所有权与智能指针

jask

09/27/2024

Rust 中智能指针的概念也直接来自于 C++。C++ 里面有 unique_ptr、shared_ptr。

Box

Box 是一个类型整体，作为智能指针 Box 可以把资源强行创建在堆上并且获得资源的所有权。

Box 的所有权分析

在创建 Box 实例的时候会发生所有权转移：资源从栈上 move 到了堆上，原来栈上的那片资源被置为无效状态。

Box 的解引用

创建一个 Box 实例把栈上的内容包起来，可以把栈上的值移动到堆上。

还可以在 Box 实例上使用解引用符号 *，把里面的堆上的值再次移动回栈上。

解引用是 Box::new() 的逆操作，可以看到整个过程是相反的。

对于具有 copy 语义的 u8 类型来说，解引用回来后,boxed 还能使用。

而对于具有 move 语义的类型来说，情况就不一样了，会发生所有权的转移。比

Box 实现了 trait

在栈上还是堆上这种细节问题了。一种类型，被 Box<> 包起来的过程就叫作这个类型的盒化 (boxed)。

Box Rust 在标准库里为 Box 实现了 Deref、Drop、AsRef 等 trait，所以 Box 可以直接调用 T 实例的方法。

Box Clone

Box 能否 Clone，需要看 T 是否实现了 Clone。

Box

trait object，它代表一种类型，这种类型可以代理一批其他的类型。但是 dyn trait 本身的尺寸在编译期是未知的，所以 dyn trait 的出现总是要借助于引用或智能指针。而 Box 是最常见的，甚至比 &dyn trait 更常见。原因就是 Box 拥有所有权，这就是 Box 方便的地方，而 &dyn Trait 不拥有所有权，有的时候就没那么方便。

如果 dyn trait 出现在结构体里，那么 Box 形式就比 &dyn trait 形式要方便得多。

Arc< T >

Arc< T > 是共享所有权模型的智能指针，也就是 shared_ptr。

clone

Arc 的主要功能是和 clone() 配合使用。在 Arc 实例上每一次新的 clone() 操作，总是会将资源的引用数 +1，而保持原来那一份资源不动，这个信息记录在 Arc 实例里面。每一个指向同一个资源的 Arc 实例走出作用域，就会给这个引用计数 -1。直到最后一个 Arc 实例消失，目标资源才会被销毁释放。

Arc< T > 的克隆行为只为改变引用计数不会克隆里面的内容。

值的修改

多所有权条件下，怎么修改 Arc 里面的值呢？答案是不能修改。虽然 Arc 是拥有所有权的，但 Arc 不提供修改 T 的能力，这也是 Arc 和 Box 不一样的地方。后面我们在并发编程部分会讲到 Mutex、RwLock 等锁。想要修改 Arc 里面的内容，必须配合这些锁才能完成，比如 Arc<Mutex>。

共享所有权的场景下，如果任意一方能随意修改被包裹的值，那就会影响其他所有权的持有者，整个就乱套了。所以要修改的话必须引入锁的机制。

Arc 和不可变引用 & 的区别

首先，它们都是共享对象的行为，本质上都是指针。但 Arc 是共享了所有权模型，而 & 只是共享借用模型。共享借用模型就得遵循借用检查器的规则——借用的有效性依赖于被借用资源的 scope。对于这个的分析是非常复杂的。而所有权模型是由自己来管理资源的 scope，所以处理起来比较方便。

Rc 和 Arc

Arc 和 Rc 之间的区别在与线程安全，Rc 相对而言并没有保证线程安全，允许更快的更新引用计数。

Rc 用于当我们希望在堆上分配一些内存供程序的多个部分读取，而且无法在编译时确定程序的哪一部分会最后结束使用它的时候。如果确实知道哪部分是最后一个结束使用的话，就可以令其成为数据的所有者，正常的所有权规则就可以在编译时生效。

RefCell 和内部可变性

内部可变性（Interior mutability）是 Rust 中的一个设计模式，它允许你即使在有不可变引用时也可以改变数据，这通常是借用规则所不允许的。为了改变数据，该模式在数据结构中使用 unsafe 代码来模糊 Rust 通常的可变性和借用规则。不安全代码表明我们在手动检查这些规则而不是让编译器替我们检查。第十九章会更详细地介绍不安全代码。

当可以确保代码在运行时会遵守借用规则，即使编译器不能保证的情况，可以选择使用那些运用内部可变性模式的类型。所涉及的 unsafe 代码将被封装进安全的 API 中，而外部类型仍然是不可变的。

通过 RefCell 在运行时检查借用规则

不同于 Rc，RefCell 代表其数据的唯一的所有权。那么是什么让 RefCell 不同于像 Box 这样的类型呢？

回忆一下借用规则：

1. 在任意时刻，只能拥有一个可变引用或任意数量的不可变引用（二者至多一个）
2. 引用必须总是有效的。

对于引用和 Box，借用规则的不可变性作用于编译时。对于 RefCell，这些不可变性作用于运行时。对于引用，如果违反这些规则，会得到一个编译错误。而对于 RefCell，如果违反这些规则程序会 panic 并退出。

