成为 Rustacean1

jask

09/27/2024

Trait 使用方式

Trait 的主要作用是用来抽象行为，类似于其他编程语言中的「接口」

根据函数调用方式，主要两种使用方式：

基于范型的静态派发

基于 trait object 的动态派发

比较重要的一点是 trait object 属于 Dynamically Sized Types（DST），在编译期无法确定大小，只能通过指针来间接访问，常见的形式有 Box &dyn trait 等。

静态派发

在 Rust 中，范型的实现采用的是单态化（monomorphization），会针对不同类型的调用者，在编译时生成不同版本的函数，所以范型也被称为类型参数。好处是没有虚函数调用的开销，缺点是最终的二进制文件膨胀。

动态派发

不是所有函数的调用都能在编译期确定调用者类型，一个常见的场景是 GUI 编程中事件响应的 callback，一般来说一个事件可能对应多个 callback 函数，而这些 callback 函数都是在编译期不确定的，因此范型在这里就不适用了。

向上转型（upcast）

对于 trait SubTrait: Base ，在目前的 Rust 版本中，是无法将 &dyn SubTrait 转换到 &dyn Base 的。这个限制与 trait object 的内存结构有关。

在 Exploring Rust fat pointers 一文中，该作者通过 transmute 将 trait object 的引用转为两个 usize，并且验证它们是指向数据与函数虚表的指针：

```rust
use std::mem::transmute;
use std::fmt::Debug;

fn main() {
    let v = vec![1, 2, 3, 4];
    let a: &Vec<u64> = &v;
    // 转为 trait object
    let b: &dyn Debug = &v;
    println!("a: {}", a as *const _ as usize);
    println!("b: {:?}", unsafe { transmute::<_, (usize, usize)>(b) });
}

// a: 140735227204568
// b: (140735227204568, 94484672107880)
```

从这里可以看出：Rust 使用 fat pointer（即两个指针）来表示 trait object 的引用，分布指向 data 与 vtable，这和 Go 中的 interface 十分类似。

尽管 fat pointer 导致指针体积变大（无法使用 Atomic 之类指令），但是好处是更明显的：

可以为已有类型实现 trait（比如 blanket implementations）

调用虚表中的函数时，只需要引用一次，而在 C++ 中，vtable 是存在对象内部的，导致每一次函数调用都需要两次引用，如下图所示：

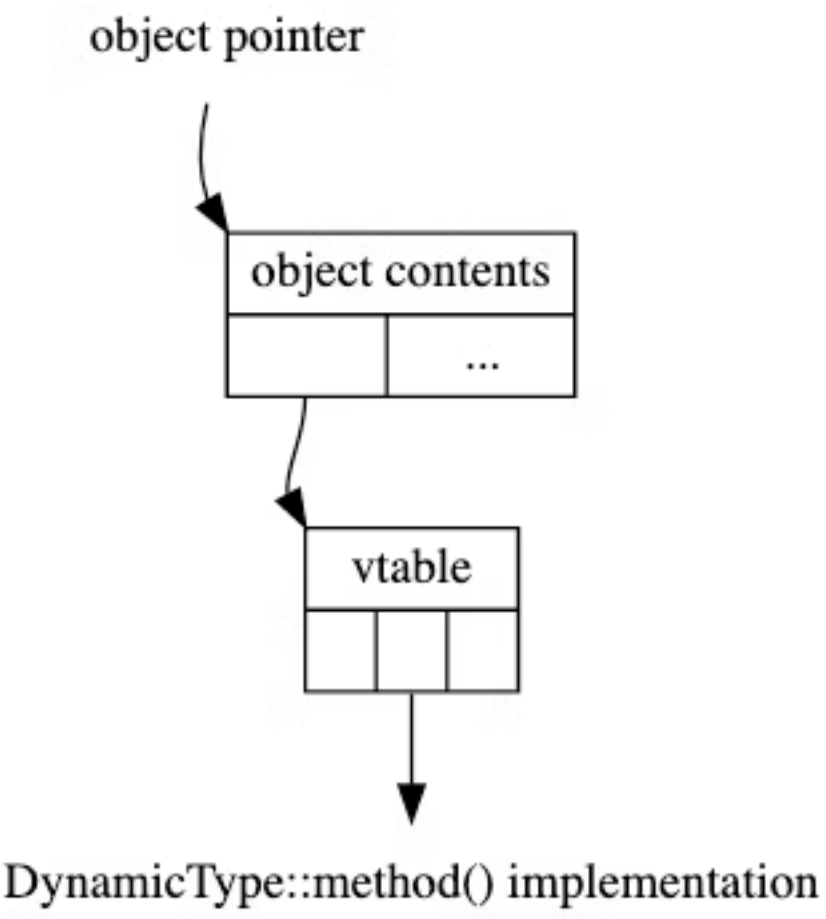
object pointer
object contents
...
vtable
DynamicType::method() implementation

