# 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

2022 年 8 月 31 日

韩文弢

1

泛型

泛型 ●00000

# 一般化类型的需求

000000

- 对于结构化数据和函数,有针对一般化类型的需求:
  - 容器: 向量、哈希表等
  - 算法: 排序、查找、求最值等
- 需要提供一般化的机制,避免编写重复的代码。

# 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。
- 用 CamelCase 的标识符来表示泛型类型。

# 泛型结构体

000000

• 考虑以下泛型结构体:

```
struct Point<T> {
        x: T,
        y: T,
}
enum List<T> {
        Nil,
        Cons(T, Box<List<T>>),
}
```

# 泛型实现

000000

● 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:

# 泛型函数

000000

- 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

程序设计训练之 Rust 编程语言

#### 类型共性的需求

- 一些类型具有共性,例如,支持美观打印、判断相等、比较大小等功能。
- 针对每种类型进行实现是可行的,但是缺乏结构性。

```
struct Point {
    x: i32,
    v: i32,
impl Point {
    fn format(&self) -> String {
        format!("({}, {})", self.x, self.y)
    fn equals(&self, other: Point) -> bool {
        self.x == other.x && self.y == other.y
```

## 解决方案:特型

- 为了抽象类型共性机制, Rust 使用特型 (trait) 的概念。
- 使用 trait 代码段来定义特型,列出特型所需的方法。
  - 与 impl 代码段不同。
  - 大多数方法只列出方法的签名,不包含定义。

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

## 实现特型

- 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。

```
struct Point {
   x: i32,
   v: i32,
impl PrettyPrint for Point {
   fn format(&self) -> String {
       format!("({}, {})", self.x, self.y)
```

## 泛型编程对类型参数的约束要求

- 在使用泛型的场景中,有时候需要对泛型的类型参数做一定的约束(也就是满足一定的条件)。
  - 回忆之前提到的 C++ 在 C++20 之前由于没有对泛型的类型参数进行约束而产生的编译时问题。

```
template <typename It>
// template <std::random access iterator It> from C++20
void my sort(It first, It last) {
  std::sort(first, last);
int main() {
  std::list<int> a {3, 1, 4, 1, 5, 9}:
 my sort(a.begin(), a.end());
```

韩文弢

## 老版本的编译错误信息

```
309 行.
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 opera
    difference type n = last - first;
                          ~~~~~ ^ ~~~~~
.../usr/include/c++/v1/ algorithm/partial sort.h:35:12: note: in instanti
   . . .
wrap iter< Iter1> operator+(typename wrap iter< Iter1>::difference typename wrap iter<
11 errors generated.
```

13 / 67

# 新版本的编译错误信息

```
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 satisfi
void my_sort(It first, It last) {
my sort.cpp:6:16: note: because 'std:: list iterator<int, void *>' does n
template <std::random access iterator It>
1 error generated.
```

韩文弢

#### 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类型和函数。
- 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())
}
```

韩文弢

# 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。
- 目前还不能指定反向特型约束。
  - 例如,不能指定一种类型 T 不支持 Clone 特型。

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

# 特型约束的泛型结构化数据类型

- 同样可以定义具有特型约束的泛型结构体或枚举类型。
- 需要在定义结构体或枚举类型的代码段头部和 impl 代码段头部都声明泛型类型。
- 只有 impl 代码段的特型约束是必须指定的。
  - 可以对同一泛型类型实现不同特型约束的 impl 代码段。

## 特型约束的泛型类型示例

```
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) \Rightarrow 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) \Rightarrow t1.equals(t2),
         Err(e1), Err(e2) \Rightarrow e1.equals(e2),
         => false
```

#### 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> {}
```

韩文弢

# 默认方法的改写

- 实现特型时可以改写默认方法的实现。
- 但是一定要想好有充分的理由这样来做。
  - 例如,一定不要重新定义 ne,导致违反了 eq 和 ne 之间的逻辑关系。

