RUST的Iterator实现代码分析

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

Iterator在函数式编程中是居于最核心的地位。在函数式编程中，最关键的就是把问题的解决方式设计成能够使用Iterator方案来解决。RUST基本上可以说是原生的Iterator语言，几乎所有的核心关键复合类型都对Iterator作出实现。

RUST的Iterator与其他语言Iterator比较

RUST定义了三种迭代器:

1. 对变量本身进行遍历的的into_iter，需要实现如下trait：

```
pub trait IntoIterator {
    type Item;
    type IntoIter: Iterator<Item = Self::Item>;
    fn into_iter(self) -> Self::IntoIter;
}
```

into_iter返回的迭代器迭代时，会消费变量及容器，完全迭代后容器将不再存在。

2. 对变量不可变引用进行遍历的iter,类型一般实现以下方法获得此迭代器:

`pub fn iter(&self) -> I:Iterator` 这种迭代器在程序中经常使用，例如，遍历游戏玩家的列表以进行统计

3. 对变量的可变引用进行遍历的iter_mut,类型一般实现以下方法获得此迭代器：

`pub fn iter_mut(&self) -> I:Iterator` 这种迭代器的一个例子是遍历游戏玩家，更新玩家在线时间。

其他语言一般仅实现第3种迭代器。

对变量本身遍历的迭代器是RUST独有的所有权和drop机制带来的一种迭代器。在适合的场景下会缩减代码量及提高效率。

一般的，RUST要求额外实现下面的两种机制
T::iter() 等同于 &T::into_iter()
T::iter_mut() 等同于 &mut T::into_iter()

Iterator Trait 定义

```
pub trait Iterator {
    /// 每次迭代时返回的变量类型.
    type Item;

    //灵魂方法
    fn next(&mut self) -> Option<Self::Item>;

    //size_hint返回值是此迭代器最少产生多少个有效迭代输出，最多产生多少有有效迭代输出。
    //所以，诸如(0..10).int_iter()，最少是10个，最多也是10个，
    //而 (0..10).filter(|x| x%2 == 0)，因为编译器不会提前计算，所以符合条件的最少可能是0个，最多是10个
    fn size_hint(&self) -> (usize, Option<usize>) {
        (0, None)
    }

    //将Iterator中所有的成员做累积操作
    //init作为f的初始值输入，
    fn fold<B, F>(mut self, init: B, mut f: F) -> B
    where
        Self: Sized,
        F: FnMut(B, Self::Item) -> B,
    {
        let mut accum = init;
        while let Some(x) = self.next() {
            accum = f(accum, x);
        }
        accum
    }

    //其他方法
    ...
    ...
}

//在定义一个trait后，
//要考虑针对已经实现这种trait的
//类型的引用/可变引用/切片/数组
//能否用adapter的方式实现该trait
//下面是Iterator的一个例子
impl<I: Iterator + ?Sized> Iterator for &mut I {
    type Item = I::Item;
    fn next(&mut self) -> Option<I::Item> {
        (**self).next()
    }
    fn size_hint(&self) -> (usize, Option<usize>) {
        (**self).size_hint()
    }
```

```
    fn advance_by(&mut self, n: usize) -> Result<(), usize> {
        (**self).advance_by(n)
    }
    fn nth(&mut self, n: usize) -> Option<Self::Item> {
        (**self).nth(n)
    }
}
```

Iterator与其他集合类型转换结构与分析

RUST提供了*集合类型*与Iterator互相转换的trait：

```
//从Iterator创建集合
pub trait FromIterator<A>: Sized {
    //从Iterator中创建集合
    fn from_iter<T: IntoIterator<Item = A>>(iter: T) -> Self;
}

//对实现Iterator trait的集合类型实现IntoIterator
impl<I: Iterator> IntoIterator for I {
    type Item = I::Item;
    type IntoIter = I;

    fn into_iter(self) -> I {
        self
    }
}

//此trait用于从一个Iterator给集合扩充成员
pub trait Extend<A> {
    //将Iterator的成员增加到集合
    fn extend<T: IntoIterator<Item = A>>(&mut self, iter: T);

    /// 仅增加一个成员
    fn extend_one(&mut self, item: A) {
        //Option实现了Iterator
        self.extend(Some(item));
    }

    //扩充容量以备后用
    fn extend_reserve(&mut self, additional: usize) {
        let _ = additional;
    }
}
```

