

程序设计训练之 Rust 编程语言

第四讲：泛型、特型与生命周期

韩文弢

清华大学计算机科学与技术系

2024 年 7 月

泛型

泛型

- 将类型作为参数，变成泛型枚举类型。
 - 考虑以下类似标准库的枚举类型 `Result<T, E>`：

```
enum Result<T, E> {  
    Ok(T),  
    Err(E),  
}
```

- **T** 和 **E** 是泛型类型。

泛型结构体

将 Point 和 List 泛型化:

```
struct Point<T> {
    x: T,
    y: T,
}

enum List<T> {
    Nil,
    Cons(T, Box<List<T>>),
}
```

泛型实现

- 为泛型结构体或枚举类型定义实现时，在 `impl` 代码段的开头声明泛型类型：

```
impl<T, E> Result<T, E> {
    fn is_ok(&self) -> bool {
        match *self {
            Ok(_) => true,
            Err(_) => false,
        }
    }
}
```

泛型函数

- 也可以定义泛型函数，将函数调用的相关类型改为泛型：
- `<T, U>` 声明了 `foo` 函数的类型参数。
 - `x: T, y: U` 使用了声明的类型参数。
- 可以读作“对于所有类型 `T` 和 `U`，`foo` 函数有两个参数，分别是 `T` 类型的 `x` 和 `U` 类型的 `y`”。

```
fn foo<T, U>(x: T, y: U) {  
    // ...  
}
```

2

特型

- 使用 `impl Trait for Type` 代码段来实现特型。
 - 所有特型所指定的方法都必须实现。
- 对于一种类型实现一种特型，要匹配一个相应的 `impl` 代码段。
- 在特型的 `impl` 代码段中，同样可以使用 `self/&self` 参数。

韩文弢


```
In file included from my_sort.cpp:1:
In file included from .../usr/include/c++/v1/algorithm:701:
.../usr/include/c++/v1/__algorithm/make_heap.h:32:34: error: invalid
  operands to binary expression ('std::__list_iterator<int, void *>'
  and 'std::__list_iterator<int, void *>')
      difference_type __n = __last - __first;
                          ~~~~~ ^ ~~~~~
.../usr/include/c++/v1/__algorithm/partial_sort.h:35:12: note: in
  instantiation of function template specialization
  'std::__make_heap<std::__less<int> &, std::__list_iterator<int, void *>>'
  requested here
      ...
__wrap_iter<_Iter1> operator+(...) _NOEXCEPT
                          ^
11 errors generated.
```

```

16 行:

my_sort.cpp:13:3: error: no matching function for call to 'my_sort'
    my_sort(a.begin(), a.end());
    ^~~~~~

my_sort.cpp:7:6: note: candidate template ignored: constraints not
      satisfied [with It = std::__list_iterator<int, void *>]
void my_sort(It first, It last) {
    ^

my_sort.cpp:6:16: note: because 'std::__list_iterator<int, void *>'
      does not satisfy 'random_access_iterator'
template <std::random_access_iterator It>
    ^

...

1 error generated.

```

使用特型约束的泛型

- 可以使用特型对泛型的类型参数进行约束。
- 能够更加准确地描述泛型函数和类型，指定对类型参数的要求。
- 泛型中的特型约束可以直接在类型参数的地方用 `T: SomeTrait` 或者用单独的子句 `where T: SomeTrait` 来指定。

```
fn cloning_machine<T: Clone>(t: T) -> (T, T) {
    (t.clone(), t.clone())
}
```

```
fn cloning_machine_2<T>(t: T) -> (T, T)
    where T: Clone {
    (t.clone(), t.clone())
}
```

多种特型约束

- 使用形如 `T: Clone + Ord` 的方式来指定多种特型约束。

```
fn clone_and_compare<T: Clone + Ord>(t1: T, t2: T) -> bool {
    t1.clone() > t2.clone()
}
```


示例：特型约束的泛型类型

```
enum Result<T, E> {
    Ok(T),
    Err(E),
}

trait PrettyPrint {
    fn format(&self) -> String;
}

impl<T: PrettyPrint, E: PrettyPrint> PrettyPrint for Result<T, E> {
    fn format(&self) -> String {
        match *self {
            Ok(t) => format!("Ok({})", t.format()),
            Err(e) => format!("Err({})", e.format()),
        }
    }
}
```

