Deeplang 内存管理方案讨论

背景与设计目标

- 嵌入式 /IoT 设备对内存管理的需求
- Why not GC
- 不希望类型系统太过复杂
- “ 内存管理工具箱”

已知的内存管理工具

- 智能指针 aka 引用计数、 Perceus 优化
- C#/Swift 的“值类型”， Rust 的 copy trait
- 唯一所有权指针
- Rust 的 borrow checker

…… 与它们的问题

- 引用计数性能开销较大、无法很好地处理循环引用
- 值类型不能覆盖所有场景
- 唯一所有权指针使用中受到的限制太多
- Rust 的类型系统过于复杂

内存管理方案提案：整体架构

- 分为四部分：值类型（可以拷贝）、引用计数、唯一所有权指针、 borrow
- 值类型：可以拷贝，因此使用最为便利。但只有比较“小”的类型才可以随便拷贝
- 引用计数：使用方便，但有一定性能开销，且无法处理循环引用。适合少量（无环的）大对象。使用 Perceus 优化
- 唯一所有权指针：无性能开销，无拷贝，但使用有限制
- borrow ： 可以使值类型和唯一所有权指针的使用更加便利

值类型

- 可以任意拷贝的类型
- 例子：数字、只包含其他值类型的结构体
- 反例：数组、字符串
- 问题：如何以 pass by reference 的方式使用值类型?
- 方案一： C# 式的 ref： ref 不可 escape 其作用域、 ref 不是类型，只是某个变量上的标记
- 方案二：复用 borrow 机制，但 borrow 内的值可以随时通过拷贝取出

引用计数

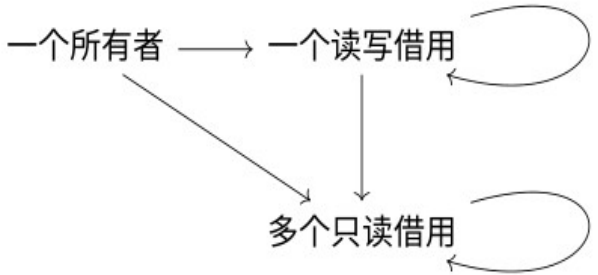
- 可以存储任意类型的值：不管能不能拷贝
- 拷贝一个引用计数指针时，它存储的对象的引用计数 +1
- 施放一个引用计数指针时，它存储的对象的引用计数 -1
- 引用计数归零时，对象被释放
- 有一定性能开销（主要来自从内存中读取对象以修改引用计数： cache 不友好）
- 无法处理循环引用
- 做成语言内建支持而不是库，进一步增强便利性
- 优化：通过 perceus 中的算法自动压缩不必要的引用计数修改

唯一所有权

- 可以存储任意类型的值：不管能不能拷贝
- 唯一所有权指针任何时刻不能有任何别名，只能有一个唯一的“所有者”
- 不能拷贝，只能移动：移动后原先的指针不再能使用
- 无性能开销，零拷贝
- 由于不能创建别名，使用高度受限
- 例如，打印一个唯一指针中的值会消费掉它的所有权，使它不再能被使用！

Borrow，用户看到的

- 改善唯一指针的可用性
- 临时地“借走”一个唯一指针的所有权，稍后再返还
- 必须保证所有借用活得比所有者短
- 通过三种“使用权限”来规范 borrow 的创建与使用
- 具体的权限类型和可以允许的借用方式如下图
- 赋值 / 释放一个指针时，它必须收回自身的所有权限



Borrow, 内部实现的

- 不希望引入类似 Rust 的生命周期的概念：对于用户学习成本较高
- … 但编译器内部必然需要类似的机制
- 如何隐藏内部的实现细节?
- 单个文件内，准确的类型可以通过类型推导获得
- 跨文件时，悲观地使用一个最弱的类型
- 假设每个函数的类型中出现的所有借用都可能出现在返回值中
- 牺牲（跨文件时的）一部分精度，以让“生命周期”完全对用户隐形
- 设计非常友好的、不涉及内部细节的错误信息（ WIP ，欢迎拿例子提问）

值类型与 pass by reference

- 问题：如何让一个函数修改一个值类型的变量，使调用者能看到这次修改?
- 不做支持的话：每次传参，值类型会被拷贝，函数中修改的是一份拷贝，调用者无法看到
- 值类型 pass by reference 支持是必要的。
- 可能的场景：调用者传入 reference ，被调用的函数修改
- 可能的场景：方法返回 this 中某个值类型 field 的 reference ，让调用者修改
- 可能的场景：是否允许将 reference 当作普通的值，存在数据结构中?

值类型与 pass by reference： ref 方案

- 允许用 ref 关键字修饰函数 / 方法的参数 / 返回值
- 修饰参数时：该参数按引用传入，可以修改
- 读取 ref 参数的值，或将它存储在数据结构中时：拷贝
- 修饰返回值时：方法可以返回 this 的一部分的引用
- 使用 ref 返回值时：需要用 ref 修饰的局部变量接受，该局部变量的行为同 ref 修饰的参数
- ref 不能随便被存储在数据结构中
- （可选）需要在结构体中放 ref 时：该结构体也被声明为 ref ，它的值表现类似 ref （不能被存在非 ref 的数据中）

值类型与 pass by reference： ref 方案

- 优点：实现和机制都比较简单
- 缺点：表达能力比较有限

值类型与 pass by reference： borrow 方案

- 复用唯一指针的 borrow 机制：对值类型也允许 borrow
- 值类型的 borrow 与唯一指针的 borrow 机制完全相同
- borrow 是 first class value，因此只需要修改类型即可实现传入 borrow 参数 / 返回 borrow/ 将 borrow 存储于数据结构中
- 但值的 borrow 比唯一指针的 borrow 更易用：永远可以通过拷贝，将数据从 borrow 中取出

值类型与 pass by reference： borrow 方案

- 优点：直接复用唯一指针的 borrow 机制，无需引入新语言构造。拷贝还是引用通过类型标记，不容易误用
- 缺点：机制比 ref 更复杂