23｜类型系统：如何在实战中使用泛型编程?

time.geekbang.org/column/article/427082

陈天 2021-10-18

00:00

1.0x
讲述：陈天大小：15.93M时长：17:23

你好，我是陈天。

从这一讲开始，我们就到进阶篇了。在进阶篇中，我们会先进一步夯实对类型系统的理解，然后再展开网络处理、Unsafe Rust、FFI 等主题。

为什么要把类型系统作为进阶篇的基石？之前讲解 rgrep 的代码时你可以看到，当要构建可读性更强、更加灵活、更加可测试的系统时，我们都要或多或少使用 trait 和泛型编程。

所以可以说在 Rust 开发中，泛型编程是我们必须掌握的一项技能。在你构建每一个数据结构或者函数时，最好都问问自己：我是否有必要在此刻就把类型定死？是不是可以把这个决策延迟到尽可能靠后的时刻，这样可以为未来留有余地？

在《架构整洁之道》里 Uncle Bob 说：架构师的工作不是作出决策，而是尽可能久地推迟决策，在现在不作出重大决策的情况下构建程序，以便以后有足够信息时再作出决策。所以，如果我们能通过泛型来推迟决策，系统的架构就可以足够灵活，可以更好地面对未来的变更。

今天，我们就来讲讲如何在实战中使用泛型编程，来延迟决策。如果你对 Rust 的泛型编程掌握地还不够牢靠，建议再温习一下第 12 和 13 讲，也可以阅读 The Rust Programming Language 第 10 章作为辅助。

泛型数据结构的逐步约束

在进入正题之前，我们以标准库的 BufReader 结构为例，先简单回顾一下，在定义数据结构和实现数据结构时，如果使用了泛型参数，到底有什么样的好处。
看这个定义的小例子：

```
pub struct BufReader<R> {

inner: R,

buf: Box<[u8]>,

pos: usize,

cap: usize,

}
```

BufReader 对要读取的 R 做了一个泛型的抽象。也就是说，R 此刻是个 File，还是一个 Cursor，或者直接是 Vec<u8>，都不重要。在定义 struct 的时候，我们并未对 R 做进一步的限制，这是最常用的使用泛型的方式。

到了实现阶段，根据不同的需求，我们可以为 R 做不同的限制。这个限制需要细致到什么程度呢？只需要添加刚好满足实现需要的限制即可。

比如在提供 capacity()、buffer() 这些不需要使用 R 的任何特殊能力的时候，可以不做任何限制：

```
impl<R> BufReader<R> {

pub fn capacity(&self) -> usize { ... }

pub fn buffer(&self) -> &[u8] { ... }

}
```

但在实现 new() 的时候，因为使用了 Read trait 里的方法，所以这时需要明确传进来的 R 满足 Read 约束：

```
impl<R: Read> BufReader<R> {

pub fn new(inner: R) -> BufReader<R> { ... }

pub fn with_capacity(capacity: usize, inner: R) -> BufReader<R> { ... }

}
```

同样，在实现 Debug 时，也可以要求 R 满足 Debug trait 的约束：

```
impl<R> fmt::Debug for BufReader<R>

where

R: fmt::Debug

{

fn fmt(&self, fmt: &mut fmt::Formatter<'_>) -> fmt::Result { ... }

}
```

如果你多花一些时间，把 bufreader.rs 对接口的所有实现都过一遍，还会发现 BufReader 在实现过程中使用了 Seek trait。
整体而言，impl BufReader 的代码根据不同的约束，分成了不同的代码块。这是一种非常典型的实现泛型代码的方式，我们可以学习起来，在自己的代码中也应用这种方法。

通过使用泛型参数，BufReader 把决策交给使用者。我们在上一讲期中考试的 rgrep 实现中也看到了，在测试和 rgrep 的实现代码中，是如何为 BufReader 提供不同的类型来满足不同的使用场景的。

