

双向链表的实现

这份文档只针对通常意义下的双向链表的实现，要实现 FreeRTOS 中的 `list.c`，只需要修改几个数据域即可。

文件的目录结构如下：

```
1  src
2  |- main.rs
3  |- lib.rs
4  |- list.rs
```

其中 `lib.rs` 内容如下：

```
1  pub mod list;
```

下文的代码全部写在 `list.rs` 中。

设置数据结构

我们设置的链表带有一个头结点，然后是相应的子节点，它的结构可以声明如下：

```
1  use std::rc::Rc;
2  use std::cell::RefCell;
3
4  pub struct List<T> {
5      head: Link<T>,
6      tail: Link<T>,
7  }
8
9  type Link<T> = Option<Rc<RefCell<Node<T>>>>;
10
11 struct Node<T> {
12     elem: T,
13     next: Link<T>,
14     prev: Link<T>,
15 }
```

在这里，使用了 `RefCell` 类型，是因为待会会使用到它提供的 `borrow` 和 `borrow_mut` 方法。而 `Rc` 的使用是为了共享变量的 `ownership`。

List 方法实现

首先，最先实现的就是 `new` 方法了。该方法主要起一个初始化的工作。它的的实现如下：

```
1  impl<T> Node<T> {
2      fn new(elem: T) -> Rc<RefCell<Self>> {
3          Rc::new(RefCell::new(Node {
```

```

4         elem: elem,
5         prev: None,
6         next: None,
7     )))
8 }
9 }
10
11 impl<T> List<T> {
12     pub fn new() -> Self {
13         List { head: None, tail: None }
14     }
15 }

```

接下来实现 `push_front` 方法。实现细节如下：

```

1 pub fn push_front(&mut self, elem: T) {
2     let new_head = Node::new(elem);
3     // Takes the value out of the option, leaving a None in its place.
4     match self.head.take() {
5         Some(old_head) => {
6             // borrow_mut 作用于RefCell, 获取里面的借用并且将其转型为mut
7             old_head.borrow_mut().prev = Some(new_head.clone());
8             new_head.borrow_mut().next = Some(old_head);
9             self.head = Some(new_head);
10        }
11        None => {
12            self.tail = Some(new_head.clone());
13            self.head = Some(new_head);
14        }
15    }
16 }

```

接着就是实现 `pop_front` 方法。具体如下：

```

1 pub fn pop_front(&mut self) -> Option<T> {
2     self.head.take().map(|old_head| {
3         match old_head.borrow_mut().next.take() {
4             Some(new_head) => {
5                 new_head.borrow_mut().prev.take();
6                 self.head = Some(new_head);
7             }
8             None => {
9                 self.tail.take();
10            }
11        }
12        // all we want to do is to get the elem
13        // Option<Rc<RefCell<Node<T>>>> -----> Node.elem
14        Rc::try_unwrap(old_head).ok().unwrap().into_inner().elem
15    })
16 }

```

然后就是实现 `peek_front` 方法：

```

1 pub fn peek_front(&self) -> Option<Ref<T>> {
2     // Returns None if the pointer is null,
3     // or else returns a reference to the value wrapped in Some.
4     self.head.as_ref().map(|node| {
5         Ref::map(node.borrow(), |node| &node.elem)
6     })
7 }

```

内存的释放:

```

1 impl<T> Drop for List<T> {
2     // This will decrement the strong reference count.
3     // If the strong reference count reaches zero
4     // then the only other references (if any) are Weak, so we drop the inner value.
5     fn drop(&mut self) {
6         while self.pop_front().is_some() {}
7     }
8 }

```

其他的相关方法附在最终代码中。

遇到的问题

- 在尝试编写 FreeRTOS 中 `list.c` 的过程中, 我们发现 `List` 和 `ListItem` 之间似乎存在着相互的引用, 这意味着两者的生命周期必须一样, 这似乎有些矛盾, 因为 `List` 的生命周期显然要比 `ListItem` 的生命周期长。我们的初步想法是重新构建 `ListItem` 中的某些 `成员`, 让其不直接指向 `List`。这一部分待定。
- 在编写 `rust` 的程序时, 最为头疼的可能就是其变量的**生命周期**了。这一个特性基本上是 `rust` 独有的, 可能理解起来会有难度。暂时接触的比较少, 需要进一步加强练习。
- 尽管我们已经仔细阅读过 `官方文档`, 但是仍然对其 `标准库` 不太熟悉, 时常会有 `invent the wheel` 的行为, 这增加了我们的负担。需要我们进一步了解和学习 `rust` 提供的标准库。

最终代码

最终代码如下(包含测试代码):

```

1 use std::rc::Rc;
2 use std::cell::{Ref, RefMut, RefCell};
3
4 pub struct List<T> {
5     head: Link<T>,
6     tail: Link<T>,
7 }
8
9 type Link<T> = Option<Rc<RefCell<Node<T>>>>>;
10
11 struct Node<T> {
12     elem: T,
13     next: Link<T>,

