学习Rust设计模式0：简介

从今天开始，将和大家一起来学习《Rust设计模式》这本书。本文档为学习这本书的学习笔记。
《Rust设计模式》原书英文版地址为：https://rust-unofficial.github.io/patterns/translations.html
《Rust设计模式》原书中文版地址为：https://fomalhauthmj.github.io/patterns/intro.html

所有笔记也放置于公众号“令狐一冲”上，有兴趣的小伙伴可以关注公众号：

简介说明：

- 设计模式是解决编写软件时常见问题的方法。
- 反面模式是解决这些相同的常见问题的方法。设计模式带来好处，反面模式带来问题。
- 惯例做法是编码时需要遵循的准则。如果你要打破它们，那么最好有一个好的理由。

令狐一冲 于2023年2月18日

学习Rust设计模式1：使用借用类型作为参数

1、通常做法

编码时应该总是**倾向于**使用借用类型而不是借用所有类型。
例如：对于String类型来说，应该倾向于使用使用&str，而不是&String；对于T类型来说，应该倾向于 使用&[T]而不是&Vec[T]；应该倾向于使用&T而不是&Box<T>。

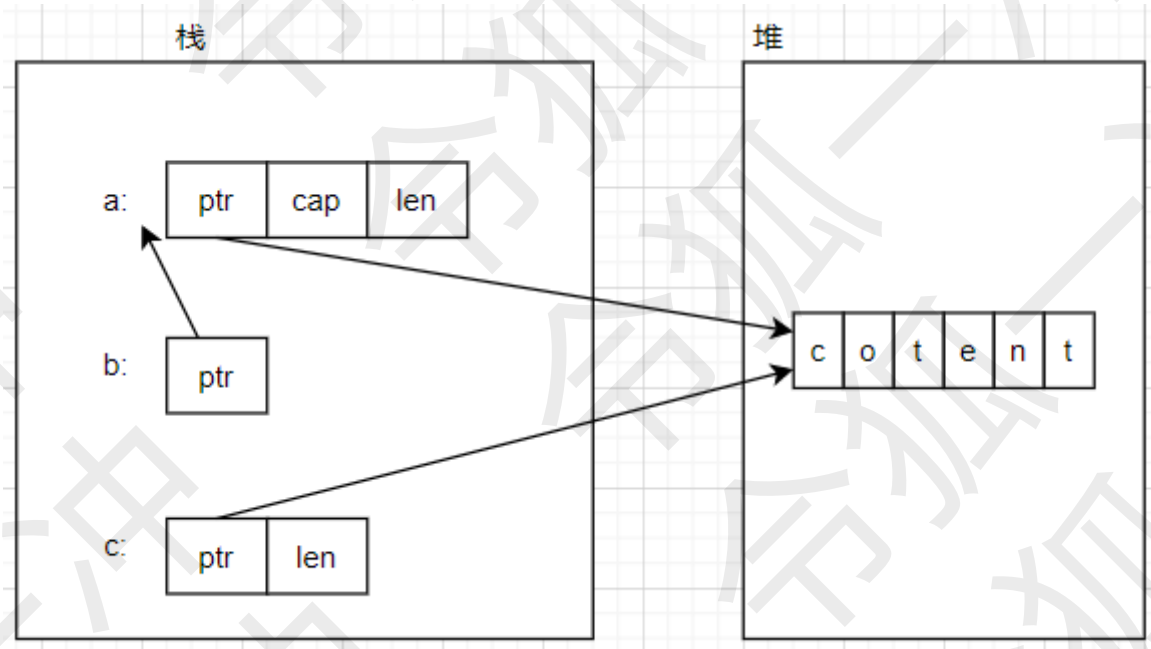
2、为什么要这样做

2.1 **原因1**

在《Rust设计模式》书中，提出“使用借用类型可以避免已经提供一层间接性的所有类型上的多层间接“。我们考虑下面几行代码：

```
let a: String = "content".to_string();
let b: &String = &a;
let c: &str = a.as_str();
```

其在内存中的布局如下：



那么结合上面的图，我们可以有如下理解：

- String类型具有一层间接，因为String类型的本质是一个具有三个字段的胖指针（三个字段分别是ptr、cap、len，ptr指向在堆上的具体的内容）；
- &String具有两层引用，是因为是String的基础上，加了&，所以b实际上指向的是a，而不是堆上的内容；
- &str类型也是一个胖指针，直接指向栈上的内容。

所以对比&String和&str，显然使用&str的效率更高。
所以从这个层面来说，这条规则也可以换种说法：应该倾向于使用间接层面更少的方式。

2.2 **原因2**

我们再考虑下面的例子：

```
fn print_use_string(word: String) {
    println!("{:?}", word);
}

fn print_use_string1(word: &String) {
    println!("{:?}", word);
}

fn print_use_str(word: &str) {
    println!("{:?}", word);
}

fn main() {
    let a = "content".to_string();
    print_use_string(a);
    // println!("a: {:?}", a);   //This will error

    let b = "content".to_string();
    print_use_string1(&b);
    println!("b: {:?}", b);
    // print_use_string1("content");  //This will error
    print_use_string1(&"content".to_string());

    let c = "content".to_string();
    print_use_str(&c);
    println!("c: {:?}", c);
    print_use_str("content");
}
```

从上面的例子，我们可以看出，在有些情况下，我们在函数中并不需要拥有变量的所有权（如print_use_string和print_use_str功能一样），所以传引用即可，而传&str比传&String效率更高。所以，在这个层面，我们也应该更倾向于使用借用类型而不是借用所有类型。

学习Rust设计模式2：使用format！串联字符

1、通常做法

对可变String操作时，应该**倾向于使用**format！，这会更加方便，尤其是在有字面和非字面字符串混合的地方。

2、原因

使用format！会更加简洁，考虑如下例子：

```
fn say_hello1(name: &str) -> String {
    let mut result = "Hello ".to_owned();
    result.push_str(name);
    result.push('!');
    result
}

fn say_hello2(name: &str) -> String {
    format!("Hello {}!", name)
}

fn main() {
    let s = "world";
    println!("{:?}", say_hello1(s));
    println!("{:?}", say_hello2(s));
}
```

