Rust's as_ref vs as_deref

By Michael Snoyman, July 5, 2021

What's wrong with this program?

```rust
fn main() {
    let option_name: Option<String> = Some("Alice".to_owned());
    match option_name {
        Some(name) => println!("Name is {}", name),
        None => println!("No name provided"),
    }
    println!("{:?}", option_name);
}
```

The compiler gives us a wonderful error message, including a hint on how to fix it:

```
error[E0382]: borrow of partially moved value: `option_name`
 --> src\main.rs:7:22
  |
4 |         Some(name) => println!("Name is {}", name),
  |              ---- value partially moved here
...
7 |     println!("{:?}", option_name);
  |                      ^^^^^^^^^^^ value borrowed here after partial move
  |
  = note: partial move occurs because value has type `String`, which does not implement the `Copy`
trait
help: borrow this field in the pattern to avoid moving `option_name.0`
  |
4 |         Some(ref name) => println!("Name is {}", name),
  |              ^^^
```

The issue here is that our pattern match on `option_name` moves the `Option<String>` value into the match. We can then no longer use `option_name` after the `match`. But this is disappointing, because our usage of `option_name` and `name` inside the pattern match doesn't actually require moving the value at all! Instead, borrowing would be just fine.

And that's exactly what the `note` from the compiler says. We can use the `ref` keyword in the identifier pattern to change this behavior and, instead of *moving* the value, we'll borrow a reference to the value. Now we're free to reuse `option_name` after the `match`. That version of the code looks like:

```rust
fn main() {
    let option_name: Option<String> = Some("Alice".to_owned());
    match option_name {
        Some(ref name) => println!("Name is {}", name),
        None => println!("No name provided"),
    }
    println!("{:?}", option_name);
}
```

For the curious, you can read more about the `ref` keyword.

More idiomatic

While this is *working* code, in my opinion and experience, it's not idiomatic. It's far more common to put the borrow on `option_name`, like so:

```rust
fn main() {
    let option_name: Option<String> = Some("Alice".to_owned());
    match &option_name {
        Some(name) => println!("Name is {}", name),
        None => println!("No name provided"),
    }
    println!("{:?}", option_name);
}
```

I like this version more, since it's blatantly obvious that we have no intention of moving `option_name` in the pattern match. Now `name` still remains as a reference, `println!` can use it as a reference, and everything is fine.

The fact that this code works, however, is a specifically added feature of the language. Before RFC 2005 "match ergonomics" landed in 2016, the code above would have failed. That's because we tried to match the `Some` constructor against a *reference* to an `Option`, and those types don't match up. To borrow the RFC's terminology, getting that code to work would require "a bit of a dance":

```rust
fn main() {
    let option_name: Option<String> = Some("Alice".to_owned());
    match &option_name {
        &Some(ref name) => println!("Name is {}", name),
        &None => println!("No name provided"),
    }
    println!("{:?}", option_name);
}
```

Now all of the types really line up explicitly:

- We have an `&Option<String>`
- We can therefore match on a `&Some` variant or a `&None` variant
- In the `&Some` variant, we need to make sure we borrow the inner value, so we add a `ref` keyword

Fortunately, with RFC 2005 in place, this extra noise isn't needed, and we can simplify our pattern match as above. The Rust language is better for this change, and the masses can rejoice.

Introducing as_ref

But what if we didn't have RFC 2005? Would we be required to use the awkward syntax above forever? Thanks to a helper method, no. The problem in our code is that `&option_name` is a reference to an `Option<String>`. And we want to pattern match on the `Some` and `None` constructors, and capture a `&String` instead of a `String` (avoiding the move). RFC 2005 implements that as a direct language feature. But there's also a method on `Option` that does just this: `as_ref`.

```rust
impl<T> Option<T> {
    pub const fn as_ref(&self) -> Option<&T> {
        match *self {
            Some(ref x) => Some(x),
            None => None,
        }
    }
}
```

This is another way of avoiding the "dance," by capturing it in the method definition itself. But thankfully, there's a great language ergonomics feature that captures this pattern, and automatically applies this rule for us. Meaning that `as_ref` isn't really necessary any more... right?

Side rant: ergonomics in Rust

I absolutely love the ergonomics features of Rust. There is no "but" in my love for RFC 2005. There is, however, a concern around learning and teaching a language with these kinds of ergonomics. These kinds of features work 99% of the time. But when they fail, as we're about to see, it can come as a large shock.

I'm guessing most Rustaceans, at least those that learned the language after 2016, never considered the fact that there was something weird about being able to pattern match a `Some` from an `&Option<String>` value. It feels natural. It *is* natural. But because you were never forced to confront this while learning the language, at some point in the distant future you'll crash into a wall when this ergonomic feature doesn't kick in.

I kind of wish there was a `--no-ergonomics` flag that we could turn on when learning the language to force us to confront all of these details. But there isn't. I'm hoping blog posts like this help out. Anyway, </rant>.

