

RUST标准库内存模块代码分析

内存模块的代码路径举例如下(以作者电脑上的路径): %USER%.rustup\toolchains\nightly-x86_64-pc-windows-msvc\lib\rustlib\src\rust\library\core\src\alloc*.*
%USER%.rustup\toolchains\nightly-x86_64-pc-windows-msvc\lib\rustlib\src\rust\library\core\src\ptr*.* %USER%.rustup\toolchains\nightly-x86_64-pc-windows-msvc\lib\rustlib\src\rust\library\core\src\mem*.*
%USER%.rustup\toolchains\nightly-x86_64-pc-windows-msvc\lib\rustlib\src\rust\library\core\src\intrinsic.rs %USER%.rustup\toolchains\nightly-x86_64-pc-windows-msvc\lib\rustlib\src\rust\library\alloc\src\alloc.rs

RUST最难以掌握特性之一是RUST的内存操作。RUST与C相同, 需要对内存做彻底的控制, 即程序可以在代码中编写专属内存管理系统, 并将内存管理系统与语言类型相关联, 在内存块与语言类型之间做自如的转换。对于当前现代语法的高级语言如Java/Python/JS/Go, 内存管理是编译器的任务, 这就导致大部分程序员对于内存管理缺乏经验, 所以对RUST内存安全相关的所有权/生命周期等缺乏实践认知。相对于C, RUST的现代语法及内存安全语法导致RUST的内存块与类型系统相互转换的细节非常复杂, 不容易被透彻理解。本章将从标准库的内存模块的代码分析中给出RUST内存操作的本质。对本章内容掌握了, RUST语言的最难关便过了。

从内存角度考察一个变量, 则每个变量具备统一的内存参数, 这些参数是:

1. 变量的首地址, 是一个usize的数值
2. 变量类型占用的内存块大小
3. 变量类型内存字节对齐的基数
4. 变量类型中成员内存顺序

如果变量成员是复合类型, 可递归上面的四个参数。

RUST认为变量类型成员顺序与编译优化不可分割, 因此, 变量成员内存顺序完全由编译器控制, 这与C不同, C中变量类型成员的顺序是不能被编译器改动的。这使得C变量的内存布局对程序员是透明的。这种透明性导致了C语言在涉及类型内存布局的操作中会出现坏代码。

与C相同, RUST具备将一块内存块直接转换成某一类型变量的能力。这一能力是RUST系统级编程及高性能的一个基石。但因为这个转换使得代码可以绕过编译器的类型系统检查, 造成了BUG也绕过了编译器的某些错误检查, 而这些错误很可能在系统运行很久之后才真正的出错, 造成排错的极高成本。

GC类语言去掉了这一能力, 但也牺牲了性能, 且无法作为系统级语言。RUST没有因噎废食, 在保留能力的同时给出这一能力明确的危险标识unsafe, 加上整体的内存安全设计, 使得此类错误更易被发现, 更易被定位, 极大的降低了错误的数目及排错的成本。unsafe容易让初学RUST语言的程序员产生排斥感, 但unsafe实际上是RUST不可分割的部

分，一个好的RUST程序员绝不是不使用unsafe，而是能够准确的把握好unsafe使用的合适场合及合适范围，必要的时候必须使用，但绝不滥用。

掌握RUST的内存，主要有如下几个部分：

1. 编译器提供的固有内存操作函数
2. 内存块与类型系统的结合点：裸指针 `*const T/*mut T`
3. 裸指针的包装结构: `NonNull<T>/Unique<T>`
4. 未初始化内存块的处理: `MaybeUninit<T>/ManuallyDrop<T>`
5. 堆内存申请及释放

裸指针标准库代码分析

裸指针 `*const T/*mut T` 将内存和类型系统相连接，裸指针代表了一个内存块，指示了内存块首地址，大小，对齐等属性及后文提到的元数据，但不保证这个内存块的有效性和安全性。

与 `*const T/*mut T` 不同，`&T/&mut T` 则保证内存块是安全和有效的，即 `&T/&mut T` 满足内存块首地址内存对齐，内存块已经完成了初始化。在RUST中，`&T/&mut T` 是被绑定在某一内存块上，只能用于读写这一内存块。

对于内存块更复杂的操作，由 `*const T/*mut T` 负责，主要有：

1. 将usize类型数值强制转换成裸指针类型，以此数值为首地址的内存块被转换为相应的类型。如果对这一转换后的内存块进行读写，可能造成内存安全问题。
2. 在不同的裸指针类型之间进行强制转换，实质上完成了裸指针指向的内存块的类型强转，如果对这一转换后的内存块进行读写，可能造成内存安全问题。
3. `*const u8` 作为堆内存申请的内存块绑定变量
4. 内存块置值操作，如清零或置一个魔术值
5. 显式的内存块拷贝操作，某些情况下，内存块拷贝是必须的高性能方式。
6. 利用指针偏移计算获取新的内存块，在数组及切片访问，字符串，协议字节填写，文件缓存等都需要指针偏移计算。
7. 从外部的C函数接口对接的指针参数
8. ...

RUST的裸指针类型不象C语言的指针类型那样仅仅是一个地址值，为满足实现内存安全的类型系统需求，并兼顾内存使用效率和方便性，RUST的裸指针实质是一个较复杂的类型结构体。

裸指针具体实现

`*const T/*mut T` 实质是个数据结构体，由两个部分组成，第一个部分是一个内存地址，第二个部分对这个内存地址的约束性描述-元数据

```

//从下面结构定义可以看到，裸指针本质就是PtrComponents<T>
pub(crate) union PtrRepr<T: ?Sized> {
    pub(crate) const_ptr: *const T,
    pub(crate) mut_ptr: *mut T,
    pub(crate) components: PtrComponents<T>,
}

pub(crate) struct PtrComponents<T: ?Sized> {
    // *const () 保证元数据部分是空
    pub(crate) data_address: *const (),
    // 不同类型指针的元数据
    pub(crate) metadata: <T as Pointee>::Metadata,
}

//下面Pointee的定义展示一个RUST的编程技巧，即trait可以只用
//来定义关联类型，Pointee即只用来指定Metadata的类型。
pub trait Pointee {
    /// The type for metadata in pointers and references to `Self`.
    type Metadata: Copy + Send + Sync + Ord + Hash + Unpin;
}

//瘦指针元数据是单元类型，即是空
pub trait Thin = Pointee<Metadata = ()>;

```

元数据的规则:

- 对于固定大小类型的指针（实现了 `Sized` Trait），RUST定义为瘦指针(thin pointer)，元数据大小为0，类型为(),这里要注意，RUST中数组也是固定大小的类型，运行中对数组下标合法性的检测，就是比较是否已经越过了数组的内存大小。
- 对于动态大小类型的指针(DST 类型)，RUST定义为胖指针(fat pointer 或 wide pointer)，元数据为：
 - 对于结构类型，如果最后一个成员是动态大小类型(结构的其他成员不允许为动态大小类型)，则元数据为此动态大小类型的元数据
 - 对于 `str` 类型，元数据是按字节计算的长度值，元数据类型是 `usize`
 - 对于切片类型，例如 `[T]` 类型，元数据是数组元素的数目值，元数据类型是 `usize`
 - 对于trait对象，例如 `dyn SomeTrait`，元数据是 `[DynMetadata][DynMetadata]`（后面代码解释）（例如：`DynMetadata`）随着RUST的发展，有可能会根据需要引入新的元数据种类。

在标准库代码当中没有指针类型如何实现Pointee Trait的代码，编译器针对每个类型自动的实现了Pointee。如下为rust编译器代码的一个摘录

```

pub fn ptr_metadata_ty(&'tcx self, tcx: TyCtxt<'tcx>) -> Ty<'tcx> {
    // FIXME: should this normalize?
    let tail = tcx.struct_tail_without_normalization(self);

```

```

match tail.kind() {
    // Sized types
    ty::Infer(ty::IntVar(_) | ty::FloatVar(_))
    | ty::Uint(_)
    | ty::Int(_)
    | ty::Bool
    | ty::Float(_)
    | ty::FnDef(..)
    | ty::FnPtr(_)
    | ty::RawPtr(..)
    | ty::Char
    | ty::Ref(..)
    | ty::Generator(..)
    | ty::GeneratorWitness(..)
    | ty::Array(..)
    | ty::Closure(..)
    | ty::Never
    | ty::Error(_)
    | ty::Foreign(..)
    | ty::Adt(..)
    // 如果是固定类型, 元数据是单元类型 tcx.types.unit, 即为空
    | ty::Tuple(..) => tcx.types.unit,

    //对于字符串和切片类型, 元数据为长度tcx.types.usize, 是元素长度
    ty::Str | ty::Slice(_) => tcx.types.usize,

    //对于dyn Trait类型, 元数据从具体的DynMetadata获取*
    ty::Dynamic(..) => {
        let dyn_metadata = tcx.lang_items().dyn_metadata().unwrap();
        tcx.type_of(dyn_metadata).subst(tcx, &[tail.into()])
    },

    //以下类型不应有元数据
    ty::Projection(_)
    | ty::Param(_)
    | ty::Opaque(..)
    | ty::Infer(ty::TyVar(_))
    | ty::Bound(..)
    | ty::Placeholder(..)
    | ty::Infer(ty::FreshTy(_) | ty::FreshIntTy(_) |
    ty::FreshFloatTy(_)) => {
        bug!("`ptr_metadata_ty` applied to unexpected type: {:?}",
tail)
    }
}
}
}