泛型参数的三种使用场景

泛型参数的使用和逐步约束就简单复习到这里，相信你已经掌握得比较好了，我们开始今天的重头戏，来学习实战中如何使用泛型编程。

先看泛型参数，它有三种常见的使用场景：

使用泛型参数延迟数据结构的绑定；

使用泛型参数和 PhantomData，声明数据结构中不直接使用，但在实现过程中需要用到的类型；

使用泛型参数让同一个数据结构对同一个 trait 可以拥有不同的实现。

用泛型参数做延迟绑定

先来看我们已经比较熟悉的，用泛型参数做延迟绑定。在 KV server 的上篇中，我构建了一个 Service 数据结构：

```
/// Service 数据结构

pub struct Service<Store = MemTable> {

inner: Arc<ServiceInner<Store>>,

}
```

它使用了一个泛型参数 Store，并且这个泛型参数有一个缺省值 MemTable。指定了泛型参数缺省值的好处是，在使用时，可以不必提供泛型参数，直接使用缺省值。这个泛型参数在随后的实现中可以被逐渐约束：

```
impl<Store> Service<Store> {

pub fn new(store: Store) -> Self { ... }

}

impl<Store: Storage> Service<Store> {

pub fn execute(&self, cmd: CommandRequest) -> CommandResponse { ... }

}
```

同样的，在泛型函数中，可以使用 impl Storage 或者 <Store: Storage> 的方式去约束：

```
pub fn dispatch(cmd: CommandRequest, store: &impl Storage) -> CommandResponse { ... }

// 等价于

pub fn dispatch<Store: Storage>(cmd: CommandRequest, store: &Store) -> CommandResponse { ... }
```

这种用法，想必你现在已经非常熟悉了，可以在开发中使用泛型参数来对类型进行延迟绑定。

使用泛型参数和幽灵数据（PhantomData）提供额外类型

在熟悉了泛型参数的基本用法后，我来考考你：现在要设计一个 User 和 Product 数据结构，它们都有一个 u64 类型的 id。然而我希望每个数据结构的 id 只能和同种类型的 id 比较，也就是说如果 user.id 和 product.id 比较，编译器就能直接报错，拒绝这种行为。该怎么做呢?

你可以停下来先想一想。

很可能会立刻想到这个办法。先用一个自定义的数据结构 Identifier<T> 来表示 id：

```
pub struct Identifier<T> {

inner: u64,

}
```

然后，在 User 和 Product 中，各自用 Identifier<Self> 来让 Identifier 和自己的类型绑定，达到让不同类型的 id 无法比较的目的。有了这个构想，你可以很快写出这样的代码（代码）：

```
#[derive(Debug, Default, PartialEq, Eq)]
pub struct Identifier<T> {
inner: u64,
}
#[derive(Debug, Default, PartialEq, Eq)]
pub struct User {
id: Identifier<Self>,
}
#[derive(Debug, Default, PartialEq, Eq)]
pub struct Product {
id: Identifier<Self>,
}
#[cfg(test)]
mod tests {
use super::*;
#[test]
fn id_should_not_be_the_same() {
let user = User::default();
let product = Product::default();
// 两个 id 不能比较，因为他们属于不同的类型
// assert_ne!(user.id, product.id);
assert_eq!(user.id.inner, product.id.inner);
}
}
```

然而它无法编译通过。为什么呢?

因为 Identifier<T> 在定义时，并没有使用泛型参数 T，编译器认为 T 是多余的，所以只能把 T 删除掉才能编译通过。但是，删除掉 T，User 和 Product 的 id 就可以比较了，我们就无法实现想要的功能了，怎么办？唉，刚刚还踌躇满志觉得可以用泛型来指点江山，现在面对这么个小问题却万念俱灭？

别急。如果你使用过任何其他支持泛型的语言，无论是 Java、Swift 还是 TypeScript，可能都接触过 Phantom Type（幽灵类型）的概念。像刚才的写法，Swift / TypeScript 会让其通过，因为它们的编译器会自动把多余的泛型参数当成 Phantom type 来用，比如下面 TypeScript 的例子，可以编译：

```
// NotUsed is allowed

class MyNumber<T, NotUsed> {

inner: T;

add: (x: T, y: T) => T;

}
```

