



下载APP



23 | 类型系统：如何在实战中使用泛型编程？

2021-10-18 陈天

《陈天 · Rust 编程第一课》

课程介绍 >



讲述：陈天


时长 17:23 大小 15.93M



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  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：

```
1 pub struct BufReader<R> {
2     inner: R,
3     buf: Box<[u8]>,
4     pos: usize,
5     cap: usize,
6 }
```

[📄 复制代码](#)

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](#)：

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

[📄 复制代码](#)

```
2     pub fn capacity(&self) -> usize { ... }
3     pub fn buffer(&self) -> &[u8] { ... }
4 }
```

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](#)：

```
1 impl<R: Read> BufReader<R> {
2     pub fn new(inner: R) -> BufReader<R> { ... }
3     pub fn with_capacity(capacity: usize, inner: R) -> BufReader<R> { ... }
4 }
```

[复制代码](#)

同样，在实现 `Debug` 时，也可以要求 [R 满足 Debug trait 的约束](#)：

```
1 impl<R> fmt::Debug for BufReader<R>
2 where
3     R: fmt::Debug
4 {
5     fn fmt(&self, fmt: &mut fmt::Formatter<'_>) -> fmt::Result { ... }
6 }
```

[复制代码](#)

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中的 `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：

 复制代码

```
1 /// Service 数据结构
2 pub struct Service<Store = MemTable> {
3     inner: Arc<ServiceInner<Store>>,
4 }
```

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：

 复制代码

```
1 impl<Store> Service<Store> {
2     pub fn new(store: Store) -> Self { ... }
3 }
4
5 impl<Store: Storage> Service<Store> {
6     pub fn execute(&self, cmd: CommandRequest) -> CommandResponse { ... }
7 }
```

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：

 复制代码

```
1 pub fn dispatch(cmd: CommandRequest, store: &impl Storage) -> CommandResponse
2 // 等价于
3 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：

 复制代码

```
1 pub struct Identifier<T> {  
2     inner: u64,  
3 }
```

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](#)）：

 复制代码


```
1 #[derive(Debug, Default, PartialEq, Eq)]  
2 pub struct Identifier<T> {  
3     inner: u64,  
4 }  
5  
6 #[derive(Debug, Default, PartialEq, Eq)]  
7 pub struct User {  
8     id: Identifier<Self>,  
9 }  
10  
11 #[derive(Debug, Default, PartialEq, Eq)]  
12 pub struct Product {  
13     id: Identifier<Self>,  
14 }
```

```
15 #[cfg(test)]
16 mod tests {
17     use super::*;
18
19     #[test]
20     fn id_should_not_be_the_same() {
21         let user = User::default();
22         let product = Product::default();
23
24         // 两个 id 不能比较, 因为他们属于不同的类型
25         // assert_ne!(user.id, product.id);
26
27         assert_eq!(user.id.inner, product.id.inner);
28     }
29 }
30
```

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：


 复制代码

```
1 // NotUsed is allowed
2 class MyNumber<T, NotUsed> {
3     inner: T;
4     add: (x: T, y: T) => T;
5 }
```

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：


 复制代码

```
1 use std::marker::PhantomData;
2
3 #[derive(Debug, Default, PartialEq, Eq)]
4 pub struct Identifier<T> {
5     inner: u64,
6     _tag: PhantomData<T>,
7 }
8
9 #[derive(Debug, Default, PartialEq, Eq)]
10 pub struct User {
11     id: Identifier<Self>,
12 }
13
14 #[derive(Debug, Default, PartialEq, Eq)]
15 pub struct Product {
16     id: Identifier<Self>,
17 }
18
19 #[cfg(test)]
20 mod tests {
21     use super::*;
22
23     #[test]
24     fn id_should_not_be_the_same() {
25         let user = User::default();
26         let product = Product::default();
27
28         // 两个 id 不能比较，因为他们属于不同的类型
29         // assert_ne!(user.id, product.id);
30
31         assert_eq!(user.id.inner, product.id.inner);
32     }
33 }
```

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)：

 复制代码

```
1 use std::{
2     marker::PhantomData,
3     sync::atomic::{AtomicU64, Ordering},
4 };
5
6 static NEXT_ID: AtomicU64 = AtomicU64::new(1);
7
8 pub struct Customer<T> {
9     id: u64,
10    name: String,
11    _type: PhantomData<T>,
12 }
13
14 pub trait Free {
15     fn feature1(&self);
16     fn feature2(&self);
17 }
18
19 pub trait Personal: Free {
20     fn advance_feature(&self);
21 }
22
23 impl<T> Free for Customer<T> {
24     fn feature1(&self) {
25         println!("feature 1 for {}", self.name);
26     }
27
28     fn feature2(&self) {
29         println!("feature 2 for {}", self.name);
30     }
31 }
32
33 impl Personal for Customer<PersonalPlan> {
34     fn advance_feature(&self) {
35         println!(
36             "Dear {}(as our valuable customer {}), enjoy this advanced feature",
37             self.name, self.id
38         );
39     }
40 }
```



