MATTHIAS ENDLER

Backend Engineer at trivago.
Likes just-in-time compilers and
hot chocolate. About me.

Published on 17th of December, 2017

Rust for Rubyists

— Idiomatic Patterns in Rust and Ruby

Recently I came across a delightful article on idiomatic Ruby. I'm not a good Ruby developer by any means, but I realized, that a lot of the patterns are also quite common in Rust. What follows is a side-by-side comparison of idiomatic code in both languages.

The Ruby code samples are from the original article.

Map and Higher-Order Functions

The first example is a pretty basic iteration over elements of a container using `map`.

```
user_ids = users.map { |user| user.id }
```

The `map` concept is also pretty standard in Rust. Compared to Ruby, we need to be a little more explicit here: If `users` is a vector of `User` objects, we first need to create an iterator from it:

```
let user_ids = users.iter().map(|user| user.id);
```

You might say that's quite verbose, but this additional abstraction allows us to express an important concept: will the iterator take ownership of the vector, or will it not?

- With `iter()`, you get a "read-only view" into the vector. After the iteration, it will be unchanged.
- With `into_iter()`, you take ownership over the vector. After the iteration, the vector will be gone. In Rust terminology, it will have *moved*.
- Read some more about the difference between `iter()` and `into_iter()` here.

The above Ruby code can be simplified like this:

```ruby
user_ids = users.map(&:id)
```

In Ruby, higher-order functions (like `map`) take blocks or procs as an argument and the language provides a convenient shortcut for method invocation — `&:id` is the same as `{|o| o.id()}`.

Something similar could be done in Rust:

```rust
let id = |u: &User| u.id;
let user_ids = users.iter().map(id);
```

This is probably not the most idiomatic way to do it, though. What you will see more often is the use of Universal Function Call Syntax in this case:[1]

```rust
let user_ids = users.iter().map(User::id);
```

In Rust, higher-order functions take **functions** as an argument. Therefore `users.iter().map(Users::id)` is more or less equivalent to `users.iter().map(|u| u.id())`.[2]

Also, `map()` in Rust returns another iterator and not a collection. If you want a collection, you would have to run `collect()` on that, as we'll see later.

Iteration with Each

Speaking of iteration, one pattern that I see a lot in Ruby code is this:

```
["Ruby", "Rust", "Python", "Cobol"].each do |lang|
  puts "Hello #{lang}!"
end
```

Since Rust 1.21, this is now also possible:

```
["Ruby", "Rust", "Python", "Cobol"]
    .iter()
    .for_each(|lang| println!("Hello {lang}!", lang = lang));
```

Although, more commonly one would write that as a normal for-loop in Rust:

```
for lang in ["Ruby", "Rust", "Python", "Cobol"].iter() {
    println!("Hello {lang}!", lang = lang);
}
```

Select and filter

Let's say you want to extract only even numbers from a collection in Ruby.

```
even_numbers = [1, 2, 3, 4, 5].map { |element| element if element.e
even_numbers = even_numbers.compact # [2, 4]
```

In this example, before calling `compact`, our `even_numbers` array had `nil` entries. Well, in Rust there is no concept of `nil` or `Null`. You don't need a `compact`. Also, `map` doesn't take predicates. You would use `filter` for that:

```
let even_numbers = vec![1, 2, 3, 4, 5]
    .iter()
    .filter(|&element| element % 2 == 0);
```

or, to make a vector out of the result

```
// Result: [2, 4]
let even_numbers: Vec<i64> = vec![1, 2, 3, 4, 5]
```

```
    .into_iter()
    .filter(|element| element % 2 == 0).collect();
```

Some hints:

- I'm using the type hint `Vec<i64>` here because, without it, Rust does not know what collection I want to build when calling `collect`.
- `vec!` is a macro for creating a vector.
- Instead of `iter`, I use `into_iter`. This way, I take ownership of the elements in the vector. With `iter()` I would get a `Vec<&i64>` instead.

In Rust, there is no `even` method on numbers, but that doesn't keep us from defining one!

```
let even = |x: &i64| x % 2 == 0;
let even_numbers = vec![1, 2, 3, 4, 5].into_iter().filter(even);
```

In a real-world scenario, you would probably use a third-party package (crate) like `num` for numerical mathematics:

```
extern crate num;
use num::Integer;

fn main() {
    let even_numbers: Vec<i64> = vec![1, 2, 3, 4, 5]
        .into_iter()
        .filter(|x| x.is_even()).collect();
}
```

In general, it's quite common to use crates in Rust for functionality that is not in the standard lib. Part of the reason why this is so well accepted is that cargo is such a rad package manager. (Maybe because it was built by no other than Yehuda Katz of Ruby fame. 😉)

As mentioned before, Rust does not have `nil`. However, there is still the concept of operations that can fail. The canonical type to express that is called `Result`.

Let's say you want to convert a vector of strings to integers.

```
let maybe_numbers = vec!["1", "2", "nah", "nope", "3"];
let numbers: Vec<_> = maybe_numbers
    .into_iter()
    .map(|i| i.parse::<u64>())
    .collect();
```

That looks nice, but maybe the output is a little unexpected. `numbers` will also contain the parsing errors:

```
[Ok(1), Ok(2), Err(ParseIntError { kind: InvalidDigit }), Err(Parse
```

Sometimes you're just interested in the successful operations. An easy way to filter out the errors is to use `filter_map`:

```
let maybe_numbers = vec!["1", "2", "nah", "nope", "3"];
let numbers: Vec<_> = maybe_numbers
    .into_iter()
    .filter_map(|i| i.parse::<u64>().ok())
    .collect();
```

I changed two things here:

- Instead of `map`, I'm now using `filter_map`.
- `parse` returns a `Result`, but `filter_map` expects an `Option`. We can convert a `Result` into an `Option` by calling `ok()` on it[3].

