`Rc<T>` 分析

`Rc<T>` 主要解决堆内存多份借用的情况，相比于 `&Box<T>` 的解决方案， `Rc<T>` 可以基本上不用考虑生命周期导致的编码负担。同时利用伴生的 `Weak<T>` 解决了变量相互之间的循环引用问题。 相比与 `Box<T>` , `Rc<T>` 是更常用的堆内存智能指针类型。

`Rc<T>` 解决了两个数据结构互指的情况，这在结构体组合，循环链表，树，图的数据结构中都有大量的应用。 `Rc<T>` 的结构定义如下：

```
//在堆内存申请的结构体
//注意，这里使用了C语言的内存布局，在内存中的成员的顺序必须按照声明的顺序排列
#[repr(C)]
struct RcBox<T: ?Sized> {
    //拥有所有权的智能指针Rc<T>的计数
    strong: Cell<usize>,
    //不拥有所有权的智能指针Weak<T>的计数
    weak: Cell<usize>,
    value: T,
}

//和Unique<T>类似
pub struct Rc<T: ?Sized> {
    //堆内存块的指针
    ptr: NonNull<RcBox<T>>,
    //表示拥有内存块的所有权，内存块由本结构释放
    phantom: PhantomData<RcBox<T>>,
}
//没有堆内存块的所有权
pub struct Weak<T: ?Sized> {
   ptr: NonNull<RcBox<T>>,
}
```

在创建两个结构体变量互指的指针时，会遇到生命周期陷阱，无论先释放那个结构变量，都会导致另外那个结构变量出现悬垂指针。但如果在代码中时刻关注这种情况，那就太不RUST。
为此， `Rc<T>` 提供了weak和strong两种堆内存指针的方式， `Rc<T>` 申请的堆内存可以没有初始化，未初始化的堆内存可以生成 `WeakT>` 用于给其他结构访问堆内存。同时堆内存用strong的方式来保护 `Rc<T>` 在未初始化时不被读写。且weak和strong可以相互之间转换，这就以RUST方式解决了生命周期陷阱问题。
对 `Rc<T>` 建议的使用方式是各需要访问堆内存的类型仅使用 `Weak<T>` 作为平时的成员指针。当需要对 `Rc<T>` 做操作时，将 `Weak<T>` upgrade为 `Rc<T>` ,操作完成后，将 `Rc<T>` 生命周期终结。

