A cheatsheet for some potentially confusing terms in Rust

Published on March 09, 2023

Learning Rust can be a bit tedious. There are various reasons for that. My colleague - Tanks - has started a blog series on this topic: Why is it hard to learn another programming language Part 1. One aspect is a combination of the opportunistic learning strategies and ambiguous or overloaded terms. In this blog post, we will look at some of such term pairs that some seasoned Rust programmers have to revisit time and again.

This article is only a short overview, will not dive into the deep end here.

⚠ Assumptions: We assume that you know what Traits are in Rust parlance. If you need a refresher, please read Traits: Defining Shared Behavior in Rust book.

1.0

`Copy` vs `Clone`

These are needed if you need to duplicate an object you have. You know, sometimes you need to *copy* & *clone* yourself out of the ownership problems, to satisfy the *borrow checker*, and just get the thing to compile. I like to think of `Copy` as `ImplicitCopy` and `Clone` as `ExplicitCopy`.

Copy

- performs bit-by-bit duplication
- behaviour is not overloadable
- is a Marker Trait

```rust
#[derive(Debug, Copy, Clone)]
struct API;

let x = API;
let y = x; // `y` is a copy of `x`
println!("{x:?}"); // OK!
```

Clone

- overloadable behaviour
- explicit duplication of an object

```rust
#[derive(Debug)]
struct API;

// explicitly implementing instead of merely deriving
// just to show
impl Clone for API {
    fn clone(&self) -> API {
        API
    }
}

let x = API;
let y = x.clone();
```

2.0

`Debug` vs `Display`

These Traits are part of the `std::fmt` module where, `fmt` stands for "format" (`fmt`'s pronunciation left for the reader as an exercise). These traits allow us to print our custom Types (structs) without writing finicky and repetitive boilerplate.

Debug

- all types can derive it
- provides more detaled *debug* output of your object
- enables the colon-question-mark specifier `{:?}`

```rust
#[derive(Debug)]
struct API;

let x = API;
println!("{x:?}");
```

Display

- no easy derive available
- have to manually write the implementation
- user facing output
- needs the empty curlies specifier `{}`

```rust
use core::fmt;
use std::fmt::Display;
```

```
struct API {
    a: i32
}

impl Display for API {
    fn fmt(&self, f: &mut fmt::Formatter) -> fmt::Result
{
        write!(f, "API.a: {}", self.a)
    }
}

let x = API { a: 1 };
println!("{x}"); // prints "API.a: 1"
```

3.0

ToString vs Display vs ToOwned

The borders of stringification and displayification and generalization continue to be muddied!

ToString

- Trait to convert value to a `String`
- automatically implemented for any Type that implements `Display`
- makes `.to_string()` available for easy stringification

```
struct API {
    a: i32
}

impl ToString for API {
    fn to_string(&self) -> String {
        format!("API.a: {}", self.a)
    }
}

let x = API {a: 1};
let y = x.to_string();
println!("{y}"); // prints "API.a: 1"
```

Display

- implementing this Trait will automatically implement `ToString`, aka "super"

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

```
struct API {
    a: i32
}

impl Display for API {
    fn fmt(&self, f: &mut fmt::Formatter) -> fmt::Result
{
        write!(f, "API.a: {}", self.a)
    }
}

let x = API { a: 1 };
let y = x.to_string();
println!("{y}"); // prints "API.a: 1"
```

ToOwned

- a generalized way to convert a borrowed Type to an owned Type
- the final Type could be a different owned Type

```
// how `str` gets converted into `String` when you run
// `to_owned()` on it. yes, we can just use
`to_string()`
// but this is just an example

// ... snip
impl ToOwned for str {
    type Owned = String;
    #[inline]
    fn to_owned(&self) -> String {
        unsafe {
String::from_utf8_unchecked(self.as_bytes().to_owned())
}
    }
    // ... snip
}
```

4.0

AsRef vs Borrow

A simple quote from an earlier version of The Rust Book –

> Choose Borrow when you want to abstract over different kinds of borrowing, or when you're building a data structure that treats owned and borrowed values in equivalent ways, such as hashing and comparison.
>
> Choose AsRef when you want to convert something to a reference directly, and you're writing generic code.

