茶歇

前面章节已经基本把core库及alloc库分析完毕。这两个库是不依赖于操作系统的。
后继主要分析的std库，即在library/std/src目录下的代码，将紧密的与操作系统发生关联。
C程序员开发一个大型的跨操作系统的应用时。在操作系统的C标准库的基础上实现一个适配层，屏蔽不同操作系统的 C标准库的差异是必须的一个工作。
RUST的标准库也同样，在library/std/src/sys; library/std/src/os下实现了对不同操作系统的适配层。这一适配层完成的工作：

1. 定义RUST本身对于操作系统的统一的类型结构，方法及函数。提供给标准库其他模块使用。
2. 将1中定义适配按照操作系统完成对系统调用的类型，函数的适配。

RUST适配操作系统调用的难度在于要将操作系统的资源及操作转换为适合RUST的所有权，借用及生命周期概念的类型及操作。
因为操作系统的系统调用基本上都是C语言，RUST设计了非常方便及简单的与C的交互机制，RUST与C的交互类似于C++与C的交互，几乎是无缝的融合。而在操作系统适配层这个层面，与其说是RUST程序，不如说是RUST语法的C程序。C程序员感觉到十分的熟悉和亲切。

在适配层之上，RUST按照自身语法的要求，对适配层的实现做了若干扩展。如针对RUST全局静态变量的扩展。这一部分代码放在library/std/src/syscommon 目录下。

操作系统的适配层提供的结构显然还太过于底层及复杂，RUST在此基础上实现了更符合现代语言的类型结构，方法，函数，作为标准库的外界接口提供给RUST的程序员。

后继分析如下进行：

1. 按照操作系统的内存管理，文件描述符，进程/线程管理，进程/线程间通信，IO, 网络，文件, 时间，异步编程，杂项的顺序做分析
2. 分析先给出RUST对操作系统适配层及适配层扩展实现，主要给出linux的相关实现代码摘要分析。适当的给一下wasi的代码分析。
3. RUST在2的基础上实现的标准库外部接口。

RUST中与C语言互通

因为涉及到大量的C语言库函数的调用，所以，我们首先要搞清楚如何与C语言互操作的内容。RUST与C语言互操作，主要就是完成RUST到C语言的类型转换，以及函数调用语义的

实现。
代码路径： library/std/src/ffi/.

