# Cow写时复制结构解析

与Borrow Trait互为逆的ToOwned trait。 一般满足 T.borrow() 返回 &U， U.to_owned()返回T

```
pub trait ToOwned {
    // 必须实现Borrow<Self> trait,  Owned.borrow()->&Self
    type Owned: Borrow<Self>;

    // 从本类型生成Owned类型，一般由指针生成原始变量
    fn to_owned(&self) -> Self::Owned;

    //替换target的内容，原内容会被drop掉
    fn clone_into(&self, target: &mut Self::Owned) {
        *target = self.to_owned();
    }
}

impl<T> ToOwned for T
//实现了Clone的类型自然实现ToOwned
where
    T: Clone
{
    type Owned = T;
    fn to_owned(&self) -> T {
        //创建一个新的T类型的变量
        self.clone()
    }

    fn clone_into(&self, target: &mut T) {
        target.clone_from(self);
    }
}
```

Cow解决一类复制问题: 在与原有变量没有变化时使用原有变量的引用来访问变量，当发生变化时，完成对变量的复制生成新的变量。

```
pub enum Cow<'a, B: ?Sized + 'a>
where
    B: ToOwned,
{
    /// 用Borrowed封装原有变量的引用
    Borrowed( &'a B),

    ///当需要对原有变量做修改，会对原油变量调用to_owned得到新变量，然后用Owned进行封
```

```
装
    Owned(<B as ToOwned>::Owned),
}
```

Cow的创建一般用`let a = Cow::Borrowed(&T)这种方式直接完成，因为是写时复制，所以需要用Borrowed()来得到初始值，否则不符合语义要求。

典型的trait实现：

```
//解引用，会返回&B
impl<B: ?Sized + ToOwned> const Deref for Cow<'_, B>
where
    B::Owned: ~const Borrow<B>,
{
    type Target = B;

    fn deref(&self) -> &B {
        match *self {
            //如果是原有的变量，则返回原有变量引用
            Borrowed(borrowed) => borrowed,
            //如果值已经被修改，则返回新变量的borrow()
            Owned(ref owned) => owned.borrow(),
        }
    }
}

//实现Borrow Trait
impl<'a, B: ?Sized> Borrow<B> for Cow<'a, B>
where
    B: ToOwned,
    <B as ToOwned>::Owned: 'a,
{
    fn borrow(&self) -> &B {
        //利用deref来返回
        &**self
    }
}

// Clone的实现，需要满足写时复制的要求。
impl<B: ?Sized + ToOwned> Clone for Cow<'_, B> {
    fn clone(&self) -> Self {
        match *self {
            //如果是原变量的引用，因为没有写，所以只需要复制一个引用即可
            Borrowed(b) => Borrowed(b),
            //如果已经对原变量做了复制，那需要再次复制x现有变量。
            //根据已知条件，只能先得到o的borrow()，即一个B的变量，然后调用B的
to_owned获得O的拷贝
            Owned(ref o) => {
                let b: &B = o.borrow();
                Owned(b.to_owned())
```

```
            }
        }
    }

    fn clone_from(&mut self, source: &Self) {
        match (self, source) {
            //仅在双方都为Owned的情况下要先borrow后再复制，注意，此时self的原dest生
命周期终止
            (&mut Owned(ref mut dest), &Owned(ref o)) =>
o.borrow().clone_into(dest),
            (t, s) => *t = s.clone(),
        }
    }
}
```

