探讨Rust智能指针

Box、Vec<T> | String、Cell | RefCell、Rc | Arc、RwLock | Mutex

苏林

本次公开课将围绕着以下几点来和大家探讨

1、什么是智能指针, “智能"究竟是什么意思?
2、智能指针是如何工作的?
3、Box、Cell、RefCell、Rc、Arc、RwLock 和 Mutex. 这些智能指针作用是什么?

探讨Rust智能指针 | Box、Vec<T> | String、Cell | RefCell、Rc | Arc、RwLock | Mutex

正如《Rust程序设计语言》书中所解释的, 指针是包含"指向"一些其他数据的地址的变量。Rust中常用的指针是引用(&), 智能指针是"具有附加元数据和功能"的指针, 例如, 它们可以计算值被借用的次数, 提供管理读写锁的方法等。

从技术上讲, String和Vec也是智能指针, 但我不会在这里介绍它们, 因为它们很常见并且通常被认为是类型而不是指针.

另请注意, 在此(Box、Cell、RefCell、Rc、Arc、RwLock 和 Mutex), 只有Arc、RwLock和Mutex是线程安全的.

什么是智能指针

```rust
fn main() {
    let x : Box<i32> = Box::new( x: 42);
    let y : i32 = *x;
    assert_eq!(y, 42);
}
```

Rust中, Trait决定了类型的行为, 那么智能指针它的行为主要涉及两个trait.

1、Deref trait -> 拥用指针语义.

2、Drop trait -> 拥用内存自动管理的机制

Copy语义、Move语义

智能指针工作机制

```rust
pub trait Deref {
    /// The resulting type after dereferencing.
    #[stable(feature = "rust1", since = "1.0.0")]
    #[rustc_diagnostic_item = "deref_target"]
    #[lang = "deref_target"]
    type Target: ?Sized;

    /// Dereferences the value.
    #[must_use]
    #[stable(feature = "rust1", since = "1.0.0")]
    #[rustc_diagnostic_item = "deref_method"]
    fn deref(&self) -> &Self::Target;
}
```

*x -> *(x.deref())

智能指针工作机制

```
1486    #[stable(feature = "rust1", since = "1.0.0")]
1487    impl<T: ?Sized, A: Allocator> Deref for Box<T, A> {
1488        type Target = T;
1489
1490        fn deref(&self) -> &T {
1491            &**self
1492        }
1493    }
```

智能指针工作机制

```rust
// 定义一个元组结构体
struct MySmartPointer<T>(T);

impl<T> MySmartPointer<T> {
    fn new(x: T) -> MySmartPointer<T> {
        MySmartPointer(x)
    }
}

impl<T> Deref for MySmartPointer<T> {
    type Target = T;

    fn deref(&self) -> &Self::Target {
        &self.0
    }
}

fn main() {
    let x: i32 = 5;
    let y: MySmartPointer<i32> = MySmartPointer::new(x);
    assert_eq!(5, x);
    assert_eq!(5, *y);
}
```

*y -> *(y.deref()) -> x

智能指针 到底 智能在何处?

大家可以思考一下.....

智能指针 到底 智能在何处?

大家可以思考一下.....

智能指针 到底 智能在何处?

1、可以自动解引用, 提升开发体验.

2、可以自动化管理内存, 安全无忧.

智能指针 到底 智能在何处?

1、可以自动解引用, 提升开发体验.

2、可以自动化管理内存, 安全无忧.

```rust
use std::ops::Deref;

// 定义一个元组结构体
struct MySmartPointer<T>(T);

impl<T> MySmartPointer<T> {
    fn new(x: T) -> MySmartPointer<T> {
        MySmartPointer(x)
    }
}

impl<T> Deref for MySmartPointer<T> {
    type Target = T;

    fn deref(&self) -> &Self::Target {
        &self.0
    }
}
```

```rust
struct User {
    name: &'static str
}

impl User {
    fn name(&self) {
        println!("{:?}", self.name);
    }
}

fn main() {
    let u = User {name: "Databend"};
    let y : MySmartPointer<User> = MySmartPointer::new( x: u);

    y.name();
}
```

智能指针 到底 智能在何处?

1、可以自动解引用, 提升开发体验.

