1_Traits_en

May 2, 2023

1 Traits

A trait tells the Rust compiler about functionality a particular type has and can share with other types. We can use traits to define shared behavior in an abstract way. We can use trait bounds to specify that a generic can be any type that has certain behavior.

A type's behavior consists of the methods we can call on that type. Different types share the same behavior if we can call the same methods on all of those types. Trait definitions are a way to group method signatures together to define a set of behaviors necessary to accomplish some purpose.

[ ]:
```
trait Summary {
    fn summarize(&self) -> String;
}
```

A trait can have multiple methods in its body: the method signatures are listed one per line and each line ends in a semicolon.

1.1 Implementing a trait on a type

[ ]:
```
fn main() {
    trait Summary {
        fn summarize(&self) -> String;
    }

    struct NewsArticle {
        headline: String,
        location: String,
        author: String,
        content: String,
    }

    impl Summary for NewsArticle {
        fn summarize(&self) -> String {
            format!("{}, by {} ({})", self.headline, self.author, self.location)
        }
    }

    struct Tweet {
        username: String,
        content: String,
        reply: bool,
        retweet: bool,
    }

    impl Summary for Tweet {
        fn summarize(&self) -> String {
            format!("{}: {}", self.username, self.content)
        }
    }

    let tweet = Tweet {
        username: String::from("horse_ebooks"),
        content: String::from("of course, as you probably already know,␣
↪people"),
        reply: false,
        retweet: false,
    };

    println!("1 new tweet: {}", tweet.summarize());
}
main();
```

One restriction to note with trait implementations is that we can implement a trait on a type only if either the trait or the type is local to our crate (package). This restriction is part of a property of programs called coherence, and more specifically the orphan rule, so named because the parent type is not present. This rule ensures that other people's code can't break your code and vice versa. Without the rule, two crates could implement the same trait for the same type, and Rust wouldn't know which implementation to use.

1.2 Default implementations

Sometimes it's useful to have default behavior for some or all of the methods in a trait instead of requiring implementations for all methods on every type. Then, as we implement the trait on a particular type, we can keep or override each method's default behavior.

[ ]:
```
fn main() {
    trait Summary {
        fn summarize_author(&self) -> String;

        fn summarize(&self) -> String {
            // Default implementations can call other methods in the same␣
↪trait,
            // even if those other methods don't have a default implementation.
            format!("(Read more from {}...)", self.summarize_author())
        }
    }

    struct Tweet {
        username: String,
        content: String,
        reply: bool,
        retweet: bool,
    }

    // we only need to define summarize_author when we implement the trait on a␣
 ↪type
    impl Summary for Tweet {
        fn summarize_author(&self) -> String {
            format!("@{}", self.username)
        }
    }

    let tweet = Tweet {
        username: String::from("horse_ebooks"),
        content: String::from("of course, as you probably already know,␣
 ↪people"),
        reply: false,
        retweet: false,
    };

    println!("1 new tweet: {}", tweet.summarize());
}
main();
```

1.3 Trait Bound Syntax

Rust allows type definitions to have type parameters. In fact we already have seen some of these. The `Option<T>` and `Result<T,E>` types are good examples. These two types do not place constraints on their parameters but it is possible to do so :

[ ]:
```
fn main() {
    trait Summary {
        fn summarize(&self) -> String;
    }

    struct Tweet {
        username: String,
        content: String,
        reply: bool,
        retweet: bool,
    }
    impl Summary for Tweet {
        fn summarize(&self) -> String {
            format!("{}: {}", self.username, self.content)
        }
    }

    struct Content<T: Summary> { // The type argument T must implement Summary
        inner: T
    }

    impl <T: Summary> Content<T> { // same constraint for this implementation
        fn print_summary(&self) {
            println!("{}", self.inner.summarize());
        }
    }

    let my_content = Content {
        inner: Tweet {
            username: String::from("horse_ebooks"),
            content: String::from("of course, as you probably already know,␣
↪people"),
            reply: false,
            retweet: false,
        }
    };
    my_content.print_summary();
}
main();
```

Trait bounds can also be placed on functions :

[ ]:
```
fn notify<T: Summary>(item: T) {
    println!("Breaking news! {}", item.summarize());
}

fn notify2(item: impl Summary) { // <- syntactic sugar for generics
    println!("Breaking news! {}", item.summarize());
}
```

Multiple trait bounds can be specified with the + syntax

[ ]:
```
fn notify<T: Summary + Copy>(item: T) {
    println!("Breaking news! {}", item.summarize());
}
```

In the examples above, the `notify` functions take ownership of their `item` parameters. We could rewrite them to only take references :

[ ]:
```
fn notify<T: Summary>(item: &T) {
    println!("Breaking news! {}", item.summarize());
}
```

Rust provides handy short syntexes for simple cases like these :

[ ]:
```
fn notify2(item: &dyn Summary) { // dyn keyword
    println!("Breaking news! {}", item.summarize());
}

fn main() {
    let tweet = Tweet {
        username: String::from("horse_ebooks"),
        content: String::from("of course, as you probably already know,␣
 ↪people"),
        reply: false,
        retweet: false,
    };
    notify1(&tweet);
    notify2(&tweet);
}
main();
```

We can also use the `impl Trait` syntax in the return position to return a value of some type that implements a trait, as shown here:

[ ]:
```
fn returns_summarizable() -> impl Summary {
    Tweet {
        username: String::from("horse_ebooks"),
        content: String::from("of course, as you probably already know,␣
 ↪people"),
        reply: false,
        retweet: false,
    }
}
```

By using `impl Summary` for the return type, we specify that the `returns_summarizable` function returns some type that implements the `Summary` trait without naming the concrete type.

