RUST基本类型代码分析(二)

整形类型标准库代码分析

NonZero数据类型

`NonZeroU8, NonZeroU16, NonZeroU32, NonZeroU64, NonZeroU128, NonZeroUsize`
`NonZeroI8, NonZeroI16, NonZeroI32, NonZeroI64, NonZeroI128, NonZeroIsize` 以上为NonZero的类型，内存结构与相应的整形数据完全相同，可以转换。上文提过，当需要0表示特殊含义时，使用NonZero类型以保证代码安全。 重要函数：

```
//利用宏简化定义代码
macro_rules! nonzero_integers {
    ( $($Ty: ident($Int: ty); )+ ) => {
        $(
            #[derive(Copy, Clone, Eq, PartialEq, Ord, PartialOrd, Hash)]
            #[repr(transparent)]
            pub struct $Ty($Int);

            impl $Ty {
                pub const unsafe fn new_unchecked(n: $Int) -> Self {
                    unsafe { Self(n) }
                }

                pub const fn new(n: $Int) -> Option<Self> {
                    if n != 0 {
                        Some(unsafe { Self(n) })
                    } else {
                        None
                    }
                }

                pub const fn get(self) -> $Int {
                    self.0
                }

            }

            //const 方式实现trait
            impl const From<$Ty> for $Int {
                fn from(nonzero: $Ty) -> Self {
                    nonzero.0
                }
            }
```

```
            //本类型和本类型"|"运算符重载
            impl const BitOr for $Ty {
                type Output = Self;
                fn bitor(self, rhs: Self) -> Self::Output {
                    unsafe { $Ty::new_unchecked(self.get() | rhs.get()) }
                }
            }
            //本类型与基础类型的"|"运算符重载
            impl const BitOr<$Int> for $Ty {
                type Output = Self;
                fn bitor(self, rhs: $Int) -> Self::Output {
                    unsafe { $Ty::new_unchecked(self.get() | rhs) }
                }
            }
            //基础类型与本类型的"|"运算符重载
            impl const BitOr<$Ty> for $Int {
                type Output = $Ty;
                fn bitor(self, rhs: $Ty) -> Self::Output {
                    unsafe { $Ty::new_unchecked(self | rhs.get()) }
                }
            }
            //"|="运算符重载
            impl const BitOrAssign for $Ty {
                fn bitor_assign(&mut self, rhs: Self) {
                    *self = *self | rhs;
                }
            }

            impl const BitOrAssign<$Int> for $Ty {
                fn bitor_assign(&mut self, rhs: $Int) {
                    *self = *self | rhs;
                }
            }

            //其他运算符的重载,略
            ...
            ...
        )+
    }
}

nonzero_integers! {
    NonZeroU8(u8);
    NonZeroU16(u16);
    NonZeroU32(u32);
    NonZeroU64(u64);
    NonZeroU128(u128);
    NonZeroUsize(usize);
    NonZeroI8(i8);
    NonZeroI16(i16);
    NonZeroI32(i32);
```

```
    NonZeroI64(i64);
    NonZeroI128(i128);
    NonZeroIsize(isize);
}
```

NonZero 类型典型的体现了RUST程序设计的安全原则，所有的异常应该用类型系统表示出来，以强制获得处理。不要用临时性的措施。这样可以最大限度的避免bug的产生。

整形数据ops数学运算符，位运算符重载实现代码分析

以Add为例说明：

```
pub trait Add<Rhs = Self> {
    type Output;

    fn add(self, rhs: Rhs) -> Self::Output;
}

//利用宏简化操作
macro_rules! add_impl {
    ($($t:ty)*) => ($(
        impl const Add for $t {
            type Output = $t;
            //"+"号编译器默认实现是uncheckd_add
            //这里使用self，是一个消费操作。
            fn add(self, other: $t) -> $t { self + other }
        }

        forward_ref_binop! { impl const Add, add for $t, $t }
    )*)
}

//利用宏实现所有整形和浮点型运算符的重载
add_impl! { usize u8 u16 u32 u64 u128 isize i8 i16 i32 i64 i128 f32 f64 }
```

其他数学运算符及位运算符与此接近，因为代码逻辑简单，请参考标准库手册，略

bool类型方法代码分析

```
    pub const fn then_some<T>(self, t: T) -> Option<T>
    where
        //~const Destruct，没有找到确切的资料，
        //应该是如果T实现Drop，则必须是 impl const Drop
        //此处~const使得const在需要的时候非const化
        T: ~const Destruct,
    {
```

```
        if self { Some(t) } else { None }
    }
    pub const fn then<T, F>(self, f: F) -> Option<T>
    where
        F: ~const FnOnce() -> T,
        F: ~const Destruct,
    {
        if self { Some(f()) } else { None }
    }
```