2、可以自动化管理内存, 安全无忧.

```rust
fn takes_str(s: &str) {
    println!("{:?}", s);
}

fn main() {
    let s : String = String::from( s: "Databend");
    takes_str(&s);
}
```

```rust
#[stable(feature = "rust1", since = "1.0.0")]
impl ops::Deref for String {
    type Target = str;

    #[inline]
    fn deref(&self) -> &str { unsafe { str::from_utf8_unchecked( v: &self.vec) } }
}
```

关于自动解引用, 需要注意的地方

1、使用*x这样手工解引用的方式, 等价于*(x.deref())

2、使用点调用或在函数参数位置上对x自动解引用则是等价于x.deref()

标准库中的智能指针

1、Box<T>

2、Vec<T> 和 String

3、Rc<T> 和 Arc<T>

4、Cell<T> 和 RefCell<T>

5、RwLock 和 Mutex

Box<T>

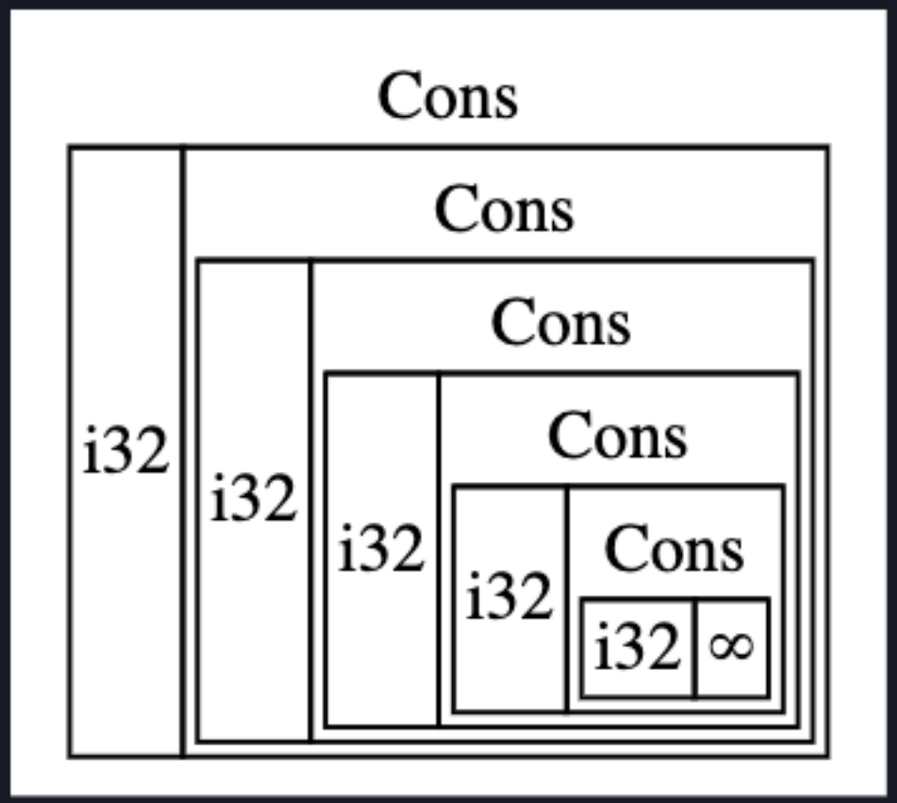
使用Box<T>的情况：

递归类型和trait对象。Rust需要在编译时知道一个类型占用多少空间，Box<T>的大小是已知的。

```
enum List {
    Cons(i32, List),
    Nil,
}
```

```
use crate::List::{Cons, Nil};

fn main() {
    let list = Cons(1, Cons(2, Cons(3, Nil)));
}
```




```
error[E0072]: recursive type `List` has infinite size
 --> src/main.rs:1:1
  |
1 | enum List {
  | ^^^^^^^^^ recursive type has infinite size
2 |     Cons(i32, List),
  |               ----- recursive without indirection
  |
  = help: insert indirection (e.g., a `Box`, `Rc`, or `&`) at some point to
  make `List` representable
```

