`LinkedList<T>` 代码分析

双向链表及其他数据结构的代码实现都是经典的实用性及训练性上佳的项目。本书对这些经典数据结构将只分析LinkedList，重点分析RUST与其他语言的不同的部分。如果对LinkedList彻底理解了，那其他数据结构也就不成为问题: `LinkedList<T>` 类型结构定义如下:

```
//这个定义表示LinkedList只支持固定长度的T类型
pub struct LinkedList<T> {
    //等同于直接用裸指针，使得代码最方便及简化，但需要对安全性额外投入精力
    //这个实际上与C语言相同，只是用Option增加了安全措施
    head: Option<NonNull<Node<T>>>,
    tail: Option<NonNull<Node<T>>>,
    len: usize,
    //marker说明本结构有一个Box<Node<T>>的所有权，并会负责调用其的drop
    //编译器应做好drop check，检查与本结构相关的Box<Node<T>>的生命周期及drop
    //marker体现了RUST的独特点
    marker: PhantomData<Box<Node<T>>>,
}

struct Node<T> {
    next: Option<NonNull<Node<T>>>,
    prev: Option<NonNull<Node<T>>>,
    element: T,
}
```

Node方法代码:

```
impl<T> Node<T> {
    fn new(element: T) -> Self {
        Node { next: None, prev: None, element }
    }

    fn into_element(self: Box<Self>) -> T {
        //消费了Box，堆内存被释放并将element拷贝到栈
        self.element
    }
}
```

LinkedList的创建及简单的增减方法:

```
impl<T> LinkedList<T> {
    //创建一个空的LinkedList
    pub const fn new() -> Self {
```

```
        LinkedList { head: None, tail: None, len: 0, marker: PhantomData }
    }
```

在头部增加一个成员及删除一个成员：

```
    //在首部增加一个节点
    pub fn push_front(&mut self, elt: T) {
        //用box从堆内存申请一个节点，push_front_node见后面函数
        self.push_front_node(box Node::new(elt));
    }
    fn push_front_node(&mut self, mut node: Box<Node<T>>) {
        // 整体全是不安全代码
        unsafe {
            node.next = self.head;
            node.prev = None;
            //需要将Box的堆内存leak出来使用。此块内存后继如果还在链表，需要由
LinkedList负责drop.后面可以看到LinkedList的drop函数的处理。
            //如果pop出链表，那会重新用这里leak出来的NonNull<Node<T>>生成Box,再由
Box释放
            let node = Some(Box::leak(node).into());

            match self.head {
                //空链表
                None => self.tail = node,
                // 目前采用NonNull<Node<T>>的方案，此处代码就很自然
                // 如果换成Box<Node<T>>的方案，这里就要类似如下:
                // 先用take将head复制到栈中创建的新变量，
                // 新变量的prev置为node
                // 用replace将新变量再复制回head。
                // 也注意，此处很容易也采用先take，修改，然后replace的方案
                // 要注意规避Option导致的这个习惯,会造成两次内存拷贝，效率太低
                Some(head) => (*head.as_ptr()).prev = node,
            }

            self.head = node;
            self.len += 1;
        }
    }

    //从链表头部删除一个节点
    pub fn pop_front(&mut self) -> Option<T> {
        //Option<T>::map，此函数后，节点的堆内存已经被释放
        //变量被拷贝到栈内存
        self.pop_front_node().map(Node::into_element)
    }
    fn pop_front_node(&mut self) -> Option<Box<Node<T>>> {
        //整体是unsafe
        self.head.map(|node| unsafe {
            //重新生成Box，以便后继可以释放堆内存
            let node = Box::from_raw(node.as_ptr());
```

```
            //更换head指针
            self.head = node.next;

            match self.head {
                None => self.tail = None,
                // push_front_node() 已经分析过
                Some(head) => (*head.as_ptr()).prev = None,
            }

            self.len -= 1;
            node
        })
    }
```

