RUST的固有（intrinsic）函数库

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

intrinsic库函数是指由编译器内置实现的函数，一般如下特点的函数用固有函数：

1. 与CPU架构相关性很大，必须利用汇编实现或者利用汇编才能具备最高性能的函数，
2. 和编译器密切相关的函数，由编译器来实现最为合适。
   上面内存库章节中已经介绍了内存部分的intrinsic库函数，本节对其他部分做简略介绍

intrinsic 原子操作函数

原子操作函数主要用于多核CPU，多线程CPU时对数据的原子操作。intrinsic库中atomic_xxx及atomic_xxx_xxx类型的函数即为原子操作函数。原子操作函数主要用于并发编程中做临界保护，并且是其他临界保护机制的基础，如Mutex，RWlock等。

数学函数及位操作函数

各种整数及浮点的数学函数实现。这一部分放在intrinsic主要是因为现代CPU对浮点计算由很多支持，这些数学函数由汇编语言来实现更具备效率，那就有必要由编译器来内置实现。

intrinsic 指令优化及调试函数

断言类: assert_xxxx 类型的函数
函数栈：caller_location

小结

intrinsic函数库是从编译器层面完成跨CPU架构的一个手段，intrinsic通常被上层的库所封装。但在操作系统编程和框架编程时，仍然会不可避免的需要接触。

RUST基本类型代码分析(一)

原生数据类型，Option类型，Result类型的某些代码是分析其他模块的基础，因此先对这些类型的部分代码做个基础分析。

整形数据类型

代码目录如下： %USER%.rustup\toolchains\nightly-x86_64-pc-windows-msvc\lib\rustlib\src\rust\library\core\src\num

整形数据类型标准库是在整形类型上实现一系列方法和trait。对整形类型标准库分析的主要目的是:

1. 更好的了解RUST是如何从细节上来保证代码安全。
2. RUST将整形数学库及一些其他常用函数纳入了整形类型的方法中，这与其他语言不同，需要了解。

整形数据的方法主要有:

1. 整形位操作：左移，右移，为1的位数目，为0的位数目，头部为0的位数目，尾部为0的位数目，头部为1的位数目，尾部为1的位数目，循环左移，循环右移
2. 整形字节序操作：字节序反转，位序反转，大小端变换
3. 整形数学函数：针对溢出做各种不同处理的加减乘除，传统的整形数学库函数如对数，幂，绝对值，取两者大值及两者小值

整形有有符号整形，无符号整形，大整形(大于计算机字长的整形)，但基本内容都是实现以上方法

无符号整形类型相关库代码分析

标准库用宏简化的对不同位长的无符号整形的方法实现。本文着重介绍若干不易注意的方法，如大小端转换，对数学方法仅给出加法做为代表。代码如下:

```
macro_rules! uint_impl {
    ($SelfT:ty, $ActualT:ident, $SignedT:ident, $BITS:expr, $MaxV:expr,
        //以下主要是rust doc文档需要
        $rot:expr, $rot_op:expr, $rot_result:expr, $swap_op:expr,
$swapped:expr,
        $reversed:expr, $le_bytes:expr, $be_bytes:expr,
        $to_xe_bytes_doc:expr, $from_xe_bytes_doc:expr) => {
```

这个宏实现所有无符号整形的方法:

```
        pub const MIN: Self = 0;
        //按位非
        pub const MAX: Self = !0;

        pub const BITS: u32 = $BITS;
```

以上是无符号整形的常量

```
//利用intrinsics的位函数完成整数的位操作相关函数，
//这里仅分析一个，其他请参考标准库手册
pub const fn count_ones(self) -> u32 {
    intrinsics::ctpop(self as $ActualT) as u32
}

//其他位操作函数
...
...
```