上面的例子中，say_hello1和say_hello2具有同样的功能，但是很显然使用format！代码更简洁。

学习Rust设计模式3：构造器和默认构造器

1、通常做法

Rust中，通常使用一个关联函数new来创建一个对象，通过Default trait来支持默认构造器。

2、示例

直接看下面代码：

```rust
struct Person {
    name: String,
    age: u32,
}

impl Person {
    // 使用new函数创建对象
    fn new(name: &str, age: u32) -> Self {
        Person { name: name.to_string(), age }
    }

    fn print(&self) {
        println!("Name: {:?}", self.name);
        println!("Age: {:?}", self.age);
    }
}

// 为Person实现Default trait
impl Default for Person {
    fn default() -> Self {
        Self { name: "".to_string(), age: 0 }
    }
}

fn main() {
    let alice = Person::new("alice", 20);
    alice.print();

    let default = Person::default();
```

```
    default.print();
}
```

在上面的例子中，我们为Person显式的实现了Default trait。其实对于复合类型来说，如果每个成员都实现了Default Trait，则我们可以直接使用#[derive(Default)]来实现Default，而不必显式的实现。我们可以直接在Person结构体上加上该宏来实现，如下：

```
[derive(Default))]
struct Person {
    name: String,
    age: u32,
}
```

学习Rust设计模式4：把集合当成智能指针

1、通常做法

通过为集合实现Deref trait，提供其拥有和借用的数据视图。

2、示例

考虑Vec<T>，Vec<T>是一个拥有T的集合，然后通过实现Deref完成&Vec<T>到&[T]的隐式解引用，从而提供借用T的集合（即&[T]）。

如下代码：

```
impl<T> Deref for Vec<T> {
    type Target = [T];
    fn deref(&self) -> &[T] {
        unsafe {
            std::slice::from_raw_parts(self.ptr.as_ptr(), self.len)
        }
    }
}
```

原因：Vec<T>提供拥有T的集合，&[T]提供借用T的集合。大部分情况下，只需要借用视图，提供两种方式，让用户在使用时在借用和拥有之间做出选择。

学习Rust设计模式5：在析构器中做最终处理

1、通常做法：

Rust中，通常在析构函数中运行退出前必须运行的代码。

2、示例：

```
#[derive(Debug)]
struct A(u8);
impl Drop for A {
    fn drop(&mut self) {
        println!("A exit");
    }
}
```

3、需要注意的问题：

即使程序崩溃，仍然会运行panic函数；但是如果drop函数中存在无线循环或者运行函数退出前崩溃，则不会运行drop函数剩下的部分。示例如下：

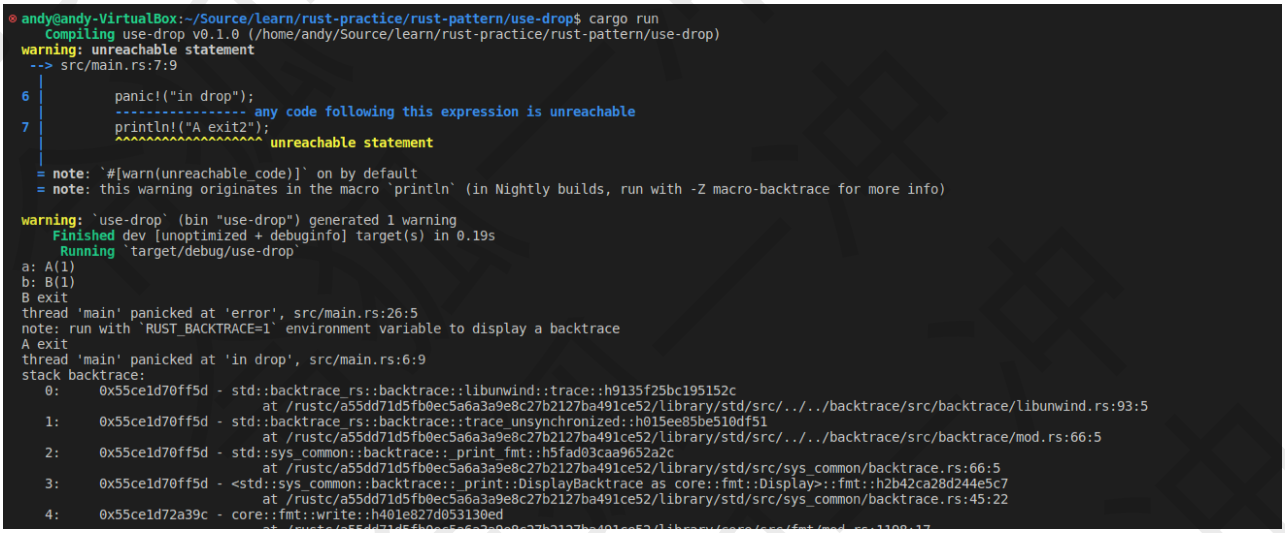
```
struct A(u8);
impl Drop for A {
    fn drop(&mut self) {
        println!("A exit");
        panic!("in drop");
        println!("A exit2");
    }
}

#[derive(Debug)]
struct B(u8);
impl Drop for B {
    fn drop(&mut self) {
        println!("B exit");
    }
}
```

```rust

fn main() {
    let a = A(1);
    {
        let b = B(1);
        println!("a: {:?}", a);
        println!("b: {:?}", b);
    }
    panic!("error");
}
```

上述代码运行如下：



可以看到panic发生后，A的drop函数仍然会运行，但是因为在A的drop函数中也发送了panic，所以“A exit2”不会打印。

学习Rust设计模式6：使用take和replace来保留所有值

1、通常做法：

对于枚举类型：

```
enum MyEnum {
    A { name: String, x: u8 },
    B { name: String },
}
```

