Rust 中的安全与不安全代码边界实践

叶家炜

Apr 06, 2025

Rust 语言以「内存安全」与「零成本抽象」著称，但这一承诺的实现依赖于开发者对安全（safe）与不安全（unsafe）代码边界的清晰认知。当我们需要操作硬件、进行极端性能优化或与 C 语言交互时，unsafe 代码就成为必须的工具。本文将通过具体代码示例，探讨如何在实践中构建可靠的安全抽象层。

1 安全与不安全代码的基础

Rust 编译器通过所有权系统和借用检查等机制，在编译期阻止了 90% 以上的内存错误。但当我们执行以下操作时，必须使用 unsafe 块：

```
// 解引用裸指针
let raw_ptr = &42 as *const i32;
let value = unsafe { *raw_ptr };

// 调用 unsafe 函数
unsafe {
    libc::printf("Hello from C\0".as_ptr() as *const i8);
}
```

关键要理解：unsafe 代码本身并不危险，真正的风险在于开发者是否正确维护了 Rust 的安全契约。标准库中 Vec<T> 的实现就是典型案例 —— 其内部大量使用 unsafe 代码，但通过严谨的抽象设计，对外暴露完全安全的 API。

2 划分边界的实践策略

2.1 封装裸指针操作

考虑实现一个安全的自定义迭代器：

```
struct SafeIter<T> {
    ptr: *const T,
    end: *const T,
    _marker: std::marker::PhantomData<T>,
}
```

```

impl<T> SafeIter<T> {
    pub fn new(slice: &[T]) -> Self {
        let ptr = slice.as_ptr();
        let end = unsafe { ptr.add(slice.len()) };
        Self {
            ptr,
            end,
            _marker: std::marker::PhantomData,
        }
    }
}

impl<T> Iterator for SafeIter<T> {
    type Item = &'static T;

    fn next(&mut self) -> Option<Self::Item> {
        if self.ptr == self.end {
            None
        } else {
            let current = unsafe { &*self.ptr };
            self.ptr = unsafe { self.ptr.add(1) };
            Some(current)
        }
    }
}
```

这段代码通过三个关键设计保障安全：

- PhantomData 标记类型所有权，防止悬垂指针
- 所有指针运算都封装在 unsafe 块内
- 生命周期被严格限定在迭代器自身

2.2 类型系统的力量

当需要实现跨线程共享时，可以借助 Send 和 Sync trait：

```
struct ThreadSafeBuffer<T> {
    data: *mut T,
    len: usize,
}
```

```

// 手动标记该类型可跨线程传递
unsafe impl<T> Send for ThreadSafeBuffer<T> where T: Send {}
unsafe impl<T> Sync for ThreadSafeBuffer<T> where T: Sync {}

impl<T> ThreadSafeBuffer<T> {
    pub fn write(&self, index: usize, value: T) {
        unsafe {
            std::ptr::write(self.data.add(index), value);
        }
    }
}
```

通过 unsafe impl 显式声明类型的安全属性，同时利用泛型约束 where T: Send 确保内部数据的线程安全性。这种模式在实现无锁数据结构时尤为重要。

3 安全验证与工具链支持

3.1 Miri 的实战应用

考虑以下看似合理的代码：

```
fn dangling_pointer() -> &'static i32 {
    let x = 42;
    unsafe { &*(&x as *const i32) }
}
```

使用 Miri 执行 cargo +nightly miri run 会立即检测到悬垂指针问题：

```
error: Undefined Behavior: using stack value after return
  --> src/main.rs:3:14
   |
3 | unsafe { &*(&x as *const i32) }
   | ^^^^^^^^^^^^^^^^^^^^^^
```

3.2 模糊测试实践

对于涉及内存操作的代码，可以使用 cargo fuzz 进行压力测试：

```
// fuzz_targets/mem_ops.rs
fuzz_target!(|data: &[u8]| {
    let mut buffer = Vec::with_capacity(data.len());
    unsafe {
```

```
        std::ptr::copy_nonoverlapping(
            data.as_ptr(),
            buffer.as_mut_ptr(),
            data.len()
        );
        buffer.set_len(data.len());
    }
    assert_eq!(&buffer, data);
});
```

该测试会生成随机输入，验证我们的内存拷贝操作是否正确处理各种边界情况。

4 常见陷阱与防御策略

4.1 未初始化内存陷阱

错误示例：

```
let mut data: i32;
unsafe {
    std::ptr::write(&mut data as *mut i32, 42);
}
println!("{}", data); // UB!
```

正确做法应使用 MaybeUninit：

```
let data = unsafe {
    let mut uninit = std::mem::MaybeUninit::<i32>::uninit();
    std::ptr::write(uninit.as_mut_ptr(), 42);
    uninit.assume_init()
};
```

MaybeUninit 通过类型系统强制要求开发者显式处理初始化状态，避免读取未初始化内存的风险。

4.2 生命周期断裂案例

考虑以下跨作用域指针传递：

```
fn create_dangling() -> &'static [i32] {
    let arr = vec![1, 2, 3];
    let slice = &arr[..];
    unsafe { std::mem::transmute(slice) }
}
```

该代码通过 transmute 强行延长生命周期，但实际内存会在函数返回后立即释放。正确做法应使用 Box::leak 显式声明内存泄漏：

```
fn valid_static() -> &'static [i32] {
    let arr = Box::new([1, 2, 3]);
    Box::leak(arr)
}
```

5 进阶场景：FFI 安全封装

与 C 语言交互时，可采用以下模式：

```
mod ffi {
    #[repr(C)]
    pub struct CContext {
        handle: *mut std::ffi::c_void,
    }

    extern "C" {
        pub fn create_context() -> *mut CContext;
        pub fn free_context(ctx: *mut CContext);
    }
}

pub struct SafeContext {
    inner: *mut ffi::CContext,
}

impl SafeContext {
    pub fn new() -> Option<Self> {
        let ptr = unsafe { ffi::create_context() };
        if ptr.is_null() {
            None
        } else {
            Some(Self { inner: ptr })
        }
    }
}

impl Drop for SafeContext {
    fn drop(&mut self) {
```

```
        unsafe {
            ffi::free_context(self.inner);
        }
    }
}
```

该封装实现了：

- 自动资源管理（通过 Drop trait）
- 空指针检查
- 类型系统保证的访问安全

6 结论

在 Rust 中使用 unsafe 代码如同操作核反应堆 —— 需要多层防护措施。通过本文展示的封装模式、验证工具和实践原则，开发者可以在保持系统级性能的同时，将风险限制在可控范围内。记住：每个 unsafe 块都应该有对应的安全证明，就像数学定理需要推导过程一样。这正是 Rust 哲学的精髓：通过严格的约束获得深层的自由。