세상에서 제일 쉬운 러스트 프로그래밍

11장. 스마트 포인터

윤인도

freedomzero91@gmail.com

포인터란?

- 포인터는 메모리 주소를 포함하는 변수 : 어떤 값이 저장된 메모리 주소를 가리킴
- 러스트에서 가장 일반적인 종류의 포인터는 레퍼런스

스마트 포인터란?

스마트 포인터는 포인터처럼 작동하지만, 레퍼런스의 개수 등의 메타데이터를 포함하고 있는 자료구조

- 예) `String`, `Vec`

`Deref` 트레이트

- `std::ops::Deref`에 정의
- 레퍼런스 또는 스마트 포인터가 가리키고 있는 값을 써야할 때 사용

레퍼런스로 값을 참조하는 것과 원본 값을 사용하는 것의 차이점

1. `x`의 가변 레퍼런스 `y`의 값은 "a"
2. `*y`를 통해 `x`의 값인 "a"를 얻을 수 있고, 이 값을 "b"로 변경
3. `x`의 값 이 변경됐기 때문에 마지막에 다시 `x`를 출력한 결과는 "b"

```
fn main() {
    let mut x = "a";
    println!("x was {}", x);
    let y = &mut x; // 1
    println!("y is {}", y);
    *y = "b"; // 2
    println!("but now x is {}", x); // 3
}
```

실행 결과

```
x was a
y is a
but now x is b
```

`*` 연산자가 호출하는 트레이트가 바로 `Deref`

```rust
use std::ops::Deref;

struct DerefExample<T> {
    _value: T,
}

impl<T> Deref for DerefExample<T> {
    type Target = T;
    fn deref(&self) -> &Self::Target {
        &self._value
    }
}

fn main() {
    let x = DerefExample { _value: 'a' };
    assert_eq!('a', *x); // x.deref()
}
```

⚠️ 주의

- `Deref`는 반드시 스마트 포인터에만 구현해야만 `*` 연산자의 사용이 헷갈리지 않습니다.
- `Deref`는 러스트에서 내부적으로 호출하는 트레이트이기 때문에 절대 실패해서는 안 됩니다.

Drop 트레이트

- 어떤 값이 범위에서 삭제(drop)될 때 러스트가 자동으로 호출하는 트레이트
- `Drop`은 특별한 2가지 경우에만 구현
    i. 값이 메모리에서 삭제될 때 추가 처리가 필요한 경우
    ii. 범위를 벗어나기 전에 강제로 메모리에서 값을 삭제하고 싶은 경우

1️⃣ 값이 메모리에서 삭제될 때 추가 처리가 필요한 경우

```
struct DropExample { value: i32 }

impl Drop for DropExample { // std::ops::Drop
    fn drop(&mut self) {
        println!("Dropping {}", self.value);
    }
}
fn main() {
    let _x = DropExample { value: 1 };
    {
        let _y = DropExample { value: 2 };
    }
}
```

실행결과

```
Dropping 2
Dropping 1
```

2️⃣ 범위를 벗어나기 전에 강제로 메모리에서 값을 삭제하고 싶은 경우

```
fn main() {
    let x = DropExample { value: 1 };
    drop(x);
    let _y = x.value; // 🤯
}
```

실행 결과

```
error[E0382]: use of moved value: `x`
  --> src/main.rs:14:14
   |
12 |     let x = DropExample { value: 1 };
   |         - move occurs because `x` has type `DropExample`,
   |             which does not implement the `Copy` trait
13 |     drop(x);
   |          - value moved here
14 |     let _y = x.value; // 🤯
   |              ^^^^^^^ value used here after move
   |
```

Box 타입

만일 어떤 타입의 크기를 컴파일 타임에 미리 알 수 없다면?

연결 리스트: `value`와 `next` 필드를 가지고 있는 구조체

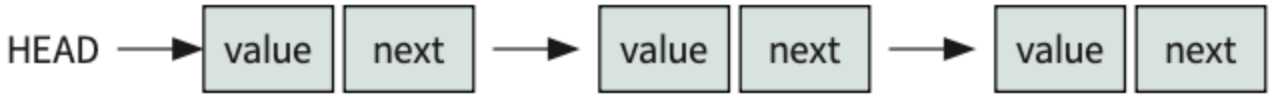

그림 11-1 연결 리스트의 구조

자기 자신을 필드값의 타입으로 갖는 재귀 형태(Recursive)의 구조체를 정의

```
struct Node {
    value: i32,
    next: Option<Node>,
}

fn main() {
    let mut head = Node {
        value: 1,
        next: None,
    };
    head.next = Some(Node {
        value: 2,
        next: None,
    });
    println!("{}", head.value);
}
```

실행 결과

```
error[E0072]: recursive type `Node` has infinite size
 --> src/main.rs:1:1
  |
1 | struct Node {
  | ^^^^^^^^^^^
2 |     value: i32,
3 |     next: Option<Node>,
  |                  ---- recursive without indirection
  |
help: insert some indirection (e.g., a `Box`, `Rc`, or `&`) to break the cycle
  |
3 |     next: Option<Box<Node>>,
  |                  ++++    +
```

`Box` 를 사용하라고 함!

```rust
struct Node {
    value: i32,
    next: Option<Box<Node>>,
}

fn main() {
    let mut head = Node {
        value: 1,
        next: None,
    };
    head.next = Some(Box::new(Node {
        value: 2,
        next: None,
    }));
    println!("{}", head.value); // 1
}
```

`Box<T>`

`Box`가 대체 무엇일까?