但 Rust 对此有洁癖。Rust 并不希望在定义类型时，出现目前还没使用，但未来会被使用的泛型参数，所以 Rust 编译器对此无情拒绝，把门关得严严实实。

不过，别担心，作为过来人，Rust 知道 Phantom Type 的必要性，所以开了一扇叫 PhantomData 的窗户：让我们可以用 PhantomData 来持有 Phantom Type。
PhantomData 中文一般翻译成幽灵数据，这名字透着一股让人不敢亲近的邪魅，但它被广泛用在处理，数据结构定义过程中不需要，但是在实现过程中需要的泛型参数。
我们来试一下：

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

#[derive(Debug, Default, PartialEq, Eq)]

pub struct Identifier<T> {

inner: u64,

_tag: PhantomData<T>,

}

#[derive(Debug, Default, PartialEq, Eq)]

pub struct User {

id: Identifier<Self>,

}

#[derive(Debug, Default, PartialEq, Eq)]
```

```
pub struct Product {
id: Identifier<Self>,
}
#[cfg(test)]
mod tests {
use super::*;
#[test]
fn id_should_not_be_the_same() {
let user = User::default();
let product = Product::default();
// 两个 id 不能比较，因为他们属于不同的类型
// assert_ne!(user.id, product.id);
assert_eq!(user.id.inner, product.id.inner);
}
}
```

Bingo！编译通过！在使用了 PhantomData 后，编译器允许泛型参数 T 的存在。

现在我们确认了：在定义数据结构时，对于额外的、暂时不需要的泛型参数，用 PhantomData 来“拥有”它们，这样可以规避编译器的报错。PhantomData 正如其名，它实际上长度为零，是个 ZST（Zero-Sized Type），就像不存在一样，唯一作用就是类型的标记。

再来写一个例子，加深对 PhantomData 的理解（代码）：

```
use std::{
marker::PhantomData,
sync::atomic::{AtomicU64, Ordering},
};
static NEXT_ID: AtomicU64 = AtomicU64::new(1);
pub struct Customer<T> {
```

```
id: u64,
name: String,
_type: PhantomData<T>,
}
pub trait Free {
fn feature1(&self);
fn feature2(&self);
}
pub trait Personal: Free {
fn advance_feature(&self);
}
impl<T> Free for Customer<T> {
fn feature1(&self) {
println!("feature 1 for {}", self.name);
}
fn feature2(&self) {
println!("feature 2 for {}", self.name);
}
}
impl Personal for Customer<PersonalPlan> {
fn advance_feature(&self) {
println!(
"Dear {}(as our valuable customer {}), enjoy this advanced feature!",
self.name, self.id
);
}
```

```
}
pub struct FreePlan;
pub struct PersonalPlan(f32);
impl<T> Customer<T> {
pub fn new(name: String) -> Self {
Self {
id: NEXT_ID.fetch_add(1, Ordering::Relaxed),
name,
_type: PhantomData::default(),
}
}
}
impl From<Customer<FreePlan>> for Customer<PersonalPlan> {
fn from(c: Customer<FreePlan>) -> Self {
Self::new(c.name)
}
}
/// 订阅成为付费用户
pub fn subscribe(customer: Customer<FreePlan>, payment: f32) ->
Customer<PersonalPlan> {
let _plan = PersonalPlan(payment);
// 存储 plan 到 DB
// ...
customer.into()
}
#[cfg(test)]
```

```
mod tests {

use super::*;

#[test]

fn test_customer() {

// 一开始是个免费用户

let customer = Customer::<FreePlan>::new("Tyr".into());

// 使用免费 feature

customer.feature1();

customer.feature2();

// 用着用着觉得产品不错愿意付费

let customer = subscribe(customer, 6.99);

customer.feature1();

customer.feature1();

// 付费用户解锁了新技能

customer.advance_feature();

}

}
```

在这个例子里，Customer 有个额外的类型 T。

通过类型 T，我们可以将用户分成不同的等级，比如免费用户是 Customer<FreePlan>、付费用户是 Customer<PersonalPlan>，免费用户可以转化成付费用户，解锁更多权益。使用 PhantomData 处理这样的状态，可以在编译期做状态的检测，避免运行期检测的负担和潜在的错误。

使用泛型参数来提供多个实现

用泛型参数做延迟绑定、结合 PhantomData 来提供额外类型，是我们经常能看到的泛型参数的用法。

有时候，对于同一个 trait，我们想要有不同的实现，该怎么办？比如一个方程，它可以是线性方程，也可以是二次方程，我们希望为不同的类型实现不同 Iterator。可以这样做（代码）：

