Why does the compiler allow pattern matching an enum in a function argument if it will never work and are there places pattern matching is useful?

Asked 3 years ago Modified 3 years ago Viewed 1k times

4

While looking at this axum crate example, I noticed a weird syntax in the function signature:

```
async fn create_user(
    Json(payload): Json<CreateUser>,
) -> impl IntoResponse {
  // blah blah
}
```

I eventually understood that `Json<T>(pub T)` is a newtype struct and wrapping `Json(payload)` means that we're extracting the `T` - we don't care about `Json<T>`, only the contained `T`.

I tested this out:

```
fn baz(Some(value): Option<i32>) {
    println!("value = {}", value);
}
```

The compiler complains that `None` is not covered.

My questions are:

- Why does the compiler accept this syntax for `enums` knowing that it will never work (i.e. there will always be a pattern that is not covered)?
- Are there other places where pattern matching in function arguments is useful?

rust

Share
Improve this question
Follow

edited Dec 8, 2021 at 17:08
Shepmaster
428k ● 111 ● 1.2k ● 1.5k

asked Dec 8, 2021 at 16:55
Midnight Exigent
625 ● 4 ● 20

3 I am not sure what you mean by your first question, as the compiler clearly does not accept this syntax for enums (hence the compiler error), as your example with `Option` shows. You can use pattern matching in function arguments with regular structures as well, to directly bind some or all of a struct's fields to a binding. This can be useful if you only need some of many fields in a structure (example), but mostly it is a matter of preference. – EvilTak Dec 8, 2021 at 17:05

It's hard to answer multiple questions made in one post. Please separate them into multiple questions so that we can help you better and so that your questions will help others in the future that have one of the same questions as you! – Shepmaster Dec 8, 2021 at 17:06

1 *knowing that [enums] will never work* — not true – Shepmaster Dec 8, 2021 at 17:10

2 thanks for the answers. why the thumbs down tho ? I genuinely thought I was asking interesting questions – Midnight Exigent Dec 9, 2021 at 2:44

1 @MidnightExigent thanks for asking this question. I had the exact same doubts when reading about Axum. – Gustavo Bezerra Apr 8, 2023 at 11:35

1 Answer

Sorted by: Highest score (default)

4

There is a distinction between *refutable* and *irrefutable* patterns.

You probably understand refutable patterns, i.e. patterns that can fail to match. These are typically tied to conditionals:

- `if let <PAT> = ...`
- `match ... { <PAT> => ... }`
- `while <PAT> = ...`

However, irrefutable patterns are perhaps more prevalent.

- `let <PAT> = ...`
- `for <PAT> in ...`
- `fn f(<PAT>: ...)`
- `|<PAT>| { ... }`
- basically anywhere you can declare a new variable is usually done via irrefutable patterns

You may understand their more advanced usage as "destructuring" or "structured bindings". Irrefutable patterns can:

- introduce new variables: `a`
- dereference: `&a`

- destructure structs: `{ field1, field2, ... }`
- destructure tuples: `(a, b)`
- destructure newtypes and single variant enums
- immediately discard the value: `_`
- mix all the above

This is explained further in Patterns and Matching in the Rust book.

So in essence, a pattern can go in place of a function parameter, but the pattern *must not fail*. Using patterns for destructuring struct fields or tuples is very common, though probably moreso for closures than functions.

answered Dec 8, 2021 at 23:19

kmdreko

59.1k ● 6 ● 91 ● 158