利用Option对Try trait的支持，以上函数可以规避代码中的if...else..., 从而支持函数链式调用。

RUST字符(char)类型标准库代码分析

RUST的字符标准库主要是编程中常用到的字符相关操作,本节摘录一些显示RUST编码特点的内容。

由字符串转换为字符类型：
见如下代码：

```
impl FromStr for char {
    type Err = ParseCharError;

    //因为字符串用utf-8编码，而char是4字节变量，所以从字符串获取字符类型
    //不是简单的字符数组取值的关系，
    fn from_str(s: &str) -> Result<Self, Self::Err> {
        //s.chars()请参考前文
        let mut chars = s.chars();
        //下面对字符串做判断，字符串中应该只有一个字符存在，否则为错误
        //具体完成utf-8的字符到char的转换在chars.next()中完成，请参考前文
        match (chars.next(), chars.next()) {
            //无法得到字符
            (None, _) => Err(ParseCharError { kind: CharErrorKind::EmptyString
}),
            //存在一个字符
            (Some(c), None) => Ok(c),
            //其他情况
            _ => Err(ParseCharError { kind: CharErrorKind::TooManyChars }),
        }
    }
}
```

str::chars()函数请见前文字符串Iterator代码分析
u32转换为char,代码如下：

```
impl TryFrom<u32> for char {
    type Error = CharTryFromError;

    fn try_from(i: u32) -> Result<Self, Self::Error> {
        if (i > MAX as u32) || (i >= 0xD800 && i <= 0xDFFF) {
            Err(CharTryFromError(()))
        } else {
            // RUST不支持as从u32转换到char，这里只能用tranmute暴力转换
            Ok(unsafe { transmute(i) })
        }
    }
}
```

从任一进制的数值转换为char,代码如下：

```
pub fn from_digit(num: u32, radix: u32) -> Option<char> {
    //不支持大于36进制的数，因为无法用英文字母表示了
    if radix > 36 {
        panic!("from_digit: radix is too high (maximum 36)");
    }
    if num < radix {
        //转换为u8，后面可以与Byte类型做加法，b'0'是Byte类型的字面量
        let num = num as u8;
        if num < 10 { Some((b'0' + num) as char) } else { Some((b'a' + num -
10) as char) }
    } else {
        None
    }
}
```

将字符转换为某一进制的数值,以下例子充分的说明了RUST的安全性，相对于只有一种加法的C，RUST显著的降低了程序Bug出现的可能性

```
    pub fn to_digit(self, radix: u32) -> Option<u32> {
        assert!(radix <= 36, "to_digit: radix is too high (maximum 36)");
        // 利用wrapping_sub同时处理大于及小于'0'的字符，并且规避溢出
        let mut digit = (self as u32).wrapping_sub('0' as u32);
        if radix > 10 {
            if digit < 10 {
                return Some(digit);
            }
            // 用saturating_add保证digit不会折返
            digit = (self as u32 | 0b10_0000).wrapping_sub('a' as
u32).saturating_add(10);
        }
        //利用bool类型的方法简化了编程
```

```
        (digit < radix).then_some(digit)
    }
```

将字符转换为"\u{xxxx}"的形式：

```
//escape_unicode充分的展示了函数式编程的设计思想
//即以迭代器为中心来设计问题解决方案，
//对于任何一个问题，首先就看是否能设计一个实现Iterator Trait的结构来解决问题
pub fn escape_unicode(self) -> EscapeUnicode {
   let c = self as u32;

    // c|1避免有32个0出现
    let msb = 31 - (c | 1).leading_zeros();

    // 计算有多少个字符
    let ms_hex_digit = msb / 4;
    //生成结构，以便用Iterator解决问题
    EscapeUnicode {
        c: self,
        state: EscapeUnicodeState::Backslash,
        hex_digit_idx: ms_hex_digit as usize,
    }
}

pub struct EscapeUnicode {
    c: char,
    state: EscapeUnicodeState,

    // 当前还有多少个字符没有转换
    hex_digit_idx: usize,
}

// 显示转换的当前状态
#[derive(Clone, Debug)]
enum EscapeUnicodeState {
    //转换完成
    Done,
    //下一步应输出右括号
    RightBrace,
    //下一步应输出字母
    Value,
    //下一步应输出左括号
    LeftBrace,
    //输出Type的字符
    Type,
    //输出斜杠，第一个状态
    Backslash,
}

impl Iterator for EscapeUnicode {
```

```
    type Item = char;

    fn next(&mut self) -> Option<char> {
        match self.state {
            EscapeUnicodeState::Backslash => {
                self.state = EscapeUnicodeState::Type;
                Some('\\')
            }
            EscapeUnicodeState::Type => {
                self.state = EscapeUnicodeState::LeftBrace;
                Some('u')
            }
            EscapeUnicodeState::LeftBrace => {
                self.state = EscapeUnicodeState::Value;
                Some('{')
            }
            EscapeUnicodeState::Value => {
                let hex_digit = ((self.c as u32) >> (self.hex_digit_idx * 4)) &
0xf;
                let c = from_digit(hex_digit, 16).unwrap();
                if self.hex_digit_idx == 0 {
                    self.state = EscapeUnicodeState::RightBrace;
                } else {
                    self.hex_digit_idx -= 1;
                }
                Some(c)
            }
            EscapeUnicodeState::RightBrace => {
                self.state = EscapeUnicodeState::Done;
                Some('}')
            }
            EscapeUnicodeState::Done => None,
        }
    }

    fn size_hint(&self) -> (usize, Option<usize>) {
        let n = self.len();
        (n, Some(n))
    }

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

    fn last(self) -> Option<char> {
        match self.state {
            EscapeUnicodeState::Done => None,

            EscapeUnicodeState::RightBrace
            | EscapeUnicodeState::Value
            | EscapeUnicodeState::LeftBrace
```

```
            | EscapeUnicodeState::Type
            | EscapeUnicodeState::Backslash => Some('}'),
        }
    }
}

impl fmt::Display for EscapeUnicode {
    fn fmt(&self, f: &mut fmt::Formatter<'_>) -> fmt::Result {
        //利用Iterator输出转换字符串
        for c in self.clone() {
            f.write_char(c)?;
        }
        Ok(())
    }
}
```

