第八课: Unsafe Rust

unsafe trait、unsafe函数、解引用裸指针、FFI

苏林

安全的 Rust 并不能适应所有的使用场景

首先: 为了内存安全，Rust 所做的这些规则往往是普适性的

其次，无论 Rust 将其内部的世界构建得多么纯粹和完美，它总归是要跟不纯粹也不完美的外界打交道，无论是硬件还是软件

可以使用unsafe的场景

先看可以使用、也推荐使用unsafe的场景

1、实现unsafe trait -> 主要是Send/Sync这两个trait
2、调用已有的unsafe接口
3、对裸指针做解引用
4、使用FFI

不推荐的使用unsafe的场景

1、访问或者修改可变静态变量
2、在宏里使用unsafe
3、使用unsafe提升性能

实现unsafe trait

名气最大的 unsafe 代码应该就是 Send / Sync 这两个 trait -> Rc/RefCell/裸指针

```
pub unsafe auto trait Send { }
```
pub unsafe auto trait Sync { }

```rust
pub struct Bytes {
    ptr: *const u8,
    len: usize,
    // inlined "trait object"
    data: AtomicPtr<()>,
    vtable: &'static Vtable,
}

// Vtable must enforce this behavior
unsafe impl Send for Bytes {}
unsafe impl Sync for Bytes {}
```

实现unsafe trait

任何 trait，只要声明成 unsafe，它就是一个 unsafe trait

调用已有的unsafe函数

看一段代码

```rust
#[stable(feature = "rust1", since = "1.0.0")]
#[rustc_const_unstable(feature = "const_str_from_utf8", issue = "91006")]
pub const fn from_utf8(v: &[u8]) -> Result<&str, Utf8Error> {
    // This should use `?` again, once it's `const`
    match run_utf8_validation(v) {
        Ok(_) => {
            // SAFETY: validation succeeded.
            Ok(unsafe { from_utf8_unchecked(v) })
        }
        Err(err) => Err(err),
    }
}
```

```rust
#[stable(feature = "str_mut_extras", since = "1.20.0")]
#[rustc_const_unstable(feature = "const_str_from_utf8", issue = "91006")]
pub const fn from_utf8_mut(v: &mut [u8]) -> Result<&mut str, Utf8Error> {
    // This should use `?` again, once it's `const`
    match run_utf8_validation(v) {
        Ok(_) => {
            // SAFETY: validation succeeded.
            Ok(unsafe { from_utf8_unchecked_mut(v) })
        }
        Err(err) => Err(err),
    }
}
```

对裸指针解引用

看一段代码

```rust
fn main() {
    let mut age = 18;

    // 不可变引用
    let r1 = &age as *const i32;
    // 可变引用
    let r2 = &mut age as *mut i32;

    // 使用裸指针, 可以绕过 immutable / mutable borrow rule

    // 然而, 对指针解引用需要使用 unsafe
    unsafe {
        println!("r1: {}, r2: {}", *r1, *r2);
    }
}
```

对裸指针解引用

看一段代码

```rust
fn main() {
    // 裸指针指向一个有问题的地址
    let r1 = 0xdeadbeef as *mut u32;

    println!("so far so good!");

    unsafe {
        // 程序崩溃
        *r1 += 1;
        println!("r1: {}", *r1);
    }
}
```

使用FFI

看一段代码

```rust
use std::mem::transmute;

fn main() {
    let data = unsafe {
        let p = libc::malloc(8);
        let arr: &mut [u8; 8] = transmute(p);
        arr
    };

    data.copy_from_slice(&[1, 2, 3, 4, 5, 6, 7, 8]);

    println!("data: {:?}", data);

    unsafe { libc::free(transmute(data)) };
}
```

