所有权规则

jask

09/27/2024

所有权

Rust 中的值有两大类：一类是固定内存长度的值，如 i32, u32, 由固定尺寸的类型组成的结构体；另一类是不固定内存长度的值，如字符串 String。

一种类型如果具有固定尺寸，那么它就能够在编译期做更多的分析。实际上固定尺寸类型也可以用来管理非固定尺寸类型。具体来说,Rust 中的非固定尺寸类型就是靠指针或引用来指向，而指针或引用本身就是一种固定尺寸的类型。

栈与堆

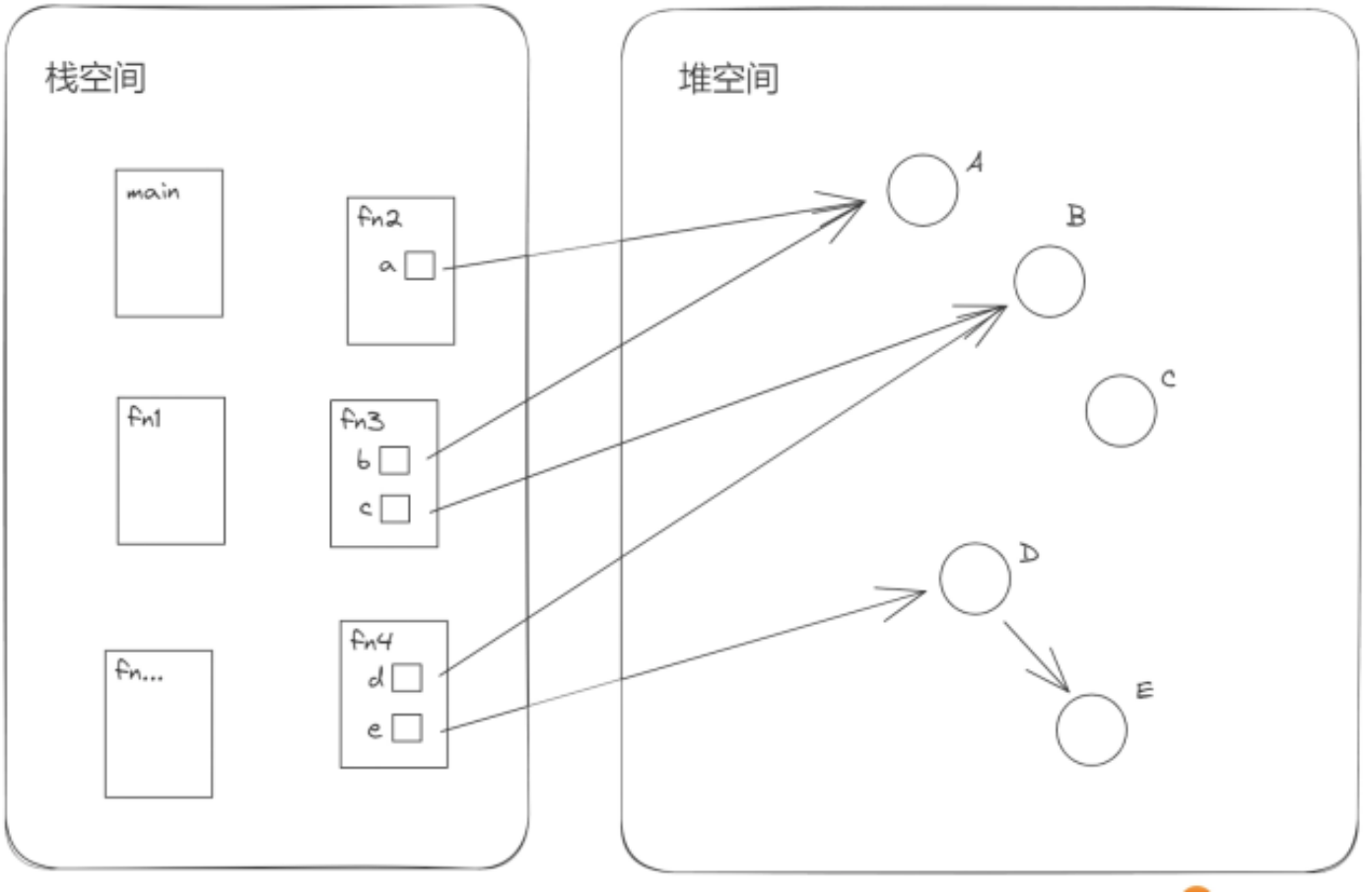
现代计算机会把内存划分为很多个区。比如二进制代码的存放区、静态数据的存放区、栈、堆等。