在尾部增加一个成员及删除一个成员

```
    //从尾部增加一个节点
    pub fn push_back(&mut self, elt: T) {
        //用box从堆内存申请一个节点
        self.push_back_node(box Node::new(elt));
    }

    fn push_back_node(&mut self, mut node: Box<Node<T>>) {
        // 整体不安全
        unsafe {
            node.next = None;
            node.prev = self.tail;
            //需要将Box的堆内存leak出来使用。此块内存后继如果还在链表，需要由
LinkedList负责drop.
            //如果pop出链表，那会重新用这里leak出来的NonNull<Node<T>>重新生成Box
            let node = Some(Box::leak(node).into());

            match self.tail {
                None => self.head = node,
                //前面代码已经有分析
                Some(tail) => (*tail.as_ptr()).next = node,
            }

            self.tail = node;
            self.len += 1;
        }
    }

    //从尾端删除节点
    pub fn pop_back(&mut self) -> Option<T> {
        self.pop_back_node().map(Node::into_element)
    }

    fn pop_back_node(&mut self) -> Option<Box<Node<T>>> {
        self.tail.map(|node| unsafe {
```

```
            //重新创建Box以便删除堆内存
            let node = Box::from_raw(node.as_ptr());
            self.tail = node.prev;

            match self.tail {
                None => self.head = None,

                Some(tail) => (*tail.as_ptr()).next = None,
            }

            self.len -= 1;
            node
        })
    }
    //删除一个节点，这个操作也是RUST比较独特的体现
    unsafe fn unlink_node(&mut self, mut node: NonNull<Node<T>>) {
        //现在拥有node的所有权，
        let node = unsafe { node.as_mut() };

        match node.prev {
            //不能复制新的节点，注意这里的写法
            Some(prev) => unsafe { (*prev.as_ptr()).next = node.next },
            // node是head节点
            None => self.head = node.next,
        };

        match node.next {
            //不能获取next的所有权，只能是这个写法
            Some(next) => unsafe { (*next.as_ptr()).prev = node.prev },
            // node是tail节点
            None => self.tail = node.prev,
        };

        self.len -= 1;
    }
    ...
}

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

        impl<'a, T> Drop for DropGuard<'a, T> {
            fn drop(&mut self) {
                //如果此函数后面的while循环出现panic，这里可以继续做释放
                //此处代码的存在应该是RUST标准库中隐藏比较深的bug导致
                while self.0.pop_front_node().is_some() {}
            }
        }
```

```
        while let Some(node) = self.pop_front_node() {
            let guard = DropGuard(self);
            //显式的drop 获取的Box<Node<T>>
            drop(node);
            //执行到此处，guard认为已经完成，不能再调用guard的drop
            mem::forget(guard);
        }
    }
}
```

以上基本上说明了RUST的LinkedList的设计及代码的一些关键点。

用Iterator来对List进行访问，Iterator的相关结构代码如下： into_iter()相关结构及方法：

```
//变量本身的Iterator的类型
pub struct IntoIter<T> {
    list: LinkedList<T>,
}

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

    /// 对LinkedList<T> 做消费
    fn into_iter(self) -> IntoIter<T> {
        IntoIter { list: self }
    }
}

impl<T> Iterator for IntoIter<T> {
    type Item = T;

    fn next(&mut self) -> Option<T> {
        //从头部获取变量
        self.list.pop_front()
    }

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

iter_mut()调用相关结构及方法

```
//可变引用的Iterator的类型
pub struct IterMut<'a, T: 'a> {
    head: Option<NonNull<Node<T>>>,
    tail: Option<NonNull<Node<T>>>,
```

```
    len: usize,
    //这个marker也标示了IterMut对LinkedList有一个可变引用
    //创建IterMut后，与之相关的LinkerList不能在被其他安全的代码修改
    marker: PhantomData<&'a mut Node<T>>,
}

impl <T> LinkedList<T> {
    ...
    pub fn iter_mut(&mut self) -> IterMut<'_, T> {
        IterMut { head: self.head, tail: self.tail, len: self.len, marker:
PhantomData }
    }
    ...
}
impl<'a, T> Iterator for IterMut<'a, T> {
    type Item = &'a mut T;

    fn next(&mut self) -> Option<&'a mut T> {
        if self.len == 0 {
            None
        } else {
            //用Option::map简化代码
            self.head.map(|node| unsafe {
                // 保存首部成员
                let node = &mut *node.as_ptr();
                // 删除首部成员
                self.len -= 1;
                self.head = node.next;
                // 返回可变引用，此处的生命周期如下：
                // 返回值生命周期小于self
                // self 生命周期小于 LinkedList
                &mut node.element
            })
        }
    }

    ...
}

//不可变引用的Iterator的类型
pub struct Iter<'a, T: 'a> {
    head: Option<NonNull<Node<T>>>,
    tail: Option<NonNull<Node<T>>>,
    len: usize,
    //对生命周期做标识，也标识了一个对LinkedList的不可变引用
    marker: PhantomData<&'a Node<T>>,
}

impl<T> Clone for Iter<'_, T> {
    fn clone(&self) -> Self {
        //本书中第一次出现这个表述
```

