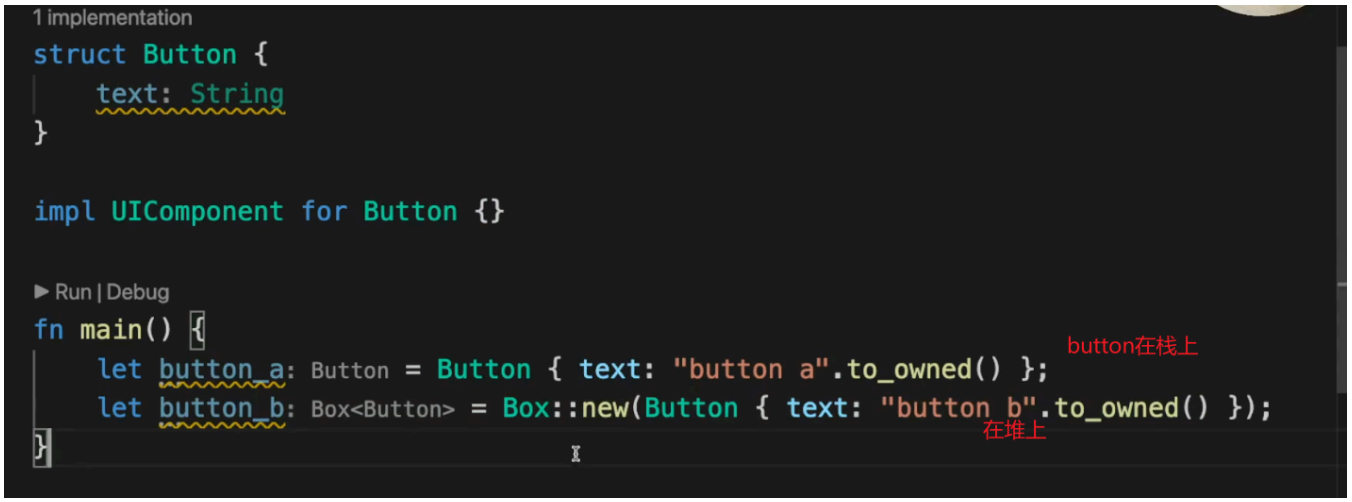
智能指针

Box

> Rust中struct与java中对象不同，rust中结构体（struct）默认是值类型，分配在栈上
>
> **Rust 不自动把对象放到堆上。**
>
> **是否在堆上，由类型（比如 Box、Rc、Arc）或容器（Vec、String）来决定。**
>
> Rust 的哲学是“显式控制内存布局”，以避免 GC 带来的性能不可预测性
>
> Box 类似c++中的唯一智能指针



> 转移所有权时候，栈上的会被复制一份，而堆上的只复制Box智能指针
>
> - 如果Button中有很多fields , 复制会产生很多开销，而Box可以避免复制大量信息
> - 另一个用途是Box与Trait结合

- 与Trait结合使用

```rust
let components: Vec<Box<dyn UIComponent>> = vec![
    Box::new(button_c),
    button_d
];
```

- 当某个类型的大小未知时，但该类型在使用的上下文中要求大小必须确定：常见的是递归类结构，如树结构
  （**因递归类型被认为是无限大小，而rust需要在编译时知道类型大小，主要是因为在栈上分配必须确定大小**）

```rust
1 implementation
struct Container {    recursive type `Container` has infinite size recursive type ha
    name: String,
    child: Container
}                                   递归树形容器结构

impl UIComponent for Container {}
```

```rust
struct Container {
    name: String,
    child: Box<Container>
}

impl UIComponent for Container {}
```

