RUST标准库的基础Trait

编译器内置Trait代码分析

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

marker trait是没有实现体，是一种特殊的类型性质，这类性质无法用类型成员来表达，因此用trait来实现是最合适的。

Send trait是标识变量类型可以安全的在线程间转移所有权的marker。
Sync trait是标识变量类型的引用可以安全的由多线程并发访问的marker。 对于auto trait，RUST的默认规则是如果T支持auto trait，则`*const T, *mut T, [T], [T;N], &T, &mut T`都自动支持该auto trait, 如果实际情况不符合这条默认规则，需要在代码中显式声明。
如果一个复合类型的所有成员都支持auto trait, 则该复合类型支持auto trait。如果实际上不符合这条默认规则，也需要显式在代码作声明。
如果一个符合类型中有成员不支持auto trait, 但该复合类型支持auto trait, 需要在代码中作显式声明。

变量在线程间安全指的是对变量操作需要具备事务性，在一个事务周期内只允许一个线程对变量进行读写。
RUST中，因为所有权和借用语法，对于大部分类型，不会出现多线程的并发操作。因此RUST类型默认都是实现了Send trait和Sync trait的，如果类型不支持Send或Sync或者有约束条件，需要进行显式的定义。 在所有权转移后可能出现多线程并发操作的基本类型只有内部可变性类型的引用，裸指针，多份所有权的智能指针三种情况。 目前可以多线程并发操作的类型有：

1. 内部可变性类型引用，
2. 具有递归到内部可变性类型成员的复合类型变量引用, 并且利用变量引用可触发内部可变操作
3. 具有递归到内部可变性引用类型成员的复合类型变量及变量引用，并且利用变量或变量引用可触发内部可变操作
4. `*const T/* mut T/NonNull<T>/Unique<T>`及引用，
5. 具有递归到4类型成员的复合类型变量及变量引用，并且利用变量及变量引用可触发对裸指针指向内容的改变
6. 支持多份所有权的智能指针，
7. 能够递归到多份所有权智能指针类型成员的复合类型变量及其引用，并且利用变量及变量引用可触发对智能指针操作。

8. 对引用转换为裸指针后进行unsafe的操作
   unsafe操作我们不做讨论。
   对于上述1-6，都需要进行明确Send trait及Sync Trait的定义。 这里要注意的是RUST结构体内的成员默认私有，所以即使类型结构体有能多线程并发操作的成员，也不代表类型本身就能够被多线程并发操作。例如：只有当取得结构类型可变引用时，才能对内部可变性成员操作，就使得结构类型可以成为支持Send和Sync的类型。 这些情况下，需要做Send和Sync的显式声明。

```
pub unsafe auto trait Send {
    // empty.
}

//以下因为与RUST的默认规则不一样，所以需要在代码中显式定义出来
impl<T: ?Sized> !Send for *const T {}
impl<T: ?Sized> !Send for *mut T {}

//以下也是因为与RUST的默认规则不一致，所以需要显式的声明
mod impls {
    // 如果将&T转移到其他线程，不支持Sync trait的类型利用&T对变量的操作导致事务性不能保证。所以不能将&T转移到其他线程，
    unsafe impl<T: Sync + ?Sized> Send for &T {}
    // &mut T具备独占性，导致线程外不会有其他写操作，包括内部可变性类型。
    unsafe impl<T: Send + ?Sized> Send for &mut T {}
}


pub unsafe auto trait Sync {
    // Empty
}
//与RUST代码默认规则不一致
impl<T: ?Sized> !Sync for *const T {}
impl<T: ?Sized> !Sync for *mut T {}

//类型内存大小固定，泛型 "T" 默认是Sized，如果表示所有类型，需要 T:?Sized.
pub trait Sized {
    // Empty.
}

//如果一个Sized的类型要强制转换为动态大小类型，那必须实现Unsize Trait
//例如 [T;N] 实现了 Unsize<[T]>
pub trait Unsize<T: ?Sized> {
    // Empty.
}

//模式匹配表达式匹配时编译器需要使用的Trait，如果一个结构实现了PartialEq，该Trait会自动被实现。
pub trait StructuralPartialEq {
    // Empty.
```

```
}

//主要用于模式匹配，如果一个结构实现了Eq，该Trait会自动被实现。
pub trait StructuralEq {
    // Empty.
}
```