```
        Iter { ..*self }
    }
}

//Iterator trait for Iter略
```

LinkedList其他的代码略。 LinkedList当然有不使用unsafe方式的实现方法，但是unsafe的实现方式最简化，效率最高。且unsafe的代码量并不高，可控性很强。盲目的排斥unsafe实际上也是一件不RUST的事。

String 类型分析

String结构定义如下：

```
pub struct String {
    vec: Vec<u8>,
}
```

`Vec<u8>`和String的关系可以与[u8]与&str的关系相对比。整个String实际上是一个大的Adapter 模式，针对Vec, [u8], &str三者做组合 String的创建函数：

```
impl String {
     pub const fn new() -> String {
        String { vec: Vec::new() }
    }

    //将str内容加到String的尾部
    pub fn push_str(&mut self, string: &str) {
        //adapter, 直接用Vec::extend_from_slice([u8])
        //具体的细节请参考Vec那节
        self.vec.extend_from_slice(string.as_bytes())
    }
    ...
}

impl ToOwned for str {
    type Owned = String;
    fn to_owned(&self) -> String {
        //这里是个adapter模式，首先从用self.as_bytes()获取[u8]，然后用通用的
[u8].to_owned()完成
        //to_owned逻辑,随后从Vec[u8]生成String
        unsafe { String::from_utf8_unchecked(self.as_bytes().to_owned()) }
    }


    fn clone_into(&self, target: &mut String) {
```

```
        //adapter模式，需要先得到Vec<u8>,因为into_bytes会消费掉String。
        //target不支持，所以需要用take先把所有权转移出来，然后获取Vec<u8>
        //这是RUST的一个通用的技巧
        let mut b = mem::take(target).into_bytes();
        //通用的[u8].clone_into
        self.as_bytes().clone_into(&mut b);
        //把新的String赋给原先的地址
        *target = unsafe { String::from_utf8_unchecked(b) }
    }
}

impl From<&str> for String {
    fn from(s: &str) -> String {
        s.to_owned()
    }
}
```

解引用方法代码：

```
impl ops::Deref for String {
    type Target = str;

    fn deref(&self) -> &str {
        //&self.vec会被强转为&[u8]
        unsafe { str::from_utf8_unchecked(&self.vec) }
    }
}

impl ops::DerefMut for String {
    fn deref_mut(&mut self) -> &mut str {
        //这里直接用&mut self.vec应该也可以，会被强转成&mut [u8]
        unsafe { str::from_utf8_unchecked_mut(&mut *self.vec) }
    }
}
```

运算符重载方法

```
impl ops::Index<ops::RangeFull> for String {
    type Output = str;

    fn index(&self, _index: ops::RangeFull) -> &str {
        unsafe { str::from_utf8_unchecked(&self.vec) }
    }
}
impl ops::Index<ops::Range<usize>> for String {
    type Output = str;
```

```
    fn index(&self, index: ops::Range<usize>) -> &str {
        //先用Index<RangeFull>取&str，然后用Index<Range>取子串
        &self[..][index]
    }
}
impl Borrow<str> for String {
    fn borrow(&self) -> &str {
        //自动解引用，利用Index<RangeFull>完成，代码最简
        &self[..]
    }
}

impl BorrowMut<str> for String {
    fn borrow_mut(&mut self) -> &mut str {
        //自动解引用，利用Index<RangeFull>完成，代码最简
        &mut self[..]
    }
}

impl Add<&str> for String {
    type Output = String;

    fn add(mut self, other: &str) -> String {
        self.push_str(other);
        self
    }
}

impl AddAssign<&str> for String {
    fn add_assign(&mut self, other: &str) {
        self.push_str(other);
    }
}
```

字符串数组连接方法：

```
//此函数主要简化多个字符串的连接
impl<S: Borrow<str>> Concat<str> for [S] {
    type Output = String;

    fn concat(slice: &Self) -> String {
        //见下个方法分析
        Join::join(slice, "")
    }
}

impl<S: Borrow<str>> Join<&str> for [S] {
    type Output = String;
```

```
    fn join(slice: &Self, sep: &str) -> String {
        unsafe { String::from_utf8_unchecked(join_generic_copy(slice,
sep.as_bytes())) }
    }
}

macro_rules! specialize_for_lengths {
    ($separator:expr, $target:expr, $iter:expr; $($num:expr),*) => {{
        let mut target = $target;
        let iter = $iter;
        let sep_bytes = $separator;
        match $separator.len() {
            $(
                // 如果分隔切片长度符合预设值
                $num => {
                    for s in iter {
                        //拷贝分隔切片到目的切片，且更新目的切片
                        copy_slice_and_advance!(target, sep_bytes);
                        //拷贝内容切片
                        let content_bytes = s.borrow().as_ref();
                        copy_slice_and_advance!(target, content_bytes);
                    }
                },
            )*
            _ => {
                // 如果分隔切片长度不符合预设值，实质也做与上段代码同样的操作
                for s in iter {
                    copy_slice_and_advance!(target, sep_bytes);
                    let content_bytes = s.borrow().as_ref();
                    copy_slice_and_advance!(target, content_bytes);
                }
            }
        }
        target
    }}
}

