

Rust与设计模式

qqc

CONTENTS

01

速览设计模式

为什么需要设计模式

SOLID

22种设计模式

02

Rust与设计模式

Rust中常见的设计模式

代码风格tips

03

阅读代码分享(Part 2)

如何设计FFI结构体

标准库中设计分享

Rust for Linux设计分享

1分钟速览设计模式

01



Why design pattern?

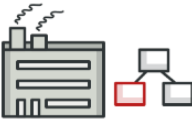

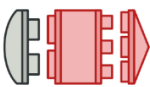

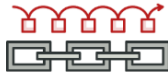


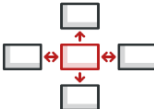

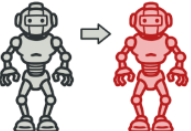

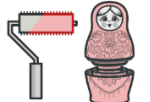
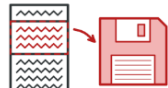


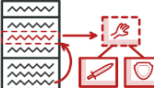




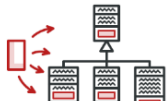

- 减小耦合
 - 避免修改一个API，影响一堆代码
 - 易于添加新功能
 - 易于测试
- 良好的编程范式有利于避免程序员的逻辑错误
 - RAII
 - State machine..
- 代码复用
 - 避免一段代码多次复制



SOLID

- **S** for “Single responsibility principle” 单一职责
 - 每个对象只有一种功能
- **O** for The Open/Closed Principle 开闭原则
 - 对拓展开发，对修改封闭
- **L** for Liskov Substitution principle 里氏替换
 - 派生类要能够替代超类
- **I** for Interface Segregation Principle 接口隔离
 - 拆分大接口为小接口
- **D** for Dependence Inversion Principle 依赖倒置
 - 依赖抽象而不是具体
- Law of **Demeter** 迪米特法则：
 - 如果两个实体不需要直接通信，就不应当发生直接调用

设计模式

 <p>工厂方法 Factory Method</p>	 <p>抽象工厂 Abstract Factory</p>	 <p>适配器 Adapter</p>	 <p>桥接 Bridge</p>	 <p>责任链 Chain of Responsibility</p>	 <p>命令 Command</p>	 <p>迭代器 Iterator</p>	 <p>中介者 Mediator</p>
 <p>生成器 Builder</p>	 <p>原型 Prototype</p>	 <p>组合 Composite</p>	 <p>装饰 Decorator</p>	 <p>备忘录 Memento</p>	 <p>观察者 Observer</p>	 <p>状态 State</p>	 <p>策略 Strategy</p>
 <p>单例 Singleton</p>		 <p>外观 Facade</p>	 <p>享元 Flyweight</p>	 <p>模板方法 Template Method</p>	 <p>访问者 Visitor</p>		
		 <p>代理 Proxy</p>					



参考



1. [设计模式目录：22种设计模式 \(refactoringguru.cn\)](http://refactoringguru.cn)
2. [深入了解23种设计模式](#)
3. [Connascence as a Software Design Metric \(practicingruby.com\)](http://practicingruby.com)

Rust中如何运用设计模式

02



Intro

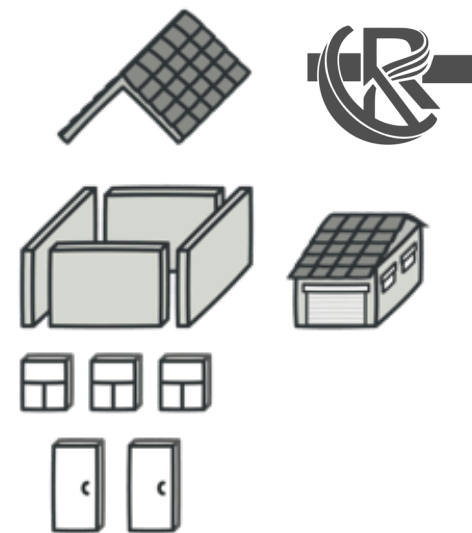
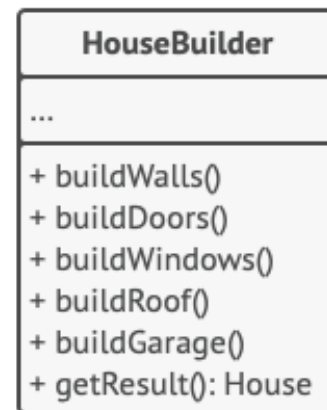
- Rust设计之初就在语言嵌入了很多优雅的模式
 - 模式匹配
 - 异常处理
 - 所有权机制
 - 零成本抽象
 - ...
- 但许多经典的设计模式Rust中仍然非常实用。
- 接下来会介绍以下内容：
 - 设计模式在Rust中的应用
 - Builder
 - State machine
 - RAI
 -
 - 代码风格