EscapeUnicode 实现了Display Trait。可以调用to_string来输出字符串 RUST的字符模块的其他转换函数与EscapeUnicode采用了类似的设计，下面列出这些转换函数，但代码分析省略 `pub fn escape_debug(self) -> EscapeDebug` char的Debug转换输出 `pub fn to_lowercase(self) -> ToLowercase` char转换为小写 `pub fn to_uppercase(self) -> ToUppercase` char转换为大写

编码为UTF-8的字符串

```
    //dst应该保证有足够的空间放置utf-8字符串，&mut str的地址就是dst
    pub fn encode_utf8(self, dst: &mut [u8]) -> &mut str {
        unsafe { from_utf8_unchecked_mut(encode_utf8_raw(self as u32, dst)) }
    }

    pub unsafe fn from_utf8_unchecked_mut(v: &mut [u8]) -> &mut str {
        //调用者保证v能被安全的转换
        unsafe { &mut *(v as *mut [u8] as *mut str)
    }

    pub fn encode_utf8_raw(code: u32, dst: &mut [u8]) -> &mut [u8] {
        let len = len_utf8(code);
        match (len, &mut dst[..]) {
            //rust语法的强大展现，逻辑很简单，分析略
            (1, [a, ..]) => {
                *a = code as u8;
            }
            (2, [a, b, ..]) => {
                *a = (code >> 6 & 0x1F) as u8 | TAG_TWO_B;
                *b = (code & 0x3F) as u8 | TAG_CONT;
            }
            (3, [a, b, c, ..]) => {
                *a = (code >> 12 & 0x0F) as u8 | TAG_THREE_B;
                *b = (code >> 6 & 0x3F) as u8 | TAG_CONT;
                *c = (code & 0x3F) as u8 | TAG_CONT;
```

```
            }
            (4, [a, b, c, d, ..]) => {
                *a = (code >> 18 & 0x07) as u8 | TAG_FOUR_B;
                *b = (code >> 12 & 0x3F) as u8 | TAG_CONT;
                *c = (code >> 6 & 0x3F) as u8 | TAG_CONT;
                *d = (code & 0x3F) as u8 | TAG_CONT;
            }
            _ => panic!(
                "encode_utf8: need {} bytes to encode U+{:X}, but the buffer
has {}",
                len,
                code,
                dst.len(),
            ),
        };
        &mut dst[..len]
    }
}
```

字符串标准库代码分析

字符串模块的一个核心是Iterator，已经在Iterator章节中有过说明。
除了Iterator，字符串其他的方法及函数库代码摘要分析如下：

```
    pub const fn len(&self) -> usize {
        //字符串的len是字符串字节数目
        self.as_bytes().len()
    }
    //是否是字符的边界
    pub fn is_char_boundary(&self, index: usize) -> bool {
        // 0 位置总是边界
        if index == 0 {
            return true;
        }

        match self.as_bytes().get(index) {

            None => index == self.len(),

            // 巧妙的对字符边界的总结: b < 128 || b >= 192
            Some(&b) => (b as i8) >= -0x40,
        }
    }
    //目前I的类型仅支持:
    // usize, ..(RangeFull), start..(RangeFrom), start..end(Range)
    // start..=end(RangeInclusive), ..end(RangeTo), ..=end(RangeToInclusive)
    // get函数不会panic,但更习惯用str[usize],或者str[Range]来完成
    pub fn get<I: SliceIndex<str>>(&self, i: I) -> Option<&I::Output> {
```

```
        i.get(self)
    }
    //对i.get给出一个分析
    unsafe impl SliceIndex<str> for ops::Range<usize> {
        type Output = str;

        fn get(self, slice: &str) -> Option<&Self::Output> {
            //必须满足Range的两端都在字符边界处，否则返回None
            if self.start <= self.end
                && slice.is_char_boundary(self.start)
                && slice.is_char_boundary(self.end)
            {
                // 重新建立了一个&[str],具体见下面的函数
                Some(unsafe { &*self.get_unchecked(slice) })
            } else {
                None
            }
        }
        //最终离不开内存和裸指针
        unsafe fn get_unchecked(self, slice: *const str) -> *const Self::Output
{
            let slice = slice as *const [u8];
            let ptr = unsafe { slice.as_ptr().add(self.start) };
            let len = self.end - self.start;
            ptr::slice_from_raw_parts(ptr, len) as *const str
        }
        ...
    }

    //其他可以用Index实现的get_xxx函数及split_at函数，略
    ...
```

下面通过字符串的查找函数给出RUST良好的程序结构设计的一个例子：

```
    //字符串查找函数，可以用模式匹配查找子串
    //支持如下例子中的查找
    /// let s = "Löwe 老虎 Léopard Gepardi";
    /// 字符的查找
    /// assert_eq!(s.find('L'), Some(0));
    /// assert_eq!(s.find('é'), Some(14));
    ///
    /// 子字符串的查找
    /// assert_eq!(s.find("pard"), Some(17));
    ///
    /// 满足函数要求的字符或字符串的查找
    /// assert_eq!(s.find(char::is_lowercase), Some(1));
    /// assert_eq!(s.find(|c: char| c.is_whitespace() || c.is_lowercase()),
Some(1));
    /// assert_eq!(s.find(|c: char| (c < 'o') && (c > 'a')), Some(4));
    ///
```

```
    /// 字符数组的查找，注意RUST中字符数组与字符串是不同的两个类型
    /// assert_eq!(s.find(['老', 'G']))
```