`Rc<T>` 的创建方法及析构方法代码如下：

```rust
//由结构体成员生成结构的辅助方法
impl<T: ?Sized> Rc<T> {
    //获取内部的RcBox
    fn inner(&self) -> &RcBox<T> {
        unsafe { self.ptr.as_ref() }
    }

    //由成员创建结构体，注意，这里没有对strong做计数增操作
    //因此，此处的内部ptr应是被别的Rc<T>解封装出来的
    fn from_inner(ptr: NonNull<RcBox<T>>) -> Self {
        Self { ptr, phantom: PhantomData }
    }
    //由裸指针创建结构体，注意，这里没有对strong做计数增操作
    //因此，此处的内部ptr应是被别的Rc<T>解封装出来的
    unsafe fn from_ptr(ptr: *mut RcBox<T>) -> Self {
        Self::from_inner(unsafe { NonNull::new_unchecked(ptr) })
    }
}

impl<T> Rc<T> {
    //由已初始化变量创建Rc<T>
    pub fn new(value: T) -> Rc<T> {
        //首先创建RcBox<T>，然后生成Box<RcBox<T>>，随后用leak得到RcBox<T>的堆内存指
针，
        //用堆内存指针创建Rc<T>，内存申请由Box<T>实际执行
        Self::from_inner(
            Box::leak(box RcBox { strong: Cell::new(1), weak: Cell::new(1),
value }).into(),
        )
    }

    //用于创建一个互相引用场景的Rc<T>
    pub fn new_cyclic(data_fn: impl FnOnce(&Weak<T>) -> T) -> Rc<T> {
        // 下面与new函数代码类似，只是value没有初始化。
        // 因为value没有初始化，strong赋值为0，但可以支持Weak<T>的引用
        let uninit_ptr: NonNull<_> = Box::leak(box RcBox {
            strong: Cell::new(0),
            weak: Cell::new(1),
            value: mem::MaybeUninit::<T>::uninit(),
        })
        .into();

        //init_ptr后继会被初始化，但此时还没有
        let init_ptr: NonNull<RcBox<T>> = uninit_ptr.cast();

        //生成Weak
        let weak = Weak { ptr: init_ptr };

        // 利用回调闭包获得value的值，将weak传递出去是因为cyclic默认结构体初始化需要
使用weak.
```

```
        // 用回调函数的处理可以让初始化一次完成，以免初始化以后还要修改结构体的指针。
        let data = data_fn(&weak);

        unsafe {
            let inner = init_ptr.as_ptr();
            //addr_of_mut!可以万无一失，写入值后，初始化已经完成
            ptr::write(ptr::addr_of_mut!((*inner).value), data);

            //可以更新strong的值为1了
            let prev_value = (*inner).strong.get();
            debug_assert_eq!(prev_value, 0, "No prior strong references should
exist");
            (*inner).strong.set(1);
        }

        //strong登场
        let strong = Rc::from_inner(init_ptr);

        // 这里是因为strong整体拥有一个weak计数，所以此处不对wek做drop处理以维持weak
计数。前面的回调函数中应该使用weak.clone增加weak的计数。
        mem::forget(weak);
        strong
    }

    //生成一个未初始化的Rc<T>，选择了直接做内存申请
    pub fn new_uninit() -> Rc<mem::MaybeUninit<T>> {
        unsafe {
            //Rc自身的内存申请函数，见下文的分析
            Rc::from_ptr(Rc::allocate_for_layout(
                Layout::new::<T>(),
                |layout| Global.allocate(layout),
                |mem| mem as *mut RcBox<mem::MaybeUninit<T>>,
            ))
        }
    }

    //防止内存不足的创建函数
    pub fn try_new(value: T) -> Result<Rc<T>, AllocError> {
        // 就是用Box::try_new来完成try的工作
        Ok(Self::from_inner(
            Box::leak(Box::try_new(RcBox { strong: Cell::new(1), weak:
Cell::new(1), value })?)
                .into(),
        ))
    }

    //对未初始化的Rc的try new
    pub fn try_new_uninit() -> Result<Rc<mem::MaybeUninit<T>>, AllocError> {
        unsafe {
            //内存申请函数需要考虑申请不到的情况
            Ok(Rc::from_ptr(Rc::try_allocate_for_layout(
```

```
                Layout::new::<T>(),
                //就用Global Allocator，没有考虑其他的Allocator
                |layout| Global.allocate(layout),
                |mem| mem as *mut RcBox<mem::MaybeUninit<T>>,
            )?))
        }
    }
    ...
}

//堆内存申请函数
impl<T: ?Sized> Rc<T> {
   unsafe fn allocate_for_layout(
        value_layout: Layout,
        allocate: impl FnOnce(Layout) -> Result<NonNull<[u8]>, AllocError>,
        mem_to_rcbox: impl FnOnce(*mut u8) -> *mut RcBox<T>,
   ) -> *mut RcBox<T> {
        // 根据T计算RcBox需要的内存块布局，注意用RcBox<()>获取仅包含strong及weak两个成员的RcBox的Layout这个技巧
        //首先计算strong及weak两个成员的Layout,然后对内部T类型的Layout加以扩充，再做对齐的补充。
        let layout = Layout::new::<RcBox<()>>().extend(value_layout).unwrap().0.pad_to_align();
        unsafe {
            //要考虑不成功的可能性
            Rc::try_allocate_for_layout(value_layout, allocate, mem_to_rcbox)
                .unwrap_or_else(|_| handle_alloc_error(layout))
        }
   }

   unsafe fn try_allocate_for_layout(
        value_layout: Layout,
        allocate: impl FnOnce(Layout) -> Result<NonNull<[u8]>, AllocError>,
        mem_to_rcbox: impl FnOnce(*mut u8) -> *mut RcBox<T>,
   ) -> Result<*mut RcBox<T>, AllocError> {
        //计算需要的内存块布局Layout
        let layout = Layout::new::<RcBox<()>>().extend(value_layout).unwrap().0.pad_to_align();

        // 申请内存，有可能不成功
        let ptr = allocate(layout)?;

        // 将裸指针类型内存类型转换成*mut RcBox<xxx>类型，xxx有可能是MaybeUninit<T>，但也可能是初始化完毕的类型。总之，调用代码会保证初始化，所以此处正常的设置strong及weak，
        let inner = mem_to_rcbox(ptr.as_non_null_ptr().as_ptr());
        unsafe {
            debug_assert_eq!(Layout::for_value(&*inner), layout);

            ptr::write(&mut (*inner).strong, Cell::new(1));
            ptr::write(&mut (*inner).weak, Cell::new(1));
```

```
        }

        Ok(inner)
    }

    //根据一个裸指针来创建RcBox<T>，返回裸指针，这个函数完成时堆内存没有初始化，后继需要写入值
    unsafe fn allocate_for_ptr(ptr: *const T) -> *mut RcBox<T> {
        unsafe {
            Self::allocate_for_layout(
                // 用*const T获取Layout
                Layout::for_value(&*ptr),
                |layout| Global.allocate(layout),
                //此处应该也可以用mem as *mut RcBox<T>,
                |mem| (ptr as *mut RcBox<T>).set_ptr_value(mem),
            )
        }
    }

    //从Box<T>转换成RcBox<T>
    fn from_box(v: Box<T>) -> Rc<T> {
        unsafe {
            //解封装Box，获取堆内存指针
            let (box_unique, alloc) = Box::into_unique(v);
            let bptr = box_unique.as_ptr();

            let value_size = size_of_val(&*bptr);
            //获得* mut RcBox<T>
            let ptr = Self::allocate_for_ptr(bptr);

            // 将T的内容拷贝入RcBox的value
            ptr::copy_nonoverlapping(
                bptr as *const T as *const u8,
                &mut (*ptr).value as *mut _ as *mut u8,
                value_size,
            );

            // 重要，这里仅仅释放堆内存，但是如果堆内存中的T类型变量还有其他需要释放的内存，则没有处理，即没有调用drop(T)，drop(T)由新生成的RcBox<T>再释放的时候负责
            box_free(box_unique, alloc);

            // 生成Rc<T>
            Self::from_ptr(ptr)
        }
    }
}

//析构
unsafe impl<#[may_dangle] T: ?Sized> Drop for Rc<T> {
    //只要strong计数为零，就drop掉堆内存变量
    //Weak可以不依赖于内存初始化。
```

```
    fn drop(&mut self) {
        unsafe {
            //strong计数减1
            self.inner().dec_strong();
            if self.inner().strong() == 0 {
                // 触发堆内存变量的drop()操作
                ptr::drop_in_place(Self::get_mut_unchecked(self));

                // 对于strong整体会有一个weak计数，需要减掉
                // 这里实际上与C语言一样容易出错。
                self.inner().dec_weak();

                if self.inner().weak() == 0 {
                    //只有weak为0的时候才能够释放堆内存
                    Global.deallocate(self.ptr.cast(),
Layout::for_value(self.ptr.as_ref()));
                }
            }
        }
    }
}

impl<T: ?Sized> Deref for Rc<T> {
    type Target = T;

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

`Weak<T>`的结构体及创建，析构方法： 在RC方法内部，Weak可以由`Weak{ptr:self_ptr}`直接创建，可见前面代码的例子，但要注意weak计数和Weak变量需要匹配

