常见 Trait

jask

09/27/2024

什么是 Trait

Trait 机制是 Rust 中用于定义共享行为的一种重要特性。Trait 可以被视为一种接口，允许你为不同的类型定义共享的方法，提供了高度的灵活性和复用性。

Trait 约束： 在函数或结构体中，你可以指定某个类型必须实现特定的 Trait，以此来约束类型的行为（可以类比 C++ 的 Concept）

Trait 作为动态和静态分发： Rust 支持两种 Trait 使用方式：

静态分发：使用泛型（如上例），在编译时确定具体类型。

动态分发：使用 Trait 对象，通过引用或指针来实现（如 Box），在运行时确定具体类型。

常用 Trait

Default

```
trait Default{
  fn default()->Self;
}
```

其他一些地方用到了 Default，比如 Option 的 unwrap_or_default()，在类型参数上调用 default() 函数。

如果是 struct，还可以使用部分更新语法，这个时候其实是 Default 在发挥作用。

Rust 标准库实际给我们提供了一个标注，也就是 #[derive()] 里面放 Default，方便我们为结构体自动实现 Default trait。

这个 Default 不是 trait 而是一个同名的派生宏。这种派生宏标注帮助我们实现了 Default trait。Rustc 能正确区分 Default 到底是宏还是 trait，因为它们出现的位置不一样。

Display

```
trait Display{
  fn fmt(&self,f:&mut Formatter<'_>)->Result;
}
```

ToString

提供了一个 to_string 方法，实现了 Display 就实现了 ToString。

Debug

Debug 跟 Display 很像，也主要是用于调试打印。打印就需要指定格式，区别在于 Debug trait 是配对 “{:?}” 格式的,Display 是配对 “{}” 的。

它们本身都是将类型表示或转换成 String 类型。一般来说,Debug 的排版信息比 Display 要多一点，因为它是给程序员调试用的，不是给最终用户看的。Debug 还配套了一个美化版本格式 “{:#?}” ，用来把类型打印得更具结构化一些，适合调试的时候查看，比如 json 结构会展开打印。

PartialEq 和 Eq

Eq 定义为 PartialEq 的 subtrait，在 PartialEq 的对称性和传递性的基础上，又添加了自反性，也就是对所有 a 都有 a == a。最典型的就是 Rust 中的浮点数只实现了 PartialEq，没实现 Eq，因为根据 IEEE 的规范，浮点数中存在一个 NaN，它不等于自己，也就是 NaN ≠ NaN。而对整数来说,PartialEq 和 Eq 都实现了。

运算符重载

std::ops 定义了 Add, Mul, Sub 等 Trait, 实现这些 Trait 从而实现重载。

Clone 和 Copy

```
trait Copy: Clone{}
```

定义为 Clone 的 subtrait, 并且不包含任何内容, 仅仅是一个标记 (marker)。

Rust 提供了 Copy 过程宏，可以自动实现 Copy trait。

Copy 和 Clone 的区别是,Copy 是浅拷贝只复制一层, 不会去深究这个值里面是否有到其他内存资源的引用, 比如一个字符串的动态数组。

一旦一个类型实现了 Copy, 它就会具备一个特别重要的特性: 再赋值的时候会复制一份自身。那么就相当于新创建一份所有权。

Copy 在 Rust 中需要手动指定。

而在所有权的第一设计原则框架下,Rust 默认选择了 move 语义。所以方便起见,Rust 设计者就只让最基础的那些类型, 比如 u32、bool 等具有 copy 语义。而用户自定义的类型, 一概默认 move 语义。如果用户想给自定义类型赋予 copy 语义内涵, 那么他需要显式地在那个类型上添加 Copy 的 derive。

ToOwned

ToOwned 相当于是 Clone 更宽泛的版本。ToOwned 给类型提供了一个 to_owned() 方法, 可以将引用转换为所有权实例。

Deref

Deref trait 可以用来把一种类型转换成另一种类型, 但是要在引用符号 &、点号操作符. 或其他智能指针的触发下才会产生转换。

还有 &Vec 可以自动转换为 &[T], 也是因为 Vec[T] 实现了 Deref。

Drop

Drop trait 用于给类型做自定义垃圾清理 (回收)。

一般来说, 我们不需要为自己的类型实现这个 trait, 除非遇到特殊情况, 比如我们要调用外部的 C 库函数, 然后在 C 那边分配了资源, 由 C 库里的函数负责释放, 这个时候我们就要在 Rust 的包装类型 (对 C 库中类型的包装) 上实现 Drop, 并调用那个 C 库中释放资源的函数。

一般指在 FFI 中有用。

闭包相关 trait

标准库中有 3 个 trait 与闭包相关, 分别是 FnOnce、FnMut、Fn。你可以看一下它们的定义。

```
trait FnOnce<Args>{
  type Output;
  fn call_once(self,args: Args)->Self::Output;
}
trait FnMut<Args>: FnOnce<Args>{
  fn call_mut(&mut self,args: Args)->Self::Output;
}
trait Fn<Args>: FnMut<Args>{
  fn call(&self,args: Args)->Self::Output;
}
```

这里有闭包的三种行为:

1. 获取了上下文环境变量的所有权, 对应 FnOnce。
2. 只获取了上下文环境变量的 &mut 引用, 对应 FnMut。
3. 只获取了上下文环境变量的 & 引用, 对应 Fn。

根据这三种不同的行为,Rust 编译器在编译时把闭包生成为这三种不同类型中的一种。

这三种不同类型的闭包, 具体类型形式我们不知道,Rust 没有暴露给我们。但是 Rust 给我们暴露了 FnOnce、FnMut、Fn 这 3 个 trait, 就刚好对应于那三种类型。结合我们前面讲到的 trait object, 就能在我们的代码中对那些类型进行描述了。

FnMut 代表的闭包类型能被调用多次, 并且能修改上下文环境变量的值, 不过有一些副作用, 在某些情况下可能会导致错误或者不可预测的行为。

Fn 代表的这类闭包能被调用多次, 但是对上下文环境变量没有副作用。

From 和 Into

```
trait From<T>{
  fn from(T)->Self;
}
trait Into<T>{
  fn into(self)->T;
}
```

From 可以把类型 T 转为自己，而 Into 可以把自己转为类型 T。

可以看到它们是互逆的 trait。实际上,Rust 只允许我们实现 From，因为实现了 From 后，自动就实现了 Into。

其实 From 是单向的。对于两个类型要互相转的话，是需要互相实现 From 的。

本身,From 和 Into 都隐含了所有权,From 的 Self 是具有所有权的,Into 的 T 也是具有所有权的。Into 有个常用的比 From 更自然的场景是，如果你已经拿到了一个变量，想把它变成具有所有权的值,Into 写起来更顺手。因为 into() 是方法，而 from() 是关联函数。

AsRef

```
trait AsRef<T>{
  fn as_ref(&self)->&T;
}
```

它把自身的引用转换成目标类型的引用。和 Deref 的区别是,**deref() 是隐式调用的，而 as_ref() 需要你显式地调用** 。

AsRef 可以让函数参数传入的类型更加多样化，不管是引用类型还是具有所有权的类型都可以传递。

可以把 Deref 看成是隐式化 (或自动化)+ 弱化版本的 AsRef。