```

以上代码中的中文注释比较清晰的说明了编译器对每一个类型（或类型指针）都实现了Pointee中元数据类型的获取。对于trait对象的元数据的具体结构定义见如下代码：

```

//dyn trait裸指针的元数据结构,此元数据会被用于获取trait的方法
pub struct DynMetadata<Dyn: ?Sized> {
    //在堆内存中的VTable变量的引用,VTable见后面的说明
    vtable_ptr: &'static VTable,
    //标示结构对Dyn的所有权关系,
    //其中PhantomData与具体变量的联系在初始化时由编译器自行推断完成,
    //这里PhantomData主要对编译器提示做Drop check时注意本结构体会
    //负责对Dyn类型变量做drop。
    phantom: crate::marker::PhantomData<Dyn>,
}

//此结构是实际的trait实现
struct VTable {
    //trait对象的drop方法的指针
    drop_in_place: fn(*mut ()),
    //trait对象的内存大小
    size_of: usize,
    //trait对象的内存对齐
    align_of: usize,
    //后继是trait对象的所有方法的指针数组
}

```

元数据类型相同的裸指针可以任意的转换，例如：可以有 `* const [usize; 3] as * const[usize; 5]` 这种语句。

元数据类型不同的裸指针之间不能转换，例如；`* const [usize;3] as *const[usize]` 这种语句无法通过编译器

裸指针的操作函数——intrinsic模块内存相关固有函数

intrinsic模块中的函数由编译器内置实现，并提供给其他模块使用。固有函数标准库没有代码，所以对其主要是了解功能和如何使用，intrinsic模块的内存函数一般不由库以外的代码直接调用，而是由mem模块和ptr模块封装后再提供给其他模块。

内存申请及释放函数：

`intrinsic::forget<T:Sized?> (<_>:T)`，代码中调用这个函数后，在变量生命周期终止时，编译器不会调用变量的drop函数。

`intrinsic::drop_in_place<T:Sized?>(to_drop: * mut T)` 在forget后，如果仍然需要对变量调用drop，则在代码中显式调用此函数以触发对变量的drop调用。

`intrinsic::needs_drop<T>()->bool`，判断T类型是否需要做drop操作，实现了Copy trait的类型会返回false

类型转换：`intrinsic::transmute<T,U>(e:T)->U`，对于内存布局相同的类型 T和U, 完成将类型T变量转换为类型U变量，此时T的所有权将转换为U的所有权

指针偏移函数：`intrinsic::offset<T>(dst: *const T, offset: usize)->* const T`，相当于C的基于类型的指针加计算

`intrinsics::ptr_offset_from<T>(ptr: *const T, base: *const T) -> isize` 相当于C的基于类型的指针减

内存块内容修改函数: `intrinsics::copy<T>(src:*const T, dst: *mut T, count:usize)`, 内存拷贝, `src`和`dst`内存可重叠, 类似c语言中的`memcpy`, 此时`dst`原有内存如果已经初始化, `dst`原有变量的`drop`实质会不执行。`src`的变量可能出现两次`drop`, 因此调用此函数的代码需要处理这种情况。

`intrinsics::copy_no_overlapping<T>(src:*const T, dst: *mut T, count:usize)`, 内存拷贝, `src`和`dst`内存不重叠, 内存安全问题同上

`intrinsics::write_bytes(dst: *mut T, val:u8, count:usize)`, C语言的`memset`的RUST实现, 此时, 原内存如果已经初始化, 则因为编译器会继续对`dst`的内存块做`drop`调用, 有可能会UB。

类型内存参数函数: `intrinsics::size_of<T>()->usize` 类型内存空间字节数

`intrinsics::min_align_of<T>()->usize` 返回类型对齐字节数

`intrinsics::size_of_val<T>(_:*const T)->usize` 返回指针指向的变量内存空间字节数

`intrinsics::min_align_of_val<T>(_: *const T)->usize` 返回指针指向的变量对齐字节数

禁止优化的内存函数: 形如 `volatile_xxxx` 的函数是通知编译器不做内存优化的操作函数, 一般硬件相关操作需要禁止优化。

`intrinsics::volatile_copy_nonoverlapping_memory<T>(dst: *mut T, src: *const T, count: usize)` 内存拷贝

`intrinsics::volatile_copy_memory<T>(dst: *mut T, src: *const T, count: usize)` 功能类似C语言`memcpy`

`intrinsics::volatile_set_memory<T>(dst: *mut T, val: u8, count: usize)` 功能类似C语言`memset`

`intrinsics::volatile_load<T>(src: *const T) -> T` 读取内存或寄存器, T类型字节对齐到2的幂次

`intrinsics::volatile_store<T>(dst: *mut T, val: T)` 内存或寄存器写入, 字节对齐

`intrinsics::unaligned_volatile_load<T>(src: *const T) -> T` 字节非对齐

`intrinsics::unaligned_volatile_store<T>(dst: *mut T, val: T)` 字节非对齐

内存比较函数: `intrinsics::raw_eq<T>(a: &T, b: &T) -> bool` 内存比较, 类似C语言`memcmp`

`pub fn ptr_guaranteed_eq<T>(ptr: *const T, other: *const T) -> bool` 判断两个指针是否判断, 相等返回true, 不等返回false


```
pub fn ptr_guaranteed_ne<T>(ptr: *const T, other: *const T) -> bool
```

 判断两个指针是否不等，不等返回true

裸指针方法

RUST针对 `*const T`/`*mut T` 的类型实现了若干方法：

```
impl <T: ?Sized> *const T {  
    ...  
}  
impl <T: ?Sized> *mut T {  
    ...  
}  
impl <T> *const [T] {  
    ...  
}  
impl <T> *mut [T] {  
    ...  
}
```

对于裸指针，RUST标准库包含了最基础的 `*const T`/`*mut T`，以及在 `*const T`/`*mut T` 基础上特化的切片类型[T]的裸指针 `*const [T]`/`*mut [T]`。标准库针对这两种类型实现了一些关联函数及方法。这里一定注意，所有针对 `*const T` 的方法在 `*const [T]` 上都是适用的。

以上有几点值得注意：

1. 可以针对原生类型实现方法(实现trait)，这体现了RUST类型系统的强大扩展性，也是对函数式编程的强大支持
2. 针对泛型约束实现方法，我们可以大致认为 `*const T`/`*mut T` 实质是一种泛型约束，`*const [T]`/`*mut [T]` 是更进一步的约束，这使得RUST可以具备更好的数据抽象能力，简化代码，复用模块。

裸指针的创建

直接从已经初始化的变量创建裸指针：

```
&T as *const T;  
&mut T as *mut T;
```

直接用usize的数值创建裸指针：

```
{  
    let a: usize = 0xf000000000000000;
```

```
unsafe {a as * const i32};  
}
```

操作系统内核经常需要直接将一个地址数值转换为某一类型的裸指针

RUST也提供了一些其他的裸指针创建关联函数：

`ptr::null<T>() -> *const T` 创建一个0值的 `*const T`，实际上就是 `0 as *const T`，用 `null()`函数明显更符合程序员的习惯

`ptr::null_mut<T>()->*mut T` 除了类型以外，其他同上

`ptr::invalid<T>(addr:usize)->*mut T` 将一个数值作为裸指针，指明这是一个无效的裸指针。

`ptr::invalid_mut<T>(addr:usize)->*mut T` 将一个数值作为可变裸指针，指明这是一个无效的指针。

以上两个函数通常是将指针变量用作他途以提高新能

`ptr::from_raw_parts<T: ?Sized>(data_address: *const (), metadata: <T as Pointee>::Metadata) -> *const T` 从内存地址和元数据创建裸指针

`ptr::from_raw_parts_mut<T: ?Sized>(data_address: *mut (), metadata: <T as Pointee>::Metadata) -> *mut T` 功能同上，创建可变裸指针

RUST裸指针类型转换时，经常使用以上两个函数获得需要的指针类型。

切片类型的裸指针创建函数如下：

`ptr::slice_from_raw_parts<T>(data: *const T, len: usize) -> *const [T]`

`ptr::slice_from_raw_parts_mut<T>(data: *mut T, len: usize) -> *mut [T]` 由裸指针类型及切片长度获得切片类型裸指针，调用代码应保证data事实上就是切片的裸指针地址。由类型裸指针转换为切片类型裸指针最突出的应用之一是内存申请，申请的内存返回 `*const u8`的指针，这个裸指针是没有包含内存大小的，只有头地址，因此需要将这个指针转换为 `*const [u8]`，将申请的内存大小包含入裸指针结构体中。

`slice_from_raw_parts`代码如下：

```
pub const fn slice_from_raw_parts<T>(data: *const T, len: usize) -> *const [T]  
{  
    //data.cast()将*const T转换为 *const()  
    from_raw_parts(data.cast(), len)  
}  
  
pub const fn from_raw_parts<T: ?Sized>(  
    data_address: *const (),  
    metadata: <T as Pointee>::Metadata,  
) -> *const T {  
    //由以下代码可以确认 *const T实质就是PtrRepr类型结构体。  
    unsafe { PtrRepr { components: PtrComponents { data_address, metadata }  
}
```



```
}.const_ptr }  
}
```

不属于方法的裸指针函数

`ptr::drop_in_place<T: ?Sized>(to_drop: *mut T)` 此函数是编译器实现的，用于由程序代码人工释放所有权，而不是交由RUST编译器处理。此函数会引发T内部成员的系列drop调用。

`ptr::metadata<T: ?Sized>(ptr: *const T) -> <T as Pointee>::Metadata` 用来返回裸指针的元数据

`ptr::eq<T>(a: *const T, b: *const T) -> bool` 比较指针，此处需要注意，地址比较不但是地址，也比较元数据

ptr模块的函数大部分逻辑都比较简单。很多就是对intrinsic 函数的直接调用。

裸指针类型转换方法

裸指针类型之间的转换：`*const T::cast<U>(self) -> *const U`，本质上就是一个 `*const T as *const U`。利用RUST的类型推断，此函数可以简化代码并支持链式调用。
`*mut T::cast<U>(self) -> *mut U` 同上。

调用以上的函数要注意，如果后继要把返回的指针转换成引用，那必须保证T类型与U类型内存布局完全一致。如果仅仅是将返回值做数值应用，则此约束可以不遵守，cast函数转换后的类型通常由编译器自行推断，有时需要仔细分析。