字节序变换是网络编程与结构化数据文件的必须功能，RUST将之在整形的方法里实现：

```
//变量内存空间的字节序交换
pub const fn swap_bytes(self) -> Self {
    intrinsics::bswap(self as $ActualT) as Self
}

//big endian 到硬件架构字节序
pub const fn from_be(x: Self) -> Self {
    #[cfg(target_endian = "big")]
    {
        x
    }
    #[cfg(not(target_endian = "big"))]
    {
        x.swap_bytes()
    }
}

//little endian 转换为硬件架构字节序
pub const fn from_le(x: Self) -> Self {
    #[cfg(target_endian = "little")]
    {
        x
    }
    #[cfg(not(target_endian = "little"))]
    {
        x.swap_bytes()
    }
}

//硬件架构字节序到big endian
pub const fn to_be(self) -> Self { // or not to be?
    #[cfg(target_endian = "big")]
    {
        self
    }
```

```
            #[cfg(not(target_endian = "big"))]
            {
                self.swap_bytes()
            }
        }

        //硬件架构字节序到little endian
        pub const fn to_le(self) -> Self {
            #[cfg(target_endian = "little")]
            {
                self
            }
            #[cfg(not(target_endian = "little"))]
            {
                self.swap_bytes()
            }
        }

        //获得大端字节序字节数组
        pub const fn to_be_bytes(self) -> [u8; mem::size_of::<Self>()] {
            self.to_be().to_ne_bytes()
        }

        //获得小端
        pub const fn to_le_bytes(self) -> [u8; mem::size_of::<Self>()] {
            self.to_le().to_ne_bytes()
        }

        //硬件平台字节序
        pub const fn to_ne_bytes(self) -> [u8; mem::size_of::<Self>()] {
            unsafe { mem::transmute(self) }
        }

        //从big endian 字节数组获得类型值
        pub const fn from_be_bytes(bytes: [u8; mem::size_of::<Self>()]) -> Self
{
            Self::from_be(Self::from_ne_bytes(bytes))
        }

        //从little endian 字节数组获得类型值
        pub const fn from_le_bytes(bytes: [u8; mem::size_of::<Self>()]) -> Self
{
            Self::from_le(Self::from_ne_bytes(bytes))
        }

        //从硬件架构字节序字节数组获得类型值
        pub const fn from_ne_bytes(bytes: [u8; mem::size_of::<Self>()]) -> Self
{
            unsafe { mem::transmute(bytes) }
        }
```

RUST的整数类形各种算术方法突出的展示了RUST对安全的极致关注。算术方法也更好的支持了链式调用的函数式编程风格。对于算术溢出，RUST给出了各种情况下的处理方案：

```
        //对溢出做检查的加法运算,溢出情况下会返回wrapping_add的值，即溢出后值回绕
        //这里每种类型运算都以加法为例，其他诸如减、乘、除、幂次请参考官方标准库手册
        pub const fn overflowing_add(self, rhs: Self) -> (Self, bool) {
            let (a, b) = intrinsics::add_with_overflow(self as $ActualT, rhs as
$ActualT);
            (a as Self, b)
        }
        //其他的对溢出做检查的算数运算，略
        ...
        ...

        //溢出后对最大值取余，即回绕
        pub const fn wrapping_add(self, rhs: Self) -> Self {
            intrinsics::wrapping_add(self, rhs)
        }
        //以边界值取余的其他数学运算方法，略
        ...
        ...

        //饱和加法，超过边界值结果为边界值
        pub const fn saturating_add(self, rhs: Self) -> Self {
            intrinsics::saturating_add(self, rhs)
        }
        //其他饱和型的数学运算，略
        ...
        ...

        //对加法有效性检查的加法运算，如发生溢出，则返回异常
        pub const fn checked_add(self, rhs: Self) -> Option<Self> {
            let (a, b) = self.overflowing_add(rhs);
            if unlikely!(b) {None} else {Some(a)}
        }

        //无检查add，  是 + 符号的默认调用函数。
        pub const unsafe fn unchecked_add(self, rhs: Self) -> Self {
            // 调用者要保证不发生错误
            unsafe { intrinsics::unchecked_add(self, rhs) }
        }
        //其他对有效性检查的数学运算，  略
        ...
        ...

        pub const fn min_value() -> Self { Self::MIN }

        pub const fn max_value() -> Self { Self::MAX }
}
```