示例：相等关系

```
enum Result<T, E> { Ok(T), Err(E) }  
// This is not the trait Rust actually uses for equality  
trait Equals {  
    fn equals(&self, other: &Self) -> bool;  
}  
  
impl<T: Equals, E: Equals> Equals for Result<T, E> {  
    fn equals(&self, other: &Self) -> bool {  
        match (*self, *other) {  
            Ok(t1), Ok(t2) => t1.equals(t2),  
            Err(e1), Err(e2) => e1.equals(e2),  
            _ => false  
        }  
    }  
}
```

- Self 是一种特殊的类型，表示当前实现的类型（也就是 self 的类型）。

继承

- 特型之间存在逻辑上的先后关系。
 - 例如, `Eq` 需要先实现 `PartialEq`, `Copy` 需要先实现 `Clone`。
- 下面的代码表示实现 `Child` 特型要先实现 `Parent` 特型。

```
trait Parent {  
    fn foo(&self) {  
        // ...  
    }  
}  
  
trait Child: Parent {  
    fn bar(&self) {  
        self.foo();  
        // ...  
    }  
}
```

默认方法

- 特型可以指定默认的方法实现。
 - 用于避免重复实现那些具有一般意义下常见实现方式的方法。
- 当某个方法在特型中提供默认实现时，特型的实现中就不用再定义这个方法。
- 定义默认实现的方式是在 `trait` 代码段写出方法的实现。

```
trait PartialEq<Rhs: ?Sized = Self> {
    fn eq(&self, other: &Rhs) -> bool;

    fn ne(&self, other: &Rhs) -> bool {
        !self.eq(other)
    }
}
```

```
trait Eq: PartialEq<Self> {}
```

默认方法的改写

- 实现特型时可以改写默认方法的实现。
- 但是一要想好有充分的理由这样做。
 - 例如，重新定义 `ne` 导致违反了 `eq` 和 `ne` 之间的逻辑关系，类似这样的事情一定不要做。

特型的自动获得

- 一些特型实现起来比较机械，编译器可以自动完成。
- 使用 `#[derive(...)]` 属性让编译器完成相应特型的自动实现。
- 这样做可以避免重复手动实现诸如 `Clone` 这样的特型。

```
#[derive(Debug, Eq, PartialEq, /* ... */)]  
enum Result<T, E> {  
    Ok(T),  
    Err(E),  
}
```

特型自动获得的限制

- Rust 语言支持自动获得下列核心特型：
 - Clone, Copy, Debug, Default, Eq,
 - Hash, Ord, PartialEq, PartialOrd.
- 可以使用宏来完成自定义特型的自动获得。
- 注意：特型的自动获得需要满足下列条件：
 - 类型的所有成员都能自动获得指定的特型。
 - 例如，Eq 不能在包含 f32 的结构体类型上自动获得，因为 f32 不是 Eq 的。

核心特型

- 有必要了解下列 Rust 的核心特型：
 - Clone, Copy
 - Debug
 - Default
 - Eq, PartialEq
 - Hash
 - Ord, PartialOrd

Clone

```
pub trait Clone: Sized {  
    fn clone(&self) -> Self;  
  
    fn clone_from(&mut self, source: &Self) { ... }  
}
```

- **Clone** 特型定义了如何复制 **T** 类型的一个值。
- 解决所有权问题的另一种方法。
 - 克隆一个对象，而不是获得所有权或者借用所有权。

Clone 示例

```
#[derive(Clone)] // without this, Bar cannot derive Clone.
struct Foo {
    x: i32,
}

#[derive(Clone)]
struct Bar {
    x: Foo,
}
```

Copy

```
pub trait Copy: Clone { }
```

- Copy 特型表示一种类型是拷贝语义，而不是 Rust 默认的移动语义。
- 类型必须可以通过位拷贝来进行拷贝（相当于 C 语言中的 `memcpy`）。
 - 包含可变引用的类型不能实现 Copy 特型（不可变引用可以）。
- 标记特型：没有实现任何方法，只是标记行为。
- 一般来说，如果一种类型可以拷贝，就应该实现 Copy 特型。