以下给出了一个针对所有的原生类型都实现Copy Trait的实现代码, 实现了Copy Trait的类型编译器不必调用drop来对类型进行内存释放。 这也是RUST针对原生类型可以直接实现trait的实例。任意模块可以定义一个trait,然后即可在原生类型上实现这个trait，这极大的提高了RUST的语法一致性及函数式编程的能力：

```
//Copy，略
pub trait Copy: Clone {
    // Empty.
}

//统一实现原生类型对Copy Trait的支持
mod copy_impls {

    use super::Copy;

    macro_rules! impl_copy {
        ($($t:ty)*) => {
            $(
                impl Copy for $t {}
            )*
        }
    }

    impl_copy! {
        usize u8 u16 u32 u64 u128
        isize i8 i16 i32 i64 i128
        f32 f64
        bool char
    }

    impl Copy for ! {}

    impl<T: ?Sized> Copy for *const T {}

    impl<T: ?Sized> Copy for *mut T {}

    impl<T: ?Sized> Copy for &T {}

    //& mut T不支持Copy，以保证RUST的借用规则
}
```

`PhantomData<T>`类型可以在其他类型结构体中定义一个变量，标记此结构体逻辑上拥有，但不需要或不方便在结构体成员变量体现的某个属性。实质上，智能指针一般都需要利用`Unique<T>`，以PhantomData来实现对堆内存的逻辑拥有权. PhantomData最常用来标记生命周期及所有权。主要给编译器提示检验类型变量的生命周期和类型构造时输入的生命周期关系。也用来提示拥有PhantomData的结构体会负责对T做drop操作。需要编译器做drop检查的时候更准确的判断出内存安全错误。 `PhantomData<T>`属性与所有权或生命周期的关系由编译器自行推断。具体实例可参考官方标准库文档及后继相关章节。
PhantomData是个单元结构体，单元结构体的变量名就是单元结构体的类型名。
所以使用的时候直接使用PhantomData即可，编译器会将泛型的类型实例化信息自动带入PhantomData中

```
pub struct PhantomData<T: ?Sized>;
```

ops 运算符 Trait 代码分析

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

RUST中，所有的运算符号都可以重载。Ops重载允许提供*两个不同类型*之间的运算。

一个小规则

在重载函数中，如果重载的符号出現，编译器用规定的默认操作来实现。例如：

```
        impl const BitAnd for u8 {
            type Output = u8;
            //下面函数内部的 & 符号不再引发重载，是编译器的默认按位与操作。
            fn bitand(self, rhs: u8) -> u8 { self & u8 }
        }
```

数学运算符 Trait

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

    //此函数会消费self，设计一些复杂结构的加法
    //时可能导致一些复杂性
    fn add(self, rhs: Rhs) -> Self::Output;
}

macro_rules! add_impl {
```

```
    ($($t:ty)*) => ($(
        //注意这里的const实现，代表trait里面的函数
        //都是const函数,为了使得加法能够给const及static赋值
        impl const Add for $t {
            type Output = $t;

            fn add(self, other: $t) -> $t { self + other }
        }

        forward_ref_binop! { impl const Add, add for $t, $t }
    )*)
}
//实现了所有数据类型的加法
add_impl! { usize u8 u16 u32 u64 u128 isize i8 i16 i32 i64 i128 f32 f64 }


pub trait AddAssign<Rhs = Self> {
    //使用可变引用，与Add不同
    fn add_assign(&mut self, rhs: Rhs);
}

macro_rules! add_assign_impl {
    ($($t:ty)+) => ($(
        impl const AddAssign for $t {
            fn add_assign(&mut self, other: $t) { *self += other }
        }

        forward_ref_op_assign! { impl const AddAssign, add_assign for $t, $t }
    )+)
}
//实现了所有数据类型的加法
add_assign_impl! { usize u8 u16 u32 u64 u128 isize i8 i16 i32 i64 i128 f32 f64
}
```

其他数学运算类似，略

位运算符 Trait

与数学运算类似，略

关系运算符Trait

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

关系运算符的代码稍微复杂，这里给出较完整的代码。

```rust
//"==" "!="的运算符trait，PartialEq用于在整个类型
//定义域内存在值无法满足相等条件的情况。例如浮点类型“NaN != NaN"
//可以定义不同与self的泛型实现不同类型"=="及"!="的运算
pub trait PartialEq<Rhs: ?Sized = Self> {
    /// “==” 重载方法
    fn eq(&self, other: &Rhs) -> bool;

    ///`!=` 重载方法
    fn ne(&self, other: &Rhs) -> bool {
        !self.eq(other)
    }
}

//对于全作用域所有值都可相等的类型。实现Eq trait，
//PartialEq和Eq区别实现，也是Rust安全性的体现之一
//相等判断还是由PartialEq的方法负责
pub trait Eq: PartialEq<Self> {
    fn assert_receiver_is_total_eq(&self) {}
}
```

对于"<,>,<=,>="等四种运算，如果全域有可能出现无法比较的情况，仅实现`PartialOrd<Rhs>`，如下：