裸指针与引用之间的类型转换：

`*const T::as_ref<'a>(self) -> Option<&'a T>` 将裸指针转换为引用，因为*const T可能为零，所有需要转换为 Option<&'a T> 类型，转换的安全性由程序员保证，尤其注意满足RUST对引用的安全要求。这里注意，**生命周期标注表明转换后的生命周期实际上与原变量的生命周期相独立**。因此，生命周期的正确性将由调用代码保证。如果没有标注，则返回的引用的生命周期应该小于self,遵循函数参数及返回值的生命周期规则。

`*mut T::as_ref<'a>(self) -> Option<&'a T>` 同上

`*mut T::as_mut<'a>(self) -> Option<&'a mut T>` 同上，但转化类型为 &mut T。

切片类型裸指针类型转换：`ptr::*const [T]::as_ptr(self) -> *const T` 将切片类型的裸指针转换为切片成员类型的裸指针，这个转换会导致指针的元数据丢失

`ptr::*mut [T]::as_mut_ptr(self) -> *mut T` 同上

裸指针结构体属性相关方法：

`ptr::*const T::to_raw_parts(self) -> (*const (), <T as super::Pointee>::Metadata)`

`ptr::*mut T::to_raw_parts(self)->>(* const (), <T as super::Pointee>::Metadata)` 由裸指针获得地址及元数据

`ptr::*const T::is_null(self)->bool`

`ptr::*mut T::is_null(self)->bool` 此函数判断裸指针的地址值是否为0

切片类型裸指针:

`ptr::*const [T]::len(self) -> usize` 获取切片长度, 直接从裸指针的元数据获取长度

`ptr::*mut [T]::len(self) -> usize` 同上

裸指针偏移计算相关方法

`ptr::*const T::offset(self, count: isize)->* const T` 得到偏移后的裸指针

`ptr::*const T::wrapping_offset(self, count: isize) -> *const T` 考虑溢出绕回的offset

`ptr::*const T::offset_from(self, origin: *const T) -> isize` 计算两个裸指针的offset值

`ptr::*mut T::offset(self, count: isize)->* mut T` 偏移后的裸指针

`ptr::*const T::wrapping_offset(self, count: isize) -> *const T` 考虑溢出绕回的offset

`ptr::*const T::offset_from(self, origin: *const T) -> isize` 计算两个裸指针的offset值

以上两个方法基本上通过intrinsic的函数实现

`ptr::*const T::add(self, count: usize) -> Self`

`ptr::*const T::wrapping_add(self, count: usize)->Self`

`ptr::*const T::sub(self, count:usize) -> Self`

`ptr::*const T::wrapping_sub(self, count:usize) -> Self`

`ptr::*mut T::add(self, count: usize) -> Self`

`ptr::*mut T::wrapping_add(self, count: usize)->Self`

`ptr::*mut T::sub(self, count:usize) -> Self`

`ptr::*mut T::wrapping_sub(self, count:usize) -> Self`

以上是对offset函数的包装, 使之更符合语义习惯, 并便于理解

裸指针直接赋值方法

//该方法用于仅给指针结构体的 address部分赋值

```
pub fn set_ptr_value(mut self, val: *const u8) -> Self {  
    // 以下代码因为只修改PtrComponent.address, 所以不能直接用相等  
    // 代码采取的方案是取self的可变引用, 将此引用转换为裸指针的裸指针,  
    let thin = &mut self as *mut *const T as *mut *const u8;  
    // 这个赋值仅仅做了address的赋值, 对于瘦指针, 这个相当于赋值操作,  
    // 对于胖指针, 则没有改变胖指针的元数据。这种操作方式仅仅在极少数的情况下  
    // 可以使用, 极度危险。  
    unsafe { *thin = val };  
}
```

```
self
}
```

本节还有一部分裸指针方法没有介绍，留到mem模块分析完以后再介绍会更易于理解。

裸指针小结

裸指针相关的代码多数比较简单，重要的是理解裸指针的概念，理解intrinsic 相关函数，这样才能够准确的理解代码。

RUST引用 &T 的安全要求

1. 引用的内存地址必须满足类型T的内存对齐要求
2. 引用的内存内容必须是初始化过的 举例：

```
#[repr(packed)]
struct RefTest {a:u8, b:u16, c:u32}
fn main() {
    let test = RefTest{a:1, b:2, c:3};
    //下面代码编译会有告警，因为test.b 内存字节位于奇数，
    // 无法用于借用
    let ref1 = &test.b
}
```

编译器出现如下警告

```
|
9 | let ref1 = &test.b;
|             ^^^^^^^
|
= note: `[warn(unaligned_references)]` on by default
= warning: this was previously accepted by the compiler but is being phased
out; it will become a hard error in a future release!
= note: for more information, see issue #82523 <https://github.com/rust-lang/rust/issues/82523>
= note: fields of packed structs are not properly aligned, and creating a
misaligned reference is undefined behavior (even if that reference is never
dereferenced)
= help: copy the field contents to a local variable, or replace the reference
with a raw pointer and use `read_unaligned`/`write_unaligned` (loads and stores
via `*p` must be properly aligned even when using raw pointers)
```

MaybeUninit<T> 标准库代码分析

RUST对于变量的要求是必须初始化后才能使用，否则就会编译告警。但在程序中，总有内存还未初始化，但需要使用情况：

1. 从堆申请的内存块，这些内存块都是没有初始化的
2. 需要定义一个新的泛型变量时，并且不合适的转移所有权进行赋值时
3. 需要定义一个新的变量，但希望不初始化便能使用其引用时
4. 定义一个数组，但必须在后继代码对数组成员初始化时
5. ...

为了处理这种需要在代码中使用未初始化内存的情况，RUST标准库定义了

```
MaybeUninit<T>
```

MaybeUninit<T> 结构定义

源代码如下：

```
#[repr(transparent)]
pub union MaybeUninit<T> {
    uninit: (),
    value: ManuallyDrop<T>,
}
```

属性 `repr(transparent)` 实际上表示外部的封装结构在内存中等价于内部的变量，

`MaybeUninit<T>` 的内存布局就是 `ManuallyDrop<T>` 的内存布局，从后文可以看到，

`ManuallyDrop<T>` 实际就是T的内存布局。所以 `MaybeUninit<T>` 在内存中实质也就是T类型。

`MaybeUninit<T>` 容器来实现对未初始化变量的封装，以便在不引发编译错误完成对T类型未初始化变量的相关操作。如果T类型的变量未初始化，那需要显式的提醒编译器不做T类型的drop操作，因为drop操作可能会对T类型内部的变量做连锁drop处理，从而引用未初始化的内容，造成UB(undefined behavior)。

RUST用 `ManuallyDrop<T>` 封装结构完成了对编译器的显式提示，对于用 `ManuallyDrop<T>` 封装的变量，生命周期终止的时候编译器不会调用drop操作。

ManuallyDrop<T> 结构及方法

源代码如下：

```
#[repr(transparent)]
pub struct ManuallyDrop<T: ?Sized> {
    value: T,
}
```

重点关注的一些方法：`ManuallyDrop<T>::new (val:T) -> ManuallyDrop<T>`，此函数返回`ManuallyDrop`变量拥有传入的T类型变量所有权，并将此块内存直接用`ManuallyDrop`封装，对于`ManuallyDrop`，编译器不做drop操作，因此也不会触发val的drop。

```
pub const fn new(value: T) -> ManuallyDrop<T> {  
    //所有权转移到结构体内部，value生命周期结束时不会引发drop  
    ManuallyDrop { value }  
}
```

`ManuallyDrop<T>::into_inner(slot: ManuallyDrop<T>)->T`，将封装的T类型变量所有权转移出来，转移出来的变量生命周期终止时，编译器会自动调用类型的drop。

```
pub const fn into_inner(slot: ManuallyDrop<T>) -> T {  
    //将value解封装，所有权转移到返回值中，编译器重新对所有权做处理  
    slot.value  
}
```

`ManuallyDrop<T>::drop(slot: &mut ManuallyDrop<T>)`，drop掉内部变量，封装入`ManuallyDrop<T>`的变量一定是在程序运行的某一时期不需要编译器drop，所以调用这个函数的时候一定要注意正确性。`ManuallyDrop<T>::deref(&self)-> & T`，返回内部包装变量的引用

```
fn deref(&self) -> &T {  
    //返回后，代码可以用&T对self.value做  
    //读操作，但不改变drop的规则  
    &self.value  
}
```

`ManuallyDrop<T>::deref_mut(&mut self)-> & mut T` 返回内部包装变量的可变引用，调用代码可以利用可变引用对内部变量赋值，但不改变drop机制

ManuallyDrop代码举例：

```
use std::mem::ManuallyDrop;  
let mut x = ManuallyDrop::new(String::from("Hello World!"));  
x.truncate(5); // 此时会调用deref  
assert_eq!(*x, "Hello");  
// 但对x的drop不会再发生
```

MaybeUninit<T> 创建方法

`MaybeUninit<T>::uninit()->MaybeUninit<T>`，可视为在栈空间上申请内存的方法，申请的内存大小是T类型的内存大小，该内存没有初始化。利用泛型和Union内存布局，RUST巧

妙的利用此函数在栈上申请一块未初始化内存。此函数非常非常非常值得关注，需要在栈空间定义一个未初始化泛型时，应第一时间想到 `MaybeUninit::<T>::uninit()`。

```
pub const fn uninit() -> MaybeUninit<T> {  
    // 变量内存布局与T类型完全一致  
    MaybeUninit { uninit: () }  
}
```