由以上注释可以看到，rust的字符串查找函数功能强大，使用直观且易于理解。后继代码将展现RUST具备的：

1. 良好的扩展性，即使是原生类型，也可以直接在其上增加自定义Trait, 从而得到最直观的代码表现，而其他语言如C++/Java是无法在已经定义好的类型上做扩充的。只能创建新类型来实现对已有类型的功能扩展。不但在代码上不直观及冗余，也造成了额外的学习负担。
2. Trait语义的强大，即使对于闭包类型，也可以实现Trait。

```
    pub fn find<'a, P: Pattern<'a>>(&'a self, pat: P) -> Option<usize> {
        //利用Pattern Trait支持了众多类型的查找
        pat.into_searcher(self).next_match().map(|(i, _)| i)
    }
```

要设计这样一个find方法:

1. 显然，参数需要是一个泛型，但泛型应该支持同样的接口，即Pattern trait
2. 需要利用find的输入泛型参数，self来构造一个结构，并以这个结构为基础来实现方法完成查找。Pattern trait 的类型显然不可能作为这个结构(字符，字符切片，字符数组，闭包函数，字符串). 这个结构只能由Pattern trait的方法构造，事实上，Pattern trait最重要的工作就是构造这个结构。
3. 2构造的结构应该支持统一的接口，真正的实现查找

具体的实现定义如下：

```
    //模式 Trait 定义及公共行为
    pub trait Pattern<'a>: Sized {
        /// 与具体类型相适配的搜索算法的实现类型，类型必须实现Searcher Trait
        type Searcher: Searcher<'a>;

        /// 创建Searcher，根据输入的str及类型自身属性
        fn into_searcher(self, haystack: &'a str) -> Self::Searcher;

        /// 检查str是否存在对模式匹配的内容
        fn is_contained_in(self, haystack: &'a str) -> bool {
            self.into_searcher(haystack).next_match().is_some()
        }

        //略
        ...
    }
```

以下为Searcher trait定义。

```
    //Pattern匹配搜索算法的具体实现Trait
    pub unsafe trait Searcher<'a> {
        /// Searcher针对的字符串
        fn haystack(&self) -> &'a str;

        /// 执行下一次搜索， 返回搜索算法给出的:
        ///   [SearchStep::Match(a,b)] haystack[a..b]匹配了模式
        ///   [SearchStep::Reject(a,b)] haystack[a..b]不能匹配模式
        ///   [SearchStep::Done]
        /// next的返回结果应该上次放回的结果首尾相连。即如果上次返回Match(0,1), next
的返回
        /// 应该是Reject(1,_)或Match(1,_)。第一个返回必须是Reject(0,_)或
match(0,_), Done之前
        /// 的返回应该是Reject(_, haystack.len()-1)或Match(_, haystack.len())
        fn next(&mut self) -> SearchStep;

        /// 找到下一个匹配结果是Match的匹配结果
        fn next_match(&mut self) -> Option<(usize, usize)> {
            loop {
                match self.next() {
                    SearchStep::Match(a, b) => return Some((a, b)),
                    SearchStep::Done => return None,
                    _ => continue,
                }
            }
        }

        /// 找到下一个Reject
        fn next_reject(&mut self) -> Option<(usize, usize)> {
            loop {
                match self.next() {
                    SearchStep::Reject(a, b) => return Some((a, b)),
                    SearchStep::Done => return None,
                    _ => continue,
                }
            }
        }
    }

    pub enum SearchStep {
        /// 匹配时输出Match及子字符串的位置
        Match(usize, usize),
        /// 确定不匹配的子字符串的位置信息， 可以有多个不匹配的子字符串
        Reject(usize, usize),
        /// 字符串已经遍历完毕
        Done,
    }
```

下面为单字符的Pattern trait的系列实现，仅展示一下相应的逻辑关系。

```
    //针对char类型的Searcher Trait具现化类型
    pub struct CharSearcher<'a> { /*略*/ }
    //实现Searcher Trait
    unsafe impl<'a> Searcher<'a> for CharSearcher<'a> {
        //略
        ...
    }

    // 针对char 的Pattern实现, 支持如 "abc".find('a') 的形态
    impl<`a, `b> Pattern<`a> for char {
        type Searcher = CharSearcher

        //略
        ...
    }
```

下面为多字符的Pattern trait的实现，因为是比较典型的设计，所以重点的进行分析： 首先，设计字符匹配的trait，并在闭包，字符数组及其引用，字符切片类型中实现

```
    //支持  "abc".find(&['a','b'])的形态
    //      "abc".find(&['a','b'][..]) 的形态 &['a','b'][..] 实质是&[char]类型,
注意与&str类型的区别
    //      "abc".find(|ch| ch > 'a' && ch < 'c') 的形态

    //利用MultiCharEq trait 综合[char; N], &[char], FnMut(char)->bool
    //字符匹配操作
    trait MultiCharEq {
        fn matches(&mut self, c: char) -> bool;
    }

    //为FnMut(char)->bool 实现MultiCharEq
    impl<F> MultiCharEq for F
    where
        F: FnMut(char) -> bool,
    {
        fn matches(&mut self, c: char) -> bool {
            (*self)(c)
        }
    }

    //为[char;N]实现 MultiCharEq
    impl<const N: usize> MultiCharEq for [char; N] {
        fn matches(&mut self, c: char) -> bool {
            self.iter().any(|&m| m == c)
        }
    }
```

```
    impl<const N: usize> MultiCharEq for &[char; N] {
        fn matches(&mut self, c: char) -> bool {
            self.iter().any(|&m| m == c)
        }
    }

    // 为&[char]实现MultiCharEq
    impl MultiCharEq for &[char] {
        #[inline]
        fn matches(&mut self, c: char) -> bool {
            self.iter().any(|&m| m == c)
        }
    }
```

