Rust 语言圣经学习笔记
Rust Course Learning Notes[1]

Learner
LaTeX by Xin Wang

Notes

Contents

List of Figures

List of Tables

Part I

Rust 语言基础学习

Chapter 1

寻找牛刀，以便小试

1.1 安装 Rust

操作系统	命令
Unix	`curl --proto '=https' --tlsv1.2 -sSf https://sh.rustup.rs | sh`
Windows	`Download and run rustup-init.exe`

Table 1.1: Rust 安装命令

Chapter 2

手把手带你实现链表

2.1 我们到底需不需要链表

Fact 2.1.1

链表真的是一种糟糕的数据结构，尽管它在部分场景下确实很有用：

- 对列表进行大量的分割和合并操作
- 无锁并发
- 要实现内核或嵌入式的服务
- 你在使用一个纯函数式语言，由于受限的语法和缺少可变性，因此你需要使用链表来解决这些问题

但是实事求是的说，这些场景对于几乎任何 **Rust** 开发都是很少遇到的，99% 的场景你可以使用 **Vec** 来替代，然后 1% 中的 99% 可以使用 **VecDeque**。由于它们具有更少的内存分配次数、更低的内存占用、随机访问和缓存亲和特性，因此能够适用于绝大多数工作场景。总之，类似于 trie 树，链表也是一种非常小众的数据结构，特别是对于 **Rust** 开发而言。

Remark.

> 本书只是为了学习链表该如何实现，如果大家只是为了使用链表，强烈推荐直接使用标准库或者社区提供的现成实现，例如 `std::collections::LinkedList`

Claim: 链表有 O(1) 的分割、合并、插入、移除性能

是的，但是你首先要考虑的是，这些代码被调用的频率是怎么样的？是否在热点路径？答案如果是否定的，那么还是强烈建议使用 **Vec** 等传统数据结构，况且整个数组的拷贝也是相当快的！
况且，**Vec** 上的 **push** 和 **pop** 操作是 **O(1)** 的，它们比链表提供的 **push** 和 **pop** 要更快！我们只需要通过一个**指针 + 内存偏移**就可以访问了。
但是如果你的整体项目确实因为某一段分割、合并的代码导致了性能低下，那么就放心大胆的使用链表吧。

Claim: 我无法接受内存重新分配的代价

是的，**Vec** 当 **capacity** 不够时，会重新分配一块内存，然后将之前的 **Vec** 全部拷贝过去，但是对于绝大多数使用场景，要么 **Vec** 不在热点路径中，要么 **Vec** 的容量可以提前预测。
对于前者，那性能如何自然无关紧要。而对于后者，我们只需要使用 `Vec::with_capacity` 提前分配足够的空间即可，同时，**Rust** 中所有的迭代器还提供了 `size_hint` 也可以解决这种问题。
当然，如果这段代码在热点路径，且你无法提前预测所需的容量，那么链表确实会更提升性能。

Claim: 链表更节省内存空间

首先，这个问题较为复杂。一个标准的数组调整策略是：增加或减少数组的长度使数组最多有一半为空，例如 **capacity** 增长是翻倍的策略。这确实会导致内存空间的浪费，特别是在 **Rust** 中，我们不会自动收缩集合类型。
但是上面说的是最坏的情况，如果是最好的情况，那整个数组其实只有 3 个指针大小 (指针在 **Rust** 中占用一个 **word** 的空间，例如 64 位机器就是 8 个字节的大小) 的内存浪费，或者说，没有浪费。
而且链表实际上也有内存浪费，例如链表中的每个元素都会占用额外的内存：单向链表浪费一个指针，双向链表浪费两个指针。当然，如果你的链表中每个元素都很大，那相对来说，这种浪费也微不足道，但是如果链表的元素较小且数量很多呢？那浪费的空间就相当可观了！
当然，这个也和使用的内存分配器有关 (**allocator**)：对链表节点的分配和回收会经常发生，这样就不会浪费内存。
总之，如果链表的元素较大，你也无法预测数组的空间，同时还有一个不错的内存分配器，那链表确实可以节省空间！

Claim: 我在函数语言中一直使用链表

对于函数语言而言，链表确实非常棒，因为你可以解决可变性问题，还能递归地去使用，当然，可能还有一定的图方便的因素，因为链表不用操心长度等问题。
但彼之蜜糖不等于吾之蜜糖，函数语言的一些使用习惯不应该带入到其它语言中，例如 **Rust**。

- 函数语言往往将链表用于迭代，但是 **Rust** 中最适合迭代的数据结构是迭代器 **Iterator**
- 函数式语言的不可变对于 **Rust** 也不是问题
- **Rust** 还支持对数组进行切片以获取其中一部分连续的元素，而在函数语言中你可能得通过链表的 **headtail** 分割来完成

其实，在函数语言中，我们也应该选择合适的数据结构来解决适合的场景，而不是一根链表挂腰间，潇潇洒洒走天下。

Claim: 链表适合构建并发数据结构

是这样的，如果有这样的需求，那么链表会非常合适！但是只有在你确实需要并发数据结构，且没有其它办法时，再考虑链表！

Claim: 链表非常适合教学目的

额... 这么说也没错，毕竟所有的编程语言课程都以链表来作为最常见的练手项目，包括本书也是服务于这个目的的。

2.2 不太优秀的单向链表：栈

2.2.1 数据布局

Fact 2.2.1

对于 **Rust** 编译器而言，所 **Box** 有栈上的类型都必须在编译期有固定的长度，一个简单的解决方案就是使用将值封装到堆上，然后使用栈上的定长指针来指向堆上不定长的值。

Fact 2.2.2

枚举有一个特点，枚举成员占用的内存空间大小跟最大的成员对齐

Fact 2.2.3

pub enum 会要求它的所有成员必须是 **pub**

Fact 2.2.4

从编程的角度而言，我们还是希望让实现细节只保留在内部，而不是对外公开

2.2.2 基本操作

Fact 2.2.5

`mem::replace` 这个非常有用的函数允许我们从一个借用中偷出一个值的同时再放入一个新值。

Fact 2.2.6

unimplemented!() 该宏可以明确地说明目前的代码还没有实现，一旦代码执行到 **unimplemented!()** 的位置，就会发生一个 **panic**。

Fact 2.2.7

panic 是一种发散函数，该函数永不返回任何值，因此可以用于需要返回任何类型的地方。

2.2.3 最后实现

Fact 2.2.8

Drop 特征，若变量实现了该特征，则在它离开作用域时将自动调用解构函数以实现资源清理释放工作，最妙的是，这一切都发生在编译期，因此没有多余的性能开销。

Fact 2.2.9

事实上，我们无需手动为自定义类型实现 **Drop** 特征，原因是 **Rust** 自动为几乎所有类型都实现了 **Drop**，例如我们自定义的结构体，只要结构体的所有字段都实现了 **Drop**，那结构体也会自动实现 **Drop**!

```
use std::mem;

pub struct List {
    head: Link,
}

enum Link {
    Empty,
    More(Box<Node>),
}

struct Node {
    elem: i32,
    next: Link,
}

impl List {
    pub fn new() -> Self {
        List { head: Link::Empty }
    }

    pub fn push(&mut self, elem: i32) {
        let new_node = Box::new(Node {
            elem: elem,
            next: mem::replace(&mut self.head, Link::Empty),
        });

        self.head = Link::More(new_node);
    }

    pub fn pop(&mut self) -> Option<i32> {
        match mem::replace(&mut self.head, Link::Empty) {
            Link::Empty => None,
            Link::More(node) => {
```

```
                self.head = node.next;
                Some(node.elem)
            }
        }
    }
}

impl Drop for List {
    fn drop(&mut self) {
        let mut cur_link = mem::replace(&mut self.head, Link::Empty);

        while let Link::More(mut boxed_node) = cur_link {
            cur_link = mem::replace(&mut boxed_node.next, Link::Empty);
        }
    }
}

#[cfg(test)]
mod test {
    use super::List;

    #[test]
    fn basics() {
        let mut list = List::new();

        // Check empty list behaves right
        assert_eq!(list.pop(), None);

        // Populate list
        list.push(1);
        list.push(2);
        list.push(3);

        // Check normal removal
        assert_eq!(list.pop(), Some(3));
        assert_eq!(list.pop(), Some(2));

        // Push some more just to make sure nothing's corrupted
        list.push(4);
        list.push(5);

        // Check normal removal
        assert_eq!(list.pop(), Some(5));
        assert_eq!(list.pop(), Some(4));

```

```
80          // Check exhaustion
81          assert_eq!(list.pop(), Some(1));
82          assert_eq!(list.pop(), None);
83      }
84  }
```

2.3 还可以的单向链表

2.3.1 优化类型定义

Claim: Option

Fact 2.3.1

为了代码可读性，我们不能直接使用冗长的类型，为此可以使用类型别名。
`type Link = Option<Box<Node>>;`

Fact 2.3.2

之前咱们用到了 `mem::replace` 这个让人胆战心惊但是又非常有用的函数，而 **Option** 直接提供了一个方法 **take** 用于替代它

Fact 2.3.3

`match option { None => None, Some(x) => Some(y) }` 这段代码可以直接使用 **map** 方法代替，**map** 会对 **Some(x)** 中的值进行映射，最终返回一个新的 **Some(y)** 值。

Claim: 泛型

Fact 2.3.4

泛型参数也是类型定义的一部分

2.3.2 定义 Peek 函数

Fact 2.3.5

peek 函数，它会返回链表的表头元素的引用
`pub fn peek(&self) -> Option<&T>`

2.3.3 IntoIter 和 Iter

Fact 2.3.6

集合类型可以通过 **IntoIter** 特征进行迭代

```
pub trait Iterator {
    type Item;
    fn next(&mut self) -> Option<Self::Item>;
}
```

Fact 2.3.7

需要注意，每个集合类型应该实现 3 种迭代器类型：

- IntoIter - T
- IterMut - &mut T
- Iter - &T

IntoIter 类型迭代器的 next 方法会拿走被迭代值的所有权
IterMut 是可变借用
Iter 是不可变借用

Fact 2.3.8

map 是一个泛型函数:
`pub fn map<U, F>(self, f: F) -> Option<U>`

Fact 2.3.9

turbofish 形式的符号::<> 可以告诉编译器我们希望用哪个具体的类型来替代泛型类型

2.3.4 IterMut 以及完整代码

Fact 2.3.10

“Sugar”指的是让语言更易读写的语法特性
“Desugar”是将这些便利语法转换回编译器能够理解的更基础的形式的过程

```
pub struct List<T> {
    head: Link<T>,
}

type Link<T> = Option<Box<Node<T>>>;