算术算法基本上是使用了intrinsics提供的函数。

下面用u8给出一个上述宏具体的实例

```
impl u8 {
    //利用宏实现 u8类型的方法
    uint_impl! { u8, u8, i8, 8, 255, 2, "0x82", "0xa", "0x12", "0x12", "0x48",
"[0x12]",
    "[0x12]", "", "" }

    pub const fn is_ascii(&self) -> bool {
        *self & 128 == 0
    }

    //其他ASCII相关函数，请参考标准库手册，略
    ...
    ...
}

//u16 实现
impl u16 {
    uint_impl! { u16, u16, i16, 16, 65535, 4, "0xa003", "0x3a", "0x1234",
"0x3412", "0x2c48",
    "[0x34, 0x12]", "[0x12, 0x34]", "", "" }
    widening_impl! { u16, u32, 16, unsigned }
}

//其他无符号整形的实现，略
...
...
```

RUST整形库代码逻辑并不复杂，宏也很简单。但因为RUST将其他语言的独立的数学库函数，单独的大小端变换等集成入整形(浮点类型)，有可能造成出于习惯而无法找到相应的函数。

浮点类型

本节主要说明RUST的数学库所在位置。 代码目录如下：
%USER%.rustup\toolchains\nightly-x86_64-pc-windows-msvc\lib\rustlib\src\rust\library\std\src\f32.rs

core库中不包含更多的数学函数，因此用了std的的f32的实现

```
impl f32 {
    ...
    ...
    pub fn abs(self) -> f32 {
```

```
        unsafe { intrinsics::fabsf32(self) }
    }

    pub fn signum(self) -> f32 {
        if self.is_nan() { Self::NAN } else { 1.0_f32.copysign(self) }
    }

    pub fn copysign(self, sign: f32) -> f32 {
        unsafe { intrinsics::copysignf32(self, sign) }
    }

    pub fn powf(self, n: f32) -> f32 {
        unsafe { intrinsics::powf32(self, n) }
    }

    pub fn sqrt(self) -> f32 {
        unsafe { intrinsics::sqrtf32(self) }
    }

    pub fn exp(self) -> f32 {
        unsafe { intrinsics::expf32(self) }
    }

    pub fn exp2(self) -> f32 {
        unsafe { intrinsics::exp2f32(self) }
    }

    pub fn sin(self) -> f32 {
        unsafe { intrinsics::sinf32(self) }
    }

    pub fn cos(self) -> f32 {
        unsafe { intrinsics::cosf32(self) }
    }

    pub fn tan(self) -> f32 {
        unsafe { cmath::tanf(self) }
    }

    pub fn asin(self) -> f32 {
        unsafe { cmath::asinf(self) }
    }

    pub fn acos(self) -> f32 {
        unsafe { cmath::acosf(self) }
    }

    pub fn atan(self) -> f32 {
        unsafe { cmath::atanf(self) }
    }
```

```
    pub fn atan2(self, other: f32) -> f32 {
        unsafe { cmath::atan2f(self, other) }
    }

    pub fn sin_cos(self) -> (f32, f32) {
        (self.sin(), self.cos())
    }
    ...
    ...
}
```

RUST将数学函数与浮点类型关联在一起，除了更好的模块性以外，应该更多的出于支持函数式编程中的链式调用为目的。

RUST Option类型标准库代码分析

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

Option虽然在RUST中具有重要地位，但它本身不是RUST语法的最底层。实际上，可以认为它只是RUST的一个很普通的类型。

```
pub enum Option<T> {
    None,
    Some(T),
}
```

借用RUST的enum语法，RUST标准库定义`Option<T>`，并将其作为解决某一变量不存在有效值的标准化解决方案。但一定要明确的是，`Option<T>`的方案是可以根据情况来作选择的，RUST的程序员完全可以根据情况另外构建合适的定制方案。最关键是`Option<T>`这种解决问题的思考方法。

很多语言通常把类型的取值域中的某一个值设计成代表值不存在,这就给bug开了一个口子，检查类型值是否存在成了程序员的责任，虽然这已经被程序员所接受并视作自己的能力之一，且无数家公司的编程规范也规定了相关内容。但RUST用`Option<T>`说了不，有了`Option<T>`后，RUST编译器承担起了类型值检查的责任，程序的正确性得到了更好的保证。由此可见，`Option<T>`对安全的保证仍然是靠基础架构代码而不是靠编译器。

`Option<T>`也提供了变量声明时无法初始化的另一个方案：
在初始化时无法确定T类型的值时，除了`MaybeUninit<T>`外，还可以用`Option<T>`来声明变量并初始化为None。