C语言类型定义适配

代码路径：library/core/ffi/mod.rs

```
macro_rules! type_alias_no_nz {
    {
      $Docfile:tt, $Alias:ident = $Real:ty;
      $( $Cfg:tt )*
    } => {
        #[doc = include_str!($Docfile)]
        $( $Cfg )*
        #[unstable(feature = "core_ffi_c", issue = "94501")]
        pub type $Alias = $Real;
    }
}

macro_rules! type_alias {
    {
      $Docfile:tt, $Alias:ident = $Real:ty, $NZAlias:ident = $NZReal:ty;
      $( $Cfg:tt )*
    } => {
        type_alias_no_nz! { $Docfile, $Alias = $Real; $( $Cfg )* }

        #[doc = concat!("Type alias for `NonZero` version of [`", stringify!
($Alias), "`]")]
        #[unstable(feature = "raw_os_nonzero", issue = "82363")]
        $( $Cfg )*
        pub type $NZAlias = $NZReal;
    }
}

//以下的定义，对所有C语言的类型以"c_xxxx"来命名，并用类型别名的形式定义为RUST的类型
//以下做了简化，仅针对linux操作系统
type_alias! { "c_char.md", c_char = c_char_definition::c_char, NonZero_c_char =
c_char_definition::NonZero_c_char;
type_alias! { "c_schar.md", c_schar = i8, NonZero_c_schar = NonZeroI8; }
type_alias! { "c_uchar.md", c_uchar = u8, NonZero_c_uchar = NonZeroU8; }
type_alias! { "c_short.md", c_short = i16, NonZero_c_short = NonZeroI16; }
type_alias! { "c_ushort.md", c_ushort = u16, NonZero_c_ushort = NonZeroU16; }
type_alias! { "c_int.md", c_int = i32, NonZero_c_int = NonZeroI32; }
type_alias! { "c_uint.md", c_uint = u32, NonZero_c_uint = NonZeroU32; }
type_alias! { "c_long.md", c_long = i32, NonZero_c_long = NonZeroI32;
type_alias! { "c_ulong.md", c_ulong = u32, NonZero_c_ulong = NonZeroU32;
type_alias! { "c_longlong.md", c_longlong = i64, NonZero_c_longlong =
NonZeroI64; }
type_alias! { "c_ulonglong.md", c_ulonglong = u64, NonZero_c_ulonglong =
```

```
NonZeroU64; }
type_alias_no_nz! { "c_float.md", c_float = f32; }
type_alias_no_nz! { "c_double.md", c_double = f64;

pub type c_size_t = usize;
pub type c_ptrdiff_t = isize;
pub type c_ssize_t = isize;

mod c_char_definition {
    cfg_if! {
        if #[cfg(any(
            all(
                target_os = "linux",
                any(
                    target_arch = "aarch64",
                    target_arch = "arm",
                    target_arch = "powerpc",
                    target_arch = "powerpc64",
                    target_arch = "s390x",
                    target_arch = "riscv64",
                    target_arch = "riscv32"
                )
            ),
            all(target_os = "fuchsia", target_arch = "aarch64")
        ))] {
            pub type c_char = u8;
            pub type NonZero_c_char = crate::num::NonZeroU8;
        }
    }
}
//以下是针对C语言的可变参数 VA_ARG给出的相关RUST匹配
pub enum c_void {
    __variant1,
    __variant2,
}

pub struct VaListImpl<'f> {
    gp_offset: i32,
    fp_offset: i32,
    overflow_arg_area: *mut c_void,
    reg_save_area: *mut c_void,
    _marker: PhantomData<&'f mut &'f c_void>,
}

pub struct VaList<'a, 'f: 'a> {
    inner: &'a mut VaListImpl<'f>,

    _marker: PhantomData<&'a mut VaListImpl<'f>>,
}

impl<'f> VaListImpl<'f> {
```

```
    pub fn as_va_list<'a>(&'a mut self) -> VaList<'a, 'f> {
        VaList { inner: self, _marker: PhantomData }
    }
}
impl<'f> Clone for VaListImpl<'f> {
    #[inline]
    fn clone(&self) -> Self {
        let mut dest = crate::mem::MaybeUninit::uninit();
        // SAFETY: we write to the `MaybeUninit`, thus it is initialized and
`assume_init` is legal
        unsafe {
            va_copy(dest.as_mut_ptr(), self);
            dest.assume_init()
        }
    }
}

impl<'f> VaListImpl<'f> {
    /// Advance to the next arg.
    #[inline]
    pub unsafe fn arg<T: sealed_trait::VaArgSafe>(&mut self) -> T {
        // SAFETY: the caller must uphold the safety contract for `va_arg`.
        unsafe { va_arg(self) }
    }

    /// Copies the `va_list` at the current location.
    pub unsafe fn with_copy<F, R>(&self, f: F) -> R
    where
        F: for<'copy> FnOnce(VaList<'copy, 'f>) -> R,
    {
        let mut ap = self.clone();
        let ret = f(ap.as_va_list());
        // SAFETY: the caller must uphold the safety contract for `va_end`.
        unsafe {
            va_end(&mut ap);
        }
        ret
    }
}

extern "rust-intrinsic" {
    //以下缺少了C语言va_start的对应，RUST不需要
    /// Destroy the arglist `ap` after initialization with `va_start` or
    /// `va_copy`.
    fn va_end(ap: &mut VaListImpl<'_>);

    /// Copies the current location of arglist `src` to the arglist `dst`.
    fn va_copy<'f>(dest: *mut VaListImpl<'f>, src: &VaListImpl<'f>);

    /// Loads an argument of type `T` from the `va_list` `ap` and increment the
    /// argument `ap` points to.
```

```
    fn va_arg<T: sealed_trait::VaArgSafe>(ap: &mut VaListImpl<'_>) -> T;
}
```

系统调用的封装

操作系统的系统调用一般出错返回-1, 为了简化对此情况的处理，将这个出错封装到RUST的Result类型，RUST实现了以下机制:

```
//对系统调用出错的判断
pub trait IsMinusOne {
    fn is_minus_one(&self) -> bool;
}

macro_rules! impl_is_minus_one {
    ($($t:ident)*) => ($(impl IsMinusOne for $t {
        fn is_minus_one(&self) -> bool {
            *self == -1
        }
    })*)
}

//对所有系统调用可能的返回类型实现了出错判断trait
impl_is_minus_one! { i8 i16 i32 i64 isize }

//对系统调用进行出错处理的封装，将错误转换为Result类型
pub fn cvt<T: IsMinusOne>(t: T) -> crate::io::Result<T> {
    //Error是对操作系统错误的封装
    if t.is_minus_one() { Err(crate::io::Error::last_os_error()) } else { Ok(t)
}
}

//对于被中断的系统调用做额外的处理封装
pub fn cvt_r<T, F>(mut f: F) -> crate::io::Result<T>
where
    T: IsMinusOne,
    F: FnMut() -> T,
{
    loop {
        match cvt(f()) {
            //如果返回是调用被中断，则再次执行系统调用
            Err(ref e) if e.kind() == ErrorKind::Interrupted => {}
            other => return other,
        }
    }
}

//对系统调用的返回值进行判断，转换为io::Result的结果
```

```
pub fn cvt_nz(error: libc::c_int) -> crate::io::Result<()> {
    if error == 0 { Ok(()) } else {
Err(crate::io::Error::from_raw_os_error(error)) }
}
```

