16不安全的Rust

2020年3月9日 12:26

unsafe存在的意义，因为计算机硬件固有的不安全性，我们需要底层编程，如果没有unsafe
有些操作我们根本完成不了。

unsafe的操作一共有四类:
(1): 解引用裸指针
(2): 调用不安全的函数或方法
(3): 访问或修改可变静态变量
(4):访问不安全的trait
(5): 访问union的字段

一:裸指针:
注意: unsafe并不会关闭借用检查器或禁用任何其他Rust安全检查:
如果在不安全代码中使用引用,它扔会被检查.
unsafe关键字只是提供了四个不会被编译器检查内存安全的功能:
这也是裸指针与引用和智能指针的区别,记住裸指针的特点:
(1): 允许忽略借用规则,可以同时拥有不可变和可变的指针,或多个只想相同位置的可变指针
(2): 不保证指向有效的内存
(3): 允许为空
(4): 不能实现任何的自动清理功能.

裸指针分为可变或不可变的,写为: *const T和*mut T.不可变意味着指针解引用之后不能直接赋值.

```
let mut num = 5;
let r1 = &num as *const i32;
let r2 = &mut num as *mut i32;
```

这里没有unsafe,可以安全的在代码中创建裸指针,只是不能在unsafe之外解引用裸指针.

```
let address= 0x02310231231_usize;
let r = address as *const i32; // 解引用的时候会出现错误
```

完整的例子:

```
let mut num = 5;
let r1 = &num as *const i32;
let r2 = &mut num as *mut i32;

unsafe {
    println!("r1 is {}", *r1);
    println!("r2 is {}", *r2); // 解引用必须在unsafe中.
```

```
}
```

二:
调用不安全的函数:

```
unsafe fn user() {
    // 注意:因为用unsafe表明了这个函数是unsafe的,就是说
    // user的函数体是unsafe的,那么就没有必须在函数体中用
    // unsafe表明了,整个函数体都是unsafe的.
}
```

三:创建一个不安全代码的安全抽象

```
fn user() {  // 整个函数是安全的
    ...     // 这里是安全的代码

    unsafe {
        ... //这里是不安全的代码
    }
}
```

四:使用extern 调用外部函数或者代码时候,要加上unsafe

```
extern "C" { // extern表明的外部代码总是不安全的,这里不需要加unsafe
                // 在调用的时候添加.
    fn abs(input: i32) -> i32;
}

fn main() {
    unsafe {
        abs(10);
    }
}
```

注意: 也可以使用extern创建一个允许其他语言调用Rust函数的接口.这里不能使用
extern块,就是说不能使用

```
extern "C" {

}
```

这样的,而是要单独的写:

```
#[no_mangle]
pub extern "C" fn call_from_c() {

}
```

注意:使用所有extern的情况下,使用无需unsafe.extern本身就表示了不安全.

五:修改或访问可变的静态变量.(访问不可变的静态变量是安全的)

六:实现不安全的trait (impl trait的时候)
当至少有一个方法中包含编译器不能验证的不变量是trait时不安全的.
可以在trait前加unsafe声明为不安全的,同时trait的实现必须标记为unsafe.

```
unsafe trait Foo {

}

unsafe impl Foo fori32 {
}
```

怎么理解呢?
因为trait的方法是只有声明没有实现.

```
unsafe trait Foo {
    fn hello(&self); // 如果Self类型是一个不能验证的不变量时trait是不安全的,
                     // 如果让安全代码去检查 trait 实现的正确性不太现实,
                        //那么把 trait 标为 unsafe 就是合理的。
}

unsafe impl Foo for *const i32 {

}
```

Sync 和 Send 标记 trait，编译器会自动为完全由 Send 和 Sync 类型组成的类型自动实现他们。如果实现了一个包含一些不是 Send 或 Sync 的类型，比如裸指针，并希望将此类型标记为 Send 或 Sync，则必须使用 unsafe。Rust 不能验证我们的类型保证可以安全的跨线程发送或在多线程键访问，所以需要我们自己进行检查并通过 unsafe 表明。

注意:这个功能自己实现的地方很少.需要注意的是:

```
trait Foo {
    unsafe fn hello(&self);
}
```

实现的时候:

```
impl Foo for My {
    unsafe fn hello(&self){...}
}
```

调用的时候:

```
unsafe {
    my.hello();
}
```