使用std::mem::take()和std::mem::replace()在不可隆name的情况下修改name。

2、示例：

如下代码：

```
use std::mem;
#[derive(Debug)]
enum MyEnum {
    A { name: String, x: u8 },
    B { name: String },
}

fn convert_a_to_b(e: &mut MyEnum) {
    if let MyEnum::A { name, x: 0 } = e {
        *e = MyEnum::B {
            name: mem::take(name),
        }
    }
}

fn convert_a_to_b2(e: &mut MyEnum) {
    if let MyEnum::A { name, x: 0 } = e {
        *e = MyEnum::B {
            name: mem::replace(name, String::new()),
        }
    }
```

```rust
}

// fn convert_a_to_b3(e: MyEnum) -> MyEnum {
//     if let MyEnum::A { name, x: 0 } = e {
//         return MyEnum::B{ name: name.clone()}
//     } else {
//         return e
//     }
// }

fn main() {
    let mut a = MyEnum::A {
        name: "A".to_string(),
        x: 0,
    };
    println!("Before Convert, a is {:?}", a);

    convert_a_to_b(&mut a);
    println!("After convert, a is {:?}", a);

    convert_a_to_b2(&mut a);
    println!("After convert2, a is {:?}", a);

    // let a =  convert_a_to_b3(a);
    // println!("After convert3, a is {:?}", a);
}
```

这种方式可以不用clone，不存在内存分配。

学习Rust设计模式7：trait对象动态分发

1、描述：

Rust的trait对象是动态分发。

2、示例：

```rust
use std::io;
use std::fs;

fn main() -> Result<(), Box<dyn std::error::Error>> {
    let arg = "-";

    // These must live longer than `readable`, and thus are declared first:
    let (mut stdin_read, mut file_read);

    // We need to ascribe the type to get dynamic dispatch.
    let readable: &mut dyn io::Read = if arg == "-" {
        stdin_read = io::stdin();
        &mut stdin_read
    } else {
        file_read = fs::File::open(arg)?;
        &mut file_read
    };

    // Read from `readable` here.

    Ok(())
}
```

此知识点，Rust的trait对象，是动态分发，运行时确定具体的对象。

其它知识点：Rust默认对代码单态化处理，每种类型的代码会进行优化，所以效率很高，但是代码也就会比较臃肿。

学习Rust设计模式8：FFI错误处理惯常做法

1、惯常做法一：简单枚举类型应该被转换为整数，并作为代码返回
代码示例如下：

```
enum SimpleError {
    IOError = 1,
    FileCorrupted = 2,
}

impl From<SimpleError> for libc::c_int {
    fn from(e: SimpleError) -> libc::c_int {
        (e as i8).into()
    }
}
```

2、惯常做法二：结构化枚举应该被转换为整数代码，并有字符串错误信息作为提示细节
代码示例如下：

```
enum SimpleError2 {
    IOError(String) ,
    FileCorrupted(String),
}

impl From<SimpleError2> for libc::c_int {
    fn from(e: SimpleError2) -> libc::c_int {
        match e {
            SimpleError2::IOError(_) => 1,
            SimpleError2::FileCorrupted(_) => 2,
        }
    }
}

#[no_mangle]
pub extern "C" fn return_err2(t: libc::c_int) -> *mut libc::c_char {
    let err = produce_err2(t as u64);
```

```
    let err_str = match err {
        SimpleError2::IOError(e) => format!("Io error: {:?}", e),
        SimpleError2::FileCorrupted(e) => format!("FileCorrupted: {:?}", e),
    };

    let c_error = unsafe {
        let malloc: *mut u8 = libc::malloc(err_str.len() + 1) as *mut _;
        if malloc.is_null() {
            return std::ptr::null_mut();
        }
        let src = err_str.as_bytes().as_ptr();
        std::ptr::copy_nonoverlapping(src, malloc, err_str.len());
        std::ptr::write(malloc.add(err_str.len()), 0);
        malloc as *mut libc::c_char
    };
    c_error
}
```

然后在导出给外部语言使用的函数中，对于具体的错误，可以加上字符串错误信息作为提示。

3、惯常做法三：自定义错误类型应该变的透明，用c表示
示例代码如下：

```
struct SimpleError3 {
    expected: char,
    line: u32,
    ch: u16,
}

#[repr(C)]
pub struct parse_error {
    pub expected: libc::c_char,
    pub line: u32,
    pub ch: u16,
}

impl From<SimpleError3> for parse_error {
    fn from(s: SimpleError3) -> parse_error {
        let SimpleError3 { expected, line, ch} = s;
```

```
        parse_error {
            expected: expected as libc::c_char,
            line,
            ch,
        }
    }
}
```

原因：保证外部语言清晰的获取错误信息，同时完全不影响Rust代码的API。

学习Rust设计模式9：FFI接受字符串处理

1、惯常做法

当FFI接受字符串时，应该遵循：

- 使用外部字符串时尽量是借用，而不是直接复制它们；
- 尽量减少从C风格字符串转换到Rust字符串时的复杂性，尽量减少unsafe代码量。

2、第一条的原因

因为C语言中使用的字符串和Rust中使用字符串存在不同的行为：

- C语言字符串时无终止的，Rust字符串会存储其长度；
- C语言字符串可以包含任意非零字节，Rust字符串必须是UTF-8；
- C语言字使用unsafe的指针操作字符串，Rust使用安全的方法与字符串进行交互。

3、第二条的原因

显而易见，不做解释

4、其它

Rust标准库提供了CString和&CStr，是String和&str相对于C语言的等价表示，使用它们可以降低Rust和C之间字符串操作代码的复杂性，也可以减少unsafe的代码量。

学习Rust设计模式10：FFI传递字符串

1、通常做法

向FFI函数传递字符串应遵循四个原则：

- 字符串的生命周期尽可能长；
- 转换过程尽量减少unsafe代码；
- 如果C代码可以修改字符串，使用Vec而不是CString；
- 除非外部API要求，否则字符串的所有权不应该转移给被调用者。

2、示例

重点看看下面这个例子（非常重要）：

```rust
extern "C" {
    fn set_err(message: *const libc::c_char);
}

// 正确示范
fn report_error_to_ffi<S: Into<String>>(err: S) -> Result<(), std::ffi::NulError> {
    let c_err = std::ffi::CString::new(err.into())?;

    unsafe {
        set_err(c_err.as_ptr());
    }

    Ok(())
}

// 错误示范
fn report_error<S: Into<String>>(err: S) -> Result<(), std::ffi::NulError> {
    unsafe {
        // SAFETY: whoops, this contains a dangling pointer!
        set_err(std::ffi::CString::new(err.into())?.as_ptr());
        //理解等价于如下：
        //let *mut ptr = null;
        //{
        //    let c_err = std::ffi::CString::new(err.into())?;
```

```
25          //    ptr = c_err.as_ptr();
26          //}
27          //set_err(ptr);
28      }
29      Ok(())
30  }
```