AsRef

- use for explicit conversion of a value to a reference given an owned Type
- most useful with "generics"

```
use std::path::Path;

struct APIName {
    inner: String
}

impl AsRef<Path> for APIName {
    fn as_ref(&self) -> &Path {
        &Path::new(&self.inner)
    }
}

// a normal function to show that you can treat the
// given object as a Path reference
fn takes_asref_path<T>(path_like: T) where T:
AsRef<Path> {
    let path: &Path = path_like.as_ref();
    println!("the path is {}", path.display());
}

let api_name = APIName { inner: "cool.dat".to_string()
};
// prints "the path is cool.dat"
takes_asref_path(api_name);
// prints "the path is cool2.dat"
takes_asref_path("cool2.dat");
// prints "the path is cool3.dat"
takes_asref_path(PathBuf::from("cool3.dat"));
```

Borrow

- dealing with borrowing but more of "borrowed-as" semantics
- can work with either reference or owned type
- has the potential to change the representation of a Type
- e.g. `Box<T>` can be borrowed-as T

```
use std::borrow::Borrow;

struct APIName {
    inner: String
}

// make it so that APIName can be
// borrowed as str slice
impl<'a> Borrow<str> for APIName {
    fn borrow(&self) -> &str {
        &self.inner
    }
}

let api_name = APIName { inner: "my awesome
api".to_string() };
let api_name_borrowed: &str = api_name.borrow();
println!("{}", api_name_borrowed); // prints "my awesome
api"
```

You will likely not implement `Borrow` as much as you will see this being used in ergonomic APIs. For example, you will find that in the `HashMap` API, we want our keys to work with both owned (say, `String`) and borrowed (say `str`) Types. This means that for a `HashMap<String, MyData>`, you will be able to call `HashMap`'s `get()` method as `map.get("hello")`, i.e. with a borrowed `str` argument. See the examples in the `HashMap` documentation for more information.

5.0

`Send` vs `Sync`

Both of these Traits are dealing with the multithreading aspects of programming. Rust wants us to be safe. A one of the firsts of its kind of safety since the advent of multithreaded computing as Jacquard loom! (sorry)

Send

- a Marker Trait, signifies a Type is safe to send to another thread

```
struct API;

unsafe impl Send for API {}
```

Sync

- a Marker Trait, signifies a Type is safe to share between threads
- as nomicon notes, T is `Sync` if and only if `&T` is `Send`, that is you have to be able to safely send the reference of a Type across threads for T to be Sync.

```
struct API;

unsafe impl Sync for API {}
```

6.0

`Sized` vs `?Sized`

Rust likes to know the size of Types at compile-time, as much as possible. This is where the `Sized` bound comes in.

Sized

- a Marker Trait, signifies that the size is known at the compile time
- this is tagged on to all Type parameters, meaning you need to opt-out by using `?` (see below)

?Sized

- same Marker Trait as above, but `?` is used to remove the constraint of knowing size at compile-time

```rust
// use the "unsizing coercion" to create a
// Dynamically Sized Type (DST)
// by adding a generic with bound of `?Sized`
struct Packet<T: ?Sized> {
    a: i32,
    b: T,
}

let sized_packet: Packet<[u8; 10]> = Packet { a: 23, b:
[21; 10] };
let dst_packet: &Packet<[u8]> = &sized_packet;
println!("{}  {:?}", dst_packet.a, &dst_packet.b);
// prints "23  [21, 21, 21, 21, 21, 21, 21, 21, 21, 21]"
```

For more details, see Exotic Sizes in the Nomicon.

7.0

`PartialEq` vs `Eq`

We can see that these Traits seem to deal with equality but what on good Earth is "partial" equality for?

PartialEq

- more relaxed than `Eq` (see below), for Types that do not have full equivalence relation
- because… floating point numbers, eek!
- apparently, there is this whole mathematical thing called partial equivalence relation

```rust
struct Point {
    x: u32,
    y: u32,
}
```

```
// we need the condition for `API == API`
impl PartialEq for Point {
    fn eq(&self, other: &Self) -> bool {
        self.x == other.x && self.y == other.y
    }
}
```

Eq

- not only just `a == b` and `a != b` but also the following must hold
  - reflexive, `a == a`
  - symmetric, `a == b => b == a`
  - transitive, `a == b && b == c => a == c`

For code example, see `PartialEq`.

8.0

Conclusion

This is not an exhaustive list of pairs but we hope this is a decent start. Jargon is one of the main barriers to entry in a field or subject. We hope this byte-sized introduction to these terms will help you.

Thanks to Tim McNamara for starting this discussion on Twitter.