```
41
42 pub struct FreePlan;
43 pub struct PersonalPlan(f32);
44
45 impl<T> Customer<T> {
46     pub fn new(name: String) -> Self {
47         Self {
48             id: NEXT_ID.fetch_add(1, Ordering::Relaxed),
49             name,
50             _type: PhantomData::default(),
51         }
52     }
53 }
54
55 impl From<Customer<FreePlan>> for Customer<PersonalPlan> {
56     fn from(c: Customer<FreePlan>) -> Self {
57         Self::new(c.name)
58     }
59 }
60
61 /// 订阅成为付费用户
62 pub fn subscribe(customer: Customer<FreePlan>, payment: f32) -> Customer<Perso
63     let _plan = PersonalPlan(payment);
64     // 存储 plan 到 DB
65     // ...
66     customer.into()
67 }
68
69 #[cfg(test)]
70 mod tests {
71     use super::*;
72
73     #[test]
74     fn test_customer() {
75         // 一开始是个免费用户
76         let customer = Customer::::new("Tyr".into());
77         // 使用免费 feature
78         customer.feature1();
79         customer.feature2();
80         // 用着用着觉得产品不错愿意付费
81         let customer = subscribe(customer, 6.99);
82         customer.feature1();
83         customer.feature1();
84         // 付费用户解锁了新技能
85         customer.advance_feature();
86     }
87 }
```

在这个例子中，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](#)）：

[📄 复制代码](#)

```
1 use std::marker::PhantomData;
2
3 #[derive(Debug, Default)]
4 pub struct Equation<IterMethod> {
5     current: u32,
6     _method: PhantomData<IterMethod>,
7 }
8
9 // 线性增长
10 #[derive(Debug, Default)]
11 pub struct Linear;
12
13 // 二次增长
14 #[derive(Debug, Default)]
15 pub struct Quadratic;
16
17 impl Iterator for Equation<Linear> {
18     type Item = u32;
19
20     fn next(&mut self) -> Option<Self::Item> {
21         self.current += 1;
22         if self.current >= u16::MAX as u32 {
23             return None;
24         }
25
26         Some(self.current)
27     }
28 }
29
30 impl Iterator for Equation<Quadratic> {
```

```
31     type Item = u32;
32
33     fn next(&mut self) -> Option<Self::Item> {
34         self.current += 1;
35         if self.current >= u32::MAX {
36             return None;
37         }
38
39         Some(self.current * self.current)
40     }
41 }
42
43 #[cfg(test)]
44 mod tests {
45     use super::*;
46
47     #[test]
48     fn test_linear() {
49         let mut equation = Equation::<Linear>::default();
50         assert_eq!(Some(1), equation.next());
51         assert_eq!(Some(2), equation.next());
52         assert_eq!(Some(3), equation.next());
53     }
54
55     #[test]
56     fn test_quadratic() {
57         let mut equation = Equation::<Quadratic>::default();
58         assert_eq!(Some(1), equation.next());
59         assert_eq!(Some(4), equation.next());
60         assert_eq!(Some(9), equation.next());
61     }
62 }
```

这个代码很好理解，但你可能会有疑问：这样做有什么好处？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：

[复制代码](#)

```
1  /// A marker that indicates that the wrapping type is compatible with `AsyncPr
2  #[derive(Debug)]
3  pub struct AsyncDestination;
4
5  /// a marker that indicates that the wrapper type is compatible with `AsyncPro
6  #[derive(Debug)]
7  pub struct AsyncFrameDestination;
8
9  /// A wrapper around an async reader that produces an asynchronous stream of p
10 #[derive(Debug)]
11 pub struct AsyncProstReader<R, T, D> {
12     reader: R,
13     pub(crate) buffer: BytesMut,
14     into: PhantomData<T>,
15     dest: PhantomData<D>,
16 }
```

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，我们希望在使用的 AsyncDestination 时，提供一种实现，而在使用的 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](#)）：

```
1 impl<R, T> Stream for AsyncProstReader<R, T, AsyncDestination>
2 where
3     T: Message + Default,
4     R: AsyncRead + Unpin,
5 {
6     type Item = Result<T, io::Error>;
7
8     fn poll_next(mut self: Pin<&mut Self>, cx: &mut Context<'_>) -> Poll<Optio
9         ...
10    }
11 }
```

复制代码

再看对另外一个对 D 的具体实现：

```
1 impl<R, T> Stream for AsyncProstReader<R, T, AsyncFrameDestination>
2 where
3     R: AsyncRead + Unpin,
4     T: Framed + Default,
5 {
6     type Item = Result<T, io::Error>;
7
8     fn poll_next(mut self: Pin<&mut Self>, cx: &mut Context<'_>) -> Poll<Optio
9         ...
10    }
11 }
```

复制代码

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：

[复制代码](#)