第一种方式在unsafe代码块之外创建CString，其作用域范围一直到13行，因此在unsafe中使用其指针一定是有效的；第二种方式相当于用临时变量直接as_ptr()，无法保证在unsafe中一直有效。
按照《Rust设计模式》书中的说法，第二种方式将导致一个悬垂指针。

学习Rust设计模式11：Option的迭代

1、描述：

Option可以被看做一个包含零或一个元素的容器。因为它实现了trait IntoIterator，所以可以用来迭代。

2、示例

这条规则其实就是想说Option实现了trait IntoIterator，可以进行迭代相关操作，所以举例如下：

```
fn main() {
    let a = Some("a string");
    let mut s1 = vec!["a", "b", "c"];
    s1.extend(a);
    println!("s1: {:?}", s1);

    println!("++++++++++++++++++++++++++");

    let b = Some("b string");
    let s1 = vec!["d", "e", "f"];

    for s in s1.iter().chain(b.iter()) {
        println!("item: {}", s);
    }
}
```

学习Rust设计模式12：传递变量到闭包

1、描述：

默认情况下，闭包通过借用捕获其环境。在将变量转移到闭包中时，在单独的作用域中使用变量重绑定。

2、示例

```rust
use std::rc::Rc;

fn main() {
    // 好的示范
    let a = Rc::new(1);
    let b = Rc::new(2);
    let c = Rc::new(3);
    let closure = {
        let b = b.clone();
        let c = c.as_ref();
        move || {
            let ret = *a + *b + *c;
            println!("ret = {:?}", ret);
        }
    };

    closure();

    println!("++++++++++++++++++++++++++++++++");
    // 差的示范
    let a = Rc::new(1);
    let b = Rc::new(2);
    let c = Rc::new(3);
    let b_cloned = b.clone();
    let c_borrowed = c.as_ref();
    let closure = move || {
        let ret = *a + *b_cloned + *c_borrowed;
        println!("ret = {:?}", ret);
    };
```

```
    closure();
}
```

学习Rust设计模式13：公共结构体和枚举类型的可扩展性

1、描述

在某些情况下，库作者可能想在不破坏向后兼容的情况下，为公共结构体添加公共字段或为公共枚举添加新的变体。在Rust中可以使用#[non_exhaustive] 和添加私有字段的方式来达成。

2、示例：

在当前的crate student中，有如下公共结构体：

```
pub struct StudentInfo {
    pub name: String,
    pub age: u32,
    pub number: u32,
}
```

但是我们未来会为StudentInfo添加公共字段。

- **使用non_exhaustive**

示例代码：

```
#[non_exhaustive]
pub struct StudentInfo {
    pub name: String,
    pub age: u32,
    pub number: u32,
}
```

在同一crate中使用该结构体和没加non_exhaustive一样，但是在外部crate使用该结构体时，如下：

```
let charlie = StudentInfo {name: "Charlie".to_string(), age: 18, number: 3};  //报错
```

因此此时创建StudentInfo需要提供创建函数进行。但是可以如下使用：

```
let StudentInfo {name, age, number} = bob; // bob是一个StudentInfo结构
```

- **添加一个私有字段**

除了上面的方式，还有一种就是添加私有字段，不允许外部crate直接创建，示例如下：

```
pub struct StudentInfo {
    pub name: String,
    pub age: u32,
    pub number: u32,
    _b: (), //添加一个私有成员
}
```

学习Rust设计模式14：简单的文档初始化

1、描述

关于编写文档的简化原则

2、示例

请关注下面代码的注释部分，从下面的代码，改进为后面的方式

```
struct Connection {
    name: String,
    stream: TcpStream,
}

impl Connection {
    /// Sends a request over the connection.
    ///
    /// # Example
    /// ```no_run
    /// # // Boilerplate are required to get an example working.
    /// # let stream = TcpStream::connect("127.0.0.1:34254");
    /// # let connection = Connection { name: "foo".to_owned(), stream };
    /// # let request = Request::new("RequestId", RequestType::Get, "payload");
    /// let response = connection.send_request(request);
    /// assert!(response.is_ok());
    /// ```
    fn send_request(&self, request: Request) -> Result<Status, SendErr> {
        // ...
    }


}
```

改进为：

```
struct Connection {
    name: String,
    stream: TcpStream,
```

```
}

impl Connection {
    /// Sends a request over the connection.
    ///
    /// # Example
    /// ```
    /// # fn call_send(connection: Connection, request: Request) {
    /// let response = connection.send_request(request);
    /// assert!(response.is_ok());
    /// # }
    /// ```
    fn send_request(&self, request: Request) {
        // ...
    }
}
```

学习Rust设计模式15：临时可变性

1、描述：

对于临时可变的变量，在可变之后进行重绑定来明确为不可变的变量。

2、示例：

```rust
fn main() {
    let mut data = vec![2, 1, 4, 10, 3, 5];
    data.sort();
    let data = data; // 进行重新绑定，data变为不可变的变量

    println!("{:?}", data[2]);

    // data.push(4); // error, data is immutable

    // 也可以使用如下使用嵌套块，和上面等价
    let data = {
        let mut data = vec![2, 1, 4, 10, 3, 5];
        data.sort();
        data
    };
    println!("{:?}", data[2]);
    // data.push(4); // error, data is immutable
}
```

学习Rust设计模式16：命令模式

1、描述：

命令模式的基本思想是将行动分离成它自己的对象，并将它们作为参数传递。

2、举例

考虑数据库操作create table和add field两个命令，每个命令都有执行和撤销的动作，那么我们想下面几种方式写代码。

（1）定义一个trait，包含execute和rollback两个函数，所有的命令struct必须实现这个trait，代码如下：

```
pub trait Migration {
    fn execute(&self) -> &str;
    fn rollback(&self) -> &str;
}

pub struct CreateTable;
impl Migration for CreateTable {
    fn execute(&self) -> &str {
        ...
    }
    fn rollback(&self) -> &str {
        ...
    }
}

pub struct AddField;
impl Migration for AddField {
    fn execute(&self) -> &str {
        ...
    }
    fn rollback(&self) -> &str {
        ...
    }
}