Builder

- Builder可能是Rust中应用最多的设计模式
- Recap:
 - 什么是Builder ----->
 - 为什么需要Builder
 - 需要传入很多参数
 - 传入的参数的组合很多
 - 需要分阶段构建
 - 怎么写Builder
- 两个例子:

```
use std::fs::OpenOptions;  
let file = OpenOptions::new()  
    .read(true)  
    .write(true)  
    .open("foo.txt");
```



```
use clap::{arg, Command};  
let matches = Command::new("git")  
    .subcommand_required(true)  
    .subcommand(  
        Command::new("clone")  
            .about("Clones repos")  
            .arg(arg!(<REMOTE> "The remote to clone"))  
            .arg_required_else_help(true),  
    )  
    .get_matches();
```



Builder cont.

- [derive_builder](#)提供了便捷的方式，让你可以方便地从一个结构体中创建出builder
这会生成出类似这样的代码

```
use derive_builder::Builder;

#[derive(Default, Builder, Debug)]
#[builder(setter(into))]
struct Channel {
    token: i32,
    special_info: i32,
}

fn main() {
    let ch = ChannelBuilder::default()
        .special_info(42u8)
        .token(19124)
        .build()
        .unwrap();
    println!("{:?}", ch);
}
```

```
#[derive(Clone, Default)]
struct ChannelBuilder {
    token: Option<i32>,
    special_info: Option<i32>,
}

#[allow(dead_code)]
impl ChannelBuilder {
    pub fn token<VALUE: Into<i32>>(&mut self, value: VALUE) -> &mut Self {
        let mut new = self;
        new.token = Some(value.into());
        new
    }
    pub fn special_info<VALUE: Into<i32>>(&mut self, value: VALUE) -> &mut Self {
        let mut new = self;
        new.special_info = Some(value.into());
        new
    }
}
```

- 当然，你也可以手写完成一个自定义的builder

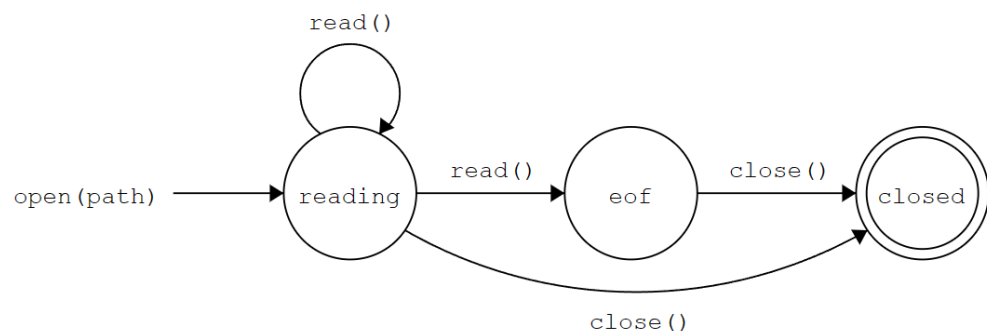


State: Typestate(session type)

- State一般与状态管理有关系
- Recap :
 - When : 状态数量多; 状态改变会影响方法..
- 传统的状态模式用一个枚举值来表示状态, 而Rust中你可以利用enum的特性来实现使用基于类型的状态管理。这个模式被称为Typestate, 也有称为session type。
- Typestate相较于传统的枚举值有很多好处:
 - 防止用户从非开始状态进入
 - 防止发生错误的转换
 - 特别地,Rust的所有权机制可以消耗掉状态, 使得用户手上的状态发生转换后就不会存在。