在编译时检查借用规则的优势是这些错误将在开发过程的早期被捕获，同时对运行时没有性能影响，因为所有的分析都提前完成了。为此，在编译时检查借用规则是大部分情况的最佳选择，这也正是其为何是 Rust 的默认行为。

类似于 Rc，RefCell 只能用于单线程场景。如果尝试在多线程上下文中使用 RefCell，会得到一个编译错误。

如下为选择 Box，Rc 或 RefCell 的理由：

Rc<T> 允许相同数据有多个所有者；Box<T> 和 RefCell<T> 有单一所有者。
Box<T> 允许在编译时执行不可变或可变借用检查；Rc<T>仅允许在编译时执行不可变借用检查；RefCell<T> 允许在运行时执行不
因为 RefCell<T> 允许在运行时执行可变借用检查，所以我们可以在即便 RefCell<T> 自身是不可变的情况下修改其内部的值。

在不可变值内部改变值就是内部可变性模式。让我们看看何时内部可变性是有用的，并讨论这是如何成为可能的。

内部可变性：不可变值的可变借用

然而，特定情况下，令一个值在其方法内部能够修改自身，而在其他代码中仍视为不可变，是很有用的。值方法外部的代码就不能修改其值了。RefCell 是一个获得内部可变性的方法。RefCell 并没有完全绕开借用规则，编译器中的借用检查器允许内部可变性并相应地在运行时检查借用规则。如果违反了这些规则，会出现 panic 而不是编译错误。

RefCell< T > 在运行时记录借用

当创建不可变和可变引用时，我们分别使用 & 和 &mut 语法。对于 RefCell 来说，则是 borrow 和 borrow_mut 方法，这属于 RefCell 安全 API 的一部分。borrow 方法返回 Ref 类型的智能指针，borrow_mut 方法返回 RefMut 类型的智能指针。这两个类型都实现了 Deref，所以可以当作常规引用对待。

RefCell 记录当前有多少个活动的 Ref 和 RefMut 智能指针。每次调用 borrow，RefCell 将活动的不可变借用计数加一。当 Ref 值离开作用域时，不可变借用计数减一。就像编译时借用规则一样，RefCell 在任何时候只允许有多个不可变借用或一个可变借用。

Rc 和 RefCell 组合使用

在 Rust 中，一个常见的组合就是 Rc 和 RefCell 在一起使用，前者可以实现一个数据拥有多个所有者，后者可以实现数据的可变性：

举例：

```
use std::cell::RefCell;
use std::rc::Rc;
fn main() {
    let s = Rc::new(RefCell::new(" 我很善变，还拥有多个主人".to_string()));

    let s1 = s.clone();
    let s2 = s.clone();
    // let mut s2 = s.borrow_mut();
    s2.borrow_mut().push_str(", oh yeah!");

    println!("{:?}\n{:?}\n{:?}", s, s1, s2);
}
```

上面代码中，我们使用 RefCell 包裹一个字符串，同时通过 Rc 创建了它的三个所有者：s、s1 和 s2，并且通过其中一个所有者 s2 对字符串内容进行了修改。

由于 Rc 的所有者们共享同一个底层的数据，因此当一个所有者修改了数据时，会导致全部所有者持有的数据都发生了变化。

性能损耗

相信这两者组合在一起使用时，很多人会好奇到底性能如何，下面我们来简单分析下。

首先给出一个大概的结论，这两者结合在一起使用的性能其实非常高，大致相当于没有线程安全版本的 C++ std::shared_ptr 指针，事实上，C++ 这个指针的主要开销也在于原子性这个并发原语上，毕竟线程安全在哪个语言中开销都不小。

内存损耗

```
struct Wrapper<T> {
    // Rc
    strong_count: usize,
    weak_count: usize,

    // Refcell
    borrow_count: isize,

    // 包裹的数据
    item: T,
}
```

CPU 损耗

从 CPU 来看，损耗如下：

对 Rc 解引用是免费的（编译期），但是 * 带来的间接取值并不免费

克隆 Rc 需要将当前的引用计数跟 0 和 usize::Max 进行一次比较，然后将计数值加 1

释放（drop）Rc 需要将计数值减 1，然后跟 0 进行一次比较

对 RefCell 进行不可变借用，需要将 isize 类型的借用计数加 1，然后跟 0 进行比较

对 RefCell 的不可变借用进行释放，需要将 isize 减 1

对 RefCell 的可变借用大致流程跟上面差不多，但是需要先跟 0 比较，然后再减 1

对 RefCell 的可变借用进行释放，需要将 isize 加 1

通过 Cell::from_mut 解决借用冲突

在 Rust 1.37 版本中新增了两个非常实用的方法：

Cell::from_mut，该方法将 &mut T 转为 &Cell

Cell::as_slice_of_cells，该方法将 &Cell<[T]> 转为 &[Cell]