```

```
struct Schema {
    commands: Vec<Box<dyn Migration>>,
}

impl Schema {
    ...
    fn add_migration(&mut self, cmd: Box<dyn Migration>) {
        ...
    }

}
```

（2）将每个命令的操作创建为函数，并存储函数指针，代码如下：

```
type FnPtr = fn() -> String;
struct Command {
    execute: FnPtr,
    rollback: FnPtr,
}
struct Schema {
    commands: Vec<Command>,
}
impl Schema {
    ...

    fn execute(&self) -> Vec<String> {
        self.commands.iter().map(|cmd| (cmd.execute)()).collect()
    }
    fn rollback(&self) -> Vec<String> {
        self.commands
            .iter()
            .rev()
            .map(|cmd| (cmd.rollback)())
            .collect()
    }
}
n add_field() -> String {
    "add field".to_string()
}
```

```
fn remove_field() -> String {
    "remove field".to_string()
}
fn main() {
    let mut schema = Schema::new();
    schema.add_migration(|| "create table".to_string(), || "drop table".to_string());
    schema.add_migration(add_field, remove_field);
    ...
}
```

使用函数指针是静态分发。

（3）使用Fn trait对象，将实现Fn trait的每个命令分别存储在向量中，代码如下：

```
type Migration<'a> = Box<dyn Fn() -> &'a str>;

struct Schema<'a> {
    executes: Vec<Migration<'a>>,
    rollbacks: Vec<Migration<'a>>,
}
...

fn main() {
    let mut schema = Schema::new();
    schema.add_migration(|| "create table", || "drop table");
    schema.add_migration(add_field, remove_field);
    ...
}
```

3、分析

上面的三种方式都实现了我们的功能，那什么时候使用trait对象，什么时候使用函数指针呢？

- 使用trait对象是动态分发，使用函数指针是静态分发；
- 当命令实现的代码比较多，比较复杂时，建议使用trait对象；
- 当命令实现代码较少时，建议使用函数指针。

换句话说，简单代码建议用静态分发，复杂代码建议动态分发。

学习Rust设计模式17：解释器模式

1、描述

解释器模式（Interpreter Pattern）提供了评估语言的语法或表达式的方式，它属于行为型模式。这种模式实现了一个表达式接口，该接口解释一个特定的上下文。这种模式被用在 SQL 解析、符号处理引擎等。

2、示例

实现一个逆波兰式的解释器，查看原文的示例代码。

学习Rust设计模式18：新类型

1、描述

使用NewType模式来提供类型安全和封装。具体为：使用单个字段的元组作为一个类型的不透明包装

2、分析

这样做有什么作用？主要如下：

- 抽象化，新类型是零成本抽象，实现类型之间共享实现细节，同时精准控制；
- 区分不同的类型（如 struct Miles(f64)和struct Kms(f64)）；

举例如下：

```rust
//1、共享类型，同时实现精准控制：
// (1)personInfo类型共享studentInfo类型，同时又只提供name和id方法
// (2)PersonInfo实现Debug trait可以打印，PersonInfo则没有提供，变成了move语义
#[derive(Debug)]
struct StudentInfo {
    name: &'static str,
    id: &'static str,
    number: &'static str,
}
impl StudentInfo {
    fn new(name: &'static str, id: &'static str, number: &'static str) -> Self {
        StudentInfo {
            name,
            id,
            number: number,
        }
    }

    fn name(&self) -> &'static str {
        self.name
    }

    fn id(&self) -> &'static str {
        self.id
    }
```

```rust

    fn number(&self) -> &'static str {
        self.number
    }
}

struct PersonInfo(StudentInfo);

impl PersonInfo {
    fn new(s: StudentInfo) -> Self {
        PersonInfo(s)
    }

    fn name(&self) -> &'static str {
        self.0.name()
    }

    fn id(&self) -> &'static str {
        self.0.id()
    }
}

struct Miles(f64);
struct Kms(f64);
fn main() {
    let s = StudentInfo {
        name: "Alice",
        id: "123456",
        number: "001",
    };
    let p = PersonInfo::new(s);
    println!("name: {:?}", p.name());
    println!("id: {:?}", p.id());

    let _m = Miles(10f64);
    let _kms = Kms(10f64);
}
```

学习Rust设计模式19：RAII守护对象

1、描述

RAII即Resource Acquisition is Initialization（资源获取即初始化）。该模式具体为：使用RAII对象作为某些资源的守护对象。

2、示例

例如下面的代码，将MyMutexGuard作为Foo获取到锁资源的守护对象。

```rust
use std::cell::RefCell;
use std::ops::Deref;

struct Foo;
impl Foo {
    fn do_something(&self) {
        println!("Do something");
    }
}

struct MyMutex<T> {
    flag: RefCell<bool>,
    data: T,
}

impl<T> MyMutex<T> {
    fn new(t: T) -> MyMutex<T> {
        MyMutex {
            flag: RefCell::new(false),
            data: t,
        }
    }

    fn lock(&self) -> Result<MyMutexGuard<T>, &'static str> {
        while *self.flag.borrow() {}
        *self.flag.borrow_mut() = true;
        MyMutexGuard::new(self)
    }
```

```rust

    fn try_lock(&self) -> Result<MyMutexGuard<T>, &'static str> {
        if *self.flag.borrow() == false {
            self.lock()
        } else {
            Err("Can't get lock")
        }
    }
}

impl<T> Drop for MyMutex<T> {
    fn drop(&mut self) {
        println!("unlock");
    }
}

struct MyMutexGuard<'a, T: 'a> {
    lock: &'a MyMutex<T>,
}

impl<'a, T> MyMutexGuard<'a, T> {
    fn new(lock: &'a MyMutex<T>) -> Result<MyMutexGuard<'a, T>, &'static str> {
        Ok(MyMutexGuard { lock })
    }
}

impl<'a, T> Deref for MyMutexGuard<'a, T> {
    type Target = T;

    fn deref(&self) -> &T {
        &self.lock.data
    }
}

impl<'a, T> Drop for MyMutexGuard<'a, T> {
    fn drop(&mut self) {
        *self.lock.flag.borrow_mut() = false;
    }
}
```

```rust

fn main() {
    let m = MyMutex::new(Foo);
    {
        println!("++++++++++");
        let m1 = m.lock();
        if let Err(e) = m.try_lock() {
            println!("Can't get lock, err: {:?}", e);
        }
        println!("++++++++++");
        m1.unwrap().do_something();
    }
    println!("++++++++++");
    let m2 = m.lock();
    m2.unwrap().do_something();
    println!("++++++++++");
}
```