Example: abstract libc file operation

- 借用cs:242的一个例子
- 右边是一个文件open, read的代码
- 下面使用enum重构
 - Trait也可以，但是往往最后还要包一层enum..



```
use std::ffi::CString;
use std::str;

fn read_all() {
    unsafe {
        // Convert string into a char*
        let path = CString::new("test.txt").unwrap();
        let fd = libc::open(path.as_ptr(), 0);

        let mut buf: Vec<u8> = Vec::new();
        loop {
            // Ensure buffer has enough capacity for next 128 bytes
            buf.reserve(128);

            // Read the bytes into the vector
            let count = libc::read(fd, buf.as_mut_ptr() as *mut libc::c_void, 128);

            // Manually set the vector's length
            buf.set_len(buf.len() + count as usize);

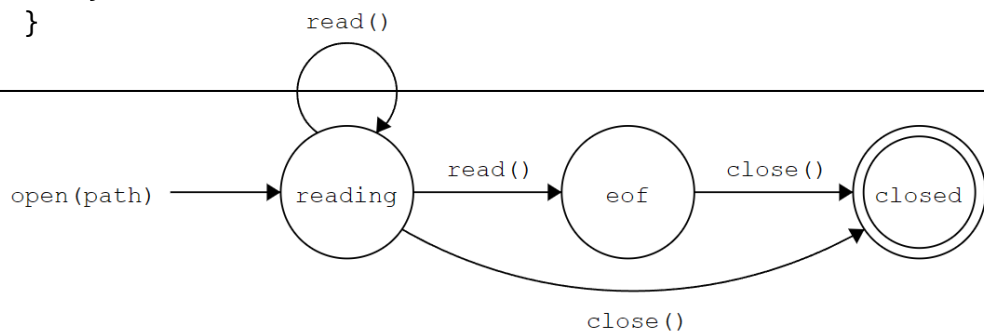
            // Check for EOF
            if count < 128 {
                break;
            }
        }

        // Print final string
        println!("{}", str::from_utf8(&buf).unwrap());
    }
}
```

Example: abstract libc file operation

```
pub struct File {
    fd: libc::c_int,
    buf: Vec<u8>,
}

impl File {
    pub fn open(path: String) -> Result<File, String> {
        unsafe {
            let fd = libc::open(CString::new(path).unwrap().as_ptr(), 0);
            if fd == -1 {
                /* assume errno_string() reads errno and converts it
                 * into a Rust string through strerror() */
                Err(std::io::Error::last_os_error().to_string())
            } else {
                Ok(File {
                    fd,
                    buf: Vec::with_capacity(1024),
                })
            }
        }
    }
}
```



```
pub enum ReadResult {
    File(File),
    FileEof(FileEof),
    Error(String),
}

impl File {
    // Read takes ownership of the file, indicated by self without & or &mut
    pub fn read(mut self, bytes: usize) -> ReadResult {
        // Ensure buf has enough space
        self.buf.reserve(bytes);
        unsafe {
            // Read the file
            let bytes_read = libc::read(self.fd, self.buf.as_mut_ptr() as *mut libc::c_void,
bytes);

            // If the read failed, then immediately return an error
            if bytes_read == -1 {
                return ReadResult::Error(std::io::Error::last_os_error().to_string());
            }

            // Increase length of the vector for the elements copied
            self.buf.set_len(self.buf.len() + bytes_read as usize);

            if (bytes_read as usize) < bytes {
                // Return EOF if we've reached the end of the file.
                // Copy all the fields of self into a new struct.
                ReadResult::FileEof(FileEof {
                    fd: self.fd,
                    buf: self.buf,
                })
            } else {
                // If we haven't reached EOF, then return back ownership
                // of self.
                ReadResult::File(self)
            }
        }
    }
}
```



Typestate: share methods

- 有时候多个状态可能共享一些变量或者结构体。
 - 例如,reading 和 eof都需要一个close方法
- 类似之前的写法就得写很多遍。如何避免重复?
 - 非typestate的写法, 都是一个类型, 就只需要写一遍
 - 折中方案: 仍然用类型表示枚举, 但是把类型作为泛型传入。

```
use std::marker::PhantomData;

pub struct Reading;
pub struct Eof;

pub struct File<S> {
    fd: libc::c_int,
    buf: Vec<u8>,
    _marker: PhantomData<S>
}

pub enum ReadResult { File(File<Reading>), FileEof(File<Eof>), Error(String) }
```

- 利用Rust phantomData的零成本抽象特性, 可以实现更灵活的状态机。



Typestate: share methods

- 这里分别为File<S>和File<Reading>实现了方法
 - Read和之前类似，省略了..
- 当然，更好的写法是把close放进drop里面。后面细说..
- 在真实项目中，这样的写法也很常见。

```
impl<S> File<S> {  
    pub fn close(mut self) -> Result<(), String> {  
        unsafe {  
            if libc::close(self.fd) == -1 {  
                Err(std::io::Error::last_os_error(  
).to_string())  
            } else {  
                Ok(())  
            }  
        }  
    }  
}
```

```
impl File<Reading> {  
    pub fn open(path: String) -> Result<File<Reading>, String> {  
        unsafe {  
            let fd = libc::open(CString::new(path).unwrap().as_ptr(), 0);  
            if fd == -1 {  
                Err(std::io::Error::last_os_error().to_string())  
            } else {  
                Ok(File {  
                    fd,  
                    buf: Vec::with_capacity(1024),  
                    // Add PhantomData value for _marker  
                    _marker: PhantomData,  
                })  
            }  
        }  
    }  
  
    pub fn read(mut self, bytes: usize) -> ReadResult {  
        todo!()  
    }  
}
```