//完成一个切片拷贝后，切片向前到未拷贝的开始处
macro_rules! copy_slice_and_advance {
    ($target:expr, $bytes:expr) => {
        let len = $bytes.len();
        //将目的切片切分成两段，首段为待拷贝空间，尾端为未拷贝空间
        let (head, tail) = { $target }.split_at_mut(len);
        head.copy_from_slice($bytes);
        $target = tail;
    };
}

//将若干个T类型的切片连接到一起形成一个基于T类型的切片
fn join_generic_copy<B, T, S>(slice: &[S], sep: &[T]) -> Vec<T>
where
```

```
    T: Copy, //最基础的成员类型
    B: AsRef<[T]> + ?Sized, //可以表示为最基础成员的切片引用
    S: Borrow<B>, //以B类型作为操作类型，所以S应该能borrow成B类型的引用
{
    let sep_len = sep.len();
    let mut iter = slice.iter();

    // 第一个成员头部没有间隔
    let first = match iter.next() {
        Some(first) => first,
        None => return vec![],
    };

    //计算iter中所有成员的长度，并加上间隔长度乘剩余成员的数目
    //得到总的长度。
    //从这个函数能够发现rust的链式编程的能力

    let reserved_len = sep_len
        .checked_mul(iter.len())//这里去掉了slice的首个成员，
        .and_then(|n| {
            //这里的重新重新生成iter，计算了所有的slice的所有成员
            slice.iter().map(|s| s.borrow().as_ref().len()).try_fold(n,
usize::checked_add)
        })
        .expect("attempt to join into collection with len > usize::MAX");

    // 创建一个有足够容量的Vec
    let mut result = Vec::with_capacity(reserved_len);
    debug_assert!(result.capacity() >= reserved_len);
    //完成first的内容拷贝
    result.extend_from_slice(first.borrow().as_ref());

    unsafe {
        let pos = result.len();
        let target = result.get_unchecked_mut(pos..reserved_len);

        //完成对剩余成员及分隔符拷贝到result
        let remain = specialize_for_lengths!(sep, target, iter; 0, 1, 2, 3, 4);

        //完成长度拷贝。
        let result_len = reserved_len - remain.len();
        result.set_len(result_len);
    }
    result
}
```

RUST的fmt相关代码

fmt给出RUST实现可变参数的解决方案。
alloc库中给出了format宏，完成对可变参数的格式化输出。
format宏代码如下：

```
macro_rules! format {
    ($($arg:tt)*) => {{
        //format宏调用后继的format函数，并由format_args宏将可变参数完成参数转换
        let res = $crate::fmt::format($crate::__export::format_args!
($($arg)*));
        res
    }}
}
```

format_args宏将可变参数转换成Arguments类型变量，可以作为RUST的可变参数支持的经典案列。

```
//因为安全的原因，下宏由编译器实现，
//format_args宏对输入的字符串和参数分析后返回类型为Arguments的变量，
macro_rules! format_args {
    ($fmt:expr) => {{ /* compiler built-in */ }};
    ($fmt:expr, $($args:tt)*) => {{ /* compiler built-in */ }};
}
//Arguments类型结构
pub struct Arguments<'a> {
    // 存放需要格式化的参数之间的字符串，对应于每一个格式化参数
    // 此字符串可以为空
    pieces: &'a [&'static str],

    // 针对每个格式化参数的格式描述
    fmt: Option<&'a [rt::v1::Argument]>,

    // 每个参数，以及生成参数的格式化字符串的函数
    args: &'a [ArgumentV1<'a>],
}
```

format_args生成Arguments举例如下： format_args!(“ab {:b} cd {:p}”, 1, 2) 结果的Arguments结构中：
其中**pieces**有两个成员，为:`"ab ", " cd "`,注意字符串中的空格 fmt有两个成员，为：

```
    //具体结构见后继的定义
    { postion:0,
      format:{align:UnKnown, flags:0, precision:Implied, width:Implied}},
    { position:1,
      format:{align:UnKnown, flags:4, precision:Implied, width:Implied}}
```

其中**args**有两个成员为:

```
    //具体的结构见后继的定义
    {1, core::fmt::num::Binary::fmt()},
    {2, core::fmt::num::Pointer::fmt()}
```

fmt及args相关的类型定义如下：

```
//rt::v1::Argument
//对非默认格式化参数，每个参数format_args!宏会生成一个Argument变量
pub struct Argument {
    //表示参数的在Arguments中的序号，
    pub position: usize,
    //格式参数，用于格式化输出
    pub format: FormatSpec,
}
pub struct FormatSpec {
    //格式化时需要填充的字符
    pub fill: char,
    pub align: Alignment,
    //FlagV1 按位赋值
    pub flags: u32,
    pub precision: Count,
    pub width: Count,
}

//上面结构中的辅助类型
pub enum Alignment {
    /// 左端对齐
    Left,
    /// 右端对齐
    Right,
    /// 中间对齐
    Center,
    /// 没有对齐
    Unknown,
}

//flags 的位，
enum FlagV1 {
    SignPlus, //0
    SignMinus,  //1
    Alternate,  //2
    SignAwareZeroPad, //3
    DebugLowerHex,  //4
    DebugUpperHex,  //5
}

pub enum Count {
    /// 字面量的值
    Is(usize),
```

```
    /// Specified using `$` and `*` syntaxes, stores the index into `args`
    Param(usize),
    /// Not specified
    Implied,
}

//以下结构可认为是针对每一个参数，都有一个格式化输出的函数与其对应
pub struct ArgumentV1<'a> {
    //类似C语言的va_arg的返回类型，可以认为是void *
    value: &'a Opaque,
    //针对value的格式化输出函数
    formatter: fn(&Opaque, &mut Formatter<'_>) -> Result,
}

//上述结构中的类型
//类似void
extern "C" {
    type Opaque;
}

//每个格式化参数需要生成一个Formatter变量
//用于存放格式化信息以指示如和生成参数的格式化字符串
//生成的格式化字符串应输出到哪里
pub struct Formatter<'a> {
    //以下到precision都是由format_arg!宏在发现参数要求非默认的格式化时
    //生成的。
    flags: u32,
    fill: char,
    align: rt::v1::Alignment,
    width: Option<usize>,
    precision: Option<usize>,

    //格式化字符串输出的缓存，当前一般就是String
    buf: &'a mut (dyn Write + 'a),
}
```

format_args宏完成Arguments的生成后， 下面的format函数将之作为参数完成格式化字符串生成.

```
//Arguments包含了本输出中所有的需要格式化的参数
pub fn format(args: Arguments<'_>) -> string::String {
    //估计了输出字符串长度，尽量减少堆内存的重新申请
    let capacity = args.estimated_capacity();
    //申请足够空间的字符串
    let mut output = string::String::with_capacity(capacity);
    //根据输入的格式化参数，完成对参数的格式化字符串输出
    output.write_fmt(args).expect("a formatting trait implementation returned
an error");
```

```
    output
}
```

使用了String::write_fmt, 是Write trait 的方法，String实现了此trait，代码如下：

```
pub trait Write {
    fn write_str(&mut self, s: &str) -> Result;
    fn write_char(&mut self, c: char) -> Result {
        self.write_str(c.encode_utf8(&mut [0; 4]))
    }

    //格式化的输出
    fn write_fmt(mut self: &mut Self, args: Arguments<'_>) -> Result {
        //见后面的write函数分析
        write(&mut self, args)
    }
}
// String的Write trait实现
impl fmt::Write for String {
    fn write_str(&mut self, s: &str) -> fmt::Result {
        self.push_str(s);
        Ok(())
    }

    fn write_char(&mut self, c: char) -> fmt::Result {
        self.push(c);
        Ok(())
    }
}

//Formatter的Write trait实现
impl Write for Formatter<'_> {
    fn write_str(&mut self, s: &str) -> Result {
        self.buf.write_str(s)
    }

    fn write_char(&mut self, c: char) -> Result {
        self.buf.write_char(c)
    }

    fn write_fmt(&mut self, args: Arguments<'_>) -> Result {
        write(self.buf, args)
    }
}

//这里是Display, Debug常用的另外一个格式化输出的宏write!
macro_rules! write {
    ($dst:expr, $($arg:tt)*) => {
        //$dst即&mut dyn Write
        $dst.write_fmt($crate::format_args!($($arg)*))
```

