Traits

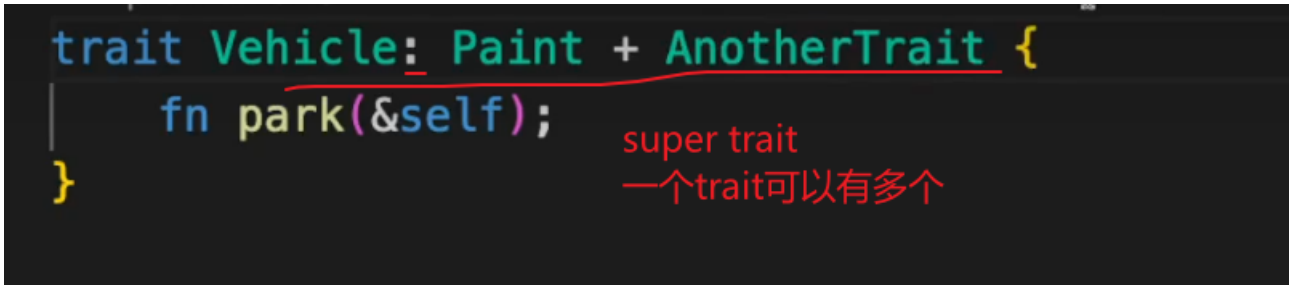
- Polymorphism- 多态

  > allow use to call methods on a interface without worrying about the concrete types that implement that interface

- Trait

  > share functionality and provide a common interface
  >
  > 因Rust中无继承，trait 类似 java中 interface
  >
  > Trait 与传统的继承不同，trait只能提供共享行为，而不能共享状态或数据
  >
  > 实现trait：`impl xxxTrait for object-x {}`
  >
  > trait可以有某个方法的默认实现，类似 java interface中可以有 default方法
  >
  > trait可以依赖其他trait， called super trait `trait Vehicle: Paint {}`, 类似java interface继承，达到约束某个实现实现该trait时必须实现其super trait

- Super trait



- Trait Bounds

  > - 用于约束参数类型
  > - 用于函数的返回值类型定义

  - 方式1

  - 

  - 方式2

  - 

- 方式3： 使用 where 语句

```
fn paint_vehicle_red<T>(object: &T) where T: Paint {
    object.paint(color: "red".to_owned());
}
```

- 函数返回值

Trait Object

> 涉及知识点: static dispatch 、dynamic dispatch
>
> Static Dispatch: 编译器在编译时已经知道要调用那个方法
>
> Dynamic Dispatch: 编译期间无法知道调用的是哪个方法

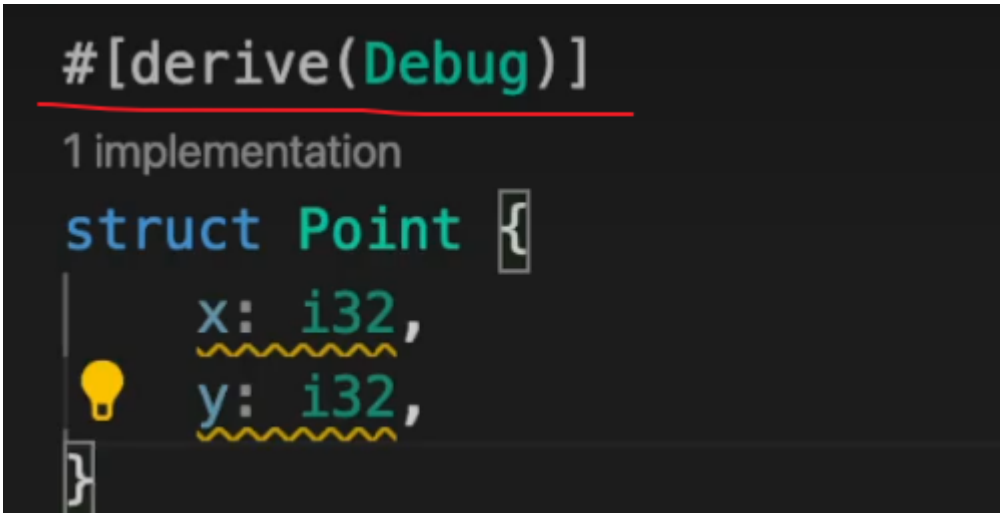
1. 集合中存放实现了一个trait的很多实例

```
let paintable_objects: Vec<&dyn Paint> = vec![&car, &house];
```

```
fn create_paintable_object(vehicle: bool) -> Box<dyn Paint> {
    if vehicle {
        Box::new(Car {
            info: VehicleInfo {
                make: "Honda".to_owned(),
                model: "Civic".to_owned(),
                year: 1995
            }
        })
    } else {
        Box::new(House {})
    }
}
```

Deriving Traits

> 常见：Debug，PartialEq



孤儿规则 - Orphan Rule

> 你只能为 **本地定义的类型** 实现 **本地定义的 trait**，或者为 **外部 trait**实现 **本地类型**，不能为 **外部类型实现外部 trait**
>
> Rust 的孤儿规则保证了 trait 实现的唯一性——你不能同时“扩展外部类型”和“使用外部 trait”，除非至少有一方是本地定义。
>
> **允许**：本地类型 + 外部 trait
>
> **允许**：本地类型 + 本地 trait
>
> **禁止**：外部类型 + 外部 trait
>
> **若确实需要，可以通过wrapper方式绕过该规则**

原因

- 防止 **冲突**：如果两个不同 crate 都给同一个外部类型实现同一个外部 trait，合并依赖时会冲突。
- 保证 **二义性安全**：编译器可以唯一确定某个 trait 的实现

对，这里的冲突指的是 **trait 实现的二义性**。详细说明：

假设：

- crate A 定义了 trait `TraitX`
- crate B 定义了类型 `TypeY`
- crate C 想实现 `TraitX` for `TypeY`

```rust
// crate C
impl TraitX for TypeY { ... }
```

问题在于：

1. crate A 也可能在未来版本里为 `TypeY` 提供实现：

   ```rust
   impl TraitX for TypeY { ... } // 来自 A crate
   ```

2. crate B 也可能在未来版本里提供：

   ```rust
   impl TraitX for TypeY { ... } // 来自 B crate
   ```

3. 当你的代码依赖了这些 crate 时，编译器 **无法判断到底该用哪一个实现**。Rust 编译期要求 **每个 trait 对每个类型只有唯一实现**，否则调用 `TraitX::foo()` 就不确定调用哪个。

孤儿规则通过要求 **至少有一个是本地 crate 的东西**，保证你可以安全地控制实现，不会出现这种冲突。

通过wrapper原始类型绕过孤儿规则