```
impl<T> Weak<T> {
    //创建一个空的Weak
    pub fn new() -> Weak<T> {
        Weak { ptr: NonNull::new(usize::MAX as *mut RcBox<T>).expect("MAX is not
0") }
    }
}
//判断Weak是否为空的关联函数
pub(crate) fn is_dangling<T: ?Sized>(ptr: *mut T) -> bool {
    let address = ptr as *mut () as usize;
    address == usize::MAX
}
impl <T:?Sized> Weak<T> {
    //从Weak中获得堆内存中T类型的变量指针
    pub fn as_ptr(&self) -> *const T {
```

```
        let ptr: *mut RcBox<T> = NonNull::as_ptr(self.ptr);

        if is_dangling(ptr) {
            ptr as *const T
        } else {
            //返回T类型变量的指针
            unsafe { ptr::addr_of_mut!((*ptr).value) }
        }
    }
    //会消费掉Weak，获取T类型变量指针，此指针以后需要重新组建Weak<T>,否则
    //堆内存中的RcBox的weak会出现计数错误
    pub fn into_raw(self) -> *const T {
        let result = self.as_ptr();
        mem::forget(self);
        result
    }

    //ptr是从into_raw得到的返回值
    pub unsafe fn from_raw(ptr: *const T) -> Self {
        let ptr = if is_dangling(ptr as *mut T) {
            ptr as *mut RcBox<T>
        } else {
            //需要从T类型的指针恢复RcBox的指针
            let offset = unsafe { data_offset(ptr) };
            unsafe { (ptr as *mut RcBox<T>).set_ptr_value((ptr as *mut
u8).offset(-offset)) }
        };
        //RcBox的weak的计数已经有了这个计数
        Weak { ptr: unsafe { NonNull::new_unchecked(ptr) } }
    }

    //创建WeakInner
    fn inner(&self) -> Option<WeakInner<'_>> {
        if is_dangling(self.ptr.as_ptr()) {
            None
        } else {
            Some(unsafe {
                let ptr = self.ptr.as_ptr();
                WeakInner { strong: &(*ptr).strong, weak: &(*ptr).weak }
            })
        }
    }

    //从Weak得到Rc，如前所述，对Rc正确的打开方式应该是仅用Weak，然后适当的时候升级到
Rc<T>,
    //并且在使用完毕后就将Rc<T>生命周期终止掉，即这个函数返回的Rc<T>生命周期最好仅在一
个函数中。
    pub fn upgrade(&self) -> Option<Rc<T>> {
        //获取内部的RcBox
        let inner = self.inner()?;
        if inner.strong() == 0 {
```

```
            None
        } else {
            //对RcBox<T>的strong增加计数
            inner.inc_strong();
            //利用RcBox生成新的Rc<T>
            Some(Rc::from_inner(self.ptr))
        }
    }
}
impl <T:?Sized> Rc<T> {
    ...

    //生成新的Weak<T>
    pub fn downgrade(this: &Self) -> Weak<T> {
        //增加weak计数
        this.inner().inc_weak();
        // 确保不出错
        debug_assert!(!is_dangling(this.ptr.as_ptr()));
        // 生成Weak<T>
        Weak { ptr: this.ptr }
    }
}
```

`Rc<T>`的其他方法：

```
impl<T: Clone> Rc<T> {
    //Rc<T> 实际上是需要配合RefCell<T>来完成对堆内存的修改需求
    //下面的函数用了类似写时复制的方式，仅能在某些场景下使用
    pub fn make_mut(this: &mut Self) -> &mut T {
        if Rc::strong_count(this) != 1 {
            // 如果Rc多于一个，则创建一个拷贝的变量
            // 申请一个未初始化的Rc
            let mut rc = Self::new_uninit();
            unsafe {
                //将self中的value值写入新创建的变量
                let data = Rc::get_mut_unchecked(&mut rc);
                (**this).write_clone_into_raw(data.as_mut_ptr());
                //这里把this代表的Rc释放掉，并赋以新值。
                //make_mut的本意是从this中生成一个mut，因此将this代表的Rc<T>释放掉
是合乎
                //函数的意义的
                *this = rc.assume_init();
            }
        } else if Rc::weak_count(this) != 0 {
            // 如果Rc<T>仅有一个strong引用，但有其他的weak引用
            // 同样需要新建一个Rc<T>
            let mut rc = Self::new_uninit();
            unsafe {
                //下面用了与strong !=1 的情况的不同写法，但应该完成了同样的工作
                let data = Rc::get_mut_unchecked(&mut rc);
```

