# 内部可变类型代码分析

内部可变性类型常常让人疑惑，感觉到是"xxxx,费二遍事"。因为在程序中，频繁操作的变量大多属于内部可变性类型的范畴, 这就更加让人愤怒。内部可变性类型通常被RUST初学者认为是可变引用独占性的补丁特性。
内部可变性类型正确的认识应该是：这是RUST的有意为之的特性，很可能是先有内部可变性，才使得可变引用的独占性成立。内部可变性设计是为了减少变量写操作的无序性，使程序员更好的去设计有可能冲突的变量写操作，减少无序写操作导致的bug。也使得程序员明确的发现变量可能冲突的写操作，产生警惕心，从而借助编译器找到更多的写冲突问题。 所以，内部可变性类型显著的提升了代码的质量。

内部可变性的基础是Borrow trait：

## Borrow trait 代码分析

代码路径如下：
%USER%.rustup\toolchains\nightly-x86_64-pc-windows-msvc\lib\rustlib\src\rust\library\core\src\borrow.rs

Borrow trait实现了对变量引用的导出，一般是在封装类型上实现。通过borrow调用可以将这些类型的内部变量引用提供给外部。通常的情况下，这些类型也都实现了Deref，AsRef等trait可以获取内部变量引用，所以这些trait之间有些重复。但Borrow trait 最主要的场景是作为内部可变性类型 `RefCell<T>` 的内部变量引用导出，这是Deref, AsRef等trait无能为力的区域。本节之后将分析 `RefCell<T>` 类型时再给出进一步阐述。

Borrow trait代码定义如下：

```
pub trait Borrow<Borrowed: ?Sized> {
    fn borrow(&self) -> &Borrowed;
}

pub trait BorrowMut<Borrowed: ?Sized>: Borrow<Borrowed> {
    fn borrow_mut(&mut self) -> &mut Borrowed;
}
//每一个类型都实现了针对自身的Borrow trait
impl<T: ?Sized> Borrow<T> for T {
    fn borrow(&self) -> &T {
        self
    }
}

//每一个类型都实现了针对自身的BorrowMut trait
impl<T: ?Sized> BorrowMut<T> for T {
```

```
    fn borrow_mut(&mut self) -> &mut T {
        self
    }
}

//每一个类型的引用都实现了对自身的Borrow trait
impl<T: ?Sized> Borrow<T> for &T {
    fn borrow(&self) -> &T {
        &**self
    }
}
//每一个类型的可变引用都实现了针对自身的Borrow trait
impl<T: ?Sized> Borrow<T> for &mut T {
    fn borrow(&self) -> &T {
        &**self
    }
}

//每一个类型的可变引用都实现了针对自身的BorrowMut
impl<T: ?Sized> BorrowMut<T> for &mut T {
    fn borrow_mut(&mut self) -> &mut T {
        &mut **self
    }
}
```

## Cell模块类型代码分析

代码路径： %USER%.rustup\toolchains\nightly-x86_64-pc-windows-msvc\lib\rustlib\src\rust\library\core\src\cell.rs

Cell类型提供了内部可变性的功能。对应于以下场景：
一个变量存在多个引用，希望通过这些引用都可以修改此变量。

RUST的可变引用与不可变引用不能同时共存，这导致了无法通过普通的引用语法完成上述场景。
RUST提供的解决方案是`Cell<T>`封装类型。思路很简单，提供一个封装类型结构，对此类型实现一个set方法来修改内部封装的变量。set方法主要是通过unsafe RUST来实现内部变量修改。
Cell模块类型的层次如下：

1. `UnsafeCell<T>`负责将内部封装的变量导出多个`* mut T`。其他模块只要基于`*mut T`生成可变引用，即可修改内部变量。这显然是违反RUST的可变引用的语法的，也是不安全的。
2. `Cell<T>`基于`UnsafeCell<T>`导出的`*mut T`实现了set方法改变内部的T类型变量。只要拥有`Cell<T>`的引用，即可以用set方法修改`Cell<T>`内部的变量。显然，直接用`Cell<T>`引用对变量进行修改会引发修改冲突，安全隐患很大。

3. `RefCell<T>`基于`Cell<T>`及`UnsafeCell<T>`，并实现Borrow Trait 及 BorrowMut Trait，可以实现在生命周期不重合的情况下的多个可变引用，且可变引用与不可变引用不可以同时存在。显然，`RefCell<T>`是与RUST内存安全理念相契合的内部可变性实现方案。