Hyper: Typestate

- Hyper中运用了大量typestate的思想。
- 介于我也还没细看，就不做详细介绍了...

```
pub(crate) struct Server<T, S, B, E>
where
    S: HttpService<IncomingBody>,
    B: Body,
{
    exec: E,
    timer: Time,
    service: S,
    state: State<T, B>,
}
```

```
enum State<T, B>
where
    B: Body,
{
    Handshaking {
        ping_config: ping::Config,
        hs: Handshake<Compat<T>, SendBuf<B::Data>>,
    },
    Serving(Serving<T, B>),
    Closed,
}
```



Combinator

- Combinator是一个函数式编程的概念，但如今也在许多函数是一等公民的语言中。
- Combinator指的是一个函数仅通过其他函数的组合来完成工作。在Rust中，最常见的应用是在异常处理中（Option and Result）
- Option和Result提供了大量combinator函数供用户转换和处理异常，同时提供了?语法糖简化常见的match语句。



Combinator: Option

- ? 语法糖

```
fn add_last_numbers(stack: &mut Vec<i32>) -> Option<i32> {  
    Some(stack.pop()? + stack.pop())  
}
```

- Transform

Input	Output	Method
Option	Result	ok_or/ ok_or_else
Option<T>	Option<T>/None	Filter
Option<Option<T>>	Option<T>	flatten
Option<T>	Option<U>	map, map_or ,map_or_else
Option<T>	Option<(T,U)>	Zip, zip_with,unzip

- Boolean Operation

- With option : and ,or ,xor
- With function: and_then, or_else

- Iterate

- Into_iter, iter, iter_mut ----->

- Modify in-place : insert, get_or_insert ...

```
fn make_iter(do_insert: bool) -> impl Iterator<Item = i32> {  
    // Explicit returns to illustrate return types matching  
    match do_insert {  
        true => return (0..4).chain(Some(42)).chain(4..8),  
        false => return (0..4).chain(None).chain(4..8),  
    }  
}
```



Combinator: Result

- ? 语法糖
- Transform

Input	Output	Method
Result<T,E>	Result<&T,&E>	As_ref, as_mut
Result<T,E>	Result<&T::Target,&E>	AS_deref,as_deref_mut
Result<T,E>	Option<E> or Option<T>	err, ok
Result<Option<T,E>>	Option<Result<T,E>>	Transpose
Option<Option<T>>	Option<T>	flatten
Result<T,E>	Result<T,U>or Result<U,E>	map, map_or ,map_err

- Boolean Operation (类似)
 - With option : and ,or ,xor
 - With function: and_then, or_else
- Iterate
 - Into_iter, iter, iter_mut



RAII(Resource Acquisition Is Initialization)

- 资源初始化在对象的构造器中完成，最终化（资源释放）在析构器中完成。
 - 在Rust中，还可以用RAII对象作为某些资源的守护对象

Why is Rust a good fit?

You can't forget to clean up

In C

```
err_translate_failed:
err_bad_object_type:
err_bad_offset:
err_bad_parent:
err_copy_data_failed:
    binder_cleanup_deferred_txn_lists(&sgc_head, &pf_head);
    binder_free_txn_fixups(t);
    trace_binder_transaction_failed_buffer_release(t->buffer);
    binder_transaction_buffer_release(target_proc, NULL, t->buffer,
                                     buffer_offset, true);

    if (target_node)
        binder_dec_node_tmppref(target_node);
    target_node = NULL;
    t->buffer->transaction = NULL;
    binder_alloc_free_buf(&target_proc->alloc, t->buffer);
err_binder_alloc_buf_failed:
err_bad_extra_size:
    if (secctx)
        security_release_secctx(secctx, secctx_sz);
err_get_secctx_failed:
    kfree(tcomplete);
    binder_stats_deleted(BINDER_STAT_TRANSACTION_COMPLETE);
err_alloc_tcomplete_failed:
    if (trace_binder_txn_latency_free_enabled())
        binder_txn_latency_free(t);
    kfree(t);
    binder_stats_deleted(BINDER_STAT_TRANSACTION);
err_alloc_t_failed:
err_bad_todo_list:
err_bad_call_stack:
err_empty_call_stack:
err_dead_binder:
err_invalid_target_handle:
    /* it keeps going ... */
```

