Working with collections in Rust: Iterators

Jakub Pastuszek
2017

This presentation

- About iterators
- Working with iterators in Rust
- Rust type system and iterators
- Available generic iterators
- Implementing iterators

About iterators

What is an iterator?

- Iterator is an object that performs traversal and gives access to data elements in a container
- Purpose of an iterator is to allow a user to process every element of a container while isolating the user from the internal structure of the container
- Usually, the container provides the methods for creating iterators

Why are iterators useful?

- The container in its entirety would use too much memory
- Infinte or circular iteration
- Abstraction of input streams
- The container is large but you only need a few items
- Allow chaining operations like deduplication, mapping, splitting, etc.

Iterators vs Indexing

- Counting loops are not suitable to all data structures, e.g. structures with no or slow random access - e.g. UTF-8 text
- Iterators can provide a consistent way to iterate on data structures of all kinds
- An iterator can enforce additional restrictions on access
- An iterator may allow the container object to be modified without invalidating the iterator
- Avoid bound checking related errors with use of iterators (no ArrayIndexOutOfBoundsException)

Types of iterators - Internal

- They take a closure (higher order functions) and run the closure until it asks to break or iteration is complete
- Push-based iteration - iteration is powered inside the iterator itself
- Used commonly in: Ruby

Types of iterators - External

- State machine, advanced by the caller in a loop
- Pull-based iteration - iterator acts as a “bag” of values that client can pull things out of
- State needs to be preserved in the iterator object between invocations
- Found in: Java; Ruby (Enumerator#next)

Types of iterators - Generator

- Generator coroutine can yield values to its caller multiple times
- Generators automatically preserve their local state between invocations
- Found in: Python, Ruby (`Kernel#to_enum`)
- There are many discussions about implementing this style of iterators in Rust

History of iterators in Rust

- Rust used to have internal iterators but it proved difficult to:
  - implement generic algorithms that work with another iterator like zip
  - ensure consistency between implementations - lifetimes, break behaviour
  - compose iterators
- Rust now uses external iterators implemented as simple state machines that can be combined together to create complex iterators
- Complex iterators optimise very well creating true zero-cost abstraction

Working with iterators in Rust

Crating iterators

- There are three common methods defined on collections to crate iterators:
  - `iter()`, which iterates over `&T`
  - `iter_mut()`, which iterates over `&mut T`
  - `into_iter()` (`IntoIterator` trait) which iterates over `T`
- Free functions like: `empty()`, `once()`, `repeat()`
- Ranges like: `1..10`, `0..`, etc.

Converting into iterator

- `IntoIterator` trait defines common method of converting types into iterators
- It is implemented for most collections three times on:
  - value (`T`) - iterate values of collection my moving them out of and finally dropping the collection along with iterator
  - reference (`&T`) - borrow collection and iterate over references to values
  - mutable reference (`&mut T`) - mutably borrow collection and iterate over mutable references to values
- Also implemented for `Option` and `Result` types - iterator that provides zero or one item

For loops - IntoIterator trait

- For loops use `IntoIterator` trait and it's `into_iter()` method to crate iterator from collection
- For loops will call `next()` method on the iterator to get each item and then run the body of the loop with that item in scope
- Many collection types will implement `IntoIterator` trait so they can be iterated over using for loops
- Programmers can also implement `IntoIterator` trait for their own types to allow them to be iterated by for loops

For loops - by value

- Items are moved out of the container by value
- Container is moved into iterator by value and dropped with the iterator
- In this example `impl<T> IntoIterator for Vec<T>` implementation is used

```
let numbers = vec![1, 2, 3, 4, 5];
for n in numbers {
    println!("{}", n)
}
// println!("{:?}", numbers); -- error[E0382]: use of moved value: `numbers`

let numbers = vec![1, 2, 3, 4, 5];
{
    let mut i = numbers.into_iter(); // IntoIterator trait
    loop {
        match i.next() {
            Some(n) => println!("{}", n),
            None => break,
        }
    }
}
// println!("{:?}", numbers); -- error[E0382]: use of moved value: `numbers`
```

For loops - by reference

- Iteration over references to items
- Iterator borrows (takes reference to) the container until iterator object is dropped
- In this example `impl<'a, T> IntoIterator for &'a Vec<T>` implementation is used

```
let numbers = vec![1, 2, 3, 4, 5];
for n in &numbers {
    println!("{}", n)
}
println!("{:?}", numbers); // [1, 2, 3, 4, 5]

let numbers = vec![1, 2, 3, 4, 5];
{
    let mut i = (&numbers).into_iter(); // IntoIterator trait
    loop {
        match i.next() {
            Some(n) => println!("{}", n),
            None => break,
        }
    }
}
println!("{:?}", numbers); // [1, 2, 3, 4, 5]
```