Iterator中的转换方法：

```
pub trait Iterator {
    ...
    ...

    fn collect<B: FromIterator<Self::Item>>(self) -> B
    where
        Self: Sized,
    {
        FromIterator::from_iter(self)
    }
    ...
}
```

以上说明，对于任意的集合类型，只要实现了FromIterator trait，即可通过collect生成。从而使得不同集合类型之间的转换变得统一，方便及松耦合。

ops::Range类型的Iterator实现

代码路径：

%USER%.rustup\toolchains\nightly-x86_64-pc-windows-msvc\lib\rustlib\src\rust\library\core\src\iter\range.rs

Range被直接实现Iterator trait，没有用其他辅助结构。 定义如下：

```
impl<A: Step> Iterator for ops::Range<A> {
    type Item = A;

    fn next(&mut self) -> Option<A> {
        self.spec_next()
    }

    fn size_hint(&self) -> (usize, Option<usize>) {
        if self.start < self.end {
            let hint = Step::steps_between(&self.start, &self.end);
            (hint.unwrap_or(usize::MAX), hint)
        } else {
            (0, Some(0))
        }
    }

    fn nth(&mut self, n: usize) -> Option<A> {
        self.spec_nth(n)
    }
    ...
    ...
```

```
}
```

Range Iterator的具体实现RangeIteratorImpl trait

```
impl<A: Step> RangeIteratorImpl for ops::Range<A> {
    type Item = A;

    default fn spec_next(&mut self) -> Option<A> {
        if self.start < self.end {
            //self.start.clone()是为了不转移self.start的所有权
            let n =
                Step::forward_checked(self.start.clone(), 1).expect("`Step`
invariants not upheld");
            //mem::replace将self.start赋值为n，返回self.start的值，这个方式适用于任
何类型，且处理了所有权问题
            //mem::replace是效率最高的代码方式
            Some(mem::replace(&mut self.start, n))
        } else {
            None
        }
    }

    ...
}
```

由上面的代码可以看出，每一次next实际都对Range本身做出了修改，这一修改是使用mem::replace实现的。要理解这是为什么。

只有基于实现 `Step Trait` 的类型的Range才支持了Iterator, 而代码关键是Step Trait的方法， `Step Trait` 的定义如下：

```
pub trait Step: Clone + PartialOrd + Sized {
    /// 从start 到end一共多少step
    fn steps_between(start: &Self, end: &Self) -> Option<usize>;

    /// 向前count步返回值
    fn forward_checked(start: Self, count: usize) -> Option<Self>;

    /// 向前count步 返回值，出错退出
    fn forward(start: Self, count: usize) -> Self {
        Step::forward_checked(start, count).expect("overflow in
`Step::forward`")
    }

    /// 向前不检查 count步
    unsafe fn forward_unchecked(start: Self, count: usize) -> Self {
        Step::forward(start, count)
```

```
    }

    /// 向后count步
    fn backward_checked(start: Self, count: usize) -> Option<Self>;

    /// 向后count步，出错退出
    fn backward(start: Self, count: usize) -> Self {
        Step::backward_checked(start, count).expect("overflow in
`Step::backward`")
    }

    /// 向后count步，出错退出
    unsafe fn backward_unchecked(start: Self, count: usize) -> Self {
        Step::backward(start, count)
    }
}
```

照此，可以实现一个自定义类型的类型, 并支持Step Trait，如此，即可使用Range符号的Iterator。例如，一个二维的点的range,例如Range<(i32, i32)>的变量((0,0)...(10,10)), 三维的点的range，数列等。