In Rust

```
}
```

android

RAIL : Mutex

- 右图是一个简单的Mutex
- 其中SystemMutex表示一个pthread的锁
- 你需要显式地调用lock(), unlock()
 - 上锁后离开作用域不会自动解锁
 - * T的生命周期不受限制，解锁后T仍然可以修改

```
use std::cell::UnsafeCell;

struct Mutex<T> {
    data: UnsafeCell<T>,
    system_mutex: SystemMutex,
}

impl<T> Mutex<T> {
    pub fn new(t: T) -> Self { /* .. */
    }

    pub fn lock(&self) -> &mut T {
        self.system_mutex.lock();
        unsafe { &mut *self.data.get() }
    }

    pub fn unlock(&self) {
        self.system_mutex.unlock();
    }

    #[allow(clippy::mut_from_ref)]
    pub fn get_mut(&self) -> &mut T {
        unsafe { &mut *self.data.get() }
    }
}
```

RAII : Mutex guard

- 可以设计一个Guard结构体，持有上锁的对象，并在析构的时候自动解锁。
- 用户只能通过Guard访问内部数据

```
impl<T> Mutex<T> {  
    pub fn lock_closure(&self, mut f: impl FnMut(&mut  
T)) {  
        self.system_mutex.lock();  
        f(self.get_mut());  
        self.system_mutex.unlock();  
    }  
}
```

- 此外，还可以通过传闭包来实现guard的效果。
- 如果我想用一个大锁保护一个聚合多个字段的结构体，但是每个字段又想独立访问...

```
pub struct MutexGuard<'a, T: ?Sized + 'a> {  
    lock: &'a Mutex<T>,  
}  
  
impl<'a, T> MutexGuard<'a, T> {  
    fn new(lock: &'a Mutex<T>) -> Self {  
        lock.system_mutex.lock();  
        MutexGuard { lock }  
    }  
  
    fn get(&mut self) -> &mut T {  
        &mut *self.lock.get_mut()  
    }  
}  
  
impl<'a, T: ?Sized> Drop for MutexGuard<'a, T> {  
    fn drop(&mut self) {  
        self.lock.system_mutex.unlock();  
    }  
}
```



RAIL : guard

- 如果我想用一个大锁保护一个聚合多个字段的结构体，但是每个字段又想独立访问...
 - 可以直接用多个锁的方案
 - 第三方锁库parking_lot提供了 `try_map`
 - 可以用unstable的话，标准库中也有 `try_map`

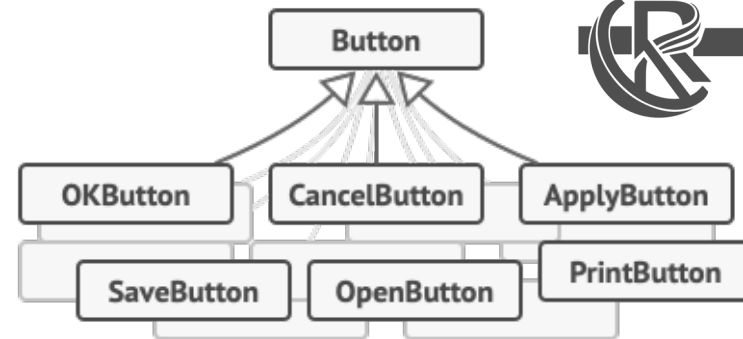
```
struct DataInner{
    field1 : i32,
    field2 : i32,
    // ...
}

struct Data(Mutex<DataInner>);

impl Data{
    fn get_field1(&self) -> &i32{
        todo!()
    }
}
```




Command



- Recap:
 - command: 将操作转化为对象
 - When&Why
 - 需要通过操作来参数化对象
 - 需要将操作放进队列里逐一执行
 - 需要回滚操作
 - 实现方式:
 - Trait 对象----->
 - 函数指针

```
type FnPtr = fn() -> String;
struct Command {
    execute: FnPtr,
    rollback: FnPtr,
}
```