```
                data.as_mut_ptr().copy_from_nonoverlapping(&**this, 1);

                //将strong引用减去，堆内存不再存在strong引用
                this.inner().dec_strong();
                // strong已经为0，所以将strong的weak计数减掉
                this.inner().dec_weak();
                //不能用*this = 的表达，因为会导致对堆内存变量的释放，这不符合语义。
                ptr::write(this, rc.assume_init());
            }
        }
        //已经确保了只有一个Rc<T>，且没有Weak<T>，可以任意对堆变量做修改了
        unsafe { &mut this.ptr.as_mut().value }
    }
}
```

Clone trait实现：

```
impl<T: ?Sized> Clone for Rc<T> {
   //clone就是增加一个strong的计数
   fn clone(&self) -> Rc<T> {
       self.inner().inc_strong();
       Self::from_inner(self.ptr)
   }
}

impl<T: ?Sized> Clone for Weak<T> {
   fn clone(&self) -> Weak<T> {
       if let Some(inner) = self.inner() {
           inner.inc_weak()
       }
       Weak { ptr: self.ptr }
   }
}
```

对`Rc<MaybeUninit<T>>`初始化后assume_init实现方法：

```
impl<T> Rc<mem::MaybeUninit<T>> {
    pub unsafe fn assume_init(self) -> Rc<T> {
        //先用ManuallyDrop将self封装以便不对self做drop操作
        //然后取出内部的堆指针形成新的Rc<T>。
        Rc::from_inner(mem::ManuallyDrop::new(self).ptr.cast())
    }
}
```

`Rc<T>`其他方法：

```rust
impl<T: ?Sized> Rc<T> {
    //相当于Rc<T>的Leak函数
    pub fn into_raw(this: Self) -> *const T {
        let ptr = Self::as_ptr(&this);
        //把堆内存指针取出后，由调用代码负责释放，
        //本结构体要规避后继的释放操作
        mem::forget(this);
        ptr
    }

    //获得堆内存变量的指针，不会涉及安全问题,注意，这里ptr不是堆内存块的首地址，而是向后有偏移
    //因为RcBox<T>采用C语言的内存布局，所以value在最后
    pub fn as_ptr(this: &Self) -> *const T {
        let ptr: *mut RcBox<T> = NonNull::as_ptr(this.ptr);

        unsafe { ptr::addr_of_mut!((*ptr).value) }
    }

    //从堆内存T类型变量的指针重建Rc<T>，注意，这里的ptr一般是调用Rc<T>::into_raw()获得的裸指针
    //ptr不是堆内存块首地址，需要减去strong和weak的内存大小
    pub unsafe fn from_raw(ptr: *const T) -> Self {
        let offset = unsafe { data_offset(ptr) };

        // 减去偏移量，得到正确的RcBox堆内存的首地址
        let rc_ptr =
            unsafe { (ptr as *mut RcBox<T>).set_ptr_value((ptr as *mut u8).offset(-offset)) };

        unsafe { Self::from_ptr(rc_ptr) }
    }
```

into_raw, from_raw要成对使用，否则就必须对这两个方法的内存由清晰的认知。否则极易出现问题。 `Rc<T>` 转换为 `Weak<T>`

```rust
    pub fn get_mut(this: &mut Self) -> Option<&mut T> {
        if Rc::is_unique(this) { unsafe { Some(Rc::get_mut_unchecked(this)) } } else { None }
    }

    pub unsafe fn get_mut_unchecked(this: &mut Self) -> &mut T {
        unsafe { &mut (*this.ptr.as_ptr()).value }
    }

}
```

`Arc<T>` 的代码实现

`Arc<T>` 是 `Rc<T>` 的多线程版本，实际上，就连代码都基本类似，只是把计数值的类型换成了原子变量 `Arc<T>` 类型结构定义如下：

```
//在堆内存分配的结构体
#[repr(C)]
struct ArcInner<T: ?Sized> {
    //用原子变量实现计数，使得计数修改不会因多线程竞争而出错
    //AtomicUsize 如下:
    // pub struct AtomicUsize { v: UnsafeCell<usize>}
    strong: atomic::AtomicUsize,
    weak: atomic::AtomicUsize,
    data: T,
}

//支持Send
unsafe impl<T: ?Sized + Sync + Send> Send for ArcInner<T> {}
//支持Sync
unsafe impl<T: ?Sized + Sync + Send> Sync for ArcInner<T> {}

//Arc<T>的结构
pub struct Arc<T: ?Sized> {
    ptr: NonNull<ArcInner<T>>,
    phantom: PhantomData<ArcInner<T>>,
}
//对Send支持
unsafe impl<T: ?Sized + Sync + Send> Send for Arc<T> {}
//对Sync支持
unsafe impl<T: ?Sized + Sync + Send> Sync for Arc<T> {}

