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：

```
src
|- main.rs
|- lib.rs
|- list.rs
```

其中 `lib.rs` 内容如下：

```
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：

```
use std::rc::Rc;
use std::cell::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>,
}
```

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：

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

```
            elem: elem,
            prev: None,
            next: None,
        }))
    }
}

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

接下来实现 `push_front` 方法。实现细节如下：

```
pub fn push_front(&mut self, elem: T) {
    let new_head = Node::new(elem);
    // Takes the value out of the option, leaving a None in its place.
    match self.head.take() {
        Some(old_head) => {
            // borrow_mut 作用于RefCell，获取里面的借用并且将其转型为mut
            old_head.borrow_mut().prev = Some(new_head.clone());
            new_head.borrow_mut().next = Some(old_head);
            self.head = Some(new_head);
        }
        None => {
            self.tail = Some(new_head.clone());
            self.head = Some(new_head);
        }
    }
}
```

接着就是实现 `pop_front` 方法。具体如下：

```
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();
            }
        }
        // all we want to do is to get the elem
        // Option<Rc<RefCell<Node<T>>>>  ----->   Node.elem
        Rc::try_unwrap(old_head).ok().unwrap().into_inner().elem
    })
}
```

然后就是实现 `peek_front` 方法：

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

内存的释放：

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

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)：

```rust
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,
        }))
    }
}

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

    pub fn push_front(&mut self, elem: T) {
        let new_head = Node::new(elem);
        match self.head.take() {
            Some(old_head) => {
                old_head.borrow_mut().prev = Some(new_head.clone());
                new_head.borrow_mut().next = Some(old_head);
                self.head = Some(new_head);
            }
            None => {
                self.tail = Some(new_head.clone());
                self.head = Some(new_head);
            }
        }
    }

    pub fn push_back(&mut self, elem: T) {
        let new_tail = Node::new(elem);
        match self.tail.take() {
            Some(old_tail) => {
                old_tail.borrow_mut().next = Some(new_tail.clone());
                new_tail.borrow_mut().prev = Some(old_tail);
                self.tail = Some(new_tail);
            }
            None => {
                self.head = Some(new_tail.clone());
                self.tail = Some(new_tail);
            }
        }
    }

    pub fn pop_back(&mut self) -> Option<T> {
        self.tail.take().map(|old_tail| {
            match old_tail.borrow_mut().prev.take() {
                Some(new_tail) => {
```

```
                    new_tail.borrow_mut().next.take();
                    self.tail = Some(new_tail);
                }
                None => {
                    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)
        })
    }

}

impl<T> Drop for List<T> {
```

```rust
    fn drop(&mut self) {
        while self.pop_front().is_some() {}
    }
}


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

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

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

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

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

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

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

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

        // ---- back -----

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

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

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

        // 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