- 闭包（放在Box里面）

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

pub struct CreateTable;
impl Migration for CreateTable {
    fn execute(&self) -> &str {
        "create table"
    }
    fn rollback(&self) -> &str {
        "drop table"
    }
}
```



Strategy

- Recap:
 - 策略和命令很像，但是一般策略说的是做一件事的不同方式；并且命令可以被放进队列里，策略一般不会。
- Rust中除了可以用trait实现，也可以用闭包实现

```
struct Adder;  
impl Adder {  
    pub fn add<F>(x: u8, y: u8, f: F) -> u8  
    where  
        F: Fn(u8, u8) -> u8,  
    {  
        f(x, y)  
    }  
}
```

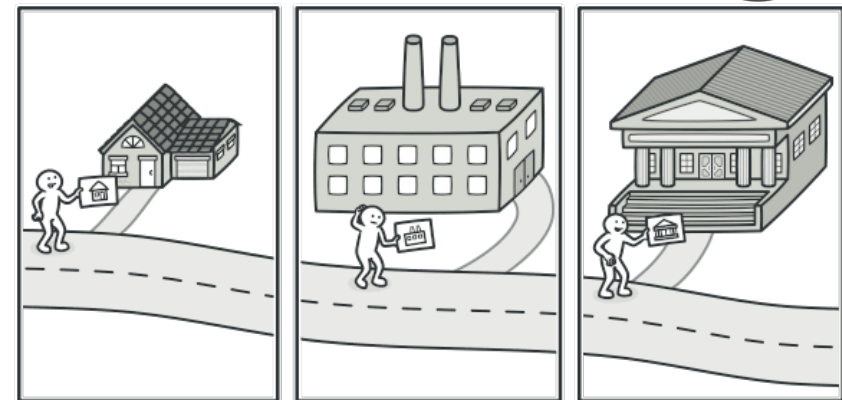
- Option中的Map就是这一思想

```
fn main() {  
    let arith_adder = |x, y| x + y;  
    let bool_adder = |x, y| {  
        if x == 1 || y == 1 {  
            1  
        } else {  
            0  
        }  
    };  
    let custom_adder = |x, y| 2 * x + y;  
  
    assert_eq!(9, Adder::add(4, 5, arith_adder));  
    assert_eq!(0, Adder::add(0, 0, bool_adder));  
    assert_eq!(5, Adder::add(1, 3, custom_adder));  
}
```

Visitor:



- Recap:
 - 将对多种对象数据的行为单独放在一个类中
 - 可以视为command的加强版
 - When&Why
 - 访问异构数据（例如树）
 - 分离解析数据(deserialize)和操作(visitor)本身
 - 双分配: Rust没有重载，虽然也可以双分配，但是要显式标出
- 最常用Visitor的两个场景是在遍历AST树和反序列化的时候。
 - [Visitor in rustc_ast::visit - Rust \(rust-lang.org\)](#)
 - [serde::de::Visitor - Rust \(rust-lang.github.io\)](#)
- 下面以serde的反序列化器为例，做了一点简化(参考blog)。





Visitor : Deserializer

- 场景1：写一个反序列化器，从文本形式转为Point

```
struct Point {  
    x: i32,  
    y: i32,  
}
```

```
impl Deserialize<&str> for Point {  
    type Error = PointDeserializationError;  
  
    fn deserialize(input: &str) -> Result<Self, Self::Error> {  
        let split = input.split(",").collect::<Vec<_>>();  
        Ok(Point {  
            x: split[0].parse().map_err(|_| PointDeserializationError)?,  
            y: split[1].parse().map_err(|_| PointDeserializationError)?,  
        })  
    }  
}
```



Visitor : Deserializer

- 场景2：写一个反序列化器，从Json形式转为Point
 - 可以用newtype封装一下json

```
struct Point {  
    x: i32,  
    y: i32,  
}
```

```
struct Json(String);  
  
impl Deserialize<Json> for Point {  
    type Error = PointDeserializationError;  
  
    fn deserialize(input: Json) -> Result<Self, Self::Error> {  
        todo!()  
    }  
}
```



Visitor : Deserializer

- 场景3：写一个反序列化器，从Map形式转为Point
 - Json似乎可以先转成Map
 - 然后再从Map变为Point
 - 但是我们的代码结构很难复用..
 - 简单地写一个函数，然后Json和Map同时调用，治标不治本