//Weak<T>的结构
pub struct Weak<T: ?Sized> {
    ptr: NonNull<ArcInner<T>>,
}

unsafe impl<T: ?Sized + Sync + Send> Send for Weak<T> {}
unsafe impl<T: ?Sized + Sync + Send> Sync for Weak<T> {}
```

`ArcInner<T>` 对应 `RcBox<T>`，`Arc<T>` 对应 `Rc<T>`，`sync::Weak<T>` 对应 `rc::Weak<T>`。逻辑与 `Rc<T>` 模块的逻辑都基本相同。 `Arc<T>` 除了 `ArcInner<T>` 与 `RcBox<T>` 有区别，计数器用原子变量实现，使得计数器的加减操作不会受多线程数据竞争的影响，从而使得 `Arc<T>` 能够在多线程环境下使用。这里需要注意 `Rc<T>` 及 `Arc<T>` 实际上仅仅是不可变引用的多线程替代(多于两个以上)，因此 `Arc<T>` 的实现中仅仅关注 `Arc<T>` 类型本身的多线程共享的保护机制。至于内部的泛型类型变量data，仍然需要泛型类型自身对多线程共享的实现。

与 `Rc<T>` 相同， `Arc<T>` 也提供了weak和strong两种堆内存指针的方式， `Arc<T>` 申请的堆内存可以没有初始化，未初始化的堆内存可以生成 `WeakT>` 用于给其他结构访问堆内存。同时堆内存用strong的方式来保护 `Arc<T>` 在未初始化时不被读写。且weak和strong可以相互之间转换，这就以rust方式解决了生命周期陷阱问题。 利用 `Weak<T>` 做指针的结构体，在需要访问堆内存时，可以从 `Weak<T>` 另外创建 `Arc<T>` ，完成访问后即可让创建的 `Arc<T>` 生命周期终结。实际上，各需要访问堆内存的类型仅使用 `Weak<T>` 应该是一个非常好的做法。

`Arc<T>` 的创建方法及析构方法代码如下：

```
//已经存在堆内存，从堆内存来创建Arc<T>
impl<T:?Sized> Arc<T> {
   //注意这里没有增加strong计数，
   fn from_inner(ptr: NonNull<ArcInner<T>>) -> Self {
       Self { ptr, phantom: PhantomData }
   }

   //注意这里没有增加strong的计数
   unsafe fn from_ptr(ptr: *mut ArcInner<T>) -> Self {
       unsafe { Self::from_inner(NonNull::new_unchecked(ptr)) }
   }
}

impl<T> Arc<T> {
    //由已初始化变量创建Arc<T>
    pub fn new(data: T) -> Arc<T> {
        //首先创建ArcInner<T>，然后生成Box<ArcInner<T>>，随后用leak得到ArcInner<T>
的堆内存指针，
        //用堆内存指针创建Rc<T>，内存申请由Box<T>实际执行
        let x: Box<_> = box ArcInner {
            strong: atomic::AtomicUsize::new(1),
            weak: atomic::AtomicUsize::new(1),
            data,
        };
        Self::from_inner(Box::leak(x).into())
   }
    //用于创建一个互相引用场景的Arc<T>
   pub fn new_cyclic(data_fn: impl FnOnce(&Weak<T>) -> T) -> Arc<T> {
        // 下面与new函数代码类似，只是value没有初始化。
        // 因为value没有初始化，strong赋值为0，但可以支持Weak<T>的引用
        let uninit_ptr: NonNull<_> = Box::leak(box ArcInner {
            strong: atomic::AtomicUsize::new(0),
            weak: atomic::AtomicUsize::new(1),
            data: mem::MaybeUninit::<T>::uninit(),
        })
        .into();

        //init_ptr后继会被初始化，但此时还没有
        let init_ptr: NonNull<ArcInner<T>> = uninit_ptr.cast();
```

```
        //生成Weak
        let weak = Weak { ptr: init_ptr };

        // 利用回调闭包获得value的值，将weak传递出去是因为cyclic默认结构体初始化需要使用weak.
        // 用回调函数的处理可以让初始化一次完成，以免初始化以后还要修改结构体的指针。
        let data = data_fn(&weak);

        // 完成对值的初始化.并转化Weak为Strong.
        unsafe {
            let inner = init_ptr.as_ptr();
            //addr_of_mut!可以万无一失，写入值后，初始化已经完成
            ptr::write(ptr::addr_of_mut!((*inner).data), data);

            //可以更新strong的值为1了,注意这里的原子函数，这个函数不会被其他线程打断导致更新失败
            let prev_value = (*inner).strong.fetch_add(1, Release);
            debug_assert_eq!(prev_value, 0, "No prior strong references should exist");
        }

        //strong登场
        let strong = Arc::from_inner(init_ptr);

        // 这里是因为strong整体拥有一个weak计数，所以此处不对weak做drop处理以维持weak计数。前面的回调函数中应该使用weak.clone增加weak的计数。
        mem::forget(weak);
        strong
    }

    //生成一个未初始化的Arc<T>，选择直接做内存申请
    pub fn new_uninit() -> Arc<mem::MaybeUninit<T>> {
        unsafe {
            //Arc自身的内存申请函数，后继有分析
            Arc::from_ptr(Arc::allocate_for_layout(
                Layout::new::<T>(),
                |layout| Global.allocate(layout),
                |mem| mem as *mut ArcInner<mem::MaybeUninit<T>>,
            ))
        }
    }
    //防止内存不足的创建函数
    pub fn try_new(data: T) -> Result<Arc<T>, AllocError> {
        // 就是用Box::try_new来完成try的工作
        let x: Box<_> = Box::try_new(ArcInner {
            strong: atomic::AtomicUsize::new(1),
            weak: atomic::AtomicUsize::new(1),
            data,
        })?;
        Ok(Self::from_inner(Box::leak(x).into()))
```