然后是基于泛型的统一的Pattern trait和Searcher的实现

```
    //利用输入类型构造一个泛型结构
    struct MultiCharEqPattern<C: MultiCharEq>(C);

    //与MultiCharEqPattern相匹配的Searcher Trait具现的结构体
    struct MultiCharEqSearcher<'a, C: MultiCharEq> {
        char_eq: C,
        haystack: &'a str,
        char_indices: super::CharIndices<'a>,
    }

    // 实现Pattern
    impl<'a, C: MultiCharEq> Pattern<'a> for MultiCharEqPattern<C> {
        type Searcher = MultiCharEqSearcher<'a, C>;

        //创建泛型Searcher结构
        fn into_searcher(self, haystack: &'a str) -> MultiCharEqSearcher<'a, C>
{
            MultiCharEqSearcher { haystack, char_eq: self.0, char_indices:
haystack.char_indices()}
        }
    }

    //针对泛型Searcher结构实现Searcher trait
    unsafe impl<'a, C: MultiCharEq> Searcher<'a> for MultiCharEqSearcher<'a, C>
{
        fn haystack(&self) -> &'a str {
            self.haystack
        }

        fn next(&mut self) -> SearchStep {
            let s = &mut self.char_indices;
            //pre_len用来计算char在字符串中占用了几个字节
            let pre_len = s.iter.iter.len();
            if let Some((i, c)) = s.next() {
```

```
                let len = s.iter.iter.len();
                //计算当前字符占用的字节数
                let char_len = pre_len - len;
                if self.char_eq.matches(c) {
                    return SearchStep::Match(i, i + char_len);
                } else {
                    return SearchStep::Reject(i, i + char_len);
                }
            }
            SearchStep::Done
        }
    }
```

下面是如何将MultiCharEqPattern及MultiCharEqSearcher应用在各类型的Pattern实现中。

```
////////////////////////////////////////////////////////////////////////////
    //利用宏简化代码
    macro_rules! pattern_methods {
        ($t:ty, $pmap:expr, $smap:expr) => {
            type Searcher = $t;

            fn into_searcher(self, haystack: &'a str) -> $t {
                //这里实际上是用self创建了MultiCharEqPattern(self)
                //随后用MutiEqPattern(self)创建MultiCharEqSearcher
                //然后封装MultiCharEqSearcher，创建一个与self类型关联的Searcher类型
的Searcher
                ($smap)(($pmap)(self).into_searcher(haystack))
            }

            fn is_contained_in(self, haystack: &'a str) -> bool {
                ($pmap)(self).is_contained_in(haystack)
            }

            fn is_prefix_of(self, haystack: &'a str) -> bool {
                ($pmap)(self).is_prefix_of(haystack)
            }

            fn strip_prefix_of(self, haystack: &'a str) -> Option<&'a str> {
                ($pmap)(self).strip_prefix_of(haystack)
            }

            fn is_suffix_of(self, haystack: &'a str) -> bool
            where
                $t: ReverseSearcher<'a>,
            {
                ($pmap)(self).is_suffix_of(haystack)
            }
```

```
            fn strip_suffix_of(self, haystack: &'a str) -> Option<&'a str>
            where
                $t: ReverseSearcher<'a>,
            {
                ($pmap)(self).strip_suffix_of(haystack)
            }
        };
    }

    // 利用宏简化代码
    macro_rules! searcher_methods {
        (forward) => {
            fn haystack(&self) -> &'a str {
                self.0.haystack()
            }
            fn next(&mut self) -> SearchStep {
                //实质是MultiCharEqSearcher<>::next
                self.0.next()
            }
            fn next_match(&mut self) -> Option<(usize, usize)> {
                self.0.next_match()
            }
            fn next_reject(&mut self) -> Option<(usize, usize)> {
                self.0.next_reject()
            }
        };
        (reverse) => {
            fn next_back(&mut self) -> SearchStep {
                self.0.next_back()
            }
            fn next_match_back(&mut self) -> Option<(usize, usize)> {
                self.0.next_match_back()
            }
            fn next_reject_back(&mut self) -> Option<(usize, usize)> {
                self.0.next_reject_back()
            }
        };
    }

    //下面这个结构比较清晰的说明了 Pattern, MultiCharEqPattern,
MultiCharEqSearcher的关系
    //使得代码更清晰
    pub struct CharArraySearcher<'a, const N: usize>(
        <MultiCharEqPattern<[char; N]> as Pattern<'a>>::Searcher,
    );

    /// 针对&[char;N]的Pattern, MultiCharEqPattern, MultiCharEqSearcher的关系
    pub struct CharArrayRefSearcher<'a, 'b, const N: usize>(
        <MultiCharEqPattern<&'b [char; N]> as Pattern<'a>>::Searcher,
    );
```

```
    // 利用上面的宏对[char;N]类型的Pattern Trait实现
    impl<'a, const N: usize> Pattern<'a> for [char; N] {
        pattern_methods!(CharArraySearcher<'a, N>, MultiCharEqPattern,
CharArraySearcher);
    }
    // 对[char;N]的searcher关联类型的Searcher Trait实现,
    unsafe impl<'a, const N: usize> Searcher<'a> for CharArraySearcher<'a, N> {
        searcher_methods!(forward);
    }

    unsafe impl<'a, const N: usize> ReverseSearcher<'a> for
CharArraySearcher<'a, N> {
        searcher_methods!(reverse);
    }

    // 针对&[char;N]的Pattern Trait 实现
    impl<'a, 'b, const N: usize> Pattern<'a> for &'b [char; N] {
        pattern_methods!(CharArrayRefSearcher<'a, 'b, N>, MultiCharEqPattern,
CharArrayRefSearcher);
    }

    // 对&[char;N]的searcher关联类型的Searcher Trait 实现
    unsafe impl<'a, 'b, const N: usize> Searcher<'a> for
CharArrayRefSearcher<'a, 'b, N> {
        searcher_methods!(forward);
    }

    unsafe impl<'a, 'b, const N: usize> ReverseSearcher<'a> for
CharArrayRefSearcher<'a, 'b, N> {
        searcher_methods!(reverse);
    }

    //针对&[char]的Searcher具现化结构体
    pub struct CharSliceSearcher<'a, 'b>(<MultiCharEqPattern<&'b [char]> as
Pattern<'a>>::Searcher);

    //Searcher Trait 实现
    unsafe impl<'a, 'b> Searcher<'a> for CharSliceSearcher<'a, 'b> {
        searcher_methods!(forward);
    }

    unsafe impl<'a, 'b> ReverseSearcher<'a> for CharSliceSearcher<'a, 'b> {
        searcher_methods!(reverse);
    }

    impl<'a, 'b> DoubleEndedSearcher<'a> for CharSliceSearcher<'a, 'b> {}

    // 对&[char]的Pattern Trait的实现
    impl<'a, 'b> Pattern<'a> for &'b [char] {
        pattern_methods!(CharSliceSearcher<'a, 'b>, MultiCharEqPattern,
CharSliceSearcher);
```

```
    }

    //针对FnMut(char)->bool的Searcher具现化结构体
    pub struct CharPredicateSearcher<'a, F>(<MultiCharEqPattern<F> as
Pattern<'a>>::Searcher)
    where
        F: FnMut(char) -> bool;

    //Searcher Trait 实现
    unsafe impl<'a, F> Searcher<'a> for CharPredicateSearcher<'a, F>
    where
        F: FnMut(char) -> bool,
    {
        searcher_methods!(forward);
    }

    unsafe impl<'a, F> ReverseSearcher<'a> for CharPredicateSearcher<'a, F>
    where
        F: FnMut(char) -> bool,
    {
        searcher_methods!(reverse);
    }

    impl<'a, F> DoubleEndedSearcher<'a> for CharPredicateSearcher<'a, F> where
F: FnMut(char) -> bool {}

    //针对FnMut(char)->bool的Pattern Trait 实现
    impl<'a, F> Pattern<'a> for F
    where
        F: FnMut(char) -> bool,
    {
        pattern_methods!(CharPredicateSearcher<'a, F>, MultiCharEqPattern,
CharPredicateSearcher);
    }
```

