Traits

Илья Шпильков

05-10-2023

Спецкурс ФТШ

Мотивация

- Не хочется писать повторяющийся код
- Абстрагирование от конкретных типов в сторону их свойств
- Гарантии системы типов

Generics

Обобщенные коллекции

```
enum Option<T> {
  Some(T),
  None
}
```

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

`Option` может хранить произвольное `T`, либо ничего.

Обобщенные коллекции

```
enum Option<T> {
    Some(T),
    None
}
```

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

Параметр типа указывается после имени в угловых скобках.

Обобщенные коллекции

```
1 enum Option<T> {
2     Some(T),
3     None
4 }
```

```
1 struct Point<T> {
2     x: T,
3     y: T,
4 }
```

Внутри определения типа можно использовать параметр типа.

Другие примеры

- `Result<T, E>` - с хранением значение `T` или информации об ошибке `E`.
- `Vec<T>` - с хранением произвольного `T`.
- `HashMap<K, V>` - где `K` можно ... .
- `BTreeMap<K, V>` - где `K` можно ... .

Другие примеры

- `Result<T, E>` - с хранением значение `T` или информации об ошибке `E`.
- `Vec<T>` - с хранением произвольного `T`.
- `HashMap<K, V>` - где `K` можно хешировать.
- `BTreeMap<K, V>` - где `K` можно сортировать.

Аналогично хочется не реализовывать одинаковые функции, а абстрагироваться от свойств типа.

```
1 fn max_i32(lhs: i32, rhs: i32) → i32 {
2   if lhs > rhs { lhs } else { rhs }
3 }
4
5 fn max_u32(lhs: u32, rhs: u32) → u32 {
6   if lhs > rhs { lhs } else { rhs }
7 }
8
9 ...
```

Generic функции

Возникает желание написать функцию с следующей сигнатурой:

```
1 fn max<T>(lhs: T, rhs: T) → T {
2   if lhs > rhs { lhs } else { rhs }
3 }
```

Но компилятор не скомпилирует такой код с ошибкой:

```
  error[E0369]: binary operation `>` cannot be applied to type `T`
 --> src/lib.rs:2:12
  |
2 |     if lhs > rhs { lhs } else { rhs }
  |        --- ^ --- T
  |        |
  |        T
  |
help: consider restricting type parameter `T`
  |
1 | fn max<T: std::cmp::PartialOrd>(lhs: T, rhs: T) -> T {
  |           ++++++++++++++++++++++
```

Trait-ы

Хотим уметь требовать от типа наличия некоторой функциональности.

```
1 trait Calc {
2   fn calc(&self, x: f64) → f64;
3 }
```

Чтобы показать, что тип обладает функциональностью, нужно в явном виде написать, что он реализует заданный трейт

```
1  struct Parabola {
2    a: f64,
3    b: f64,
4    c: f64,
5  }
6
7  impl Calc for Parabola {
8    fn calc(&self, x: f64) → f64 {
9        self.a * x.powf(2.0) + self.b * x + self.c
10   }
11 }
```

Дальше трейты могут использоваться в качестве ограничений при написании обобщенных функций

```
1 fn derivative<F: Calc>(f: &F, x: f64) → f64 {
2   const EPS: f64 = 0.0001;
3
4   (f.calc(x + EPS) - f.calc(x)) / EPS
5 }
```

В таком виде `trait`-ы не сильно отличаются от `interface` из java/C#, но на самом деле они предоставляют другой набор возможностей.

Трейты из стандартной библиотеки

Default

```rust
pub trait Default {
    // Required method
    fn default() -> Self;
}
```

- функция внутри `trait`-а не обязана иметь в сигнатуре `self`, `&self` или `&mut self`.
- возвращаемый значение должно быть того же типа, что и тип, который реализует этот `trait`
- внутри определения `trait`-а можно ссылаться на тип, который и будет реализовывать этот `trait`.
- автоматическая реализация через `#[derive(Default)]` над определением типа.

Return type polymorhism

В примере ниже компилятор автоматически выводит полный тип `nums` исходя из его использования.

```
1 let mut nums: Vec<_> = Default::default();
2 nums.push(1);
3 nums.push(2);
```

Clone

```rust
pub trait Clone {
    // Required method
    fn clone(&self) -> Self;

    // Provided method
    fn clone_from(&mut self, source: &Self) { ... }
}
```

- `trait` может иметь стандартную реализацию метода внутри него, но её можно перегрузить.
- `derive`

Copy

Почти дословное определение из стандартной библиотеки:

```
#[lang = "copy"]
pub trait Copy: Clone {
    // Empty.
}
```

- Не имеет собственных методов (такие `trait`-ы называют маркерными)
- Требует, чтобы тип также реализовывал `trait Clone`
- Компилятор знает про этот `trait`, поэтому позволяет использовать перемеенные из которых сделали `move`.
- Компилятор проверяет, что все поля или все альтернативы тоже реализуют `trait Copy`

