RUST标准库体系概述

RUST语言的设计目标是能编写操作系统内核的系统编程语言。使用静态编译，不采用GC机制，保证开发出的应用极高性能；具备现代编程语言的高效率语法, 并在编译阶段就保证内存安全，并发安全，分支安全等安全性。 现代高级语言的标准库是语言的一个紧密的组成部分，标准库负责语言众多关键特性实现。RUST的库也是如此，但与其他采用GC方案的语言不同，其他语言编程目标是在操作系统之上运行的用户态程序，只需要考虑一种模型。RUST则需要考虑操作系统内核与用户态两种模型。C语言在解决这个问题的方法是只提供用户态的标准库，操作系统内核的库由各操作系统自行实现。 RUST的现代语言特性决定了标准库无法象C语言那样把操作系统内核及用户态程序区分成完全独立的两个部分，所以只能更细致的设计，做模块化的处理。RUST标准库体系分为三个模块：语言核心库–core; alloc库；用户态 std库。

core库

RUST语言核心库，适用于操作系统内核及用户态，包括RUST的基础类型，基本Trait, 类型行为函数，其他函数等内容。core库是硬件架构和操作系统无关的可移植库。主要内容：

编译器内置intrinsics函数

包括内存操作函数，算数函数，位操作函数，原子变量操作函数等， 这些函数通常与CPU硬件架构紧密相关，且一般需要汇编来提供最佳性能。 intrinsic函数实际上也是对CPU指令的屏蔽层。

基本Trait

推荐这个链接 本节给出core库中的Trait一览

运算符（ops）Trait

主要是各种用于表达式的RUST符号重载，包括算数计算符号，逻辑运算符号，位操作符号，解引用(*)符号, [index]数组下标符号，
`../start..end/start../start..=end/..end/..=end` Range符号， ?号，||{...}闭包符号等，
RUST原则是所有的运算符号都要能重载, 所以所有运算操作都定义了重载Trait。

编译器内部实现的派生宏 Trait

如果类型结构中的每一个变量都实现了该Trait, 则此结构的该Trait可通过派生宏实现.
Clone, Copy: Copy浅复制，Clone提供深复制.
Debug: 类型的格式化输出.
Default: 类型的default值

Eq, Ord，PartialEQ, PartialOrd: 实现后可以对类型的变量做大,小,相等比较.
Sync, Send: 实现此Trait的类型变量的引用可以安全在线程间共享.
Hash: 实现结构的整体Hash值，这个Trait Hash是因为复杂才被加入，意义没有前面的大

Iterator

迭代器，RUST基础构架之一，也是RUST所有学习资料的重点。不赘述，本书后面章节将关注其代码实现。

类型转换Trait

AsRef， AsMut, From，Into，TryFrom，TryInto, FloatToInt, FromStr

字符串Trait

此处略，后面章节单独做分析

异步编程Trait

此处略，后面章节单独分析

内存相关Trait

此处略，后面章节单独分析

基本数据类型

包括整数类型，浮点类型，布尔类型，字符类型，单元类型，内容主要是实现运算符Trait, 类型转换Trait, 派生宏Trait等，字符类型包括对unicode，ascii的不同编码的处理。整数类型有大小端变换的处理。

数组、切片及Range

主要为类型结构对Iterator Trait, 运算符Trait, 类型转换Trait, 派生宏Trait的实现。

Option/Result/Marker等关键的语言级别Enum类型

RUST安全特性的重点，也是各种学习资料的重点，不赘述，后面章节将关注其一些代码实现

RUST内存相关类型及内容

alloc, mem, ptr等模块，RUST的内存操作，后继章节重点详述。

RUST字符串相关库

字符串str，string，fmt, panic, debug, log等

RUST时间库

Duration等

alloc库

alloc库主要实现需要进行动态堆内存申请的智能指针类型，集合类型及他们的行为，函数，Trait等内容，仅建立在core库模块之上。std会对alloc模块库的内容做重新的封装。alloc库适用于操作系统内核及用户态程序。 包括： 1.基本内存申请；Allocator Trait; Allocator的实现结构Global 2.基础智能指针：Box, Rc, 3.动态数组内存类型: RawVec, Vec 4.字符串类型：&str, String 5.并发编程指针类型: Arc 6.指针内访问类型: Cell, RefCell 还有些其他类型，一般仅在标准库内部使用，后文在需要的时候再介绍及分析。