`MaybeUninit<T>::new(val:T)->MaybeUninit<T>`，内部用`ManuallyDrop`封装了`val`，然后用`MaybeUninit`封装`ManuallyDrop`。因为如果`T`没有初始化过，调用这个函数会编译失败，所以此时内存实际上已经初始化过了。调用此函数要额外注意`val`的`drop`必须在后继有交代。

```
pub const fn new(val: T) -> MaybeUninit<T> {  
    // val这个时候是初始化过的。  
    MaybeUninit { value: ManuallyDrop::new(val) }  
}
```

`MaybeUninit<T>::zeroed()->MaybeUninit<T>`，申请了`T`类型内存并清零。

```
pub fn zeroed() -> MaybeUninit<T> {  
    let mut u = MaybeUninit::<T>::uninit();  
    unsafe {  
        // 因为没有初始化，所以不存在所有权问题，  
        // 必须使用ptr::write_bytes，否则无法给内存清零  
        // ptr::write_bytes直接调用了intrinsics::write_bytes  
        u.as_mut_ptr().write_bytes(0u8, 1);  
    }  
    u  
}
```

对未初始化的变量赋值的方法

将值写入 `MaybeUninit<T>: MaybeUninit<T>::write(val)->&mut T`，这个函数将未初始化的变量初始化，如果调用此方法后不希望解封装，那后继的赋值使用返回的`&mut T`，再次使用`write`会出现内存安全问题。代码如下：

```
pub const fn write(&mut self, val: T) -> &mut T {  
    // 下面这个赋值，会导致原*self的MaybeUninit<T>的变量生命周期截止，  
    // 会调用drop。但不会对内部的T类型变量做drop调用。所以如果*self内部  
    // 的T类型变量已经被初始化且需要做drop，那会造成内存泄漏。  
    // 所以下面这个等式实际上隐含了self内部的T类型变量必须是未初始化的  
    // 或者T类型变量不需要drop。  
    *self = MaybeUninit::new(val);  
}
```



```
// 函数调用后的赋值用返回的&mut T来做。
unsafe { self.assume_init_mut() }
}
```

初始化后解封装的方法

用`assume_init`返回初始化后的变量并消费掉 `MaybeUninit<T>` 变量，这是最标准的做法：

`MaybeUninit<T>::assume_init()->T` ,代码如下：

```
pub const unsafe fn assume_init(self) -> T {
    // 调用者必须保证self已经初始化了
    unsafe {
        intrinsics::assert_inhabited::<T>();
        //把T的所有权返回，编译器会主动对T调用drop
        ManuallyDrop::into_inner(self.value)
    }
}
```

`assume_init_read`是不消费`self`的情况下获得内部`T`变量，内部`T`变量的所有权已经转移到返回变量，后继要注意不能再次调用其他解封装函数。否则解封装后，会出现双份所有权，引发两次对同一变量的`drop`，导致UB。

```
pub const unsafe fn assume_init_read(&self) -> T {

    unsafe {
        intrinsics::assert_inhabited::<T>();
        //会调用ptr::read
        self.as_ptr().read()
    }
}
//此函即ptr::read，会复制一个变量，此时注意，
//实际上src指向的变量的所有权已经转移给了返回变量，
//所以调用此函数的前提是src后继一定不能调用T类型的drop函数，
//例如src本身处于ManallyDrop，或后继对src调用forget，或给src绑定新变量。
//在RUST中，不支持 let xxx = *(&T) 这种转移所有权的方式，
//因此对于只有指针输入，又要转移所有权的，智能利用浅拷贝进行粗暴转移。
pub const unsafe fn read<T>(src: *const T) -> T {`
    //利用MaybeUninit::uninit申请未初始化的T类型内存
    let mut tmp = MaybeUninit::<T>::uninit();
    unsafe {
        //完成内存拷贝
        copy_nonoverlapping(src, tmp.as_mut_ptr(), 1);
        //初始化后的内存解封装并返回
        tmp.assume_init()
    }
}
```

与上个函数比较类似的 `ManuallyDrop<T>::take` 方法，用take函数将变量复制并获得变量的所有权。此时原变量仍然保留在ManuallyDrop中，后继不能再调用其他解封装函数，否则可能会出现UB。这里要特别注意理解take已经把变量的所有权转移到返回变量中。

RUST中的take方法及replace方法的含义是，原变量的地址不能变动，但内容可以获取及更新，因为内容的获取及更新必然导致所有权的转移，为了确保正确性，RUST对很多类型提供了take,replace方法。

```
pub unsafe fn take(slot: &mut ManuallyDrop<T>) -> T {  
    // 拷贝内部变量，并返回内部变量的所有权  
    // 返回后，原有的变量所有权已经消失，不能再用into_inner来返回  
    // 否则会UB  
    unsafe { ptr::read(&slot.value) }  
}
```

`MaybeUninit<T>::assume_init_drop(&self)` 对于已经初始化过的MaybeUninit，如果所有权一直没有转移，则必须调用此函数以触发T类型的drop函数完成所有权的释放。

`MaybeUninit<T>::assume_init_ref(&self)->&T` 返回内部T类型变量的借用，调用者应保证内部T类型变量已经初始化，返回值按照一个普通的引用使用。**根据RUST的生命周期省略规则，此时&T的生命周期小于&self的生命周期，编译器可以借此检查出生命周期的错误。RUST很多从裸指针转换为引用的生命周期都是利用函数的输入及输出的生命周期规则约束才能保证编译器对生命周期的正确检查** `MaybeUninit<T>::assume_init_mut(&mut self)->&mut T` 返回内部T类型变量的可变借用，调用者应保证内部T类型变量已经初始化，返回值按照一个普通的可变引用使用。此时&mut T的生命周期小于&mut self。此函数通常也用于防止assume_init导致的栈拷贝以提高性能。

MaybeUninit<T> 的方法

创建一个MaybeUninit的未初始化数组：

`MaybeUninit<T>::uninit_array<const LEN:usize>()->[Self; LEN]` 此处对LEN的使用方式需要注意，这是不常见的一个泛型写法,这个函数同样的申请了一块内存。代码：

```
pub const fn uninit_array<const LEN: usize>() -> [Self; LEN] {  
    unsafe { MaybeUninit::<[MaybeUninit<T>; LEN]>::uninit().assume_init() }  
}
```

这里要注意区别数组类型和数组元素的初始化。对于数组 `[MaybeUninit<T>;LEN]` 这一类型本身来说，初始化就是确定整体的内存大小，所以数组类型的初始化在声明后就已经完成了。这时assume_init()是正确的。这是一个理解上的盲点。

`MaybeUninit<T>::array_assume_init<const N:usize>(array: [Self; N]) -> [T; N]` 这个函数没有把所有权转移出来，代码分析如下：

```

pub unsafe fn array_assume_init<const N: usize>(array: [Self; N]) -> [T; N]
{
    unsafe {
        //最后调用是*const T::read(), 此处 as *const _的写法可以简化代码,
        //read后, 所有权已经转移到返回值
        //返回后, 此数组内所有的MaybeUninit变量成员不能再解封装
        (&array as *const _ as *const [T; N]).read()
    }
}

```

MaybeUninit典型案例

对T类型变量申请内存及赋值:

```

use std::mem::MaybeUninit;

// 获得一个未初始化的i32引用类型内存
let mut x = MaybeUninit::<&i32>::uninit();
// 将&0写入变量, 完成初始化
x.write(&0);
// 将初始化后的变量解封装供后继的代码使用。
let x = unsafe { x.assume_init() };

```

以上代码, 编译器不会对x.write进行报警, 这是 MaybeUninit<T> 的最重要的应用, 这个例子展示了RUST如何给未初始化内存赋值的处理方式。调用assume_init前, 必须保证变量已经被正确初始化。

更复杂的初始化例子:

```

use std::mem::{self, MaybeUninit};

let data = {
    // data在声明后实际上就已经初始化完毕。
    let mut data: [MaybeUninit<Vec<u32>>; 1000] = unsafe {
        //这里注意实际调用是
        //MaybeUninit::<[MaybeUninit<Vec<u32>>;1000]>::uninit(),
        //RUST的类型推断机制完成了泛型实例化
        MaybeUninit::uninit().assume_init()
    };

    for elem in &mut data[..] {
        elem.write(vec![42]);
    }

    // 直接用transmute完成整个数组类型的转换
    // 仔细思考一下, 这里除了用transmute, 似乎没有其他办法了,
    unsafe { mem::transmute::<_, [Vec<u32>; 1000]>(data) }
}

```

```
};

assert_eq!(&data[0], &[42]);
```

下面例子说明一块内存被 `MaybeUninit<T>` 封装后，编译器将不再对其做释放，必须在代码中显式释放：

```
use std::mem::MaybeUninit;
use std::ptr;

let mut data: [MaybeUninit<String>; 1000] = unsafe {
    MaybeUninit::uninit().assume_init() };
// 初始化了500个String变量
let mut data_len: usize = 0;
for elem in &mut data[0..500] {
    //write没有将所有权转移出ManuallyDrop
    elem.write(String::from("hello"));
    data_len += 1;
}
//编译器无法自动调用drop释放String变量，
//必须显式用drop_in_place释放
for elem in &mut data[0..data_len] {
    //实际上也可以调用assume_init_drop来完成此工作
    unsafe { ptr::drop_in_place(elem.as_mut_ptr()); }
}
```

上例中，在没有`assume_init()`调用的情况下，必须手工调用`drop_in_place`释放内存。

`MaybeUninit<T>` 是一个非常重要的类型结构，未初始化内存是编程中不可避免要遇到的情况，`MaybeUninit<T>` 也就是RUST编程中必须熟练使用的一个类型。

裸指针模块再分析

有了 `MaybeUninit<T>` 做基础后，可以对裸指针其他至关重要的标准库函数做出分析

`ptr::read<T>(src: *const T) -> T` 此函数在 `MaybeUninit<T>` 节中已经给出了代码，`ptr::read`是对所有类型通用的一种复制方法，需要指出，此函数完成浅拷贝，复制后，`src`指向的变量的所有权会转移至返回值。所以，调用此函数的代码必须保证`src`指向的变量生命周期结束后不会被编译器自动调用`drop`，否则可能导致重复`drop`，出现UB问题。