Cow<'a, T>的一些方法

```
impl<B: ?Sized + ToOwned> Cow<'_, B> {
    pub const fn is_borrowed(&self) -> bool {
        match *self {
            Borrowed(_) => true,
            Owned(_) => false,
        }
    }

    pub const fn is_owned(&self) -> bool {
        !self.is_borrowed()
    }

    //这个函数说明要对变量进行改变，因此，如果还是原变量的引用，则需要做复制操作
    pub fn to_mut(&mut self) -> &mut <B as ToOwned>::Owned {
        match *self {
            Borrowed(borrowed) => {
                //复制操作，复制原变量后，然后用Owned包装
                *self = Owned(borrowed.to_owned());
                match *self {
                    Borrowed(..) => unreachable!(),
                    Owned(ref mut owned) => owned,
                }
            }
            Owned(ref mut owned) => owned,
        }
    }

    //此函数也说明后继要对Cow进行修改，所以先消费Cow
    pub fn into_owned(self) -> <B as ToOwned>::Owned {
        match self {
            Borrowed(borrowed) => borrowed.to_owned(),
            Owned(owned) => owned,
        }
```

```
    }
}

//由slice生成Cow的代码例
impl<'a, T: Clone> From<&'a [T]> for Cow<'a, [T]> {
    fn from(s: &'a [T]) -> Cow<'a, [T]> {
        //先生成Borrowed
        Cow::Borrowed(s)
    }
}
```