```

```rust
struct Node<T> {
    elem: T,
    next: Link<T>,
}

impl<T> List<T> {
    pub fn new() -> Self {
        List { head: None }
    }

    pub fn push(&mut self, elem: T) {
        let new_node = Box::new(Node {
            elem: elem,
            next: self.head.take(),
        });

        self.head = Some(new_node);
    }

    pub fn pop(&mut self) -> Option<T> {
        self.head.take().map(|node| {
            self.head = node.next;
            node.elem
        })
    }

    pub fn peek(&self) -> Option<&T> {
        self.head.as_ref().map(|node| {
            &node.elem
        })
    }

    pub fn peek_mut(&mut self) -> Option<&mut T> {
        self.head.as_mut().map(|node| {
            &mut node.elem
        })
    }

    pub fn into_iter(self) -> IntoIter<T> {
        IntoIter(self)
    }

    pub fn iter(&self) -> Iter<'_, T> {
        Iter { next: self.head.as_deref() }
    }
```

```
52
53      pub fn iter_mut(&mut self) -> IterMut<'_, T> {
54          IterMut { next: self.head.as_deref_mut() }
55      }
56  }
57
58  impl<T> Drop for List<T> {
59      fn drop(&mut self) {
60          let mut cur_link = self.head.take();
61          while let Some(mut boxed_node) = cur_link {
62              cur_link = boxed_node.next.take();
63          }
64      }
65  }
66
67  pub struct IntoIter<T>(List<T>);
68
69  impl<T> Iterator for IntoIter<T> {
70      type Item = T;
71      fn next(&mut self) -> Option<Self::Item> {
72          // access fields of a tuple struct numerically
73          self.0.pop()
74      }
75  }
76
77  pub struct Iter<'a, T> {
78      next: Option<&'a Node<T>>,
79  }
80
81  impl<'a, T> Iterator for Iter<'a, T> {
82      type Item = &'a T;
83      fn next(&mut self) -> Option<Self::Item> {
84          self.next.map(|node| {
85              self.next = node.next.as_deref();
86              &node.elem
87          })
88      }
89  }
90
91  pub struct IterMut<'a, T> {
92      next: Option<&'a mut Node<T>>,
93  }
94
95  impl<'a, T> Iterator for IterMut<'a, T> {
96      type Item = &'a mut T;
```

```rust

    fn next(&mut self) -> Option<Self::Item> {
        self.next.take().map(|node| {
            self.next = node.next.as_deref_mut();
            &mut node.elem
        })
    }
}

#[cfg(test)]
mod test {
    use super::List;

    #[test]
    fn basics() {
        let mut list = List::new();

        // Check empty list behaves right
        assert_eq!(list.pop(), None);

        // Populate list
        list.push(1);
        list.push(2);
        list.push(3);

        // Check normal removal
        assert_eq!(list.pop(), Some(3));
        assert_eq!(list.pop(), Some(2));

        // Push some more just to make sure nothing's corrupted
        list.push(4);
        list.push(5);

        // Check normal removal
        assert_eq!(list.pop(), Some(5));
        assert_eq!(list.pop(), Some(4));

        // Check exhaustion
        assert_eq!(list.pop(), Some(1));
        assert_eq!(list.pop(), None);
    }

    #[test]
    fn peek() {
        let mut list = List::new();
```

```
142         assert_eq!(list.peek(), None);
143         assert_eq!(list.peek_mut(), None);
144         list.push(1); list.push(2); list.push(3);
145 
146         assert_eq!(list.peek(), Some(&3));
147         assert_eq!(list.peek_mut(), Some(&mut 3));
148 
149         list.peek_mut().map(|value| {
150             *value = 42
151         });
152 
153         assert_eq!(list.peek(), Some(&42));
154         assert_eq!(list.pop(), Some(42));
155     }
156 
157     #[test]
158     fn into_iter() {
159         let mut list = List::new();
160         list.push(1); list.push(2); list.push(3);
161 
162         let mut iter = list.into_iter();
163         assert_eq!(iter.next(), Some(3));
164         assert_eq!(iter.next(), Some(2));
165         assert_eq!(iter.next(), Some(1));
166         assert_eq!(iter.next(), None);
167     }
168 
169     #[test]
170     fn iter() {
171         let mut list = List::new();
172         list.push(1); list.push(2); list.push(3);
173 
174         let mut iter = list.iter();
175         assert_eq!(iter.next(), Some(&3));
176         assert_eq!(iter.next(), Some(&2));
177         assert_eq!(iter.next(), Some(&1));
178     }
179 
180     #[test]
181     fn iter_mut() {
182         let mut list = List::new();
183         list.push(1); list.push(2); list.push(3);
184 
185         let mut iter = list.iter_mut();
186         assert_eq!(iter.next(), Some(&mut 3));
```

```
187         assert_eq!(iter.next(), Some(&mut 2));
188         assert_eq!(iter.next(), Some(&mut 1));
189     }
190 }
```

2.4 持久化单向链表

2.4.1 数据布局和基本操作

Fact 2.4.1

对于新的链表来说，最重要的就是我们可以自由地操控列表的尾部 (**tail**)。

Fact 2.4.2

标准库为我们提供了引用计数的数据结构: **Rc/Arc**，引用计数可以被认为是一种简单的 **GC**，对于很多场景来说，引用计数的数据吞吐量要远小于垃圾回收，而且引用计数还存在循环引用的风险！但我们没有其它选择。

Fact 2.4.3

使用 **Rc** 意味着我们的数据将无法被改变，因为它不具备内部可变性。

Fact 2.4.4

需要注意的是, **Rc** 在 **Rust** 中并不是一等公民，它没有被包含在 `std::prelude` 中，因此我们必须手动引入 `use std::rc::Rc`

Fact 2.4.5

`Rc::clone`，对于该方法而言，clone 仅仅是增加引用计数，并不是复制底层的数据。虽然 **Rc** 的性能要比 **Box** 的引用方式低一点，但是它依然是多所有权前提下最好的解决方式或者说之一。

2.4.2 Drop、Arc 及完整代码

Fact 2.4.6

`Rc::Try_unwrap`，该方法会判断当前的 **Rc** 是否只有一个强引用，若是，则返回 **Rc** 持有的值，否则返回一个错误。

Fact 2.4.7

不可变链表的一个很大的好处就在于多线程访问时自带安全性，毕竟共享可变性是多线程危险的源泉，最好也是最简单的解决办法就是直接干掉可变性。
但是 **Rc<T>** 本身并不是线程安全的，原因是它内部的引用计数器并不是线程安全的，通俗来讲，计数器没有加锁也没有实现原子性。
Arc<T> 是线程安全的。

Fact 2.4.8

Rust 通过提供 **Send** 和 **Sync** 两个特征来保证线程安全

```rust
use std::rc::Rc;

pub struct List<T> {
    head: Link<T>,
}

type Link<T> = Option<Rc<Node<T>>>;

struct Node<T> {
    elem: T,
    next: Link<T>,
}

impl<T> List<T> {
    pub fn new() -> Self {
        List { head: None }
    }

    pub fn prepend(&self, elem: T) -> List<T> {
        List { head: Some(Rc::new(Node {
            elem: elem,
            next: self.head.clone(),
        }))}
    }

    pub fn tail(&self) -> List<T> {
        List { head: self.head.as_ref().and_then(|node| node.next.clone()) }
    }

    pub fn head(&self) -> Option<&T> {
        self.head.as_ref().map(|node| &node.elem)
    }

```

```
34      pub fn iter(&self) -> Iter<'_, T> {
35          Iter { next: self.head.as_deref() }
36      }
37  }
38
39  impl<T> Drop for List<T> {
40      fn drop(&mut self) {
41          let mut head = self.head.take();
42          while let Some(node) = head {
43              if let Ok(mut node) = Rc::try_unwrap(node) {
44                  head = node.next.take();
45              } else {
46                  break;
47              }
48          }
49      }
50  }
51
52  pub struct Iter<'a, T> {
53      next: Option<&'a Node<T>>,
54  }
55
56  impl<'a, T> Iterator for Iter<'a, T> {
57      type Item = &'a T;
58
59      fn next(&mut self) -> Option<Self::Item> {
60          self.next.map(|node| {
61              self.next = node.next.as_deref();
62              &node.elem
63          })
64      }
65  }
66
67  #[cfg(test)]
68  mod test {
69      use super::List;
70
71      #[test]
72      fn basics() {
73          let list = List::new();
74          assert_eq!(list.head(), None);
75
76          let list = list.prepend(1).prepend(2).prepend(3);
77          assert_eq!(list.head(), Some(&3));
78
```

```
79          let list = list.tail();
80          assert_eq!(list.head(), Some(&2));
81
82          let list = list.tail();
83          assert_eq!(list.head(), Some(&1));
84
85          let list = list.tail();
86          assert_eq!(list.head(), None);
87
88          // Make sure empty tail works
89          let list = list.tail();
90          assert_eq!(list.head(), None);
91      }
92
93      #[test]
94      fn iter() {
95          let list = List::new().prepend(1).prepend(2).prepend(3);
96
97          let mut iter = list.iter();
98          assert_eq!(iter.next(), Some(&3));
99          assert_eq!(iter.next(), Some(&2));
100         assert_eq!(iter.next(), Some(&1));
101     }
102 }
```

2.5 不咋样的双端队列

2.5.1 数据布局和基本操作

Fact 2.5.1

让 **Rc** 可变，就需要使用 **RefCell** 的配合

Fact 2.5.2

Rc 最怕的就是引用形成循环，而双向链表恰恰如此。因此，当使用默认的实现来 **drop** 我们的链表时，两个节点会将各自的引用计数减少到 1，然后就不会继续减少，最终造成内存泄漏。
所以，这里最好的实现就是将每个节点 **pop** 出去，直到获得 **None**

2.5.2 Peek

Fact 2.5.3

borrow 的定义:

```
fn borrow<'a>(&'a self) -> Ref<'a, T>
fn borrow_mut<'a>(&'a self) -> RefMut<'a, T>
```

Fact 2.5.4

这里返回的并不是 `&T` 或 `&mut T`，而是一个 **Ref** 和 **RefMut**，它们就是在借用到的引用外包裹了一层。而且 **Ref** 和 **RefMut** 分别实现了 **Deref** 和 **DerefMut**，在绝大多数场景中，我们都可以像使用 `&T` 一样去使用它们。
`Ref::map` 是一个将 **Ref** 映射到另一个 **Ref** 的函数

```
1           fn map<U, F>(orig: Ref<'b, T>, f: F) -> Ref<'b, U>
2               where F: FnOnce(&T) -> &U,
3                     U: ?Sized
```

2.5.3 基本操作的对称镜像

Fact 2.5.5

双向链表的对称操作:

- tail <-> head
- next <-> prev
- front -> back

2.5.4 迭代器

Fact 2.5.6

由于是转移所有权，因此 **IntoIter** 一直都是最好实现的

Fact 2.5.7

关于双向链表，有一个有趣的事实，它不仅可以从前向后迭代，还能反过来。
DoubleEndedIterator，它继承自 **Iterator**(通过 **supertrait**)，因此意味着要实现该特征，首先需要实现 **Iterator**。
只要为 **DoubleEndedIterator** 实现 **next_back** 方法，就可以支持双向迭代了:**Iterator** 的 **next** 方法从前往后，而 **next_back** 从后向前。

Fact 2.5.8

map_split 是 **Rust** 中一个用于将借用的值拆分为两部分的函数。其主要作用是允许你在处理一个整体对象时，将其安全地分割成两个独立的部分，从而能够更灵活地操作数据结构。
具体来说，**map_split** 函数接收一个对类型 **T** 引用，并通过一个闭包（函数）将其分割成两个部分，分别返回对类型 **U** 和 **V** 的引用。这在处理复杂数据结构时尤其有用，因为它允许你在不违反 **Rust** 的借用规则的前提下，安全地访问和修改数据。

```
            pub fn map_split<U, V, F>(orig: Ref<'b, T>, f: F) -> (Ref<'b, U>, Ref<'b, V>) where
                F: FnOnce(&T) -> (&U, &V),
                U: ?Sized,
                V: ?Sized,
```

2.5.5 最终代码

Fact 2.5.9

我们实现了一个 100% 安全但是功能残缺的双向链表

```
use std::rc::Rc;
use std::cell::{Ref, RefMut, RefCell};

pub struct List<T> {
    head: Link<T>,
    tail: Link<T>,
}

type Link<T> = Option<Rc<RefCell<Node<T>>>>;

struct Node<T> {
    elem: T,
    next: Link<T>,
    prev: Link<T>,
}


impl<T> Node<T> {
    fn new(elem: T) -> Rc<RefCell<Self>> {
        Rc::new(RefCell::new(Node {
            elem: elem,
            prev: None,
            next: None,
        }))
    }
}
```

```
27
28  impl<T> List<T> {
29      pub fn new() -> Self {
30          List { head: None, tail: None }
31      }
32
33      pub fn push_front(&mut self, elem: T) {
34          let new_head = Node::new(elem);
35          match self.head.take() {
36              Some(old_head) => {
37                  old_head.borrow_mut().prev = Some(new_head.clone());
38                  new_head.borrow_mut().next = Some(old_head);
39                  self.head = Some(new_head);
40              }
41              None => {
42                  self.tail = Some(new_head.clone());
43                  self.head = Some(new_head);
44              }
45          }
46      }
47
48      pub fn push_back(&mut self, elem: T) {
49          let new_tail = Node::new(elem);
50          match self.tail.take() {
51              Some(old_tail) => {
52                  old_tail.borrow_mut().next = Some(new_tail.clone());
53                  new_tail.borrow_mut().prev = Some(old_tail);
54                  self.tail = Some(new_tail);
55              }
56              None => {
57                  self.head = Some(new_tail.clone());
58                  self.tail = Some(new_tail);
59              }
60          }
61      }
62
63      pub fn pop_back(&mut self) -> Option<T> {
64          self.tail.take().map(|old_tail| {
65              match old_tail.borrow_mut().prev.take() {
66                  Some(new_tail) => {
67                      new_tail.borrow_mut().next.take();
68                      self.tail = Some(new_tail);
69                  }
70                  None => {
71                      self.head.take();
```

```
                }
            }
            Rc::try_unwrap(old_tail).ok().unwrap().into_inner().elem
        })
    }

    pub fn pop_front(&mut self) -> Option<T> {
        self.head.take().map(|old_head| {
            match old_head.borrow_mut().next.take() {
                Some(new_head) => {
                    new_head.borrow_mut().prev.take();
                    self.head = Some(new_head);
                }
                None => {
                    self.tail.take();
                }
            }
            Rc::try_unwrap(old_head).ok().unwrap().into_inner().elem
        })
    }

    pub fn peek_front(&self) -> Option<Ref<T>> {
        self.head.as_ref().map(|node| {
            Ref::map(node.borrow(), |node| &node.elem)
        })
    }

    pub fn peek_back(&self) -> Option<Ref<T>> {
        self.tail.as_ref().map(|node| {
            Ref::map(node.borrow(), |node| &node.elem)
        })
    }

    pub fn peek_back_mut(&mut self) -> Option<RefMut<T>> {
        self.tail.as_ref().map(|node| {
            RefMut::map(node.borrow_mut(), |node| &mut node.elem)
        })
    }

    pub fn peek_front_mut(&mut self) -> Option<RefMut<T>> {
        self.head.as_ref().map(|node| {
            RefMut::map(node.borrow_mut(), |node| &mut node.elem)
        })
    }