For loops - by mutable reference

- In this case iteration works with mutable references to items allowing them to be modified
- The container is mutably borrowed by the iterator until it is dropped
- In this example `impl<'a, T> IntoIterator for &'a mut Vec<T>` implementation is used

```
let mut numbers = vec![1, 2, 3, 4, 5];
for n in &mut numbers {
    *n *= 2;
    println!("{}", n)
}
println!("{:?}", numbers); // [2, 4, 6, 8, 10]

let mut numbers = vec![1, 2, 3, 4, 5];
{
    let mut i = (&mut numbers).into_iter(); // IntoIterator trait
    loop {
        match i.next() {
            Some(n) => {
                *n *= 2;
                println!("{}", n)
            }
            None => break,
        }
    }
}
println!("{:?}", numbers); // [2, 4, 6, 8, 10]
```

Chaining iterators

- Many functions implemented on `Iterator` trait return new iterator wrapping current one and providing some operations on items or iteration

- Stdlib generic iterators examples:

  - `fn map<B, F>(self, f: F) -> Map<Self, F> where F: FnMut(Self::Item) -> B`

  - `fn filter<P>(self, predicate: P) -> Filter<Self, P> where P: FnMut(&Self::Item) -> bool`

  - `fn filter_map<B, F>(self, f: F) -> FilterMap<Self, F> where F: FnMut(Self::Item) -> Option<B>`

  - `fn cloned<'a, T>(self) -> Cloned<Self> where Self: Iterator<Item=&'a T>, T: 'a + Clone`

Chaining iterators - example

- For loop over chained iterator:

```
for word in "The quick brown fox jumps over the lazy dog"
    .split_whitespace()
    .map(|w| w.chars())
    .map(|mut c| {
        c.next().into_iter().flat_map(|c| c.to_uppercase()).chain(
            c.flat_map(|c| c.to_lowercase()),
        )
    })
    .map(|c| c.collect::<String>())
{
    print!("{:?} ", word); // "The" "Quick" "Brown" "Fox" "Jumps" "Over" "The" "Lazy" "Dog"
}
```

Consuming iterators

- `FromIterator` trait is used by `collect()` and implemented for many collections
  - Allows any iterator to be collected into a collection of different types like: `Vec`, `HashMap` (from `(K, V)` tuples), `LinkedList`, `String` (from `str`, `char`)
  - Collecting into `Result<V, E>` from iterator of `Result<A, E>`s: `impl<A, E, V> FromIterator<Result<A, E>> for Result<V, E> where V: FromIterator<A>`
- `Extend` trait that allows extending collections with elements from iterators via `extend()`
- `Product` trait works with `product()` and is used to generate items by multiplying them with items from another iterator - it is implemented for numbers and `Result`

Returning iterators

- Dynamic dispatch: `-> Box<Iterator<Item=Foo>>`
- Static dispatch (nightly only): `-> impl Iterator<Item=Foo>>`
- Own wrapper type: `-> MyIter<Item=Foo>`

```
 1 #![feature(conservative_impl_trait)]
 2
 3 fn even(from: u64, to: u64) -> impl Iterator<Item = u64> {
 4     // type: std::iter::Filter<std::ops::Range<u64>, [closure@<anon>:4:27: 4:41]>
 5     (from..to + 1).filter(|i| i % 2 == 0)
 6 }
 7
 8 fn main() {
 9     println!("{:?}", even(1, 10).collect::<Vec<_>>()) // [2, 4, 6, 8, 10]
10 }
```

Rust type system and iterators

Properties of iterators in Rust

- No ambiguity between end of iteration and null-like value
- No need for extra check to determine end of iteration
- Container cannot be mutated (borrwock) avoiding iterator invalidation problem (no ConcurrentModificationException)
- Iterated items cannot be tied to iterator object due to lifetimes so iterator does not to be kept around for items to be valid pointers
- Iterators fully consuming the container - move semantics - e.g. `into_iter()` or value, `drain()`
- Iterators are lazy - nothing really happens until you call `next()`
- Zero-cost - iterator state machines optimise very well (most of the time)