```
    };
}

//此函数是格式化输入的核心函数，output当前可暂时认为是String
pub fn write(output: &mut dyn Write, args: Arguments<'_>) -> Result {
    //创建格式化参数的变量,buf设置为output
    let mut formatter = Formatter::new(output);
    let mut idx = 0;

    match args.fmt {
        //如果所有参数都是默认格式输出
        None => {
            // 对所有的参数进行轮询
            for (i, arg) in args.args.iter().enumerate() {
                //获取该参数前需要输出的字符串
                let piece = unsafe { args.pieces.get_unchecked(i) };
                if !piece.is_empty() {
                    //向output输出获取的字符串
                    formatter.buf.write_str(*piece)?;
                }
                //调用每个参数的格式化输出函数，向formatter输出格式化参数字符串
                //此时formatter所有的成员都是默认
                (arg.formatter)(arg.value, &mut formatter)?;
                idx += 1;
            }
        }
        //如果有参数不是默认格式输出
        Some(fmt) => {
            // 对所有参数进行轮询
            for (i, arg) in fmt.iter().enumerate() {
                // 获取该参数前应该输出的字符串
                let piece = unsafe { args.pieces.get_unchecked(i) };
                if !piece.is_empty() {
                    //向output输出获取的字符串
                    formatter.buf.write_str(*piece)?;
                }
                //生成格式并输出格式化参数字符串
                unsafe { run(&mut formatter, arg, args.args) }?;
                idx += 1;
            }
        }
    }

    // 如果还有额外的字符串
    if let Some(piece) = args.pieces.get(idx) {
        //输出该字符串
        formatter.buf.write_str(*piece)?;
    }

    Ok(())
}
```

```
//非默认个数输出的格式化字符串输出函数
unsafe fn run(fmt: &mut Formatter<'_>, arg: &rt::v1::Argument, args: &
[ArgumentV1<'_>]) -> Result {
    //根据格式化参数的格式完成fmt的格式参数设置
    fmt.fill = arg.format.fill;
    fmt.align = arg.format.align;
    fmt.flags = arg.format.flags;
    unsafe {
        fmt.width = getcount(args, &arg.format.width);
        fmt.precision = getcount(args, &arg.format.precision);
    }

    debug_assert!(arg.position < args.len());
    //获取格式化参数
    let value = unsafe { args.get_unchecked(arg.position) };

    // 真正的进行格式化
    (value.formatter)(value.value, fmt)
}


impl<'a> Arguments<'a> {
    /// format_args!()完成字符串和参数解析后，如果都是默认格式，用下面的函数创建
    /// Arguments变量
    pub const fn new_v1(pieces: &'a [&'static str], args: &'a [ArgumentV1<'a>])
-> Arguments<'a> {
        if pieces.len() < args.len() || pieces.len() > args.len() + 1 {
            panic!("invalid args");
        }
        Arguments { pieces, fmt: None, args }
    }

    //format_args!()完成字符串和参数解析后，如果格式化格式不是默认格式，用下面的函数
创建Arguments
    pub const fn new_v1_formatted(
        pieces: &'a [&'static str],
        args: &'a [ArgumentV1<'a>],
        fmt: &'a [rt::v1::Argument],
        _unsafe_arg: UnsafeArg,
    ) -> Arguments<'a> {
        Arguments { pieces, fmt: Some(fmt), args }
    }

    //预估格式化后字符串长度
    pub fn estimated_capacity(&self) -> usize {
        //计算所有除格式化参数外的长度
        let pieces_length: usize = self.pieces.iter().map(|x| x.len()).sum();

        if self.args.is_empty() {
            pieces_length
```

```
        } else if !self.pieces.is_empty() && self.pieces[0].is_empty() &&
pieces_length < 16 {
            //如果字符串以格式化参数作为起始且除格式化以外的字符小于16
            0
        } else {
            //其他情况，为了防止额外申请堆内存，事先申请更多的内存
            pieces_length.checked_mul(2).unwrap_or(0)
        }
    }
}
```

以输出为二进制的isize的格式化为例，分析一下格式化具体的实现类型结构及方法： 首先，fmt::Binary负责二进制格式化trait

```
//对于不同进制的格式化实现trait
macro_rules! integer {
    ($Int:ident, $Uint:ident) => {
        int_base! { fmt::Binary   for $Int as $Uint  -> Binary }
        int_base! { fmt::Octal    for $Int as $Uint  -> Octal }
        int_base! { fmt::LowerHex for $Int as $Uint  -> LowerHex }
        int_base! { fmt::UpperHex for $Int as $Uint  -> UpperHex }

        int_base! { fmt::Binary   for $Uint as $Uint -> Binary }
        int_base! { fmt::Octal    for $Uint as $Uint -> Octal }
        int_base! { fmt::LowerHex for $Uint as $Uint -> LowerHex }
        int_base! { fmt::UpperHex for $Uint as $Uint -> UpperHex }
    };
}
//在isize,usize实现上述的格式化trait
integer! { isize, usize }