## `UnsafeCell<T>`代码分析

UnsafeCell是RUST的内部可变结构的最底层基础设施，Cell结构和RefCell结构都是用UnsafeCell来实现内部可变性的。

```
pub struct UnsafeCell<T: ?Sized> {
    value: T,
}
impl<T> UnsafeCell<T> {
    //创建封装结构
    pub const fn new(value: T) -> UnsafeCell<T> {
        UnsafeCell { value }
    }

    //解封装
    pub const fn into_inner(self) -> T {
        self.value
    }
}
//对任意T的类型，可以为T.into() 创建UnsafeCell类型变量
impl<T> const From<T> for UnsafeCell<T> {
    fn from(t: T) -> UnsafeCell<T> {
        UnsafeCell::new(t)
    }
}

impl<T: ?Sized> UnsafeCell<T> {
    pub const fn get(&self) -> *mut T {
        // 将裸指针导出，这是为什么起名是UnsafeCell的原因
        // 此裸指针的安全性由调用代码保证，调用代码可以使用此裸指针改变内部封装的变量
        self as *const UnsafeCell<T> as *const T as *mut T
    }

    //给出一个正常的可变引用，此引用存在期间，get及raw_get调用会编译器告警
    pub const fn get_mut(&mut self) -> &mut T {
        &mut self.value
    }

    //参数与get有区别，是关联函数
    pub const fn raw_get(this: *const Self) -> *mut T {
        this as *const T as *mut T
    }
}
```

```
//显然，UnsafeCell不支持Sync，即使内部变量支持Sync，这与RUST的默认规则不一致，需要显式声明
impl <T:?Sized> !Sync for UnsafeCell<T> {}
```

可以看到，UnsafeCell的get函数返回了裸指针，UnsafeCell逃脱RUST对引用安全检查的方法实际上就是个通常的unsafe 的裸指针操作，没有任何神秘性可言。

## `Cell<T>` 代码分析

Cell 内部包装UnsafeCell， 利用UnsafeCell的方法获得裸指针后，用unsafe代码对内部变量进行赋值，从而绕开了RUST语言编译器对引用的约束。Cell的赋值实际上和直接使用裸指针赋值是等同的，但因为提供了方法，没有直接暴露裸指针，所以保证了安全性。

```
#[repr(transparent)]
pub struct Cell<T: ?Sized> {
    value: UnsafeCell<T>,
}
```

Cell创建方法：

```
impl<T> const From<T> for Cell<T> {
    fn from(t: T) -> Cell<T> {
        Cell::new(t)
    }
}

impl<T> Cell<T> {
    pub const fn new(value: T) -> Cell<T> {
        Cell { value: UnsafeCell::new(value) }
    }
```

`Cell<T>` 改变内部变量的方法:

```
    pub fn set(&self, val: T) {
        //实际调用mem::replace
        let old = self.replace(val);
        //这里不调用drop，old也应该因为生命周期终结被释放。
        //此处调用drop以确保万无一失
        drop(old);
    }

    pub fn swap(&self, other: &Self) {
        //此处注意，ptr::eq不仅仅比较地址，也比较元数据
        if ptr::eq(self, other) {
```

```
            return;
        }
        //此段不会出现在跨线程的场景下
        unsafe {
            ptr::swap(self.value.get(), other.value.get());
        }
    }

    //此函数也会将原有的值及所有权返回
    pub fn replace(&self, val: T) -> T {
        // 利用unsafe粗暴将指针转变为可变引用，然后赋值，此处必须用
        // replace，原有值的所有权需要有交代。
        mem::replace(unsafe { &mut *self.value.get() }, val)
    }
```

获取内部值的解封装方法:

```
    pub const fn into_inner(self) -> T {
        //解封装
        self.value.into_inner()
    }
}

impl<T: Default> Cell<T> {
    //take后，变量所有权已经转移出来
    pub fn take(&self) -> T {
        self.replace(Default::default())
    }
}

impl<T: Copy> Cell<T> {
    pub fn get(&self) -> T {
        //只适合于Copy Trait类型，否则会导致所有权转移，引发UB
        unsafe { *self.value.get() }
    }
```

对函数式编程支持的方法

```
    //函数式编程，因为T支持Copy，所以没有所有权问题
    pub fn update<F>(&self, f: F) -> T
    where
        F: FnOnce(T) -> T,
    {
        let old = self.get();
        let new = f(old);
        self.set(new);
        new
    }
```

```
}
```

获取内部变量指针的方法：

```
impl<T: ?Sized> Cell<T> {
    //通常应该不使用这个机制，安全隐患非常大
    pub const fn as_ptr(&self) -> *mut T {
        self.value.get()
    }
    //获取内部的可变引用，调用这个函数会占用&mut self，
    //需要等到返回值生命周期结束才能释放。
    pub fn get_mut(&mut self) -> &mut T {
        self.value.get_mut()
    }

    pub fn from_mut(t: &mut T) -> &Cell<T> {
        // 利用repr[transparent]直接做转换
        unsafe { &*(t as *mut T as *const Cell<T>) }
    }
}
```

