智能指针

Box代码分析

除了数组外的智能指针的堆内存申请，一般都先由Box来完成，然后再将申请到的内存转移到智能指针自身的结构中。

以下为Box结构定义及创建方法相关内容：

```
//Box结构
pub struct Box<
    T: ?Sized,
    //默认的堆内存申请为Global单元结构体，可修改为其他
    A: Allocator = Global,
  //用Unique<T>表示对申请的堆内存拥有所有权
>(Unique<T>, A);
```

Box的创建方法：

```
//以Global作为默认的堆内存分配器的实现
impl<T> Box<T> {
    pub fn new(x: T) -> Self {
        //box 是关键字，就是实现从堆内存申请内存，写入内容然后形成Box<T>
        //这个关键字的功能可以从后继的方法中分析出来，此方法实际等同与new_in(x,
Global);
        box x
    }
    ...
}

//不限定堆内存分配器的更加通用的方法实现
impl<T, A: Allocator> Box<T, A> {

    //Box::new(x) 实际上的逻辑等同与 Box::new_in(x, Global)
    pub fn new_in(x: T, alloc: A) -> Self {
        //new_uninit_in见后面代码分析
        let mut boxed = Self::new_uninit_in(alloc);
        unsafe {
            //实际是MaybeUninit<T>::as_mut_ptr()得到*mut T, ::write将x写入申请的堆
内存中
            boxed.as_mut_ptr().write(x);
            //从Box<MaybeUninit<T>,A>转换为Box<T,A>
            boxed.assume_init()
        }
```

```rust
    }

    //内存部分章节有过分析
    pub fn new_uninit_in(alloc: A) -> Box<mem::MaybeUninit<T>, A> {
        //获取Layout以便申请堆内存
        let layout = Layout::new::<mem::MaybeUninit<T>>();
        //见后面的代码分析
        match Box::try_new_uninit_in(alloc) {
            Ok(m) => m,
            Err(_) => handle_alloc_error(layout),
        }
    }

    //内存申请的真正执行函数
    pub fn try_new_uninit_in(alloc: A) -> Result<Box<mem::MaybeUninit<T>, A>,
AllocError> {
        //申请内存需要的内存Layout
        let layout = Layout::new::<mem::MaybeUninit<T>>();
        //申请内存并完成错误处理，cast将NonNull<[u8]>转换为NonNull<MaybeUninit<T>>
        //NonNull<MaybeUninit<T>>.as_ptr为 *mut <MaybeUninit<T>>
        //后继Box的drop会释放此处的内存
        //from_raw_in即将ptr转换为Unique<T>并形成Box结构变量
        let ptr = alloc.allocate(layout)?.cast();
        unsafe { Ok(Box::from_raw_in(ptr.as_ptr(), alloc)) }
    }

    ...
}

impl<T, A: Allocator> Box<mem::MaybeUninit<T>, A> {
    //申请的未初始化内存，初始化后，应该调用这个函数将
    //Box<MaybeUninit<T>>转换为Box<T>,
    pub unsafe fn assume_init(self) -> Box<T, A> {
        //因为类型不匹配，且无法强制转换，所以先将self消费掉并获得
        //堆内存的裸指针，再用裸指针生成新的Box，完成类型转换V
        let (raw, alloc) = Box::into_raw_with_allocator(self);
        unsafe { Box::from_raw_in(raw as *mut T, alloc) }
    }
}
impl<T: ?Sized, A: Allocator> Box<T, A> {
    //从裸指针构建Box类型，裸指针应该是申请堆内存返回的指针
    //用这个方法生成Box，当Box被drop时，会引发对裸指针的释放操作
    pub unsafe fn from_raw_in(raw: *mut T, alloc: A) -> Self {
        //由裸指针生成Unique，再生成Box
        Box(unsafe { Unique::new_unchecked(raw) }, alloc)
    }

    //此函数会将传入的b:Box消费掉，并将内部的Unique也消费掉，
    //返回裸指针，此时裸指针指向的内存已经不会再被drop.
    pub fn into_raw_with_allocator(b: Self) -> (*mut T, A) {
        let (leaked, alloc) = Box::into_unique(b);
```

```
        (leaked.as_ptr(), alloc)
    }
    pub fn into_unique(b: Self) -> (Unique<T>, A) {
        //对b的alloc做了一份拷贝
        let alloc = unsafe { ptr::read(&b.1) };
        //Box::leak(b)返回&mut T可变引用，具体分析见下文
        //leak(b)生成的&mut T实质上已经不会有Drop调用释放
        (Unique::from(Box::leak(b)), alloc)
    }

    //将b消费掉，并将b内的变量取出来返回
    pub fn leak<'a>(b: Self) -> &'a mut T
    where
        A: 'a,
    {
        //生成ManuallyDrop<Box<T>>，消费掉了b，此时不会再对b做Drop调用，导致了一个内存leak
        //ManuallyDrop<Box<T>>.0 是Box<T>，ManuallyDrop<T>没有.0的语法，因此会先做解引用，是&Box<T>
        //&Box<T>.0即Unique<T>，Unique<T>.as_ptr获得裸指针，然后利用unsafe代码生成可变引用
        unsafe { &mut *mem::ManuallyDrop::new(b).0.as_ptr() }
    }
    ...
}

unsafe impl< T: ?Sized, A: Allocator> Drop for Box<T, A> {
    fn drop(&mut self) {
        // FIXME: Do nothing, drop is currently performed by compiler.
    }
}
```