The return value contains all successfully converted strings:

```
[1, 2, 3]
```

The `filter_map` is similar to the `select` method in Ruby:

```
[1, 2, 3, 4, 5].select { |element| element.even? }
```

Random numbers

Here's how to get a random number from an array in Ruby:

```
[1, 2, 3].sample
```

That's quite nice and idiomatic! Compare that to Rust:

```
let mut rng = thread_rng();
rng.choose(&[1, 2, 3, 4, 5])
```

For the code to work, you need the `rand` crate. Click on the snippet for a running example.

There are some differences to Ruby. Namely, we need to be more explicit about what random number generator we want *exactly*. We decide for a lazily-initialized thread-local random number generator, seeded by the system. In this case, I'm using a slice instead of a vector. The main difference is that the slice has a fixed size while the vector does not.

Within the standard library, Rust doesn't have a `sample` or `choose` method on the slice itself. That's a design decision: the core of the language is kept small to allow evolving the language in the future.

This doesn't mean that you cannot have a nicer implementation today. For instance, you could define a `Choose` trait and implement it for `[T]`.

```
extern crate rand;
use rand::{thread_rng, Rng};

trait Choose<T> {
    fn choose(&self) -> Option<&T>;
}

impl<T> Choose<T> for [T] {
    fn choose(&self) -> Option<&T> {
        let mut rng = thread_rng();
        rng.choose(&self)
    }
}
```

This boilerplate could be put into a crate to make it reusable for others. With that, we arrive at a solution that rivals Ruby's elegance.

```
[1, 2, 4, 8, 16, 32].choose()
```

Implicit returns and expressions

Ruby methods automatically return the result of the last statement.

```
def get_user_ids(users)
  users.map(&:id)
end
```

Same for Rust. Note the missing semicolon.

```
fn get_user_ids(users: &[User]) -> Vec<u64> {
    users.iter().map(|user| user.id).collect()
}
```

But in Rust, this is just the beginning, because *everything* is an expression. The following block splits a string into characters, removes the `h`, and returns the result as a `HashSet`. This `HashSet` will be assigned to `x`.

```
let x: HashSet<_> = {
    // Get unique chars of a word {'h', 'e', 'l', 'o'}
    let unique = "hello".chars();
    // filter out the 'h'
    unique.filter(|&char| char != 'h').collect()
};
```

Same works for conditions:

```rust
let x = if 1 > 0 { "absolutely!" } else { "no seriously" };
```

Since a `match` statement is also an expression, you can assign the result to a variable, too!

```rust
enum Unit {
    Meter,
    Yard,
    Angstroem,
    Lightyear,
}

let length_in_meters = match unit {
    Unit::Meter => 1.0,
    Unit::Yard => 0.91,
    Unit::Angstroem => 0.0000000001,
    Unit::Lightyear => 9.461e+15,
};
```

Multiple Assignments

In Ruby you can assign multiple values to variables in one step:

```ruby
def values
  [1, 2, 3]
end

one, two, three = values
```

In Rust, you can only decompose tuples into tuples, but not a vector into a tuple for example. So this will work:

```rust
let (one, two, three) = (1, 2, 3);
```

But this won't:

```rust
let (one, two, three) = [1, 2, 3];
//    ^^^^^^^^^^^^^^^^^ expected array of 3 elements, found tuple
```

Neither will this:

```rust
let (one, two, three) = [1, 2, 3].iter().collect();
// a collection of type `(_, _, _)` cannot be built from an iterato
```

But with nightly Rust, you can now do this:

```rust
let [one, two, three] = [1, 2, 3];
```

On the other hand, there's a lot more you can do with destructuring apart from multiple assignments. You can write beautiful, ergonomic code using pattern syntax.

```rust
let x = 4;
let y = false;

match x {
    4 | 5 | 6 if y => println!("yes"),
    _ => println!("no"),
}
```

To quote *The Book*:

> *This prints `no` since the if condition applies to the whole pattern `4 | 5 | 6`, not only to the last value 6.*

String interpolation

Ruby has extensive string interpolation support.

```ruby
programming_language = "Ruby"
"#{programming_language} is a beautiful programming language"
```

This can be translated like so:

```
let programming_language = "Rust";
format!("{} is also a beautiful programming language", programming_
```

Named arguments are also possible, albeit much less common:

```
println!("{language} is also a beautiful programming language", lan
```

Rust's `println!()` syntax is even more extensive than Ruby's. Check the docs if you're curious about what else you can do.

That’s it!

Ruby comes with syntactic sugar for many common usage patterns, which allows for very elegant code. Low-level programming and raw performance are no primary goals of the language.

If you do need that, Rust might be a good fit, because it provides fine-grained hardware control with comparable ergonomics. If in doubt, Rust favors explicitness, though; it eschews magic.

Did I whet your appetite for idiomatic Rust? Have a look at this Github project. I'd be thankful for contributions.

Footnotes

1. Thanks to Florian Gilcher for the hint.↩

2. Thanks to masklin for pointing out multiple inaccuracies.↩

3. In the first version, I sait that `ok()` would convert a `Result` into a `boolean`, which was wrong. Thanks to isaacg for the correction.↩

💬 Comments available on Reddit, Hacker News.