韩文弢

## 特型的自动获得

- 一些特型实现起来比较直观,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(Eq, PartialEq, Debug)]
enum Result<T, E> {
    Ok(T),
    Err(E)
}
```

## 特型自动获得的限制

- 只能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。

韩文弢

# 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) { ... }
}
```

- ullet 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 (memcpy)。
  - 包含引用的类型不能实现 Copy 特型。
- 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。

韩文茂

# 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,需要满足如下重要性质:

```
k1 == k2 \rightarrow hash(k1) == 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。

韩文弢

#### 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。

```
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 <= b 或 b <= 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, &N) -> Vec<E>;
    // etc
}
```

- 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 { /*...*/ }
```

#### 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> { /*...*/ }
```

韩文弢

# 关联类型的应用

- 例如,在标准库中,迭代器特型 Iterator 具有表示项目类型的关联类型 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
    }
}
```

• 但是这不是一种好的实践方式,应尽量避免。

## 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。

韩又茂

# 特型的作用域规则示例: Display

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

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

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

## Drop

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

- 表示可销毁的特型(实际上是所有类型)。
- 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。

韩文茂

# Drop 的特点

- 一般情况下,不需要实现 Drop。
  - 默认的实现可以正常工作。
  - 也不需要用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。

韩文茂

#### Sized 和 ?Sized

- 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。

韩文弢

3

特型对象

和人汉

## 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) { /*...*/ }
}
```

#### 静态分发

- 可以通过**静态分发** (static dispatching) 的方式在满足约束 T: Foo 的任意类型上调用 bar 的不同版本。
- 代码编译时,编译器会给每个不同的 bar 生成对应的特化版本。
  - 对于实现 Foo 特型的每种类型,都会生成对应的函数。

```
fn blah(x: T) where T: Foo {
    x.bar()
fn main() {
    let s = "Foo".to_string();
    let u = 12;
    blah(s);
    blah(u):
```

韩文弢

# 动态分发

- Rust 也可以通过特型对象 (trait objects) 进行动态分发 (dynamic dispatching)。
- 特型对象要用像 Box<dyn Foo> 或 &dyn Foo 的形式来使用(相当于 C++ 中的对象 指针)。
- 背后的数据类型要实现 Foo 特型。
- 当使用动态分发时,特型背后的具体的数据类型被抹去,无法获得。

## 错误使用特型对象的示例

```
trait Foo { /*...*/ }
impl Foo for char { /*...*/ }
impl Foo for i32 \{ /* ... */ \}
fn use_foo(f: &dyn Foo) {
    // No way to figure out if we got a `char` or an `i32`
    // or anything else!
    match *f {
        // What type do we have? I dunno...
        // error: mismatched types: expected `Foo`, found ` `
        198 => println!("i32"),
        'c' => println!("char"),
        => println!("Something else..."),
```

## 特型对象的实现机制

- 使用特型对象时,只能在运行时进行方法的分发。
  - 编译器不知道特型引用背后的实际类型,类型信息已经抹去。
- 这样做会带来一定的运行时开销,但是在处理一些情况时会有用(例如动态大小的类型)。
  - 实际上特型对象只能通过指针的方式来使用,会增加指向方法的 vtable。

韩文茂

## 对象安全性

- 不是所有的特型都可以以特型对象的形式安全地使用。
- 例如, 创建 &dyn Clone 会引起编译错误, 因为 Clone 不是对象安全的。
- 特型是对象安全的,需要满足以下条件(不完整,具体见参考手册):
  - 所有超特型也都是对象安全的。
  - 不能以 Sized 为超特型。
  - 所有关联函数能够从特型对象进行分发:
    - 不带任何类型参数。
    - 方法除接收方 (receiver) 外,其他地方不能使用 Self 类型。
    - 接收方是引用或者某种指针形式的类型,如 &Self、&mut Self、Box<Self>等。
    - 没有 where Self: Sized 子句。

4

生命周期

## 生命周期的意义

- 考虑以下情况:
  - 甲方获取了一项资源。
  - ② 乙方通过引用借用了甲方的这项资源。
  - ③ 甲方对这项资源使用完毕,对它进行释放。
  - 乙方还保留着对这项资源的引用,并开始使用它。
  - ⑤ 乙方挂了……
- 之前提到过以上的场景在 Rust 中是不允许出现的。
- 需要有机制保证第 3 步不会发生在第 4 步之前。

#### 生命周期的显式表示

● 通常情况下,引用具有隐式的生命周期,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) {
    // ...
}
```