```
1 pub trait Storage {
2     ...
3     /// 遍历 HashTable, 返回 kv pair 的 Iterator
4     fn get_iter(&self, table: &str) ->
5         Result<Box<dyn Iterator<Item = Kvpair>>, KvError>;
6 }
```

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？

[复制代码](#)

```
1 // 目前 trait 还不支持
2 fn get_iter(&self, table: &str) -> Result<impl Iterator<Item = Kvpair>, KvErro
```

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

[复制代码](#)

```
1 pub trait ImplTrait {
2     // 允许
3     fn impl_in_args(s: impl Into<String>) -> String {
4         s.into()
5     }
6
7     // 不允许
```

```
8     fn impl_as_return(s: String) -> impl Into<String> {
9         s
10    }
11 }
```

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：

[复制代码](#)

```
1 // 可以正确编译
2 pub fn generics_as_return_working(i: u32) -> impl Iterator<Item = u32> {
3     std::iter::once(i)
4 }
5
6 // 期待泛型类型，却返回一个具体类型
7 pub fn generics_as_return_not_working<T: Iterator<Item = u32>>(i: u32) -> T {
8     std::iter::once(i)
9 }
```

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：

[复制代码](#)

```
1 // 返回 trait object
2 pub fn trait_object_as_return_working(i: u32) -> Box<dyn Iterator<Item = u32>>
3     Box::new(std::iter::once(i))
4 }
```

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

[复制代码](#)


```
1 pub trait Storage {
2     ...
3     /// 遍历 HashTable，返回 kv pair 的 Iterator
4     fn get_iter(&self, table: &str) ->
5         Result<Box<dyn Iterator<Item = Kvpair>>, KvError>;
6 }
```


现在你是不是更好地理解，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：


 复制代码

```
1 pub fn consume_iterator<F, Iter, T>(mut f: F)
2 where
3     F: FnMut(i32) -> Iter, // F 是一个闭包，接受 i32，返回 Iter 类型
4     Iter: Iterator<Item = T>, // Iter 是一个 Iterator，Item 是 T 类型
5     T: std::fmt::Debug, // T 实现了 Debug trait
6 {
7     // 根据 F 的类型，f(10) 返回 iterator，所以可以用 for 循环
8     for item in f(10) {
9         println!("{:?}", item); // item 实现了 Debug trait，所以可以用 {:?} 打印
10    }
11 }
```

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：

1. 参数 F 是一个闭包，接受 i32，返回 Iter 类型；
2. 参数 Iter 是一个 Iterator，Item 是 T 类型；
3. 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：

 复制代码

```
1 #[cfg(test)]
2 mod tests {
3     use super::*;
4
5     #[test]
```

```
6     fn test_consume_iterator() {  
7         // 不会 panic 或者出错  
8         consume_iterator(|i| (0..i).into_iter())  
9     }  
10 }
```

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？

 复制代码

```
1 fn for_each_concurrent<Fut, F>(  
2     self,  
3     limit: impl Into<Option<usize>>,  
4     f: F,  
5 ) -> ForEachConcurrent<Self, Fut, F>  
6 where  
7     F: FnMut(Self::Item) -> Fut,  
8     Fut: Future<Output = ()>,  
9     Self: Sized,  
10 {  
11 { ... }
```

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。

分享给需要的人，Ta 订阅后你可得 **20 元现金奖励**

 生成海报并分享

 赞 6  提建议

© 版权归极客邦科技所有，未经许可不得传播售卖。页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇 [加餐 | 期中测试：参考实现讲解](#)

下一篇 [24 | 类型系统：如何在实战中使用 Trait Object？](#)

1024 活动特惠

VIP 年卡直降 ¥2000

新课上线即解锁, 享 365 天畅看全场

超值拿下 ¥999



精选留言 (3)

写留言



Marvichov

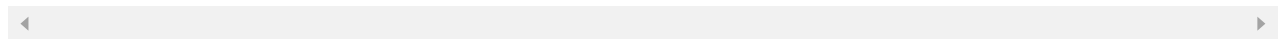
2021-10-18

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...

展开

作者回复:



3



Marvichov

2021-10-18

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<...

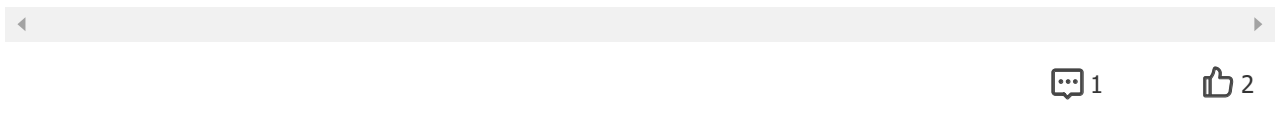
展开

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 ownership。

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 ownership 的用法。



罗杰 

2021-10-20

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。

展开 ∨

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。

