【社区投稿】自动特征auto trait的扩散规则

原创 爱国的张浩予 Rust语言中文社区 2024年12月25日 23:44 重庆

自动特征`auto trait`的扩散规则

公式化地概括， `auto trait = marker trait + derived trait` 。其中，等号右侧的 `marker` 与 `derived` 是在Rustonomicon书中的引入的概念，鲜见于Rust References。所以，若略感生僻，不奇怪。

`marker trait` 与 `derived trait` 精准概括了 `auto trait` 功能的两面性

1. 前者指明 `auto trait` **实现类**具备了由 `rustc` 编译器和 `std` 标准库对其约定的“天赋异能 intrinsic properties”。
2. 后者描述了这些“天赋异能”沿 `auto trait` **实现类**数据结构【自内及外】的继承性与扩散性。

接下来逐一解释。

`marker trait`标识“天赋”特征是什么

既然是“**天赋**”，那么 `auto trait` 就**没有**任何抽象成员方法待被“**后天**实现”或关联项待被“**后天**赋值” — 这也是 `marker trait` 别名的由来。 `rustc` 甚至未配备专项检查器以静态分析与推断 @Rustacean 对 `auto trait` 的实现是否合理。类似于 `unsafe` 块，@Rustacean 需向 `rustc` 承诺：知道自已正在干什么，和提交**可供其它程序模块信任**的“天赋异能“代码实现。否则，运行时程序就会执行出未定义行为 U.B.。相比于传统的 `std` 程序接口和 `rustc` 内存安全承诺，这是一项“反向契约” — 即，由 `rustc` 充当“甲方”和规划功能要求，而由 @Rustacean 充当“乙方”完成功能代码和提供正确性保证。在硕大的 `Rust` 标准库中，这类“天赋异能“的”反向契约”并不多见，但包括

auto trait	天赋异能
std::marker::Send	自定义数据结构的"跨线程"（所有权转移）数据复制 — 传值。"所有权转移"意味着一旦当前线程内的变量值被传给另一线程，那么当前线程上下文就**再也不能**访问该变量的值了。
std::marker::Sync	自定义数据结构的"跨线程"内存共享 — 传引用。此外，由 trait Send 至 trait Sync 的转换关系可概括为：若 <&T: Send>，那么 <T: Sync>。
std::marker::Unpin	鉴于内存效率优先原则，`trait Unpin`实现类并**不承诺**其实例总是被锚定于内存中的预定**位置不动**。即，为了减少内存碎片的数量，`trait Unpin`实现类对象会在其生命周期内被来回腾挪于【栈】内存各处 — 此处理强调【栈】是因为`Rust`对【堆】内存不会做这类腾挪处理。再加之`rustc`会为所有数据类型都自动实现`trait Unpin`特征，所以`trait Unpin`的正面教条定义几乎毫无用处。相反，它的"否定"形式`!Unpin`和`impl !Unpin for T {}`才是 @Rustacean 在处理**"自引用 self-referential"数据结构**（和异步编程）的刚性需求 — 禁止自引用结构体实例在内存中腾挪移动和防范**脏指针**。此处仅对该话题抛砖引玉，不再展开了。
std::panic::UnwindSafe	当`std::panic::catch_unwind(闭包)`被用来监控执行闭包内的（受控）程序崩溃 unwinding panic 时，被监控闭包的捕获变量都必须实现此特征。
std::panic::RefUnwindSafe	

输入图片说明

`derived trait`明确“天赋”特征如何扩散

概括起来， `auto trait` 的扩散规则就四个字“**由内及外**”。其遇到不同的场景，伴有不同解释的扩散链条

场景一：变量 → 指针

以变量的数据类型为内，和以指向该变量值的指针/引用为外 — 变量值（数据类型 `T` ）实现的 `auto trait` 会自动扩散至它的各类指针与引用：

- `&T`
- `&mut T`
- `*const T`
- `*mut T`

所以，该扩散链条也被记作【类型 → 指针】。

场景二：字段 → 结构体

以字段的数据类型为内，和以父数据结构为外 — **所有**字段（数据类型）都实现的 `auto trait` 会自动扩散至它们的紧上一层数据结构：

- `structs`
- `enums`
- `unions`
- `tuples`

所以，该扩散链条也被记作【字段 → 结构】。

场景三：元素 → 集合

以集合元素的数据类型为内，和以集合容器为外 — 由元素（数据类型 `T` ）实现的 `auto trait` 会自动扩散至该元素的紧上一层集合容器：

- `[T; n]`
- `[T]`
- `Vec<T>`

场景四：捕获变量 → 闭包

以捕获变量的数据类型为内，和以闭包为外 — **所有**捕获变量（数据类型）都实现的 `auto trait` 会自动扩散至引用（或所有权占用）这些捕获变量的闭包。