以上是Box的最常用的创建方法的代码。对于所有的堆申请，申请后的内存变量类型是MaybeUninit，然后对MaybeUninit用ptr::write完成初始化，随后再assume_init进入正常变量状态，这是rust的基本套路。

Box的Pin方法：

```
impl<T> Box<T> {
    //如果T没有实现Unpin Trait，则内存不会移动
    pub fn pin(x: T) -> Pin<Box<T>> {
        //任意的指针可以Into到Pin,因为Pin实现了任意类型的可变引用的From trait
        (box x).into()
    }
    ...
}
impl<T:?Sized> Box<T> {}
    pub fn into_pin(boxed: Self) -> Pin<Self>
    where
```

```
        A: 'static,
    {
        unsafe { Pin::new_unchecked(boxed) }
    }
    ...
}
//不限定堆内存分配器的更加通用的方法实现
impl<T, A: Allocator> Box<T, A> {
    //生成Box<T>后，在用Into<Pin> Trait生成Pin<Box>
    pub fn pin_in(x: T, alloc: A) -> Pin<Self>
    where
        A: 'static,
    {
        Self::new_in(x, alloc).into()
    }

    ...
}
```

Box<[T]>的方法：

```
impl<T,A:Allocator> Box<T, A> {
    //切片
    pub fn into_boxed_slice(boxed: Self) -> Box<[T], A> {
        //要转换指针类型，需要先得到裸指针
        let (raw, alloc) = Box::into_raw_with_allocator(boxed);
        //将裸指针转换为切片裸指针，再生成Box，此处因为不知道长度，
        //只能转换成长度为1的切片指针
        unsafe { Box::from_raw_in(raw as *mut [T; 1], alloc) }
    }
    ...
}

impl<T, A: Allocator> Box<[T], A> {
    //使用RawVec作为底层堆内存管理结构，并转换为Box
    pub fn new_uninit_slice_in(len: usize, alloc: A) ->
Box<[mem::MaybeUninit<T>], A> {
        unsafe { RawVec::with_capacity_in(len, alloc).into_box(len) }
    }

    //内存清零
    pub fn new_zeroed_slice_in(len: usize, alloc: A) ->
Box<[mem::MaybeUninit<T>], A> {
        unsafe { RawVec::with_capacity_zeroed_in(len, alloc).into_box(len) }
    }
}
impl<T, A: Allocator> Box<[mem::MaybeUninit<T>], A> {
    //初始化完毕,
    pub unsafe fn assume_init(self) -> Box<[T], A> {
        let (raw, alloc) = Box::into_raw_with_allocator(self);
```

```
        unsafe { Box::from_raw_in(raw as *mut [T], alloc) }
    }
}
```

其他方法及trait:

```
impl<T: Default> Default for Box<T> {
    /// Creates a `Box<T>`, with the `Default` value for T.
    fn default() -> Self {
        box T::default()
    }
}

impl<T,A:Allocator> Box<T, A> {
    //消费掉Box，获取内部变量
    pub fn into_inner(boxed: Self) -> T {
        //对Box的*操作就是完成Box接口从堆内存到栈内存拷贝
        //然后调用Box的drop，返回栈内存。编译器内置的操作
        *boxed
    }
    ...
}
```

以上即为Box创建及析构的所有相关代码，其中较难理解的是leak方法。在RUST中，惯例对内存申请一般会使用Box来实现，如果需要将申请的内存以另外的智能指针结构做封装，则调用Box::leak将堆指针传递出来