`Option<T>`是对T类型变量的封装，在使用的时候会带来一些不便，针对这点，`Option<T>`提供了很酷的打开方式：用以map为代表的方法来完成函数链式调用。当然，Try trait及各种解封装方法也极大的方便了编程。

`Option<T>`创建： 直接用`Some(val)`做包装，或者直接使用`None`。

`Option<T>`的指针获取方法源代码如下： RUST的习惯是每个复合类型都要有`as_ref/as_mut/as_ptr/as_mut_ptr`来获取"不可变引用/可变引用/不可变裸指针/可变裸指针"，每个复合类型可以根据自己的需求来实现这些方法，`Option<T>`没有实现裸指针相关内容

```
impl<T> Option<T> {
    //根据Option<T>自身的设计，只能返回Option<&T>
    pub const fn as_ref(&self) -> Option<&T> {
        match *self {
            Some(ref x) => Some(x),
            None => None,
        }
    }

    //类似于as_ref，但返回的是可变引用
    pub const fn as_mut(&mut self) -> Option<&mut T> {
        //略
    }
```

对于所有的封装结构类型，如何方便的解封装都是重要的内容，RUST往往利用Try trait(后文有详述)及闭包来获得更精炼的解封装代码实践。 以下解封装函数，看过源码后功能即一目了然，不同封装结构的解封装方法功能都类似，可以从`Option<T>`对这些功能做出总结。

```
    //解封装函数，正常返回封装中的变量，异常输出期待的错误消息
    pub fn expect(self, msg: &str) -> T {
        match self {
            Some(val) => val,
            None => expect_failed(msg),
        }
    }

    //解封装函数，正常返回封装中的变量，异常触发panic
    pub const fn unwrap(self) -> T {
        match self {
            Some(val) => val,
            None => panic!("called `Option::unwrap()` on a `None` value"),
        }
    }

    //解封装，正常返回封装中变量，异常返回变量默认值
    pub fn unwrap_or(self, default: T) -> T {
        match self {
            Some(x) => x,
            None => default,
        }
```

```
    }

    //解封装，正常返回封装中变量，异常执行闭包并返回闭包返回值
    pub fn unwrap_or_else<F: FnOnce() -> T>(self, f: F) -> T {
        match self {
            Some(x) => x,
            None => f(),
        }
    }

    //确认不会异常的解封装函数
    pub unsafe fn unwrap_unchecked(self) -> T {
        debug_assert!(self.is_some());
        match self {
            Some(val) => val,
            // SAFETY: the safety contract must be upheld by the caller.
            None => unsafe { hint::unreachable_unchecked() },
        }
    }
```

针对函数式编程的链式调用设计的方法：

```
    //主要用于函数式编程，map即是对值集合中的每个值作为闭包输入变量，并输出闭包输出
    // Option<T>的map对异常不处理
    pub fn map<U, F: FnOnce(T) -> U>(self, f: F) -> Option<U> {
        match self {
            Some(x) => Some(f(x)),
            None => None,
        }
    }

    //正常时将变量输入闭包，返回闭包返回值，异常则返回默认值
    pub fn map_or<U, F: FnOnce(T) -> U>(self, default: U, f: F) -> U {
        match self {
            Some(t) => f(t),
            None => default,
        }
    }

    //正常将变量输入闭包，返回闭包返回值，异常返回另一闭包返回值
    pub fn map_or_else<U, D: FnOnce() -> U, F: FnOnce(T) -> U>(self, default:
D, f: F) -> U {
        match self {
            Some(t) => f(t),
            None => default(),
        }
    }

    //将Option转换为Result，也是为支持函数式编程
    pub fn ok_or<E>(self, err: E) -> Result<T, E> {
```

```
        match self {
            Some(v) => Ok(v),
            None => Err(err),
        }
    }

    //同上，None时调用默认函数处理
    pub fn ok_or_else<E, F: FnOnce() -> E>(self, err: F) -> Result<T, E> {
        match self {
            Some(v) => Ok(v),
            None => Err(err()),
        }
    }

    //Option<T>的与运算，正常返回输入参数，异常返回None
    pub fn and<U>(self, optb: Option<U>) -> Option<U> {
        match self {
            Some(_) => optb,
            None => None,
        }
    }

    //主要用于函数式编程，与and 形成系列，值为Some(x)调用函数并返回函数值
    pub fn and_then<U, F: FnOnce(T) -> Option<U>>(self, f: F) -> Option<U> {
        match self {
            Some(x) => f(x),
            None => None,
        }
    }

    //如果是Some(x)，判断是否满足预设条件
    pub fn filter<P: FnOnce(&T) -> bool>(self, predicate: P) -> Self {
        if let Some(x) = self {
            if predicate(&x) {
                return Some(x);
            }
        }
        None
    }

    //如果是Some(x)返回本身，如果是None，返回预设值
    pub fn or(self, optb: Option<T>) -> Option<T> {
        match self {
            Some(_) => self,
            None => optb,
        }
    }

    //如果是Some(x)返回本身，否则返回预设函数
    pub fn or_else<F: FnOnce() -> Option<T>>(self, f: F) -> Option<T> {
        match self {
```

```
            Some(_) => self,
            None => f(),
        }
    }

    //类似xor操作
    pub fn xor(self, optb: Option<T>) -> Option<T> {
        match (self, optb) {
            //一方为Some,一方为None，返回Some值
            (Some(a), None) => Some(a),
            (None, Some(b)) => Some(b),
            //两者都为Some，或两者都为None，返回None
            _ => None,
        }
    }
```