从Cow<'a, T>可以看到RUST基础语法的强大能力，大家可以思考一下如何用其他语言来实现这一写时复制的类型，会发现很难实现。

## Vec 分析

动态数组，结构体及创建，析构方法相关：

```
pub struct Vec<T, A: Allocator = Global> {
    //RawVec作为堆内存结构，RawVec的容量可能大于Vec的有效长度
    buf: RawVec<T, A>,
    //Vec中真正的成员数目，一般小于RawVec的容量
    len: usize,
}

macro_rules! vec {
    () => (
        $crate::vec::Vec::new()
    );
    ($elem:expr; $n:expr) => (
        $crate::vec::from_elem($elem, $n)
    );
    ($($x:expr),*) => (
        //首先生成Box<[T;N]>，然后利用slice的into_vec生成`Vec<T>`
        $crate::slice::into_vec(box [$($x),*])
    );
    //这里实际上就是完成($x,)=>$x。去掉了','号。
    ($($x:expr,)*) => (vec![$($x),*])
}

impl<T, A: Allocator> ops::Deref for Vec<T, A> {
    type Target = [T];

    fn deref(&self) -> &[T] {
        //用裸指针类型变换方式形成切片的裸指针，再转换为切片引用
        //返回的&[T]拥有与返回作用域匹配的生命周期
        //这个返回值是有隐含的生命周期类型的，不应长于self的生命周期
        unsafe { slice::from_raw_parts(self.as_ptr(), self.len) }
```

```
    }
}

impl<T, A: Allocator> ops::DerefMut for Vec<T, A> {
    fn deref_mut(&mut self) -> &mut [T] {
        //返回值中有隐含的生命周期类型，不应长于self的生命周期
        unsafe { slice::from_raw_parts_mut(self.as_mut_ptr(), self.len) }
    }
}
//Vec<T>的Index下标实现,实际上就是切片Index实现
impl<T, I: SliceIndex<[T]>, A: Allocator> Index<I> for Vec<T, A> {
    type Output = I::Output;

    fn index(&self, index: I) -> &Self::Output {
        //&**self会将Vec转换为&[T]
        Index::index(&**self, index)
    }
}

impl<T, I: SliceIndex<[T]>, A: Allocator> IndexMut<I> for Vec<T, A> {
    fn index_mut(&mut self, index: I) -> &mut Self::Output {
        IndexMut::index_mut(&mut **self, index)
    }
}

impl<T> Vec<T> {
    pub const fn new() -> Self {
        //初始化buf为空
        Vec { buf: RawVec::NEW, len: 0 }
    }
    //其他创建函数，因为仅仅是对其他函数的封装调用，代码略
    pub fn with_capacity(capacity: usize) -> Self;
    pub unsafe fn from_raw_parts(ptr: *mut T, length: usize, capacity: usize) -
> Self;
}

//由Box转换为Vec，这是RUST的最令人无语的地方，内存安全导致必须对类型做各种其他语言不需要的复杂的变换
pub fn into_vec<T, A: Allocator>(b: Box<[T], A>) -> Vec<T, A> {
    unsafe {
        let len = b.len();
        let (b, alloc) = Box::into_raw_with_allocator(b);
        Vec::from_raw_parts_in(b as *mut T, len, len, alloc)
    }
}

//所有支持SpecFromElem trait的类型可以直接转换生成n个初始值为elem的Vec动态数组
pub fn from_elem_in<T: Clone, A: Allocator>(elem: T, n: usize, alloc: A) ->
Vec<T, A> {
    <T as SpecFromElem>::from_elem(elem, n, alloc)
}
```

```rust
pub(super) trait SpecFromElem: Sized {
    fn from_elem<A: Allocator>(elem: Self, n: usize, alloc: A) -> Vec<Self, A>;
}
//所有实现了Clone的类型自然支持SpecFromElem trait
impl<T: Clone> SpecFromElem for T {
    default fn from_elem<A: Allocator>(elem: Self, n: usize, alloc: A) ->
Vec<Self, A> {
        //见下文分析
        let mut v = Vec::with_capacity_in(n, alloc);
        v.extend_with(n, ExtendElement(elem));
        v
    }
}
//drop方法
unsafe impl<#[may_dangle] T, A: Allocator> Drop for Vec<T, A> {
    fn drop(&mut self) {
        unsafe {
            //这里的drop_in_place调用会引发Vec<T>内部的成员变量自身的drop，所以只
drop有意义的值
            //成员变量有些可能已经被释放过，会出现悬垂指针，所以用may_dangle来通知编
译器
            ptr::drop_in_place(ptr::slice_from_raw_parts_mut(self.as_mut_ptr(),
self.len))
        }
        //会自动调用RawVec的drop释放堆内存
    }
}

impl<T, A: Allocator> Vec<T, A> {
    //对RawVec做初始化，实际是空值
    pub const fn new_in(alloc: A) -> Self {
        Vec { buf: RawVec::new_in(alloc), len: 0 }
    }

    //具体见RawVec的函数说明，这里创建了一个容量为输入参数的RawVec，
    //但Vec本身的长度为0，标示成员都还没有初始化
    pub fn with_capacity_in(capacity: usize, alloc: A) -> Self {
        Vec { buf: RawVec::with_capacity_in(capacity, alloc), len: 0 }
    }

    //利用原始数据生成Vec，调用代码应该保证安全性：
    // *mut T应该是从堆申请的，符合RawVec<T>申请规则的大小和对齐，并且是用alloc来做的
申请
    pub unsafe fn from_raw_parts_in(ptr: *mut T, length: usize, capacity:
usize, alloc: A) -> Self {
        unsafe { Vec { buf: RawVec::from_raw_parts_in(ptr, capacity, alloc),
len: length } }
    }

    //生成原始数据，此处首先要使得self被禁止drop，
```

```rust
    //一般后继利用这些原始数据生成新的RawVec，重新启用新RawVec的Drop
    pub fn into_raw_parts(self) -> (*mut T, usize, usize) {
        let mut me = ManuallyDrop::new(self);
        //me自动解引用后得到Vec
        (me.as_mut_ptr(), me.len(), me.capacity())
    }

    //生成原始数据，包括alloc的变量
    pub fn into_raw_parts_with_alloc(self) -> (*mut T, usize, usize, A) {
        let mut me = ManuallyDrop::new(self);
        let len = me.len();
        let capacity = me.capacity();
        let ptr = me.as_mut_ptr();
        let alloc = unsafe { ptr::read(me.allocator()) };
        (ptr, len, capacity, alloc)
    }

    pub fn capacity(&self) -> usize {
        //RawVec的capacity
        self.buf.capacity()
    }
    pub fn allocator(&self) -> &A {
        self.buf.allocator()
    }

    pub fn len(&self) -> usize {
        self.len
    }
    //极度不安全，最好不要用这个函数改变len
    pub unsafe fn set_len(&mut self, new_len: usize) {
        debug_assert!(new_len <= self.capacity());
        //没有做任何处理就改变了len，可能造成内存泄漏，或者调用未初始化内存造成UB
        self.len = new_len;
    }
    pub fn is_empty(&self) -> bool {
        self.len() == 0
    }
```