PartialEq

```rust
pub trait PartialEq<Rhs = Self> {
    fn eq(&self, other: &Rhs) → bool;
    fn ne(&self, other: &Rhs) → bool { ... }
}
```

- Трейты могут иметь собственные generic параметры.
- `Rhs = Self` выставляет значение по умолчанию.
- `PartialEq<A>` и `PartialEq<B>` – разные трейты, поэтому их можно реализовывать по отдельности.
- Способ перегрузить оператор `==` у типа.
- `derive`

Eq

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

Маркерный `trait` с контрактом, что `==` – отношение эквивалентности:

Eq

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

Маркерный `trait` с контрактом, что `==` – отношение эквивалентности:

- Рефлексивность: `a == a`
- Симметричность: `a == b` $\iff$ `b == a`
- Транзитивность: `a == b` и `b == c` $\Rightarrow$ `a == c`

PartialOrd

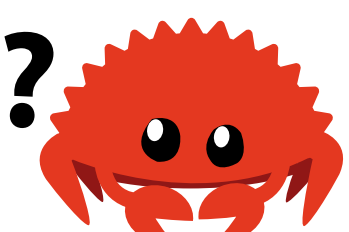
```rust
pub trait PartialOrd<Rhs = Self>: PartialEq<Rhs> {
    fn partial_cmp(&self, other: &Rhs) → Option<Ordering>;
    fn lt(&self, other: &Rhs) → bool { ... }
    fn le(&self, other: &Rhs) → bool { ... }
    fn gt(&self, other: &Rhs) → bool { ... }
    fn ge(&self, other: &Rhs) → bool { ... }
}
```

- Способ перегрузить операторы `<`, `>`, `<=`, `>=`.

Контракты:

- Консистентность с `==`: `a == b` ⇔ `partial_cmp(a, b) == Some(Equal)`.
- Дуальность: `a < b` ⇔ `b > a`
- Частичный порядок

Условия на частичный нестрогий порядок:



Условия на частичный нестрогий порядок:

- Рефлексивность: $a \leq a$
- Антисимметричность $a \leq b$ и $b \leq a \Rightarrow a = b$
- Транзитивность: $a \leq b$ и $b \leq c \Rightarrow a \leq c$

Ord

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

- Реализация должна быть консистентна с `PartialOrd`
- Используется в сортировках, и упорядоченых контейнерах (`BTreeMap`)
- `f32`, `f64` – **не** `Ord`
- Образует линейный порядок

std::ops

В модуле `std::ops` содержится полный список операторов для перегрузки:

- `Add`, `AddAssign`, `Sub`, `SubAssign`
- `BitAnd`, `BitAndAssign`, `BitOr`, `BitOrAssign`
- ...

Hash

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

pub trait Hasher {
    fn finish(&self) -> u64;
    fn write(&mut self, bytes: &[u8]);
    fn write_*(&mut self, i: *) {  ...  }
}
```

- Внутри `trait`-ов могут находиться другие generic функции
- Частый паттерн разделения возможности сделать некоторую операцию (`Hash`) и конкретного алгоритма (`Hasher`).

From/Into

```
1 pub trait From<T> {
2   fn from(value: T) → Self;
3 }
4
5 pub trait Into<T> {
6   fn into(self) → T;
7 }
```

Blanket Implementation

Можно реализовать `trait` не для конкретного типа, а вообще для любого.

```
impl<T> From<T> for T {
  fn from(value: T) → Self {
    value
  }
}

impl<T, U: From<T>> Into<U> for T {
  fn into(value: T) → U {
    U::from(value)
  }
}
```

- Для **любых** типов `T` и `U`, если реализован для `U` реализован `From<T>`, то реализован `Into<U>` для `T`.
- По причине выше, нужно реализовывать только `From`

TryFrom/TryInto

```rust
pub trait TryFrom<T> {
  type Error;

  fn try_from(value: T) -> Result<Self, Self::Error>;
}

pub trait TryInto<T> {
  type Error;

  fn try_into(self) -> Result<T, Self::Error>;
}
```

- Трейты могут содержать ассоциированные типы. В данном случае имплиментатор, должен указать тип ошибки, который возвращает `try_from` в случае неудачи.
- Аналогично `From`/`Into`, `TryInto` автоматически реализуется из `TryFrom`.

Deref/DerefMut

```rust
pub trait Deref {
  type Target;
  fn deref(&self) → &Self::Target;
}

pub trait DerefMut: Deref {
  fn deref_mut(&mut self) → &mut Self::Target;
}
```

- Перегрузка оператора *
- Компилятор автоматически расставляет `&` если знает, метода нет на `self`, но есть на `&self`
- Компилятор автоматически расставляет `deref`, если метод есть на `Target`

Index/IndexMut