RawVec代码分析

RawVec用于指向一块从堆内存申请出来的某一类型数据的数组buffer，可以未初始化或初始化为零。与数组有关的智能指针底层的内存申请基本上都采用了RawVec RawVec的结构体，创建及Drop相关方法：

```
enum AllocInit {
    /// 内存块没有初始化
    Uninitialized,
    /// 内存块被初始化为0
    Zeroed,
}
pub(crate) struct RawVec<T, A: Allocator = Global> {
    //指向堆内存地址
    ptr: Unique<T>,
    //内存块中含有T类型变量的数目
    cap: usize,
    //Allocator 变量
    alloc: A,
```

```rust
}

impl<T> RawVec<T, Global> {
    //语法上的要求，一些const fn 只能调用const fn，所以这里设定了一个const 变量
    pub const NEW: Self = Self::new();

    // 一些创建方法，但仅仅是对其他函数调用，代码略
    pub const fn new() -> Self;
    pub fn with_capacity(capacity: usize) -> Self;
    pub fn with_capacity_zeroed(capacity: usize) -> Self;
}

impl<T, A: Allocator> RawVec<T, A> {
    // 最少申请的容量大小
    const MIN_NON_ZERO_CAP: usize = if mem::size_of::<T>() == 1 {
        8
    } else if mem::size_of::<T>() <= 1024 {
        4
    } else {
        1
    };

    //设置一个内存块大小为0的变量
    pub const fn new_in(alloc: A) -> Self {
        // `cap: 0` means "unallocated". zero-sized types are ignored.
        Self { ptr: Unique::dangling(), cap: 0, alloc }
    }

    //申请给定容量的内存块，内存块未初始化
    pub fn with_capacity_in(capacity: usize, alloc: A) -> Self {
        //见后继说明
        Self::allocate_in(capacity, AllocInit::Uninitialized, alloc)
    }

    //申请给定容量的内存块，内存块初始化为全零
    pub fn with_capacity_zeroed_in(capacity: usize, alloc: A) -> Self {
        Self::allocate_in(capacity, AllocInit::Zeroed, alloc)
    }

    //堆内存申请函数
    fn allocate_in(capacity: usize, init: AllocInit, alloc: A) -> Self {
        //ZST的类型不用申请
        if mem::size_of::<T>() == 0 {
            Self::new_in(alloc)
        } else {
            //获取T类型的Layout,注意是用array类型来获取整个size
            let layout = match Layout::array::<T>(capacity) {
                Ok(layout) => layout,
                Err(_) => capacity_overflow(),
            };
            //看堆内存是否有足够的空间
```

```
            match alloc_guard(layout.size()) {
                Ok(_) => {}
                Err(_) => capacity_overflow(),
            }
            //申请内存返回是NonNull<[u8]>, NonNull<[u8]>包含了长度信息
            let result = match init {
                AllocInit::Uninitialized => alloc.allocate(layout),
                AllocInit::Zeroed => alloc.allocate_zeroed(layout),
            };
            //处理可能的错误
            let ptr = match result {
                Ok(ptr) => ptr,
                Err(_) => handle_alloc_error(layout),
            };

            Self {
                //直接将NonNull<[u8]>转化为NonNull<T>,再转换为 *mut T
                //再生成Unique<T>, 注意*mut T此时没有长度信息
                ptr: unsafe { Unique::new_unchecked(ptr.cast().as_ptr()) },
                //用申请的字节数，ptr不要和上面一行的ptr搞混掉。
                //NonNull<[u8]>附带有长度信息
                cap: Self::capacity_from_bytes(ptr.len()),
                alloc,
            }
        }
    }

    //由元数据直接生成，调用代码需要保证输入参数是正确的
    pub unsafe fn from_raw_parts_in(ptr: *mut T, capacity: usize, alloc: A) ->
Self {
        Self { ptr: unsafe { Unique::new_unchecked(ptr) }, cap: capacity, alloc
}
    }

    //返回与allocator申请到的一致的内存变量
    fn current_memory(&self) -> Option<(NonNull<u8>, Layout)> {
        if mem::size_of::<T>() == 0 || self.cap == 0 {
            None
        } else {
            // We have an allocated chunk of memory, so we can bypass runtime
            // checks to get our current layout.
            unsafe {
                let align = mem::align_of::<T>();
                let size = mem::size_of::<T>() * self.cap;
                let layout = Layout::from_size_align_unchecked(size, align);
                Some((self.ptr.cast().into(), layout))
            }
        }
    }
    ...
}
```

```rust
//may_dangle指明T中在释放的时候有可能会出现悬垂指针，但保证不会对悬垂指针做访问，编译器可以放宽strictly alive的规则，
//PhantomData<T>会针对T类型取消掉may_dangle的作用
unsafe impl<#[may_dangle] T, A: Allocator> Drop for RawVec<T, A> {
    /// .
    fn drop(&mut self) {
        if let Some((ptr, layout)) = self.current_memory() {
            //释放内存
            unsafe { self.alloc.deallocate(ptr, layout) }
        }
    }
}
```