Vec容量相关方法：

```rust
    //在当前的len的基础上扩张输入的参数的内存容量
    //不一定会出发对内存的重新申请，因为RawVec的容量可能是够的
    //容量不能超出usize::MAX
    pub fn reserve(&mut self, additional: usize) {
        self.buf.reserve(self.len, additional);
    }

    //精确的扩张容量
    pub fn reserve_exact(&mut self, additional: usize) {
        self.buf.reserve_exact(self.len, additional);
```

```
    }

    //如果reserve不成功，返回错误类型
    pub fn try_reserve(&mut self, additional: usize) -> Result<(),
TryReserveError> {
        self.buf.try_reserve(self.len, additional)
    }

    //精确的容量
    pub fn try_reserve_exact(&mut self, additional: usize) -> Result<(),
TryReserveError> {
        self.buf.try_reserve_exact(self.len, additional)
    }

    //收缩内部buf容量到正好是Vec长度
    pub fn shrink_to_fit(&mut self) {
        if self.capacity() > self.len {
            self.buf.shrink_to_fit(self.len);
        }
    }

    //收缩容量
    pub fn shrink_to(&mut self, min_capacity: usize) {
        if self.capacity() > min_capacity {
            self.buf.shrink_to_fit(cmp::max(self.len, min_capacity));
        }
    }
    //在有变量存在的情况下做收缩
    pub fn truncate(&mut self, len: usize) {
        unsafe {
            if len > self.len {
                return;
            }
            //计算需要删除的容量
            let remaining_len = self.len - len;
            //形成需要删除的部分的切片类型
            let s = ptr::slice_from_raw_parts_mut(self.as_mut_ptr().add(len),
remaining_len);
            //修改Vec的长度。
            self.len = len;
            //调用drop_in_place，使得切片能够对内部的成员调用drop以完成删除
            //注意，此时不涉及Vec内部的buf删除，仅仅是删除Vec的成员
            ptr::drop_in_place(s);
        }
    }
```

将`Vec<T>`转换成其他类型

```
    //转换为Box<[T], A>类型。
    pub fn into_boxed_slice(mut self) -> Box<[T], A> {
```

```
        unsafe {
            //此处重要，进入Box后，堆内存的容量必须是切片长度的内存，否则释放会引发问
题
            self.shrink_to_fit();
            //本Vec的Drop需要禁止，由Box负责内存释放
            let me = ManuallyDrop::new(self);
            //这里将RawVec做了一个拷贝，实际上是将RawVec所有权转移出来，必须的
            //拷贝是效率高的做法
            let buf = ptr::read(&me.buf);
            let len = me.len();
            //用RawVec生成Box
            buf.into_box(len).assume_init()
        }
    }

    pub fn as_slice(&self) -> &[T] {
        //会自动解引用
        self
    }

    pub fn as_mut_slice(&mut self) -> &mut [T] {
        self
    }

    pub fn as_ptr(&self) -> *const T {
        let ptr = self.buf.ptr();
        unsafe {
            assume(!ptr.is_null());
        }
        ptr
    }

    pub fn as_mut_ptr(&mut self) -> *mut T {
        let ptr = self.buf.ptr();
        unsafe {
            assume(!ptr.is_null());
        }
        ptr
    }
```