Debug

```
pub trait Debug {  
    fn fmt(&self, &mut Formatter) -> Result;  
}
```

- 定义能够使用 `{:?}` 格式选项进行输出。
- 产生的是用于调试的输出信息，不是美观的输出格式。
- 一般来说，`Debug` 特型应该通过自动获得的方式实现。

Debug 示例

```
#[derive(Debug)]  
struct Point {  
    x: i32,  
    y: i32,  
}  
  
let origin = Point { x: 0, y: 0 };  
println!("The origin is: {:?}", origin);  
// The origin is: Point { x: 0, y: 0 }
```

Default

```
pub trait Default: Sized {  
    fn default() -> Self;  
}
```

- 为一种类型定义一个默认值。

Eq 与 PartialEq

```
pub trait PartialEq<Rhs: ?Sized = Self> {  
    fn eq(&self, other: &Rhs) -> bool;  
  
    fn ne(&self, other: &Rhs) -> bool { /* ... */ }  
}
```

```
pub trait Eq: PartialEq<Self> {}
```

- 定义通过 `==` 操作符判断相等关系的特型。

Eq 与 PartialEq 的解释

- PartialEq 表示部分等价关系 (partial equivalence relation)。
 - 对称性: 若 $a == b$, 则 $b == a$
 - 传递性: 若 $a == b$ 且 $b == c$, 则 $a == c$
- ne 具有使用 eq 的默认实现。
- Eq 表示等价关系 (equivalence relation)。
 - 除对称性和传递性外, 还需要满足自反性。
 - 自反性: $a == a$
- Eq 没有定义更多的方法, 也是一种标记特型。

Hash

```
pub trait Hash {  
    fn hash<H: Hasher>(&self, state: &mut H);  
  
    fn hash_slice<H: Hasher>(data: &[Self], state: &mut H)  
        where Self: Sized { /* ... */ }  
}
```

- 表示可哈希的类型。
- H 类型参数是抽象的哈希状态，用于计算哈希值。
- 如果同时实现了 Eq 特型，需要满足如下性质：

$$k1 == k2 \rightarrow \text{hash}(k1) == \text{hash}(k2)$$

PartialOrd

```
pub trait PartialOrd<Rhs: ?Sized = Self>: PartialEq<Rhs> {  
    // Ordering is one of Less, Equal, Greater  
    fn partial_cmp(&self, other: &Rhs) -> Option<Ordering>;  
  
    fn lt(&self, other: &Rhs) -> bool { /* ... */ }  
    fn le(&self, other: &Rhs) -> bool { /* ... */ }  
    fn gt(&self, other: &Rhs) -> bool { /* ... */ }  
    fn ge(&self, other: &Rhs) -> bool { /* ... */ }  
}
```

- 表示（可能）可以进行比较的特型。

PartialOrd 的解释

- 对所有的 a 、 b 、 c ，比较操作必须满足：
 - 反对称性：若 $a < b$ ，则 $!(a > b)$ ；若 $a > b$ ，则 $!(a < b)$ 。
 - 传递性：若 $a < b$ 且 $b < c$ ，则 $a < c$ ；对 $==$ 和 $>$ 同样成立。
- `lt`、`le`、`gt`、`ge` 具有基于 `partial_cmp` 的默认实现。

Ord

```
pub trait Ord: Eq + PartialOrd<Self> {  
    fn cmp(&self, other: &Self) -> Ordering;  
}
```

- 实现该特型的类型形成全序关系 (total order)。
- 全序关系需要满足的性质除反对称性和传递性外，还需要满足完全性：
 - 对所有的 a 和 b ，有 $a \leq b$ 或 $b \leq a$ 成立。
- 此特型可以保证类型的值能够按字典序排列。

关联类型的需求

- 考虑如下的 Graph 特型:

```
trait Graph<N, E> {  
    fn edges(&self, node: &N) -> Vec<E>;  
    // ...  
}
```

- 这里, N 和 E 是泛型类型参数, 但是它们和 Graph 之间的联系不明确。
- 如果有函数要使用 Graph 时, 它必须也是 N 和 E 的泛型。

```
fn distance<N, E, G: Graph<N,E>>(graph: &G, start: &N, end: &N)  
    -> u32 { /* ... */ }
```

- 逻辑: 特型 Graph 是 N 和 E 的泛型, 还是特型 Graph 带着 N 和 E 两种关联的类型?