```

```
117     pub fn into_iter(self) -> IntoIter<T> {
118         IntoIter(self)
119     }
120 }
121 
122 impl<T> Drop for List<T> {
123     fn drop(&mut self) {
124         while self.pop_front().is_some() {}
125     }
126 }
127 
128 pub struct IntoIter<T>(List<T>);
129 
130 impl<T> Iterator for IntoIter<T> {
131     type Item = T;
132 
133     fn next(&mut self) -> Option<T> {
134         self.0.pop_front()
135     }
136 }
137 
138 impl<T> DoubleEndedIterator for IntoIter<T> {
139     fn next_back(&mut self) -> Option<T> {
140         self.0.pop_back()
141     }
142 }
143 
144 #[cfg(test)]
145 mod test {
146     use super::List;
147 
148     #[test]
149     fn basics() {
150         let mut list = List::new();
151 
152         // Check empty list behaves right
153         assert_eq!(list.pop_front(), None);
154 
155         // Populate list
156         list.push_front(1);
157         list.push_front(2);
158         list.push_front(3);
159 
160         // Check normal removal
161         assert_eq!(list.pop_front(), Some(3));
```

```
162         assert_eq!(list.pop_front(), Some(2));
163
164         // Push some more just to make sure nothing's corrupted
165         list.push_front(4);
166         list.push_front(5);
167
168         // Check normal removal
169         assert_eq!(list.pop_front(), Some(5));
170         assert_eq!(list.pop_front(), Some(4));
171
172         // Check exhaustion
173         assert_eq!(list.pop_front(), Some(1));
174         assert_eq!(list.pop_front(), None);
175
176         // ---- back -----
177
178         // Check empty list behaves right
179         assert_eq!(list.pop_back(), None);
180
181         // Populate list
182         list.push_back(1);
183         list.push_back(2);
184         list.push_back(3);
185
186         // Check normal removal
187         assert_eq!(list.pop_back(), Some(3));
188         assert_eq!(list.pop_back(), Some(2));
189
190         // Push some more just to make sure nothing's corrupted
191         list.push_back(4);
192         list.push_back(5);
193
194         // Check normal removal
195         assert_eq!(list.pop_back(), Some(5));
196         assert_eq!(list.pop_back(), Some(4));
197
198         // Check exhaustion
199         assert_eq!(list.pop_back(), Some(1));
200         assert_eq!(list.pop_back(), None);
201     }
202
203     #[test]
204     fn peek() {
205         let mut list = List::new();
206         assert!(list.peek_front().is_none());
```

```
207         assert!(list.peek_back().is_none());
208         assert!(list.peek_front_mut().is_none());
209         assert!(list.peek_back_mut().is_none());
210 
211         list.push_front(1); list.push_front(2); list.push_front(3);
212 
213         assert_eq!(&*list.peek_front().unwrap(), &3);
214         assert_eq!(&mut *list.peek_front_mut().unwrap(), &mut 3);
215         assert_eq!(&*list.peek_back().unwrap(), &1);
216         assert_eq!(&mut *list.peek_back_mut().unwrap(), &mut 1);
217     }
218 
219     #[test]
220     fn into_iter() {
221         let mut list = List::new();
222         list.push_front(1); list.push_front(2); list.push_front(3);
223 
224         let mut iter = list.into_iter();
225         assert_eq!(iter.next(), Some(3));
226         assert_eq!(iter.next_back(), Some(1));
227         assert_eq!(iter.next(), Some(2));
228         assert_eq!(iter.next_back(), None);
229         assert_eq!(iter.next(), None);
230     }
231 }
```

2.6 不错的 unsafe 队列

2.6.1 数据布局

Fact 2.6.1

Box 并没有实现 **Copy** 特征

Fact 2.6.2

the lifetime must be valid for the lifetime 'a as defined on the impl
意思是说生命周期至少要和**'a** 一样长

Fact 2.6.3

自引用 (self-referential) 是指在结构体内部持有对自身其他部分的引用，这在 **Rust** 中是一个常见问题，因为 **Rust** 的借用检查器无法安全地处理这种情况。**Rust** 的安全性和所有权系统使得创建自引用结构体变得非常困难。这是因为：

- 当结构体的字段持有对同一结构体中其他字段的引用时，在移动或克隆结构体时引用会失效。
- **Rust** 无法推断出在这种情况下如何正确地管理内存，从而避免数据竞争和悬挂引用。

为了解决自引用问题，可以使用裸指针，就不会形成自引用的问题，也不再违反 **Rust** 严苛的借用规则。

2.6.2 基本操作

Fact 2.6.4

`*mut` 裸指针通过 `std::ptr::null_mut` 函数可以获取一个 **null**，当然，还可以使用 0 as `*mut _`。

Fact 2.6.5

通过强制转换 **Coercions**，将一个普通的引用变成裸指针。

```
let raw_tail: *mut _ = &mut *new_tail;
```

Fact 2.6.6

当使用裸指针时，一些 **Rust** 提供的便利条件也将不复存在，例如由于不安全性的存在，裸指针需要我们手动去解引用 (**deref**)

Fact 2.6.7

不是所有的裸指针代码都有 **unsafe** 的身影。原因在于：**创建原生指针是安全的行为，而解引用原生指针才是不安全的行为**。

2.6.3 Miri

Fact 2.6.8

Miri 目前只能在 **nightly Rust** 上安装

```
$ rustup +nightly component add miri
```

Fact 2.6.9

是一种临时性的规则运用，如果你不想每次都使用 **+nightly-2022-01-21**，可以使用 `rustup override set` 命令对当前项目的 **Rust** 版本进行覆盖

Fact 2.6.10

miri 可以生成 Rust 的中间层表示 MIR，对于编译器来说，我们的 Rust 代码首先会被编译为 MIR，然后再提交给 LLVM 进行处理。
可以通过 rustup component add miri 来安装它，并通过 cargo miri 来使用，同时还可以使用 cargo miri test 来运行测试代码。
miri 可以帮助我们检查常见的未定义行为 (UB = Undefined Behavior)，以下列出了一部分:

- 内存越界检查和内存释放后再使用 (use-after-free)
- 使用未初始化的数据
- 数据竞争
- 内存对齐问题

UB 检测是必须的，因为它发生在运行时，因此很难发现，如果 miri 能在编译期检测出来，那自然是最好不过的。

Fact 2.6.11

miri 的使用很简单:

```
$ cargo +nightly miri test
```

2.6.4 栈借用

Definition 2.6.12: 栈借用 (Stacked Borrows)

栈借用 (Stacked Borrows) 是一种用于确保 **Rust** 编程语言内存安全的借用机制。它通过追踪每个借用的栈帧，确保变量在生命周期内不会被不安全地访问或修改。这个机制利用了一种栈 (**stack**) 来管理借用的引用，从而避免数据竞争和未定义行为。
具体来说，每次变量被借用时，都会记录到这个“栈”中。这样在同一时间内，只能有一个可变引用或多个不可变引用，从而严格遵守 **Rust** 的借用规则。这种机制特别有助于在多线程编程和复杂的数据结构中保持内存安全。

Definition 2.6.13: 指针混叠 (Pointer Aliasing)

对于同一块内存，存在多个指针指向，或者说，两个指针指向的内存存在重叠。

Claim: 指针混叠有什么问题?

Rust 一方面通过借用规则来保证安全，给程序加了很多限制，让我们写起来很费劲。另一方面因为这些限制的存在，**Rust** 编译器可以对程序做很多优化。
比如当一个值的引用是不可变引用时，编译器知道无论怎么操作，最终这块内存的内容不会发生改变。那么再次读取时就只需读 **cache**，而不需要去内存中读。
编译器可以跟踪所有引用的情况，从而在可以的时候做出优化。没有了不可变的保证，编译器的优化很可能导致问题，程序产生未定义行为 (**UB**)。

Claim: Rust 是怎么解决这个问题的?

借用栈 (borrow stack)。

Definition 2.6.14: 借用 (Reborrow)

指的是在已经存在对某个对象的引用的情况下，创建该对象的新引用。这通常用于延长引用的生命周期或创建具有不同可变性的引用（例如，从可变引用创建不可变引用）。
重借是 **Rust** 所有权和借用系统中的一个关键概念，它确保了内存安全并防止数据竞争。当你重借一个引用时，原始引用会暂时被挂起，而使用新的引用。

Fact 2.6.15

这个栈的顶部借用就是当前正在使用 (**live**) 的借用，而它清晰的知道在它使用的期间不会发生混叠。当对一个指针进行再借用时，新的借用会被插入到栈的顶部，并变成 **live** 状态。如果要将一个旧的指针变成 **live**，就需要将借用栈上在它之前的借用全部弹出 (**pop**)。
只要保证使用时只能使用栈顶的指针，指针混叠就不会出现问题。反过来，这样是不允许的。
这一切都是 **safe Rust** 所保证的，使用顺序不对，就不让过编译。
通过栈借用的方式，我们保证了尽管存在多个再借用，但是在同一个时间，只会有一个可变引用访问目标内存，再也不用担心指针混叠的问题了。只要不去访问一个已经被弹出借用栈的指针，就会非常安全!

Claim: 那在 unsafe 中呢?

对于引用，在 unsafe 中也是一样的规则。
但是如果出现裸指针，就无法保证了。
在 unsafe 中正需要你自己来保证正确性，不留神的话就会写出 UB 来。

Definition 2.6.16: 什么是 UB(Undefined Behavior)?

按照 **Rust** 的规则写代码，**Rust** 保证给你正确的程序。
超出 **Rust** 的规则写代码，**Rust** 不保证给你写出什么东西来。

Claim: 有没有什么办法检测不小心写出的 UB?

这就是 miri 做的事。

Claim: 为了保证不出 UB，应该遵守哪些规则?

1. 对于引用和该引用产生的指针
   可以想象引用和其产生的指针之间，遵守借用栈的规则。对于不同引用产生的指针，**Miri** 可以区分出各自属于哪个引用（用 **Miri** 的术语，属于哪个 **tag**），从而遵守借用栈规则。

2. 对于同一引用产生的多个指针
   当一个引用产生一个指针，然后用该指针产生更多指针时，这些指针之间可以不区分先后顺序随意使用。它们共享同一个 **tag**。**Miri** 表示没意见。因为编译器只会根据引用来调整优化，裸指针的信息它无法获知。因此你一旦获得一个引用的指针，就可以用它生成更多指针来随意使用。因此当我们需要使用裸指针时，最好在用引用生成裸指针后，就一直只使用裸指针，不要和引用混在一起用。最后需要返回引用时，再将裸指针转回引用。

3. 对于不可变引用
   对于不可变引用，可以随意产生更多的不可变引用，使用顺序也没有限制。同时，不可变引用不能产生可变引用。当一个不可变引用入栈，需要保证：在其出栈时，其引用的值不会发生任何变化。换到指针，也是一样的。强行转换成可变指针，编译可通过，但 miri 会有意见。但如果你不真正修改值，miri 可以网开一面。

4. 对于数组
   普通情况下，**Rust** 无法通过下标来判断对数组不同部分的借用是不相交的。不允许对数组进行两次可变引用。但是 **Rust** 提供了一个方法，将不安全操作封装成一个安全方法 **split_at_mut**，将一个切片拆成两个切片，通过 **split_at_mut**，把一个可变切片引用，变成了多个可变切片引用。

5. 对于 **Cell/RefCell**
   由于这两个具有内部可变性，因此其不可变引用可以修改其内部的值。**Cell/RefCell** 是两个特例。当出现这两个东西时，**Rust** 知道它们没有不变性保证，不会去优化它们。因为其不可变引用实际上可变的特性，其栈式借用也有些不一样的地方。对于 `&UnsafeCell<T>`，其实际上是可变的，而且可以无限复制 (因为在名义上是一个不可变引用)，其行为和 `*mut T` 一样。从一个引用产生的多个裸指针可以随意顺序使用。

Fact 2.6.17

Once a shared reference is on the borrow-stack, everything that gets pushed on top of it only has read permissions.