一下是为所有整数类型实现Step的宏：

```
macro_rules! step_identical_methods {
    () => {
        unsafe fn forward_unchecked(start: Self, n: usize) -> Self {
            // 调用代码需要保证加法不会越界.
            unsafe { start.unchecked_add(n as Self) }
        }

        unsafe fn backward_unchecked(start: Self, n: usize) -> Self {
            // 调用代码需要保证减法不会越界.
            unsafe { start.unchecked_sub(n as Self) }
        }

        fn forward(start: Self, n: usize) -> Self {
            // debug 编译情况下 以下代码对溢出会panic，release以下代码会被优化掉
            if Self::forward_checked(start, n).is_none() {
                let _ = Self::MAX + 1;
            }
            // release 编译采用的加法
            start.wrapping_add(n as Self)
        }

        fn backward(start: Self, n: usize) -> Self {
            // debug编译，以下代码在debug目标对溢出会panic，release会被优化掉.
            if Self::backward_checked(start, n).is_none() {
                let _ = Self::MIN - 1;
```

```
            }
            // release编译采用的加法
            start.wrapping_sub(n as Self)
        }
    };
}

macro_rules! step_integer_impls {
    {
        //比CPU字长小的无符号整数类型及有符号整数类型
        narrower than or same width as usize:
            $( [ $u_narrower:ident $i_narrower:ident ] ),+;
        //比CPU字长大的无符号整数类型及有符号整数类型
        wider than usize:
            $( [ $u_wider:ident $i_wider:ident ] ),+;
    } => {
        $(
            //为所有比CPU字长小的无符号整数类型的Step实现
            impl Step for $u_narrower {
                //通用实现
                step_identical_methods!();

                fn steps_between(start: &Self, end: &Self) -> Option<usize> {
                    if *start <= *end {
                        // u_nrrower类型字长必须小于usize字长
                        Some((*end - *start) as usize)
                    } else {
                        None
                    }
                }

                fn forward_checked(start: Self, n: usize) -> Option<Self> {
                    //将类型转换可能不成功显化，这是需要养成的RUST的特有思维
                    match Self::try_from(n) {
                        //checked_add完成溢出检查
                        Ok(n) => start.checked_add(n),
                        Err(_) => None,
                    }
                }

                fn backward_checked(start: Self, n: usize) -> Option<Self> {
                    match Self::try_from(n) {
                        Ok(n) => start.checked_sub(n),
                        Err(_) => None, // if n is out of range,
`unsigned_start - n` is too
                    }
                }
            }

            //略
            ...
```

```
    }
}
```

Range实现Iterator的代码不复杂，但是从类型转换及加减法的处理上深刻的体现了RUST的安全理念。

slice的Iterator实现

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

首先定义了适合&[T]的Iter结构：

```
pub struct Iter<'a, T: 'a> {
    //当前元素的指针，与end用不同的类型表示
    ptr: NonNull<T>,
    //尾元素指针，用ptr == end以快速检测iterator是否为空
    end: *const T,
    //这里PhantomData 主要用来做生命周期标识，用来做Iter结构体与切片之间的生命周期关系检测
    _marker: PhantomData<&'a T>,
}

pub struct IterMut<'a, T: 'a> {
    ptr: NonNull<T>,
    end: *mut T,
    _marker: PhantomData<&'a mut T>,
}
```

这里，一个疑惑就是为什么不用下标及切片长度来作为Iter结构。这是因为可变的Iterator实现无法支持。 例如，给出如下结构：

```
pub struct IterMut <'a, T:'a> {
    current: usize,
    len: usize,
    slice: 'a mut &[T]
}
```