学习Rust设计模式20：策略模式

1、描述

策略模式是一种实现关注点分离的技术，其基本思想是，给定一个解决特定问题的算法，只在抽象层面定义算法的骨架，将具体的算法分成不同的部分。

2、示例

需求：输入data，生成报告，报告的格式可能是json，也可能是text。

采用策略模式设计如下：

Report提供生成报告功能，Report生成报告时使用Formatter格式化内容生成报告。将Formatter抽象成接口，然后具体的格式化算法（text、json）实现该接口。

这样做的好处是，Report不用关注text、json这些具体算法的实现。

report包的代码如下：

```rust
//report/src/formatter.rs
use std::collections::HashMap;
pub type Data = HashMap<String, u32>;
pub trait Formatter {
    fn format(&self, data: &Data, buf: &mut String);
}
```

```rust
//report/src/report.rs
use crate::formatter::Formatter;
use std::collections::HashMap;

pub struct Report;

impl Report {
    pub fn generate<T: Formatter>(g: T, s: &mut String) {
        let mut data = HashMap::new();
        data.insert("one".to_string(), 1);
        data.insert("two".to_string(), 2);
        g.format(&data, s);
```

```
    }
}
```

json包代码如下：

```
use report::formatter::{Data, Formatter};

pub struct Json;
impl Formatter for Json {
    fn format(&self, data: &Data, buf: &mut String) {
        buf.push('[');
        for (k, v) in data.into_iter() {
            let entry = format!(r#"{{"{}":"{}"}}"#, k, v);
            buf.push_str(&entry);
            buf.push(',');
        }
        buf.pop(); // remove extra , at the end
        buf.push(']');
    }
}
```

text包代码如下：

```
use report::formatter::{Data, Formatter};

pub struct Text;
impl Formatter for Text {
    fn format(&self, data: &Data, buf: &mut String) {
        for (k, v) in data {
            let entry = format!("{} {}\n", k, v);
            buf.push_str(&entry);
        }
    }
}

```

从上面可以看出，我们可以实现多个具体的算法，只要其实现的Formatter trait即可，这些算法可以单独成包，极大的提高的程序设计的灵活性。

学习Rust设计模式21：访问器

1、描述

访问器模式允许在同一数据上写入多种不同的算法而不修改其数据（或主要行为）。

2、示例

定义一组数据，然后为定义方位数据的Visit trait，不同的算法实现该trait。

```
// 数据类型
mod ast {
    pub struct Name {
        pub value: String,
    }

    pub enum Expr {
        IntLit(i64),
        Add(Box<Expr>, Box<Expr>),
        Sub(Box<Expr>, Box<Expr>),
    }
}

// 抽象接口
mod visit {
    use crate::ast::*;

    pub trait Visitor<T> {
        fn visit_name(&mut self, n: &Name) -> T;
        fn visit_expr(&mut self, e: &Expr) -> T;
    }
}


// 具体的访问行为
mod interpreter{
    use crate::visit::*;
    use crate::ast::*;

```

```
    pub struct Interpreter;
    impl Visitor<i64> for Interpreter {
        fn visit_name(&mut self, _n: &Name) -> i64 { 1i64 }

        fn visit_expr(&mut self, e: &Expr) -> i64 {
            match *e {
                Expr::IntLit(n) => n,
                Expr::Add(ref lhs, ref rhs) => self.visit_expr(lhs) + self.visit_expr(rhs),
                Expr::Sub(ref lhs, ref rhs) => self.visit_expr(lhs) - self.visit_expr(rhs),
            }
        }
    }
}

fn main() {
    use crate::visit::Visitor;
    let n = ast::Name{value: "alice".to_string()};
    let e = ast::Expr::IntLit(1i64);

    let mut inp = interpreter::Interpreter;
    let i = inp.visit_name(&n);
    println!("i: {:?}", i);

    let x = inp.visit_expr(&e);
    println!("x: {:?}", x);

}
```

学习Rust设计模式22：生成器

1、描述

通过生成器的调用来构造一个对象。

2、示例

生成器对象提供配置和构建的方法，能够方便的构建对象，代码如下：

```rust
#[derive(Debug)]
pub struct Teacher {
    name: String,
    age: String,
    gender: String,
    subject: String,
}

impl Teacher {
    pub fn builder() -> TeacherBuilder {
        TeacherBuilder::default()
    }
}

#[derive(Debug, Default)]
pub struct TeacherBuilder {
    name: String,
    age: String,
    gender: String,
    subject: String,
}

impl TeacherBuilder {
    pub fn new() -> Self {
        TeacherBuilder {
            name: "".to_string(),
            age: "".to_string(),
            gender: "".to_string(),
            subject: "".to_string(),
```

```
        }
    }

    pub fn name(mut self, name: String) -> Self {
        self.name = name;
        self
    }

    pub fn age(mut self, age: String) -> Self {
        self.age = age;
        self
    }

    pub fn gender(mut self, gender: String) -> Self {
        self.gender = gender;
        self
    }

    pub fn subject(mut self, subject: String) -> Self {
        self.subject = subject;
        self
    }

    pub fn build(self) -> Teacher {
        Teacher {
            name: self.name,
            age: self.age,
            gender: self.gender,
            subject: self.subject,
        }
    }
}

fn main() {
    let _math_teacher1 = TeacherBuilder::new()
        .name("Alice".to_string())
        .age("36".to_string())
        .gender("male".to_string())
        .subject("math".to_string())
        .build();
```

```
70
71      let _math_teacher2 = TeacherBuilder::new()
72          .name("Bob".to_string())
73          .gender("male".to_string())
74          .subject("math".to_string())
75          .build();
76
77      let _english_teacher = TeacherBuilder::new()
78          .gender("female".to_string())
79          .subject("english".to_string())
80          .build();
81  }
```