Fact 2.6.18

在代码中混杂地使用 Box 和裸指针，很容易写出 UB 的代码。因此后面的实现中会尝试不去操作 Box，完全使用裸指针。

Fact 2.6.19

Miri 还提供了试验性的模式，可以通过环境的方式来开启该模式:

```
cargo +nightly miri test
```

Fact 2.6.20

使用裸指针，应该遵守一个原则：一旦开始使用裸指针，就要尝试着只使用它。
现在，我们依然希望在接口中使用安全的引用去构建一个安全的抽象，例如在函数参数中使用引用而不是裸指针，这样我们的用户就无需操心 **unsafe** 的问题。
为此，我们需要做以下事情:

- 在开始时，将输入参数中的引用转换成裸指针
- 在函数体中只使用裸指针
- 返回之前，将裸指针转换成安全的指针

但是由于数据结构中的字段都是私有的，无需暴露给用户，因此无需这么麻烦，直接使用裸指针即可。
事实上，一个依然存在的问题就是还在继续使用 **Box**, 它会告诉编译器: hey，这个看上去很像是 `&mut`，因为它唯一的持有那个指针。
但是我们在链表中一直使用的裸指针是指向 **Box** 的内部，所以无论何时我们通过正常的方式访问 **Box**，我们都有可能让该裸指针的再借用变得不合法。

Claim: 总结

- **Rust** 通过借用栈来处理再借用
- 只有栈顶的元素是处于 **live** 状态的 (被借用)
- 当访问栈顶下面的元素时，该元素会变为 **live**，而栈顶元素会被弹出 (**pop**)
- 从借用栈中弹出的元素无法再被借用
- 借用检查器会保证我们的安全代码遵守以上规则
- **Miri** 可以在一定程度上保证裸指针在运行时也遵循以上规则

2.6.5 测试栈借用

Fact 2.6.21

Miri 为何可以一定程度上提前发现这些 **UB** 问题? 因为它会去获取 **rustc** 对我们的程序最原生、且没有任何优化的视角，然后对看到的内容进行解释和跟踪。只要这个过程能够开始，那这个解决方法就相当有效，但是问题来了，该如何让这个过程开始? 要知道 **Miri** 和 **rustc** 是不可能去逐行分析代码中的所有行为的，这样做的结果就是编译时间大大增加!
因此我们需要使用测试用例来让程序中可能包含 **UB** 的代码路径被真正执行到，当然，就算你这么做了，也不能完全依赖 **Miri**。既然是分析，就有可能遗漏，也可能误杀友军。

Fact 2.6.22

Miri 对于这种裸指针派生是相当纵容的：当它们都共享同一个借用时 (**borrowing**, 也可以用 **Miri** 的称呼:**tag**)。
当代码足够简单时，编译器是有可能介入跟踪所有派生的裸指针，并尽可能去优化它们的。但是这套规则比引用的那套脆弱得多！

Fact 2.6.23

Rust 不允许我们对数组的不同元素进行单独的借用，注意到提示了吗？可以使用`.split_at_mut(position)`来将一个数组分成多个部分

Fact 2.6.24

将一个切片转换成指针，那指针是否还拥有访问整个切片的权限。

Fact 2.6.25

在借用栈中，一个不可变引用，它上面的所有引用 (在它之后被推入借用栈的引用) 都只能拥有只读的权限。

Fact 2.6.26

Rust 中用于内部可变性的核心原语 (primitive)。
如果你拥有一个引用 `&T`，那一般情况下, **Rust** 编译器会基于 `&T` 指向不可变的数据这一事实来进行相关的优化。通过别名或者将 `&T` 强制转换成 `&mut T` 是一种 **UB** 行为。
而 `UnsafeCell<T>` 移除 `&T` 的不可变保证：一个不可变引用 `&UnsafeCell<T>` 指向一个可以改变的数据。这就是内部可变性。

2.6.6 数据布局 2

Fact 2.6.27

将安全的指针 `&`、`&mut` 和 **Box** 跟不安全的裸指针 `*mut` 和 ***const** 混用是 **UB** 的根源之一，原因是安全指针会引入额外的约束，但是裸指针并不会遵守这些约束。

Fact 2.6.28

请大家牢记：当使用裸指针时，**Option** 对我们是相当不友好的，所以这里不再使用。

Fact 2.6.29

`pub fn into_raw(b: Box<T>) -> *mut T`
消费掉 **Box**(拿走所有权)，返回一个裸指针。该指针会被正确的对齐且不为 **null** 在调用该函数后，调用者需要对之前被 **Box** 所管理的内存负责，特别地，调用者需要正确的清理 **T** 并释放相应的内存。最简单的方式是通过 `Box::from_raw` 函数将裸指针再转回到 **Box**，然后 **Box** 的析构器就可以自动执行清理了。
注意：这是一个关联函数，因此 `b.into_raw()` 是不正确的，我们得使用 `Box::into_raw(b)`。因此该函数不会跟内部类型的同名方法冲突。

Fact 2.6.30

从安全的东东开始，将其转换成裸指针，最后再将裸指针转回安全的东东以实现安全的 **drop**。

```
let x = Box::new(String::from("Hello"));
let ptr = Box::into_raw(x);
let x = unsafe { Box::from_raw(ptr) };
```

2.6.7 额外的操作

Definition 2.6.31: PhantomData

PhantomData 是零大小 zero sized 的类型。
在你的类型中添加一个 `PhantomData<T>` 字段，可以告诉编译器你的类型对 **T** 进行了使用，虽然并没有。说白了，就是让编译器不再给出 **T** 未被使用的警告或者错误。

Fact 2.6.32

大概最适用于 **PhantomData** 的场景就是一个结构体拥有未使用的生命周期，典型的就是在 **unsafe** 中使用。

Fact 2.6.33

你必须要遵循混叠 (**Aliasing**) 的规则，原因是返回的生命周期**'a** 只是任意选择的，并不能代表数据真实的生命周期。特别的，在这段生命周期的过程中，指针指向的内存区域绝不能被其它指针所访问。

2.6.8 最终代码

```
use std::ptr;

pub struct List<T> {
    head: Link<T>,
    tail: *mut Node<T>,
}

type Link<T> = *mut Node<T>;

```

```
struct Node<T> {
    elem: T,
    next: Link<T>,
}

pub struct IntoIter<T>(List<T>);

pub struct Iter<'a, T> {
    next: Option<&'a Node<T>>,
}

pub struct IterMut<'a, T> {
    next: Option<&'a mut Node<T>>,
}

impl<T> List<T> {
    pub fn new() -> Self {
        List { head: ptr::null_mut(), tail: ptr::null_mut() }
    }
    pub fn push(&mut self, elem: T) {
        unsafe {
            let new_tail = Box::into_raw(Box::new(Node {
                elem: elem,
                next: ptr::null_mut(),
            }));

            if !self.tail.is_null() {
                (*self.tail).next = new_tail;
            } else {
                self.head = new_tail;
            }

            self.tail = new_tail;
        }
    }
    pub fn pop(&mut self) -> Option<T> {
        unsafe {
            if self.head.is_null() {
                None
            } else {
                let head = Box::from_raw(self.head);
                self.head = head.next;

                if self.head.is_null() {
                    self.tail = ptr::null_mut();
```

```
                }

                Some(head.elem)
            }
        }
    }

    pub fn peek(&self) -> Option<&T> {
        unsafe {
            self.head.as_ref().map(|node| &node.elem)
        }
    }

    pub fn peek_mut(&mut self) -> Option<&mut T> {
        unsafe {
            self.head.as_mut().map(|node| &mut node.elem)
        }
    }

    pub fn into_iter(self) -> IntoIter<T> {
        IntoIter(self)
    }

    pub fn iter(&self) -> Iter<'_, T> {
        unsafe {
            Iter { next: self.head.as_ref() }
        }
    }

    pub fn iter_mut(&mut self) -> IterMut<'_, T> {
        unsafe {
            IterMut { next: self.head.as_mut() }
        }
    }
}

impl<T> Drop for List<T> {
    fn drop(&mut self) {
        while let Some(_) = self.pop() { }
    }
}

impl<T> Iterator for IntoIter<T> {
    type Item = T;
    fn next(&mut self) -> Option<Self::Item> {
```

```
100         self.0.pop()
101     }
102 }
103
104 impl<'a, T> Iterator for Iter<'a, T> {
105     type Item = &'a T;
106
107     fn next(&mut self) -> Option<Self::Item> {
108         unsafe {
109             self.next.map(|node| {
110                 self.next = node.next.as_ref();
111                 &node.elem
112             })
113         }
114     }
115 }
116
117 impl<'a, T> Iterator for IterMut<'a, T> {
118     type Item = &'a mut T;
119
120     fn next(&mut self) -> Option<Self::Item> {
121         unsafe {
122             self.next.take().map(|node| {
123                 self.next = node.next.as_mut();
124                 &mut node.elem
125             })
126         }
127     }
128 }
129
130 #[cfg(test)]
131 mod test {
132     use super::List;
133     #[test]
134     fn basics() {
135         let mut list = List::new();
136
137         // Check empty list behaves right
138         assert_eq!(list.pop(), None);
139
140         // Populate list
141         list.push(1);
142         list.push(2);
143         list.push(3);
144
```

```
145         // Check normal removal
146         assert_eq!(list.pop(), Some(1));
147         assert_eq!(list.pop(), Some(2));
148 
149         // Push some more just to make sure nothing's corrupted
150         list.push(4);
151         list.push(5);
152 
153         // Check normal removal
154         assert_eq!(list.pop(), Some(3));
155         assert_eq!(list.pop(), Some(4));
156 
157         // Check exhaustion
158         assert_eq!(list.pop(), Some(5));
159         assert_eq!(list.pop(), None);
160 
161         // Check the exhaustion case fixed the pointer right
162         list.push(6);
163         list.push(7);
164 
165         // Check normal removal
166         assert_eq!(list.pop(), Some(6));
167         assert_eq!(list.pop(), Some(7));
168         assert_eq!(list.pop(), None);
169     }
170 
171     #[test]
172     fn into_iter() {
173         let mut list = List::new();
174         list.push(1); list.push(2); list.push(3);
175 
176         let mut iter = list.into_iter();
177         assert_eq!(iter.next(), Some(1));
178         assert_eq!(iter.next(), Some(2));
179         assert_eq!(iter.next(), Some(3));
180         assert_eq!(iter.next(), None);
181     }
182 
183     #[test]
184     fn iter() {
185         let mut list = List::new();
186         list.push(1); list.push(2); list.push(3);
187 
188         let mut iter = list.iter();
189         assert_eq!(iter.next(), Some(&1));
```

```
190         assert_eq!(iter.next(), Some(&2));
191         assert_eq!(iter.next(), Some(&3));
192         assert_eq!(iter.next(), None);
193     }
194
195     #[test]
196     fn iter_mut() {
197         let mut list = List::new();
198         list.push(1); list.push(2); list.push(3);
199
200         let mut iter = list.iter_mut();
201         assert_eq!(iter.next(), Some(&mut 1));
202         assert_eq!(iter.next(), Some(&mut 2));
203         assert_eq!(iter.next(), Some(&mut 3));
204         assert_eq!(iter.next(), None);
205     }
206
207     #[test]
208     fn miri_food() {
209         let mut list = List::new();
210
211         list.push(1);
212         list.push(2);
213         list.push(3);
214
215         assert!(list.pop() == Some(1));
216         list.push(4);
217         assert!(list.pop() == Some(2));
218         list.push(5);
219
220         assert!(list.peek() == Some(&3));
221         list.push(6);
222         list.peek_mut().map(|x| *x *= 10);
223         assert!(list.peek() == Some(&30));
224         assert!(list.pop() == Some(30));
225
226         for elem in list.iter_mut() {
227             *elem *= 100;
228         }
229
230         let mut iter = list.iter();
231         assert_eq!(iter.next(), Some(&400));
232         assert_eq!(iter.next(), Some(&500));
233         assert_eq!(iter.next(), Some(&600));
234         assert_eq!(iter.next(), None);
```

```
235         assert_eq!(iter.next(), None);
236
237         assert!(list.pop() == Some(400));
238         list.peek_mut().map(|x| *x *= 10);
239         assert!(list.peek() == Some(&5000));
240         list.push(7);
241
242         // Drop it on the ground and let the dtor exercise itself
243     }
244 }
```

2.7 生产级的双向 unsafe 队列

2.7.1 数据布局

Fact 2.7.1

传统的方法是对堆栈的简单扩展——只需将头部和尾部指针存储在堆栈上。这很好，但它有一个缺点：极端情况。现在我们的列表有两个边缘，这意味着极端情况的数量增加了一倍。很容易忘记一个并有一个严重的错误。

```
[ptr, ptr] <-> (ptr, A, ptr) <-> (ptr, B, ptr)
  ↑                                       ↑
  +---------------------------------------+
```

Fact 2.7.2

虚拟节点方法试图通过在我们的列表中添加一个额外的节点来消除这些极端情况，该节点不包含任何数据，但将两端链接成一个环。

```
[ptr] -> (ptr, ?DUMMY?, ptr) <-> (ptr, A, ptr) <-> (ptr, B, ptr)
           ↑                                                ↑
           +------------------------------------------------+
```

Fact 2.7.3

通过执行此操作，每个节点始终具有指向列表中上一个和下一个节点的实际指针。即使你从列表中删除了最后一个元素，你最终也只是拼接了虚拟节点以指向它自己。

```
[ptr] -> (ptr, ?DUMMY?, ptr)
           ↑              ↑
           +--------------+
```

Fact 2.7.4

对于像 **Rust** 这样的语言来说，这些虚拟节点方案的问题确实超过了便利性，所以我们将坚持传统的布局。

2.7.2 型变与子类型

Fact 2.7.5

建造 Rust collections 时，有这五个可怕的难题：

- Variance
- Drop Check
- NonNull Optimizations
- The isize::MAX Allocation Rule
- Zero-Sized Types