显然，当IterMut结构是可变借用时，无法再返回一个内部成员的借用用作迭代器的迭代返回值。

```
impl<'a, T> IterMut<'a, T> {
    pub(super) fn new(slice: &'a mut [T]) -> Self {
        let ptr = slice.as_mut_ptr();
```

```
        unsafe {
            assume(!ptr.is_null());

            let end = if mem::size_of::<T>() == 0 {
                (ptr as *mut u8).wrapping_add(slice.len()) as *mut T
            } else {
                ptr.add(slice.len())
            };

            Self { ptr: NonNull::new_unchecked(ptr), end, _marker: PhantomData
}
        }
    }

    ...
    ...
}
//用宏来实现切片的Iterator trait
iterator! {struct IterMut -> *mut T, &'a mut T, mut, {mut}, {}}

//上面的宏定义
macro_rules! iterator {
    (
        struct $name:ident -> $ptr:ty,
        $elem:ty,
        $raw_mut:tt,
        {$( $mut_:tt )?},
        {$($extra:tt)*}
    ) => {
        // 正向next函数辅助宏，实际的逻辑见post_inc_start函数
        macro_rules! next_unchecked {
            ($self: ident) => {& $( $mut_ )? *$self.post_inc_start(1)}
        }

        // 反向的next函数
        macro_rules! next_back_unchecked {
            ($self: ident) => {& $( $mut_ )? *$self.pre_dec_end(1)}
        }

        // 0长度元素next的移动
        macro_rules! zst_shrink {
            ($self: ident, $n: ident) => {
                //0元素数组因为不能移动指针，所以移动尾指针
                $self.end = ($self.end as * $raw_mut u8).wrapping_offset(-$n)
as * $raw_mut T;
            }
        }

        //具体的方法实现
        // $name 即 IterMut
        impl<'a, T> $name<'a, T> {
```

```
            // 从Iterator获得切片.
            fn make_slice(&self) -> &'a [T] {
                // Iter::ptr::as_ptr，由内存首地址和切片长度创建切片指针，然后转换为
引用
                unsafe { from_raw_parts(self.ptr.as_ptr(), len!(self)) }
            }

            //实质的next
            unsafe fn post_inc_start(&mut self, offset: isize) -> * $raw_mut T
{
                if mem::size_of::<T>() == 0 {
                    //0字节元素偏移实现，调整end的值，ptr不变
                    zst_shrink!(self, offset);
                    self.ptr.as_ptr()
                } else {
                    //非0字节元素，返回首地址，然后后移正确的字节
                    let old = self.ptr.as_ptr();
                    self.ptr = unsafe {
NonNull::new_unchecked(self.ptr.as_ptr().offset(offset)) };
                    old
                }
            }

            // 从尾部做Iterator的实际实现函数
            unsafe fn pre_dec_end(&mut self, offset: isize) -> * $raw_mut T {
                if mem::size_of::<T>() == 0 {
                    //对于0字节元素，从头部及从尾部逻辑相同
                    zst_shrink!(self, offset);
                    self.ptr.as_ptr()
                } else {
                    //尾部的end即偏移后的位置。
                    self.end = unsafe { self.end.offset(-offset) };
                    self.end
                }
            }
        }

        //Iterator的实现，即
        //impl<'a, T> Iterator for IterMut<'a, T>
        impl<'a, T> Iterator for $name<'a, T> {
            // $elem即&'a T
            type Item = $elem;

            fn next(&mut self) -> Option<$elem> {
                unsafe {
                    //安全性确认
                    assume(!self.ptr.as_ptr().is_null());
                    if mem::size_of::<T>() != 0 {
                        assume(!self.end.is_null());
                    }
                    if is_empty!(self) {
```

```
                    //Iter为空的话，返回None
                    None
                } else {
                    //实际调用post_inc_start(1)
                    Some(next_unchecked!(self))
                }
            }
        }

        fn size_hint(&self) -> (usize, Option<usize>) {
            //用len!宏计算Iter的长度
            let exact = len!(self);
            (exact, Some(exact))
        }

        fn count(self) -> usize {
            len!(self)
        }

        fn nth(&mut self, n: usize) -> Option<$elem> {
            //如果n大于Iter的长度，清空
            if n >= len!(self) {
                if mem::size_of::<T>() == 0 {
                    self.end = self.ptr.as_ptr();
                } else {
                    unsafe {
                        self.ptr = NonNull::new_unchecked(self.end as *mut
T);

                    }
                }
                return None;
            }
            // 否则，失效前n-1个元素，然后做next
            unsafe {
                self.post_inc_start(n as isize);
                Some(next_unchecked!(self))
            }
        }

        fn advance_by(&mut self, n: usize) -> Result<(), usize> {
            //取长度与n中的小值
            let advance = cmp::min(len!(self), n);

            //失效advance-1个值
            unsafe { self.post_inc_start(advance as isize) };
            //返回
            if advance == n { Ok(()) } else { Err(advance) }
        }

        //从尾部Iterator
        fn last(mut self) -> Option<$elem> {
```

```
                //实质调用post_dec_end(1)
                self.next_back()
            }

            //其他，略
            ...
            ...

        }
    }
}

//判断Iterator是否为空的宏
macro_rules! is_empty {
    // 可以满足0字节元素的切片及非0字节元素的切片
    ($self: ident) => {
        //Iter::ptr == Iter::end
        $self.ptr.as_ptr() as *const T == $self.end
    };
}

//取Iterator长度的宏
macro_rules! len {
    ($self: ident) => {{
        let start = $self.ptr;
        let size = size_from_ptr(start.as_ptr());
        //判断元素是否为0字节
        if size == 0 {
            // 用end减start得到0字节元素的切片长度
            ($self.end as usize).wrapping_sub(start.as_ptr() as usize)
        } else {
            //非0字节，用内存字节数除以单元素长度
            let diff = unsafe { unchecked_sub($self.end as usize,
start.as_ptr() as usize) };
            unsafe { exact_div(diff, size) }
        }
    }};
}
```