从上面的例子我们可以看出，通过TeacherBuilder，我们可以灵活的生成Teacher对象。

学习Rust设计模式23：Fold

1、描述

在数据集的每个元素上运行一个算法，使每个元素变成一个新的元素，从而创建一个新的集合。 类似于迭代器的map操作。

2、示例

分别有三个模块，collection是定义的数据集；fold模块定义的抽象的folder接口，提供了默认的实现；rename模块实现了folder的trait，提供了新的fold实现，代码如下：

```rust
// 数据集
mod collection {
    #[derive(Debug)]
    pub enum Data1 {
        _Number(u32),
        _Name(String),
        _Other,
    }

    #[derive(Debug)]
    pub struct Data2 {
        pub name: String,
    }
}

// 抽象folder接口
mod fold {
    use crate::collection::*;

    pub trait Folder {
        fn fold_data1(&mut self, data: Box<Data1>) -> Box<Data1> {
            data
        }

        fn fold_data2(&mut self, data: Box<Data2>) -> Box<Data2> {
            data
        }
```

```
    }
}

// 重新实现新的folder
mod rename {
    use crate::collection::*;
    use crate::fold::*;

    pub struct Renamer;
    impl Folder for Renamer {
        fn fold_data1(&mut self, data: Box<Data1>) -> Box<Data1> {
            match *data {
                Data1::_Name(_) => Box::new(Data1::_Name("foo".to_string())),
                _ => data,
            }
        }

        fn fold_data2(&mut self, _data: Box<Data2>) -> Box<Data2> {
            Box::new(Data2 {
                name: "foo".to_string(),
            })
        }
    }
}

fn main() {
    use collection::*;
    use fold::*;
    use rename::*;

    let mut renamer = Renamer;

    let d1 = Box::new(Data1::_Name("alice".to_string()));
    println!("Before fold, d1: {:?}", d1);
    let d1 = renamer.fold_data1(d1);
    println!("After fold, d1: {:?}", d1);

    let d2 = Box::new(Data2{name: "alice".to_string()});
    println!("Before fold, d2: {:?}", d2);
```

```
    let d2 = renamer.fold_data2(d2);
    println!("After fold, d2: {:?}", d2);
}
```

学习Rust设计模式24：组合结构体

1、描述

为了绕过编译器的借用检查，一种设计是将一个大的结构体设计为几个小结构体的组合。

2、示例

对于如下代码：

```
struct A {
    a: u32,
    b: u32,
    c: u32,
}

fn foo(a: &mut A) -> &u32 {
    &a.b
}
fn bar(a: &mut A) -> u32 {
    a.a + a.b
}
fn baz(a: &mut A) {
    let x = foo(a);
    // let y = bar(a);  // 此行会报错
    println!("x: {:?}", x);
}
```

因为编译器的规则，第15行打开将会报错。此时我们可以使用组合结构体来绕过编译器，使用组合结构体的代码如下：

```
struct A1 {
    e: B,
    f: C,
}
struct B {
    b: u32,
}
```

```rust
struct C {
    a: u32,
    b: u32,
}

fn foo2(b: &mut B) -> &u32 {
    &b.b
}
fn bar2(c: &mut C) -> u32 {
    c.a + c.b
}

fn baz2(a: &mut A1) {
    let x = foo2(&mut a.e);
    let y = bar2(&mut a.f); //现在可以编译过了
    println!("x: {}", x);
    println!("y: {}", y);
}
```

学习Rust设计模式25：倾向于较小的包

1、描述

倾向于选择能做好一件事的较小的包。

2、优势

- 一般来说，小的包更容易理解，模块化更好；
- crates允许在项目之间重用代码；
- Rust的编译单元是crate，分割成更多的crate可以使更多的代码被并行构建。

3、劣势

- 当一个项目同时依赖一个crate的多个冲突版本时，这可能导致“依赖地狱”；
- crates.io上的软件包没有经过组织。一个crate可能写得很差，有无用的文档，或直接是恶意的。
- 两个小crate的优化程度可能低于一个大crate，因为编译器默认不执行链接时优化。

学习Rust设计模式26：把不安全因素放在小模块中

1、描述

有unsafe的代码，放在一个尽量小的模块中。然后被嵌入到其它模块中时，其它模块就可以只包含安全的代码。

2、优势

- 限制必须被审计的不安全代码；
- 编写外部模块更容易。

3、劣势

写起来麻烦。

学习Rust设计模式27：FFI之基于对象的API

1、描述

当在Rust中设计暴露于其他语言的API时，有一些重要的设计原则与正常的Rust API设计相反：

- 所有的封装类型都应该被Rust**拥有**，由用户**管理**，并且**不透明**。
- 所有的事务性数据类型都应该由用户**拥有**，并且是**透明**的。
- 所有的库行为应该是作用于封装类型的函数。
- 所有的库行为都应该被封装成类型，且不是基于结构，而是基于**出处/生命周期**。

学习Rust设计模式28：FFI之类型合并

1、描述：

风险最低的API是“综合封装”，即与一个对象的所有可能的交互都被放入一个“封装器类型”中，保持着Rust API的整洁。

2、示例：

```
struct MyVecWrapper<'a> {
    my_vec: Vec<&'a str>,
    iter_next: usize,
}

impl<'a> MyVecWrapper<'a> {
    pub fn first_item(&mut self) -> Option<&'a str> {
        self.iter_next = 0;
        self.next_item()
    }

    pub fn next_item(&mut self) -> Option<&'a str> {
        if let Some(next) = self.my_vec.get(self.iter_next) {
            self.iter_next += 1;
            Some(next)
        } else {
            None
        }
    }
}

fn main() {
    let a = "hello".to_string();
    let b = "world".to_string();
    let v = vec![&a, &b];

    // 模拟其它语言直接使用vector中的元素
    let first = v[0];
    println!("first string: {:?}", first);
```

```rust

    println!("++++++++++++++++++++++++");
    // 下面的方式为：对象的所有可能的交互都被放入一个“封装器类型”中
    let a = "hello".to_string();
    let b = "world".to_string();
    let mut m_v = MyVecWrapper {
        my_vec: vec![&a, &b],
        iter_next: 0,
    };
    // 模拟其它语言获取该元素的引用
    let first = m_v.first_item().unwrap();
    println!("first string: {:?}", first);
}
```

