第二十一讲：异步编程 (Asynchronous Programming)

第 4 节：Self-Referential Structs & Pin

向勇、陈渝

清华大学计算机系

xyong,yuchen@tsinghua.edu.cn

2020 年 5 月 5 日

Self-Referential Structs

```
async fn pin_example() -> i32 {
    let array = [1, 2, 3];
    let element = &array[2];
    async_write_file("foo.txt", element.to_string()).await;
    *element
}
```

Self-Referential Structs

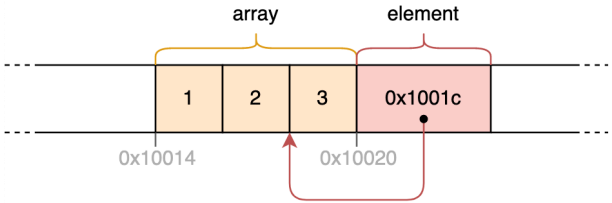
```
async fn pin_example() -> i32 {
    let array = [1, 2, 3];
    let element = &array[2];
    async_write_file("foo.txt", element.to_string()).await;
    *element
}
```

The struct for the "waiting on write" state

```
struct WaitingOnWriteState {
    array: [1, 2, 3],
    element: 0x1001c, // address of the last array element
}
```

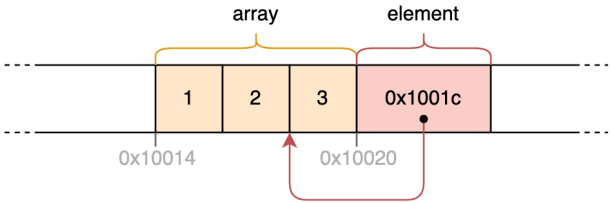
The Problem with Self-Referential Structs

Memory layout of self-referential struct

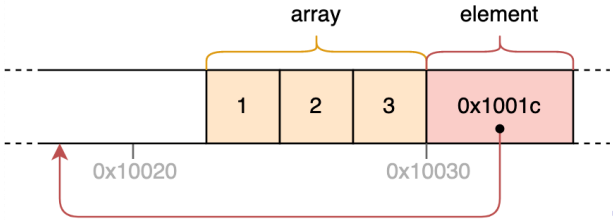


The Problem with Self-Referential Structs

Memory layout of self-referential struct



After moving this struct to a different memory address



Possible approaches to solve the dangling pointer problem

- Update the pointer on move: Require extensive changes to Rust that would result in potentially huge performance losses.

Possible approaches to solve the dangling pointer problem

- Update the pointer on move: Require extensive changes to Rust that would result in potentially huge performance losses.
- Store an offset instead of self-references:: Require the compiler to detect all self-references or need a runtime system again to analyze references and correctly create the state structs.

Possible approaches to solve the dangling pointer problem

- Update the pointer on move: Require extensive changes to Rust that would result in potentially huge performance losses.
- Store an offset instead of self-references:: Require the compiler to detect all self-references or need a runtime system again to analyze references and correctly create the state structs.
- Forbid moving the struct: This approach can be implemented at the type system level without additional runtime costs

Possible approaches to solve the dangling pointer problem

- Update the pointer on move: Require extensive changes to Rust that would result in potentially huge performance losses.
- Store an offset instead of self-references:: Require the compiler to detect all self-references or need a runtime system again to analyze references and correctly create the state structs.
- Forbid moving the struct: This approach can be implemented at the type system level without additional runtime costs
  - It puts the burden of dealing with move operations on possibly self-referential structs on the programmer

Defination of Pin

- Pin wraps a pointer. A reference to an object is a pointer
  - Reference type. In order to break apart a large future into its smaller components, and put an entire resulting future into some immovable location, we need a reference type for methods like ‘poll‘

Defination of Pin

- Pin wraps a pointer. A reference to an object is a pointer
  - Reference type. In order to break apart a large future into its smaller components, and put an entire resulting future into some immovable location, we need a reference type for methods like ‘poll‘
- Pin gives some guarantees about the pointee (the data it points to)
  - Never to move before being dropped. To store references into itself, we decree that by the time you initially ‘poll‘, and promise to never move an immobile future again

Defination of Pin

- Pin wraps a pointer. A reference to an object is a pointer
  - Reference type. In order to break apart a large future into its smaller components, and put an entire resulting future into some immovable location, we need a reference type for methods like ‘poll‘
- Pin gives some guarantees about the pointee (the data it points to)
  - Never to move before being dropped. To store references into itself, we decree that by the time you initially ‘poll‘, and promise to never move an immobile future again

```
trait Future {
    type Item;
    type Error;

    fn poll(self: Pin<Self>, cx: &mut task::Context) -> Poll<Self::Item,
}
```

Pinning to the heap

Pinning to the heap is safe so the user doesn't need to implement any unsafe code.

- Once the data is allocated on the heap it will have a stable address

Pinning to the heap

Pinning to the heap is safe so the user doesn't need to implement any unsafe code.

- Once the data is allocated on the heap it will have a stable address
- Examp: Pinning to the heap