对于切片，RUST的所有权，借用等规定导致其迭代器实际上是一个非常好的编码训练工具，代码粗略看一遍后值得自己将其实现一遍，可以有效提高对RUST的认识和编码水平。

字符串Iterator代码分析

字符串&str本质上是一个[u8]类型，并在此类型的基础上实现了对utf-8的处理。 因此，对字符串的Iterator的设计自然想到用适配器的模式来重用[u8]切片类型的Iterator的基础设施。

题外话，&str.len()返回字符串切片字节占用数，&str.chars().count()返回字符数目。 字符串切片获取Iterator有如下3个函数 &str::chars() 获得以UTF-8编码的字符串的Iterator &str::bytes() 获得一个[u8]的Iterator &str::char_indices() 获得一个元组，第一个成员是字符字节数组的序号，第二个成员是字符本身 bytes()主要用于提高在程序员确定采用ASCII字符串下的运行效率。 我们以&str::chars()的Iterator来看一下具体的实现

```
pub struct Chars<'a> {
    //利用slice通用的iter做实例化,实际是一个adapter设计模式
    pub(super) iter: slice::Iter<'a, u8>,
}

pub fn chars(&self) -> Chars<'_> {
    //self.as_bytes()获得一个&[u8]
    Chars { iter: self.as_bytes().iter() }
}
impl<'a> Iterator for Chars<'a> {
    type Item = char;

    fn next(&mut self) -> Option<char> {
        //next_code_point见后面代码分析
        next_code_point(&mut self.iter).map(|ch| {
            unsafe { char::from_u32_unchecked(ch) }
        })
    }

    fn count(self) -> usize {
        // 利用切片iterator的filter来实现
        self.iter.filter(|&&byte| !utf8_is_cont_byte(byte)).count()
    }

    fn size_hint(&self) -> (usize, Option<usize>) {
        let len = self.iter.len();
        //最少按四个字节一个字符，最多按一个字节一个字符
        ((len + 3) / 4, Some(len))
    }

    fn last(mut self) -> Option<char> {
        self.next_back()
    }

}

pub fn next_code_point<'a, I: Iterator<Item = &'a u8>>(bytes: &mut I) ->
Option<u32> {
    // iterator.next
    let x = *bytes.next()?;
    if x < 128 {
        //ascii字符
        return Some(x as u32);
    }
```

```
    //因为是字符串，此时第二个字节一定会有
    let init = utf8_first_byte(x, 2);
    //获取下一个字节，一定存在
    let y = unwrap_or_0(bytes.next());
    let mut ch = utf8_acc_cont_byte(init, y);
    if x >= 0xE0 {
        // 三个字节的UTF-8
        let z = unwrap_or_0(bytes.next());
        let y_z = utf8_acc_cont_byte((y & CONT_MASK) as u32, z);
        ch = init << 12 | y_z;
        if x >= 0xF0 {
            //四个字节的UTF-8
            let w = unwrap_or_0(bytes.next());
            ch = (init & 7) << 18 | utf8_acc_cont_byte(y_z, w);
        }
    }

    Some(ch)
}
```