多字符搜索代码不复杂，但结构设计则可圈可点。而且似乎是不得不这样做设计。RUST利用泛型及trait能够自然的得到比较好的设计结果。
我们针对泛型做一个方法时，自然会对泛型用一个共用的trait——Pattern来约束。因为方法实现需要不同于泛型但紧密关联的另一个结构体，那这个结构体类型便自然的形成trait里的一个关联类型Searcher。而这个关联类型也自然应该用另一个trait——Searcher来约束。
Searcher的变量应该在Pattern的方法被创建出来。Searcher trait应该提供查找的方法。
这就是RUST语法自然导致好的设计的一个例子。

以下对子字符串搜索给出一些详细的解释，主要说明TwoWay算法

```
    //针对str实现的pattern,  支持如"abc".find("ab")的形态
```

```rust
impl<'a, 'b> Pattern<'a> for &'b str {
    //StrSeacher见下面该结构的代码注释
    type Searcher = StrSearcher<'a, 'b>;

    fn into_searcher(self, haystack: &'a str) -> StrSearcher<'a, 'b> {
        StrSearcher::new(haystack, self)
    }

    //略
}

pub struct StrSearcher<'a, 'b> {
    // 被查找目标字符串
    haystack: &'a str,
    // 查找的子字符串
    needle: &'b str,
    // 查找算法实现体
    searcher: StrSearcherImpl,
}

enum StrSearcherImpl {
    //两种搜索算法，后继还可以根据需要再扩充其他的算法
    Empty(EmptyNeedle),
    TwoWay(TwoWaySearcher),
}


impl<'a, 'b> StrSearcher<'a, 'b> {
    fn new(haystack: &'a str, needle: &'b str) -> StrSearcher<'a, 'b> {
        if needle.is_empty() {
            //略
            ...
            ...
        } else {
            StrSearcher {
                haystack,
                needle,
                searcher: StrSearcherImpl::TwoWay(TwoWaySearcher::new(
                    needle.as_bytes(),
                    haystack.len(),
                )),
            }
        }
    }
}

unsafe impl<'a, 'b> Searcher<'a> for StrSearcher<'a, 'b> {
    fn haystack(&self) -> &'a str {
        self.haystack
    }
```

```rust
        fn next(&mut self) -> SearchStep {
            //此处隐藏StrSearcher后继不会更换算法。如果更换搜索算法，应该将
StrSearcher整体做替换
            //
            match self.searcher {
                StrSearcherImpl::Empty(ref mut searcher) => {
                    //略
                    ...
                    ...
                }
                StrSearcherImpl::TwoWay(ref mut searcher) => {
                    if searcher.position == self.haystack.len() {
                        return SearchStep::Done;
                    }
                    let is_long = searcher.memory == usize::MAX;
                    match searcher.next::<RejectAndMatch>(
                        self.haystack.as_bytes(),
                        self.needle.as_bytes(),
                        is_long,
                    ) {
                        SearchStep::Reject(a, mut b) => {
                            // 因为searcher使用&[u8]来搜索，返回可能不是字节边界
                            while !self.haystack.is_char_boundary(b) {
                                b += 1;
                            }
                            searcher.position = cmp::max(b, searcher.position);
                            SearchStep::Reject(a, b)
                        }
                        //这个表示语法注意一下
                        otherwise => otherwise,
                    }
                }
            }
        }

        fn next_match(&mut self) -> Option<(usize, usize)> {
            match self.searcher {
                StrSearcherImpl::Empty(..) => loop {
                    //略
                    ...
                },
                StrSearcherImpl::TwoWay(ref mut searcher) => {
                    let is_long = searcher.memory == usize::MAX;
                    // 如果匹配，那匹配点一定是字符边界
                    if is_long {
                        searcher.next::<MatchOnly>(
                            self.haystack.as_bytes(),
                            self.needle.as_bytes(),
                            true,
                        )
                    } else {
```

```
                        searcher.next::<MatchOnly>(
                            self.haystack.as_bytes(),
                            self.needle.as_bytes(),
                            false,
                        )
                    }
                }
            }
        }
    }
    /*  查找子字符串算法的关键问题如下：
        1. 每次比较的匹配的位置在哪里？
        2. 不匹配时应移动多少个位置开始新一次匹配？

        显然，如果子字符串不存在周期性的重复，那每次比较如果不同就只能后移一个字符然后开始新的匹配
        所以，算法主要就是在子字符串中存在周期性重复的字符的情况下来如何更好的提高效率
        TwoWay算法仅在子字符串整体有周期性时发生左右，仅是内部少量字符的周期性，TwoWay算法不考虑。
        设待比较字符串为H，子字符串为S，周期字符串为w，周期为p，w的前缀为w-则S为w(w|w-)+
        对于S，TwoWay算法找到一个crit_pos，先从S的crit_pos的位置开始与H做比较到S的尾部，如果对应字符位置crit_pos+i
        的比较不成功，会在S上偏移i，清除记录，然后继续比较。
        如果直到尾部比较都成功，则会记录，然后开始比较头部，如果头部比较不成功，则偏移p，后继比较会考虑比较成功的记录。

        TwoWay不是最快的算法，但占用内存少，且也在一定程度上提高了效率。是比较适合的库方法
        */
    struct TwoWaySearcher {
        // constants
        /// 每次比较的开始位置，从此位置向尾部
        crit_pos: usize,
        /// 每次反向比较的开始位置，从此位置向前部
        crit_pos_back: usize,
        // 周期
        period: usize,
        /// 子字符串的位图，用来做一个快速甄别和判断
        byteset: u64,

        // 在待比较字符串的位置,从头部向后查找
        position: usize,
        // 待比较字符串的位置，从尾部向前查找
        end: usize,
        /// 在尾部比较成功后，记录已经比较过的字符串
        memory: usize,
        /// 同上，不过是反方向比较
        memory_back: usize,
    }
```

```
    impl TwoWaySearcher {
        fn new(needle: &[u8], end: usize) -> TwoWaySearcher {
            let (crit_pos_false, period_false) =
TwoWaySearcher::maximal_suffix(needle, false);
            let (crit_pos_true, period_true) =
TwoWaySearcher::maximal_suffix(needle, true);

            //找到更偏向尾部的位置
            let (crit_pos, period) = if crit_pos_false > crit_pos_true {
                (crit_pos_false, period_false)
            } else {
                (crit_pos_true, period_true)
            };

            //这里可以看出，只有从头部开始的周期字符串获得支持
            if needle[..crit_pos] == needle[period..period + crit_pos] {
                let crit_pos_back = needle.len()