Fact 2.7.6

基本上子类型并不总是安全的。特别是，当涉及可变引用时，它就更不安全了，。因为你可能会使用诸如 `mem::swap` 的东西

Fact 2.7.7

可变引用是 **invariant**(不变的)，这意味着它们会阻止对泛型参数子类型化。因此，为了安全起见，`&mut T` 在 **T** 上是不变的，并且 `Cell<T>` 在 **T** 上也是不变的 (因为内部可变性)，因为 `&Cell<T>` 本质上就像 `&mut T`。

Fact 2.7.8

几乎所有不是 **invariant** 的东西都是 **covariant**(协变的) ，这意味着子类型可以正常工作（也有 **contravariant**(逆变的) 的类型使子类型倒退，但它们真的很少见，没有人喜欢它们，所以我不会再提到它们)。

Fact 2.7.9

由于 **Rust** 的所有权系统，**Vec<T>** 在语义上等同于 **T**，这意味着它可以安全地保持 **covariant**(协变的)

Fact 2.7.10

***mut T** 是 **invariant**(不变的)，因为它很有可能被” 作为”`&mut T` 使用。

Fact 2.7.11

与 **mut T** 不同，**NonNull** 在 **T** 上是 **covariant**(协变的)。这使得使用 **NonNull** 构建 **covariant**(协变的)* 类型成为可能，但如果在不应该是 **covariant**(协变的) 的地方中使用，则会带来不健全的风险。

Fact 2.7.12

NonNull 只是滥用了 ***const T** 是 **covariant**(协变的) 这一事实，并将其存储起来。这就是 **Rust** 中集合的协变方式！

2.7.3 基础结构

Fact 2.7.13

PhantomData 是一种奇怪的类型，没有字段，所以你只需说出它的类型名称就能创建一个。

2.7.4 恐慌与安全

Fact 2.7.14

在默认情况下，恐慌会被 **unwinding**。**unwind** 只是" 让每个函数立即返回" 的一种花哨说法。
当函数返回时，析构函数会运行，而且可以捕获 **unwind**。在这两种情况下，代码都可能在恐慌发生后继续运行，因此我们必须非常小心，确保我们的不安全的集合在恐慌发生时始终处于某种一致的状态，因为每次恐慌都是隐式的提前返回！

Fact 2.7.15

调试断言哪行在某种意义上更糟糕，因为它可能将一个小问题升级为关键问题。

Fact 2.7.16

Rust 的集合所有权和借用系统的" 杀手级应用" 之一：如果一个操作需要一个 `&mut Self`，那么我们就能保证对我们的集合拥有独占访问权，而且我们可以暂时破坏 **invariants**(不可变性)，因为我们知道没有人能偷偷摸摸地破坏它。

2.7.5 无聊的组合

Fact 2.7.17

双端迭代器 (**DoubleEndedIterator**) 是 **Rust** 中的一种特殊迭代器，它不仅可以从前向后遍历集合，还可以从后向前遍历。这种迭代器非常适合需要双向遍历的场景。
在实现双端迭代器时，需要实现 **DoubleEndedIterator** trait，该 trait 继承自 **Iterator** trait，并额外要求实现 **next_back** 方法。**next_back** 方法用于从后向前返回集合中的下一个元素。

Fact 2.7.18

精确大小迭代器 (**ExactSizeIterator**) 是 **Rust** 中的一种迭代器，它不仅实现了 **Iterator** trait，还提供了一个额外的方法 **len**，用于返回迭代器中剩余元素的数量。这个特性使得精确大小迭代器在处理需要知道确切元素数量的场景中非常有用。
要实现 **ExactSizeIterator**，需要在实现 **Iterator** trait 的基础上，额外实现 **len** 方法。

2.7.6 其它特征

Fact 2.7.19

在计算哈希值时，确保将集合的长度包含在内是非常重要的。这可以避免前缀碰撞。

2.7.7 测试

2.7.8 Send,Sync 和编译测试

概念	定义	默认状态	用户操作
Opt-in	某个功能默认关闭，用户需主动启用。	关闭	用户选择加入
Opt-out	某个功能默认开启，用户需主动禁用。	开启	用户选择退出
Built-in	某个功能是语言原生支持的，无需额外安装或配置。	原生支持	无需额外操作
Built-out	某个功能通过外部库或扩展实现，或默认自动实现某些特性（较少见）。	非原生	无需操作（或显式禁用）

Table 2.1: Opt-in、Opt-out、Built-in 和 Built-out 概念的对比

Fact 2.7.20

unsafe 的 **Opt-in Built-out** 特征 (**OIBITs**): **Send** 和 **Sync**，它们实际上是 (**opt-out**) 和 (**built-out**)。

Fact 2.7.21

Send 表示你的类型可以安全地发送到另一个线程。
Sync 表示你的类型可以在线程间安全共享 (`&Self: Send`, 当且仅当 `&T` 是 **Send** 时，**T** 是 **Sync** 的)

Fact 2.7.22

Send 和 **Sync** 是 **Rust** 的并发故事的基础。因此，存在大量的特殊工具来使它们正常工作。首先，它们是不安全的 **Trait**, 这意味着它们的实现是不安全的，而其他不安全的代码可以认为它们是正确实现的。由于它们是标记特性 (它们没有像方法那样的相关项目)，正确实现仅仅意味着它们具有实现者应该具有的内在属性。不正确地实现 **Send** 或 **Sync** 会导致未定义行为。

Fact 2.7.23

Send 和 **Sync** 也是自动派生的 **Trait**。这意味着，与其它 **Trait** 不同，如果一个类型完全由 **SendSync** 类型组成，那么它就 **Send** 或 **Sync**。几乎所有的基本数据类型都是 **Send** 和 **Sync**，因此，几乎所有你将与之交互的类型都是 **Send** 和 **Sync**。

Fact 2.7.24

主要的例外情况包括:

- 原始指针既不是 **Send** 也不是 **Sync**(因为它们没有安全防护)
- **UnsafeCell** 不是 **Sync** 的 (因此 **Cell** 和 **RefCell** 也不是)
- **Rc** 不是 **Send** 或 **Sync** 的 (因为 **Refcount** 是共享的、不同步的)

Fact 2.7.25

在 **Rust** 中，**compile_fail** 是一个用于文档测试的注释，它表示代码块应该无法编译。它通常用于验证某些代码在特定情况下会产生编译错误。
可以在 **compile_fail** 后面指定一个错误代码，以确保代码块因特定错误而无法编译。这在验证特定错误时非常有用。(但这只适用于 nightly，在 not-nightly 版本运行时，它将被默默忽略)。
使用 **compile_fail** 可以帮助你在编写文档和测试时确保代码的正确性，并验证某些代码在特定情况下会产生编译错误。这对于编写安全和可靠的 **Rust** 代码非常重要。

2.7.9 实现游标

Fact 2.7.26

游标用于遍历和操作链表。

Fact 2.7.27

一个非常重要的注意事项：这些方法必须通过 `&mut self` 借用我们的游标，并且结果必须与借用绑定。我们不能让用户获得可变引用的多个副本，也不能让他们在持有该引用的情况下使用我们的 API!

2.7.10 测试游标

Fact 2.7.28

```
cargo test
cargo +nightly miri test
cargo clippy
cargo fmt
```

2.7.11 最终代码

```rust
use std::cmp::Ordering;
use std::fmt::{self, Debug};
use std::hash::{Hash, Hasher};
use std::iter::FromIterator;
use std::marker::PhantomData;
use std::ptr::NonNull;

pub struct LinkedList<T> {
    front: Link<T>,
    back: Link<T>,
    len: usize,
    _boo: PhantomData<T>,
}

type Link<T> = Option<NonNull<Node<T>>>;

struct Node<T> {
    front: Link<T>,
    back: Link<T>,
    elem: T,
}

pub struct Iter<'a, T> {
    front: Link<T>,
    back: Link<T>,
    len: usize,
    _boo: PhantomData<&'a T>,
}

pub struct IterMut<'a, T> {
    front: Link<T>,
    back: Link<T>,
    len: usize,
    _boo: PhantomData<&'a mut T>,
}

pub struct IntoIter<T> {
    list: LinkedList<T>,
}

pub struct CursorMut<'a, T> {
    list: &'a mut LinkedList<T>,
    cur: Link<T>,
```

```
    index: Option<usize>,
}

impl<T> LinkedList<T> {
    pub fn new() -> Self {
        Self {
            front: None,
            back: None,
            len: 0,
            _boo: PhantomData,
        }
    }

    pub fn push_front(&mut self, elem: T) {
        // SAFETY: it's a linked-list, what do you want?
        unsafe {
            let new = NonNull::new_unchecked(Box::into_raw(Box::new(Node {
                front: None,
                back: None,
                elem,
            })));
            if let Some(old) = self.front {
                // Put the new front before the old one
                (*old.as_ptr()).front = Some(new);
                (*new.as_ptr()).back = Some(old);
            } else {
                // If there's no front, then we're the empty list and need
                // to set the back too.
                self.back = Some(new);
            }
            // These things always happen!
            self.front = Some(new);
            self.len += 1;
        }
    }

    pub fn push_back(&mut self, elem: T) {
        // SAFETY: it's a linked-list, what do you want?
        unsafe {
            let new = NonNull::new_unchecked(Box::into_raw(Box::new(Node {
                back: None,
                front: None,
                elem,
            })));
            if let Some(old) = self.back {
```

```
89                  // Put the new back before the old one
90                  (*old.as_ptr()).back = Some(new);
91                  (*new.as_ptr()).front = Some(old);
92              } else {
93                  // If there's no back, then we're the empty list and need
94                  // to set the front too.
95                  self.front = Some(new);
96              }
97              // These things always happen!
98              self.back = Some(new);
99              self.len += 1;
100         }
101     }
102
103     pub fn pop_front(&mut self) -> Option<T> {
104         unsafe {
105             // Only have to do stuff if there is a front node to pop.
106             self.front.map(|node| {
107                 // Bring the Box back to life so we can move out its value and
108                 // Drop it (Box continues to magically understand this for us).
109                 let boxed_node = Box::from_raw(node.as_ptr());
110                 let result = boxed_node.elem;
111
112                 // Make the next node into the new front.
113                 self.front = boxed_node.back;
114                 if let Some(new) = self.front {
115                     // Cleanup its reference to the removed node
116                     (*new.as_ptr()).front = None;
117                 } else {
118                     // If the front is now null, then this list is now empty!
119                     self.back = None;
120                 }
121
122                 self.len -= 1;
123                 result
124                 // Box gets implicitly freed here, knows there is no T.
125             })
126         }
127     }
128
129     pub fn pop_back(&mut self) -> Option<T> {
130         unsafe {
131             // Only have to do stuff if there is a back node to pop.
132             self.back.map(|node| {
133                 // Bring the Box front to life so we can move out its value and
```

```
                // Drop it (Box continues to magically understand this for us).
                let boxed_node = Box::from_raw(node.as_ptr());
                let result = boxed_node.elem;

                // Make the next node into the new back.
                self.back = boxed_node.front;
                if let Some(new) = self.back {
                    // Cleanup its reference to the removed node
                    (*new.as_ptr()).back = None;
                } else {
                    // If the back is now null, then this list is now empty!
                    self.front = None;
                }

                self.len -= 1;
                result
                // Box gets implicitly freed here, knows there is no T.
            })
        }
    }

    pub fn front(&self) -> Option<&T> {
        unsafe { self.front.map(|node| &(*node.as_ptr()).elem) }
    }

    pub fn front_mut(&mut self) -> Option<&mut T> {
        unsafe { self.front.map(|node| &mut (*node.as_ptr()).elem) }
    }

    pub fn back(&self) -> Option<&T> {
        unsafe { self.back.map(|node| &(*node.as_ptr()).elem) }
    }

    pub fn back_mut(&mut self) -> Option<&mut T> {
        unsafe { self.back.map(|node| &mut (*node.as_ptr()).elem) }
    }

    pub fn len(&self) -> usize {
        self.len
    }

    pub fn is_empty(&self) -> bool {
        self.len == 0
    }