场景五：函数 → 函数指针

函数项 `fn` 与函数指针 `fn ptr` **总是**会被 `rustc` 编译时自动实现**全部** `auto trait` 。

扩散链条的“串连”

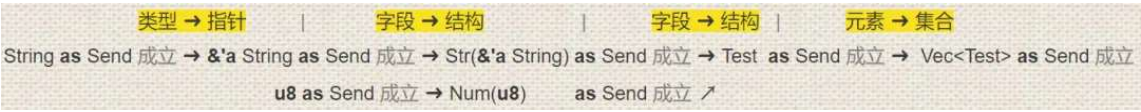
前四个场景的扩散链是可以**多重嵌套衔接**的。举个例子， `Vec<Wrapping<u8>>` 一定满足 `trait Send` 限定条件，因为这条扩散链条的存在：

输入图片说明

再举个更复杂的例子，假设有如下枚举类

```
enum Test<'a> {
    Str(&'a String),
    Num(u8)
}
```

那么 `Vec<Test>` 也一定满足 `trait Send` 限定条件，因为此扩散链条的存在：



输入图片说明

`auto trait`扩散链条的“阻断”

1. 安装 `nightly` 版 `rustc` 编译器。然后，在代码中，
2. 开启 `#![feature(negative_impls)]` 的 `feature-gate` **编译开关**，
3. **否定**实现 `auto trait` 。比如， `impl !Unpin for Test {}`

于是， `rustc` 就不会再对遇到的【类型定义】自动添加曾被否定实现过的 `auto trait` 了。在众多 `auto trait` 中，仅 `trait Unpin` 可**绕过**对 `nightly` 编译工具链的依赖和仅凭 `stable` 标准库内符号类型 `std::marker::PhantomPinned` 定义的**幻影字段**阻止 `rustc` 悄悄地实现自动特征。

> 【幻影字段】是仅作用于编译时的**零成本抽象**项。它被用来帮助编译器理解 @Rustacean 提交的代码和推断 @Rustacean 的程序设计意图。其语义功能很像 `typescript` 中的【@ 装饰器】。即，
>
> 1. 辅助代码静态分析
> 2. 辅助编译器生成垫片程序
> 3. 编译后立即抹除
> 4. 对【运行时】不可见 — 这也是【零成本】的由来。但，世间任何事物都有两面性和是双刃剑。“零成本”是省 `CPU` ，但更费脑细胞呀！ `Rust` 编程的心智成本高已是行

> 业共识了。
>
> 另值一提的是，【 Rust 幻影字段】与【 typescript 装饰器】皆都不同于【 Java 的 @ 注释】，因为
>
> - 前者是给**编译器**看和解读的 — 充其量是代码正文的旁白注脚。
> - 后者是给运行时 VM 用和执行的 — 这已算是正文指令的一部分了。
>
> 它们就是两个不同“位面”的东西。
>
> 我日常仅用过 std::marker::PhantomPinned 与 std::marker::PhantomData 两类**幻影字段**
>
> - 前者解决 stable 编译工具链对 trait Unpin 的否定实现 — 就是这里正在讲的事
> - 后者被用于“类型状态 Type State Pattern ”设计模式中，将【运行时】对象状态的（动态）信息编码入【编译时】对象类型的（静态）定义里，以扩展 Rust 类型系统的应用场景至对状态集的“状态管理”。若您对“类型状态”设计模式有兴趣，推荐移步至我的另一篇主题文章对照 OOP 浅谈【类型状态】设计模式保证有收获。
>
> 这闲篇扯远了，让咱们重新回到文章的主题上来。