`ptr::read_unaligned<T>(src: *const T) -> T` 当数据结构中有未内存对齐的成员变量时，需要用此函数读取内容并转化为内存对齐的变量。否则会引发UB(undefined behavior) 如下例：

从字节数组中读一个`usize`的值：

```

use std::mem;

fn read_usize(x: &[u8]) -> usize {
    assert!(x.len() >= mem::size_of::<usize>());

    let ptr = x.as_ptr() as *const usize;
    //此处必须用ptr::read_unaligned, 因为不确定字节是否对齐
    unsafe { ptr.read_unaligned() }
}

```

例子中，为了从byte串中读取一个usize，需要用read_unaligned来获取值，不能象C语言那样通过指针类型转换直接获取值。

ptr::write<T>(dst: *mut T, src: T) 代码如下：

```

pub const unsafe fn write<T>(dst: *mut T, src: T) {
    unsafe {
        //浅拷贝
        copy_nonoverlapping(&src as *const T, dst, 1);
        //必须调用forget, 这里所有权已经转移。不允许再对src做drop操作
        intrinsics::forget(src);
    }
}

```

write函数本质上就是一个所有权转移的操作。完成src到dst的浅拷贝，然后调用了forget(src)，这使得src的Drop不再被调用。从而将所有权转移到dst。此函数是mem::replace，mem::transmute_copy的基础。底层由intrinsics::copy_no_overlapping支持。这个函数中，如果dst已经初始化过，那原dst变量的所有权将被丢失掉，有可能引发内存泄漏。

ptr::write_unaligned<T>(dst: *mut T, src: T) 与read_unaligned相对应。举例如下：

```

#[repr(packed, C)]
struct Packed {
    _padding: u8,
    unaligned: u32,
}

let mut packed: Packed = unsafe { std::mem::zeroed() };

// Take the address of a 32-bit integer which is not aligned.
// In contrast to `&packed.unaligned as *mut _`, this has no undefined
behavior.
// 对于结构中字节没有按照2幂次对齐的成员，要用addr_of_mut!宏来获得地址，无法用取引用的方式。
let unaligned = std::ptr::addr_of_mut!(packed.unaligned);

```

```
unsafe { std::ptr::write_unaligned(unaligned, 42) };

assert_eq!({packed.unaligned}, 42); // `{...}` forces copying the field
instead of creating a reference.
```

`ptr::read_volatile<T>(src: *const T) -> T` 是 `intrinsics::volatile_load` 的封装
`ptr::write_volatile<T>(dst: *mut T, src:T)` 是 `intrinsics::volatile_store` 的封装

`ptr::macro addr_of($place:expr)` 因为用 `&` 获得引用必须是字节按照2的幂次对齐的地址，所以用这个宏获取非地址对齐的变量地址

```
pub macro addr_of($place:expr) {
    //关键字是&raw const, 这个是RUST的原始引用语义,
    //但目前还没有在官方做公开。区别与&, &要求地址必
    //须满足字节对齐和初始化, &raw 则没有这个问题
    &raw const $place
}
```

`ptr::macro addr_of_mut($place:expr)` 作用同上。

```
pub macro addr_of_mut($place:expr) {
    &raw mut $place
}
```

指针的通用函数请参考[Rust库函数参考](#)

NonNull<T> 代码分析

结构体定义如下：

```
#[repr(transparent)]
pub struct NonNull<T: ?Sized> {
    pointer: *const T,
}
```

属性 `repr(transparent)` 实际上表示外部的封装结构在内存中等价于内部的变量。

`NonNull<T>` 在内存中与 `*const T` 完全一致。可以直接强转为 `* const T`。

裸指针的值因为可以为0，如果敞开来用，会有很多无法控制的代码隐患。按照RUST的习惯，标准库定义了非0的指针封装结构 `NonNull<T>`，从而可以用 `Option<NonNull<T>>` 来对值可能为0的裸指针做出强制安全代码逻辑。不需要Option的则认为裸指针不会取值为0。

`NonNull<T>` 本身是协变(covariant)类型。

RUST中的协变，在RUST中，不同的生命周期被视为不同的类型，对于带有生命周期的类

型变量做赋值操作时，仅允许子类型赋给基类型(长周期赋给短周期)，为了从基本类型生成复合类型的子类型和基类型的关系，RUST引入了协变性。从基本类型到复合类型的协变性有 协变(covariant)/逆变(contracovariant)/不变(invariant)三种 程序员分析代码时，可以从基本类型之间的生命周期关系及协变性确定复合类型变量之间的生命周期关系，从而做合适的赋值操作。

因为 `NonNull<T>` 实际上是封装 `* mut T` 类型，但 `* mut T` 与 `NonNull<T>` 的协变性不同，所以程序员如果不能确定需要协变类型，就不要使用 `NonNull<T>`

`NonNull<T>` 创建关联方法

创建一个悬垂(dangling)指针，保证指针满足类型内存对齐要求。该指针可能指向一个正常的变量，所以不能认为指向的内存是未初始化的。dangling实际表示 `NonNull<T>` 无意义，与 `NonNull<T>` 的本意有些违背，因为这个语义可以用None来实现。

```
pub const fn dangling() -> Self {
    unsafe {
        //取内存对齐地址作为裸指针的地址。
        //调用者应保证不对此内存地址进行读写
        let ptr = mem::align_of::<T>() as *mut T;
        NonNull::new_unchecked(ptr)
    }
}
```

new函数，由输入的 `*mut T` 裸指针创建 `NonNull<T>`。代码如下：

```
pub fn new(ptr: *mut T) -> Option<Self> {
    if !ptr.is_null() {
        //ptr的安全性已经检查完毕
        Some(unsafe { Self::new_unchecked(ptr) })
    } else {
        None
    }
}
```

`NonNull::<T>::new_unchecked(* mut T)->Self` 用 `* mut T` 生成NonNull，不检查 `* mut T` 是否为0，调用者应保证 `* mut T` 不为0。

from_raw_parts函数，类似裸指针的from_raw_parts。

```
pub const fn from_raw_parts(
    data_address: NonNull<>>,
    metadata: <T as super::Pointee>::Metadata,
) -> NonNull<T> {
    unsafe {
```

```
//需要先用from_raw_parts_mut形成* mut T指针
```

```
NonNull::new_unchecked(super::from_raw_parts_mut(data_address.as_ptr(),  
metadata))  
    }  
}
```

由From trait创建 NonNull<T>

```
impl<T: ?Sized> const From<&mut T> for NonNull<T> {  
    fn from(reference: &mut T) -> Self {  
        unsafe { NonNull { pointer: reference as *mut T } }  
    }  
}  
  
impl<T: ?Sized> const From<&T> for NonNull<T> {  
    fn from(reference: &T) -> Self {  
        //此处说明NonNull也可以接收不可变引用,  
        //代码要注意在后继不能将这个变量转换为可变引用  
        unsafe { NonNull { pointer: reference as *const T } }  
    }  
}
```

NonNull<T> 类型转换方法

NonNull<T> 的方法基本与 *const T/* mut T 相同，也容易理解，下文仅做罗列和简单说明

NonNull::<T>::as_ptr(self)->* mut T 返回内部的pointer 裸指针

NonNull::<T>::as_ref<'a>(&self)->&'a T 返回的引用的生命周期与self的生命周期独立，由调用代码保证正确性。

NonNull::<T>::as_mut<'a>(&mut self)->&'a mut T 与 as_ref类似，但返回可变引用。

NonNull::<T>::cast<U>(self)->NonNull<U> 指针类型转换，程序员应该保证T和U的内存布局相同

NonNull<[T]> 方法

NonNull::<[T]>::slice_from_raw_parts(data: NonNull<T>, len: usize) -> Self 将类型指针转化为类型的切片类型指针，实质是 ptr::slice_from_raw_parts 的一种包装。

NonNull::<[T]>::as_non_null_ptr(self) -> NonNull<T> * const [T]::as_ptr的NonNull版本

NonNull<T> 的使用实例

以下的实例展示了 NonNull<T> 在动态申请堆内存的使用：

```

impl Global {
    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)),
        // SAFETY: `layout` is non-zero in size,
        size => unsafe {
            //raw_ptr是 *const u8类型
            let raw_ptr = if zeroed { alloc_zeroed(layout) } else {
alloc(layout) };

            //NonNull::new处理了raw_ptr为零的情况, 返回NonNull<u8>,
            //此时内存长度还与T不匹配
            let ptr = NonNull::new(raw_ptr).ok_or(AllocError)?;
            //将NonNull<u8>转换为NonNull<[u8]>, NonNull<[u8]>已经
            //是类型T的内存长度。后继可以直接转换为T类型的指针了。
            //这个转换极为重要。
            Ok(NonNull::slice_from_raw_parts(ptr, size))
        },
    }
}
....
}

```

基本上, 如果 `* const T`/`*mut T` 要跨越函数使用, 或作为数据结构体的成员时, 应将之转化成 `NonNull<T>` 或 `Unique<T>`。`*const T` 应该仅仅保持在单一函数内。

NonNull<T> 与 MaybeUninit<T> 相关函数

```

NonNull<T>::as_uninit_ref<'a>(&self) -> &'a MaybeUninit<T>

```

```

pub unsafe fn as_uninit_ref<'a>(&self) -> &'a MaybeUninit<T> {
    // self.cast将NonNull<T>转换为NonNull<MaybeUninit<T>>
    //self.cast.as_ptr将NonNull<MaybeUninit<T>>转换为 *mut MaybeUninit<T>
    unsafe { &*self.cast().as_ptr() }
}