```
use std::marker::PhantomData;
#[derive(Debug, Default)]
pub struct Equation<IterMethod> {
current: u32,
_method: PhantomData<IterMethod>,
}
// 线性增长
#[derive(Debug, Default)]
pub struct Linear;
// 二次增长
#[derive(Debug, Default)]
pub struct Quadratic;
impl Iterator for Equation<Linear> {
type Item = u32;
fn next(&mut self) -> Option<Self::Item> {
self.current += 1;
if self.current >= u16::MAX as u32 {
return None;
}
Some(self.current)
}
}
impl Iterator for Equation<Quadratic> {
type Item = u32;
fn next(&mut self) -> Option<Self::Item> {
self.current += 1;
```

```
if self.current >= u32::MAX {
return None;
}
Some(self.current * self.current)
}
}
#[cfg(test)]
mod tests {
use super::*;
#[test]
fn test_linear() {
let mut equation = Equation::<Linear>::default();
assert_eq!(Some(1), equation.next());
assert_eq!(Some(2), equation.next());
assert_eq!(Some(3), equation.next());
}
#[test]
fn test_quadratic() {
let mut equation = Equation::<Quadratic>::default();
assert_eq!(Some(1), equation.next());
assert_eq!(Some(4), equation.next());
assert_eq!(Some(9), equation.next());
}
}
```

这个代码很好理解，但你可能会有疑问：这样做有什么好处么？为什么不构建两个数据结构 LinearEquation 和 QuadraticEquation，分别实现 Iterator 呢?

的确，对于这个例子，使用泛型的意义并不大，因为 Equation 自身没有很多共享的代码。但如果 Equation，只除了实现 Iterator 的逻辑不一样，其它大量的代码都是相同的，并且未来除了一次方程和二次方程，还会支持三次、四次……，那么，用泛型数据结构来统一相同的逻辑，用泛型参数的具体类型来处理变化的逻辑，就非常有必要了。

来看一个真实存在的例子AsyncProstReader，它来自之前我们在 KV server 里用过的 async-prost 库。async-prost 库，可以把 TCP 或者其他协议中的 stream 里传输的数据，分成一个个 frame 处理。其中的 AsyncProstReader 为 AsyncDestination 和 AsyncFrameDestination 提供了不同的实现，你可以不用关心它具体做了些什么，只要学习它的接口的设计：

/// A marker that indicates that the wrapping type is compatible with `AsyncProstReader` with Prost support.

#[derive(Debug)]

pub struct AsyncDestination;

/// a marker that indicates that the wrapper type is compatible with `AsyncProstReader` with Framed support.

#[derive(Debug)]

pub struct AsyncFrameDestination;

/// A wrapper around an async reader that produces an asynchronous stream of prost-decoded values

#[derive(Debug)]

pub struct AsyncProstReader<R, T, D> {

reader: R,

pub(crate) buffer: BytesMut,

into: PhantomData<T>,

dest: PhantomData<D>,

}

这个数据结构虽然使用了三个泛型参数，其实数据结构中真正用到的只有一个 R，它可以是一个实现了 AsyncRead 的数据结构（稍后会看到）。另外两个泛型参数 T 和 D，在数据结构定义的时候其实并不需要，只是在数据结构的实现过程中，才需要用到它们的约束。其中，

T 是从 R 中读取出的数据反序列化出来的类型，在实现时用 prost::Message 约束。

D 是一个类型占位符，它会根据需要被具体化为 AsyncDestination 或者 AsyncFrameDestination。

类型参数 D 如何使用，我们可以先想像一下。实现 AsyncProstReader 的时候，我们希望在使用 AsyncDestination 时，提供一种实现，而在使用 AsyncFrameDestination 时，提供另一种实现。也就是说，这里的类型参数 D，在 impl 的时候，会被具体化成某个类型。

拿着这个想法，来看 AsyncProstReader 在实现 Stream 时，D 是如何具体化的。这里你不用关心 Stream 具体是什么以及如何实现。实现的代码不重要，重要的是接口（代码）：

```
impl<R, T> Stream for AsyncProstReader<R, T, AsyncDestination>

where

T: Message + Default,

R: AsyncRead + Unpin,

{

type Item = Result<T, io::Error>;

fn poll_next(mut self: Pin<&mut Self>, cx: &mut Context<'_>) ->
Poll<Option<Self::Item>> {

...