                    - cmp::max(
                        TwoWaySearcher::reverse_maximal_suffix(needle, period,
false),
                        TwoWaySearcher::reverse_maximal_suffix(needle, period,
true),
                    );

                TwoWaySearcher {
                    crit_pos,
                    crit_pos_back,
                    period,
                    byteset: Self::byteset_create(&needle[..period]),

                    position: 0,
                    end,
                    memory: 0,
                    memory_back: needle.len(),
                }
            } else {
                // 字符串内没有周期性，及仅具备局部周期的字符串

                TwoWaySearcher {
                    crit_pos,
                    crit_pos_back: crit_pos,
                    period: cmp::max(crit_pos, needle.len() - crit_pos) + 1,
                    byteset: Self::byteset_create(needle),

                    position: 0,
                    end,
                    memory: usize::MAX, // Dummy value to signify that the
period is long
                    memory_back: usize::MAX,
                }
            }
```

```
        }

        fn byteset_create(bytes: &[u8]) -> u64 {
            bytes.iter().fold(0, |a, &b| (1 << (b & 0x3f)) | a)
        }

        fn byteset_contains(&self, byte: u8) -> bool {
            (self.byteset >> ((byte & 0x3f) as usize)) & 1 != 0
        }

        fn next<S>(&mut self, haystack: &[u8], needle: &[u8], long_period:
bool) -> S::Output
        where
            S: TwoWayStrategy,
        {
            // `next()` uses `self.position` as its cursor
            let old_pos = self.position;
            let needle_last = needle.len() - 1;
            'search: loop {
                // Check that we have room to search in
                // position + needle_last can not overflow if we assume slices
                // are bounded by isize's range.
                let tail_byte = match haystack.get(self.position + needle_last)
{
                    Some(&b) => b,
                    None => {
                        self.position = haystack.len();
                        return S::rejecting(old_pos, self.position);
                    }
                };

                //及早返回不匹配的信息
                if S::use_early_reject() && old_pos != self.position {
                    return S::rejecting(old_pos, self.position);
                }

                // 用位图判断出tail_byte不在子字符串中，可以立刻偏移到下一个字节再比
较
                if !self.byteset_contains(tail_byte) {
                    self.position += needle.len();
                    if !long_period {
                        self.memory = 0;
                    }
                    continue 'search;
                }

                // 如果memory有值且大于crip_pos，那就从memory开始比较，memory前的已
经匹配完毕
                // long_period 没有memory的逻辑，和暴力比较无差异
                let start =
                    if long_period { self.crit_pos } else {
```

```
cmp::max(self.crit_pos, self.memory) };
                for i in start..needle.len() {
                    if needle[i] != haystack[self.position + i] {
                        self.position += i - self.crit_pos + 1;
                        if !long_period {
                            self.memory = 0;
                        }
                        continue 'search;
                    }
                }

                // See if the left part of the needle matches
                let start = if long_period { 0 } else { self.memory };
                for i in (start..self.crit_pos).rev() {
                    if needle[i] != haystack[self.position + i] {
                        //period后面的字符已经比较完毕，period一般大于crit_pos
                        self.position += self.period;
                        if !long_period {
                            self.memory = needle.len() - self.period;
                        }
                        continue 'search;
                    }
                }

                // 比较全部完成，
                let match_pos = self.position;

                // 为下一次比较做准备
                self.position += needle.len();
                if !long_period {
                    self.memory = 0; // set to needle.len() - self.period for
overlapping matches
                }

                return S::matching(match_pos, match_pos + needle.len());
            }
        }

        // 略

    }
    // TwoWayStrategy allows the algorithm to either skip non-matches as
quickly
    // as possible, or to work in a mode where it emits Rejects relatively
quickly.
    trait TwoWayStrategy {
        type Output;
        fn use_early_reject() -> bool;
        fn rejecting(a: usize, b: usize) -> Self::Output;
        fn matching(a: usize, b: usize) -> Self::Output;
    }
```

```rust
/// Skip to match intervals as quickly as possible
enum MatchOnly {}

impl TwoWayStrategy for MatchOnly {
    type Output = Option<(usize, usize)>;

    fn use_early_reject() -> bool {
        false
    }
    fn rejecting(_a: usize, _b: usize) -> Self::Output {
        None
    }
    fn matching(a: usize, b: usize) -> Self::Output {
        Some((a, b))
    }
}

/// Emit Rejects regularly
enum RejectAndMatch {}

impl TwoWayStrategy for RejectAndMatch {
    type Output = SearchStep;

    fn use_early_reject() -> bool {
        true
    }
    fn rejecting(a: usize, b: usize) -> Self::Output {
        SearchStep::Reject(a, b)
    }
    fn matching(a: usize, b: usize) -> Self::Output {
        SearchStep::Match(a, b)
    }
}
```