```
struct Point {  
    x: i32,  
    y: i32,  
}
```

```
impl Deserialize<HashMap<K,V>> for Point {  
    type Error = PointDeserializationError;  
  
    fn deserialize(input: HashMap<K,V>>) -> Result<Self, Self::Error> {  
        todo!()  
    }  
}
```



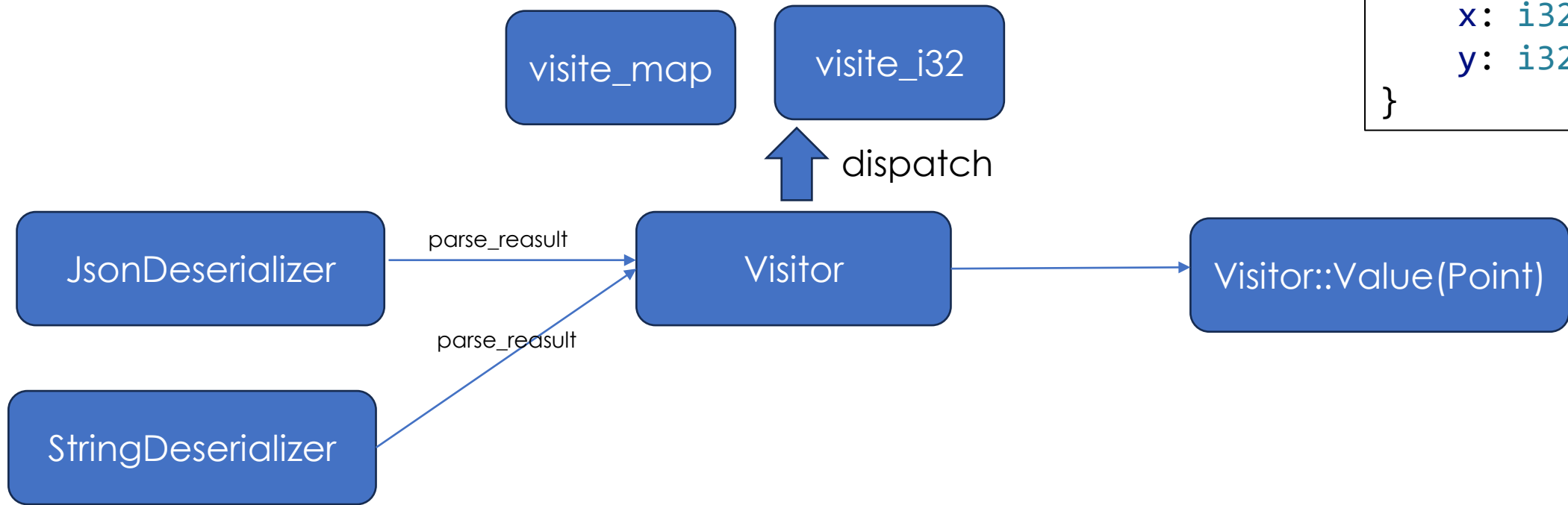
Visitor : Deserializer

```
struct JsonSerializer {  
    //...  
}  
  
impl<> JsonSerializer {  
    fn parse_map(&self) -> Result<Box<dyn MapAccess>, JsonError> {  
        todo!()  
    }  
}  
  
struct JsonError;  
  
impl<'de> Deserializer<'de> for JsonSerializer {  
    type Error = JsonError;  
    fn deserialize_struct<V>(self, visitor: V) -> Result<V::Value, Self::Error>  
    where  
        V: Visitor<'de>  
    {  
        //imagine there is a method to get a map access object on the JsonSerializer  
        let map = self.parse_map().map_err(|_| JsonError)?;  
  
        visitor.visit_map(map).map_err(|_| JsonError)  
    }  
}
```

```
trait Visitor<'de> {  
    type Value;  
    fn visit_map<M>(self, map: M) -> Result<Self::Value,  
M:: Error>  
    where  
        M: MapAccess<'de>;  
}  
  
trait Deserializer<'de> {  
    type Error;  
    fn deserialize_struct<V>(self, visitor: V) ->  
Result<V::Value, Self::Error>  
    where  
        V: Visitor<'de>;  
}
```



Deserializer的解耦



```
struct Point {  
    x: i32,  
    y: i32,  
}
```

解析

加工/校验



Observer

- Recap:
 - Observer即订阅者模式，通过创建一个中心化的注册结构体，解耦订阅者和发布者
- 基于Rust安全的类型系统，我们可以设计一个更安全的订阅器。

Observer

- 这是一个简单的订阅器，对应的类似C语言中传递一个void*的情形。
- 基于字符串分发
- 可能有一些问题：
 - 用户打错字(typo)
 - Data处的约束太弱，可能会传错数据；
 - 或者Listener入参会被错误转化。

```
type EventListener = Box<dyn Fn(&dyn Any)>;