插入与删除方法：

```
    //在index的位置插入一个变量
    pub fn insert(&mut self, index: usize, element: T) {
        #[cold]
        #[inline(never)]
        fn assert_failed(index: usize, len: usize) -> ! {
            panic!("insertion index (is {}) should be <= len (is {})", index,
len);
        }
```

```
        //如果index大于Len，出错
        let len = self.len();
        if index > len {
            assert_failed(index, len);
        }

        //如果预留的空间不够，则至少扩充1个成员空间
        if len == self.buf.capacity() {
            self.reserve(1);
        }

        unsafe {
            {
                //获得index的成员内存地址
                let p = self.as_mut_ptr().add(index);
                //此处将index之后所有成员内存向后偏移一个地址，最高的效率
                ptr::copy(p, p.offset(1), len - index);
                //将变量写入index的成员地址
                ptr::write(p, element);
            }
            //修改长度
            self.set_len(len + 1);
        }
    }

    pub fn remove(&mut self, index: usize) -> T {
        #[cold]
        #[inline(never)]
        #[track_caller]
        fn assert_failed(index: usize, len: usize) -> ! {
            panic!("removal index (is {}) should be < len (is {})", index,
len);
        }

        //如果index大于Vec的长度，出错
        let len = self.len();
        if index >= len {
            assert_failed(index, len);
        }
        unsafe {
            let ret;
            {
                // 得到index的成员地址
                let ptr = self.as_mut_ptr().add(index);
                // 将成员变量拷贝出来，并转移了所有权
                ret = ptr::read(ptr);

                // 将index+1后的所有成员内存拷贝到前面一个地址
                ptr::copy(ptr.offset(1), ptr, len - index - 1);
            }
            //改变长度
```

```
            self.set_len(len - 1);
            //将删除的变量及所有权返回。
            ret
        }
    }

    //在尾部插入一个元素
    pub fn push(&mut self, value: T) {
        //如果预留空间不够，则扩充一个空间
        if self.len == self.buf.capacity() {
            self.reserve(1);
        }
        unsafe {
            //获取当前尾部成员后面的内存地址
            let end = self.as_mut_ptr().add(self.len);
            //将变量写入内存地址
            ptr::write(end, value);
            //长度加1
            self.len += 1;
        }
    }

    //取出尾部成员
    pub fn pop(&mut self) -> Option<T> {
        if self.len == 0 {
            None
        } else {
            unsafe {
                self.len -= 1;
                //将尾部成员变量读出并连同所有权共同返回，此处因为self.len已经减1，后
继drop时不会再对
                //原尾部成员drop。所以尾部成员的所有权已经被处理掉了
                Some(ptr::read(self.as_ptr().add(self.len())))
            }
        }
    }

    //删除所有成员
    pub fn clear(&mut self) {
        //重用
        self.truncate(0)
    }

    //将另一个Vec加到本身
    pub fn append(&mut self, other: &mut Self) {
        unsafe {
            self.append_elements(other.as_slice() as _);
            //此时other中成员的所有权都已经转移到self
            //直接set_len(0)来完成forget操作
            other.set_len(0);
        }
```

```
    }

    unsafe fn append_elements(&mut self, other: *const [T]) {
        //处理self的长度
        let count = unsafe { (*other).len() };
        self.reserve(count);
        let len = self.len();
        //一次性将other的成员拷贝到self，效率最高的方法，拷贝后所有权均已转移到self
        unsafe { ptr::copy_nonoverlapping(other as *const T,
self.as_mut_ptr().add(len), count) };
        //完成长度改动
        self.len += count;
    }

    //leak方法，此方法后，需要再次将返回值转换到RavVec，否则会内存泄漏
    pub fn leak<'a>(self) -> &'a mut [T]
    where
        A: 'a,
    {
        //本Vec变量不再被drop
        let mut me = ManuallyDrop::new(self);
        //生成可变切片引用，此引用没有后继处理的话会造成内存泄漏
        unsafe { slice::from_raw_parts_mut(me.as_mut_ptr(), me.len) }
    }
    ...
}
```