不推荐的使用unsafe的场景

1、访问或者修改可变静态变量
2、在宏里使用unsafe
3、使用unsafe提升性能

访问或者修改可变静态变量

全局的 static 变量，以及使用 lazy_static 来声明复杂的 static 变量。然而之前遇到的 static 变量都是不可变的

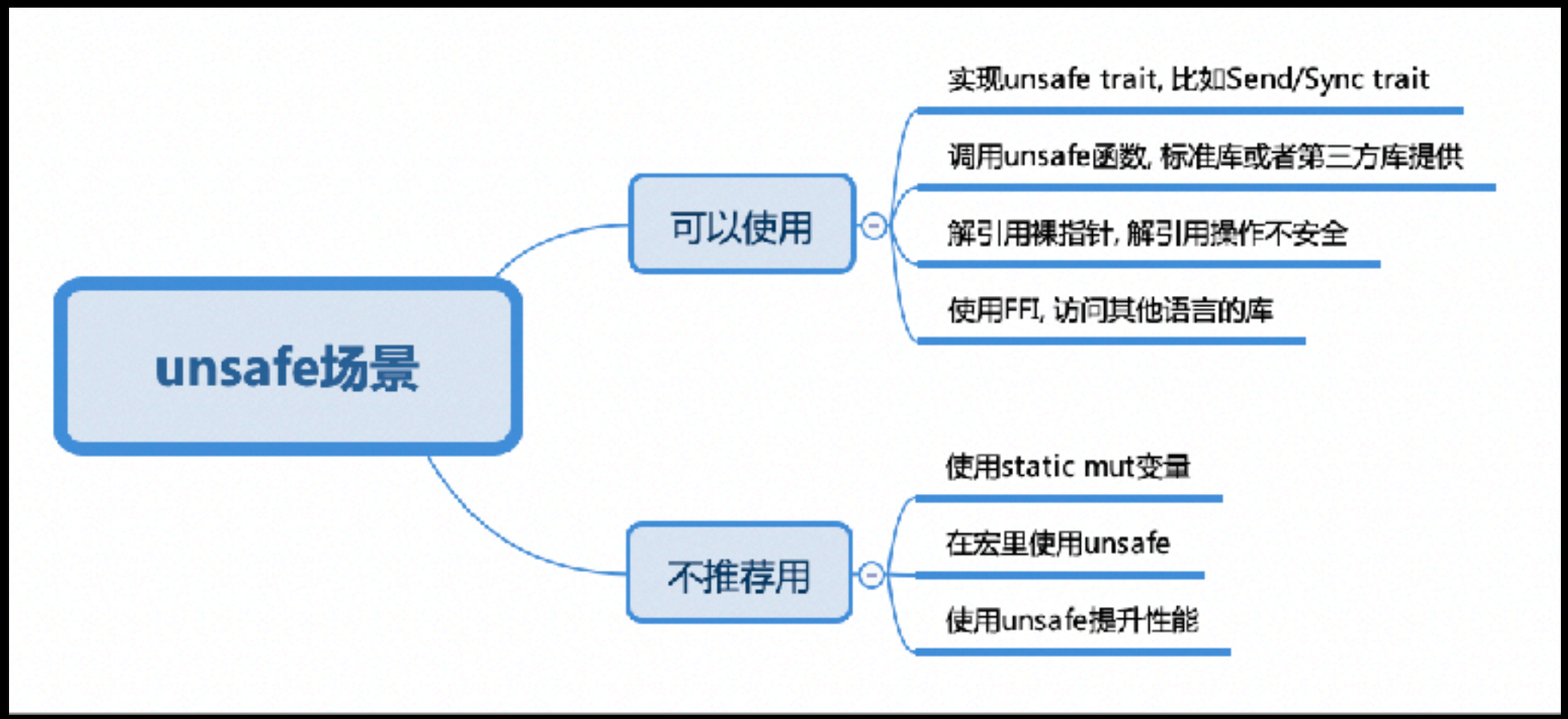
Rust 还支持可变的 static 变量，可以使用 static mut 来声明

看段代码

使用 static mut. 任何需要static mut 的地方. AtomicXXX/Mutex/RwLock来取代

小结

unsafe 代码，是 Rust 这样的系统级语言必须包含的部分，当 Rust 跟硬件、操作系统，以及其他语言打交道，unsafe 是必不可少的



QA环节

加群一起交流Rust & Databend

Databend

Rust语言中文社区

知数堂