其他方法

```
    //不解封装的重新设置内部的值，并返回值的可变引用
    //例子：let a = None; a.insert(1);
    //上例也是一种常用方法，利用None可以实现不知道初始值但需要有一个变量的情况。
    pub fn insert(&mut self, value: T) -> &mut T {
        //原有*self会被drop
        *self = Some(value);
        //确认不会为None
        unsafe { self.as_mut().unwrap_unchecked() }
    }

    //使用一个闭包生成变量
    pub fn get_or_insert_with<F: FnOnce() -> T>(&mut self, f: F) -> &mut T {
        if let None = *self {
            *self = Some(f());
        }

        match self {
            //此处RUST专门设计了针对引用的match语法
            //如果仅仅依照普通的语法来分析，此处是有问题的，具体见此节后的分析。
            Some(v) => v,
            None => unsafe { hint::unreachable_unchecked() },
        }
    }

    //针对Option的zip操作
    pub fn zip<U>(self, other: Option<U>) -> Option<(T, U)> {
        match (self, other) {
            (Some(a), Some(b)) => Some((a, b)),
            _ => None,
        }
    }
```

```
    //执行一个函数
    pub fn zip_with<U, F, R>(self, other: Option<U>, f: F) -> Option<R>
    where
        F: FnOnce(T, U) -> R,
    {
        //此处，顺序应该是先执行self? other? ，然后再调用函数
        Some(f(self?, other?))
    }
```

下面的take及replace对于`Option<T>`非常重要，`Option<T>`多用于包装引用或者智能指针且作为结构体成员。因为在使用结构体引用时，无法单独的转移结构体成员的所有权，而经常需要在这种情况下对`Option<T>`成员的值作改动，此时便只能用take来获取所有权，修改后再用replace将值更新。

```
    //mem::replace分析请参考前文，用None替换原来的变量，并用新变量返回self，同时也完成了所有权的转移
    pub const fn take(&mut self) -> Option<T> {
        mem::replace(self, None)
    }O

    //用新value替换原变量，并把原变量返回
    pub const fn replace(&mut self, value: T) -> Option<T> {
        mem::replace(self, Some(value))
    }

}
```

`Option<T>`的take及replace组合因为引入两次拷贝，降低效率。所以当采用这种形式作更新方案时，要考虑是否可以用unsafe的方式得到性能更高的方案。

对结构体引用类型`&T/&mut T`的match语法研究

上节的代码中：

```
    pub fn get_or_insert_with<F: FnOnce() -> T>(&mut self, f: F) -> &mut T {
        ...
        match self {
            //这里因为没有自动解引用，所以self应该是&Some(T)的类型
            //这就和下面的Some(v)出现冲突了，&Some(T)怎么可能变成Some(v)
            //而v 为什么会是一个引用变量。这是一般的match无法解释的。
            //按照理解中的语法，应该是
            //&Some(ref v) => v
            Some(v) => v,
            None => unsafe { hint::unreachable_unchecked() },
        }
    }
```