```
    }

     //对未初始化的Rc的try new
    pub fn try_new_uninit() -> Result<Arc<mem::MaybeUninit<T>>, AllocError> {
        unsafe {
             //内存申请函数需要考虑申请不到的情况
            Ok(Arc::from_ptr(Arc::try_allocate_for_layout(
                Layout::new::<T>(),
                 //就用Global Allocator，没有考虑其他Allocator
                |layout| Global.allocate(layout),
                |mem| mem as *mut ArcInner<mem::MaybeUninit<T>>,
            )?))
        }
    }
     ...
}

//堆内存申请函数
impl<T: ?Sized> Arc<T> {
    unsafe fn allocate_for_layout(
        value_layout: Layout,
        allocate: impl FnOnce(Layout) -> Result<NonNull<[u8]>, AllocError>,
        mem_to_arcinner: impl FnOnce(*mut u8) -> *mut ArcInner<T>,
    ) -> *mut ArcInner<T> {
        // 根据T计算ArcInner需要的内存块布局，与Rc<T>的同名函数基本相同，请参考
        let layout = Layout::new::<ArcInner<()>>
().extend(value_layout).unwrap().0.pad_to_align();
        unsafe {
            //要考虑不成功的可能性
            Arc::try_allocate_for_layout(value_layout, allocate,
mem_to_arcinner)
                .unwrap_or_else(|_| handle_alloc_error(layout))
        }
    }

    unsafe fn try_allocate_for_layout(
        value_layout: Layout,
        allocate: impl FnOnce(Layout) -> Result<NonNull<[u8]>, AllocError>,
        mem_to_arcinner: impl FnOnce(*mut u8) -> *mut ArcInner<T>,
    ) -> Result<*mut ArcInner<T>, AllocError> {
        //计算需要的内存块布局layout
        let layout = Layout::new::<ArcInner<()>>
().extend(value_layout).unwrap().0.pad_to_align();

        // 申请内存，有可能不成功
        let ptr = allocate(layout)?;

        // 将裸指针类型内存类型转换成*mut ArcInner<xxx>类型，xxx有可能是
MaybeUninit<T>，但也可能是初始化完毕的类型。总之，调用代码会保证初始化，所以此处正常的
设置strong及weak，
        let inner = mem_to_arcinner(ptr.as_non_null_ptr().as_ptr());
```

```
        debug_assert_eq!(unsafe { Layout::for_value(&*inner) }, layout);

        unsafe {
            ptr::write(&mut (*inner).strong, atomic::AtomicUsize::new(1));
            ptr::write(&mut (*inner).weak, atomic::AtomicUsize::new(1));
        }

        Ok(inner)
    }

    //根据一个裸指针来创建ArcInner<T>，返回裸指针，这个函数完成时堆内存没有初始化，后继需要写入值
    unsafe fn allocate_for_ptr(ptr: *const T) -> *mut ArcInner<T> {
        unsafe {
            Self::allocate_for_layout(
                // 用*const T获取Layout
                Layout::for_value(&*ptr),
                |layout| Global.allocate(layout),
                //此处应该也可以用mem as *mut ArcInner<T>,
                |mem| (ptr as *mut ArcInner<T>).set_ptr_value(mem) as *mut ArcInner<T>,
            )
        }
    }

    //从Box<T>转换成Arc<T>
     fn from_box(v: Box<T>) -> Arc<T> {
        unsafe {
            //解封装Box，获取堆内存指针
            let (box_unique, alloc) = Box::into_unique(v);
            let bptr = box_unique.as_ptr();

            let value_size = size_of_val(&*bptr);
            //获得* mut ArcInner<T>
            let ptr = Self::allocate_for_ptr(bptr);

            // 将T的内容拷贝入ArcInner的value
            ptr::copy_nonoverlapping(
                bptr as *const T as *const u8,
                &mut (*ptr).data as *mut _ as *mut u8,
                value_size,
            );

            // 重要，这里仅仅释放堆内存，但是如果堆内存中的T类型变量还有其他需要释放的内存，则没有处理，即没有调用drop(T)，drop(T)由新生成的ArcInner<T>再释放的时候负责
            box_free(box_unique, alloc);

            // 生成Arc<T>
            Self::from_ptr(ptr)
        }
    }
```

```
}

//析构
unsafe impl<#[may_dangle] T: ?Sized> Drop for Arc<T> {
    fn drop(&mut self) {
        //如果当前的strong不是1,则返回，fetch_xxx函数返回之前的值
        if self.inner().strong.fetch_sub(1, Release) != 1 {
            return;
        }

        acquire!(self.inner().strong);

        unsafe {
            //见下面代码的分析
            self.drop_slow();
        }
    }
}
impl <T:?Sized> Arc<T> {
    ...
    unsafe fn drop_slow(&mut self) {
        // 对堆内存的变量做drop操作，注意，这里不释放堆内存，只是释放变量所有权
        unsafe { ptr::drop_in_place(Self::get_mut_unchecked(self)) };