CStr及CString代码分析

代码路径：library/std/src/ffi/c_str.rs
RUST定义CStr及CString主要的目的就是与C的各种库函数交互。
因此CStr及CString不涉及字符串的迭代器，格式化，加减，分裂，字符查找等等操作。只是负责做String及str与C语言之间的转换及与转换相关及调试相关的若干功能。
之所以设计CString，是因为如果需要保存C语言的字符串，需要用堆内存的方式来完成。同时，传递给C语言的字符串，需要位于堆内存中代码才会比较简单及安全。
一般的处理C语言交互传入的字符串的过程是:首先需要用CStr将字符串进行包装，使得保证后继操作复合RUST的安全规则；如果需要对字符串做保存，那需要用CStr生成一个CString。当然，也可以直接转化为String，要根据具体的情况和需求处理，但仅使用String的方式在某些场景下显然效率不高并且也是不合适的。
如果是需要将RUST的str转换成C语言的字符串，则先转换成CString.

CString及CStr类型结构定义如下：

```
pub struct CString {
    // C语言的字符串是一个以0位结尾的字节数组. 通常的，申请的空间大小会大于字符串长度，因此
    // 下面的切片长度不能用于判断字符串长度
    inner: box<[u8]>,
}
pub struct CStr {
    // 此处没有太好的办法，C语言对字符串实际上会存储在一个申请后就固定的字节数组里，然后用指针表示字符串类型
    // 但RUST显然不可能用裸指针来实现，切片类型是最接近的。但要注意C语言中的实际上是个固定的数组
    inner: [c_char],
}
```

CStr主要的需求是对C语言的char*进行封装并定义转换方法，将C语言的字符串安全化。并在需要的时候转化成str或CString类型

```
impl CStr {
    //主要的创建方法，这个函数接收一个已经由C语言模块传递过来的char *指针，然后创建RUST
    //需要的CStr引用 并返回
    // 调用代码应该保证传入参数的正确性。此函数返回的引用生命周期由调用代码的上下文决定
```

```
    // 生命周期的正确性也由调用代码保证。
    pub unsafe fn from_ptr<'a>(ptr: *const c_char) -> &'a CStr {
        //将* const c_char转换成 &[u8]
        unsafe {
            //调用C语言的库函数libc::strlen获得字符串长度，这里实际可以用RUST自行实现
            let len = sys::strlen(ptr);
            let ptr = ptr as *const u8;
            //先创建&[u8]，然后创建Self类型引用
            Self::_from_bytes_with_nul_unchecked(slice::from_raw_parts(ptr, len as usize + 1))
        }
    }

    pub fn from_bytes_until_nul(bytes: &[u8]) -> Result<&CStr, FromBytesUntilNulError> {
        //core库实现了memchr，查找到字符串尾部字节位置
        let nul_pos = memchr::memchr(0, bytes);
        match nul_pos {
            Some(nul_pos) => {
                // slice仅保留有效的字节.
                let subslice = &bytes[..nul_pos + 1];
                // 见后继的分析
                Ok(unsafe { CStr::from_bytes_with_nul_unchecked(subslice) })
            }
            None => Err(FromBytesUntilNulError(())),
        }
    }
    //从准备好的[u8]创建CStr的引用并返回
    pub const unsafe fn from_bytes_with_nul_unchecked(bytes: &[u8]) -> &CStr {
        debug_assert!(!bytes.is_empty() && bytes[bytes.len() - 1] == 0);
        //见后继的分析
        unsafe { Self::_from_bytes_with_nul_unchecked(bytes) }
    }

    const unsafe fn _from_bytes_with_nul_unchecked(bytes: &[u8]) -> &Self {
        // 利用裸指针转换，注意这里CStr结构定义没有用#[repr(transparent)]或#[repr(C)]，这里直接做转换的根据感觉有些不足，
        //返回的生命周期要小于bytes，但因为bytes基本上是从一个裸指针转换而来的，所以
        //这里的&Self的生命周期的正确性还是要由调用代码负责
        unsafe { &*(bytes as *const [u8] as *const Self) }
    }

    //将CStr转换成C语言的字符串，需要保证复合C语言字符串的规则
    //此函数可能引发一个潜在问题如下例:
    /// use std::ffi::CString;
    ///
    /// let ptr = CString::new("Hello").expect("CString::new failed").as_ptr();
    /// unsafe {
    ///     // 这里会出现悬垂指针,见后面的解释
    ///     *ptr;
```

```
    /// }
    /// ```
    ///
    /// 以上悬垂指针是因为`as_ptr`没有生命周期，因为CString创建的变量又没有变量声明与之绑定，所以其在执行完as_ptr后立即被释放。
    /// 可使用如下的方法
    /// ```no_run
    /// # #![allow(unused_must_use)]
    /// use std::ffi::CString;
    ///
    /// //声明一个变量，生命周期一般会到作用域的尾部。
    /// let hello = CString::new("Hello").expect("CString::new failed");
    /// let ptr = hello.as_ptr();
    /// unsafe {
    ///     // `ptr` is valid because `hello` is in scope
    ///     *ptr;
    /// }
    /// ```
    pub const fn as_ptr(&self) -> *const c_char {
        self.inner.as_ptr()
    }

    //转换成去掉尾部0的[u8]切片引用
    pub fn to_bytes(&self) -> &[u8] {
        let bytes = self.to_bytes_with_nul();
        // SAFETY: to_bytes_with_nul returns slice with length at least 1
        unsafe { bytes.get_unchecked(..bytes.len() - 1) }
    }

    //转换成[u8]切片引用,尾部仍然有0
    pub fn to_bytes_with_nul(&self) -> &[u8] {
        unsafe { &*(&self.inner as *const [c_char] as *const [u8]) }
    }

    //转换成&str
    pub fn to_str(&self) -> Result<&str, str::Utf8Error> {
        str::from_utf8(self.to_bytes())
    }

    //转换成CString
    pub fn into_c_string(self: Box<CStr>) -> CString {
        //将堆内存从Box取出
        let raw = Box::into_raw(self) as *mut [u8];
        //重新形成Box结构，然后创建CString
        CString { inner: unsafe { Box::from_raw(raw) } }
    }
}
```