栈上的操作比堆高效，因为栈上内存的分配和回收只需移动栈顶指针就行了。这就决定了分配和回收时都必须精确计算这个指针的增减量。

因此栈上一般放固定尺寸的值。另一方面，栈的容量也是非常有限的，因此也不适合放尺寸太大的值，比如一个有 1000 万个元素的数组。

栈空间与堆空间

在一般的程序语言设计中，栈空间都会与函数关联起来。每一个函数的调用，都会对应一个帧，也叫做 frame 栈帧，就像图片栈空间里的方块 main、fn1、fn2 等。一个函数被调用，就会分配一个新的帧，函数调用结束后，这个帧就会被自动释放掉。因此栈帧是一个运行时的事物。函数中的参数、局部变量之类的资源，都会放在这个帧里面，比如图里 fn2 中的局部变量 a，这个帧释放时，这些局部变量就会被一起回收掉。



函数的调用会形成层级关系，因此栈空间中的帧可能会同时存在很多个，并且在它们之间也对应地形成层级关系。

可能的函数调用关系为,main 函数中调用了函数 fn1,fn1 中调用了函数 fn2,fn2 中调用了函数 fn3,fn3 中调用了函数 fn4,fn4 调用了更深层次的其他函数。这样的话，在程序执行的某个时刻,main 函数、fn1、fn2、fn3、fn4 等对应的帧副本就同时存在于栈中了。

如果一个资源没有被任何一个栈帧中的变量引用或间接引用，如图中的 C，那么它实际是一个被泄漏的资源，也就是发生了内存泄漏。

计算机程序内存管理的复杂性，主要就在于堆内存的管理比较复杂——既要高效，又要安全。

变量与可变性

Rust 允许 shadowing。意思很好理解，就是定义了一个新的变量名，只不过这个变量名和老的相同。原来那个变量就被遮盖起来了，访问不到了。这种方式最大的用处是程序员不用再去费力地想另一个名字了！变量的 Shadow 甚至支持新的变量的类型和原来的不一样。

所有权

Rust 默认是移动语义。

对于 String s1，s2，Rust 虽然也是把字符串的引用由 s1 拷贝到了 s2，但是只保留了最新的 s2 到字符串的指向，同时却把 s1 到字符串的指向给“抹去”了。

所有权的基础是三条定义。

1. Rust 中，每一个值都有一个所有者。
2. 任何一个时刻，一个值只有一个所有者。
3. 当所有者所在作用域（scope）结束的时候，其管理的值会被一起释放掉。

这种堆内存资源随着关联的栈上局部变量一起被回收的内存管理特性，叫作 RAII(ResourceAcquisition Is Initialization)。它实际不是 Rust 的原创，而是 C++ 创造的。

Rust 从某个角度看似乎不如其他语言强大：

几乎所有其他实用的语言都允许开发者构建任意形式的对象图谱，对象之间可以互相任意引用。而 Rust 牺牲了在这方面的灵活性从而获得了更强的分析能力和实体关系的可追踪性。

而且 Rust 对于上述规则还是有一些拓展：

1. 可以把值从一个所有者转移到另一个所有者，从而方便构建、重塑和销毁关系树。
2. 标准库提供了基于引用计数的指针类型 Rc 和 Arc，使用他们可以在满足限制条件的前提下将值指定给多个所有者。
3. 对一个值，可以借用其引用。引用的是生命期有限的非所有指针。