```rust
pub trait Index<Idx>
{
  type Output;
  fn index(&self, index: Idx) → &Self::Output;
}

pub trait IndexMut<Idx>: Index<Idx> {
  fn index_mut(&mut self, index: Idx) → &mut Self::Output;
}
```

- Способ перегрузить оператор `container[index]`.
- Так как `Idx` – параметр, то разная индексация на `[T]` работает засчет, того есть несколько типов `Range`-ей

Drop

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

- Реализация RAII
- Компилятор самостоятельно вызывает эту функцию в конце блока, в котором живет объект
- Нельзя реализовывать вместе с `Copy`
- Нельзя позвать руками, но можно использовать `std::mem::drop`, которая принимает аргумент **по значению**
- Принимает себя по ссылке, а не по значению

Обход ограничения на `&mut self` внутри `drop`

- Поле, которое нужно в `drop` по значению можно завернуть в `Option`
- Внутри `drop` владение над объектом можно забрать через `Option::take`

```
impl<T> Option<T> {
    fn take(&mut self) → Option<T> { ... }
}
```

Не надо писать руками то, что можно сгенерировать автоматически

```
#[derive(
  Clone, PartialEq, Eq, PartialOrd, Ord,
  Hash, Debug, Serialize, Deserialize
)]
struct User {
  name: String,
  surname: String,
  age: u32,
}
```

Full-Quallified syntax

```
1 trait Trait {
2   fn foo(&self);
3 }
```

Следующие вызовы делают одно и тоже:

```
1 let x: Type = ... ;
2 x.foo();
3 Trait::foo(&x);
4 <Type as Trait>::foo(&x);
```

where clauses

Перед фигурными скобками можно указать `where` и набор ограничений на типы:

```
1 struct Point<T> where T: Copy { ... }
2
3 trait Copy where Self: Clone { ... }
4
5 impl<T, U> Into<U> for T where U: From<T> { ... }
6
7 fn default_pair<T>() → (T, T) where T: Default { ... }
```

Const generics

- Константы могут выступать параметрами.
- Внутри трейта могут находиться ассоциированные константы

```
struct ArrayPair<T, const N: usize> {
  fst: [T; N],
  snd: [T; N],
}
```

```
trait Named {
  const NAME: &'static str;
}
```

Ограничения

```rust
trait AllVariants {
  const N: usize;

  fn all_variants() -> [Self; Self::N];
}
```

- Ассоцированные константы **не** могут участвовать в сигнатурах

GATs

```
1 pub trait PointerFamily {
2   type Ptr<T>: Deref<Target = T>;
3 }
```

- Ассоциированные типы тоже могут быть параметризованны generic-ами
- На ассоциированные типы можно накладывать ограничения

Impl on forgein types

`impl` блоки отделены от блоков с определением типа. Это помогает делать реализации трейта для произвольного `T`. В частности, можно делать реализации трейта для типа из других модулей.

```
impl Calc for i32 {
  fn calc(&self, _: f64) → f64 {
      *self as f64
  }
}
```

И позже использовать: `1.calc(0.)`

Orphan rules

Эта возможность приводит к тому, что в двух разных крейтах могут определить один и тот же `trait` для того же типа.

Непонятно, что с этим делать, поэтому запретим подобные ситуации

Правило: `impl Trait for Type` допустимо, если выполнено хотя бы одно условие:

- `Trait` определен локально
- `Type` определен локально

Полное правило можно почитать тут

Мономорфизация

Как устроено внутри?

- `trait` - свойство, известное на этапе компиляции
- в конечном счете будет запускаться `main`, в котором все функции вызываются с известными на момент компиляции типами
- для каждой generic функции компилятор может сгенерировать необходимое количество экзмеляров функции, где вместо generic параметров будут конкретные типы.

Эффективность

Так как generic функции раскрываются в пачку непараметризованных, то к ним можно применять все оптимизации, которые применяются к обычным функциям.

Одна из самых важных оптимизаций – inline'инг:

- компилятор подставляет на место вызова функции ее код
- экономия на вызове функции (на самом деле не так важно)
- контексные оптимизации

А минусы будут?

- Большой размер бинарника
- Невозможность динамической линковки

Гарантии системы типов

Сколько существует реализаций функции с такой сигнатурой?

```
1 fn foo<T>(x: &[T]) → T { ... }
```

Гарантии системы типов

Сколько существует реализаций функции с такой сигнатурой?

```
fn foo<T>(x: &[T]) → T { ... }
```

На самом деле – не одной разумной: в качестве аргумента принимается `&[T]`, а возвращается `T`, но на тип `T` нет ограничений (в первую очередь — на `Clone`), и получить владеющее значение из ссылки *нельзя*.

Гарантии системы типов

Сколько существует реализаций функции с такой сигнатурой?

```
1 fn foo<T>(x: &[T]) → T { ... }
```

На самом деле – не одной разумной: в качестве аргумента принимается `&[T]`, а возвращается `T`, но на тип `T` нет ограничений (в первую очередь — на `Clone`), и получить владеющее значение из ссылки *нельзя*.

Это позволяет рассуждать о поведении функции, зная только ее сигнатуру.