请细读下面**自引用结构体**的类型定义（特别含注释内容）和体会 std::marker::PhantomPinned 如何被用来声明结构体内的幻影字段：

```
use ::std::marker::PhantomPinned;
struct SelfReferential {
    // 整个结构体内唯一包含了有效信息的字段。
    a: String,
    // 自引用前一个字段`a`的值。
    ref_a: *const String,
    // 1. 这是对【幻影字段】的定义
    // 2. 因为该字段并不会真的被后续功能代码用到，
    //    所以为了压制来自编译器的`useless field`警告，
    //    字段名以`_`为前缀
    _marker: PhantomPinned
} // 于是，自引用结构体`Test`就是 !Unpin 的了
```

即便不太理解，也请不要质疑【自引用结构体】的存在必要性。至少在**异步**程序块中，跨 .await 轮询点的变量引用都依赖这套机制。仅因为 async {} 语法糖把每次构造 trait Future 实现类的细节都隐藏了起来，所以 @Rustacean 对自引用结构体的直观感受会比较弱。

auto trait扩散不至的自定义数据结构

若数据结构定义内含有**未**实现 auto trait 的字段（比如，

```
use ::std::env::Vars;
struct Dumb(Vars);
```

其中 `Dumb.0` 字段 `Vars` 明确是 `!Send` 的），那么 @Rustacean 就有必要考虑

- 要么，重新规划程序设计，以规避要求 `struct Dumb` 满足 `trait Send` 限定条件
- 要么，给 `struct Dumb` 添加 `unsafe` 的 `auto trait` 实现块。

```
unsafe impl Send for Dumb {};
```

后者的 `unsafe impl` 语法前缀就是 `rustc` 对程序作者最后的警告：“你真的明白，你正在做什么事吗？”。

【快排序】综合例程

先贴源码，再做详解。敲黑板强调：代码内的注释同样重要呀！推荐同正文一样重视和仔细阅读。

```
fn quick_sort<T: Ord + Send>(v: &mut [T]) {
    if v.len() <= 1 {
        return;
    }
    let mid = {
        let pivot = v.len() - 1;
        let mut i = 0;
        for j in 0..pivot {
            if v[j] <= v[pivot] {
                v.swap(i, j);
                i += 1;
            }
        }
        v.swap(i, pivot);
        i
    };
    // 1. 此处，虽然 lo 与 hi 是两个崭新的【切片】胖指针实例，
    //    但由【切片】胖指针引用的底层 Vec<i32> 数据值却只有
    //    一份呀！
    let (lo, hi) = v.split_at_mut(mid);
    // 2. 所以，后续的【多线程+递归】是修改的同一个 Vec<i32>
    //    实例。
    rayon::join(|| quick_sort(lo),
```

```
                || quick_sort(hi));
    // 3. 至此，虽然此函数没有返回值，但仍可沿函数的【输入输出】
    //    实参 v: &mut [T] 向调用端传递排序结果。
}
fn main() {
    use ::rand::prelude::*;
    // 1. 生成一个大数字集合
    let mut numbers: Vec<i32> = (1000..10000).collect();
    // 2. 乱序数组内容
    numbers.shuffle(&mut rand::thread_rng());
    // 3. 多线程快排序
    quick_sort(&mut numbers);
    // 4. 打印排序结果
    println!("Sorted: {:?}", &numbers[..10]);
    // 5. 一个单例 Vec<i32> 对象贯穿整个多线程快排序 demo 始终。
}
```

上述例程的难点并不是 `rayon::join()` 如何在后台悄悄唤起多个线程加速大数据集【快排序】— 这不需要 @Rustacean 操心，咱们只要多读读 API 文档和了解Work Stealing工作原理就足够了。相反，曾经困惑过我一段时间的痛点是：

为什么虽然泛型类型参数 `<T: Ord + Send>` 的书面语义是“跨线程·数据复制”但程序的实际执行结果却是对 `Vec<i32>` 单例的“跨线程·内存共享”？即，多个线程透过【切片】胖指针，修改同一个 `Vec<T>` 变长数组实例。