```

```
179     pub fn clear(&mut self) {
180         // Oh look it's drop again
181         while self.pop_front().is_some() {}
182     }
183 
184     pub fn iter(&self) -> Iter<T> {
185         Iter {
186             front: self.front,
187             back: self.back,
188             len: self.len,
189             _boo: PhantomData,
190         }
191     }
192 
193     pub fn iter_mut(&mut self) -> IterMut<T> {
194         IterMut {
195             front: self.front,
196             back: self.back,
197             len: self.len,
198             _boo: PhantomData,
199         }
200     }
201 
202     pub fn cursor_mut(&mut self) -> CursorMut<T> {
203         CursorMut {
204             list: self,
205             cur: None,
206             index: None,
207         }
208     }
209 }
210 
211 impl<T> Drop for LinkedList<T> {
212     fn drop(&mut self) {
213         // Pop until we have to stop
214         while self.pop_front().is_some() {}
215     }
216 }
217 
218 impl<T> Default for LinkedList<T> {
219     fn default() -> Self {
220         Self::new()
221     }
222 }
223 
```

```rust
impl<T: Clone> Clone for LinkedList<T> {
    fn clone(&self) -> Self {
        let mut new_list = Self::new();
        for item in self {
            new_list.push_back(item.clone());
        }
        new_list
    }
}

impl<T> Extend<T> for LinkedList<T> {
    fn extend<I: IntoIterator<Item = T>>(&mut self, iter: I) {
        for item in iter {
            self.push_back(item);
        }
    }
}

impl<T> FromIterator<T> for LinkedList<T> {
    fn from_iter<I: IntoIterator<Item = T>>(iter: I) -> Self {
        let mut list = Self::new();
        list.extend(iter);
        list
    }
}

impl<T: Debug> Debug for LinkedList<T> {
    fn fmt(&self, f: &mut fmt::Formatter<'_>) -> fmt::Result {
        f.debug_list().entries(self).finish()
    }
}

impl<T: PartialEq> PartialEq for LinkedList<T> {
    fn eq(&self, other: &Self) -> bool {
        self.len() == other.len() && self.iter().eq(other)
    }
}

impl<T: Eq> Eq for LinkedList<T> {}

impl<T: PartialOrd> PartialOrd for LinkedList<T> {
    fn partial_cmp(&self, other: &Self) -> Option<Ordering> {
        self.iter().partial_cmp(other)
    }
}
```

```
269
270 impl<T: Ord> Ord for LinkedList<T> {
271     fn cmp(&self, other: &Self) -> Ordering {
272         self.iter().cmp(other)
273     }
274 }
275
276 impl<T: Hash> Hash for LinkedList<T> {
277     fn hash<H: Hasher>(&self, state: &mut H) {
278         self.len().hash(state);
279         for item in self {
280             item.hash(state);
281         }
282     }
283 }
284
285 impl<'a, T> IntoIterator for &'a LinkedList<T> {
286     type IntoIter = Iter<'a, T>;
287     type Item = &'a T;
288
289     fn into_iter(self) -> Self::IntoIter {
290         self.iter()
291     }
292 }
293
294 impl<'a, T> Iterator for Iter<'a, T> {
295     type Item = &'a T;
296
297     fn next(&mut self) -> Option<Self::Item> {
298         // While self.front == self.back is a tempting condition to check here,
299         // it won't do the right for yielding the last element! That sort of
300         // thing only works for arrays because of "one-past-the-end" pointers.
301         if self.len > 0 {
302             // We could unwrap front, but this is safer and easier
303             self.front.map(|node| unsafe {
304                 self.len -= 1;
305                 self.front = (*node.as_ptr()).back;
306                 &(*node.as_ptr()).elem
307             })
308         } else {
309             None
310         }
311     }
312
313     fn size_hint(&self) -> (usize, Option<usize>) {
```

```
314         (self.len, Some(self.len))
315     }
316 }
317
318 impl<'a, T> DoubleEndedIterator for Iter<'a, T> {
319     fn next_back(&mut self) -> Option<Self::Item> {
320         if self.len > 0 {
321             self.back.map(|node| unsafe {
322                 self.len -= 1;
323                 self.back = (*node.as_ptr()).front;
324                 &(*node.as_ptr()).elem
325             })
326         } else {
327             None
328         }
329     }
330 }
331
332 impl<'a, T> ExactSizeIterator for Iter<'a, T> {
333     fn len(&self) -> usize {
334         self.len
335     }
336 }
337
338 impl<'a, T> IntoIterator for &'a mut LinkedList<T> {
339     type IntoIter = IterMut<'a, T>;
340     type Item = &'a mut T;
341
342     fn into_iter(self) -> Self::IntoIter {
343         self.iter_mut()
344     }
345 }
346
347 impl<'a, T> Iterator for IterMut<'a, T> {
348     type Item = &'a mut T;
349
350     fn next(&mut self) -> Option<Self::Item> {
351         // While self.front == self.back is a tempting condition to check here,
352         // it won't do the right for yielding the last element! That sort of
353         // thing only works for arrays because of "one-past-the-end" pointers.
354         if self.len > 0 {
355             // We could unwrap front, but this is safer and easier
356             self.front.map(|node| unsafe {
357                 self.len -= 1;
358                 self.front = (*node.as_ptr()).back;
```

```
359                 &mut (*node.as_ptr()).elem
360             })
361         } else {
362             None
363         }
364     }
365
366     fn size_hint(&self) -> (usize, Option<usize>) {
367         (self.len, Some(self.len))
368     }
369 }
370
371 impl<'a, T> DoubleEndedIterator for IterMut<'a, T> {
372     fn next_back(&mut self) -> Option<Self::Item> {
373         if self.len > 0 {
374             self.back.map(|node| unsafe {
375                 self.len -= 1;
376                 self.back = (*node.as_ptr()).front;
377                 &mut (*node.as_ptr()).elem
378             })
379         } else {
380             None
381         }
382     }
383 }
384
385 impl<'a, T> ExactSizeIterator for IterMut<'a, T> {
386     fn len(&self) -> usize {
387         self.len
388     }
389 }
390
391 impl<T> IntoIterator for LinkedList<T> {
392     type IntoIter = IntoIter<T>;
393     type Item = T;
394
395     fn into_iter(self) -> Self::IntoIter {
396         IntoIter { list: self }
397     }
398 }
399
400 impl<T> Iterator for IntoIter<T> {
401     type Item = T;
402
403     fn next(&mut self) -> Option<Self::Item> {
```

```
404         self.list.pop_front()
405     }
406 
407     fn size_hint(&self) -> (usize, Option<usize>) {
408         (self.list.len, Some(self.list.len))
409     }
410 }
411 
412 impl<T> DoubleEndedIterator for IntoIter<T> {
413     fn next_back(&mut self) -> Option<Self::Item> {
414         self.list.pop_back()
415     }
416 }
417 
418 impl<T> ExactSizeIterator for IntoIter<T> {
419     fn len(&self) -> usize {
420         self.list.len
421     }
422 }
423 
424 impl<'a, T> CursorMut<'a, T> {
425     pub fn index(&self) -> Option<usize> {
426         self.index
427     }
428 
429     pub fn move_next(&mut self) {
430         if let Some(cur) = self.cur {
431             unsafe {
432                 // We're on a real element, go to its next (back)
433                 self.cur = (*cur.as_ptr()).back;
434                 if self.cur.is_some() {
435                     *self.index.as_mut().unwrap() += 1;
436                 } else {
437                     // We just walked to the ghost, no more index
438                     self.index = None;
439                 }
440             }
441         } else if !self.list.is_empty() {
442             // We're at the ghost, and there is a real front, so move to it!
443             self.cur = self.list.front;
444             self.index = Some(0)
445         } else {
446             // We're at the ghost, but that's the only element... do nothing.
447         }
448     }
```

```
449
450     pub fn move_prev(&mut self) {
451         if let Some(cur) = self.cur {
452             unsafe {
453                 // We're on a real element, go to its previous (front)
454                 self.cur = (*cur.as_ptr()).front;
455                 if self.cur.is_some() {
456                     *self.index.as_mut().unwrap() -= 1;
457                 } else {
458                     // We just walked to the ghost, no more index
459                     self.index = None;
460                 }
461             }
462         } else if !self.list.is_empty() {
463             // We're at the ghost, and there is a real back, so move to it!
464             self.cur = self.list.back;
465             self.index = Some(self.list.len - 1)
466         } else {
467             // We're at the ghost, but that's the only element... do nothing.
468         }
469     }
470
471     pub fn current(&mut self) -> Option<&mut T> {
472         unsafe { self.cur.map(|node| &mut (*node.as_ptr()).elem) }
473     }
474
475     pub fn peek_next(&mut self) -> Option<&mut T> {
476         unsafe {
477             let next = if let Some(cur) = self.cur {
478                 // Normal case, try to follow the cur node's back pointer
479                 (*cur.as_ptr()).back
480             } else {
481                 // Ghost case, try to use the list's front pointer
482                 self.list.front
483             };
484
485             // Yield the element if the next node exists
486             next.map(|node| &mut (*node.as_ptr()).elem)
487         }
488     }
489
490     pub fn peek_prev(&mut self) -> Option<&mut T> {
491         unsafe {
492             let prev = if let Some(cur) = self.cur {
493                 // Normal case, try to follow the cur node's front pointer
```

```
                (*cur.as_ptr()).front
            } else {
                // Ghost case, try to use the list's back pointer
                self.list.back
            };

            // Yield the element if the prev node exists
            prev.map(|node| &mut (*node.as_ptr()).elem)
        }
    }

    pub fn split_before(&mut self) -> LinkedList<T> {
        // We have this:
        //
        //     list.front -> A <-> B <-> C <-> D <- list.back
        //                                 ^
        //                                cur
        //
        //
        // And we want to produce this:
        //
        //     list.front -> C <-> D <- list.back
        //                    ^
        //                   cur
        //
        //
        //    return.front -> A <-> B <- return.back
        //
        if let Some(cur) = self.cur {
            // We are pointing at a real element, so the list is non-empty.
            unsafe {
                // Current state
                let old_len = self.list.len;
                let old_idx = self.index.unwrap();
                let prev = (*cur.as_ptr()).front;

                // What self will become
                let new_len = old_len - old_idx;
                let new_front = self.cur;
                let new_back = self.list.back;
                let new_idx = Some(0);

                // What the output will become
                let output_len = old_len - new_len;
                let output_front = self.list.front;
```

```
                let output_back = prev;

                // Break the links between cur and prev
                if let Some(prev) = prev {
                    (*cur.as_ptr()).front = None;
                    (*prev.as_ptr()).back = None;
                }

                // Produce the result:
                self.list.len = new_len;
                self.list.front = new_front;
                self.list.back = new_back;
                self.index = new_idx;

                LinkedList {
                    front: output_front,
                    back: output_back,
                    len: output_len,
                    _boo: PhantomData,
                }
            }
        } else {
            // We're at the ghost, just replace our list with an empty one.
            // No other state needs to be changed.
            std::mem::replace(self.list, LinkedList::new())
        }
    }

    pub fn split_after(&mut self) -> LinkedList<T> {
        // We have this:
        //
        //     list.front -> A <-> B <-> C <-> D <- list.back
        //                           ^
        //                          cur
        //
        //
        // And we want to produce this:
        //
        //     list.front -> A <-> B <- list.back
        //                           ^
        //                          cur
        //
        //
        //    return.front -> C <-> D <- return.back
        //
```

```
584         if let Some(cur) = self.cur {
585             // We are pointing at a real element, so the list is non-empty.
586             unsafe {
587                 // Current state
588                 let old_len = self.list.len;
589                 let old_idx = self.index.unwrap();
590                 let next = (*cur.as_ptr()).back;
591 
592                 // What self will become
593                 let new_len = old_idx + 1;
594                 let new_back = self.cur;
595                 let new_front = self.list.front;
596                 let new_idx = Some(old_idx);
597 
598                 // What the output will become
599                 let output_len = old_len - new_len;
600                 let output_front = next;
601                 let output_back = self.list.back;
602 
603                 // Break the links between cur and next
604                 if let Some(next) = next {
605                     (*cur.as_ptr()).back = None;
606                     (*next.as_ptr()).front = None;
607                 }
608 
609                 // Produce the result:
610                 self.list.len = new_len;
611                 self.list.front = new_front;
612                 self.list.back = new_back;
613                 self.index = new_idx;
614 
615                 LinkedList {
616                     front: output_front,
617                     back: output_back,
618                     len: output_len,
619                     _boo: PhantomData,
620                 }
621             }
622         } else {
623             // We're at the ghost, just replace our list with an empty one.
624             // No other state needs to be changed.
625             std::mem::replace(self.list, LinkedList::new())
626         }
627     }
628 
```

```
629     pub fn splice_before(&mut self, mut input: LinkedList<T>) {
630         // We have this:
631         //
632         // input.front -> 1 <-> 2 <- input.back
633         //
634         // list.front -> A <-> B <-> C <- list.back
635         //                     ^
636         //                    cur
637         //
638         //
639         // Becoming this:
640         //
641         // list.front -> A <-> 1 <-> 2 <-> B <-> C <- list.back
642         //                                 ^
643         //                                cur
644         //
645         unsafe {
646             // We can either `take` the input's pointers or `mem::forget`
647             // it. Using `take` is more responsible in case we ever do custom
648             // allocators or something that also needs to be cleaned up!
649             if input.is_empty() {
650                 // Input is empty, do nothing.
651             } else if let Some(cur) = self.cur {
652                 // Both lists are non-empty
653                 let in_front = input.front.take().unwrap();
654                 let in_back = input.back.take().unwrap();
655
656                 if let Some(prev) = (*cur.as_ptr()).front {
657                     // General Case, no boundaries, just internal fixups
658                     (*prev.as_ptr()).back = Some(in_front);
659                     (*in_front.as_ptr()).front = Some(prev);
660                     (*cur.as_ptr()).front = Some(in_back);
661                     (*in_back.as_ptr()).back = Some(cur);
662                 } else {
663                     // No prev, we're appending to the front
664                     (*cur.as_ptr()).front = Some(in_back);
665                     (*in_back.as_ptr()).back = Some(cur);
666                     self.list.front = Some(in_front);
667                 }
668                 // Index moves forward by input length
669                 *self.index.as_mut().unwrap() += input.len;
670             } else if let Some(back) = self.list.back {
671                 // We're on the ghost but non-empty, append to the back
672                 let in_front = input.front.take().unwrap();
673                 let in_back = input.back.take().unwrap();
```

```
674
675                 (*back.as_ptr()).back = Some(in_front);
676                 (*in_front.as_ptr()).front = Some(back);
677                 self.list.back = Some(in_back);
678             } else {
679                 // We're empty, become the input, remain on the ghost
680                 std::mem::swap(self.list, &mut input);
681             }
682
683             self.list.len += input.len;
684             // Not necessary but Polite To Do
685             input.len = 0;
686
687             // Input dropped here
688         }
689     }
690
691     pub fn splice_after(&mut self, mut input: LinkedList<T>) {
692         // We have this:
693         //
694         // input.front -> 1 <-> 2 <- input.back
695         //
696         // list.front -> A <-> B <-> C <- list.back
697         //                     ^
698         //                    cur
699         //
700         //
701         // Becoming this:
702         //
703         // list.front -> A <-> B <-> 1 <-> 2 <-> C <- list.back
704         //                     ^
705         //                    cur
706         //
707         unsafe {
708             // We can either `take` the input's pointers or `mem::forget`
709             // it. Using `take` is more responsible in case we ever do custom