```rust
// "<" ">" ">=" "<=" 运算符重载结构，事实上关系运算只需要重载这个Trait
// Ord Trait 不用编码，
// 可以为一个类型实现不同于此类型的PartialEq
pub trait PartialOrd<Rhs: ?Sized = Self>: PartialEq<Rhs> {
    // 显然，只能有一个比较函数，对于全域都满足比较的，此函数内部一般用Ord
    // Trait的cmp，对于无法比较的，需要实现独立的代码，如浮点,因为存在不可比较
    //的值，所以需要用Option
    fn partial_cmp(&self, other: &Rhs) -> Option<Ordering>;

    // "<" 运算符重载
    fn lt(&self, other: &Rhs) -> bool {
        matches!(self.partial_cmp(other), Some(Less))
    }

    //"<="运算符重载
    fn le(&self, other: &Rhs) -> bool {
        // Pattern `Some(Less | Eq)` optimizes worse than negating `None |
Some(Greater)`.
        !matches!(self.partial_cmp(other), None | Some(Greater))
    }

    //">"运算符重载，代码略
    fn gt(&self, other: &Rhs) -> bool;

    //">="运算符重载，代码略
    fn ge(&self, other: &Rhs) -> bool;
```

```rust
    //eq已经在PatialEq中包含
}

//Ord是全域值都可比较的Trait，其与PartialOrd结果应该一致
pub trait Ord: Eq + PartialOrd<Self> {
    //通常partial_cmp() == Some(cmp()),因为全域值
    //都可以比较，不会出现Ordering之外的情况
    fn cmp(&self, other: &Self) -> Ordering;

    fn max(self, other: Self) -> Self
    where
        Self: Sized,
    {
        //见下面代码分析
        max_by(self, other, Ord::cmp)
    }

    fn min(self, other: Self) -> Self
    where
        Self: Sized,
    {
        //见下面代码分析
        min_by(self, other, Ord::cmp)
    }

    fn clamp(self, min: Self, max: Self) -> Self
    where
        Self: Sized,
    {
        assert!(min <= max);
        if self < min {
            min
        } else if self > max {
            max
        } else {
            self
        }
    }
}
//用于表示关系结果的结构体,注意此结构在函数式编程中的实用性
#[derive(Clone, Copy, PartialEq, Debug, Hash)]
#[repr(i8)]
pub enum Ordering {
    /// 小于.
    Less = -1,
    /// 等于.
    Equal = 0,
    /// 大于.
    Greater = 1,
}
```

```rust
impl Ordering {
    //对Ordering做逆操作，代码略
    pub const fn reverse(self) -> Ordering ;

    //用来简化代码及更好的支持函数式编程
    //举例:
    // let x: (i64, i64, i64) = (1, 2, 7);
    // let y: (i64, i64, i64) = (1, 5, 3);
    // let result = x.0.cmp(&y.0).then(x.1.cmp(&y.1)).then(x.2.cmp(&y.2));
    pub const fn then(self, other: Ordering) -> Ordering {
        match self {
            Equal => other,
            _ => self,
        }
    }

    //用来简化代码实及支持函数式编程
    pub fn then_with<F: FnOnce() -> Ordering>(self, f: F) -> Ordering {
        match self {
            Equal => f(),
            _ => self,
        }
    }
}

//用输入的闭包比较函数获取两个值中大的一个
pub fn max_by<T, F: FnOnce(&T, &T) -> Ordering>(v1: T, v2: T, compare: F) -> T
{
    match compare(&v1, &v2) {
        Ordering::Less | Ordering::Equal => v2,
        Ordering::Greater => v1,
    }
}

//用输入的闭包比较函数获取两个值中小的一个
pub fn min_by<T, F: FnOnce(&T, &T) -> Ordering>(v1: T, v2: T, compare: F) -> T
{
    match compare(&v1, &v2) {
        Ordering::Less | Ordering::Equal => v1,
        Ordering::Greater => v2,
    }
}

//cmp::min 作为两变量取小的api调用
pub fn min<T: Ord>(v1: T, v2: T) -> T {
    v1.min(v2)
}

//对变量生成key，两变量取小的key值变量的api
pub fn min_by_key<T, F: FnMut(&T) -> K, K: Ord>(v1: T, v2: T, mut f: F) -> T {
```

```
    min_by(v1, v2, |v1, v2| f(v1).cmp(&f(v2)))
}

//cmp::max 作为两变量取大的api调用
pub fn max<T: Ord>(v1: T, v2: T) -> T {
    v1.max(v2)
}

//对变量生成key，两变量取大的key值的api
pub fn max_by_key<T, F: FnMut(&T) -> K, K: Ord>(v1: T, v2: T, mut f: F) -> T {
    max_by(v1, v2, |v1, v2| f(v1).cmp(&f(v2)))
}
```

以下是利用泛型和Adapter模式的典型的解决一类问题的RUST解决方案，下面是对有序的类型实现逆序的方案