}

}
```

再看对另外一个对 D 的具体实现：

```
impl<R, T> Stream for AsyncProstReader<R, T, AsyncFrameDestination>

where

R: AsyncRead + Unpin,

T: Framed + Default,

{

type Item = Result<T, io::Error>;

fn poll_next(mut self: Pin<&mut Self>, cx: &mut Context<'_>) ->
Poll<Option<Self::Item>> {

...

}
```

```
}
```

在这个例子里，除了 Stream 的实现不同外，AsyncProstReader 的其它实现都是共享的。所以我们有必要为其增加一个泛型参数 D，使其可以根据不同的 D 的类型，来提供不同的 Stream 实现。

AsyncProstReader 综合使用了泛型的三种用法，感兴趣的话你可以看源代码。如果你无法一下子领悟它的代码，也不必担心。很多时候，这样的高级技巧在阅读代码时用途会更大一些，起码你能搞明白别人的代码为什么这么写。至于自己写的时候是否要这么用，你可以根据自己掌握的程度来决定。

毕竟，我们写代码的首要目标是正确地实现所需要的功能，在正确性的前提下，优雅简洁的表达才有意义。

泛型函数的高级技巧

如果你掌握了泛型数据结构的基本使用方法，那么泛型函数并不复杂，因为在使用泛型参数和对泛型参数进行约束方面是一致的。

之前的课程中，我们已经在函数参数中多次使用泛型参数了，想必你已经有足够的掌握。关于泛型函数，我们讲两点，一是返回值如果想返回泛型参数，该怎么处理？二是对于复杂的泛型参数，该如何声明?

返回值携带泛型参数怎么办?

在 KV server 中，构建 Storage trait 的 get_iter 接口时，我们已经见到了这样的用法：

```
pub trait Storage {

...

/// 遍历 HashTable，返回 kv pair 的 Iterator

fn get_iter(&self, table: &str) ->

Result<Box<dyn Iterator<Item = Kvpair>>, KvError>;

}
```

对于 get_iter() 方法，并不关心返回值是一个什么样的 Iterator，只要它能够允许我们不断调用 next() 方法，获得一个 Kvpair 的结构，就可以了。在实现里，使用了 trait object。

你也许会有疑惑，为什么不能直接使用 impl Iterator 呢?

```
// 目前 trait 还不支持

fn get_iter(&self, table: &str) -> Result<impl Iterator<Item = Kvpair>, KvError>;
```

原因是 Rust 目前还不支持在 trait 里使用 impl trait 做返回值：

```
pub trait ImplTrait {

// 允许

fn impl_in_args(s: impl Into<String>) -> String {

s.into()

}

// 不允许

fn impl_as_return(s: String) -> impl Into<String> {

s

}

}
```

那么使用泛型参数做返回值呢？可以，但是在实现的时候会很麻烦，你很难在函数中正确构造一个返回泛型参数的语句：

```
// 可以正确编译

pub fn generics_as_return_working(i: u32) -> impl Iterator<Item = u32> {

std::iter::once(i)

}

// 期待泛型类型，却返回一个具体类型

pub fn generics_as_return_not_working<T: Iterator<Item = u32>>(i: u32) -> T {

std::iter::once(i)

}
```

那怎么办？很简单，我们可以返回 trait object，它消除了类型的差异，把所有不同的实现 Iterator 的类型都统一到一个相同的 trait object 下：

```
// 返回 trait object

pub fn trait_object_as_return_working(i: u32) -> Box<dyn Iterator<Item = u32>> {

Box::new(std::iter::once(i))

}
```

明白了这一点，回到刚才 KV server 的 Storage trait：

```
pub trait Storage {

...