710             // allocators or something that also needs to be cleaned up!
711             if input.is_empty() {
712                 // Input is empty, do nothing.
713             } else if let Some(cur) = self.cur {
714                 // Both lists are non-empty
715                 let in_front = input.front.take().unwrap();
716                 let in_back = input.back.take().unwrap();
717
718                 if let Some(next) = (*cur.as_ptr()).back {
```

```
719                     // General Case, no boundaries, just internal fixups
720                     (*next.as_ptr()).front = Some(in_back);
721                     (*in_back.as_ptr()).back = Some(next);
722                     (*cur.as_ptr()).back = Some(in_front);
723                     (*in_front.as_ptr()).front = Some(cur);
724                 } else {
725                     // No next, we're appending to the back
726                     (*cur.as_ptr()).back = Some(in_front);
727                     (*in_front.as_ptr()).front = Some(cur);
728                     self.list.back = Some(in_back);
729                 }
730                 // Index doesn't change
731             } else if let Some(front) = self.list.front {
732                 // We're on the ghost but non-empty, append to the front
733                 let in_front = input.front.take().unwrap();
734                 let in_back = input.back.take().unwrap();
735 
736                 (*front.as_ptr()).front = Some(in_back);
737                 (*in_back.as_ptr()).back = Some(front);
738                 self.list.front = Some(in_front);
739             } else {
740                 // We're empty, become the input, remain on the ghost
741                 std::mem::swap(self.list, &mut input);
742             }
743 
744             self.list.len += input.len;
745             // Not necessary but Polite To Do
746             input.len = 0;
747 
748             // Input dropped here
749         }
750     }
751 }
752 
753 unsafe impl<T: Send> Send for LinkedList<T> {}
754 unsafe impl<T: Sync> Sync for LinkedList<T> {}
755 
756 unsafe impl<'a, T: Send> Send for Iter<'a, T> {}
757 unsafe impl<'a, T: Sync> Sync for Iter<'a, T> {}
758 
759 unsafe impl<'a, T: Send> Send for IterMut<'a, T> {}
760 unsafe impl<'a, T: Sync> Sync for IterMut<'a, T> {}
761 
762 #[allow(dead_code)]
763 fn assert_properties() {
```

```
764     fn is_send<T: Send>() {}
765     fn is_sync<T: Sync>() {}
766 
767     is_send::<LinkedList<i32>>();
768     is_sync::<LinkedList<i32>>();
769 
770     is_send::<IntoIter<i32>>();
771     is_sync::<IntoIter<i32>>();
772 
773     is_send::<Iter<i32>>();
774     is_sync::<Iter<i32>>();
775 
776     is_send::<IterMut<i32>>();
777     is_sync::<IterMut<i32>>();
778 
779     fn linked_list_covariant<'a, T>(x: LinkedList<&'static T>) -> LinkedList<&'a T> {
780         x
781     }
782     fn iter_covariant<'i, 'a, T>(x: Iter<'i, &'static T>) -> Iter<'i, &'a T> {
783         x
784     }
785     fn into_iter_covariant<'a, T>(x: IntoIter<&'static T>) -> IntoIter<&'a T> {
786         x
787     }
788 
789     /// ```compile_fail,E0308
790     /// use linked_list::IterMut;
791     ///
792     /// fn iter_mut_covariant<'i, 'a, T>(x: IterMut<'i, &'static T>) -> IterMut<'i, &'a T> { x }
793     /// ```
794     fn iter_mut_invariant() {}
795 }
796 
797 #[cfg(test)]
798 mod test {
799     use super::LinkedList;
800 
801     fn generate_test() -> LinkedList<i32> {
802         list_from(&[0, 1, 2, 3, 4, 5, 6])
803     }
804 
805     fn list_from<T: Clone>(v: &[T]) -> LinkedList<T> {
806         v.iter().map(|x| (*x).clone()).collect()
807     }
808 
```

```
809     #[test]
810     fn test_basic_front() {
811         let mut list = LinkedList::new();
812 
813         // Try to break an empty list
814         assert_eq!(list.len(), 0);
815         assert_eq!(list.pop_front(), None);
816         assert_eq!(list.len(), 0);
817 
818         // Try to break a one item list
819         list.push_front(10);
820         assert_eq!(list.len(), 1);
821         assert_eq!(list.pop_front(), Some(10));
822         assert_eq!(list.len(), 0);
823         assert_eq!(list.pop_front(), None);
824         assert_eq!(list.len(), 0);
825 
826         // Mess around
827         list.push_front(10);
828         assert_eq!(list.len(), 1);
829         list.push_front(20);
830         assert_eq!(list.len(), 2);
831         list.push_front(30);
832         assert_eq!(list.len(), 3);
833         assert_eq!(list.pop_front(), Some(30));
834         assert_eq!(list.len(), 2);
835         list.push_front(40);
836         assert_eq!(list.len(), 3);
837         assert_eq!(list.pop_front(), Some(40));
838         assert_eq!(list.len(), 2);
839         assert_eq!(list.pop_front(), Some(20));
840         assert_eq!(list.len(), 1);
841         assert_eq!(list.pop_front(), Some(10));
842         assert_eq!(list.len(), 0);
843         assert_eq!(list.pop_front(), None);
844         assert_eq!(list.len(), 0);
845         assert_eq!(list.pop_front(), None);
846         assert_eq!(list.len(), 0);
847     }
848 
849     #[test]
850     fn test_basic() {
851         let mut m = LinkedList::new();
852         assert_eq!(m.pop_front(), None);
853         assert_eq!(m.pop_back(), None);
```

```
854         assert_eq!(m.pop_front(), None);
855         m.push_front(1);
856         assert_eq!(m.pop_front(), Some(1));
857         m.push_back(2);
858         m.push_back(3);
859         assert_eq!(m.len(), 2);
860         assert_eq!(m.pop_front(), Some(2));
861         assert_eq!(m.pop_front(), Some(3));
862         assert_eq!(m.len(), 0);
863         assert_eq!(m.pop_front(), None);
864         m.push_back(1);
865         m.push_back(3);
866         m.push_back(5);
867         m.push_back(7);
868         assert_eq!(m.pop_front(), Some(1));
869 
870         let mut n = LinkedList::new();
871         n.push_front(2);
872         n.push_front(3);
873         {
874             assert_eq!(n.front().unwrap(), &3);
875             let x = n.front_mut().unwrap();
876             assert_eq!(*x, 3);
877             *x = 0;
878         }
879         {
880             assert_eq!(n.back().unwrap(), &2);
881             let y = n.back_mut().unwrap();
882             assert_eq!(*y, 2);
883             *y = 1;
884         }
885         assert_eq!(n.pop_front(), Some(0));
886         assert_eq!(n.pop_front(), Some(1));
887     }
888 
889     #[test]
890     fn test_iterator() {
891         let m = generate_test();
892         for (i, elt) in m.iter().enumerate() {
893             assert_eq!(i as i32, *elt);
894         }
895         let mut n = LinkedList::new();
896         assert_eq!(n.iter().next(), None);
897         n.push_front(4);
898         let mut it = n.iter();
```

```
899         assert_eq!(it.size_hint(), (1, Some(1)));
900         assert_eq!(it.next().unwrap(), &4);
901         assert_eq!(it.size_hint(), (0, Some(0)));
902         assert_eq!(it.next(), None);
903     }
904 
905     #[test]
906     fn test_iterator_double_end() {
907         let mut n = LinkedList::new();
908         assert_eq!(n.iter().next(), None);
909         n.push_front(4);
910         n.push_front(5);
911         n.push_front(6);
912         let mut it = n.iter();
913         assert_eq!(it.size_hint(), (3, Some(3)));
914         assert_eq!(it.next().unwrap(), &6);
915         assert_eq!(it.size_hint(), (2, Some(2)));
916         assert_eq!(it.next_back().unwrap(), &4);
917         assert_eq!(it.size_hint(), (1, Some(1)));
918         assert_eq!(it.next_back().unwrap(), &5);
919         assert_eq!(it.next_back(), None);
920         assert_eq!(it.next(), None);
921     }
922 
923     #[test]
924     fn test_rev_iter() {
925         let m = generate_test();
926         for (i, elt) in m.iter().rev().enumerate() {
927             assert_eq!(6 - i as i32, *elt);
928         }
929         let mut n = LinkedList::new();
930         assert_eq!(n.iter().rev().next(), None);
931         n.push_front(4);
932         let mut it = n.iter().rev();
933         assert_eq!(it.size_hint(), (1, Some(1)));
934         assert_eq!(it.next().unwrap(), &4);
935         assert_eq!(it.size_hint(), (0, Some(0)));
936         assert_eq!(it.next(), None);
937     }
938 
939     #[test]
940     fn test_mut_iter() {
941         let mut m = generate_test();
942         let mut len = m.len();
943         for (i, elt) in m.iter_mut().enumerate() {
```

```
944             assert_eq!(i as i32, *elt);
945             len -= 1;
946         }
947         assert_eq!(len, 0);
948         let mut n = LinkedList::new();
949         assert!(n.iter_mut().next().is_none());
950         n.push_front(4);
951         n.push_back(5);
952         let mut it = n.iter_mut();
953         assert_eq!(it.size_hint(), (2, Some(2)));
954         assert!(it.next().is_some());
955         assert!(it.next().is_some());
956         assert_eq!(it.size_hint(), (0, Some(0)));
957         assert!(it.next().is_none());
958     }
959
960     #[test]
961     fn test_iterator_mut_double_end() {
962         let mut n = LinkedList::new();
963         assert!(n.iter_mut().next_back().is_none());
964         n.push_front(4);
965         n.push_front(5);
966         n.push_front(6);
967         let mut it = n.iter_mut();
968         assert_eq!(it.size_hint(), (3, Some(3)));
969         assert_eq!(*it.next().unwrap(), 6);
970         assert_eq!(it.size_hint(), (2, Some(2)));
971         assert_eq!(*it.next_back().unwrap(), 4);
972         assert_eq!(it.size_hint(), (1, Some(1)));
973         assert_eq!(*it.next_back().unwrap(), 5);
974         assert!(it.next_back().is_none());
975         assert!(it.next().is_none());
976     }
977
978     #[test]
979     fn test_eq() {
980         let mut n: LinkedList<u8> = list_from(&[]);
981         let mut m = list_from(&[]);
982         assert!(n == m);
983         n.push_front(1);
984         assert!(n != m);
985         m.push_back(1);
986         assert!(n == m);
987
988         let n = list_from(&[2, 3, 4]);
```

```
989         let m = list_from(&[1, 2, 3]);
990         assert!(n != m);
991     }
992
993     #[test]
994     fn test_ord() {
995         let n = list_from(&[]);
996         let m = list_from(&[1, 2, 3]);
997         assert!(n < m);
998         assert!(m > n);
999         assert!(n <= n);
1000        assert!(n >= n);
1001    }
1002
1003    #[test]
1004    fn test_ord_nan() {
1005        let nan = 0.0f64 / 0.0;
1006        let n = list_from(&[nan]);
1007        let m = list_from(&[nan]);
1008        assert!(!(n < m));
1009        assert!(!(n > m));
1010        assert!(!(n <= m));
1011        assert!(!(n >= m));
1012
1013        let n = list_from(&[nan]);
1014        let one = list_from(&[1.0f64]);
1015        assert!(!(n < one));
1016        assert!(!(n > one));
1017        assert!(!(n <= one));
1018        assert!(!(n >= one));
1019
1020        let u = list_from(&[1.0f64, 2.0, nan]);
1021        let v = list_from(&[1.0f64, 2.0, 3.0]);
1022        assert!(!(u < v));
1023        assert!(!(u > v));
1024        assert!(!(u <= v));
1025        assert!(!(u >= v));
1026
1027        let s = list_from(&[1.0f64, 2.0, 4.0, 2.0]);
1028        let t = list_from(&[1.0f64, 2.0, 3.0, 2.0]);
1029        assert!(!(s < t));
1030        assert!(s > one);
1031        assert!(!(s <= one));
1032        assert!(s >= one);
1033    }
```

```
1034
1035    #[test]
1036    fn test_debug() {
1037        let list: LinkedList<i32> = (0..10).collect();
1038        assert_eq!(format!("{:?}", list), "[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]");
1039
1040        let list: LinkedList<&str> = vec!["just", "one", "test", "more"]
1041            .iter()
1042            .copied()
1043            .collect();
1044        assert_eq!(format!("{:?}", list), r#"["just", "one", "test", "more"]"#);
1045    }
1046
1047    #[test]
1048    fn test_hashmap() {
1049        // Check that HashMap works with this as a key
1050
1051        let list1: LinkedList<i32> = (0..10).collect();
1052        let list2: LinkedList<i32> = (1..11).collect();
1053        let mut map = std::collections::HashMap::new();
1054
1055        assert_eq!(map.insert(list1.clone(), "list1"), None);
1056        assert_eq!(map.insert(list2.clone(), "list2"), None);
1057
1058        assert_eq!(map.len(), 2);
1059
1060        assert_eq!(map.get(&list1), Some(&"list1"));
1061        assert_eq!(map.get(&list2), Some(&"list2"));
1062
1063        assert_eq!(map.remove(&list1), Some("list1"));
1064        assert_eq!(map.remove(&list2), Some("list2"));
1065
1066        assert!(map.is_empty());
1067    }
1068
1069    #[test]
1070    fn test_cursor_move_peek() {
1071        let mut m: LinkedList<u32> = LinkedList::new();
1072        m.extend([1, 2, 3, 4, 5, 6]);
1073        let mut cursor = m.cursor_mut();
1074        cursor.move_next();
1075        assert_eq!(cursor.current(), Some(&mut 1));
1076        assert_eq!(cursor.peek_next(), Some(&mut 2));
1077        assert_eq!(cursor.peek_prev(), None);
1078        assert_eq!(cursor.index(), Some(0));
```

```
        cursor.move_prev();
        assert_eq!(cursor.current(), None);
        assert_eq!(cursor.peek_next(), Some(&mut 1));
        assert_eq!(cursor.peek_prev(), Some(&mut 6));
        assert_eq!(cursor.index(), None);
        cursor.move_next();
        cursor.move_next();
        assert_eq!(cursor.current(), Some(&mut 2));
        assert_eq!(cursor.peek_next(), Some(&mut 3));
        assert_eq!(cursor.peek_prev(), Some(&mut 1));
        assert_eq!(cursor.index(), Some(1));

        let mut cursor = m.cursor_mut();
        cursor.move_prev();
        assert_eq!(cursor.current(), Some(&mut 6));
        assert_eq!(cursor.peek_next(), None);
        assert_eq!(cursor.peek_prev(), Some(&mut 5));
        assert_eq!(cursor.index(), Some(5));
        cursor.move_next();
        assert_eq!(cursor.current(), None);
        assert_eq!(cursor.peek_next(), Some(&mut 1));
        assert_eq!(cursor.peek_prev(), Some(&mut 6));
        assert_eq!(cursor.index(), None);
        cursor.move_prev();
        cursor.move_prev();
        assert_eq!(cursor.current(), Some(&mut 5));
        assert_eq!(cursor.peek_next(), Some(&mut 6));
        assert_eq!(cursor.peek_prev(), Some(&mut 4));
        assert_eq!(cursor.index(), Some(4));
    }

    #[test]
    fn test_cursor_mut_insert() {
        let mut m: LinkedList<u32> = LinkedList::new();
        m.extend([1, 2, 3, 4, 5, 6]);
        let mut cursor = m.cursor_mut();
        cursor.move_next();
        cursor.splice_before(Some(7).into_iter().collect());
        cursor.splice_after(Some(8).into_iter().collect());
        // check_links(&m);
        assert_eq!(
            m.iter().cloned().collect::<Vec<_>>(),
            &[7, 1, 8, 2, 3, 4, 5, 6]
        );
        let mut cursor = m.cursor_mut();
```

```
        cursor.move_next();
        cursor.move_prev();
        cursor.splice_before(Some(9).into_iter().collect());
        cursor.splice_after(Some(10).into_iter().collect());
        check_links(&m);
        assert_eq!(
            m.iter().cloned().collect::<Vec<_>>(),
            &[10, 7, 1, 8, 2, 3, 4, 5, 6, 9]
        );

        /* remove_current not impl'd
        let mut cursor = m.cursor_mut();
        cursor.move_next();
        cursor.move_prev();
        assert_eq!(cursor.remove_current(), None);
        cursor.move_next();
        cursor.move_next();
        assert_eq!(cursor.remove_current(), Some(7));
        cursor.move_prev();
        cursor.move_prev();
        cursor.move_prev();
        assert_eq!(cursor.remove_current(), Some(9));
        cursor.move_next();
        assert_eq!(cursor.remove_current(), Some(10));
        check_links(&m);
        assert_eq!(m.iter().cloned().collect::<Vec<_>>(), &[1, 8, 2, 3, 4, 5, 6]);
        */

        let mut m: LinkedList<u32> = LinkedList::new();
        m.extend([1, 8, 2, 3, 4, 5, 6]);
        let mut cursor = m.cursor_mut();
        cursor.move_next();
        let mut p: LinkedList<u32> = LinkedList::new();
        p.extend([100, 101, 102, 103]);
        let mut q: LinkedList<u32> = LinkedList::new();
        q.extend([200, 201, 202, 203]);
        cursor.splice_after(p);
        cursor.splice_before(q);
        check_links(&m);
        assert_eq!(
            m.iter().cloned().collect::<Vec<_>>(),
            &[200, 201, 202, 203, 1, 100, 101, 102, 103, 8, 2, 3, 4, 5, 6]
        );
        let mut cursor = m.cursor_mut();
        cursor.move_next();
```

```
1169            cursor.move_prev();
1170            let tmp = cursor.split_before();
1171            assert_eq!(m.into_iter().collect::<Vec<_>>(), &[]);
1172            m = tmp;
1173            let mut cursor = m.cursor_mut();
1174            cursor.move_next();
1175            cursor.move_next();
1176            cursor.move_next();
1177            cursor.move_next();
1178            cursor.move_next();
1179            cursor.move_next();
1180            cursor.move_next();
1181            let tmp = cursor.split_after();
1182            assert_eq!(
1183                tmp.into_iter().collect::<Vec<_>>(),
1184                &[102, 103, 8, 2, 3, 4, 5, 6]
1185            );
1186            check_links(&m);
1187            assert_eq!(
1188                m.iter().cloned().collect::<Vec<_>>(),
1189                &[200, 201, 202, 203, 1, 100, 101]
1190            );
1191        }
1192
1193        fn check_links<T: Eq + std::fmt::Debug>(list: &LinkedList<T>) {
1194            let from_front: Vec<_> = list.iter().collect();
1195            let from_back: Vec<_> = list.iter().rev().collect();
1196            let re_reved: Vec<_> = from_back.into_iter().rev().collect();
1197
1198            assert_eq!(from_front, re_reved);
1199        }
1200    }
```