&str的Iterator实现是一个说明Iterator设计模式优越性的经典实例。如果直接使用循环，则&str与&[T]必然会有很多的重复代码，使用Iterator模式后，重复代码被抽象到了Iterator模块中。&str复用了&[T]的iter。

array 的Iterator实现

Unsize Trait

```
pub trait Unsize<T: ?Sized> {
    // Empty.
}
```

实现了Unsize Trait，可以把一个固定内存大小的变量强制转换为相关的可变大小类型， 如[T;N]实现了Unsize<[T]>, 因此[T;N]可以转换为[T]，一般是指针转换。

Iter所用的结构

```
pub struct IntoIter<T, const N: usize> {
    /// data是迭代中的数组.
    /// 这个数组中，只有data[alive]是有效的，访问其他的部分，即data[..alive.start]及data[end..]会发生UB
    /// [MaybeUninit<T>;N]的用法需要体会，
    data: [MaybeUninit<T>; N],

    /// 表明数组中有效的成员的下标范围.
```

```
    /// 必须满足:
    /// - `alive.start <= alive.end`
    /// - `alive.end <= N`
    alive: Range<usize>,
}
```

上面这个结构是因为需要对array内成员做消费设计的。因为数组成员不支持所有权转移，所以采用了这种设计方式。数组的Iterator实现是理解所有权的一个极佳例子。

into_iter实现

```
impl<T, const N: usize> IntoIter<T, N> {
    pub fn new(array: [T; N]) -> Self {
        //
        // 因为RUST特性目前还不支持数组的transmute，所以用了内存跨类型的
transmute_copy，此函数将从栈中申请一块内存。
        // 拷贝完毕后，原数组的所有权已经转移到data，data内数据事实上已经初始化，但仍
然还是MaybeUninit<T>的类型。此时，需要对原数组调用mem::forget反应所有权已经失去。
        // mem::forget不会导致内存泄漏。
        unsafe {
            let iter = Self { data: mem::transmute_copy(&array), alive: 0..N };
            mem::forget(array);
            iter
        }
    }

    pub fn as_slice(&self) -> &[T] {
        // 仅针对有效的部分返回切片引用。已经消费的不返回。
        unsafe {
            //此处调用SliceIndex::<Range>::get_unchecked
            //slice是&[MaybeUninit<T>]类型
            let slice = self.data.get_unchecked(self.alive.clone());
            MaybeUninit::slice_assume_init_ref(slice)
        }
    }

    pub fn as_mut_slice(&mut self) -> &mut [T] {
        unsafe {
            //此处调用SliceIndex::<Range>::get_unchecked_mut
            //slice 是 & mut [MaybeUninit<T>]类型
            let slice = self.data.get_unchecked_mut(self.alive.clone());
            MaybeUninit::slice_assume_init_mut(slice)
        }
    }
}

impl<T, const N: usize> Iterator for IntoIter<T, N> {
    type Item = T;
    fn next(&mut self) -> Option<Self::Item> {
```

```
        // 下面使用Range的Iterator特性实现next. alive的start会变化，从而导致start之前的数组元素无法再被访问。因为已经被消费掉。
        // Option::map完成下标值传递。
        self.alive.next().map(|idx| {
            // SliceIndex::<usize>::get_unchecked, MaybeUninit::<T>::assume_init_read()
            // 前面有过说明，assume_init_read()从堆栈中申请了T大小的内存，然后进行内存拷贝，然后返回变量
            // 此时array元素的所有权转移到返回值。
            unsafe { self.data.get_unchecked(idx).assume_init_read() }
        })
    }
    ...
    ...
}

impl<T, const N: usize> Drop for IntoIter<T, N> {
    // 这里没有被消费掉的成员必须显示释放掉。
    fn drop(&mut self) {
        // as_mut_slice()获得所有具有所有权的元素，这些元素需要调用drop来释放。这里，data变量中的元素始终封装在MaybeUninit<T>中
        unsafe { ptr::drop_in_place(self.as_mut_slice()) }
    }
}
```