RawVec转换为Box<[T],A>:

```rust
impl<T, A: Allocator> RawVec<T, A> {
    //将内存块中0到len-1之间的内存块，转换为Box<[MaybeUninit<T>]>类型，len应该小于self.capacity,
    //由调用者保证
    pub unsafe fn into_box(self, len: usize) -> Box<[MaybeUninit<T>], A> {
        debug_assert!(
            len <= self.capacity(),
            "`len` must be smaller than or equal to `self.capacity()`"
        );

        //RUST不再对self做drop调用
        let me = ManuallyDrop::new(self);
        unsafe {
            //me作为解引用，获取ptr，然后直接将裸指针强制转换为MaybeUninit<T>,
            //生成slice的可变引用
            let slice = slice::from_raw_parts_mut(me.ptr() as *mut MaybeUninit<T>, len);
            //用Box::from_raw_in生成Box<[MaybeUninit<T>]>，注意这里需要对me.alloc做个拷贝
            //因为me已经被forget，所以不能再用原先的alloc.
            Box::from_raw_in(slice, ptr::read(&me.alloc))
        }
    }
```

RawVec内部成员获取方法:

```rust
    pub fn ptr(&self) -> *mut T {
        self.ptr.as_ptr()
    }

    pub fn capacity(&self) -> usize {
        if mem::size_of::<T>() == 0 { usize::MAX } else { self.cap }
    }
```

```
    pub fn allocator(&self) -> &A {
        &self.alloc
    }
```