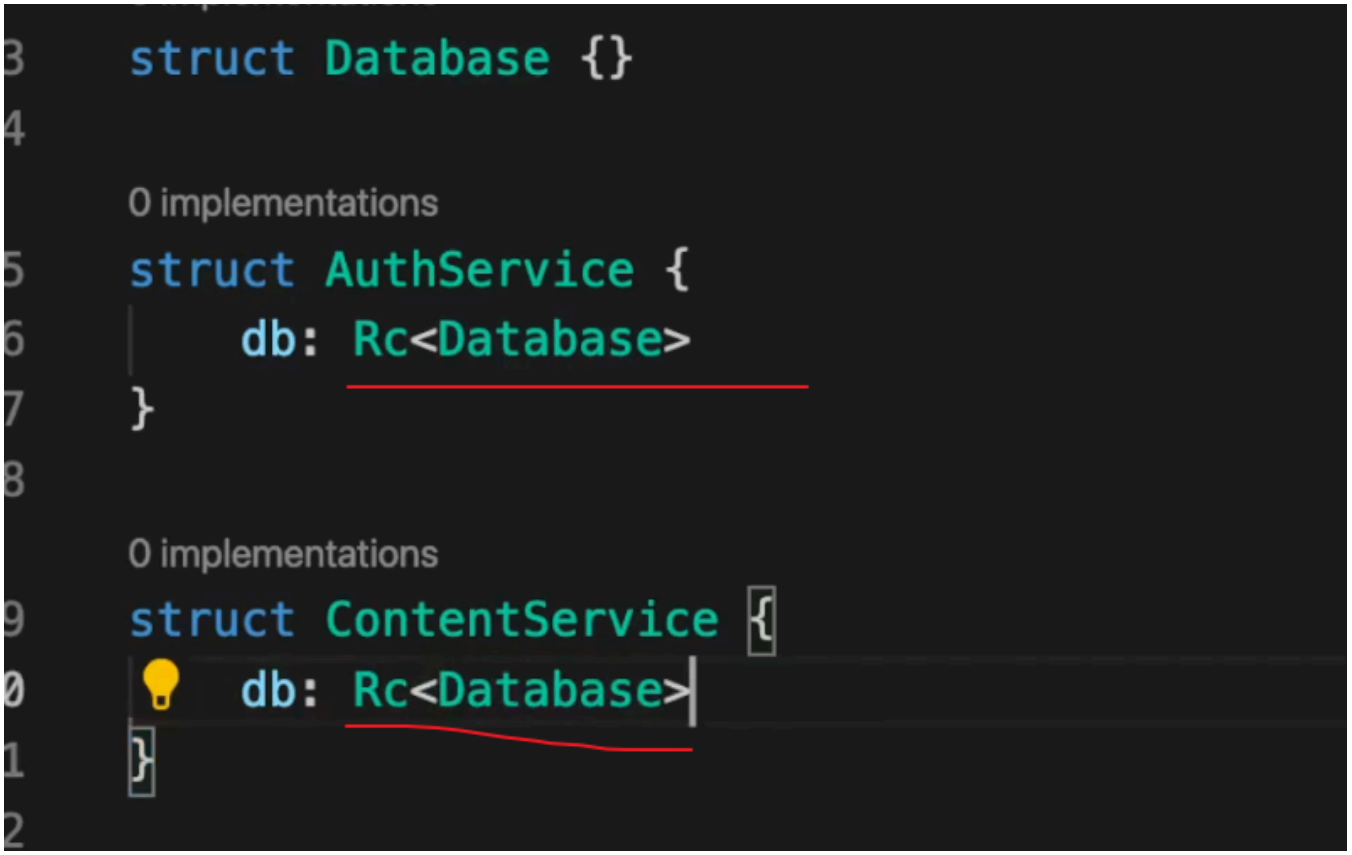
Rc

> 解决需要共享某些类型数据时候
>
> Rc 只能用于单线程应用
>
> Rc 类似C++中的 共享智能指针

```rust
struct Database {}

0 implementations
struct AuthService {
    db: Database
}
                                Auth 与 Content都需要使用Database

0 implementations
struct ContentService {
    db: Database
}
```



RefCell

> - RefCell使用了内部可变性（打破了rust编译时规则，所以内部使用了unsafe,然后外侧又包装了safe api,主要是某些内存节省场景)
>
>> **Rust编译器及借用规则是保守的， 有时候，代码本身是安全的，但开发者没法向rust编译器证明，所以提供类似*内部可变性*一类特性以便在必要时候绕过这些限制。**
>
> - 一般都会跟 Rc 结合使用：同时获得共享所有权和可变性
> - 当多次使用`borrow_mut()`，虽然程序编译能通过，但运行期会panic
>
>> **应该谨慎使用RefCell, 遵循所有权的责任在程序员身上**
>
> -

```rust
struct Database {
    max_connections: u32
}

0 implementations
struct AuthService {
    db: Rc<Database>
}

0 implementations
struct ContentService {
    db: Rc<Database>
}
```

```rust
▶ Run | Debug
fn main() {
    let db: Rc<Database> = Rc::new(Database {
        max_connections: 100
    });
    let auth_service: AuthService = AuthService { db: Rc::clone(self: &db) };
    let content_service: ContentService = ContentService { db: Rc::clone(self: &db) };
    db.max_connections = 200;    cannot assign to data in an `Rc` trait `DerefMut` is
}
```

RC SmartPoynter只允许值的不可变共享所有权。

The RC SmartPoynter only allows immutable shared ownership of a value.

> *注意此处Rc 内部嵌套RefCell: 因只用RefCell没法达到共享数据库服务目的*

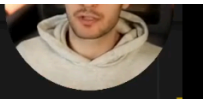
```rust
0 implementations
struct AuthService {
    db: Rc<RefCell<Database>>
}

0 implementations
struct ContentService {
    db: Rc<RefCell<Database>>
}
```

```rust
▶ Run | Debug
fn main() {
    let db: Rc<RefCell<Database>> = Rc::new(RefCell::new(Database {
        max_connections: 100
    }));
    let auth_service: AuthService = AuthService { db: Rc::clone(self: &db) };
    let content_service: ContentService = ContentService { db: Rc::clone(self: &db) };
    db.borrow_mut().max_connections = 200;
}
```

```rust
22
23      let mut r1: RefMut<Database> = db.borrow_mut();
24      let r2: RefMut<Database> = db.borrow_mut();
25      r1.max_connections = 200;
26  }
27
```

RefCell 多次获取可变引用，
编译时通过，但运行期会panic



PROBLEMS 5 OUTPUT TERMINAL DEBUG CONSOLE zsh

```
warning: `smart_pointers` (bin "smart_pointers") generated 5 warnings
    Finished dev [unoptimized + debuginfo] target(s) in 0.02s
     Running `target/debug/smart_pointers`
thread 'main' panicked at 'already borrowed: BorrowMutError', src/main.rs:24:17
note: run with `RUST_BACKTRACE=1` environment variable to display a backtrace
```

同样，应该谨慎使用refsels，因为遵循所有权规则的责任在程序员身上。