Box<T>

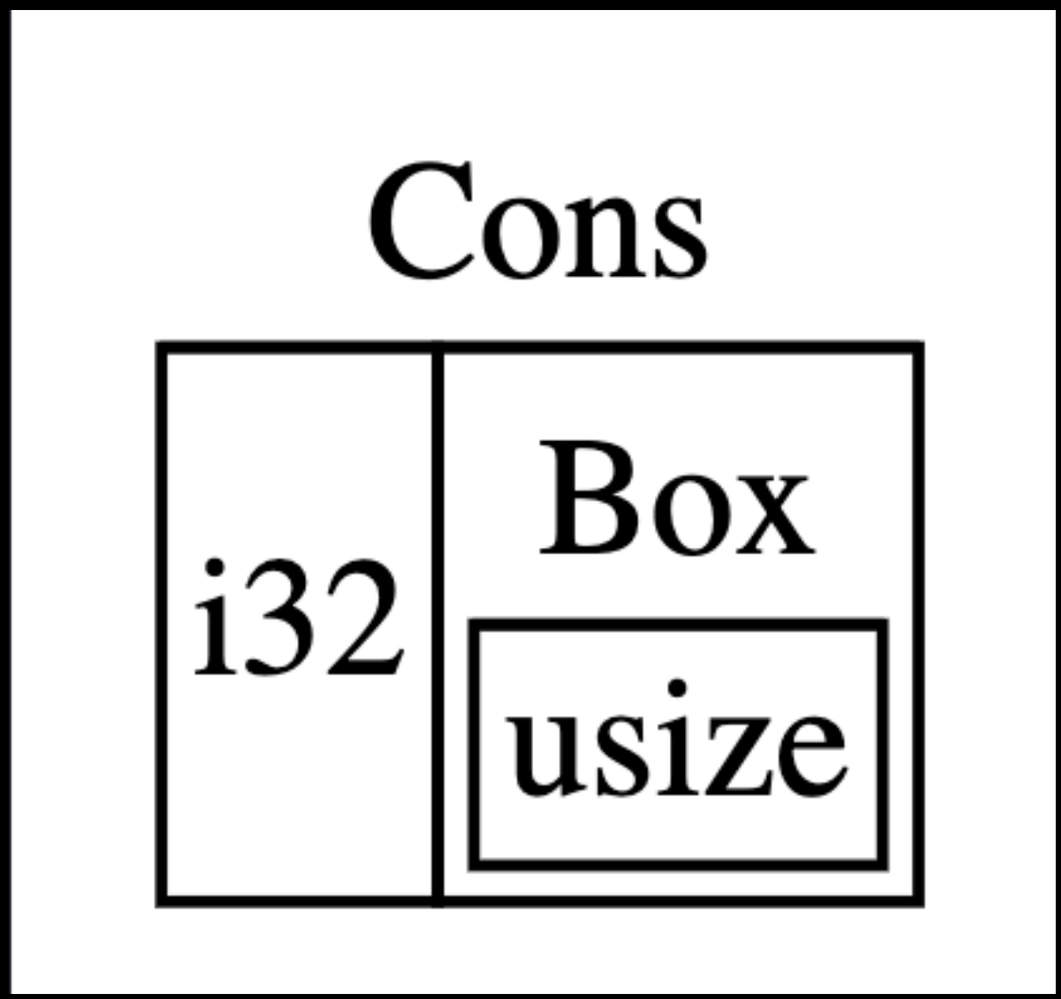
使用Box<T>的情况：

递归类型和trait对象。Rust需要在编译时知道一个类型占用多少空间，Box<T>的大小是已知的。

```
enum List {
    Cons(i32, Box<List>),
    Nil,
}

use crate::List::{Cons, Nil};

fn main() {
    let list = Cons(1,
        Box::new(Cons(2,
            Box::new(Cons(3,
                Box::new(Nil))))));
}
```



Cell<T> 与 RefCall<T>

Cell<T> 其实和 Box<T> 很像，但后者同时不允许存在多个对其的可变引用，如果我们真的很想做这样的操作，在需要的时候随时改变其内部的数据，而不去考虑 Rust 中的不可变引用约束，就可以使用 Cell<T>。Cell<T> 允许多个共享引用对其内部值进行更改，实现了「内部可变性」。

```rust
fn main() {
    let x = Cell::new(1);
    let y = &x;
    let z = &x;
    x.set(2);
    y.set(3);
    z.set(4);
    println!("{}", x.get());  // Output: 4
}
```

Cell<T> 与 RefCall<T>

因为 Cell<T> 对 T 的限制：只能作用于实现了 Copy 的类型，所以应用场景依旧有限（安全的代价）。但是我如果就是想让任何一个 T 都可以塞进去该咋整呢？RefCell<T> 去掉了对 T 的限制，但是别忘了要牢记初心，不忘继续践行 Rust 的内存安全的使命，既然不能在读写数据时简单的 Copy 出来进去了，该咋保证内存安全呢？相对于标准情况的静态借用，RefCell<T> 实现了运行时借用，这个借用是临时的，而且 Rust 的 Runtime 也会随时紧盯 RefCell<T> 的借用行为：同时只能有一个可变借用存在，否则直接 Painc。也就是说 RefCell<T> 不会像常规时一样在编译阶段检查引用借用的安全性，而是在程序运行时动态的检查，从而提供在不安全的行为下出现一定的安全场景的可行性。

```rust
use std::cell::RefCell;
use std::thread;

fn main() {
    thread::spawn(move || {
        let c = RefCell::new(5);
        let m = c.borrow();

        let b = c.borrow_mut();
    }).join();
    // Error: thread '<unnamed>' panicked at 'already borrowed: BorrowMutErr
}
```

QA环节

加群一起交流Rust & Datafuse

Datafuse Rust语言中文社区 知数堂 zhishuteng.com