学习Rust设计模式29：反面例子之克隆引用

所谓反面模式，就是反面例子，是我们在编码时应该坚决避免的使用方式。今天我们要学习的反面模式是通过Clone来满足借用检查器。

1、描述

在Rust中，对应借用只有下面两种情况：

- 只存在一个可变引用；
- 存在多个不可变引用。

当开发者通过克隆变量来绕过上面这两种情况时就出现本节要说的反面模式，即通过Clone来满足借用检查器。

2、示例

```rust
fn print_str(s: &str) {
    println!("Str is : {:?}", s);
}

fn main() {
    // 最开始的意图:
    // let mut a = "Hello world".to_string();
    // let b = &mut a;
    // println!("Before modify, a: ");
    // print_str(&a);

    // println!("After modify, a: ");
    // b.push('!');
    // print_str(&b);

    // 初学者经常会犯的错误，也是本节要说明的反面模式:
    let mut a = "Hello world".to_string();
    let b = &mut (a.clone()); //执行这行后，b和最开始的a已经不是同一个东西了
    println!("Before modify, a: ");
    print_str(&a);

    println!("After modify, a: ");
```

```
    b.push('!');
    print_str(&b);


    // 正确的做法：更换一下打印代码的顺序
    // tips：  对于初学者来说，适当的更改一下所写代码的顺序，能很大程度上减少clone的使用
    let mut a = "Hello world".to_string();
    println!("Before modify, a: ");
    print_str(&a);

    let b = &mut a;
    println!("After modify, a: ");
    b.push('!');
    print_str(&b);

}
```

学习Rust设计模式30：反面例子之拒绝警告

1、描述

在写代码时，为了不允许出现警告，添加了 #![deny(warnings)]。这种是反面例子，也是Rust设计模式中不推荐的方式。

推荐的方式在代码中明确指定我们想要拒绝的lint。

2、示例

下面为不推荐的方式：

```
// 不推荐的方式：拒绝所有警告
#![deny(warnings)]
fn main() {
    let a = "hello";
    println!("Hello, world!");
}
```

下面为推荐的方式：

```
// 推荐的方式：  明确制定拒绝的lint
#![deny(unused_variables)]
fn main() {
    let a = "hello";
    println!("Hello, world!");
}
```

学习Rust设计模式31：反面例子之滥用解引用

1、描述

反面例子：滥用Deref trait来模拟结构体间的继承，从而重用方法。

2、示例

反面例子：

```
use std::ops::Deref;

struct Foo;
impl Foo {
    fn m(&self) {
        println!("Use foo's method!");
    }
}

struct Bar {
    f: Foo,
}

// 不推荐的方式：滥用deref来模拟继承
impl Deref for Bar {
    type Target = Foo;
    fn deref(&self) -> &Foo {
        &self.f
    }
}
```

更推荐的方式应该如下：

```
...
// 推荐的方式：显式的实现m方法
impl Bar {
    fn m(&self) {
        self.f.m();
```

```
    }
}
...
```

学习Rust设计模式32：函数式编程

函数式编程是通过应用和组合函数构建程序的一种编程范式。它是一种声明式编程范式，其中函数定义表达式树，每个表达式返回一个值，而不是改变程序状态的命令式语句序列。

1、编程范式

- 命令式编程，描述的是**如何做某事；**
- 函数式编程，描述的是**什么做什么。**

示例如下：

```rust
// 命令式编程：
//      描述的是如何做某事
fn cmd_function() {
    let mut sum = 0;
    for i in 1..11 {
        sum += i;
    }
    println!("{}", sum);
}

// 函数式编程：
//      描述的是什么做什么
fn function_function() {
    println!("{}", (1..11).fold(0, |a, b| a + b));
}

fn main() {
    println!("命令式编程：");
    cmd_function();

    println!("函数式编程：");
    function_function();
}
```

2、作为类型类的泛型

Rust更像是函数式语言的一个关键就是泛型的工作方式。在Rust中，泛型参数创建了“类型类约束”，用户填写的每个不同的参数实际上都会改变类型。也就是说Vec<u32>和Vec<u8>是两种不同的类型（单态化）。例子：

```
// 学生住校信息，如果是住校生，则需要提供其寝室号信息
mod example1 {
    pub enum ResidenceInformation {
        DayStudent,
        ResidentStudent,
    }

    pub struct Student {
        pub name: String,
        pub res_info: ResidenceInformation,
        pub bedroom_no: Option<u32>,
    }

    impl Student {
        pub fn print_name(&self) {
            println!("name is: {:?}", self.name);
        }

        pub fn print_res_info(&self) {
            if !self.bedroom_no.is_none() {
                println!("bed room number is: {:?}", self.bedroom_no);
            }
        }
    }
}

// 使用泛型实现上面的功能
mod example2 {
    pub struct DayStudent;
    pub struct ResidentStudent(pub u32);

    pub struct Student<T> {
        pub name: String,
        pub res_info: T,
    }
```

```rust

    impl<T> Student<T> {
        pub fn print_name(&self) {
            println!("name is: {:?}", self.name);
        }
    }

    impl Student<ResidentStudent> {
        pub fn print_res_info(&self) {
            println!("bed room number is: {:?}", self.res_info.0);
        }
    }
}

fn main() {
    let alice = example1::Student {
        name: "alice".to_string(),
        res_info: example1::ResidenceInformation::DayStudent,
        bedroom_no: None,
    };

    alice.print_name();
    alice.print_res_info();

    let bob = example1::Student {
        name: "bob".to_string(),
        res_info: example1::ResidenceInformation::ResidentStudent,
        bedroom_no: Some(1),
    };

    bob.print_name();
    bob.print_res_info();
    println!("++++++++++++++++++++++++++++++++++++++++++++");

    let charlie = example2::Student {
        name: "charlie".to_string(),
        res_info: example2::DayStudent,
    };
    charlie.print_name();
```

```
    println!("++++++++++++++++++++++++++++++++++++++++++++");

    let dave = example2::Student {
        name: "dave".to_string(),
        res_info: example2::ResidentStudent(1),
    };
    dave.print_name();
    dave.print_res_info();
}
```