CString的相关实现如下：

```rust
impl CString {
    pub fn new<T: Into<Vec<u8>>>(t: T) -> Result<CString, NulError> {
        trait SpecNewImpl {
            fn spec_new_impl(self) -> Result<CString, NulError>;
        }

        impl<T: Into<Vec<u8>>> SpecNewImpl for T {
            default fn spec_new_impl(self) -> Result<CString, NulError> {
                let bytes: Vec<u8> = self.into();
                match memchr::memchr(0, &bytes) {
                    Some(i) => Err(NulError(i, bytes)),
                    None => Ok(unsafe { CString::_from_vec_unchecked(bytes) }),
                }
            }
        }

        //此函数用来防止多次申请内存
        fn spec_new_impl_bytes(bytes: &[u8]) -> Result<CString, NulError> {
            // 此处checked_add的优化效率最高，bytes中没有0，所以需要加1
            let capacity = bytes.len().checked_add(1).unwrap();

            // 申请堆内存，并将bytes写入堆内存,此处申请可以防止重复申请，但无论成功与否都会申请内存
            let mut buffer = Vec::with_capacity(capacity);
            //此时还没有给buffer的尾部赋0
            buffer.extend(bytes);

            // 看bytes内是否有0值
            match memchr::memchr(0, bytes) {
                //有0，出错了，将buffer在参数返回，由外部代码处理
                Some(i) => Err(NulError(i, buffer)),
                //无0，生成CString，生成函数中会赋0
                None => Ok(unsafe { CString::_from_vec_unchecked(buffer) }),
            }
        }

        //可以从[u8]切片生成CString
        impl SpecNewImpl for &'_ [u8] {
            fn spec_new_impl(self) -> Result<CString, NulError> {
                spec_new_impl_bytes(self)
            }
        }

        //支持从str生成CString
        impl SpecNewImpl for &'_ str {
            fn spec_new_impl(self) -> Result<CString, NulError> {
                spec_new_impl_bytes(self.as_bytes())
            }
        }
```

```
        //支持从可变[u8]生成CString
        impl SpecNewImpl for &'_ mut [u8] {
            fn spec_new_impl(self) -> Result<CString, NulError> {
                spec_new_impl_bytes(self)
            }
        }

        t.spec_new_impl()
    }

    //从Vec创建CString,实际是从String创建的支持函数
    pub unsafe fn from_vec_unchecked(v: Vec<u8>) -> Self {
        debug_assert!(memchr::memchr(0, &v).is_none());
        unsafe { Self::_from_vec_unchecked(v) }
    }

    //Vec<u8>已经完成安全检查，不会出错
    unsafe fn _from_vec_unchecked(mut v: Vec<u8>) -> Self {
        //以下就是增加尾部的0值
        v.reserve_exact(1);
        v.push(0);
        //将堆内存从Vec结构转移至Box结构
        Self { inner: v.into_boxed_slice() }
    }

    //从C语言字符串创建CString，此时c语言的字符串应该是前期RUST代码申请的堆内存
    //要规避不是RUST申请的堆内存的情况
    pub unsafe fn from_raw(ptr: *mut c_char) -> CString {
        // ptr应该从CString::into_raw得到的,此方法使用后，可以省略一次内存拷贝
        unsafe {
            //得到字符串长度
            let len = sys::strlen(ptr) + 1; // Including the NUL byte
            //形成正确的切片引用
            let slice = slice::from_raw_parts_mut(ptr, len as usize);
            //形成CString
            CString { inner: Box::from_raw(slice as *mut [c_char] as *mut [u8])
}
        }
    }

    pub fn into_raw(self) -> *mut c_char {
        //CString已经包含了0值