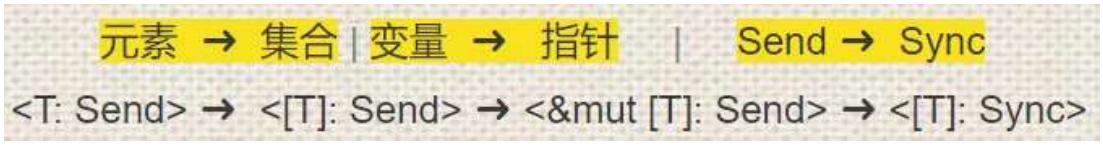
推导明白这条逻辑链（元素 `Send` ➔ 集合 `Sync` ）先后耗费了我不少心神。但总结起来也无非如下几步：

1. 依据前文介绍的 `auto trait` 扩散规则，对特征 `trait Send` 和泛型类型参数 `T` ，构造初始扩散链条：

输入图片说明

2. 依据 `trait Send` 至 `trait Sync` 的（单向）转换关系，有 `<&S: Send> → <S: Sync>` 。

3. 将 #2 代入 #1，进一步完善 `trait Send` 扩散链条，有



输入图片说明

4. 依据 `rustc` 赋予 `trait Sync` 的语义，集合 `[T]` 被允许跨线程引用与多线程共享

5. 又因为 `fn quick_sort()` 的形参是对 `Vec<i32>` 实例的可修改引用 `&mut` ，所以多个线程被允许并行修改同一个变长数组实例。

你不会以为故事就此结束了吧？难道你没有发觉例程中多线程代码有缺了点儿什么的异样吗？没错！细心的读者可能早就想问：

对单实例变长数组的**并行修改**，为什么未采用【读写锁 `RwLock` 】或【互斥锁 `Mutex` 】加以同步保护呢？甚至 `rustc` 在编译时连警告提示都没有输出？What's wrong?

好问题！您有心了。快速回答是：虽然 `Vec<i32>` 实例同时被多个线程并行修改不假，但每个线程并行修改的切片却只是同一变长数组内**彼此衔接却并不相交的**“子段”。所以，在快排序过程中，事实上**没有**任何数据竞争发生 — 这是彻头彻尾的**算法胜利**。果真，编程的尽头是数学与算法啊！此外， `rustc` 能顺利地接受与成功地编译这样的代码也足已破除人们以往对它保守且不变通的刻板印象。

结束语

这次就先分享这一个小知识点。文章里埋的有关 `Unpin` 的坑以后再填。2024搞了一年的“鸿蒙 `Next ArkTs` ”真不容易。哎！天大地大，饭辙最大。我是一颗螺丝钉，甲方爸爸需要什么，我就研究什么。但，年底写篇 `Rust` 知识分享文章压压惊。

Rust投稿分享 25

Rust投稿分享 · 目录

上一篇 · 【社区投稿】给 NdArray 装上 CUDA 的轮子

阅读原文 阅读 1427

写留言

留言 15

渡鸦 天津 2小时前

敬佩你的专注和专业，加油啊兄弟

❤️🍆的大君 天津 2小时前

致敬大佬

豚鼠 天津 4小时前

撤恩 北京 8小时前

当且仅当 T 是 Sync，&T 是 Send 与 Sync，&mut T 是 Sync
当且仅当 T 是 Send，&mut T 是 Send

开心的人 北京 9小时前

新知识点+1，作者大大请保持输出，持续分享

Absalom 重庆 9小时前

学习学习，感觉好深奥

王鹏 北京 9小时前

仔细深入，膜拜大佬~~~

Nuyoah 北京 9小时前

膜拜大佬

吉祥茹意 天津 9小时前

确实很是努力用心，一定会有收获的，努力，努力，再努力！

momo 广东 17小时前

T: Send 能推出 [T]: Sync ? 感觉很不科学。quick_sort 能成是因为 split_at_mut 把一个 切片分成不重叠两部分，递归拆分下去也是不重叠的，根本埠需要Sync。

谢银峰 重庆 18小时前

大佬在重庆吗？希望可以交流一下rust学习心得

洋仔 四川 2小时前

加好友交流下，哈哈

谢银峰 重庆 1小时前

回复 洋仔 : 最近下班有时间就研究一下，遇到了很多麻烦事，正在寻找最佳实践，已经rust的编程应该遵循那种方式，不然全是坑，这谁顶得住

2条回复