```
//对于实现了PartialOrd的类型实现一个Ord的反转，这个设计是典型的RUST的思考方式，
//利用一个Adpater设计模式+泛型，很轻松的解决了一类需求
//adapter的设计模式例子
pub struct Reverse<T>(pub T);

impl<T: PartialOrd> PartialOrd for Reverse<T> {
    fn partial_cmp(&self, other: &Reverse<T>) -> Option<Ordering> {
        other.0.partial_cmp(&self.0)
    }

    fn lt(&self, other: &Self) -> bool {
        other.0 < self.0
    }

    //其他方法，略
    ...
    ...
}
```

以下是关系运算trait在原生类型上的实现

```
// 具体的实现宏
mod impls {
    use crate::cmp::Ordering::{self, Equal, Greater, Less};
    use crate::hint::unreachable_unchecked;

    //PartialEq在原生类型上的实现,利用宏减少重复代码
    macro_rules! partial_eq_impl {
        ($($t:ty)*) => ($(
            //Rhs类型默认为Self
            impl PartialEq for $t {
                fn eq(&self, other: &$t) -> bool { (*self) == (*other) }
```

```
                fn ne(&self, other: &$t) -> bool { (*self) != (*other) }
            }
        )*)
    }
    //单元类型，一定相等
    impl PartialEq for () {
        fn eq(&self, _other: &()) -> bool {
            true
        }
        fn ne(&self, _other: &()) -> bool {
            false
        }
    }
    //所有类型都实现PartialEq
    partial_eq_impl! {
        bool char usize u8 u16 u32 u64 u128 isize i8 i16 i32 i64 i128 f32 f64
    }

    macro_rules! eq_impl {
        ($($t:ty)*) => ($(
            #[stable(feature = "rust1", since = "1.0.0")]
            impl Eq for $t {}
        )*)
    }

    //浮点不实现Eq
    eq_impl! { () bool char usize u8 u16 u32 u64 u128 isize i8 i16 i32 i64 i128
}

    //关系运算，利用宏减少代码，这个宏仅仅针对浮点数
    macro_rules! partial_ord_impl {
        ($($t:ty)*) => ($(
            #[stable(feature = "rust1", since = "1.0.0")]
            impl PartialOrd for $t {
                fn partial_cmp(&self, other: &$t) -> Option<Ordering> {
                    //RUST的典型的代码，要记住这种简练的语法表达
                    //这个表达主要是考虑到浮点，注意这里是用了impl PartialOrd<&B>
for &A
                    //从而self <= other导致对 (&f32).le()的调用
                    //为什么不直接使用(*self <= *other, *self >= *other)呢
                    //提交了PR，最新的代码库已经修改了
                    match (self <= other, self >= other) {
                        (false, false) => None,
                        (false, true) => Some(Greater),
                        (true, false) => Some(Less),
                        (true, true) => Some(Equal),
                    }
                }
                //不使用默认函数
                fn lt(&self, other: &$t) -> bool { (*self) < (*other) }
                fn le(&self, other: &$t) -> bool { (*self) <= (*other) }
```

```
                fn ge(&self, other: &$t) -> bool { (*self) >= (*other) }
                fn gt(&self, other: &$t) -> bool { (*self) > (*other) }
            }
        )*)
    }

    //仅在浮点数实现
    partial_ord_impl! { f32 f64 }

    //为支持全域值可比较的类型实现的宏
    macro_rules! ord_impl {
        ($($t:ty)*) => ($(
            impl PartialOrd for $t {
                //复用Ord的cmp函数
                fn partial_cmp(&self, other: &$t) -> Option<Ordering> {
                    Some(self.cmp(other))
                }
                fn lt(&self, other: &$t) -> bool { (*self) < (*other) }
                fn le(&self, other: &$t) -> bool { (*self) <= (*other) }
                fn ge(&self, other: &$t) -> bool { (*self) >= (*other) }
                fn gt(&self, other: &$t) -> bool { (*self) > (*other) }
            }

            impl Ord for $t {
                fn cmp(&self, other: &$t) -> Ordering {
                    if *self < *other { Less }
                    else if *self == *other { Equal }
                    else { Greater }
                }
            }
        )*)
    }
    //浮点数不支持Ord
    ord_impl! { char usize u8 u16 u32 u64 u128 isize i8 i16 i32 i64 i128 }

    //A实现了PartialEq<B>, PartialOrd<B>后，对&A实现PartialEq<&B>,
PartialOrd<&B>
    impl<A: ?Sized, B: ?Sized> PartialEq<&B> for &A
    where
        A: PartialEq<B>,
    {
        fn eq(&self, other: &&B) -> bool {
            //注意这个调用方式，此时不能用self.eq调用。
            //eq方法参数为引用
            PartialEq::eq(*self, *other)
        }
        fn ne(&self, other: &&B) -> bool {
            PartialEq::ne(*self, *other)
        }
    }
```