std库

std是在操作系统支撑下运行的只适用于用户态程序的库，core库实现的内容基本在std库也有对应的实现。其他内容主要是将操作系统系统调用封装为适合rust特征的结构和Trait,包括： 1.进程，线程库 2.网络库 3.文件操作库 4.环境变量及参数 5.互斥与同步库，读写锁 6.定时器 7.输入输出的数据结构， 8.系统事件，对epoll,kevent等的封装 可以将std库看做基本常用的容器类型及操作系统封装库。

小结

RUST的目标和现代编程语言的特点决定了它的库需要仔细的模块化设计。RUST的alloc库及std库都是基于core库。RUST的库设计非常巧妙和仔细，使得RUST完美的实现了对各种硬件架构平台的兼容，对各种操作系统平台的兼容。

RUST泛型小议

RUST是一门生存在泛型的基础之上的语言。其他语言不使用泛型也不影响编程，泛型只是一个语法中的强大工具。与之相对，RUST离开泛型就无法编写程序，泛型与语法共生。

直接针对泛型的方法和trait实现

其他语言的泛型，是作为类型结构体成员，或是函数的输入/返回参数出现在代码中，是配角。RUST的泛型则可以作为主角，可以直接对泛型实现方法和trait。如：

```rust
//T:?Sized是所有的类型， 不带约束的T实际是 T:Sized
//即类型内存空间固定，所以 T:?Sized才是全部的类型
impl<T: ?Sized> Borrow<T> for T {
    fn borrow(&self) -> &T {
        self
    }
}

impl<T: ?Sized> BorrowMut<T> for T {
    fn borrow_mut(&mut self) -> &mut T {
        self
    }
}
```

以上代码对所有的类型都实现了Borrow的trait。
直接针对泛型做方法和trait的实现是强大的工具，它的作用：

- 针对泛型的代码会更内聚，方法总比函数具备更明显的模块性
- 逻辑更清晰及系统化更好
- 具备更好的可扩展性
- 更好的支持函数式编程

泛型的层次关系

RUST的泛型从一般到特殊会形成一种层次结构，有些类似于面对对象的基类和子类关系：
最基层： `T:?Sized` ?Sized的约束表明了所有的类型

一级子层： 默认内存空间固定类型 `T` ； 裸指针类型 `* const T/* mut T` ; 切片类型 `[T]` ; 数组类型 `[T;N]` ; 引用类型 `&T/&mut T` ; trait约束类型 `T:trait` ; 泛型元组 `(T, U...)` ; 泛型复合类型 `struct <T>; enum <T>; union<T>` 及具体类型 `u8/u16/i8/bool/f32/&str/String...`

二级子层： 对一级子层的T赋以具体类型 如： `* const u8; [i32]` ，或者将一级子层中的T再次做一级子层的具化，例如： `* const [T]; [*const T]; &(*const T); * const T where T:trait; struct <T:trait>`

可以一直递归下去。 显然，针对基层类型实现的方法和trait可以应用到层级高的泛型类型中。 例如：

```rust
impl <T> Option<T> {...}
impl<T, U> Option<(T, U)> {...}
impl<T: Copy> Option<&T> {...}
impl<T: Default> Option<T> {...}
```

