**Rust —avoid recursive iterators 1**

[[](https://medium.com/@techhara?source=post_page-----50e5701e3743--------------------------------)](https://medium.com/@techhara?source=post_page-----50e5701e3743--------------------------------)

[TechHara](https://medium.com/@techhara?source=post_page-----50e5701e3743--------------------------------)

·

Follow

3 min read

·

3 days ago

53

1

Say you have an iterator over integers, and you are tasked to create an iterator that returns sum of pairs. For example

let iter = [0,1,2,3,4,5,6,7,8,9,10].into\_iter();  
// consume iter and generate an iterator that returns sum of pairs  
let sum\_iter = todo!();  
let xs: Vec<\_> = sum\_iter.collect(); // [0+1, 2+3, 4+5, 6+7, 8+9]

See if you can implement this using **recursion**. If there is an odd number of integers, you can return None for the last element.

// try to implement yourself  
// Hint: you will probably need to use Iterator::chain() method.

Test your *recursive* implementation with an iterator of size, say 1,000,000 elements. What happens?

*I am almost certain that it will****crash with stack overflow****.*

Here is my very first attempt of recursive implementation

fn add\_two(mut iter: impl Iterator<Item = u32>) -> Box<dyn Iterator<Item = u32>> {  
 match (iter.next(), iter.next()) {  
 (Some(x), Some(y)) => Box::new(std::iter::once(x + y).chain(add\_two(iter))),  
 \_ => Box::new(std::iter::empty()),  
 }  
}  
  
fn main() {  
 let iter = 0..1\_000\_000;  
 // consume iter and generate an iterator that returns sum of pairs  
 let sum\_iter = add\_two(iter); // [0+1, 2+3, 4+5, 6+7, ...]  
 println!("{}", sum\_iter.count());  
}

Running this code [results](https://play.rust-lang.org/?version=stable&mode=debug&edition=2021&gist=595a3fe589cc53a6a3dcd42643cf5029) in stack overflow



The interesting thing is that the stack overflow occurs at let sum\_iter = add\_two(iter);, not even before evaluating the iterator with count(). Did someone say iterators are *lazy* in Rust? Apparently not.

**Chain is culprit**

The reason for stack overflow is because of Iterator::chain() method. This method is not lazy. It can’t be. Its argument is a value, not a function. Only those functions that take a function as an argument can be lazy, such as map(). That’s why it crashes during the construction, even before evaluating. That is just a terrible implementation.

*Recursive Iterator::chain() is prone to stack overflow*

Well, let’s create a lazy version of chain(). There are a few ways to do it.

// https://github.com/rust-itertools/itertools/issues/370  
.chain(std::iter::once\_with(|| ...).flatten())  
// https://stackoverflow.com/questions/49455885/chain-two-iterators-while-lazily-constructing-the-second-one  
.chain([()].into\_iter().flat\_map(|\_| ...))  
// https://stackoverflow.com/questions/49455885/chain-two-iterators-while-lazily-constructing-the-second-one  
.chain\_with(|| ...)

Here is the revised implementation using chain\_with() method following [this](https://stackoverflow.com/questions/49455885/chain-two-iterators-while-lazily-constructing-the-second-one)

fn add\_two(mut iter: impl Iterator<Item = u32> + 'static) -> Box<dyn Iterator<Item = u32>> {  
 match (iter.next(), iter.next()) {  
 (Some(x), Some(y)) => Box::new(std::iter::once(x + y).chain\_with(|| add\_two(iter))),  
 \_ => Box::new(std::iter::empty()),  
 }  
}  
  
trait IteratorExt: Iterator {  
 fn chain\_with<F, I>(self, f: F) -> ChainWith<Self, F, I::IntoIter>  
 where  
 Self: Sized,  
 F: FnOnce() -> I,  
 I: IntoIterator<Item = Self::Item>,  
 {  
 ChainWith {  
 base: self,  
 factory: Some(f),  
 iterator: None,  
 }  
 }  
}  
  
impl<I: Iterator> IteratorExt for I {}  
  
struct ChainWith<B, F, I> {  
 base: B,  
 factory: Option<F>,  
 iterator: Option<I>,  
}  
  
impl<B, F, I> Iterator for ChainWith<B, F, I::IntoIter>  
where  
 B: Iterator,  
 F: FnOnce() -> I,  
 I: IntoIterator<Item = B::Item>,  
{  
 type Item = I::Item;  
 fn next(&mut self) -> Option<Self::Item> {  
 if let Some(b) = self.base.next() {  
 return Some(b);  
 }  
  
 // Exhausted the first, generate the second  
  
 if let Some(f) = self.factory.take() {  
 self.iterator = Some(f().into\_iter());  
 }  
  
 self.iterator  
 .as\_mut()  
 .expect("There must be an iterator")  
 .next()  
 }  
}  
  
fn main() {  
 let iter = 0..1\_000\_000;  
 // consume iter and generate an iterator that returns sum of pairs  
 let sum\_iter = add\_two(iter); // [0+1, 2+3, 4+5, 6+7, ...]  
 println!("{}", sum\_iter.count());  
}

Running this one still [results](https://play.rust-lang.org/?version=stable&mode=debug&edition=2021&gist=48be2846dfd07ed93a18476e73f664d9) in stack overflow (after a long wait time). This time, however, it crashes while executing the last line, i.e., sum\_iter.count().

Surprisingly, lazy version of chain() does not help. In the next part, we will continue to analyze the root cause of the stack overflow, so stay tuned!