关联类型

- 使用关联类型来反映这种设计上的逻辑。
- 使用特型代码段里的 `type` 定义来表示特型关联的泛型类型。
- 特型在实现时来指定关联类型实际指代的类型。

```
trait Graph {  
    type N;  
    type E;  
    fn edges(&self, &Self::N) -> Vec<Self::E>;  
}  
  
impl Graph for MyGraph {  
    type N = MyNode;  
    type E = MyEdge;  
    fn edges(&self, n: &MyNode) -> Vec<MyEdge> { /* ... */ }  
}
```

示例：关联类型

// 泛型特型

```
trait Graph<N, E> {
    fn edges(&self, node: &N) -> Vec<E>;
}

fn distance<N, E, G: Graph<N,E>>(graph: &G, start: &N, end: &N)
    -> u32 { /* ... */ }
```

// 带关联类型的特型

```
trait Graph {
    type N;
    type E;
    fn edges(&self, &Self::N) -> Vec<Self::E>;
}

fn distance<Graph>(graph: &G, start: &G::N, end: &G::N)
    -> u32 { /* ... */ }
```


关联类型的应用

- 例如，在标准库中，迭代器特型 `Iterator` 具有表示元素类型的关联类型 `Item`。
- 诸如 `Iterator::next` 这样的特型方法会返回 `Option<Self::Item>` 类型的值。
 - 可以方便地指定迭代容器时所获得的值的类型。
- 关联类型可以在实现特型时指定某些特定的相关联的类型，而不是全部作为泛型的类型参数。

特型的作用域

- 假设在程序中定义了某特型 `Foo`。
- 在 `Rust` 中，可以在任何类型上实现这种特型，哪怕不是你写的类型。

```
trait Foo {  
    fn bar(&self) -> bool;  
}  
  
impl Foo for i32 {  
    fn bar(&self) -> bool {  
        true  
    }  
}
```

- 这样做是否合理，要慎重考虑。

特型的作用域规则

- 实现特型的作用域规则如下：
 - 需要 **use** 要用的特型来访问类型中由这种特型定义的方法，即使已经有对类型的访问权限也还是不够的。
 - 为了写一个特型实现的 **impl** 代码段，需要要么拥有（也就是通过自己的代码来定义）该特型，要么拥有该类型，两者至少其一。

特型的作用域规则示例: Display

```
use std::fmt::Display;

impl Display for Point {
    fn fmt(&self, f: &mut fmt::Formatter) -> std::fmt::Result {
        write!(f, "Point ({}", self.x, self.y)
    }
}

println!("{}", Point { x: 3, y: 4 });
```

- 定义 {} 格式化选项的输出行为。
- 用于美观打印, 不能由编译器自动获得。
- 可以用 `write!` 宏来做具体的实现。

特型的作用域规则示例: `rand::seq::SliceRandom`

```
use rand::seq::SliceRandom;

let mut rng = rand::thread_rng();
let mut bytes = "Hello, random!".to_string().into_bytes();
// 等价于 SliceRandom::shuffle(&bytes, &mut rng);
bytes.shuffle(&mut rng);
let str = String::from_utf8(bytes).unwrap();
println!("{}", str);
```

特型方法消歧

```
trait Flyable { fn go(&self); }
trait Walkable { fn go(&self); }
struct Bird;
impl Flyable for Bird { /* ... */ }
impl Walkable for Bird { /* ... */ }

let bird = Bird;
// Compile error: bird.go();
Flyable::go(&bird);
Walkable::go(&bird);
```

Drop 特型

```
pub trait Drop {  
    fn drop(&mut self);  
}
```

- 表示可销毁的特型（所有类型都要实现）。
- Drop 特型提供 `drop` 方法，用于将对象销毁，会由编译器自动生成，不能显式调用。
- 可以使用 `std::mem::drop` 函数来调用 `drop` 方法。