```

```

14     prev: Link<T>,
15 }
16
17
18 impl<T> Node<T> {
19     fn new(elem: T) -> Rc<RefCell<Self>> {
20         Rc::new(RefCell::new(Node {
21             elem: elem,
22             prev: None,
23             next: None,
24         })))
25     }
26 }
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();
72     }
73 }
74 Rc::try_unwrap(old_tail).ok().unwrap().into_inner().elem
75 })
76 }
77
78 pub fn pop_front(&mut self) -> Option<T> {
79     self.head.take().map(|old_head| {
80         match old_head.borrow_mut().next.take() {
81             Some(new_head) => {
82                 new_head.borrow_mut().prev.take();
83                 self.head = Some(new_head);
84             }
85             None => {
86                 self.tail.take();
87             }
88         }
89         Rc::try_unwrap(old_head).ok().unwrap().into_inner().elem
90     })
91 }
92
93 pub fn peek_front(&self) -> Option<Ref<T>> {
94     self.head.as_ref().map(|node| {
95         Ref::map(node.borrow(), |node| &node.elem)
96     })
97 }
98
99 pub fn peek_back(&self) -> Option<Ref<T>> {
100     self.tail.as_ref().map(|node| {
101         Ref::map(node.borrow(), |node| &node.elem)
102     })
103 }
104
105 pub fn peek_back_mut(&mut self) -> Option<RefMut<T>> {
106     self.tail.as_ref().map(|node| {
107         RefMut::map(node.borrow_mut(), |node| &mut node.elem)
108     })
109 }
110
111 pub fn peek_front_mut(&mut self) -> Option<RefMut<T>> {
112     self.head.as_ref().map(|node| {
113         RefMut::map(node.borrow_mut(), |node| &mut node.elem)
114     })
115 }
116
117 }
118
119 impl<T> Drop for List<T> {

```

```
120     fn drop(&mut self) {
121         while self.pop_front().is_some() {}
122     }
123 }
124
125
126 #[cfg(test)]
127 mod test {
128     use super::List;
129
130     #[test]
131     fn basics() {
132         let mut list = List::new();
133
134         // Check empty list behaves right
135         assert_eq!(list.pop_front(), None);
136
137         // Populate list
138         list.push_front(1);
139         list.push_front(2);
140         list.push_front(3);
141
142         // Check normal removal
143         assert_eq!(list.pop_front(), Some(3));
144         assert_eq!(list.pop_front(), Some(2));
145
146         // Push some more just to make sure nothing's corrupted
147         list.push_front(4);
148         list.push_front(5);
149
150         // Check normal removal
151         assert_eq!(list.pop_front(), Some(5));
152         assert_eq!(list.pop_front(), Some(4));
153
154         // Check exhaustion
155         assert_eq!(list.pop_front(), Some(1));
156         assert_eq!(list.pop_front(), None);
157
158         // ---- back ----
159
160         // Check empty list behaves right
161         assert_eq!(list.pop_back(), None);
162
163         // Populate list
164         list.push_back(1);
165         list.push_back(2);
166         list.push_back(3);
167
168         // Check normal removal
169         assert_eq!(list.pop_back(), Some(3));
170         assert_eq!(list.pop_back(), Some(2));
171
172         // Push some more just to make sure nothing's corrupted
```

```

173         list.push_back(4);
174         list.push_back(5);
175
176         // Check normal removal
177         assert_eq!(list.pop_back(), Some(5));
178         assert_eq!(list.pop_back(), Some(4));
179
180         // Check exhaustion
181         assert_eq!(list.pop_back(), Some(1));
182         assert_eq!(list.pop_back(), None);
183     }
184
185     #[test]
186     fn peek() {
187         let mut list = List::new();
188         assert!(list.peek_front().is_none());
189         assert!(list.peek_back().is_none());
190         assert!(list.peek_front_mut().is_none());
191         assert!(list.peek_back_mut().is_none());
192
193         list.push_front(1); list.push_front(2); list.push_front(3);
194
195         assert_eq!(&*list.peek_front().unwrap(), &3);
196         assert_eq!(&mut *list.peek_front_mut().unwrap(), &mut 3);
197         assert_eq!(&*list.peek_back().unwrap(), &1);
198         assert_eq!(&mut *list.peek_back_mut().unwrap(), &mut 1);
199     }
200
201
202 }

```

相关文献

- <https://doc.rust-lang.org/book/ch10-03-lifetime-syntax.html>
- https://doc.rust-lang.org/rust-by-example/custom_types/enum/c_like.html
- <https://doc.rust-lang.org/std/collections/struct.LinkedList.html>
- <https://github.com/alexchandel/rust-rtos>
- <https://github.com/beschaef/rtos>
- <https://rust-unofficial.github.io/too-many-lists/fourth-final.html>