//int_base的宏定义
macro_rules! int_base {
    (fmt::$Trait:ident for $T:ident as $U:ident -> $Radix:ident) => {
        impl fmt::$Trait for $T {
            fn fmt(&self, f: &mut fmt::Formatter<'_>) -> fmt::Result {
                //具体的函数
                $Radix.fmt_int(*self as $U, f)
            }
        }
    };
}

//int_base宏中的$Radix的类型结构定义
struct Binary;

//fmt_int定义在下面的trait中
//此trait实现不同进制的整数的格式化通用操作
trait GenericRadix: Sized {
```

```
    ///进制
    const BASE: u8;

    /// 格式化的前缀字符串.
    const PREFIX: &'static str;

    /// x为十进制的数字，返回值是self进制的x的字符的编码数值
    fn digit(x: u8) -> u8;

    /// 将某一个数值按输入的格式化变量的要求进行格式化.
    fn fmt_int<T: DisplayInt>(&self, mut x: T, f: &mut fmt::Formatter<'_>) ->
fmt::Result {
        //首先获取足够的字符串空间来存放格式化后的内容
        //对于二进制，需要128个字节
        let zero = T::zero();
        let is_nonnegative = x >= zero;
        let mut buf = [MaybeUninit::<u8>::uninit(); 128];
        let mut curr = buf.len();
        //见后继DisplayInt的分析
        let base = T::from_u8(Self::BASE);
        if is_nonnegative {
            //从最低位到最高位填充buf
            for byte in buf.iter_mut().rev() {
                //余值填入当前的buf
                let n = x % base; // Get the current place value.
                //减掉已经填充的值
                x = x / base; // Deaccumulate the number.
                //将值转换为字符并写入buf
                byte.write(Self::digit(n.to_u8())); // Store the digit in the
buffer.
                curr -= 1;
                if x == zero {
                    // No more digits left to accumulate.
                    break;
                };
            }
        } else {
            //仍然从最低位到最高位
            for byte in buf.iter_mut().rev() {
                //获得当前位的值,负数
                let n = zero - (x % base); // Get the current place value.
                x = x / base; // Deaccumulate the number.
                byte.write(Self::digit(n.to_u8())); // Store the digit in the
buffer.
                curr -= 1;
                if x == zero {
                    // No more digits left to accumulate.
                    break;
                };
            }
        }
```

```
        //获取有意义的切片
        let buf = &buf[curr..];
        //生成utf-8字符串
        let buf = unsafe {
            str::from_utf8_unchecked(slice::from_raw_parts(
                MaybeUninit::slice_as_ptr(buf),
                buf.len(),
            ))
        };
        //Formatter会根据参数生成符合格式化的其他填充内容
        f.pad_integral(is_nonnegative, Self::PREFIX, buf)
    }
}

//以下为对isize及usize实现GenericRadix trait的代码
macro_rules! radix {
    ($T:ident, $base:expr, $prefix:expr, $($x:pat => $conv:expr),+) => {
        impl GenericRadix for $T {
            const BASE: u8 = $base;
            const PREFIX: &'static str = $prefix;
            fn digit(x: u8) -> u8 {
                match x {
                    $($x => $conv,)+
                    x => panic!("number not in the range 0..={}: {}",
Self::BASE - 1, x),
                }
            }
        }
    }
}

//这里只列出二进制，其他进制略
radix! { Binary,    2, "0b", x @  0 ..=  1 => b'0' + x }

//Formatter的其他方法如下：
impl<'a> Formatter<'a> {
    //对整形的格式化填充内容，在基础内容的基础上填充格式化需要的其他字符，完成对类型的
格式化输出
    pub fn pad_integral(&mut self, is_nonnegative: bool, prefix: &str, buf:
&str) -> Result {
        //获取基础内容字符串的长度
        //作为计算总长度的基础
        let mut width = buf.len();

        //是否需要正负符号
        let mut sign = None;
        if !is_nonnegative {
            //负数需要符号
            sign = Some('-');
            //输出的字符串长度+1
            width += 1;
```