切片类型相关方法

```
//Unsized Trait实现
impl<T: CoerceUnsized<U>, U> CoerceUnsized<Cell<U>> for Cell<T> {}

impl<T> Cell<[T]> {
    pub fn as_slice_of_cells(&self) -> &[Cell<T>] {
        // 粗暴的直接转换
        unsafe { &*(self as *const Cell<[T]> as *const [Cell<T>]) }
    }
}

impl<T, const N: usize> Cell<[T; N]> {
    pub fn as_array_of_cells(&self) -> &[Cell<T>; N] {
        // 粗暴的直接转换
        unsafe { &*(self as *const Cell<[T; N]> as *const [Cell<T>; N]) }
    }
}
```

`Cell<T>` 仅对支持Send的T支持Send trait

```
//按规则，此处代码可以不写，但估计可以减轻编译器负担或编译器有额外要求。
unsafe impl<T: ?Sized> Send for Cell<T> where T: Send {}
```

```
//按规则也可以不写，但估计可以减轻编译器负担或编译器有额外要求
unsafe impl<T:?Sized> !Sync for Cell<T> {}
```

## `RefCell<T>` 代码分析

`RefCell<T>` 设计的思路：

1. 基本类型RefCell，负责存储内部可变的变量及计数器
2. Ref类型, 作为执行borrow()后生成的返回结果，通过解引用可以直接获得内部变量的引用，drop调用时会减少计数器不可变引用计数
3. RefMut类型, 作为执行borrow_mut()后生成的返回结构，通过解引用可以直接获得内部变量的可变引用，对内部变量进行修改。drop调用时会减少计数器可变引用计数
4. 不采用在Ref及RefMut中包含&RefCell的方式来实现对RefCell内部计数器的操作，这样在逻辑上有些混乱
5. 单独设计BorrowRef作为Ref计数器的借用类型，目的是利用此类型的drop函数完成对RefCell中计数器的不可变借用计数操作
6. 单独设计BorrowRefMut作为RefMut计数器的借用类型，目的是利用此类型的drop函数完成对RefCell中计数器的可变借用计数器操作

以下为RefCell类型相关的结构， 删除了一些和debug相关的内容，使代码简化及理解简单

```
pub struct RefCell<T: ?Sized> {
    //用以标识对外是否有可变引用，是否有不可变引用，有多少个不可变引用
    //是引用计数的实现体
    borrow: Cell<BorrowFlag>,
    //包装内部的变量
    value: UnsafeCell<T>,
}
```

引用计数类型BorrowFlag的定义：

```
// 正整数表示RefCell执行borrow()调用
// 生成的不可变引用"Ref"的数目
//
// 负整数表示RefCell执行borrow_mut()调用
// 生成的可变引用"RefMut"的数目
//
// 多个RefMut存在的条件是在多个RefMut指向
// 同一个"RefCell"的不同部分的情况，如多个
// RefMut指向一个slice的不重合的部分。
type BorrowFlag = isize;
// 0表示没有执行过borrow()或borrow_mut()调用
const UNUSED: BorrowFlag = 0;
```

```rust
//有borrow_mut()被执行且生命周期没有终结
fn is_writing(x: BorrowFlag) -> bool {
    x < UNUSED
}

//有borrow()被执行且生命周期没有终结
fn is_reading(x: BorrowFlag) -> bool {
    x > UNUSED
}
```

`RefCell<T>`创建方法：

```rust
impl<T> RefCell<T> {
    pub const fn new(value: T) -> RefCell<T> {
        RefCell {
            value: UnsafeCell::new(value),
            //初始化一定是UNUSED
            borrow: Cell::new(UNUSED),
        }
    }
```

解封装方法：

```rust

    //实际会消费RefCell，并将内部变量返回，因为Ref及RefMut有PhantomData
    //所以，存在borrow及borrow_mut时，调用此方法会编译出错，没有安全问题。
    pub const fn into_inner(self) -> T {
        self.value.into_inner()
    }
```

### borrow()相关的结构及代码

`RefCell<T>` borrow()的代码：

```rust
impl<T: ?Sized> RefCell<T> {
    //Borrow Trait实现，返回Ref<`a, T>类型变量，见下面分析
    pub fn borrow(&self) -> Ref<'_, T> {
        //真正是try_borrow()
        self.try_borrow().expect("already mutably borrowed")
    }
    //不可变引用 borrow真正实现
    pub fn try_borrow(&self) -> Result<Ref<'_, T>, BorrowError> {
        match BorrowRef::new(&self.borrow) {
            Some(b) => {
                // 保证了self.borrow一定是is_reading()为真，直接从裸指针
                //转换，对RUST来讲，转换后的引用与原变量没有内存安全的联系。
                // 从这个函数看，RefCell<T>应该尽量使用RefCell的方法操作，除非绝对把
```

```
握
                // 不要直接将内部变量的正常引用导出，否则安全隐患巨大。
                // 这里返回的Ref变量的生命周期不能长于self，*self.value.get()本身没
有生命周期
                // &*self.value.get()加入了生命周期，而这个生命周期受到函数生命周期
语法的限制
                // 从而返回值的生命周期小于了self的生命周期。这是RUST生命周期的一个重
要规则。不安全
                // 由此进入安全
                Ok(Ref { value: unsafe { &*self.value.get() }, borrow: b })
            }
            None => Err(BorrowError {

            }),
        }
    }
    ...
}
```