```

```

NonNull<T>::as_uninit_mut<'a>(&self) -> &'a mut MaybeUninit<T>

```

```

NonNull<[T]>::as_uninit_slice<'a>(&self) -> &'a [MaybeUninit<T>]

```

```

pub unsafe fn as_uninit_slice<'a>(&self) -> &'a [MaybeUninit<T>] {
    // 下面的函数调用ptr::slice_from_raw_parts
    unsafe { slice::from_raw_parts(self.cast().as_ptr(), self.len()) }
}

```

```

NonNull<[T]>::as_uninit_slice_mut<'a>(&self) -> &'a mut [MaybeUninit<T>]

```

Unique<T> 代码分析

Unique<T> 类型结构定义如下:

```
#[repr(transparent)]
pub struct Unique<T: ?Sized> {
    pointer: *const T,
    _marker: PhantomData<T>,
}
```

和 NonNull<T> 对比, Unique<T> 多了 PhantomData<T> 类型成员。这个定义使得编译器知晓, Unique<T> 拥有了pointer指向的内存的所有权, NonNull<T> 没有这个特性。具备所有权后, Unique<T> 可以实现Send, Sync等trait。因为获得了所有权, 此块内存无法用于他处, 这也是Unique的名字由来原因。指针在被 Unique<T> 封装前, 必须保证是NonNull的。对于RUST从堆内存申请的内存块, 其指针都是用 Unique<T> 封装后来作为智能指针结构体内部成员变量, 保证智能指针结构体拥有申请出来的内存块的所有权。

Unique<T> 模块的函数及代码与 NonNull<T> 函数代码相类似, 此处不分析。

Unique::cast<U>(self)->Unique<U> 类型转换, 程序员应该保证T和U的内存布局相同

Unique::<T>::new(* mut T)->Option<Self> 此函数内部判断* mut T是否为0值

Unique::<T>::new_unchecked(* mut T)->Self 封装* mut T, 调用代码应该保证* mut T的安全性

Unique::as_ptr(self)->* mut T

Unique::as_ref(&self)->& T 因为Unique具备所有权, 此处&T的生命周期与self相同, 不必特别声明声明周期

Unique::as_mut(&mut self)->& mut T 同上

mem模块函数

泛型类型创建

mem::zeroed<T>() -> T 返回一个内存块清零的泛型变量, 内存块在栈空间, 代码如下:

```
pub unsafe fn zeroed<T>() -> T {
    // 调用代码必须确认T类型的变量可以取全零值
    unsafe {
        intrinsics::assert_zero_valid::<T>();
        MaybeUninit::zeroed().assume_init()
    }
}
```

mem::uninitialized<T>() -> T 返回一个未初始化过的泛型变量, 内存块在栈空间。

```
pub unsafe fn uninitialized<T>() -> T {
    // 调用者必须确认T类型的变量允许未初始化的任意值
    unsafe {
        intrinsics::assert_uninit_valid::<T>();
        MaybeUninit::uninit().assume_init()
    }
}
```

泛型类型拷贝与替换

下面的take及replace都是针对在只有引用的情况下对变量值作更新。这对于结构体成员更新值非常重要。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，而更新值往往需要获取所有权，修改，然后再将新值赋回。此场景一般用take来获取所有权，修改后再用replace将值更新。

`mem::take<T: Default>(dest: &mut T) -> T` 将dest设置为默认内容(不改变所有权)，用一个新变量返回dest的内容。这里有一个坑，即任何类型的default()必然能够满足多次drop不会出现内存安全问题。

```
pub fn take<T: Default>(dest: &mut T) -> T {
    //即mem::replace, 见下文
    //此处, 对于引用类型, 编译器禁止用*dest来转移所有权,
    //所以不能用let xxx = *dest; xxx这种形式返回T
    //其他语言简单的事情在RUST中必须用一个较难理解的方法
    //式来进行解决。replace()对所有权有仔细的处理
    replace(dest, T::default())
}
```

`mem::replace<T>(dest: &mut T, src: T) -> T` 用src的内容赋值dest(不改变所有权)，用一个新变量返回dest的内容。replace函数的难点在于了解所有权的转移。

```
pub const fn replace<T>(dest: &mut T, src: T) -> T {
    unsafe {
        //因为要替换dest, 所以必须对dest原有变量的所有权做处理,
        //因此先用read将*dest的所有权转移到T, 交由调用者进行处理,
        //RUST不支持对引用类型做解引用的相等来转移所有权。将一个引用
        //的所有权进行转移的方式只有粗暴的内存浅拷贝这种方法。
        //使用这个函数, 调用代码必须了解T类型的情况, T类型有可能需要
        //显式的调用drop函数。ptr::read前文已经分析过。
        let result = ptr::read(dest);
        //ptr::write本身会导致src的所有权转移到dest, 后继不允许
        //在src生命周期终止时做drop。ptr::write会用forget(src)做到这一点。
        ptr::write(dest, src);
        result
    }
}
```

```

    }
}

```

`mem::transmute_copy<T, U>(src: &T) -> U` 新建类型U的变量，并把src的内容拷贝到U。调用者应保证T类型的内容与U一致，src后继的所有权问题需要做处理。

```

pub const unsafe fn transmute_copy<T, U>(src: &T) -> U {
    if align_of::<U>() > align_of::<T>() {
        // 如果两个类型字节对齐U 大于 T. 使用read_unaligned
        unsafe { ptr::read_unaligned(src as *const T as *const U) }
    } else {
        // 用read即可完成
        unsafe { ptr::read(src as *const T as *const U) }
    }
}

```

所有权转移的底层实现

所有权的本质是只能对变量做一次drop操作。变量的drop操作会引起变量结构体内部成员的链式drop。所以，只要引发了变量的浅拷贝，所有权便被转移。原先放置变量的那块内存就必须被处理，forget及ManuallyDrop是两种典型方案。不涉及裸指针的代码，一般不必考虑所有权必须人工处理的情况。但一旦涉及到裸指针，那就必须注意看是否出现了一个变量的双份或多份拷贝，每多一次拷贝，意味着编译器会对变量多做一次drop，触发UB。

变量调用drop的时机

如下例子：

```

struct TestPtr {a: i32, b:i32}
impl Drop for TestPtr {
    fn drop(&mut self) {
        println!("{}", self.a, self.b);
    }
}

fn main() {
    let test = Box::new(TestPtr{a:1,b:2});
    let test1 = *test;
    let mut test2 = TestPtr{a:2, b:3};
    // 此行代码会导致先释放test2拥有所有权的变量，
    // 然后再给test2赋值。代码后的输出会给出证据
    // 将test1的所有权转移给test2，无疑代表着test2
    // 现有的所有权会在后继无法访问，因此drop被立即调用。
    test2 = test1;
    println!("{}", test2);
}

```


输出： 2 3 TestPtr { a: 1, b: 2 } 1 2

其他函数

`mem::forget<T>(t:T)` 通知RUST不做变量的drop操作

```
pub const fn forget<T>(t: T) {  
    //没有使用intrinsic::forget, 实际上效果  
    //一致, 这里应该是尽量规避用intrinsic函数  
    let _ = ManuallyDrop::new(t);  
}
```

`mem::forget_unsized<T:Sized?>` 对intrinsic::forget的封装

`mem::size_of<T>()->usize` / `mem::min_align_of<T>()->usize` / `mem::size_of_val<T>(val:&T)->usize` / `mem::min_align_of_val<T>(val: &T)->usize` / `mem::needs_drop<T>()->bool` 基本就是直接调用intrinsic模块的同名函数

`mem::drop<T>(_x:T)` 释放内存

RUST堆内存申请及释放

RUST类型系统的内存布局

RUST提供了 `Layout` 内存布局类型, 此布局类型结构主要用于做堆内存申请。 `Layout` 的数据结构如下:

```
pub struct Layout {  
    // 类型需占用的内存大小, 用字节数目表示  
    size_: usize,  
    // 按照此字节数目进行类型内存对齐,  
    // NonZeroUsize见代码后面文字分析  
    align_: NonZeroUsize,  
}
```

`NonZeroUsize` 是一种非0值的`usize`, 这种类型主要应用于不可取0的值, 本结构中, 字节对齐属性变量不能被置0, 所以用 `NonZeroUsize` 来确保安全性。如果用`usize`类型, 那代码中就可能会把0置给`align_`, 导致bug产生。这是RUST的一个设计规则, 所有的约束要在类型定义即显性化, 从而使bug在编译中就被发现。

每一个RUST的类型都有自身独特的内存布局`Layout`。一种类型的`Layout`可以用 `intrinsic::<T>::size_of()` 及 `intrinsic::<T>::min_align_of()` 获得的类型内存大小和对齐来获得。RUST的内存布局更详细原理阐述请参考[RUST内存布局] (<https://doc.rust-lang.org/nomicon/data.html>), `Layout`比较有典型意义的函数:

```

impl Layout {
    ...
    ...

    //array函数是计算n个T类型变量形成的数组所需的Layout,
    //是从代码了解Rust Layout概念的一个好的实例
    //这里主要注意的是T类型的对齐会导致内存申请不是T类型的内存大小*n
    //而且对齐也是数组的算法
    pub fn array<T>(n: usize) -> Result<Self, LayoutError> {
        //获得n个T类型的内存Layout
        let (layout, offset) = Layout::new::<T>().repeat(n)?;
        debug_assert_eq!(offset, mem::size_of::<T>());
        //以完全对齐的大小, 得出数组的Layout
        Ok(layout.pad_to_align())
    }

    //计算n个T类型需要的内存Layout, 以及成员之间的空间
    pub fn repeat(&self, n: usize) -> Result<(Self, usize), LayoutError> {
        // 所有的成员必须以成员的对齐大小来做内存对齐, 首先计算对齐需要的padding空间
        let padded_size = self.size() + self.padding_needed_for(self.align());
        // 计算共需要多少内存空间, 如果溢出, 返回error
        let alloc_size = padded_size.checked_mul(n).ok_or(LayoutError)?;

        //由已经验证过得原始数据生成Layout, 并返回单成员占用的空间
        unsafe { Ok((Layout::from_size_align_unchecked(alloc_size,
self.align()), padded_size)) }
    }

    //填充以得到一个与T类型完全对齐的, 最小的内存大小的Layout
    pub fn pad_to_align(&self) -> Layout {
        //得到T类型与对齐之间的空间大小
        let pad = self.padding_needed_for(self.align());
        // 完全对齐的大小
        let new_size = self.size() + pad;

        //以完全对齐的大小生成新的Layout
        Layout::from_size_align(new_size, self.align()).unwrap()
    }

    //计算T类型长度与完全对齐的差
    pub const fn padding_needed_for(&self, align: usize) -> usize {
        let len = self.size();

        // 实际上相当与C语言的表达式
        // len_rounded_up = (len + align - 1) & !(align - 1);
        // 就是对对齐大小做除, 如果有余数, 商加1, 是一种常用的方式.
        // 但注意, 在rust中C语言的"+"等同于wrapping_add, C语言的 "-" 等同于
        // wrapping_sub
        let len_rounded_up = len.wrapping_add(align).wrapping_sub(1) &
!align.wrapping_sub(1);
    }
}

