从C++到Rust，内存所有权管理Ownership，讲道理

原创 Ajonbin AJonbin的杂货铺 2024年02月24日 21:07 美国

之前讲了Rust的各种指针，今天来谈谈由此引出的Rust最核心的概念，Ownership。

这里的ownership是指对内存的所有权。

这也是Rust和其他编程语言的主要区别。

这个最主要的区别说白了就是谁来释放内存。

编程语言一般有两种设计原则：

- 控制第一，也就是由代码来直接控制内存的分配和释放，这样的缺点是代码会有bug，从而导致对内存的管理失效。
- 安全第一，也就是由语言本身提供内存回收释放的机制，这样的缺点是代码不知道对象内存会在什么时候被释放，因此就没有办法来做合理的内存规划和管理。同时，内存回收机制也是额外的开销。

对于一个C++来说，对内存的控制性是第一重要的原则。

代码决定什么时候分配和释放内存。这样就对内存的使用有了控制权和知情权。特别是在嵌入式开发中，可以实现对内存的使用有一个合理的规划。

但是这也造成了代码安全性的问题。特别对于复杂的程序，一旦内存释放的代码出现问题，那么重复释放内存，无效指针（dangling pointers）这些内存错误就会出现，并导致程序崩溃或是程序漏洞。

这也就是C++常被诟病的安全问题。

可能就是因为大部分C++不那么安全，大多数现代编程语言，像Java，python这些都遵循了安全第一的原则，将内存的释放控制在语言本身，而不是交给应用代码。也就是采用了”垃圾回收机制“来释放已经不使用的内存。

这样，应用代码就不需要关心什么时候释放内存，只管使用，降低了程序在内存管理方面设计复杂性，同时也就提高了安全性。

但是这样的代价是应用代码对内存释放一无所知，这对有高性能需求的系统程序来说是一个致命缺点。

同时垃圾回收任务本身，垃圾回收的策略也会给debug程序带来额外的复杂性。

这似乎是一个两难问题，内存可控还是安全。

Rust设计之初就考虑了这个问题，并且提出了一个不太一样的解题思路：

通过严格限制指针的使用方法，来同时达到内存可控和安全。

而这种严格的限制是在编译期由编译期来检查的。

这也就是大多数C++程序员在刚接触Rust会觉得，Rust程序要编译成功是很困难的。

好在Rust编译器的错误提示还是比较易读懂的。这点比C++好太多了。

那么如何来限制指针的使用呢，核心设计就是每块内存都有一个所有者owner，当owner生命周期结束，那么这个owner所拥有的内存也将被释放。

这里的内存块可能是：

- 一个值Value，对应的owner是一个变量；
- Struct结构体中一个field，对应的owner是这个结构体
- 数据容器中的某个元素，对应的owner就是这个容器
- ...

Ownership概念其实挺简单的，就是你的东西你负责。

你可以把你拥有的东西送给别人，这就是后面要讲的**Move**，所有权转移。

你可以把你拥有的东西借给别人，别人可以用，但负责人还是你。借东西**Borrow**就是通过reference引用来实现的。

当然凡事都有例外。

原生的数据类型，比如整数，浮点数和字符这些简单的数据类型就没有ownership。别人要用既不是用Move"送"，也不是用Borrow"借"，而是直接拷贝复制一个。

另外，在标准库还提供了另外两种指针类型，Rc（Reference-Count）和Arc（Atomic-Reference-Count），来实现多个owner。每增加个owner都会增加一个计数，owner消亡后，计数减1。当计数为0时，内存值被释放。这就和C++的智能指针就很类似了。

今天讲的有点枯燥了，早点结束，这次"讲道理"，下次"摆事实"。