slice转换为`Vec<T>`:

```
    //从slice转换为Vec的trait定义
    pub trait ConvertVec {
        fn to_vec<A: Allocator>(s: &[Self], alloc: A) -> Vec<Self, A>
        where
            Self: Sized;
    }

    //所有支持Clone的类型都支持slce到Vec的转换
    impl<T: Clone> ConvertVec for T {
        default fn to_vec<A: Allocator>(s: &[Self], alloc: A) -> Vec<Self, A> {
            //用来做如果出现panic的时候，保证vec的正确性
            struct DropGuard<'a, T, A: Allocator> {
                vec: &'a mut Vec<T, A>,
                num_init: usize,
            }
            impl<'a, T, A: Allocator> Drop for DropGuard<'a, T, A> {
                fn drop(&mut self) {
                    unsafe {
                        //非正常退出，vec中只有self.num_init被正常初始化
                        self.vec.set_len(self.num_init);
                    }
```

```
                }
            }
            //创建具备足够空间的Vec<T>变量
            let mut vec = Vec::with_capacity_in(s.len(), alloc);
            let mut guard = DropGuard { vec: &mut vec, num_init: 0 };
            //将vec中没有初始化的以[MaybeUninit<T>]返回
            let slots = guard.vec.spare_capacity_mut();

            //对s做迭代，取出下标及成员
            for (i, b) in s.iter().enumerate().take(slots.len()) {
                //guard中初始化的数目，如果panic，则要drop掉
                guard.num_init = i;
                //对b做clone并写入slots[i]，这里要注意所有权，b.clone()新创建了一个
所有权
                //并将所有权转移到了slots[i]中
                slots[i].write(b.clone());
            }
            //循环结束，guard任务已经完成，drop不应再被调用
            core::mem::forget(guard);
            //设置vec的len
            unsafe {
                vec.set_len(s.len());
            }
            vec
        }
    }

    impl<T: Copy> ConvertVec for T {
        fn to_vec<A: Allocator>(s: &[Self], alloc: A) -> Vec<Self, A> {
            let mut v = Vec::with_capacity_in(s.len(), alloc);
            unsafe {
                //对于支持Copy的变量，直接做块的拷贝效率最高。
                s.as_ptr().copy_to_nonoverlapping(v.as_mut_ptr(), s.len());
                v.set_len(s.len());
            }
            v
        }
    }
```

ToOwned trait 代码：

```
    impl<T: Clone> ToOwned for [T] {
    type Owned = Vec<T>;

    fn to_owned(&self) -> Vec<T> {
        //[T].to_vec的调用，T:Clone，见上面代码
        self.to_vec()
    }

   fn clone_into(&self, target: &mut Vec<T>) {
```

```
        //略，请读者自行分析
    }
}
```

Clone trait实现代码：

```
impl<T: Clone, A: Allocator + Clone> Clone for Vec<T, A> {
    fn clone(&self) -> Self {
        let alloc = self.allocator().clone();
        //实质上是对[T]::to_vec()调用，见上面的代码
        <[T]>::to_vec_in(&**self, alloc)
    }

    fn clone_from(&mut self, other: &Self) {
        //略，会导致太长的篇幅，留给读者自己分析
    }
}
```