        Box::into_raw(self.into_inner()) as *mut c_char
    }

    //转换成String类型
    pub fn into_string(self) -> Result<String, IntoStringError> {
        String::from_utf8(self.into_bytes()).map_err(|e| IntoStringError {
            error: e.utf8_error(),
            inner: unsafe { Self::_from_vec_unchecked(e.into_bytes()) },
```

```
        })
    }

    pub fn into_bytes(self) -> Vec<u8> {
        //消费了CString，Box中的堆内存转移到Vec
        let mut vec = self.into_inner().into_vec();
        //删掉尾部的0值
        let _nul = vec.pop();
        debug_assert_eq!(_nul, Some(0u8));
        vec
    }

    pub fn into_bytes_with_nul(self) -> Vec<u8> {
        //不对尾部的0值做处理
        self.into_inner().into_vec()
    }

    //将CString转换为[u8]切片引用
    pub fn as_bytes(&self) -> &[u8] {
        // 删除尾部的0值
        unsafe { self.inner.get_unchecked(..self.inner.len() - 1) }
    }

    //保留尾部的0值
    pub fn as_bytes_with_nul(&self) -> &[u8] {
        &self.inner
    }

    //转换为CStr的引用
    pub fn as_c_str(&self) -> &CStr {
        &*self
    }


    pub fn into_boxed_c_str(self) -> Box<CStr> {
        //Box取出堆内存指针，然后转换，再封装入Box，RUST这个实在是麻烦
        unsafe { Box::from_raw(Box::into_raw(self.into_inner()) as *mut CStr) }
    }

    fn into_inner(self) -> Box<[u8]> {
        //将Box取出，如果直接解封装的方式，因为会调用self的drop函数，会再调用内部的
Box的drop。
        //用MannuallyDrop来规避是不想再重构了，这个代码的例子不应学习
        //如果有需要inner，那就不应该用Box<[u8]>这种方式来设计
        //这个设计导致必须用下面这种技巧，带来理解上的复杂性
        let this = mem::ManuallyDrop::new(self);
        unsafe { ptr::read(&this.inner) }
    }

    //从Vec生成CString
    pub unsafe fn from_vec_with_nul_unchecked(v: Vec<u8>) -> Self {
```

```
        debug_assert!(memchr::memchr(0, &v).unwrap() + 1 == v.len());
        unsafe { Self::_from_vec_with_nul_unchecked(v) }
    }

    //同上，无需再检查0值
    unsafe fn _from_vec_with_nul_unchecked(v: Vec<u8>) -> Self {
        Self { inner: v.into_boxed_slice() }
    }

    //此函数为从String转换为CString准备
    pub fn from_vec_with_nul(v: Vec<u8>) -> Result<Self, FromVecWithNulError> {
        //确定0值的位置
        let nul_pos = memchr::memchr(0, &v);
        match nul_pos {
            //如果0值的位置正确
            Some(nul_pos) if nul_pos + 1 == v.len() => {
                // 创建CString
                Ok(unsafe { Self::_from_vec_with_nul_unchecked(v) })
            }
            //出错处理
            Some(nul_pos) => Err(FromVecWithNulError {
                error_kind: FromBytesWithNulErrorKind::InteriorNul(nul_pos),
                bytes: v,
            }),
            None => Err(FromVecWithNulError {
                error_kind: FromBytesWithNulErrorKind::NotNulTerminated,
                bytes: v,
            }),
        }
    }
}

//drop函数
impl Drop for CString {
    fn drop(&mut self) {
        unsafe {
            //消费了Box，堆内存已经拷贝到栈，然后将C语言的字符串设置为空字符串。
            *self.inner.get_unchecked_mut(0) = 0;

        }
    }
}

impl ops::Deref for CString {
    type Target = CStr;

    fn deref(&self) -> &CStr {
        unsafe { CStr::_from_bytes_with_nul_unchecked(self.as_bytes_with_nul()) }
}
```

```
    }
}
```