```
        } else if self.sign_plus() {
            //格式化要求输出+号
            sign = Some('+');
            width += 1;
        }

        let prefix = if self.alternate() {
            //要求输出进制前缀
            width += prefix.chars().count();
            Some(prefix)
        } else {
            None
        };

        // 将符号及进制前缀输出
        fn write_prefix(f: &mut Formatter<'_>, sign: Option<char>, prefix:
Option<&str>) -> Result {
            if let Some(c) = sign {
                f.buf.write_char(c)?;
            }
            if let Some(prefix) = prefix { f.buf.write_str(prefix) } else {
Ok(()) }
        }

        match self.width {
            //格式化参数中没有对字宽有要求
            None => {
                //写入符号及前缀
                write_prefix(self, sign, prefix)?;
                //写入基本内容
                self.buf.write_str(buf)
            }
            //格式化参数有最小字宽要求,且当前字宽已经大于
            //最小字宽
            Some(min) if width >= min => {
                write_prefix(self, sign, prefix)?;
                self.buf.write_str(buf)
            }
            //格式化参数有最小字宽要求，当前字宽小于最小字宽
            //格式化参数规定填充0
            Some(min) if self.sign_aware_zero_pad() => {
                //不管输入的格式化参数中填充属性是什么，改变成为0
                //因为后继要恢复，replace恰如其分
                let old_fill = crate::mem::replace(&mut self.fill, '0');
                //不管输入的格式化参数中对齐属性是什么，改变成为右侧对齐
                let old_align = crate::mem::replace(&mut self.align,
rt::v1::Alignment::Right);
                //写入符号和前缀
                write_prefix(self, sign, prefix)?;
                //填充min-width个0，右侧对齐
                //如果随后还要pading,则在post_padding返回
```

```
                //padding方法见后继分析
                let post_padding = self.padding(min - width,
rt::v1::Alignment::Right)?;
                //写入基本内容
                self.buf.write_str(buf)?;
                //继续完成padding
                post_padding.write(self)?;
                //恢复格式化参数中填充属性及对齐属性内容
                self.fill = old_fill;
                self.align = old_align;
                Ok(())
            }
            // 格式化有最小字宽要求，当前字宽小于最小字宽，
            // 填充为空
            Some(min) => {
                //先进性填充
                let post_padding = self.padding(min - width,
rt::v1::Alignment::Right)?;
                //写入符号及前缀
                write_prefix(self, sign, prefix)?;
                //写入基本内容
                self.buf.write_str(buf)?;
                //继续完成padding
                post_padding.write(self)
            }
        }
    }

    //完成格式化中的填充功能
    pub(crate) fn padding(
        &mut self,
        padding: usize,
        default: rt::v1::Alignment,
    ) -> result::Result<PostPadding, Error> {
        let align = match self.align {
            rt::v1::Alignment::Unknown => default,
            _ => self.align,
        };

        //确定基础内容之前padding和之后padding的字符数目
        let (pre_pad, post_pad) = match align {
            rt::v1::Alignment::Left => (0, padding),
            rt::v1::Alignment::Right | rt::v1::Alignment::Unknown => (padding,
0),
            rt::v1::Alignment::Center => (padding / 2, (padding + 1) / 2),
        };

        //完成基础内容之前的padding输出
        for _ in 0..pre_pad {
            self.buf.write_char(self.fill)?;
        }
```

```
        //创建基础内容之后padding的结构
        Ok(PostPadding::new(self.fill, post_pad))
    }

}

//PostPadding实现，在格式化内容基础内容之后进行填充
pub(crate) struct PostPadding {
    //填充字符
    fill: char,
    //填充字符数目
    padding: usize,
}

impl PostPadding {
    fn new(fill: char, padding: usize) -> PostPadding {
        PostPadding { fill, padding }
    }

    pub(crate) fn write(self, f: &mut Formatter<'_>) -> Result {
        //输出padding内容
        for _ in 0..self.padding {
            f.buf.write_char(self.fill)?;
        }
        Ok(())
    }
}
```

以上是格式化输出的代码基本脉络，格式化输出还有很多其他代码，请读者参考这个脉络自行研究

对输出做格式化是非常复杂的可变参数支持的例子。从对以上的代码分析，在RUST支持可变参数的途径：

1. 首先定义一个支持可变参数的宏，例如foramt_args宏，这个宏将可变参数转变成一个数据结构，数据结构需要根据需要进行设计。
2. 根据数据结构设计方法或函数。

Vec中的vec！宏也是一个典型的可变参数实现，但其用途较单纯，因此也非常简单。可变参数是非常具有直观性及方便的语法。写一些库的时候需要经常用到。