Limitations and problems

- Chaining iterators together can crate very long and complex nested types - this makes them hard or even impossible to write down:
  - for return type of functions to get static dispatch
  - storing them in structures
  - e.g. type signature:
    ```
    std::iter::Take<std::iter::Filter<std::iter::Map<std::iter::Cycle<std::iter::Cloned<std::slice::Iter<'_, {integer}>>>, [closure@<anon>:3:47: 3:56]>, [closure@<anon>:3:65: 3:75]>>
    ```
- Not always zero-cost - e.g. `chain()`

Iterators and impl RFC

- Prevents leaking of information from APIs - when you only should know what traits are implemented on returned object and not what the object itself is
- Returning complex types without need of writing them down or hiding in custom type
- Returning types containing closures - this is normally impossible as their type are anonymous
- Available only on nightly
- Still being discussed as to where to allow usage (e.g. structurs, arguments, etc.)

Available generic iterators

Stdlib iterators

- `Iterator` trait provides many wrapper iterators and algorithms; some examples:
  - operation on items and iteration: `map()`, `cloned()`, `cycle()`, `zip()`, `unzip()`, `partition()`
  - access: `nth()`, `first()`, `last()`, `skip()`, `take()`, `filter()`, `peekable()`, `skip_while()`, `take_while()`
  - predicate functions: `any()`, `all()`
  - lookup: `find()`, `position()`, …
  - reduction functions: `count()`, `sum()`, `min()`, `max()`, `fold()`, …
  - comparison: `cmp()`, `eq()`, `lt()`, …
- `DoubleEndedIterator` trait - an iterator able to yield elements from both ends
- `ExactSizeIterator` trait - an iterator that knows its exact length

More iterators with Itertools

`interleave()`, `interleave_shortest()`, `intersperse()`, `zip_longest()`, `zip_eq()`, `batching()`, `group_by()`, `chunks()`, `tuple_windows()`, `tuples()`, `tee()`, `step()`, `map_results()`, `merge()`, `merge_by()`, `kmerge()`, `kmerge_by()`, `cartesian_product()`, `coalesce()`, `dedup()`, `unique()`, `unique_by()`, `peeking_take_while()`, `take_while_ref()`, `while_some()`, `tuple_combinations()`, `combinations()`, `pad_using()`, `flatten()`, `with_position()`, `next_tuple()`, `find_position()`, `dropping()`, `dropping_back()`, `foreach()`, `collect_vec()`, `set_from()`, `join()`, `format()`, `format_with()`, `fold_results()`, `fold_options()`, `fold1()`, `fold_while()`, `sorted()`, `sorted_by()`, `partition_map()`, `minmax()`, `minmax_by_key()`, `minmax_by()`

Parallel iterators with Rayon

- Standard single sequential iteration:

```
let total_price = stores.iter()
    .map(|store| store.compute_price(&list))
    .sum();
```

- Multithreaded (work stealing) iteration:

```
let total_price = stores.par_iter()
    .map(|store| store.compute_price(&list))
    .sum();
```

Implementing iterators

Implementing traits

- Iterator object that contains iteration state and implements `Iterator` trait
- Container may implement `IntoIterator` trait for value and reference types that provides custom iterator

Iterator trait

- Trait signature
  - associated type:
    `Item` - The type of the elements being iterated over
  - one required method:
    `fn next(&mut self) -> Option<Self::Item>`
    Advances the iterator and returns the next value
- Many more methods are provided by this trait like: `size_hitnt()`, `count()`, `first()`, `last()`, `map()`, `zip()`, `collect()`, `filter()`, …

Example iterator

```
struct Coll {
    state: Option<u64>
}

fn main() {
    let iter = Coll {
        state: Some(13)
    };

    println!("{:?}", iter.collect::<Vec<_>>());
    // [13, 40, 20, 10, 5, 16, 8, 4, 2, 1]
}
```

```
impl Iterator for Coll {
    type Item = u64;

    fn next(&mut self) -> Option<u64> {
        let ret = self.state;
        self.state = match self.state {
            None => None,
            Some(n) if n <= 1 => None,
            Some(n) if n % 2 == 0 => Some(n / 2),
            Some(n) => Some(n * 3 + 1)
        };
        ret
    }
}
```

Example implementations

- Iterator with lifetimes on Item
  https://gist.github.com/jpastuszek/bcabed36f21e813fce00d812413d85b6
- Iterating structure fields
- Map-shuffle-reduce groups iterator
- Tree depth first search iterator:
  https://github.com/jpastuszek/chappie/blob/af987edda7527bb4f1cd3b94b296c38beff22ab3/src/search.rs

Thank you!
Q & A

Linear types and iterators

- It is possible to define an iterator that is guaranteed to be fully consumed