CString, CStr其他代码略。

代码工程中的一个技巧

在对不同的CPU架构，不同的操作系统进行适配的时候，通常在代码中采用如下的组织方式：

1. 有接口定义文件，在C语言中一般用头文件，在RUST中用mod.rs文件，这个文件负责向其他模块提供一致的API访问界面.
2. 每种CPU架构或者每种操作系统各自建立一个目录(模块).
3. 每种CPU架构或者每种操作系统各自实现接口的代码都在此目录下实现.
4. 利用编译参数控制对于特定CPU架构，操作系统仅编译特定目录下的代码.
   举例如下：

```rust
mod common;

cfg_if::cfg_if! {
    if #[cfg(unix)] {
        mod unix;
        pub use self::unix::*;
    } else if #[cfg(windows)] {
        mod windows;
        pub use self::windows::*;
    } else if #[cfg(target_os = "solid_asp3")] {
        mod solid;
        pub use self::solid::*;
    } else if #[cfg(target_os = "hermit")] {
        mod hermit;
        pub use self::hermit::*;
    } else if #[cfg(target_os = "wasi")] {
        mod wasi;
        pub use self::wasi::*;
    } else if #[cfg(target_family = "wasm")] {
        mod wasm;
        pub use self::wasm::*;
    } else if #[cfg(all(target_vendor = "fortanix", target_env = "sgx"))] {
        mod sgx;
        pub use self::sgx::*;
    } else {
        mod unsupported;
        pub use self::unsupported::*;
    }
}
```

以上的mod.rs实现RUST对编译的控制，按照事先的target_os的配置控制了编译的目录。这是RUST的一个优势，C语言需要在Makefile里面来控制编译的目录。 RUST用`pub use self::windows::*`的语法，将特定的操作系统的模块重导出为`std::sys::*`，从而对其他的RUST模块实现了对不同操作系统API接口访问的统一。 类似的设计方式可能会在多种场景下遇到，例如对不同数据库API的适配，对不同3D API的适配等等。

OsString 代码分析

操作系统系统调用采用的字符串类型很可能与C语言不同,单纯只有CStr及CString满足不了需求。按照与CStr及CString类似的实现，RUST也实现了OsStr及OsString。显然，这个模块包括了操作系统相关及操作系统无关的两个部分： 操作系统无关部分代码路径如下：
library/src/std/src/ffi/os_str.rs
操作系统相关部分代码路径如下，(仅列出linux及windows)：
library/src/std/src/sys/unix/os_str.rs
library/src/std/src/sys/windows/os_str.rs

linux操作系统相关部分的接口类型结构定义：

```
#[repr(transparent)]
pub struct Buf {
    pub inner: Vec<u8>,
}

#[repr(transparent)]
pub struct Slice {
    pub inner: [u8],
}
```

windows操作系统相关部分的接口结构定义：

```
pub struct Buf {
    pub inner: Wtf8Buf,
}
pub struct Slice {
    pub inner: Wtf8,
}
pub struct Wtf8Buf {
    bytes: Vec<u8>,
}
pub struct Wtf8 {
    bytes: [u8],
}
```

OsString及OsStr的定义：

```
pub struct OsString {
    inner: Buf,
}

pub struct OsStr {
    inner: Slice,
}
```

OsString及OsStr实际上是两个适配器，每个方法基本上都是做个透传，如：

```
impl OsString {
    pub fn new() -> OsString {
        OsString { inner: Buf::from_string(String::new()) }
    }

    pub fn into_string(self) -> Result<String, OsString> {
        self.inner.into_string().map_err(|buf| OsString { inner: buf })
    }
    ...
}
```

OsString及OStr在unix上的结构定义与RUST的String及str基本一致，代码略

std的内存管理分析

std库与core库在内存管理RUST提供的机制是统一的。即Allocator trait 与 GlobalAlloc trait。
从std库可以发现RUST为什么将内存管理分成了Allocator及GlobalAlloc两个trait。
GlobalAlloc trait是操作系统无关及操作系统相关的界面接口。GlobalAlloc的主要功能就是对操作系统的系统调用进行封装，并完成RUST的内存类型与操作系统的系统调用的类型转换。
Allocator是RUST自身的内存管理模块，其他的RUST模块如果有内存需求，同过Allocator triat来完成。Allocator使用GlobalAlloc完成对操作系统的使用。

std库用System 作为这两个trait的实现载体，core库中用Global重新实现了Allocator，Global没有实现GlobalAlloc,因为Global需要适配非操作系统情况，具体请参考02-内存一章, System的代码如下：