Drop 特型的说明

- 一般情况下，不需要实现 `Drop`。
 - 默认的实现可以正常工作。
 - 也不需要自动获得功能去产生。
- 什么时候需要手动实现？
 - 如果在对象销毁时有特殊行为。
 - 例如，`Rust` 的引用计数指针类型 `Rc<T>` 就有特殊的 `Drop` 规则：当引用计数大于 1 时，`drop` 只是对计数器减 1；但是当减 1 之后引用计数降为 0 时，要真的删除这个 `Rc` 对象。

Sized 和 ?Sized 特型

- **Sized** 表示一种类型在编译时就可以知道是固定的大小。
- 而 **?Sized** 表示一种类型的大小可能是不固定的。
- 默认情况下，所有类型都是 **Sized**，而指定 **?Sized** 可以撤销这一规定。
 - 例如，像 `[T]` 和 `str`（没有 `&`）都是 **?Sized** 的。
- 一般来说，跟指针相关的泛型的类型参数里的特型约束会出现 **?Sized**，例如 `Box<T>` 就有 `T: ?Sized`。
- 很少直接使用这两种特型，一般都是在特型约束中出现的。

3

特型对象

特型对象的引入

- 考虑以下特型和实现:

```
trait Foo { fn bar(&self); }
```

```
impl Foo for String {  
    fn bar(&self) { /* ... */ }  
}
```

```
impl Foo for usize {
    fn bar(&self) { /* ... */ }
}
```


错误使用特型对象的示例

// 接上一页的特型和类型定义

```
fn draw(shape: &dyn Shape) {  
    match *shape {  
        Rectangle { .. } => println!("This is a rectangle"),  
        Circle { .. } => println!("This is a circle"),  
    }  
}
```

韩文弢

- 清华大学计算机科学与技术系

生命周期

韩文弢

- 清华大学计算机科学与技术系

生命周期的显式表示

- 通常情况下，引用具有隐式生命周期，不需要额外关注：

```
fn foo(x: &i32) {  
    // ...  
}
```

- 在必要的时候，也可以显式指定生命周期：

```
fn bar<'a>(x: &'a i32) {  
    // ...  
}
```

- 参数。

- `&'a i32` 类型是一个 `i32` 的引用，它的生命周期至少有 `'a` 那么长。

生命周期的用处

- 一般情况下，编译器可以推断生命周期，不需要显式声明。
- 在涉及多个引用或者要返回引用的场景时，可能需要显式指定生命周期。
 - 来看几个例子：

```
fn print(s: &str); // 省略
fn print<'a>(s: &'a str); // 展开
```

```
fn debug(lvl: usize, s: &str); // 省略
fn debug<'a>(lvl: usize, s: &'a str); // 展开
```

```
fn substr(s: &str, until: usize) -> &str;           // 省略
fn substr<'a>(s: &'a str, until: usize) -> &'a str;   // 展开
```

```
fn get_str() -> &str; // 非法
```

```
fn frob(s: &str, t: &str) -> &str; // 非法
```


- 结构体和结构体成员也可以具有生命周期参数。

```
let p1 = Pizza(vec![1, 2, 3, 4]);
{
    let s1 = PizzaSlice { pizza: &p1, index: 2 }; // 可行
}
```


韩文弢

- 清华大学计算机科学与技术系

程序设计训练之 Rust 编程语言

- 66 / 71

静态生命周期 'static

- `'static` 表示整个程序的生命周期，拥有 `'static` 生命周期的引用在整个程序运行过程中都有效。
 - 也就是说，这样的数据在程序中永远不会失效。
- 所有的 `&str` 字面值是 `'static` 生命周期的。

```
let s1: &str = "Hello";  
let s2: &'static str = "World";
```

生命周期的解释

- 如何理解显式的生命周期？
 - 如果引用 **r** 具有生命周期 '**a**'，首先会确保 **r** 不会比它所引用的数据（也就是 ***r**）的所有者活得更久。
 - 其次，对于其他也拥有生命周期 '**a**' 的对象，能够确保它们至少和 **r** 活得一样久。
- 生命周期的概念比较抽象，需要在实践中体会和理解。

5

小结