`Ref<'b, T>`相关类型结构：

```
//RefCell<T> borrow()调用获取的类型
pub struct Ref<'b, T: ?Sized + 'b> {
    //对RefCell<T>中value的引用
    value: &'b T,
    //对RefCell<T>中borrow引用的封装
    borrow: BorrowRef<'b>,
}

//Deref将获得内部value
impl<T: ?Sized> Deref for Ref<'_, T> {
    type Target = T;

    fn deref(&self) -> &T {
        self.value
    }
}
```

针对不可变借用的计数逻辑实现类型

```
//对RefCell<T>中成员变量borrow的引用封装类型
struct BorrowRef<'b> {
    borrow: &'b Cell<BorrowFlag>,
}

impl<'b> BorrowRef<'b> {
    //每次new，代表对RefCell<T>产生了borrow()调用，需增加不可变引用计数
```

```
    fn new(borrow: &'b Cell<BorrowFlag>) -> Option<BorrowRef<'b>> {
        // 引用计数加1,
        let b = borrow.get().wrapping_add(1);
        if !is_reading(b) {
            // 1.如果有borrow_mut()调用且生命周期没有终结
            // 2.如果到达isize::MAX
            None
        } else {
            // 增加一个不可变借用计数:
            borrow.set(b);
            Some(BorrowRef { borrow })
        }
    }
}

// Drop，代表对RefCell<T>的borrow()调用的返回变量生命周期结束，需减少不可变引用计数
impl Drop for BorrowRef<'_> {
    fn drop(&mut self) {
        let borrow = self.borrow.get();
        //一定应该是正整数
        debug_assert!(is_reading(borrow));
        //不可变引用计数减一
        self.borrow.set(borrow - 1);
    }
}
impl Clone for BorrowRef<'_> {
    //每次clone实际上增加了一次RefCell<T>的不可变引用,
    fn clone(&self) -> Self {
        //不可变引用计数加1
        let borrow = self.borrow.get();
        debug_assert!(is_reading(borrow));
        assert!(borrow != isize::MAX);
        self.borrow.set(borrow + 1);
        BorrowRef { borrow: self.borrow }
    }
}
```

对Ref结构的方法：

```
impl<'b, T: ?Sized> Ref<'b, T> {
    /// 与再执行RefCell<T>::borrow等价。但用clone可以在不必有RefCell<T>的情况下增加引用
    /// 不选择实现Clone Trait，是因为要用RefCell<T>.borrow().clone()来复制
    /// RefCell<T>
    pub fn clone(orig: &Ref<'b, T>) -> Ref<'b, T> {
        Ref { value: orig.value, borrow: orig.borrow.clone() }
    }

    //通常的情况下，F的返回引用与Ref中的引用是强相关的，即获得返回引用等同于获得Ref中value的引用
```

```
    pub fn map<U: ?Sized, F>(orig: Ref<'b, T>, f: F) -> Ref<'b, U>
    where
        F: FnOnce(&T) -> &U,
    {
        Ref { value: f(orig.value), borrow: orig.borrow }
    }

    //同上，例如value是一个切片引用，filter后获得切片的一部分
    pub fn filter_map<U: ?Sized, F>(orig: Ref<'b, T>, f: F) -> Result<Ref<'b,
U>, Self>
    where
        F: FnOnce(&T) -> Option<&U>,
    {
        match f(orig.value) {
            Some(value) => Ok(Ref { value, borrow: orig.borrow }),
            None => Err(orig),
        }
    }

    /// leak调用后，此Ref不再调用drop，从而导致RefCell中的计数器无法恢复原状，也会导
致可变引用无法再被创建
    pub fn leak(orig: Ref<'b, T>) -> &'b T {
        mem::forget(orig.borrow);
        orig.value
    }
}

impl<'b, T: ?Sized + Unsize<U>, U: ?Sized> CoerceUnsized<Ref<'b, U>> for
Ref<'b, T> {}
```