/// 遍历 HashTable，返回 kv pair 的 Iterator

fn get_iter(&self, table: &str) ->

Result<Box<dyn Iterator<Item = Kvpair>>, KvError>;

}
```

现在你是不是更好地理解了，在这个 trait 里，为何我们需要使用 Box<dyn Iterator<Item = Kvpair>> ？

不过使用 trait object 是有额外的代价的，首先这里有一次额外的堆分配，其次动态分派会带来一定的性能损失。

复杂的泛型参数该如何处理?

在泛型函数中，有时候泛型参数可以非常复杂。比如泛型参数是一个闭包，闭包返回一个 Iterator，Iterator 中的 Item 又有某个约束。看下面的示例代码：

```
pub fn comsume_iterator<F, Iter, T>(mut f: F)

where

F: FnMut(i32) -> Iter, // F 是一个闭包，接受 i32，返回 Iter 类型

Iter: Iterator<Item = T>, // Iter 是一个 Iterator，Item 是 T 类型

T: std::fmt::Debug, // T 实现了 Debug trait

{

// 根据 F 的类型，f(10) 返回 iterator，所以可以用 for 循环

for item in f(10) {

println!("{:?}", item); // item 实现了 Debug trait，所以可以用 {:?} 打印

}

}
```

这个代码的泛型参数虽然非常复杂，不过一步步分解，其实并不难理解其实质：

参数 F 是一个闭包，接受 i32，返回 Iter 类型；

参数 Iter 是一个 Iterator，Item 是 T 类型；

参数 T 是一个实现了 Debug trait 的类型。

这么分解下来，我们就可以看到，为何这段代码能够编译通过，同时也可以写出合适的测试示例，来测试它：

```
#[cfg(test)]

mod tests {

use super::*;

#[test]

fn test_consume_iterator() {

// 不会 panic 或者出错

comsume_iterator(|i| (0..i).into_iter())

}

}
```

小结

泛型编程在 Rust 开发中占据着举足轻重的地位，几乎你写的每一段代码都或多或少会使用到泛型有关的结构，比如标准库的 Vec<T>、HashMap<K, V> 等。当我们自己构建数据结构和函数时要思考，是否使用泛型参数，让代码更加灵活、可扩展性更强。

当然，泛型编程也是一把双刃剑。任何时候，当我们引入抽象，即便能做到零成本抽象，要记得抽象本身也是一种成本。

当我们把代码抽象成函数、把数据结构抽象成泛型结构，即便运行时几乎并无添加额外成本，它还是会带来设计时的成本，如果抽象得不好，还会带来更大的维护上的成本。做系统设计，我们考虑 ROI（Return On Investment）时，要把 TCO（Total Cost of Ownership）也考虑进去。这也是为什么过度设计的系统和不做设计的系统，它们长期的 TCO 都非常糟糕。

建议你在自己的代码中使用复杂的泛型结构前，最好先做一些准备。

首先，自然是了解使用泛型的场景，以及主要的模式，就像本文介绍的那样；之后，可以多读别人的代码，多看优秀的系统，都是如何使用泛型来解决实际问题的。同时，不要着急把复杂的泛型引入到你自己的系统中，可以先多写一些小的、测试性质的代码，就像文中的那些示例代码一样，从小处着手，去更深入地理解泛型；

有了这些准备打底，最后在你的大型项目中，需要的时候引入自己的泛型数据结构或者函数，去解决实际问题。

思考题

如果你理解了今天讲的泛型的用法，那么阅读 futures 库时，遇到类似的复杂泛型声明，比如说 StreamExt trait 的 for_each_concurrent，你能搞明白它的参数 f 代表什么吗？你该怎么使用这个方法呢？