```

```

        //减去len, 得到差值
        len_rounded_up.wrapping_sub(len)
    }

    //不检查输入参数, 根据输入参数表示的原始数据生成Layout变量, 调用代码应保证安全性
    pub const unsafe fn from_size_align_unchecked(size: usize, align: usize) ->
    Self {
        // 必须保证align满足不为0.
        Layout { size_: size, align_: unsafe {
        NonZeroUsize::new_unchecked(align) } }
    }

    //对参数进行检查, 生成一个类型的Layout
    pub const fn from_size_align(size: usize, align: usize) -> Result<Self,
    LayoutError> {
        //必须保证对齐是2的幂次
        if !align.is_power_of_two() {
            return Err(LayoutError);
        }

        //满足下面的表达式, 则size将不可能对齐
        if size > usize::MAX - (align - 1) {
            return Err(LayoutError);
        }

        // 参数已经检查完毕.
        unsafe { Ok(Layout::from_size_align_unchecked(size, align)) }
    }
    ...
    ...
}

```

`#[repr(transparent)]` 内存布局模式

`repr(transparent)`用于仅包含一个成员变量的类型, 该类型的内存布局与成员变量类型的内存布局完全一致。类型仅仅具备编译阶段的意义, 在运行时, 类型变量与其成员变量可以认为是一个相同变量, 可以相互无障碍类型转换。使用`repr(transparent)`布局的类型基本是一种封装结构。

`#[repr(packed)]` 内存布局模式

强制类型成员变量以1字节对齐, 此种结构在协议分析和结构化二进制数据文件中经常使用

`#[repr(align(n))]` 内存布局模式

强制类型以2的幂次对齐

`#[repr(RUST)]` 内存布局模式

默认的布局方式，采用此种布局，RUST编译器会根据情况来自行优化内存

`#[repr(C)]` 内存布局模式

采用C语言布局方式，所有结构变量按照声明的顺序在内存排列。默认4字节对齐。

RUST堆内存申请与释放接口

资深的C/C++程序员都了解，在大型系统开发时，往往需要自行实现内存管理模块，以根据系统的特点优化内存使用及性能，并作出内存跟踪。对于操作系统，内存管理模块更是核心功能。对于C/C++小型系统，没有内存管理，仅仅是调用操作系统的内存系统调用，内存管理交给操作系统负责。操作系统内存管理模块接口是内存申请及内存释放的系统调用。对于GC语言，内存管理由虚拟机或语言运行时负责，利用语言提供的new来完成类型结构内存获取。RUST的内存管理分成了三个界面：

1. 由智能指针类型提供的类型创建函数，一般有new，与其他的GC类语言相同，同时增加了一些更直观的函数。
2. 智能指针使用实现Allocator Trait的类型做内存申请及释放。Allocator使用编译器提供的函数名申请及释放内存。
3. 实现了GlobalAlloc Trait的类型来完成独立的内存管理模块，并用#[global_allocator]注册入编译器，替代编译器默认的内存申请及释放函数。这样，RUST达到了：
4. 对于小规模程序，拥有与GC语言相类似的内存获取机制
5. 对于大型程序和操作系统内核，从语言层面提供了独立的内存管理模块接口，达成了将现代语法与内存管理模块共同存在，相互配合的目的。但因为所有权概念的存在，从内存申请到转换为类型系统仍然还存在复杂的工作。堆内存申请和释放的Trait GlobalAlloc定义如下：

```
pub unsafe trait GlobalAlloc {  
    // 申请内存，因为Layout中内存大小不为0，所以，alloc不会申请大小为0的内存  
    unsafe fn alloc(&self, layout: Layout) -> *mut u8;  
    // 释放内存  
    unsafe fn dealloc(&self, ptr: *mut u8, layout: Layout);  
  
    // 申请后的内存应初始化为0  
    unsafe fn alloc_zeroed(&self, layout: Layout) -> *mut u8 {  
        let size = layout.size();  
        let ptr = unsafe { self.alloc(layout) };  
        if !ptr.is_null() {  
            // 此处必须使用write_bytes，确保每个字节都清零  
            unsafe { ptr::write_bytes(ptr, 0, size) };  
        }  
        ptr  
    }  
}
```

```

    //其他方法
    ...
    ...
}

```

在内核编程或大的框架系统编程中，开发人员通常开发自定义的堆内存管理模块，模块实现GlobalAlloc Trait并添加#[global_allocator]标识。对于用户态，RUST标准库有默认的GlobalAlloc实现。

```

extern "Rust" {
    // 编译器会将实现了GlobalAlloc Trait, 并标记
    //#[global_allocator]的四个方法自动转化为以下的函数
    #[rustc_allocator]
    #[rustc_allocator_nounwind]
    fn __rust_alloc(size: usize, align: usize) -> *mut u8;
    #[rustc_allocator_nounwind]
    fn __rust_dealloc(ptr: *mut u8, size: usize, align: usize);
    #[rustc_allocator_nounwind]
    fn __rust_realloc(ptr: *mut u8, old_size: usize, align: usize, new_size:
    usize) -> *mut u8;
    #[rustc_allocator_nounwind]
    fn __rust_alloc_zeroed(size: usize, align: usize) -> *mut u8;
}

//对__rust_xxxxx_再次封装
pub unsafe fn alloc(layout: Layout) -> *mut u8 {
    unsafe { __rust_alloc(layout.size(), layout.align()) }
}

pub unsafe fn dealloc(ptr: *mut u8, layout: Layout) {
    unsafe { __rust_dealloc(ptr, layout.size(), layout.align()) }
}

pub unsafe fn realloc(ptr: *mut u8, layout: Layout, new_size: usize) -> *mut u8
{
    unsafe { __rust_realloc(ptr, layout.size(), layout.align(), new_size) }
}

pub unsafe fn alloc_zeroed(layout: Layout) -> *mut u8 {
    unsafe { __rust_alloc_zeroed(layout.size(), layout.align()) }
}

```

再实现Allocator Trait，对以上四个函数做封装处理。作为RUST其他模块对堆内存的申请和释放接口。

```

pub unsafe trait Allocator {
    fn allocate(&self, layout: Layout) -> Result<NonNull<[u8]>, AllocError>;
}

```

```

    fn allocate_zeroed(&self, layout: Layout) -> Result<NonNull<[u8]>,
AllocError> {
        let ptr = self.allocate(layout)?;
        // SAFETY: `alloc` returns a valid memory block
        // 复杂的类型转换, 实际是调用 *const u8::write_bytes(0, layout.size_)
        unsafe { ptr.as_non_null_ptr().as_ptr().write_bytes(0, ptr.len()) }
        Ok(ptr)
    }

    unsafe fn deallocate(&self, ptr: NonNull<u8>, layout: Layout);

    ...
}

```

Global 实现了 Allocator Trait。Rust大部分alloc库数据结构的实现使用Global作为Allocator。

```

unsafe impl Allocator for Global {
    fn allocate(&self, layout: Layout) -> Result<NonNull<[u8]>, AllocError> {
        //上文已经给出alloc_impl的说明
        self.alloc_impl(layout, false)
    }

    fn allocate_zeroed(&self, layout: Layout) -> Result<NonNull<[u8]>,
AllocError> {
        self.alloc_impl(layout, true)
    }

    unsafe fn deallocate(&self, ptr: NonNull<u8>, layout: Layout) {
        if layout.size() != 0 {
            // SAFETY: `layout` is non-zero in size,
            // other conditions must be upheld by the caller
            unsafe { dealloc(ptr.as_ptr(), layout) }
        }
    }
    ...
    ...
}

```

Allocator使用GlobalAlloc接口获取内存，然后将GlobalAlloc申请到的* mut u8转换为确定大小的单一指针NonNull<[u8]>，并处理申请内存可能出现的不成功。NonNull<[u8]>此时内存布局与 T的内存布局已经相同，后继可以转换为真正需要的T的指针并进一步转化为相关类型的引用，从而符合RUST类型系统安全并进行后继的处理。以上是堆内存的申请和释放。基于泛型，RUST也巧妙实现了栈内存的申请和释放机制 `mem::MaybeUninit<T>`

用Box的内存申请做综合举例：


```

//此处A是一个A:Allocator类型
pub fn try_new_uninit_in(alloc: A) -> Result<Box<mem::MaybeUninit<T>, A>,
AllocError> {
    //实质是T类型的内存Layout
    let layout = Layout::new::<mem::MaybeUninit<T>>();
    //allocate(layout)?返回NonNull<u8>,
    //NonNull<u8>::<MaybeUninit<T>>::cast()返回
    //NonNull<MaybeUninit<T>>
    let ptr = alloc.allocate(layout)?.cast();
    //as_ptr 成为 *mut MaybeUninit<T>类型裸指针
    unsafe { Ok(Box::from_raw_in(ptr.as_ptr(), alloc)) }
}

pub unsafe fn from_raw_in(raw: *mut T, alloc: A) -> Self {
    //使用Unique封装* mut T, 并拥有了*mut T指向的变量的所有权
    Box(unsafe { Unique::new_unchecked(raw) }, alloc)
}