### borrow_mut() 相关结构及代码

`RefCell<T>`的borrow_mut()代码：

```
impl<T: ?Sized> RefCell<T> {
    //BorrowMut Trait实现,返回RefMut<`a, T>类型变量
    pub fn borrow_mut(&self) -> RefMut<'_, T> {
        self.try_borrow_mut().expect("already borrowed")
    }

    pub fn try_borrow_mut(&self) -> Result<RefMut<'_, T>, BorrowMutError> {
        match BorrowRefMut::new(&self.borrow) {
            Some(b) => {
                // 一定不存在非可变引用，也仅有本次的可变引用，这个可变引用直接从
                // 裸指针转换，对RUST编译器，转换后的可变引用与原变量没有内存安全的联
系。
                Ok(RefMut { value: unsafe { &mut *self.value.get() }, borrow: b
})
            }
```

```
            None => Err(BorrowMutError {

            }),
        }
    }
    ...
}
```

从RefCell borrow_mut返回的结构体`RefMut<'b, T>`结构代码:

```
pub struct RefMut<'b, T: ?Sized + 'b> {
    //可变引用
    value: &'b mut T,
    //计数器
    borrow: BorrowRefMut<'b>,
}
```

`BorrowRefMut<T>`结构及逻辑:

```
//作用与BorrowRef相同
struct BorrowRefMut<'b> {
    borrow: &'b Cell<BorrowFlag>,
}

//RefMut生命周期终止时调用
impl Drop for BorrowRefMut<'_> {
    fn drop(&mut self) {
        //可变引用计数减一(数学运算为加)
        let borrow = self.borrow.get();
        debug_assert!(is_writing(borrow));
        self.borrow.set(borrow + 1);
    }
}

impl<'b> BorrowRefMut<'b> {
    fn new(borrow: &'b Cell<BorrowFlag>) -> Option<BorrowRefMut<'b>> {
        //初始的borrow_mut，引用计数必须是0，不存在其他可变引用
        match borrow.get() {
            UNUSED => {
                borrow.set(UNUSED - 1);
                Some(BorrowRefMut { borrow })
            }
            _ => None,
        }
    }

    // 不通过RefCell获取新的RefMut的方法，对于新的RefMut，
```

```
    // 必须是一个整体的可变引用分为几个组成部分的可变引用，
    // 如结构体成员，或数组成员。且可变引用之间互相不重合，
    // 不允许两个可变引用能修改同一块内存
    fn clone(&self) -> BorrowRefMut<'b> {
        //不可变引用计数增加(算数减)
        let borrow = self.borrow.get();
        debug_assert!(is_writing(borrow));
        // Prevent the borrow counter from underflowing.
        assert!(borrow != isize::MIN);
        self.borrow.set(borrow - 1);
        BorrowRefMut { borrow: self.borrow }
    }
}
```

RefMut的代码：

```
//Deref后返回内部变量的引用
impl<T: ?Sized> Deref for RefMut<'_, T> {
    type Target = T;

    fn deref(&self) -> &T {
        self.value
    }
}
//DerefMut返回内部变量可变引用
impl<T: ?Sized> DerefMut for RefMut<'_, T> {
    fn deref_mut(&mut self) -> &mut T {
        self.value
    }
}
```

### `RefCell<T>` 其他方法

改变内部值的方法：

```


impl<T: ?Sized> RefCell<T> {
    //将原有内部变量替换为新值，既然是RefCell，通常应使用borrow_mut
    //获得可变引用，再对值做修改，下面函数实际也是用borrow_mut完成，
    //但更多应该是用在泛型中
    pub fn replace(&self, t: T) -> T {
        mem::replace(&mut *self.borrow_mut(), t)
    }

    //同上，只是用函数获取新值
    pub fn replace_with<F: FnOnce(&mut T) -> T>(&self, f: F) -> T {
        let mut_borrow = &mut *self.borrow_mut();
        let replacement = f(mut_borrow);
```

```
        mem::replace(mut_borrow, replacement)
    }

    //两个引用交换值，也交换了值的所有权
    pub fn swap(&self, other: &Self) {
        mem::swap(&mut *self.borrow_mut(), &mut *other.borrow_mut())
    }
}
```

直接获取内部变量指针：

```
    //此函数如果没有绝对的安全把握，不要用
    pub fn as_ptr(&self) -> *mut T {
        self.value.get()
    }

    //此函数如果没有绝对的安全把握，不要用
    pub fn get_mut(&mut self) -> &mut T {
        self.value.get_mut()
    }
```

其他方法：

```
    //在leak操作后，做leak的逆操作，实际上对计数器进行了恢复，
    pub fn undo_leak(&mut self) -> &mut T {
        *self.borrow.get_mut() = UNUSED;
        self.get_mut()
    }

    //规避计数器计数的方法，与borrow操作近似
    pub unsafe fn try_borrow_unguarded(&self) -> Result<&T, BorrowError> {
        //如果没有borrow_mut(),则返回引用
        if !is_writing(self.borrow.get()) {
            Ok(unsafe { &*self.value.get() })
        } else {
            Err(BorrowError {
            })
        }
    }
}
```