以下是unix操作系统相关的部分，代码位置：library/std/src/sys/unix/alloc.rs
不同的操作系统，其内存申请的系统调用都不一致，因此对GlobalAlloc的实现也不一致。类unix的操作系统主要使用了libc的库函数实现操作系统的系统调用。后继还会看到更多的libc中与操作系统交互的代码，分析RUST std库的代码，必须熟练的掌握libc库。

```rust
//单元结构体，仅用来作为内存管理的实现载体
pub struct System;

unsafe impl GlobalAlloc for System {
    unsafe fn alloc(&self, layout: Layout) -> *mut u8 {
        // 用libc来实现内存申请，这里适配的难点在于对齐的适配，
        // libc的malloc对齐是不能指定的。
        // 只有在申请的内存对齐小于MIN_ALIGN而且申请的内存大小大于对齐大小时
        // 才能调用libc的malloc做申请.
        // 对齐的内存申请C程序员不是太长接触
        if layout.align() <= MIN_ALIGN && layout.align() <= layout.size() {
            libc::malloc(layout.size()) as *mut u8
        } else {
            //见随后的分析
            aligned_malloc(&layout)
        }
    }

    unsafe fn alloc_zeroed(&self, layout: Layout) -> *mut u8 {
        // 一样需要处理对齐
        if layout.align() <= MIN_ALIGN && layout.align() <= layout.size() {
            libc::calloc(layout.size(), 1) as *mut u8
        } else {
            let ptr = self.alloc(layout);
            //不能用calloc处理时需要清零
            if !ptr.is_null() {
                ptr::write_bytes(ptr, 0, layout.size());
            }
            ptr
        }
    }

    unsafe fn dealloc(&self, ptr: *mut u8, _layout: Layout) {
        //都使用free做释放
        libc::free(ptr as *mut libc::c_void)
    }

    unsafe fn realloc(&self, ptr: *mut u8, layout: Layout, new_size: usize) ->
*mut u8 {
        //对齐处理
        if layout.align() <= MIN_ALIGN && layout.align() <= new_size {
            libc::realloc(ptr as *mut libc::c_void, new_size) as *mut u8
        } else {
            //无法对齐时的处理，
            realloc_fallback(self, ptr, layout, new_size)
        }
    }
}

//此函数用于libc的realloc无法支持RUST语义时使用
```

```
pub unsafe fn realloc_fallback(
    alloc: &System,
    ptr: *mut u8,
    old_layout: Layout,
    new_size: usize,
) -> *mut u8 {
    // Docs for GlobalAlloc::realloc require this to be valid:
    let new_layout = Layout::from_size_align_unchecked(new_size,
old_layout.align());

    let new_ptr = GlobalAlloc::alloc(alloc, new_layout);
    if !new_ptr.is_null() {
        let size = cmp::min(old_layout.size(), new_size);
        ptr::copy_nonoverlapping(ptr, new_ptr, size);
        GlobalAlloc::dealloc(alloc, ptr, old_layout);
    }
    new_ptr
}
cfg_if::cfg_if! {
    if #[cfg(target_os = "wasi")] {
        //wasi提供aligned_alloc的支持
        unsafe fn aligned_malloc(layout: &Layout) -> *mut u8 {
            libc::aligned_alloc(layout.align(), layout.size()) as *mut u8
        }
    } else {
        //其他需要用posix_memalign来完成
        unsafe fn aligned_malloc(layout: &Layout) -> *mut u8 {
            let mut out = ptr::null_mut();
            // posix_memalign requires that the alignment be a multiple of
`sizeof(void*)`.
            // Since these are all powers of 2, we can just use max.
            let align = layout.align().max(crate::mem::size_of::<usize>());
            let ret = libc::posix_memalign(&mut out, align, layout.size());
            if ret != 0 { ptr::null_mut() } else { out as *mut u8 }
        }
    }
}
```