Figure 1: 如图

如果 trait 有继承关系时，vtable 是怎么存储不同 trait 的方法的呢？在目前的实现中，是依次存放在一个 vtable 中的：

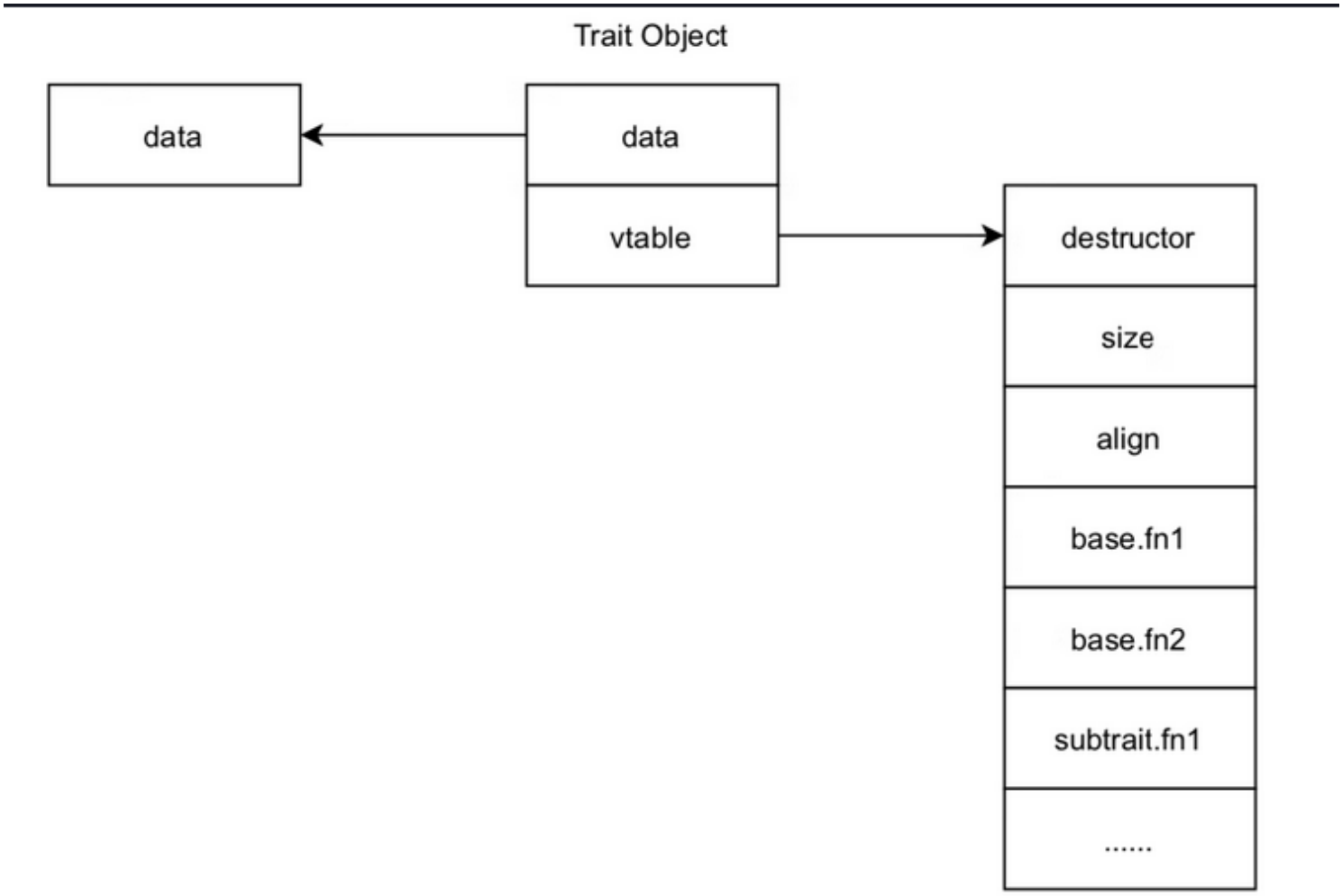

Figure 2: 如图

可以看到，所有 trait 的方法是顺序放在一起，并没有区分方法属于哪个 trait，这样也就导致无法进行 upcast

Object Safety

在 Rust 中，并不是所有的 trait 都可用作 trait object，需要满足一定的条件，称之为 object safety 属性。

1. 函数返回类型不能是 Self（即当前类型）。这主要因为把一个对象转为 trait object 后，原始类型信息就丢失了，所以这里的 Self 也就无法确定了。
2. 函数中不允许有范型参数。主要原因在于单态化时会生成大量的函数，很容易导致 trait 内的方法膨胀。

例如：

```
#![allow(unused)]
fn main() {
trait Trait {
    fn foo<T>(&self, on: T);
    // more methods
}

// 10 implementations
fn call_foo(thing: Box<Trait>) {
    thing.foo(true); // this could be any one of the 10 types above
    thing.foo(1);
    thing.foo("hello");
```

```
}

// 总共会有 10 * 3 = 30 个实现
}
```

3. Trait 不能继承 Sized。这是由于 Rust 会默认为 trait object 实现该 trait，生成类似下面的代码：

```
#![allow(unused)]
fn main() {
trait Foo {
    fn method1(&self);
    fn method2(&mut self, x: i32, y: String) -> usize;
}

// autogenerated impl
impl Foo for TraitObject {
    fn method1(&self) {
        // `self` is an `&Foo` trait object.

        // load the right function pointer and call it with the opaque data pointer
        (self.vtable.method1)(self.data)
    }
    fn method2(&mut self, x: i32, y: String) -> usize {
        // `self` is an `&mut Foo` trait object

        // as above, passing along the other arguments
        (self.vtable.method2)(self.data, x, y)
    }
}
}
```

如果 Foo 继承了 Sized，那么就要求 trait object 也是 Sized，而 trait object 是 DST 类型，属于?Sized ，所以 trait 不能继承 Sized。

对于非 safe 的 trait，能修改成 safe 是最好的方案，如果不能，可以尝试范型的方式。