```

以上代码可以看到，`NonNull<u8>` 可以直接通过cast 转换为 `NonNull<MaybeUninit<T>>`，这是另一种 `MaybeUninit<T>` 的生成方法，直接通过指针类型转换将未初始化的内存转换为 `MaybeUninit<T>`。

RUST的全局变量内存探讨

RUST支持const 及 static类型的全局变量，且static支持可写操作。所有对static的写操作都是unsafe的。需要特别注意的，全局变量不支持非Copy trait类型所有权转移，这也很好理解，所有权转移实际上一个内存"move"的操作。但static变量的内存显然是没有办法"move"的。

static这一性质导致如果要有转移所有权的操作，必须使用mem::replace的方式进行，RUST标准库中很多类型基于mem::replace实现类型自身的replace方法或take方法。

C/C++程序员比较习惯于设计全局变量及其变种静态方法，RUST的全局变量所有权的限制会对这个设计思维有较大的冲击 推荐这篇全局变量的链接：[rust-global-variable](#)
下面是摘录的一张图 全局变量使用

RUST所有权，生命周期，借用探讨

RUST在定义一个变量时，实际上把变量在逻辑上分成了两个部分，变量的内存块与变量内容。

变量类型定义了内存块内容的格式，变量声明语句定义了一个内存块，变量初始化赋值则在内存块中写入初始化变量内容。

所有权指变量内容的独占性。所有权转移指的是变量内容在不同的内存块之间的转移(浅拷贝)。当变量内容转移到新的内存块，旧的内存块就失去了这个变量内容的所有权。由此可

见，变量名实际仅代表一个内存块，内存块的变量内容与变量名是一个暂时关联关系，RUST定义这种关联关系为**绑定**。

设计所有权的目的是保证对变量进行清理操作的正确，如果一个变量内容在多个内存块中有效，变量清理的正确性用静态编译的方法无法保证。

这里有个例外，就是实现Copy trait的类型变量不做所有权转移操作，实现Copy trait的类型可通过栈拷贝完成变量内容赋值，清理也可以仅通过通常的调用栈返回完成。

RUST被设计成自动调用变量类型的drop以完成清理，对变量的生命周期跟踪成为一个必然的选择，在判断变量的生命周期终结的时候调用变量的drop函数。

RUST采用生命周期仅与内存块(变量名)相关联的设计，这样的设计容易对生命周期进行跟踪。没有绑定所有权的内存块在生命周期终结不做任何操作，拥有所有权的内存块生命周期终结会自动触发变量的drop操作。

如果仅仅考虑drop操作，那生命周期的方案不会太复杂，但RUST决定用生命周期同时解决另一个问题，变量引用导致的野指针问题。因为所有权的关系，RUST将变量引用改了一个RUST的名字——**借用**，意味着对所有权的借用。用生命周期解决借用导致的野指针问题思路很简单，就是借用的生命周期应该短于所有权的生命周期。但这个简单的思路却需要极为复杂的设计来完成，对这个复杂设计的理解也成了RUST最被人诟病的点。

理想的生命周期方案是完全由编译器搞定，程序员不要参与。但这显然不可能，编译器没办法在所有的情况下都能够完成全部的推断，势必需要程序员在编码中给出提示。生命周期因此成为rust的一个语法部分。

首先，生命周期被设计成一种实现继承语法的**类型**，**每一个生命周期都是一个类型**，不同的生命周期之间的关系用类型继承语法来完成。生命周期类型的继承具体而言：假设有两个生命周期类型A和B，如果A完全被B包含在内，那就说B继承于A。A是基类型，B是子类型。从继承的概念，B类型能被转换为A类型，A类型无法转换为B类型。也就是说B类型的值能赋给A类型的变量，A类型的值无法赋给B类型变量。生命周期是类型这一点与直观感觉有区别，毕竟，一个作用域给人的感觉就应该是个值。但是，用类型这个方案：

1. 可以利用类型系统来完成生命周期方案，没有给rust编译器增加太大的负担，代码也几乎不受影响。
2. 利用继承语法，在变量赋值时根据类型能否转换完成生命周期长短的判断，是极为巧妙的，简化的，自然的设计。

因为生命周期仅对内存块有意义，而在转移所有权的操作中，是两个不同的内存块发生的联系，他们的生命周期彼此独立。所以所有权转移时，所有权变量的类型层次上不涉及生命周期类型转换。如果类型成员中有引用，则见下面的内容。当对一个引用类型变量做赋值时，便出现了生命周期类型转换，举例分析如下：

1. 当声明一个类型引用的变量时，例如：`let a: &i32` 实质声明了一个i32类型引用的内存块，这个内存块有一个生命周期泛型，假设为'a，
2. 假设要对此变量赋值为另一个变量的引用，例如：`let b:i32 = 4; a = &b; &b` 实质是对b的内存块进行引用，该内存块的生命周期假设为'b，

3. 赋值实质是将一个&'b i32 类型的变量赋值给&'a i32类型变量。则必然发生**类型转换**关系，这时，只有当'b是'a的子类型时，即'b长于'a时，这个**类型转换**才能被编译器认为正确。

以上实际就是生命周期的奥秘所在了，RUST对生命周期设计的关键点就是：

1. 在变量赋值时捕捉触发生命周期类型转换的情况
2. 确保类型转换不正确时，给出生命周期不正确的编译错误警告。

但是，还有些其他情况需要考虑。

因为引用类型是泛型的一种，那由泛型派生的类型的赋值也就会出现类型转换的问题，例如：`*const T, *mut T, Box<T>, ...` 具体的类型可以参考RUST Reference。这时，需要由T的继承关系推断出派生类型的继承关系。这就是变异性Variance特性存在的意义，Variance存在三种情况

1. 协变covariant, 泛型是子类，派生类型也是子类。泛型是父类，派生类型也是父类
2. 逆变contravariant, 泛型是子类，派生类型是父类。泛型是父类，派生类型是子类
3. 不变invariant, 泛型的子类还是父类都推导不出派生类型是否是子类还是父类。

复合类型之间的继承的关系可根据成员变异性得出。因为在RUST编程中引用派生类型及其赋值操作的广泛性，所以变异性是一个重要的需要被理解的概念。完整的变异性请参考RUST Reference。

对生命周期推断的复杂性，RUST采用了每个函数自决的方式(推断)。

每一个函数的生命周期类型转换处理正确与否在函数内完成判断(以下为根据逻辑进行的推断，可能不准确)：

1. 函数作用域会有一个生命周期泛型；
2. 函数的定义会定义函数参数的生命周期泛型，以及这些生命周期泛型之间的继承关系。显然，函数作用域生命周期泛型是所有输入参数生命周期泛型的基类型
3. 函数的定义会定义输出的生命周期泛型，以及输出的生命周期泛型与输入参数生命周期泛型的继承关系。如果输出是一个借用或由借用派生的类型或者有借用成员的复合类型，则输出的生命周期泛型必须是某一输入生命周期泛型的基类型。
4. 编译器会分析函数中的作用域，针对每个作用域生成生命周期泛型，并形成这些生命周期泛型之间的继承关系，当然，函数内所有生命周期泛型都是函数作用域生命周期泛型的基类型。
5. 根据这些生命周期泛型及他们之间的继承关系，处理函数内操作时引发的生命周期泛型类型转换，并对错误的转换做出错警告。
6. 如果调用了其他函数，则对调用函数的输入参数及输出之间的转换是否正确判断转移至调用函数。

如果一个复合类型内部存在引用类型成员或递归至引用类型成员，则必须明确此复合类型的生命周期泛型与成员生命周期泛型的继承关系。一般复合类型的生命周期应该是基类

型。

RUST编译器做了很多工作以避免生命周期泛型标注在代码中出现。这部分的工作仍然在持续进行中。举几个生命周期的例子：

```
impl *const T{
    pub const unsafe fn as_ref<'a>(self) -> Option<&'a T> {
        if self.is_null() { None } else { unsafe { Some(&*self) } }
    }
}
```

因为*const T没有生命周期类型与之相关。所以上面这个函数必须声明一个生命周期泛型用于标注返回的生命周期，此泛型独立存在，不与其他生命周期泛型有关系。因此返回的引用变量的生命周期完全决定于调用此函数的代码定义。因为返回引用的生命周期应短于self指向的内存块的生命周期，这只能由调用此函数的代码即程序员来保证，RUST编译器此时无能为力。

RUST中，对于申请的堆内存内存块，通常将其与一个位于栈内存空间的智能指针的类型变量相结合。智能指针类型变量生命周期终止时，调用drop方法释放堆内存的内存块。智能指针类型通常会提供leak函数，将堆内存的内存块与智能指针类型的关联切断。这通常是一个中间状态，需要尽快再将堆内存与另一个智能指针类型的变量建立联系，以便其能重新被纳入生命周期的体系中。

小结

本章主要分析了RUST标准库内存相关模块，内存相关模块代码多数不复杂，主要是要对内存块与类型系统之间的转换要有比较深刻的理解，并能领会在实际编码过程中在那些场景会使用内存相关的代码和API。RUST的内存安全给编码加了非常多的限制，有些时候这些限制只有通过内存API来有效的突破。如将引用指向的变量所有权转移出来的take函数。后继我们会看到几乎每个标准库的模块都大量的使用了ptr, mem模块中的方法和函数。只要是大型系统，不熟悉内存模块的代码，基本上无法做出良好的程序。