```
}
```

以上较完整的给出了关系运算Trait的代码，可以看到，RUST标准库除了对原生类型做了Trait的实现，也针对受约束的泛型尽可能的做了关系运算符 Trait的实现，以便最大的减少后继的开发量。程序员需要精通RUST的标准库已经针对那些泛型类型做好了实现，避免再重复的造轮子。

？运算符 Trait代码分析

代码路径：try_trait.rs

?操作引入有两个目的：

1. 作为解封装的最简化代码表达形式
2. 作为try...catch...的RUST实现方式

Try trait定义如下：

```
pub trait Try: FromResidual {
    /// ?操作如果结果正常，返回的解封装的正常变量类型
    /// 具体实例可参考随后的Option的Try trait实现
    type Output;

    /// ?操作如果结果异常，返回解封装的异常变量类型
    type Residual;

    /// 从Self::Output解封装的正常类型变量获得封装后的类型变量的函数。当然，封装类型
实现了Try trait
    /// 函数必须符合下面代码的原则，
    /// `Try::from_output(x).branch() --> ControlFlow::Continue(x)`.
    /// 例子:
    /// ```
    /// assert_eq!(<Result<_, String> as Try>::from_output(3), Ok(3));
    /// assert_eq!(<Option<_> as Try>::from_output(4), Some(4));
    /// assert_eq!(
    ///     <std::ops::ControlFlow<String, _> as Try>::from_output(5),
    ///     std::ops::ControlFlow::Continue(5),
    /// );
    fn from_output(output: Self::Output) -> Self;

    /// branch函数会返回ControlFlow类型变量，用以标识代码继续流程还是中断流程并提前返
回
    /// 例子:
    ///
    /// assert_eq!(Ok::<_, String>(3).branch(), ControlFlow::Continue(3));
    /// assert_eq!(Err::<String, _>(3).branch(), ControlFlow::Break(Err(3)));
    ///
```

```
    /// assert_eq!(Some(3).branch(), ControlFlow::Continue(3));
    /// assert_eq!(None::<String>.branch(), ControlFlow::Break(None));
    ///
    /// assert_eq!(ControlFlow::<String, _>::Continue(3).branch(),
ControlFlow::Continue(3));
    /// assert_eq!(
    ///     ControlFlow::<_, String>::Break(3).branch(),
    ///     ControlFlow::Break(ControlFlow::Break(3)),
    /// );
    fn branch(self) -> ControlFlow<Self::Residual, Self::Output>;
}

pub trait FromResidual<R = <Self as Try>::Residual> {
    /// 该函数从解封装的异常类型变量获取封装后的类型变量。封装后的类型实现了Try
Trait。
    ///
    /// 此函数必须符合下面代码的原则
    /// `FromResidual::from_residual(r).branch() --> ControlFlow::Break(r)`.
    /// 例子:
    /// assert_eq!(Result::<String, i64>::from_residual(Err(3_u8)), Err(3));
    /// assert_eq!(Option::<String>::from_residual(None), None);
    /// assert_eq!(
    ///     ControlFlow::<_, String>::from_residual(ControlFlow::Break(5)),
    ///     ControlFlow::Break(5),
    /// );
    fn from_residual(residual: R) -> Self;
}
```

Try Trait对? 操作支持的举例如下:

```
//不用? 操作的代码
 pub fn simple_try_fold_3<A, T, R: Try<Output = A>>(
     iter: impl Iterator<Item = T>,
     mut accum: A,
     mut f: impl FnMut(A, T) -> R,
 ) -> R {
     for x in iter {
         let cf = f(accum, x).branch();
         match cf {
             ControlFlow::Continue(a) => accum = a,
             ControlFlow::Break(r) => return R::from_residual(r),
         }
     }
     R::from_output(accum)
 }
// 使用? 操作的代码:
 fn simple_try_fold<A, T, R: Try<Output = A>>(
     iter: impl Iterator<Item = T>,
     mut accum: A,
     mut f: impl FnMut(A, T) -> R,
```

```
) -> R {
    for x in iter {
        accum = f(accum, x)?;
    }
    R::from_output(accum)
}
```

由上，可推断出T?表示如下代码

```
match((T as Try).branch()) {
    ControlFlow::Continue(a) => a,
    ControlFlow::Break(r) => return (T as Try)::from_residual(r),
}
```

ControlFlow类型代码如下, 主要用于指示代码控制流程指示， 逻辑上可类比于continue, break 关键字 代码如下：

```
pub enum ControlFlow<B, C = ()> {
    //代码过程继续执行，可以从C中得到代码过程的中间结果
    Continue(C),
    /// 代码过程应退出，可以从B中得到代码退出时的中间结果
    Break(B),
}
```

Option的Try Trait实现

实现代码如下：