借用与引用

其实在 Rust 中，借用和引用是一体两面。你把东西借给别人用，也就是别人持有了对你这个东西的引用。这里你理解就好，后面我们会混用这两个词。

在 Rust 中，变量前用“&”符号来表示引用，比如 &x。其实引用也是一种值，并且是固定尺寸的值，一般来说，与机器 CPU 位数一致，比如 64 位或 32 位。因为是值，所以就可以赋给另一个变量。同时它又是固定的而且是小尺寸的值，那其实赋值的时候，就可以直接复制一份这个引用。

可以将变量按一个新的维度划分为所有权型变量和引用型变量。

实际上默认 &x 指的是不可变引用。而要获取到可变引用，需要使用 &mut 符号，如 &mutx。

为什么会有可变引用的存在呢？这个事情是这样的。到目前为止，如果要对一个变量内容进行修改，我们必须拥有所有权型变量才行。

而很多时候，我们没法拥有那个资源的所有权，比如你引用一个别人的库，它没有把所有权类型暴露出来，但是确实又有更新其内部状态的需求。因此需要一个东西，它既是一种引用，又能够修改指向资源的内容。于是就引入了可变引用。

println! 会默认对所有权变量做不可变借用操作。

可变引用调用的时机（对应代码里的第 7 行）和不可变引用调用的时机（对应代码里的第 6 行），好像有顺序要求。目前我们尚不清楚这种机制是什么。

- 注意：引用的最后一次调用时机很关键。*

一个所有权型变量的作用域是从它定义时开始到花括号结束。而引用型变量的作用域不是这样，引用型变量的作用域是从它定义起到它最后一次使用时结束。

同时，我们发现还存在一条规则：一个所有权型变量的可变引用与不可变引用的作用域不能交叠，也可以说不能同时存在。

同一个所有权型变量的可变借用之间的作用域也不能交叠。

提示在有借用的情况下，不能对所有权变量进行更改值的操作（写操作）。

有可变借用存在的情况下也一样。

提示在有借用的情况下，不能对所有权变量进行更改值的操作（写操作）。

引用（借用）的一些规则：

所有权型变量的作用域是从它定义时开始到所属那层花括号结束。

引用型变量的作用域是从它定义起到它最后一次使用时结束。

引用（不可变引用和可变引用）型变量的作用域不会长于所有权变量的作用域。这是肯定的，不然就会出现悬锤引用，这是典型的内存安全问题。

一个所有权型变量的不可变引用可以同时存在多个，可以复制多份。

一个所有权型变量的可变引用与不可变引用的作用域不能交叠，也可以说不能同时存在。（有没有感觉很像数据库的排他锁和共享锁??? ）

某个时刻对某个所有权型变量只能存在一个可变引用，不能有超过一个可变借用同时存在，也可以说，对同一个所有权型变量的可变借用之间的作用域不能交叠。（真像吧）

在有借用存在的情况下，不能通过原所有权型变量对值进行更新。当借用完成后（借用的作用域结束后），物归原主，又可以使用所有权型变量对值做更新操作了。

可变引用的再赋值，会执行移动操作。赋值后，原来的那个可变引用变量就不能用了。

这有点类似于所有权的转移，因此一个所有权型变量的可变引用也具有所有权特征，它可以被理解为那个所有权变量的独家代理，具有排它性。

多级引用

可变引用和不可变引用是可以同时存在的。

思考题

请思考，为何在不可变引用存在的情况下（只是读操作），原所有权变量也无法写入？

不可变引用的作用域跨越了所有权变量的写入过程，意味着同一个作用域同时存在可变引用和不可变引用，编译器为了防止读取错误，不能通过编译。可以把 `a = 20` 放到引用之前，即可编译通过。

简单的嗦：借用规则不允许，防止数据竞争，编译时检查。