更清晰的，再请参考如下代码：

```
struct TestStructA {a:i32,b:i32}
fn main() {
    let c = TestStructA{a:1, b:2};
    let d = [1,2,3];

    match ((&c, &d)) {
        //用&TestStructA表示c的类型，&[]表示切片类型，可以内部绑定引用或Copy trait变量
        (&TestStructA{a:ref u, b:w}, &[ref x, y, ..]) => println!("{} {} {} {}", *u, w, *x, y),
        _  => println!("match nothing"),
    }
}
```

上面代码的match是按照正常理解思路的一个写法，对结构内部的变量需要用引用绑定来获取，但结构内部变量如果实现Copy Trait，那可以不用引用绑定。但如果结构内部变量没有实现Copy，则必须使用引用，否则会因为错误的所有权转移导致编译器告警。

为了编码上的方便，RUST针对引用绑定的代码，支持如下简化形式：

```
struct TestStructA {a:i32,b:i32}
fn main() {
    let c = TestStructA{a:1, b:2};
    let d = [1, 2, 3];

    match ((&c, &d)) {
        //对比上述代码，头部少了&，模式绑定内部少了 ref，但代码功能完全一致，但这个代码不支持Copy trait的变量绑定了。
        (TestStructA{a: u}, [x,..]) => println!("{} {}", *u, *x),
        _  => println!("match nothing"),
    }
}
```

这是RUST的标准写法，但如果不知道RUST为这个语义专门做了语法设计，很可能会对这里的类型绑定感到疑惑。 从实际的使用场景分析，对结构体引用做match，其目的就是对结构体内部的成员的引用做pattern绑定。而且如果结构体内部的成员不支持Copy，那也不可能对结构体成员做pattern绑定。所以，此语法也是在RUST的所有权定义下的一个必然的简化选择。

RUST Result类型标准库代码分析

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

`Result<T,E>`实际是针对其他语言的try...catch...的对应设计。try...catch...试图简化方法/函数返回值错误处理。但仍然显得繁琐及影响代码的阅读体验，同时使用了复杂的实现机制。`Result<T,E>`与?运算符的配合使得代码对错误处理实现了最简练化，代码的主体功能一目了然。实现上仅仅使用了语法规则做了代码简化，但机制是最普通的函数返回。且可以由出错代码封装具体的变量，可以按需要定制获得具体的错误信息如变量，描述，函数，文件行等，提供了比try...catch...更强一筹的错误处理手段。