```
impl<T> ops::Try for Option<T> {
    type Output = T;
    // Infallible是一种错误类型，但该错误永远也不会发生，
    // Residual 只可能是None，所以是Option类型，但是因为不会返回Some()，
    // 所以T使用Infallible来表示不会有Some，这也表现了RUST的安全理念，
    // 一定在类型定义的时候保证代码安全。
    type Residual = Option<convert::Infallible>;

    fn from_output(output: Self::Output) -> Self {
        Some(output)
    }

    fn branch(self) -> ControlFlow<Self::Residual, Self::Output> {
        match self {
            Some(v) => ControlFlow::Continue(v),
            None => ControlFlow::Break(None),
        }
    }
```

```
}

impl<T> const ops::FromResidual for Option<T> {
    fn from_residual(residual: Option<convert::Infallible>) -> Self {
        match residual {
            None => None,
        }
    }
}
```

所以，一个Option? 等同于如下代码：

```
    match(Option<T>.branch()) {
        ControlFlow::Continue(a) => a,
        //下面代码实际就是return None
        ControlFlow::Break(None) => return (Option<T>::from_residual(None)),
    }
```

Result<T,E>类型的Try Trait请自行分析

小结

利用Try Trait，程序员可以实现自定义类型的?，提供函数式编程的有力手段并简化代码，提升代码的理解度。

Range 运算符代码分析

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

Range是符号 … , start...end , start... , ...end , ...=end，start...=end 形式
代码书写虽然采用了上面的形式，但编译器将其转换成了不同的具体结构。如下：

`..` 的数据结构是 `RangeFull` ,如下：

```
struct RangeFull;
```

`start.. end` 的数据结构 是 `Range<Idx>` ,如下

```
pub struct Range<Idx> {
    pub start: Idx,
    pub end: Idx,
}
```

`start..` 的数据结构是 `RangeFrom<Idx>`，代码略 `.. end` 的数据结构是 `RangeTo<Idx>`，略 `start..=end` 的数据结构是 `RangeInclusive<Idx>` 略 `..=end` 的数据结构是 `RangeToInclusive<Idx>`,略

以上的Idx需要满足Idx:PartialOrd

为了明确上述结构中的边界值是否属于Range内部，定义了Range的边界类型结构Bound 源代码：

```
pub enum Bound<T> {
    /// 边界包括在Range内
    Included(T),
    /// 边界不包括在Range内
    Excluded(T),
    /// 边界是无限的，边界不存在
    Unbounded,
}
```

利用 `RangeBounds<T: ?Sized>` 的trait实现了对Range的边界取值及判断某值是否在Range中。 所有Range类型都实现了此trait。代码如下：

```
pub trait RangeBounds<T: ?Sized> {
    /// 获取范围的起始值
    ///
    /// 例子
    /// assert_eq!((..10).start_bound(), Unbounded);
    /// assert_eq!((3..10).start_bound(), Included(&3));
    fn start_bound(&self) -> Bound<&T>;

    /// 获取范围的终止值.
    /// 例子
    /// assert_eq!((3..).end_bound(), Unbounded);
    /// assert_eq!((3..10).end_bound(), Excluded(&10));
    fn end_bound(&self) -> Bound<&T>;

    /// 范围是否包括某个值.
    /// 例子
    /// assert!( (3..5).contains(&4));
    /// assert!(!(3..5).contains(&2));
    ///
    /// assert!( (0.0..1.0).contains(&0.5));
    /// assert!(!(0.0..1.0).contains(&f32::NAN));
    /// assert!(!(0.0..f32::NAN).contains(&0.5));
    /// assert!(!(f32::NAN..1.0).contains(&0.5));
    fn contains<U>(&self, item: &U) -> bool
    where
        T: PartialOrd<U>,
        U: ?Sized + PartialOrd<T>,
```

```
    {
        //比较有意思的典型的RUST代码
        (match self.start_bound() {
            Included(start) => start <= item,
            Excluded(start) => start < item,
            Unbounded => true,
        }) && (match self.end_bound() {
            Included(end) => item <= end,
            Excluded(end) => item < end,
            Unbounded => true,
        })
    }
}
```

RangeBounds针对RangeFull，RangeTo, RangeInclusive, RangeToInclusive, RangeFrom, Range结构都进行了实现。同时针对(Bound, Bound)的元组做了实现。