RUST程序需要处理内存对齐，所以调用了一些不常见的libc内存函数。 以下为操作系统无关部分的内存管理实现。

```
impl System {
    //具体的内存申请实现，与Global类似，可参考前文的解释
    fn alloc_impl(&self, layout: Layout, zeroed: bool) -> Result<NonNull<[u8]>,
AllocError> {
        match layout.size() {
            0 => Ok(NonNull::slice_from_raw_parts(layout.dangling(), 0)),
            size => unsafe {
                let raw_ptr = if zeroed {
                    //System也实现了GlobalAlloc trait，这与core库中不同，core库中
```

```
对GlobalAlloc trait中方法的调用是使用了编译器提供了包装。这使得core库即可以适用于用户态也可以适用于内核态。但std库就是在用户态，所以解决方法更直接。
                    GlobalAlloc::alloc_zeroed(self, layout)
                } else {
                    GlobalAlloc::alloc(self, layout)
                };
                let ptr = NonNull::new(raw_ptr).ok_or(AllocError)?;
                Ok(NonNull::slice_from_raw_parts(ptr, size))
            },
        }
    }

    //内存不足，需要增加空间的申请操作
    unsafe fn grow_impl(
        &self,
        ptr: NonNull<u8>,
        old_layout: Layout,
        new_layout: Layout,
        zeroed: bool,
    ) -> Result<NonNull<[u8]>, AllocError> {
        debug_assert!(
            new_layout.size() >= old_layout.size(),
            "`new_layout.size()` must be greater than or equal to
`old_layout.size()`"
        );

        match old_layout.size() {
            //旧的空间是0，那相当于申请一个新空间的操作
            0 => self.alloc_impl(new_layout, zeroed),

            //旧的内存块与新内存块的对齐是一致的
            old_size if old_layout.align() == new_layout.align() => unsafe {
                //直接调用realloc的逻辑即可，
                let new_size = new_layout.size();

                intrinsics::assume(new_size >= old_layout.size());
                //realloc的逻辑保证旧的内存块的内容被保留
                let raw_ptr = GlobalAlloc::realloc(self, ptr.as_ptr(),
old_layout, new_size);
                let ptr = NonNull::new(raw_ptr).ok_or(AllocError)?;
                if zeroed {
                    //旧内存块的内容不变，仅对新内存块处理
                    raw_ptr.add(old_size).write_bytes(0, new_size - old_size);
                }
                //形成新的NonNull<[u8]>返回
                Ok(NonNull::slice_from_raw_parts(ptr, new_size))
            },

            //旧内存块与新内存块的对齐不一致
            old_size => unsafe {
                //需要按照新的内存布局参数重新申请一块内存
```

```
                let new_ptr = self.alloc_impl(new_layout, zeroed)?;
                //将旧内存块的内容拷贝到新内存
                ptr::copy_nonoverlapping(ptr.as_ptr(), new_ptr.as_mut_ptr(),
old_size);
                //将旧内存块释放掉
                Allocator::deallocate(&self, ptr, old_layout);
                Ok(new_ptr)
            },
        }
    }
}

// 实现Allocator trait，std库后继会使用此trait完成内存管理操作。
unsafe impl Allocator for System {
    //申请内存块
    fn allocate(&self, layout: Layout) -> Result<NonNull<[u8]>, AllocError> {
        self.alloc_impl(layout, false)
    }
    //申请内存块，并将内存块清零
    fn allocate_zeroed(&self, layout: Layout) -> Result<NonNull<[u8]>,
AllocError> {
        self.alloc_impl(layout, true)
    }
    //释放内存块
    unsafe fn deallocate(&self, ptr: NonNull<u8>, layout: Layout) {
        if layout.size() != 0 {
            unsafe { GlobalAlloc::dealloc(self, ptr.as_ptr(), layout) }
        }
    }
    //增长内存空间
    unsafe fn grow(
        &self,
        ptr: NonNull<u8>,
        old_layout: Layout,
        new_layout: Layout,
    ) -> Result<NonNull<[u8]>, AllocError> {
        unsafe { self.grow_impl(ptr, old_layout, new_layout, false) }
    }
    //增长内存空间，增长的部分进行清零
    unsafe fn grow_zeroed(
        &self,
        ptr: NonNull<u8>,
        old_layout: Layout,
        new_layout: Layout,
    ) -> Result<NonNull<[u8]>, AllocError> {
        unsafe { self.grow_impl(ptr, old_layout, new_layout, true) }
    }
    //收缩内存空间，此时必须重新申请内存。
    unsafe fn shrink(
        &self,
        ptr: NonNull<u8>,
```

```
        old_layout: Layout,
        new_layout: Layout,
    ) -> Result<NonNull<[u8]>, AllocError> {
        debug_assert!(
            new_layout.size() <= old_layout.size(),
            "`new_layout.size()` must be smaller than or equal to
`old_layout.size()`"
        );

        match new_layout.size() {
            // 收缩空间至0，实际上就是释放内存
            0 => unsafe {
                Allocator::deallocate(&self, ptr, old_layout);
                //返回一个dangling的指针表示悬垂指针。此处应该用
Option<NonNull<[u8]>>返回才符合
                //rust的习惯用法吧，目前的返回代码后继有额外的判断负担
                Ok(NonNull::slice_from_raw_parts(new_layout.dangling(), 0))
            },

            //如果内存对齐相同
            new_size if old_layout.align() == new_layout.align() => unsafe {
                intrinsics::assume(new_size <= old_layout.size());
                //realloc函数会保留原内存内容
                let raw_ptr = GlobalAlloc::realloc(self, ptr.as_ptr(),
old_layout, new_size);
                let ptr = NonNull::new(raw_ptr).ok_or(AllocError)?;
                Ok(NonNull::slice_from_raw_parts(ptr, new_size))
            },

            //对齐不同，必须重新申请内存
            new_size => unsafe {
                let new_ptr = Allocator::allocate(&self, new_layout)?;
                //将原内存内容拷贝入新内存
                ptr::copy_nonoverlapping(ptr.as_ptr(), new_ptr.as_mut_ptr(),
new_size);
                //释放原内存
                Allocator::deallocate(&self, ptr, old_layout);
                Ok(new_ptr)
            },
        }
    }
}
```

以上是std库的通用实现,代码位置：library/std/src/alloc.rs