The ability to return a type that is only specified by the trait it implements is especially useful in the context of closures and iterators. Closures and iterators create types that only the compiler knows or types that are very long to specify.

However, you can only use `impl Trait` if you're returning a single type. For example, in the previous code, you could not return either a `NewsArticle` or a `Tweet`.

When the trait bounds become complex, they can be put in a `where` clause in the function's signature :

[ ]:
```rust
use std::fmt::Display;
use std::fmt::Debug;

fn some_function<T, U>(t: T, u: U) -> i32
    where T: Display + Clone,  // <-- where clause
          U: Clone + Debug
{ 0 }
```

1.4 Using trait bounds to conditionally implement methods

By using a trait bound with an `impl` block that uses generic type parameters, we can implement methods conditionally for types that implement the specified traits.

[ ]:
```rust
use std::fmt::Display;

struct Pair<T> {
    x: T,
    y: T,
}

impl<T> Pair<T> { // <-- no constraint on T
    fn new(x: T, y: T) -> Self {
        Self {
            x,
            y,
        }
    }
}

impl<T: Display + PartialOrd> Pair<T> { // <-- methods in this impl are only␣
 ↪defined
                                        // <-- if T implements Display and␣
 ↪PartialOrd
    fn cmp_display(&self) {
        if self.x >= self.y {
            println!("The largest member is x = {}", self.x);
        } else {
            println!("The largest member is y = {}", self.y);
        }
    }
}
```

1.5 Standard traits

We can also conditionally implement a trait for any type that implements another trait. For example, the standard library implements the ToString trait on any type that implements the Display trait. The impl block in the standard library looks similar to this code:

```
impl<T: Display> ToString for T {
    // --snip--
}
```

Therefore, we can call the `to_string` method defined by the `ToString` trait on any type that implements the `Display` trait. For example, we can turn integers into their corresponding `String` values like this because integers implement `Display`:

[ ]:
```
let s = 3.to_string();
```

1.6 Operator overloading

In Rust, many of the operators can be overloaded via traits. That is, some operators can be used to accomplish different tasks based on their input arguments. This is possible because operators are syntactic sugar for method calls. For example, the `+` operator in `a + b` calls the `add` method (as in `a.add(b)`). This `add` method is part of the `Add` trait. Hence, the `+` operator can be used by any implementor of the `Add` trait.

[ ]:
```
use std::ops;

#[derive(Debug)]
struct Point {
    x: i32,
    y: i32,
}

impl ops::Add<Point> for Point {
    type Output = Point;

    fn add(self, p: Point) -> Point {
        Point{
            x: self.x + p.x,
            y: self.y + p.y,
        }
    }
}

fn main() {
    let a = Point{x: 10, y: 10};
    let b = Point{x: 20, y: 20};
    println!("Translated point = {:#?}", a + b);
}
main();
```

2 Exercises

Exercise 1: Create two structures *Rectangle* and *Triangle*, having each a length and a height of type float. Write the code that calculates the area for each structure. Decide whether to use traits/

generics or not.

[ ]:
```
fn main() {

}
main();
```

Solution 1:

[ ]:
```
struct Rectangle {
    length: f64,
    height: f64
}

struct Triangle {
    length: f64,
    height: f64
}

trait HasArea {
    fn area(&self) -> f64;
}

impl HasArea for Rectangle {
    fn area(&self) -> f64 { self.length * self.height }
}

impl HasArea for Triangle {
    fn area(&self) -> f64 { self.length * self.height / 2.0}
}

fn print_shape_area_static<T: HasArea>(shape: &T) {
    println!("area = {}", shape.area());
}

fn print_shape_area_dynamic(shape: &dyn HasArea) {
    println!("area = {}", shape.area());
}

fn main() {
    let rectangle = Rectangle { length: 5.0, height: 4.0 };
    let triangle = Triangle  { length: 3.0, height: 4.0 };
    print_shape_area_static(&rectangle);
    print_shape_area_static(&triangle);
    print_shape_area_dynamic(&rectangle);
    print_shape_area_dynamic(&triangle);
}
main();
```

Exercice 2: Simplify the code below to use a generic approach

[ ]:
```
fn largest_i32(list: &[i32]) -> i32 {  // find the largest i32 in a slice
    let mut largest = list[0];

    for &item in list.iter() {
        if item > largest {
            largest = item;
        }
    }

    largest
}

fn largest_char(list: &[char]) -> char { // find the largest char in a slice
    let mut largest = list[0];

    for &item in list.iter() {
        if item > largest {
            largest = item;
        }
    }

    largest
}

fn main() {
    let number_list = vec![34, 50, 25, 100, 65];

    let result = largest_i32(&number_list);
    println!("The largest number is {}", result);

    let char_list = vec!['y', 'm', 'a', 'q'];

    let result = largest_char(&char_list);
    println!("The largest char is {}", result);
}
main();
```

Tip: : Use the trait `std::cmp::Ord` and `std::marker::Copy`.

Solution :

[ ]:
```
use std::cmp::Ord;

fn largest<T: Copy + Ord>(list: &[T]) -> T {
    let mut largest = list[0];
```

```rust
    for &item in list.iter() {
        if item > largest {
            largest = item;
        }
    }

    largest
}

fn largest_idiomatic<T: Copy + Ord + Default>(list: &[T]) -> T {
    list.iter().copied().max().unwrap_or_default()
}

fn main() {
    let number_list = vec![34, 50, 25, 100, 65];

    let result = largest_idiomatic(&number_list);
    println!("The largest number is {}", result);

    let char_list = vec!['y', 'm', 'a', 'q'];

    let result = largest_idiomatic(&char_list);
    println!("The largest char is {}", result);
}
main();
```