2.8 使用高级技巧实现链表

2.8.1 双单向链表

2.8.2 栈上的链表

Part II

Reference

Chapter 3

Rus 借用机制

3.1 编译时借用检查 (Compile-time Borrow Checking)

Rust 的借用规则是确保安全并发和避免数据竞争的核心机制

3.1.1 借用类型

- 不可变借用 (`&T`): 你可以同时拥有多个不可变借用，但不能有任何可变借用。
- 可变借用 (`&mut T`): 你只能有一个可变借用，并且在这个可变借用存在期间不能有任何不可变借用。

3.1.2 借用规则

1. 规则 1: 在给定的作用域中，任意时刻只能有一个可变借用 (`&mut T`)，或者任意数量的不可变借用 (`&T`)，但不能同时存在。
2. 规则 2: 借用必须始终是有效的 (**lifetime**)。

3.1.3 借用检查

编译时检查: **Rust** 编译器会在编译时进行借用检查，确保所有借用规则都被遵守。这意味着在编译时，所有可能的借用冲突都会被捕捉并引发错误。

3.2 运行时借用检查 (Runtime Borrow Checking)

Rust 的运行时借用检查 (Runtime Borrow Checking) 主要通过 **RefCell<T>** 类型实现。**RefCell** 是 **Rust** 标准库中的一种类型，它允许你在编译时无法确定借用规则的情况下，通过运行时借用检查来实现内部可变性。

3.2.1 什么是 RefCell<T>

RefCell<T> 是 **Rust** 中的一种智能指针类型，提供了内部可变性的功能。内部可变性 (Interior Mutability): 允许通过不可变引用来修改数据。**RefCell** 提供了类似于 `&T` 和 `&mut T` 的借用机制，但这些借用检查是在运行时进行的。

3.2.2 工作原理

不可变借用 (Ref<T>)

- 通过调用 `RefCell::borrow` 方法获取。
- 运行时检查当前是否存在可变借用，如果没有，则允许获取不可变借用。
- 允许同时存在多个不可变借用。

可变借用 (RefMut<T>)

- 通过调用 `RefCell::borrow_mut` 方法获取。
- 运行时检查当前是否存在任何借用 (包括不可变借用和其他可变借用)，如果没有，则允许获取可变借用。
- 只能存在一个可变借用，并且在它存在时不能有其他借用。

3.2.3 违反借用规则

如果在运行时违反了借用规则，**RefCell** 会触发恐慌 (**panic**)，以防止不安全操作。

3.2.4 用途

- **RefCell** 经常用于实现复杂数据结构，如图、树、链表等。
- 特别适合那些需要在运行时动态管理借用规则的场景。

3.2.5 多线程和 RefCell

RefCell 不是线程安全的。在多线程环境中, 可以使用 `std::sync::Mutex` 或 `std::sync::RwLock` 来代替 **RefCell**, 以实现线程安全的内部可变性。

3.3 对比

3.3.1 编译时 vs 运行时

- `&T` 和 `&mut T`: 借用检查是在编译时完成的。这意味着在编译时，你的代码必须符合借用规则，否则编译器会报错。
- **Ref<T>** 和 **RefMut<T>**: 借用检查是在运行时完成的。这允许你在编译时无法确定借用规则的情况下，仍然能够安全地借用。

3.3.2 使用场景

- `&T` 和 `&mut T`: 用于编译时可以确定借用规则的场景。通常在大部分 Rust 代码中使用。适用于大多数情况下，你能清晰地在编译时知道借用规则。
- **Ref<T>** 和 **RefMut<T>**: 用于编译时无法确定借用规则，但可以在运行时安全地检查借用规则的场景。适用于更复杂的场景，比如需要在运行时管理借用规则，特别是在实现复杂的数据结构时。

W

3.3.3 内部可变性

- `&T` 和 `&mut T`: 使用常规的借用类型，无法实现内部可变性 (即，在不可变引用下修改数据)。
- **Ref<T>** 和 **RefMut<T>**: **RefCell<T>** 通过 **Ref<T>** 和 **RefMut<T>** 允许在内部实现可变性，允许你在不可变引用下修改数据，这在某些场景下非常有用。

3.3.4 性能影响

- `&T` 和 `&mut T`: 因为它们的借用是在编译时检查的，所以性能损失很小。
- **Ref<T>** 和 **RefMut<T>**: 因为借用是运行时检查的，所以在使用 RefCell 的地方会有额外的性能开销 (例如，每次借用时都会进行借用检查)。

3.3.5 错误处理

- `&T` 和 `&mut T`: 如果违反借用规则，编译器会直接报错。
- **Ref<T>** 和 **RefMut<T>**: 如果在运行时违反借用规则，会导致程序恐慌 (**panic**)。因此，使用 **RefCell** 时要小心避免运行时借用错误。

3.3.6 多线程

- `&T` 和 `&mut T`: 可以安全地在线程间共享（前提是满足 Send 和 Sync trait)。
- **Ref<T>** 和 **RefMut<T>**: **RefCell** 不在线程间安全。如果需要多线程中的内部可变性，可以使用 `std::sync::Mutex` 或 `std::sync::RwLock`。

Bibliography

[1] Sunface and https://github.com/sunface/rust course/graphs/contributors. Rust 语言圣经 (rust course), Nov 2021.