        // 所有的strong会创建一个Weak，对这个Weak做drop操作
        drop(Weak { ptr: self.ptr });
    }
    ...
}

impl<T: ?Sized> Deref for Arc<T> {
    type Target = T;

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

`Weak<T>`的结构体及创建，析构方法： 在RC方法内部，Weak可以由`Weak{ptr:self_ptr}`直接创建，可见前面代码的例子，但要注意weak计数和Weak变量需要匹配

```
impl<T> Weak<T> {
    //创建一个空的Weak
    pub fn new() -> Weak<T> {
        Weak { ptr: NonNull::new(usize::MAX as *mut ArcInner<T>).expect("MAX is
not 0") }
    }
}
```

```rust
struct WeakInner<'a> {
    weak: &'a atomic::AtomicUsize,
    strong: &'a atomic::AtomicUsize,
}
//判断Weak是否为空的关联函数
pub(crate) fn is_dangling<T: ?Sized>(ptr: *mut T) -> bool {
    let address = ptr as *mut () as usize;
    address == usize::MAX
}

impl <T:?Sized> Weak<T> {
    pub fn as_ptr(&self) -> *const T {
        let ptr: *mut ArcInner<T> = NonNull::as_ptr(self.ptr);

        if is_dangling(ptr) {
            ptr as *const T
        } else {
            //返回T类型变量的指针
            unsafe { ptr::addr_of_mut!((*ptr).data) }
        }
    }
    //会消费掉Weak，获取T类型变量指针，此指针以后需要重新组建Weak<T>,否则
    //堆内存中的ArcInner的weak会出现计数错误
    pub fn into_raw(self) -> *const T {
        let result = self.as_ptr();
        mem::forget(self);
        result
    }

    //ptr是从into_raw得到的返回值
    pub unsafe fn from_raw(ptr: *const T) -> Self {
        let ptr = if is_dangling(ptr as *mut T) {
            ptr as *mut ArcInner<T>
        } else {
            //需要从T类型的指针恢复ArcInner的指针
            let offset = unsafe { data_offset(ptr) };
            unsafe { (ptr as *mut ArcInner<T>).set_ptr_value((ptr as *mut
u8).offset(-offset)) }
        };
        //ArcInner的weak的计数已经有了这个计数
        Weak { ptr: unsafe { NonNull::new_unchecked(ptr) } }
    }

    //创建一个WeakInner
    fn inner(&self) -> Option<WeakInner<'_>> {
        if is_dangling(self.ptr.as_ptr()) {
            None
        } else {
            //获取ArcInner<T>中strong和weak的引用
            Some(unsafe {
                let ptr = self.ptr.as_ptr();
```

```
                WeakInner { strong: &(*ptr).strong, weak: &(*ptr).weak }
            })
        }
    }
    //从Weak得到Arc，如前所述，对Arc正确的打开方式应该是仅用Weak，然后适当的时候升级到
Arc<T>
    //并且在使用完毕后就将Arc<T>生命周期终止掉，即这个函数返回Arc<T>生命周期最好仅在一
个函数中。
    pub fn upgrade(&self) -> Option<Arc<T>> {
        //获取内部的ArcInner
        let inner = self.inner()?;
        //原子操作获得strong的值
        let mut n = inner.strong.load(Relaxed);

        //因为是多线程操作，所以此时n已经可能被改写，所以用loop
        //来确保n在已经改写的情况下正确
        loop {
            //如果strong是0，那堆内存已经被释放掉，不能再使用
            if n == 0 {
                return None;
            }

            // 不能多于最大的引用数目
            if n > MAX_REFCOUNT {
                abort();
            }

            //以下确保在strong当前值是n的时候做加1操作
            match inner.strong.compare_exchange_weak(n, n + 1, Acquire, Relaxed)
{
                //当前值为1且已经加1，生成Arc<T>
                Ok(_) => return Some(Arc::from_inner(self.ptr)), // null checked
above
                //如果当前值不为n，则将n设置为当前值，进入下一轮循环。
                Err(old) => n = old,
            }
        }
    }
}

impl <T:?Sized> Arc<T> {
    ...

    //生成新的Weak<T>
    pub fn downgrade(this: &Self) -> Weak<T> {
        // 获取weak count.
        let mut cur = this.inner().weak.load(Relaxed);

        //要确定当前的weak count与上面取得一致
        loop {
            // 如果是usize::MAX，证明在创建过程中，等创建完毕后
```

```
            // 再获取一次
            if cur == usize::MAX {
                hint::spin_loop();
                cur = this.inner().weak.load(Relaxed);
                continue;
            }

            //确保在weak与当前值一致的情况下做原子操作，将weak加1
            match this.inner().weak.compare_exchange_weak(cur, cur + 1, Acquire,
Relaxed) {
                Ok(_) => {
                    // 确保不创建对不存在的变量的Weak
                    debug_assert!(!is_dangling(this.ptr.as_ptr()));
                    //创建Weak
                    return Weak { ptr: this.ptr };
                }
                //如果当前的值与取值不一致，将取值更换为当前值，再做一次循环
                Err(old) => cur = old,
            }
        }
    }
}
```