内部值获取方法：

```
impl<T: Default> RefCell<T> {
    //对RefCell<T>应该不使用这个函数，尤其是在有borrow()/borrow_mut()
    //且生命周期没有终结时
    pub fn take(&self) -> T {
```

```
        self.replace(Default::default())
    }
}
```

系统编译器内嵌trait实现：

```
//支持线程间转移
unsafe impl<T: ?Sized> Send for RefCell<T> where T: Send {}
//不支持线程间共享
impl<T: ?Sized> !Sync for RefCell<T> {}

impl<T: Clone> Clone for RefCell<T> {
    //clone实际上仅仅是增加计数
    fn clone(&self) -> RefCell<T> {
        //self.borrow().clone 实质是((*self.borrow()).clone)
        //连续解引用后做clone的调用
        //Ref<T>不支持Clone，所以解引用的到&T
        RefCell::new(self.borrow().clone())
    }

    fn clone_from(&mut self, other: &Self) {
        //self.get_mut().clone_from 实质是
        // (*self.get_mut()).clone_from()
        // &mut T不支持Clone，所以解引用到T
        self.get_mut().clone_from(&other.borrow())
    }
}

impl<T: Default> Default for RefCell<T> {
    fn default() -> RefCell<T> {
        RefCell::new(Default::default())
    }
}

impl<T: ?Sized + PartialEq> PartialEq for RefCell<T> {
    fn eq(&self, other: &RefCell<T>) -> bool {
        *self.borrow() == *other.borrow()
    }
}

impl<T> const From<T> for RefCell<T> {
    fn from(t: T) -> RefCell<T> {
        RefCell::new(t)
    }
}

impl<T: CoerceUnsized<U>, U> CoerceUnsized<RefCell<U>> for RefCell<T> {}
```

RefCell的代码实现，是理解RUST解决问题的思维的好例子。 编程中，RefCell的计数器是针对RUST语法的一个精巧的设计，利用drop的自动调用，编程只需要关注new，这就节省了程序员极大的精力，也规避了错误的发生。borrow_mut()机制则解决了多个可修改借用。 利用RUST的非安全个性和自动drop的机制，可以自行设计出RefCell这样的标准库解决方案，而不是借助于编译器。这是RUST的一个突出特点，也是其能与C一样成为系统级语言的原因。

## Pin及UnPin

Pin主要解决需要程序员在编程时要时刻注意处理可能的变量地址改变的情况。利用Pin，程序员只需要在初始的时候注意到这个场景并定义好。后继就可以不必再关心。 Pin是一个对指针&mut T的包装结构，包装后因为&mut T的独占性。封装结构外，不可能再存在变量的引用及不可变引用。所有的引用都只能使用Pin来完成，导致RUST的需要引用的一些内存操作无法进行，如实质上是指针交换的调用mem::swap，从而保证了指针指向的变量在代码中会被固定在某个内存位置。当然，编译器也不会再做优化。

实现Unpin Trait的类型不受Pin的约束，RUST中实现Copy trait的类型基本上都实现了Unpin Trait。 结构定义

```
#[repr(transparent)]
#[derive(Copy, Clone)]
pub struct Pin<P> {
    pointer: P,
}
```

Pin变量创建：

```
impl<P: Deref<Target: Unpin>> Pin<P> {
    // 支持Unpin类型可以用new创建Pin<T>
    pub const fn new(pointer: P) -> Pin<P> {
        unsafe { Pin::new_unchecked(pointer) }
    }
    ...
}

impl<P: Deref> Pin<P> {
    //实现Deref的类型，用下面的行为创建Pin<T>, 调用者应该保证P可以被Pin,
    pub const unsafe fn new_unchecked(pointer: P) -> Pin<P> {
        Pin { pointer }
    }
    ...
}
```

Pin自身的new方法仅针对Pin实际上不起作用的Unpin类型。对于其他不支持Unpin的类型，通常使用智能指针提供的Pin创建方法，如Boxed::pin。 new_unchecked则提供给其他智能指针的安全的创建方法内部使用。

```
impl <P: Deref<Target: Unpin>> Pin<P> {
    ...
    /// 解封装，取消内存pin操作
    pub const fn into_inner(pin: Pin<P>) -> P {
        pin.pointer
    }
}

impl <P:Deref> Pin<P> {
    ...

    //对应于new_unchecked
    pub const unsafe fn into_inner_unchecked(pin: Pin<P>) -> P {
        pin.pointer
    }
}
```