数组的Iterator最关键的点就是如何将数组成员的所有权取出，这是RUST语法带来的额外的麻烦和复杂性。最终的解决办法显示了RUST编码的所有权转移的一些通用的底层技巧。

```
impl<T, const N: usize> IntoIterator for [T; N] {
    type Item = T;
    type IntoIter = IntoIter<T, N>;

    /// 创建消费型的iterator，如果T不实现`Copy`，则调用此函数后，数组不可再被访问。
    fn into_iter(self) -> Self::IntoIter {
        IntoIter::new(self)
    }
}
```

以上创建消费数组成员的Iterator。

iter(), iter_mut()实现

下面的数组成员引用的Iterator实质上是将数组强制转换为切片类型，应用切片类型的迭代器。

```rust
impl<'a, T, const N: usize> IntoIterator for &'a [T; N] {
    type Item = &'a T;
    type IntoIter = Iter<'a, T>;

    fn into_iter(self) -> Iter<'a, T> {
        //点号导致self强制转换成[T], 然后调用切片类型的iter
        self.iter()
    }
}

impl<'a, T, const N: usize> IntoIterator for &'a mut [T; N] {
    type Item = &'a mut T;
    type IntoIter = IterMut<'a, T>;


    fn into_iter(self) -> IterMut<'a, T> {
        //self被强制转换为切片类型
        self.iter_mut()
    }
}
```

Iterator的适配器代码分析

Map 适配器代码分析

Map相关代码如下:

```rust

pub trait Iterator {
    //其他内容
    ...
    ...

    //创建map Iterator
    fn map<B, F>(self, f: F) -> Map<Self, F>
    where
        Self: Sized,
        F: FnMut(Self::Item) -> B,
    {
        Map::new(self, f)
    }
    ...
}

//此结构是一个adapter的结构
pub struct Map<I, F> {
    // Map的底层Iterator
```

```
    pub(crate) iter: I,
    // Map操作闭包函数
    f: F,
}

impl<I, F> Map<I, F> {
    //由Iterator::map 函数和这个函数可以理解Iterator的lazy特性，
    //Iterator的创建实际上仅仅建立了数据结构，直到next才有操作。
    pub(in crate::iter) fn new(iter: I, f: F) -> Map<I, F> {
        Map { iter, f }
    }
}
```

Map适配器结构相当直接而简单。

```
//针对Map实现Iterator
impl<B, I: Iterator, F> Iterator for Map<I, F>
where
    F: FnMut(I::Item) -> B,
{
    type Item = B;

    fn next(&mut self) -> Option<B> {
        //利用底层Iterator的next，Option::map实现next
        self.iter.next().map(&mut self.f)
    }

    fn size_hint(&self) -> (usize, Option<usize>) {
        self.iter.size_hint()
    }

    //其他函数，其实现技巧与next类似
    ...
    ...
}
```