## `Vec<T>` Iterator分析

into_iter() 相关结构及代码分析：

```
pub struct IntoIter<T, A: Allocator = Global,> {
    pub(super) buf: NonNull<T>,
    pub(super) phantom: PhantomData<T>,
    pub(super) cap: usize,
    pub(super) alloc: A,
    pub(super) ptr: *const T,
    pub(super) end: *const T,
}

impl<T, A: Allocator> IntoIterator for Vec<T, A> {
    type Item = T;
    type IntoIter = IntoIter<T, A>;

    fn into_iter(self) -> IntoIter<T, A> {
        unsafe {
            //将Vec的所有成员所有权转入IntoIter，Vec自身不再做drop操作
            let mut me = ManuallyDrop::new(self);
            let alloc = ptr::read(me.allocator());
            //以下的处理与slice的Iterator非常类似
            let begin = me.as_mut_ptr();
            let end = if mem::size_of::<T>() == 0 {
                //0长度(ZST)类型处理方式
                arith_offset(begin as *const i8, me.len() as isize) as *const T
            } else {
                begin.add(me.len()) as *const T
```

```
            };
            let cap = me.buf.capacity();
            IntoIter {
                buf: NonNull::new_unchecked(begin),
                phantom: PhantomData,
                cap,
                alloc,
                ptr: begin,
                end,
            }
        }
    }
}

impl<T, A: Allocator> Iterator for IntoIter<T, A> {
    type Item = T;

    //以下与slice的Iterator相关方法非常类似
    fn next(&mut self) -> Option<T> {
        if self.ptr as *const _ == self.end {
            None
        } else if mem::size_of::<T>() == 0 {
            // 这里不同，slice的库是end前移，这里是ptr后移
            self.ptr = unsafe { arith_offset(self.ptr as *const i8, 1) as *mut
T };
            //如果T:Default，应该返回<T as Default>::default。
            //不能返回None，下面的代码应该是最好的表达方式。
            Some(unsafe { mem::zeroed() })
        } else {
            //更改头指针，保证不再访问头指针之前的变量
            let old = self.ptr;
            self.ptr = unsafe { self.ptr.offset(1) };
            //ptr::read将成员变量读到栈中并转移了所有权。
            //原有变量此时已经不会再被访问，IntoIter生命周期
            //终止的时候会释放RawVec堆内存
            Some(unsafe { ptr::read(old) })
        }
    }

    //其他函数略
    ...
}
//其他辅助函数
impl<T, A: Allocator> IntoIter<T, A> {
    //由Iterator生成slice引用
    pub fn as_slice(&self) -> &[T] {
        unsafe { slice::from_raw_parts(self.ptr, self.len()) }
    }
    //生成slice可变引用
    pub fn as_mut_slice(&mut self) -> &mut [T] {
        unsafe { &mut *self.as_raw_mut_slice() }
```

```
    }

    //返回Allocator trait
    pub fn allocator(&self) -> &A {
        &self.alloc
    }

    //返回裸指针
    fn as_raw_mut_slice(&mut self) -> *mut [T] {
        ptr::slice_from_raw_parts_mut(self.ptr as *mut T, self.len())
    }

}

//drop函数
unsafe impl<#[may_dangle] T, A: Allocator> Drop for IntoIter<T, A> {
    fn drop(&mut self) {
        struct DropGuard<'a, T, A: Allocator>(&'a mut IntoIter<T, A>);

        impl<T, A: Allocator> Drop for DropGuard<'_, T, A> {
            fn drop(&mut self) {
                unsafe {
                    //
                    let alloc = ptr::read(&self.0.alloc);
                    // 恢复RawVec
                    let _ = RawVec::from_raw_parts_in(self.0.buf.as_ptr(),
self.0.cap, alloc);
                    // RawVec生命周期终结，堆内存被释放
                }
            }
        }

        let guard = DropGuard(self);

        unsafe {
            //self.ptr之前的成员变量已经被消费,这里是一个slice的drop调用，会递归调用
slice内的所有成员的drop
            ptr::drop_in_place(guard.0.as_raw_mut_slice());
        }
        // guard生命周期终止，drop被调用
    }
}
```

以上代码要尤其注意next()时对成员的所有权处理，以及drop时的处理。 iter();iter_mut()相关的代码分析略。

Iterator与Vec的转换： 从一个Iterator增加成员：

```rust
impl<T, A: Allocator> Extend<T> for Vec<T, A> {
    fn extend<I: IntoIterator<Item = T>>(&mut self, iter: I) {
        //见下面的分析
        <Self as SpecExtend<T, I::IntoIter>>::spec_extend(self,
iter.into_iter())
    }

    fn extend_one(&mut self, item: T) {
        self.push(item);
    }

    fn extend_reserve(&mut self, additional: usize) {
        self.reserve(additional);
    }
}
pub(super) trait SpecExtend<T, I> {
    fn spec_extend(&mut self, iter: I);
}

impl<T, A: Allocator> SpecExtend<T, IntoIter<T>> for Vec<T, A> {
    fn spec_extend(&mut self, mut iterator: IntoIter<T>) {
        unsafe {
            self.append_elements(iterator.as_slice() as _);
        }
        //以下方式是最快的清理方式。
        iterator.ptr = iterator.end;
    }
}
impl<T: Clone, A: Allocator> Vec<T, A> {
    pub fn extend_from_slice(&mut self, other: &[T]) {
        self.spec_extend(other.iter())
    }
    ...
}
```

`Vec<T>`整体上具备极高的效率，的所有难于理解的内容点基本都是各种类型转换时的所有权相关处理。这也是所有的RUST的数据结构基础类型实现上相对比其他语言需要额外理解的复杂之处。 对`Vec<T>`的代码熟悉是提高所有权认识的有效途径。