#[derive(Default)]
struct EventRegistry {
    listeners: HashMap<String, Vec<EventListener>>,
}

impl EventRegistry {
    fn add_event_listener(&mut self, event: String, f: EventListener) {
        self.listeners.entry(event).or_insert_with(Vec::new).push(f);
    }

    fn trigger(&self, event: String, data: &dyn Any) {
        let listeners = self.listeners.get(&event).unwrap();
        for listener in listeners.iter() {
            listener(data);
        }
    }
}
```



Observer



- 可以利用强类型系统的优势，接受类型作为订阅的键
- 这样Listener也可以接受一个确定类型的参数
- 如何实现TypeMap这个结构体可以参考页脚的链接

```
struct EventDispatcher(TypeMap);
type ListenerVec<E> = Vec<Box<dyn EventListener<E>>>>;

impl EventDispatcher {
    fn add_event_listener<E>(&mut self, f: impl EventListener<E>) {
        if !self.0.has::
```



其他

- **Decorator** : [Are derive macros in Rust similar to decorators in Python? - help - The Rust Programming Language Forum \(rust-lang.org\)](#)
- **Interpreter** : 宏
- **Adapter** : 常见应用为不同系统提供抽象层
 - Lock: [rust/library/std/src/sys/sync/mutex/mod.rs at master · rust-lang/rust \(github.com\)](#)
 - Mlsdisk : [mlsdisk/core/src/os at main · asterinas/mlsdisk \(github.com\)](#)
- **Singleton** : lazy_static, static Mutex
- **Cow**
- **Iterator**: Iter trait
- **Newtype pattern**: 零成本封装与抽象
 - **Refinement** : 通过类型约束值
 - Liquid Types for Rust: [flux-rs/flux: Refinement Types for Rust \(github.com\)](#)
 - **Witness** : 通过newtype约束参数
 - [Witnesses - Type-Driven API Design in Rust \(willcrichton.net\)](#)
 - [Requests - Rocket Web Framework](#)
 - **bitflag**
- 类型擦除



代码风格

- 以API设计者的思路去写代码：
 - Rust官方为我们提供一份Rust API Guidelines:
 - [About - Rust API Guidelines \(rust-lang.github.io\)](https://rust-lang.github.io/rust-style/)
- Rust linting和Clippy是你的好伙伴
 - [Introduction - Clippy Documentation \(rust-lang.org\)](https://rust-lang.org/clippy/)
- 多阅读网上的Blog
 - 类似于effective c++，Rust有大量这样的idioms. 有些易于被模式化，可以被clippy识别，但有些则需要开发者自己注意

More Tips: Naming

- [Naming](#)

- [标识符的命名规范](#)
- get不用加前缀,set需要。但get需要标明mut
- Conversion取名的规范

Prefix	Cost	Ownership
as_	Free	borrowed -> borrowed
to_	Expensive	borrowed -> borrowed borrowed -> owned (non-Copy types) owned -> owned (Copy types)
into_	Variable	owned -> owned (non-Copy types)

- `str::as_bytes()` : `&str -> &[u8]`, cost is free
- `str::to_lowercase` : `&str -> String`
- `String::into_bytes`: `String` to `Vec<u8>`
- `BufWriter::into_inner` : 会刷新缓存



More tips:尽可能泛化参数

- 函数参数倾向于使用&T而不是&Arc<T>
- 接收字符串时，倾向于接收一个impl FromStr的参数
- 如果一个类型可以被转化为std中的某个类型T，可以为它实现Deref<Target=T>
- 传入迭代器可以考虑使用T: Iterator<Item=U>
- 如果希望能接受多种类型，可以用泛型或者Trait “概括” 需要接受的参数。

```
pub fn find<'a, P>(&'a self, pat: P) -> Option<usize>  
where  
    P: Pattern<'a>;
```

```
pub fn open<P: AsRef<Path>>(path: P) -> Result<File>;
```



参考

1. [Rust Design Patterns\(中译\)](#)
2. [Nine Rules for Elegant Rust Library APIs | by Carl M. Kadie
| Towards Data Science](#)
3. [Naming - Rust API Guidelines \(rust-lang.github.io\)](#)
4. [Type-Driven API Design in Rust](#)
5. [Elegant Library APIs in Rust](#)

代码阅读分享

03



如何设计一个FFI struct



Std里是如何运用设计模式



参考



1. Inside-rust-std-library : <https://github.com/Warrenren/inside-rust-std-library/issues>



Rust for Linux中的设计