以上代码中，对于多线程的处理需要额外注意并理解。这是原子变量处理多线程的典型用法

`Arc<T>`的其他方法：

```
impl<T: Clone> Arc<T> {
    //Rc<T> 实际上是需要配合RefCell<T>来完成对堆内存的修改需求
    //下面的函数用了类似写时复制的方式，仅能在某些场景下使用
    pub fn make_mut(this: &mut Self) -> &mut T {
        //判断strong的值是否为1，如果为1，则设置为0，以防止其他线程做修改
        if this.inner().strong.compare_exchange(1, 0, Acquire,
Relaxed).is_err() {
            // strong不为1，需要创建一个复制的Arc<T>变量
            let mut arc = Self::new_uninit();
            unsafe {
                let data = Arc::get_mut_unchecked(&mut arc);
                (**this).write_clone_into_raw(data.as_mut_ptr());
                *this = arc.assume_init();
            }
        } else if this.inner().weak.load(Relaxed) != 1 {
            //当前为原strong为1且已经strong已经做了减1操作为0
            //那此时weak如果为1，证明没有多余的Weak<T>被派生
            //如果weak不为1，则证明有其他的Weak<T>存在，需要创建一个复制的Arc<T>

            //这里因为strong已经被减1，所以本线程已经没有Arc<T>，所以创建一个
            //Weak并由此变量的drop完成对weak计数的处理
            let _weak = Weak { ptr: this.ptr };
```

```
            // 创建一个新的复制的Arc<T>
            let mut arc = Self::new_uninit();
            unsafe {
                let data = Arc::get_mut_unchecked(&mut arc);
                data.as_mut_ptr().copy_from_nonoverlapping(&**this, 1);
                ptr::write(this, arc.assume_init());
            }
        } else {
            // strong及weak都是1，则恢复strong为1，直接使用当前的Arc<T>
            this.inner().strong.store(1, Release);
        }

        //返回&mut T
        unsafe { Self::get_mut_unchecked(this) }

    }
}
```

上面的函数与`Rc<T>::make_mut()`有所不同，是因为原子变量的原因带来的，可以对比学习，更深刻的了解原子变量。 Clone trait实现：

```
impl<T: ?Sized> Clone for Arc<T> {
    fn clone(&self) -> Arc<T> {
        //增加一个strong计数
        let old_size = self.inner().strong.fetch_add(1, Relaxed);

        if old_size > MAX_REFCOUNT {
            abort();
        }
        //从内部创建一个新的ARC<T>
        Self::from_inner(self.ptr)
    }
}

impl<T: ?Sized> Clone for Weak<T> {
    fn clone(&self) -> Weak<T> {
        if let Some(inner) = self.inner() {
            inner.inc_weak()
        }
        Weak { ptr: self.ptr }
    }
    fn clone(&self) -> Weak<T> {
        let inner = if let Some(inner) = self.inner() {
            inner
        } else {
            //inner不存在，直接创建一个Weak<T>
            return Weak { ptr: self.ptr };
        };
        //对weak计数加1
```

```
        let old_size = inner.weak.fetch_add(1, Relaxed);

        if old_size > MAX_REFCOUNT {
            abort();
        }
        //创建Weak<T>
        Weak { ptr: self.ptr }
    }
}
```

对`Arc<MaybeUninit<T>>`初始化后assume_init实现方法：

```
impl<T> Arc<mem::MaybeUninit<T>> {
    pub unsafe fn assume_init(self) -> Arc<T> {
        //先用ManuallyDrop将self封装以便不对self做drop操作
        //然后取出内部的堆指针形成新的Arc<T>。
        Arc::from_inner(mem::ManuallyDrop::new(self).ptr.cast())
    }
}
```

`Arc<T>`其他方法：

```
impl<T: ?Sized> Arc<T> {
    //相当于Arc<T>的Leak函数
    pub fn into_raw(this: Self) -> *const T {
        let ptr = Self::as_ptr(&this);
        //把堆内存指针取出后，由调用代码负责释放，
        //本结构体要规避后继的释放操作
        mem::forget(this);
        ptr
    }

    //获得堆内存变量的指针，不会涉及安全问题,注意，这里ptr不是堆内存块的首地址，而是向后有偏移
    //因为ArcInner<T>采用C语言的内存布局，所以value在最后
    pub fn as_ptr(this: &Self) -> *const T {
        let ptr: *mut ArcInner<T> = NonNull::as_ptr(this.ptr);

        unsafe { ptr::addr_of_mut!((*ptr).value) }
    }

    //从堆内存T类型变量的指针重建Arc<T>，注意，这里的ptr一般是调用`Arc<T>::into_raw()获得的裸指针
    //ptr不是堆内存块首地址，需要减去strong和weak的内存大小
    pub unsafe fn from_raw(ptr: *const T) -> Self {
        let offset = unsafe { data_offset(ptr) };

        // 减去偏移量，得到正确的ArcInner堆内存的首地址
```

```
        let rc_ptr =
            unsafe { (ptr as *mut ArcInner<T>).set_ptr_value((ptr as *mut
u8).offset(-offset)) };

        unsafe { Self::from_ptr(rc_ptr) }
    }
```

into_raw, from_raw要成对使用，否则就必须对这两个方法的内存有清晰的认知。否则极易出现问题。

```
    pub fn get_mut(this: &mut Self) -> Option<&mut T> {
        if this.is_unique() { unsafe { Some(Arc::get_mut_unchecked(this)) } }
else { None }
    }

    pub unsafe fn get_mut_unchecked(this: &mut Self) -> &mut T {
        unsafe { &mut (*this.ptr.as_ptr()).data }
    }

}
```