```
fn for_each_concurrent<Fut, F>(

self,

limit: impl Into<Option<usize>>,

f: F,

) -> ForEachConcurrent<Self, Fut, F>

where

F: FnMut(Self::Item) -> Fut,

Fut: Future<Output = ()>,

Self: Sized,

{

{ ... }
```

今天你已经完成了 Rust 学习的第 23 次打卡。如果你觉得有收获，也欢迎你分享给身边的朋友，邀他一起讨论。我们下节课见。

10人觉得很赞给文章提建议

良师益友

Command + Enter 发表

0/2000字

提交留言

精选留言(4)

- 

Marvichov 

和老师的对应下...

1. 使用泛型参数延迟数据结构的绑定；
2. 使用泛型参数和 PhantomData，声明数据结构中不直接使用，但在实现过程中需要用到的类型
3. 使用泛型参数让同一个数据结构对同一个 trait 可以拥有不同的实现。

1. 面向interface编程; 只不过静态多态
2. 引入自由参数 -> 大部分impl共享; 剩下的, 根据自由参数类型的不同做template specialization -> 比如tag struct -> 本质还是代码共享
3. NewTypePattern; 一套代码给多个不同的type共用; 这个blog里面的例子比较生动: https://www.greyblake.com/blog/2021-10-11-phantom-types-in-rust/; golang里面也经常用kilometer, mile来做例子, 类似于`type mile i32`;
作者回复: 👍

2021-10-18

3

- 

Marvichov 

Cpp里面用tag和多generic param的例子也很多...

比如Cpp的iterator, 多个泛型做参数, 不需要PhantomData;

```
template<
    class Category, // tag data, 类似于 AsyncProstReader D
    class T,
    class Distance = std::ptrdiff_t,
    class Pointer = T*,
    class Reference = T&
> struct iterator;
```

所以, 感觉PhantomData的主要用途是compile time ownership check;

我的疑问也就主要集中在ownership...

问题1:

from PhantomData doc:
> Adding a PhantomData field to your type tells the compiler that your type acts as though it stores a value of type T, even though it doesn't really.

不太明白, 为啥需要ownership. 比如AsyncProstReader的T是约束R的return type的, 按理说不用own T; 而且into和dest member全程没被调用过

```
/// A wrapper around an async reader that produces an asynchronous stream of
prost-decoded values
#[derive(Debug)]
pub struct AsyncProstReader<R, T, D> {
    reader: R,
    pub(crate) buffer: BytesMut,
    into: PhantomData<T>,
    dest: PhantomData<D>,
}
```

如果不需要对T, D的ownership, 为啥不来个`PhantomDataNotOwned`来满足这样的场景: 不需要ownership, 但是这个generic type T不是多余的呢?

问题2:

> This information is used when computing certain safety properties.

这句目前理解不了...假设对T有ownership, 没看出有啥特殊的safety需求

作者回复: 我的理解是大部分时候 PhantomData 跟其它语言的 Phantom Type 是一个作用，为数据结构提供声明时没用用到，但在实现时需要用到的类型。因为这里你实实在在就只用 T 来保证类型的正确性，并没有涉及到 owership。

但在有些场合下，比如 Unique<T>，这里，如果没用 PhantomData<T>的话，你想想 Unique<T> 是否 own T？并不 own，因为 pointer 是一个指针类型，所以从类型上，Unique<T> 不 own T，但这里 Unique<T> 应该 own T 才对。所以 Rust 使用 PhantomData 来表述这个作用，见：https://github.com/rust-lang/rfcs/blob/master/text/0769-sound-generic-drop.md#phantom-data

```Rust
pub struct Unique<T: ?Sized> {
    pointer: *const T,
    _marker: PhantomData<T>,
}
```

这属于 PhantomData 的高级用法，大部分时候我们用类型系统解决问题需要使用 PhantomData 时，都是大家在其他语言中惯常的用法，所以我没有提这个 owership 的用法。

2021-10-18

1

2

- 

Custer 

1. 参数 F 是一个闭包，接收 Self::Item,返回 Fut 类型；
2. 参数 Fut 是一个 Future<Output=()>;

所以 f 是一个闭包接收 Self::Item 闭包的返回值是 Future<Output=()>

使用参考源代码：

```rust
.for_each_concurrent(
    /* limit */ 2,
    |rx| async move {
        rx.await.unwrap();
    }
)
```

作者回复：👍

2021-10-28

- 

罗杰 

impl Iterator for Equation<Quadratic> 判断返回 None 的地方是不是应该写成 `if self.current >= u16::MAX as u32`，不然会有逻辑错误。

作者回复: 👍是的，非常厉害！这个 bug 我也是又看了一遍代码才发现。

2021-10-20

收起评论