## 显式生命周期的含义

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

- 'a(注意是单引号那个撇,可以读作"撇 a"或者"生命周期 a")是一个**命名的**生命周期 参数。
  - <'a> 在泛型参数中声明生命周期参数。
  - &'a i32 类型是一个 i32 的引用,它的生命周期至少有 'a 那么长。

韩文弢

#### 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:

## 多个生命周期参数

```
fn borrow_x_or_y<'a>(x: &'a str, y: &'a str) -> &'a str;
```

- 在 borrow\_x\_or\_y 函数中,所有输入和输出的引用有着同样的生命周期。
  - ullet 也就是说,引用  ${f x}$  和  ${f y}$  的生命周期至少会和返回的引用的生命周期一样长。
  - ullet 换句话说,只要返回的引用还存在,那么引用  ${\bf x}$  和  ${\bf y}$  也必须还存在。

```
fn borrow_p<'a, 'b>(p: &'a str, q: &'b str) -> &'a str;
```

- 在 borrow\_p 函数中,返回的引用的生命周期和 p 一样长。
  - q 的生命周期是独立的,跟 p 没有关系。
  - ullet p 的借用至少要在返回引用存在期间持续。

#### 结构体相关的生命周期

• 结构体和结构体成员也可以具有生命周期参数。

```
struct Pizza(Vec<i32>):
struct PizzaSlice<'a> {
   pizza: &'a Pizza, // <- references in structs must
    index: u32.
               // ALWAYS have explicit lifetimes
let p1 = Pizza(vec![1, 2, 3, 4]);
   let s1 = PizzaSlice { pizza: &p1, index: 2 }; // this is okay
```

韩文弢

#### 结构体相关的生命周期(续)

```
struct Pizza(Vec<i32>):
struct PizzaSlice<'a> {
    pizza: &'a Pizza, // <- references in structs must
    index: u32,
               // ALWAYS have explicit lifetimes
let s2:
   let p2 = Pizza(vec![1, 2, 3, 4]);
    s2 = PizzaSlice { pizza: &p2, index: 2 };
   // no good - why?
drop(s2); // to undo NLL
```

## 生命周期之间的关系

• 如果结构体或枚举类型的成员是引用,则必须显式指定生命周期。

```
struct Foo<'a, 'b> {
   v: &'a Vec<i32>,
   s: &'b str,
}
```

- 可以指定生命周期之间的关系("活得比你久")。
  - 语法与泛型中的特型约束相同:<'b: 'a>('b 生命周期要涵盖 'a 生命周期)。

韩文茂

#### impl 代码段中的生命周期

- 接上面的例子,在实现 Foo 结构体的方法时也需要指定生命周期。
- 下面的代码可以视为 "用到生命周期 'a 和 'b 的 Foo 结构体的实现同样要用到生命周期 'a 和 'b"。

```
impl<'a, 'b> Foo<'a, 'b> {
  fn new(v: &'a Vec<i32>, s: &'b str) -> Foo<'a, 'b> {
    Foo {
      v: v,
      s: s,
    }
}
```

韩文弢

#### 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

# 本讲小结

- 泛型:同一份代码作用于不同的类型
- 特型: 提取类型之间的共性
- 特型对象: Rust 中动态分发的实现机制
- 生命周期: 比比谁活得久

下一讲:项目管理与常用库