```
impl<T> RangeBounds<T> for RangeFrom<T> {
    fn start_bound(&self) -> Bound<&T> {
        Included(&self.start)
    }
    fn end_bound(&self) -> Bound<&T> {
        Unbounded
    }
}

impl<T> RangeBounds<T> for Range<T> {
    fn start_bound(&self) -> Bound<&T> {
        Included(&self.start)
    }
    fn end_bound(&self) -> Bound<&T> {
        Excluded(&self.end)
    }
}
//其他略
```

Range的灵活性

完全可以定义 ((0,0)...(100,100))； (“1st”...“30th”)这种极有表现力的Range。 Range使用的时候，需要先定义一个取值集合，定义类型表示这个集合，针对类型实现PartialOrd。就可以对这个集合的类型用Range符号了。 值得注意的是，对于 `Range<Idx>` ,如果一个变量类型为U, 则如果实现了 `PartialOrd<U> for Idx` ， 那U就有可能属于Range, 即U可以与Idx不同。 Range操作符多用于与Index运算符结合或与Iterator Trait结合使用，在后继的Index运算符和Iterator中会研究Range是如何与他们结合的。

小结

基于泛型的Range类型提供了非常好的语法手段，只要某类型支持排序，那就可以定义一个在此类型基础上实现的Range类型。再结合Index和Iterator, 将高效的实现极具冲击力的代码。

RUST的Index 运算符代码分析

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

数组下标符号[]由Index, IndexMut两个Trait完成重载。数组下标符号重载使得程序更有可读性。两个Trait如下定义:

```
// [T][Idx] 形式重载
pub trait Index<Idx: ?Sized> {
    /// The returned type after indexing.
    type Output: ?Sized;

    /// 如果传入的参数超过内存界限将马上引发panic
    fn index(&self, index: Idx) -> &Self::Output;
}
//mut [T][Idx]形式重载
pub trait IndexMut<Idx: ?Sized>: Index<Idx> {
    fn index_mut(&mut self, index: Idx) -> &mut Self::Output;
}
```

由以上可以看出类似["Hary"], ["Bold"]之类的下标表达形式都是可以存在的。

切片数据结构[T]的Index实现

切片的Index实现采用了一个辅助的trait `SliceIndex<[T]>` 来支持。

```
impl<T, I> ops::Index<I> for [T]
where
    I: SliceIndex<[T]>,
{
    type Output = I::Output;

    fn index(&self, index: I) -> &I::Output {
        index.index(self)
    }
}

impl<T, I> ops::IndexMut<I> for [T]
where
```

```
    I: SliceIndex<[T]>,
{
    fn index_mut(&mut self, index: I) -> &mut I::Output {
        index.index_mut(self)
    }
}
```

SliceIndex trait 被设计同时满足Index trait及切片类型自身方法的需求。因为这些需求在逻辑上是同领域的。集中在SliceIndex trait模块内聚性更好。如：`[T]::get<I:SliceIndex>(&self, I)->Option<&I::Output>` 就是直接调用SliceIndex中的方法来实现切片成员的获取。

以下是SliceIndex trait的实现

```
mod private_slice_index {
    use super::ops;
    //在私有模块中定义一个Sealed Trait，后继的SliceIndex继承Sealed。
    //带来的结果是只有在本模块实现了Sealed Trait的类型才能实现SliceIndex
    //即使SliceIndex是公有定义，其他类型仍然不能够实现SliceIndex
    pub trait Sealed {}

    impl Sealed for usize {}
    impl Sealed for ops::Range<usize> {}
    impl Sealed for ops::RangeTo<usize> {}
    impl Sealed for ops::RangeFrom<usize> {}
    impl Sealed for ops::RangeFull {}
    impl Sealed for ops::RangeInclusive<usize> {}
    impl Sealed for ops::RangeToInclusive<usize> {}
    impl Sealed for (ops::Bound<usize>, ops::Bound<usize>) {}
}

pub unsafe trait SliceIndex<T: ?Sized>: private_slice_index::Sealed {
    /// 此类型通常为T或者T的引用，切片，裸指针类型
    type Output: ?Sized;

    // 从slice变量中用self获取Option<Output>变量
    fn get(self, slice: &T) -> Option<&Self::Output>;

    fn get_mut(self, slice: &mut T) -> Option<&mut Self::Output>;

    //slice是序列的头指针，后面的具体实现会看到为什么用 *const
    unsafe fn get_unchecked(self, slice: *const T) -> *const Self::Output;

    unsafe fn get_unchecked_mut(self, slice: *mut T) -> *mut Self::Output;

    //如果self超出slice的安全范围，会panic
    fn index(self, slice: &T) -> &Self::Output;

    fn index_mut(self, slice: &mut T) -> &mut Self::Output;
```

```
}

//为usize实现SliceIndex
unsafe impl<T> SliceIndex<[T]> for usize {
    type Output = T;

    //此函数主要用在不适合使用下标时，例如不确定切片长度，又不希望panic
    fn get(self, slice: &[T]) -> Option<&T> {
        // 这里slice 被强制转化成了* const [T]
        if self < slice.len() { unsafe { Some(&*self.get_unchecked(slice)) } }
else { None }
    }

    fn get_mut(self, slice: &mut [T]) -> Option<&mut T> {
        //这里slice 被强制转化成了*mut [T]
        if self < slice.len() { unsafe { Some(&mut
*self.get_unchecked_mut(slice)) } } else { None }
    }