When RFC 2005 fails

We can fairly easily create a contrived example of match ergonomics failing to solve our problem. Let's "improve" our program above by factoring out the greet logic to its own helper function:

```
fn try_greet(option_name: Option<&String>) {
    match option_name {
        Some(name) => println!("Name is {}", name),
        None => println!("No name provided"),
    }
}

fn main() {
    let option_name: Option<String> = Some("Alice".to_owned());
    try_greet(&option_name);
    println!("{:?}", option_name);
}
```

This code won't compile:

```
error[E0308]: mismatched types
  --> src\main.rs:10:15
   |
10 |     try_greet(&option_name);
   |               ^^^^^^^^^^^^
   |               |
   |               expected enum `Option`, found `&Option<String>`
   |               help: you can convert from `&Option<T>` to `Option<&T>` using `.as_ref()`:
`&option_name.as_ref()`
   |
   = note:   expected enum `Option<&String>`
           found reference `&Option<String>`
```

Now we've bypassed any ability to use match ergonomics at the call site. With what we know about `as_ref`, it's easy enough to fix this. But, at least in my experience, the first time someone runs into this kind of error, it's a bit surprising, since most of us have never previously thought about the distinction between `Option<&T>` and `&Option<T>`.

These kinds of errors tend to pop up when combining together other helper functions, such as `map`, which circumvent the need for explicit pattern matching.

As an aside, you could solve this compile error pretty easily, without resorting to `as_ref`. Instead, you could change the type signature of `try_greet` to take a `&Option<String>` instead of an `Option<&String>`, and then allow the match ergonomics to kick in within the body of `try_greet`. One reason not to do this is that, as mentioned, this was all a contrived example to demonstrate a failure. But the other reason is more important: neither `&Option<String>` nor `Option<&String>` are good argument types. Let's explore that next.

When as_ref fails

We're taught pretty early in our Rust careers that, when receiving an argument to a function, we should prefer taking references to slices instead of references to owned objects. In other words:

```rust
fn greet_good(name: &str) {
    println!("Name is {}", name);
}

fn greet_bad(name: &String) {
    println!("Name is {}", name);
}
```

And in fact, if you pass this code by `clippy`, it will tell you to change the signature of `greet_bad`. The clippy lint description provides a great explanation of this, but suffice it to say that `greet_good` is more general in what it accepts than `greet_bad`.

The same logic applies to `try_greet`. Why should we accept `Option<&String>` instead of `Option<&str>`? And interestingly, clippy doesn't complain in this case like it did in `greet_bad`. To see why, let's change our signature like so and see what happens:

```rust
fn try_greet(option_name: Option<&str>) {
    match option_name {
        Some(name) => println!("Name is {}", name),
        None => println!("No name provided"),
    }
}

fn main() {
    let option_name: Option<String> = Some("Alice".to_owned());
    try_greet(option_name.as_ref());
    println!("{:?}", option_name);
}
```

This code no longer compiles:

```
error[E0308]: mismatched types
  --> src\main.rs:10:15
   |
10 |     try_greet(option_name.as_ref());
   |               ^^^^^^^^^^^^^^^^^^^^ expected `str`, found struct `String`
   |
   = note: expected enum `Option<&str>`
              found enum `Option<&String>`
```

This is another example of ergonomics failing. You see, when you call a function with an argument of type `&String`, but the function expects a `&str`, deref coercion kicks in and will perform a conversion for you. This is a piece of Rust ergonomics that we all rely on regularly, and every once in a while it completely fails to help us. This is one of those times. The compiler will not automatically convert a `Option<&String>` into an `Option<&str>`.

(You can also read more about coercions in the nomicon.)

Fortunately, there's another helper method on `Option` that does this for us. `as_deref` works just like `as_ref`, but additionally performs a `deref` method call on the value. Its implementation in `std` is interesting:

```rust
impl<T: Deref> Option<T> {
    pub fn as_deref(&self) -> Option<&T::Target> {
        self.as_ref().map(|t| t.deref())
    }
}
```

But we can also implement it more explicitly to see the behavior spelled out:

```rust
use std::ops::Deref;

fn try_greet(option_name: Option<&str>) {
    match option_name {
        Some(name) => println!("Name is {}", name),
        None => println!("No name provided"),
    }
}

fn my_as_deref<T: Deref>(x: &Option<T>) -> Option<&T::Target> {
    match *x {
        None => None,
        Some(ref t) => Some(t.deref())
    }
}

fn main() {
    let option_name: Option<String> = Some("Alice".to_owned());
    try_greet(my_as_deref(&option_name));
    println!("{:?}", option_name);
}
```

And to bring this back to something closer to real world code, here's a case where combining `as_deref` and `map` leads to much cleaner code than you'd otherwise have:

```rust
fn greet(name: &str) {
    println!("Name is {}", name);
}

fn main() {
    let option_name: Option<String> = Some("Alice".to_owned());
    option_name.as_deref().map(greet);
    println!("{:?}", option_name);
}
```

Real-ish life example

Like most of my blog posts, this one was inspired by some real world code. To simplify the concept down a bit, I was parsing a config file, and ended up with an `Option<String>`. I needed some code that would either provide the value from the config, or default to a static string in the source code. Without `as_deref`, I could have used `STATIC_STRING_VALUE.to_string()` to get types to line up, but that would have been ugly and inefficient. Here's a somewhat intact representation of that code:

```rust
use serde::Deserialize;

#[derive(Deserialize)]
struct Config {
    some_value: Option<String>
}

const DEFAULT_VALUE: &str = "my-default-value";

fn main() {
    let mut file = std::fs::File::open("config.yaml").unwrap();
    let config: Config = serde_yaml::from_reader(&mut file).unwrap();
    let value = config.some_value.as_deref().unwrap_or(DEFAULT_VALUE);
    println!("value is {}", value);
}
```