RawVec内存空间预留，扩充，收缩相关方法：

```
    //保留空间，确保申请的内存大小满足输入参数的规定，否则的话，扩充内存
    pub fn reserve(&mut self, len: usize, additional: usize) {
        #[cold]
        fn do_reserve_and_handle<T, A: Allocator>(
            slf: &mut RawVec<T, A>,
            len: usize,
            additional: usize,
        ) {
            handle_reserve(slf.grow_amortized(len, additional));
        }

        if self.needs_to_grow(len, additional) {
            do_reserve_and_handle(self, len, additional);
        }
    }

    /// The same as `reserve`, but returns on errors instead of panicking or
aborting.
    pub fn try_reserve(&mut self, len: usize, additional: usize) -> Result<(),
TryReserveError> {
        if self.needs_to_grow(len, additional) {
            self.grow_amortized(len, additional)
        } else {
            Ok(())
        }
    }

    pub fn reserve_exact(&mut self, len: usize, additional: usize) {
        handle_reserve(self.try_reserve_exact(len, additional));
    }

    pub fn try_reserve_exact(
        &mut self,
        len: usize,
        additional: usize,
    ) -> Result<(), TryReserveError> {
        if self.needs_to_grow(len, additional) { self.grow_exact(len,
additional) } else { Ok(()) }
    }

    //收缩空间置给定大小
    pub fn shrink_to_fit(&mut self, amount: usize) {
```

```
        handle_reserve(self.shrink(amount));
    }
}

impl<T, A: Allocator> RawVec<T, A> {
    //判断内存块空间是否足够
    fn needs_to_grow(&self, len: usize, additional: usize) -> bool {
        //wrapping_sub防止溢出
        additional > self.capacity().wrapping_sub(len)
    }

    //从字节数得出内存块数目
    fn capacity_from_bytes(excess: usize) -> usize {
        debug_assert_ne!(mem::size_of::<T>(), 0);
        excess / mem::size_of::<T>()
    }

    //根据NonNull来设置结构体ptr及容量
    fn set_ptr(&mut self, ptr: NonNull<[u8]>) {
        //ptr.cast会转换NonNull<[u8]>到NonNull<T>
        self.ptr = unsafe { Unique::new_unchecked(ptr.cast().as_ptr()) };
        //由字节数获得内存块数目
        self.cap = Self::capacity_from_bytes(ptr.len());
    }

    // 增长到满足len+additional的空间,
    fn grow_amortized(&mut self, len: usize, additional: usize) -> Result<(),
TryReserveError> {
        // This is ensured by the calling contexts.
        debug_assert!(additional > 0);

        if mem::size_of::<T>() == 0 {
            return Err(CapacityOverflow.into());
        }

        // 计算需要的容量值，不能超过usize::MAX.
        let required_cap =
len.checked_add(additional).ok_or(CapacityOverflow)?;

        // 每次以2的指数递增，且不能小于最小内存容量
        //  `cap <= isize::MAX` and the type of `cap` is `usize`.
        let cap = cmp::max(self.cap * 2, required_cap);
        let cap = cmp::max(Self::MIN_NON_ZERO_CAP, cap);

        //重新计算内存大小
        let new_layout = Layout::array::<T>(cap);

        // 见后文.
        let ptr = finish_grow(new_layout, self.current_memory(), &mut
self.alloc)?;
        //更新ptr及cap
```

```
        self.set_ptr(ptr);
        Ok(())
    }

    // 与`grow_amortized`基本一致。只是要正好是len+additional的大小
    fn grow_exact(&mut self, len: usize, additional: usize) -> Result<(),
TryReserveError> {
        if mem::size_of::<T>() == 0 {
            // Since we return a capacity of `usize::MAX` when the type size is
            // 0, getting to here necessarily means the `RawVec` is overfull.
            return Err(CapacityOverflow.into());
        }

        let cap = len.checked_add(additional).ok_or(CapacityOverflow)?;
        let new_layout = Layout::array::<T>(cap);

        // `finish_grow` is non-generic over `T`.
        let ptr = finish_grow(new_layout, self.current_memory(), &mut
self.alloc)?;
        self.set_ptr(ptr);
        Ok(())
    }

    //收缩内存到amount长度
    fn shrink(&mut self, amount: usize) -> Result<(), TryReserveError> {
        assert!(amount <= self.capacity(), "Tried to shrink to a larger
capacity");

        let (ptr, layout) = if let Some(mem) = self.current_memory() { mem }
else { return Ok(()) };
        let new_size = amount * mem::size_of::<T>();

        let ptr = unsafe {
            let new_layout = Layout::from_size_align_unchecked(new_size,
layout.align());
            //利用Allcator的函数完成内存申请，拷贝原有内容，并释放原内存
            self.alloc
                .shrink(ptr, layout, new_layout)
                .map_err(|_| AllocError { layout: new_layout, non_exhaustive:
() })?
        };
        //更换指针和容量，这里虽然更换了self的内容，但没有改变编译器对self的所有权的认
识
        self.set_ptr(ptr);
        Ok(())
    }
}
//内存增长具体实现
fn finish_grow<A>(
    new_layout: Result<Layout, LayoutError>,
    current_memory: Option<(NonNull<u8>, Layout)>,
```

```rust
    alloc: &mut A,
) -> Result<NonNull<[u8]>, TryReserveError>
where
    A: Allocator,
{
    // 检查new_layout是否为错误
    let new_layout = new_layout.map_err(|_| CapacityOverflow)?;
    //确保新的Layout的大小不引发异常
    alloc_guard(new_layout.size())?;

    let memory = if let Some((ptr, old_layout)) = current_memory {
        //原先已经申请过内存
        debug_assert_eq!(old_layout.align(), new_layout.align());
        unsafe
            // The allocator checks for alignment equality
            intrinsics::assume(old_layout.align() == new_layout.align());
            //调用Allocator的grow函数增长内存
            alloc.grow(ptr, old_layout, new_layout)
        }
    } else {
        //原先未申请过内存
        alloc.allocate(new_layout)
    };

    memory.map_err(|_| AllocError { layout: new_layout, non_exhaustive: ()
}.into())
}

fn handle_reserve(result: Result<(), TryReserveError>) {
    match result.map_err(|e| e.kind()) {
        Err(CapacityOverflow) => capacity_overflow(),
        Err(AllocError { layout, .. }) => handle_alloc_error(layout),
        Ok(()) => { /* yay */ }
    }
}

fn alloc_guard(alloc_size: usize) -> Result<(), TryReserveError> {
    if usize::BITS < 64 && alloc_size > isize::MAX as usize {
        Err(CapacityOverflow.into())
    } else {
        Ok(())
    }
}

fn capacity_overflow() -> ! {
    panic!("capacity overflow");
}
```