指针转换

```
impl<P: Deref> Pin<P> {
    ...
    /// 需要返回一个Pin的引用，以为P自身就是指针，返回P是
    /// 不合理及不安全的，所以此函数被用来返回Pin住的解引
    /// 用的指针类型
    pub fn as_ref(&self) -> Pin<&P::Target> {
        // SAFETY: see documentation on this function
        unsafe { Pin::new_unchecked(&*self.pointer) }
    }
}
impl <P:DerefMut> Pin<P> {
    ...
    /// 需要返回一个Pin的可变引用，以为P自身就是指针，
    /// 所以此函数被用来返回Pin住的解引用的指针类型
    pub fn as_mut(&mut self) -> Pin<&mut P::Target> {
        unsafe { Pin::new_unchecked(&mut *self.pointer) }
    }
}
impl <'a, T:?Sized> Pin<&'a T> {
    //&T 不会导致在安全RUST领域的类如mem::replace之类的地址改变操作
    pub const fn get_ref(self) -> &'a T {
        self.pointer
    }
}
```

```rust
impl<'a, T: ?Sized> Pin<&'a mut T> {
    //不可变引用可以随意返回，不会影响Pin的语义
    pub const fn into_ref(self) -> Pin<&'a T> {
        Pin { pointer: self.pointer }
    }

    //Unpin的可变引用可以返回，Pin对Unpin类型无作用
    pub const fn get_mut(self) -> &'a mut T
    where
        T: Unpin,
    {
        self.pointer
    }

    //后门，要确定安全，会导致Pin失效
    pub const unsafe fn get_unchecked_mut(self) -> &'a mut T {
        self.pointer
    }
    ...
}

impl<T: ?Sized> Pin<&'static T> {
    pub const fn static_ref(r: &'static T) -> Pin<&'static T> {
        unsafe { Pin::new_unchecked(r) }
    }
}

impl<'a, P: DerefMut> Pin<&'a mut Pin<P>> {
    pub fn as_deref_mut(self) -> Pin<&'a mut P::Target> {
        unsafe { self.get_unchecked_mut() }.as_mut()
    }
}

impl<T: ?Sized> Pin<&'static mut T> {
    pub const fn static_mut(r: &'static mut T) -> Pin<&'static mut T> {
        // SAFETY: The 'static borrow guarantees the data will not be
        // moved/invalidated until it gets dropped (which is never).
        unsafe { Pin::new_unchecked(r) }
    }
}

impl<P: Deref> Deref for Pin<P> {
    type Target = P::Target;
    fn deref(&self) -> &P::Target {
        Pin::get_ref(Pin::as_ref(self))
    }
}
//只有Unpin才支持对mut的DerefMut trait，不支持Unpin的，
//不能用DerefMut，以保证Pin
impl<P: DerefMut<Target: Unpin>> DerefMut for Pin<P> {
    fn deref_mut(&mut self) -> &mut P::Target {
```

```
        Pin::get_mut(Pin::as_mut(self))
    }
}
```

内部可变性函数:

```
impl <P:DerefMut> Pin<P> {
    //修改值，实质也提供了内部可变性
    pub fn set(&mut self, value: P::Target)
    where
        P::Target: Sized,
    {
        *(self.pointer) = value;
    }
}

impl<'a, T: ?Sized> Pin<&'a T> {
    //函数式编程，func返回的pointer与self.pointer应该强相关，如结构中
    //某一变量的引用，或slice中某一元素的引用
    pub unsafe fn map_unchecked<U, F>(self, func: F) -> Pin<&'a U>
    where
        U: ?Sized,
        F: FnOnce(&T) -> &U,
    {
        let pointer = &*self.pointer;
        let new_pointer = func(pointer);

        // SAFETY: the safety contract for `new_unchecked` must be
        // upheld by the caller.
        unsafe { Pin::new_unchecked(new_pointer) }
    }

}

impl<'a, T: ?Sized> Pin<&'a mut T> {

    pub unsafe fn map_unchecked_mut<U, F>(self, func: F) -> Pin<&'a mut U>
    where
        U: ?Sized,
        F: FnOnce(&mut T) -> &mut U,
    {
        // 这个可能导致Pin住的内容移动，调用者要保证不出问题
        let pointer = unsafe { Pin::get_unchecked_mut(self) };
        let new_pointer = func(pointer);
        unsafe { Pin::new_unchecked(new_pointer) }
    }
}
```

利用Pin的封装及基于trait约束的方法实现，使得指针pin在内存中的需求得以实现。是RUST利用封装语义完成语言需求的又一经典案例

## Lazy分析

OnceCell是一种内部可变的类型，其用于初始化没有初始值，仅支持赋值一次的类型。Once一般用于支持全局静态变量。

```
pub struct OnceCell<T> {
    // Option<T>支持None作为初始化的值
    inner: UnsafeCell<Option<T>>,
}
```

OnceCell封装UnsafeCell以支持内部可变性。 创建方法:

```
impl<T> const From<T> for OnceCell<T> {
    fn from(value: T) -> Self {
        OnceCell { inner: UnsafeCell::new(Some(value)) }
    }
}
impl<T> OnceCell<T> {
    /// 初始化为空，支持静态全局变量初始化
    pub const fn new() -> OnceCell<T> {
        //注意，此时给UnsafeCell分配T类型的地址空间
        OnceCell { inner: UnsafeCell::new(None) }
    }
```