Chain 适配器代码分析

相关代码如下：

```
pub trait Iterator {
    ...
    ...
    //创建Chain Iterator
    fn chain<U>(self, other: U) -> Chain<Self, U::IntoIter>
    where
        Self: Sized,
        U: IntoIterator<Item = Self::Item>,
```

```
    {
        Chain::new(self, other.into_iter())
    }
    ...
    ...
}


pub struct Chain<A, B> {
    //迭代器A
    a: Option<A>,
    //迭代器B
    b: Option<B>,
}
impl<A, B> Chain<A, B> {
    pub(in super::super) fn new(a: A, b: B) -> Chain<A, B> {
        Chain { a: Some(a), b: Some(b) }
    }
}

macro_rules! fuse {
    ($self:ident . $iter:ident . $($call:tt)+) => {
        //$iter可能已经被置为None
        match $self.$iter {
            //若$iter不为None,则调用iter的系列函数
            Some(ref mut iter) => match iter.$($call)+ {
                //函数返回None
                None => {
                    //设置$iter为None,并返回None
                    $self.$iter = None;
                    None
                }
                //其他返回函数返回值
                item => item,
            },
            //a为None时返回None
            None => None,
        }
    };
}

//与fuse类似，略
macro_rules! maybe {
    ($self:ident . $iter:ident . $($call:tt)+) => {
        match $self.$iter {
            Some(ref mut iter) => iter.$($call)+,
            None => None,
        }
    };
}
```

```rust
impl<A, B> Iterator for Chain<A, B>
where
    A: Iterator,
    B: Iterator<Item = A::Item>,
{
    type Item = A::Item;

    fn next(&mut self) -> Option<A::Item> {
        //先执行self.a.next
        match fuse!(self.a.next()) {
            //若self.a.next返回None，则执行self.b.next
            None => maybe!(self.b.next()),
            //不为None，返回a的返回值
            item => item,
        }
    }
    ...
    ...
}
```

其他

Iterator的adapter还有很多，如StepBy, Filter, Zip, Intersperse等等。具体请参考标准库手册。基本上所有的adapter都是遵循Adapter的设计模式来实现的。且每一个适配器的结构及代码逻辑都是比较简单且易理解的。

小结

RUST的Iterater的adapter是突出的体现RUST的语法优越性的特性，借助Trait和强大的泛型机制，与c/c++/java相比较，RUST以很少的代码在标准库就实现了最丰富的adapter。而其他语言标准库往往不存在这些适配器，需要其他库来实现。 Iterator的adapter实现了强大的基于Iterator的函数式编程基础设施。函数式编程的基础框架之一便是基于Iterator和闭包实现丰富的adapter。这也凸显了RUST在语言级别对函数式编程的良好支持。

Option的Iterator实现代码分析

Option实现Iterator是比较令人疑惑的，毕竟用Iterator肯定代码更多，逻辑也复杂。主要目的应该是为了重用Iterator构建的各种adapter，及为了函数式编程的需要。仅分析IntoIterator Trait所涉及的结构及方法 相关类型结构定义：

```rust
//into_iter的结构
pub struct IntoIter<A> {
    //实际的Iterator实现结构
    inner: Item<A>,
}
```

```rust
//Item同时满足into_iter(), iter(), iter_mut()
//标准库编码者的设计方式，当然也可以用其他设计
struct Item<A> {
    opt: Option<A>,
}

impl<T> IntoIterator for Option<T> {
    type Item = T;
    type IntoIter = IntoIter<T>;

    //创建Iterator的实现结构体，self所有权传入结构体
    fn into_iter(self) -> IntoIter<T> {
        IntoIter { inner: Item { opt: self } }
    }
}

//具体实现者
impl<A> Iterator for Item<A> {
    type Item = A;

    fn next(&mut self) -> Option<A> {
        //所有权传出，并用None替换原变量的值
        self.opt.take()
    }

    fn size_hint(&self) -> (usize, Option<usize>) {
        match self.opt {
            Some(_) => (1, Some(1)),
            None => (0, Some(0)),
        }
    }
}

//消费变量的Iterator实现
impl<A> Iterator for IntoIter<A> {
    type Item = A;

    fn next(&mut self) -> Option<A> {
        self.inner.next()
    }

    fn size_hint(&self) -> (usize, Option<usize>) {
        self.inner.size_hint()
    }
}
```

Result<T,E>的 Iterator与Option的Iterator非常相似，略