以上是标准库对Option 的不同泛型的方法实现定义。遵循了从一般到特殊的规则。

类似的实现再试举如下几例：

```
impl <T:?Sized> *const T {...}
impl <T:?Sized> *const [T] {...}
impl <T:?Sized> *mut T{ ...}
impl <T:?Sized> *mut [T] {...}
impl <T> [T] { ...}
impl <T, const N:usize> [T;N]{...}
impl AsRef<[u8]> for str {...}
impl AsRef<str> for str {...}
```

当在代码中需要实现一个新的trait时，都要考虑其是否具备满足所有的类型或某类特殊类型的集体需求，如果是，就可以考虑基于泛型实现。当然，要按照泛型层级从一般到特殊来编写代码。
基于泛型来实现trait或方法，是一种微妙的提升代码良好设计的语言特点，留给读者去体会。

RUST内存安全杂述

经过对标准库源代码的学习，很容易能够发现，rust编译器提供的安全特性实际很少：
明确的安全特性：

1. 变量必须初始化之后才能使用；
2. 引用必须是内存对齐的，引用指向的变量必须已经初始化；
3. 模块成员默认私有
4. 严格的类型及类型无效值限制
5. 基础类型都满足Copy/Send/Sync auto trait
6. if及match的分支语法

明确的不安全特性：

1. 裸指针解引用；
2. 线程间转移变量必须支持Send, 共享变量必须支持Sync
3. 所有的FFI调用,unsafe intrinsic函数调用
4. 对类型产生无效类型值
5. 嵌入式汇编使用
6. 含有以上成分的代码单元

为安全提供的工具

1. 所有权，生命周期，自动drop；
2. 自动解引用

可以看到，编译器提供的安全特性实际上只是实现内存安全的基础设施。RUST程序员仍然需要依靠这些基础设施来构建整个程序的安全大厦。
标准库提供了大量的语言类基础类型结构及操作系统相关的基础类型结构，并完成了这些基础类型结构的安全。如果仅仅使用标准库提供的类型结构，则一般不必额外考虑内存安全问题。但是如果超纲，则新的类型的内存安全必须由创建此类型的代码来做保证，而在当前RUST生态还不完善的情况下，这是必然要发生的事。

RUST的内存安全指的是编译器**提供基础设施**，程序员利用基础设施**创建**内存安全的类型结构使编译器能够保证此类型的内存安全。

在标准库的实践中，可以发现安全是由大量的不安全代码所实现。尤其在操作系统适配的那一层次上，rust实际就是语法改变了的C，为了提高性能所采用的那些技巧和C毫无二致，令人发指，但又不得不佩服。这些代码的安全高度依赖于程序员，语言本身基本没有保障。

RUST的安全本质上仍然是一批高水平程序员实现的一个语言安全框架。

RUST官方语言库安全戏法的一些套路：
安全类型结构基本上是一个封装类型结构， 真正要操作的原始变量被封装在内，并且，此封装类型结构拥有原始变量的所有权。例如：RefCell, 智能指针，Rc，Arc，Mutex等。用于实现不同场景下的安全。
实现封装类型结构的Drop trait，完成生命周期结束时需要的清理操作，例如，释放堆内存，关闭文件描述符等
实现对封装类型结构的借用函数，对于复杂操作，往往需要一个额外的专用于借用的封装类型结构及可变借用的封装类型结构，如Ref, RefMut分别是RefCell的借用及可变借用的结构。不同的封装类型根据意义的不同会有不同的借用操作，如RefCell的borrow(), borrow_mut（）， Rc的clone（）， Mutex的lock()，但都是满足在某种安全机制下的获取原始变量的借用。
实现对于封装类型结构或者借用封装类型结构的Deref trait及 DerefMut trait，得到原始变量的引用或可变引用，从而实现对于原始结构的访问及更改。
实现对于借用封装类型的Drop trait，完成针对借用的清理工作，如减少计数，释放锁等。
RUST的安全实际上都是由这些安全封装类型完成。可以说，每一个安全封装类型都是程序员的血汗得来的教训。

从后继标准库的源代码分析中，可以真实的熟悉RUST的安全戏法。