过引入不可变引用和可变引用，Rust允许程序以更安全的方式操作数据，同时避免了多线程环境下的数据竞争问题。此外，Rust的引用还具有生命周期的概念，这确保了引用的有效性，防止了悬垂引用等问题的发生。

原文链接: https://juejin.cn/post/7364004802856599602