获取内部引用

```
    pub fn get(&self) -> Option<&T> {
        // 生成一个内部变量的引用，
        unsafe { &*self.inner.get() }.as_ref()
    }

    /// 直接用返回结果取可以&mut T，然后再解封装后用可变引用即
    /// 可改变内部封装变量的值，会突破只赋值一次的既定语义，
    /// 此函数最好不使用
    pub fn get_mut(&mut self) -> Option<&mut T> {
        unsafe { &mut *self.inner.get() }.as_mut()
    }
```

对内部值进行修改方法：

```rust
    /// 通过此函数仅能给OnceCell内部变量做一次赋值
    pub fn set(&self, value: T) -> Result<(), T> {
        // SAFETY: Safe because we cannot have overlapping mutable borrows
        let slot = unsafe { &*self.inner.get() };
        if slot.is_some() {
            return Err(value);
        }

        let slot = unsafe { &mut *self.inner.get() };
        *slot = Some(value);
        Ok(())
    }

    //见下面函数
    pub fn get_or_init<F>(&self, f: F) -> &T
    where
        F: FnOnce() -> T,
    {
        //Ok::<T,!>(f()) 即Result类型初始化，例如Ok::<i32,!>(3)
        match self.get_or_try_init(|| Ok::<T, !>(f())) {
            Ok(val) => val,
        }
    }

    //有值就返回值，没有值用f生成值
    pub fn get_or_try_init<F, E>(&self, f: F) -> Result<&T, E>
    where
        F: FnOnce() -> Result<T, E>,
    {
        if let Some(val) = self.get() {
            return Ok(val);
        }
        /// 下面代码关键是cold，防止优化后的代码出现意外，因为此函数会被多次调用
        /// 这是一个较冷门的知识点
        #[cold]
        fn outlined_call<F, T, E>(f: F) -> Result<T, E>
        where
            F: FnOnce() -> Result<T, E>,
        {
            f()
        }
        let val = outlined_call(f)?;
        assert!(self.set(val).is_ok(), "reentrant init");
        Ok(self.get().unwrap())
    }
```

解封装方法：

```
    //消费了OnceCell,并且返回内部变量
    pub fn into_inner(self) -> Option<T> {
        self.inner.into_inner()
    }

    //替换OnceCell，并将替换的OnceCell消费掉，并且返回内部变量
    pub fn take(&mut self) -> Option<T> {
        mem::take(self).into_inner()
    }
}
```

OnceCell对trait的实现：

```
impl<T> Default for OnceCell<T> {
    fn default() -> Self {
        Self::new()
    }
}

impl<T: Clone> Clone for OnceCell<T> {
    fn clone(&self) -> OnceCell<T> {
        let res = OnceCell::new();
        if let Some(value) = self.get() {
            match res.set(value.clone()) {
                Ok(()) => (),
                Err(_) => unreachable!(),
            }
        }
        res
    }
}

impl<T: PartialEq> PartialEq for OnceCell<T> {
    fn eq(&self, other: &Self) -> bool {
        self.get() == other.get()
    }
}
```

基于OnceCell实现惰性结构Lazy,惰性结构在第一次调用解引用的时候被赋值，随后使用这个值。 此结构强迫代码区分初始化必须有值及不必赋值的情况。

```
/// 惰性类型，在第一次使用时进行赋值和初始化
pub struct Lazy<T, F = fn() -> T> {
    //初始化可以为空
    cell: OnceCell<T>,
    //对cell做初始化赋值的函数
    init: Cell<Option<F>>,
```

```rust
}

impl<T, F> Lazy<T, F> {
    /// 函数作为变量被保存
    pub const fn new(init: F) -> Lazy<T, F> {
        Lazy { cell: OnceCell::new(), init: Cell::new(Some(init)) }
    }
}

impl<T, F: FnOnce() -> T> Lazy<T, F> {
    //完成赋值操作
    pub fn force(this: &Lazy<T, F>) -> &T {
        //如果cell为空，则用init作初始化赋值，注意这里init的take调用已经将init替换成
None,
        this.cell.get_or_init(|| match this.init.take() {
            Some(f) => f(),
            None => panic!("`Lazy` instance has previously been poisoned"),
        })
    }
}

//在对Lazy解引用时才进行赋值操作
impl<T, F: FnOnce() -> T> Deref for Lazy<T, F> {
    type Target = T;
    fn deref(&self) -> &T {
        Lazy::force(self)
    }
}

impl<T: Default> Default for Lazy<T> {
    /// Creates a new lazy value using `Default` as the initializing function.
    fn default() -> Lazy<T> {
        Lazy::new(T::default)
    }
}
```

# 小结

从内部可变类型，以及前面的NonNull, Unique, NonZeroSize,都是典型的由不安全类型到安全类型的实现例。