    //此函数主要用在不适合下标的情况下
    unsafe fn get_unchecked(self, slice: *const [T]) -> *const T {
        //slice.as_ptr()获得* const T，利用* const T的add方法来取得slice成员的地址
        unsafe { slice.as_ptr().add(self) }
    }

    unsafe fn get_unchecked_mut(self, slice: *mut [T]) -> *mut T {
        //as_mut_ptr返回* mut T指针，在用add方法获得成员地址。
        unsafe { slice.as_mut_ptr().add(self) }
    }

    fn index(self, slice: &[T]) -> &T {
        //使用编译器内置支持，为了效率直接使用了内置的数组下标表示。此操作可能引发
panic
        &(*slice)[self]
    }

    fn index_mut(self, slice: &mut [T]) -> &mut T {
        // 使用编译器内置下标运算符，可能引发panic
        &mut (*slice)[self]
    }
}
```

以上就是针对[T]的以无符号数作为下标取出单一元素的ops::Index 及 ops::IndexMut的底层实现。

针对Range做下标的代码实现

```
unsafe impl<T> SliceIndex<[T]> for ops::Range<usize> {
    type Output = [T];
```

```
    //不会引发panic的方法
    //此处注意，self是Rang<usize>
    fn get(self, slice: &[T]) -> Option<&[T]> {
        //提前做判断
        if self.start > self.end || self.end > slice.len() {
            None
        } else {
            unsafe { Some(&*self.get_unchecked(slice)) }
        }
    }

    //可变引用获取
    fn get_mut(self, slice: &mut [T]) -> Option<&mut [T]> {
        if self.start > self.end || self.end > slice.len() {
            None
        } else {
            unsafe { Some(&mut *self.get_unchecked_mut(slice)) }
        }
    }

    //不对输出参数做判断，调用者要保证输入参数没有问题
    unsafe fn get_unchecked(self, slice: *const [T]) -> *const [T] {
        // 先将*const [T] 转换为 * const T，完成指针运算，然后再转换成* const [T]
        unsafe { ptr::slice_from_raw_parts(slice.as_ptr().add(self.start),
self.end - self.start) }
    }

    //与上面函数类似，略
    unsafe fn get_unchecked_mut(self, slice: *mut [T]) -> *mut [T] {
        unsafe {
            ptr::slice_from_raw_parts_mut(slice.as_mut_ptr().add(self.start),
self.end - self.start)
        }
    }

    fn index(self, slice: &[T]) -> &[T] {
        //超出范围会直接panic
        if self.start > self.end {
            slice_index_order_fail(self.start, self.end);
        } else if self.end > slice.len() {
            slice_end_index_len_fail(self.end, slice.len());
        }
        //将* const [T]转化为切片引用
        unsafe { &*self.get_unchecked(slice) }
    }

    fn index_mut(self, slice: &mut [T]) -> &mut [T] {
        //超出范围会直接panic
        if self.start > self.end {
            slice_index_order_fail(self.start, self.end);
        } else if self.end > slice.len() {
```

```
            slice_end_index_len_fail(self.end, slice.len());
        }
        unsafe { &mut *self.get_unchecked_mut(slice) }
    }
}
```

以上是实现用Range从slice中取出子slice的实现。同样是使用裸指针来达到最高效率。实际上，不用裸指针就没法实现。

```
unsafe impl<T> SliceIndex<[T]> for ops::RangeTo<usize> {
    type Output = [T];

    fn get(self, slice: &[T]) -> Option<&[T]> {
        //将RangeTo转换成Range，然后对ops::Range<usize>的方法直接调用
        (0..self.end).get(slice)
    }

    fn get_mut(self, slice: &mut [T]) -> Option<&mut [T]> {
        //对ops::Range<usize>的方法直接调用
        (0..self.end).get_mut(slice)
    }

    //其他方法也是直接对Range<usize>的实现做调用，  略
}
```

RangeFrom, RangeInclusive, RangeToInclusive, RangeFull等与RangeTo的实现类似，略。

小结

RUST切片的下标计算展示了裸指针的使用技巧，在数组类的成员操作中，基本无法脱离裸指针。在这里，只要不越界，裸指针操作是安全的。

数组数据结构[T;N]的ops::Index实现

```
//注意这里的常量的Trait约束的写法
impl<T, I, const N: usize> Index<I> for [T; N]
where
    [T]: Index<I>,
{
    type Output = <[T] as Index<I>>::Output;

    fn index(&self, index: I) -> &Self::Output {
        Index::index(self as &[T], index)
    }
}

impl<T, I, const N: usize> IndexMut<I> for [T; N]
```

```
where
    [T]: IndexMut<I>,
{
    fn index_mut(&mut self, index: I) -> &mut Self::Output {
        IndexMut::index_mut(self as &mut [T], index)
    }
}
```

以上， `self as &[T]` 即把[T;N]转化为了切片[T], 所以数组的Index就是[T]的Index实现