`Result<T,E>`的Try trait十分重要，另外，以map为代表的函数同样打开函数链式调用的通道。
`Result<T,E>`值得关注方法的源代码如下：

```
pub enum Result<T, E> {
    /// Contains the success value
    Ok(T),

    /// Contains the error value
    Err(E),
}
impl<T, E> Result<T, E> {

    //应用于函数式编程，如果是Ok，利用闭包直接处理Result值，返回需要的新Result类型
    pub fn map<U, F: FnOnce(T) -> U>(self, op: F) -> Result<U, E> {
        match self {
            Ok(t) => Ok(op(t)),
            Err(e) => Err(e),
        }
    }

    //如果是Ok，利用闭包处理Result值，返回需要的类型，如果是Err返回默认值
    pub fn map_or<U, F: FnOnce(T) -> U>(self, default: U, f: F) -> U {
        match self {
            Ok(t) => f(t),
            Err(_) => default,
        }
    }

    //如果是Ok，调用闭包处理Result，返回需要的类型，如果是Err，调用错误闭包函数处理错误
    pub fn map_or_else<U, D: FnOnce(E) -> U, F: FnOnce(T) -> U>(self, default: D, f: F) -> U {
        match self {
            Ok(t) => f(t),
            Err(e) => default(e),
        }
    }
```

```
    //如果是Err，调用闭包函数处理错误，返回需要的类型， Ok则返回原值
    pub fn map_err<F, O: FnOnce(E) -> F>(self, op: O) -> Result<T, F> {
        match self {
            Ok(t) => Ok(t),
            Err(e) => Err(op(e)),
        }
    }

    //Result传递，Ok则返回给定的Result类型值，否则返回原值
    pub fn and<U>(self, res: Result<U, E>) -> Result<U, E> {
        match self {
            Ok(_) => res,
            Err(e) => Err(e),
        }
    }

    //Ok 则调用闭包处理,返回需要的Result类型值， 否则返回原值
    pub fn and_then<U, F: FnOnce(T) -> Result<U, E>>(self, op: F) -> Result<U,
E> {
        match self {
            Ok(t) => op(t),
            Err(e) => Err(e),
        }
    }

    //Ok返回原值，Err返回传入的默认Result类型值
    pub fn or<F>(self, res: Result<T, F>) -> Result<T, F> {
        match self {
            Ok(v) => Ok(v),
            Err(_) => res,
        }
    }

    //Ok返回原值，Err调用函数进行处理，返回需要的Result类型值
    pub fn or_else<F, O: FnOnce(E) -> Result<T, F>>(self, op: O) -> Result<T,
F> {
        match self {
            Ok(t) => Ok(t),
            Err(e) => op(e),
        }
    }

    //解封装，Ok返回封装内的值，Err返回默认值
    pub fn unwrap_or(self, default: T) -> T {
        match self {
            Ok(t) => t,
            Err(_) => default,
        }
    }

    //解封装， Ok返回封装内的值， Err调用处理函数处理
```

```
    pub fn unwrap_or_else<F: FnOnce(E) -> T>(self, op: F) -> T {
        match self {
            Ok(t) => t,
            Err(e) => op(e),
        }
    }

    //确认返回一定是Ok时的解封装函数
    pub unsafe fn unwrap_unchecked(self) -> T {
        debug_assert!(self.is_ok());
        match self {
            Ok(t) => t,
            // SAFETY: the safety contract must be upheld by the caller.
            Err(_) => unsafe { hint::unreachable_unchecked() },
        }
    }

    //确认返回一定是Err时调用的解封装函数
    pub unsafe fn unwrap_err_unchecked(self) -> E {
        debug_assert!(self.is_err());
        match self {
            // SAFETY: the safety contract must be upheld by the caller.
            Ok(_) => unsafe { hint::unreachable_unchecked() },
            Err(e) => e,
        }
    }
}
```

`Result<T,E>`的解封装函数如下：

```
impl<T, E: fmt::Debug> Result<T, E> {
    //典型的expect解封装方法，内容略
    pub fn expect(self, msg: &str) -> T ;

    //典型的unwrap解封装方法，内容略
    pub fn unwrap(self) -> T ;
}

impl<T: fmt::Debug, E> Result<T, E> {
    //解封装，对于Ok输出参数指定的信息并退出，Err解封装
    pub fn expect_err(self, msg: &str) -> E {
        match self {
            Ok(t) => unwrap_failed(msg, &t),
            Err(e) => e,
        }
    }

    //解封装，对于Ok输出固定的信息并退出，Err解封装
    pub fn unwrap_err(self) -> E {
        match self {
```

```
            Ok(t) => unwrap_failed("called `Result::unwrap_err()` on an `Ok`
value", &t),
            Err(e) => e,
        }
    }
}


impl<T: Default, E> Result<T, E> {
    //解封装，Ok解封装， Err返回T的Default值
    pub fn unwrap_or_default(self) -> T {
        match self {
            Ok(x) => x,
            Err(_) => Default::default(),
        }
    }
}

impl<T, E: Into<!>> Result<T, E> {
    //解封装，Ok解封装，Err返回Never类型
    pub fn into_ok(self) -> T {
        match self {
            Ok(x) => x,
            Err(e) => e.into(),
        }
    }
}

impl<T: Into<!>, E> Result<T, E> {
    //解封装，Err解封装， Ok返回Never类型
    pub fn into_err(self) -> E {
        match self {
            Ok(x) => x.into(),
            Err(e) => e,
        }
    }
}

impl<T, E> Result<Option<T>, E> {
    //将Result<>转换为Option
    pub const fn transpose(self) -> Option<Result<T, E>> {
        match self {
            Ok(Some(x)) => Some(Ok(x)),
            Ok(None) => None,
            Err(e) => Some(Err(e)),
        }

    }
}
```