以上对字符串查找的方法进行了分析，利用Pattern的还有以下的方法：

```rust
//生成一个支持Iterator的结构完成split
pub fn split<'a, P: Pattern<'a>>(&'a self, pat: P) -> Split<'a, P> {
    Split(SplitInternal {
        start: 0,
        end: self.len(),
        matcher: pat.into_searcher(self),
        allow_trailing_empty: true,
        finished: false,
    })
}

//略
```

```
    ...
    ...
```

切片标准库代码分析

切片排序

插入排序

```rust
/// 插入排序，复杂度O(n^2).
fn insertion_sort<T, F>(v: &mut [T], is_less: &mut F)
where
    F: FnMut(&T, &T) -> bool,
{
    //排序场景下，基本不能使用iterator
    for i in 1..v.len() {
        //利用
        shift_tail(&mut v[..i + 1], is_less);
    }
}

/// 将最后的值左移到遇到更小的值.
fn shift_tail<T, F>(v: &mut [T], is_less: &mut F)
where
    F: FnMut(&T, &T) -> bool,
{
    let len = v.len();

    // 因为是对泛型排序，RUST的排序算法比较复杂， 需要指出，&mut [T] 保证了外界不会有对数组或数组元素的引用，而数组元素本身的内存
    // 浅拷贝等同于所有权转移，不会出现内存安全问题。
    unsafe {
        if len >= 2 && is_less(v.get_unchecked(len - 1), v.get_unchecked(len - 2)) {
            // ManuallyDrop把drop的权利从rust编译器接管
            let mut tmp = mem::ManuallyDrop::new(ptr::read(v.get_unchecked(len - 1)));
            // CopyOnDrop会在drop的时候做src到dest的拷贝
            let mut hole = CopyOnDrop { src: &mut *tmp, dest: v.get_unchecked_mut(len - 2) };
            ptr::copy_nonoverlapping(v.get_unchecked(len - 2), v.get_unchecked_mut(len - 1), 1);

            //正常的排序内存置换操作
            for i in (0..len - 2).rev() {
                if !is_less(&*tmp, v.get_unchecked(i)) {
                    break;
                }
```

```
                ptr::copy_nonoverlapping(v.get_unchecked(i),
v.get_unchecked_mut(i + 1), 1);
                hole.dest = v.get_unchecked_mut(i);
            }
        }
    }
}
```

上面的排序算法最重要的是理解在元素转移的过程为什么没有影响所有权，为什么没有引发内存安全问题。这个例子充分说明了内存先关函数的重要性。