- `Box`를 사용하면 스택이 아닌 힙에 데이터를 저장
- 스택에는 힙 데이터에 대한 포인터만 저장

`Box<T>` 사용하기

아래 예제는 `Box`를 사용하여 `i32` 값을 힙에 저장하는 방법을 보여줍니다.

```
fn main() {
    let my_box = Box::new(5);
    println!("my_box = {}", my_box); // 5

    let num = *my_box; // my_box.deref()
    assert_eq!(num, 5);
}
```

스코프 마지막에 `my_box` 역시 삭제됨

`Box`는 주로 다음과 같은 상황에 사용됩니다.

- 컴파일 타임에 크기를 알 수 없는 타입을 사용하는 경우
- 특정 타입이 아닌, 특정 트레이트를 구현하는 타입의 변수의 소유권을 가져오고 싶은 경우

첫 번째는 위에서 이미 살펴본 `Node`의 경우입니다. 이제 두 번째 경우를 자세히 살펴보겠습니다.

`dyn` 과 `Box` 로 트레이트 타입 표현하기

크레이트를 추가

```
cargo add rand
```

```rust
struct Dog {}
struct Cat {}

trait Animal {
    fn noise(&self) -> &'static str;
}

impl Animal for Dog {
    fn noise(&self) -> &'static str {
        "🐶멍멍!"
    }
}

impl Animal for Cat {
    fn noise(&self) -> &'static str {
        "🐱야옹!"
    }
}
```

무작위 수가 0.5보다 작으면 `Dog`를, 그렇지 않으면 `Cat`을 반환하는 함수

```
fn random_animal() -> impl Animal {
    if rand::random::<f64>() < 0.5 {
        Dog {}
    } else {
        Cat {}
    }
}

fn main() {
    for _ in 0..10 {
        println!("{}", random_animal().noise());
    }
}
```

```
error[E0308]: `if` and `else` have incompatible types
  --> src/main.rs:24:9
   |
21 | /     if rand::random::<f64>() < 0.5 {
22 | |         Dog {}
   | |         ------ expected because of this
23 | |     } else {
24 | |         Cat {}
   | |         ^^^^^^ expected struct `Dog`, found struct `Cat`
25 | |     }
   | |_____- `if` and `else` have incompatible types
   |
help: you could change the return type to be a boxed trait object
   |
20 | fn random_animal() -> Box<dyn Animal> {
   |                       ~~~~~~~      +
help: if you change the return type to expect trait objects,
box the returned expressions
   |
22 ~         Box::new(Dog {})
23 |     } else {
24 ~         Box::new(Cat {})
   |
```

컴파일러는 `random_animal` 함수의 리턴 타입을 `Box`와 `dyn`을 사용해 감싸라고 조언

```rust
fn random_animal() -> Box<dyn Animal> {
    if rand::random::<f64>() < 0.5 {
        Box::new(Dog {})
    } else {
        Box::new(Cat {})
    }
}
```

실행 결과

```
🐱야옹!
🐶멍멍!
🐶멍멍!
🐶멍멍!
🐱야옹!
🐱야옹!
🐶멍멍!
🐶멍멍!
🐶멍멍!
🐶멍멍!
```

- 리턴 타입으로 `impl Trait`을 사용하면 컴파일러가 리턴 타입의 크기를 미리 알 수 없음
- 따라서 `Box`와 `dyn` 키워드로 리턴 타입에 서로 다른 타입을 사용할 수 있게 하는 것

함수의 입력으로는 `impl Trait`이 가능하지만 리턴 타입으로는 불가능한 이유?

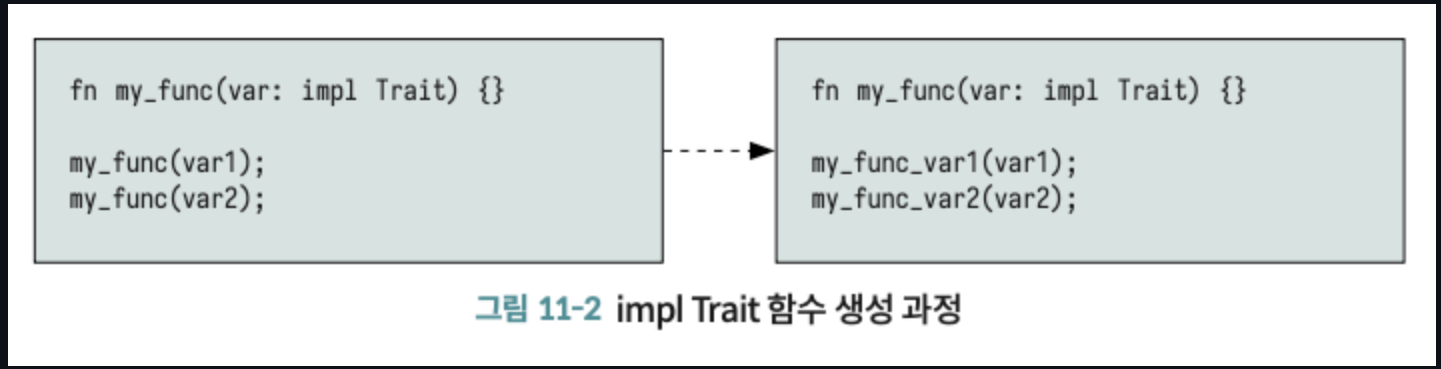

그림 11-2 impl Trait 함수 생성 과정

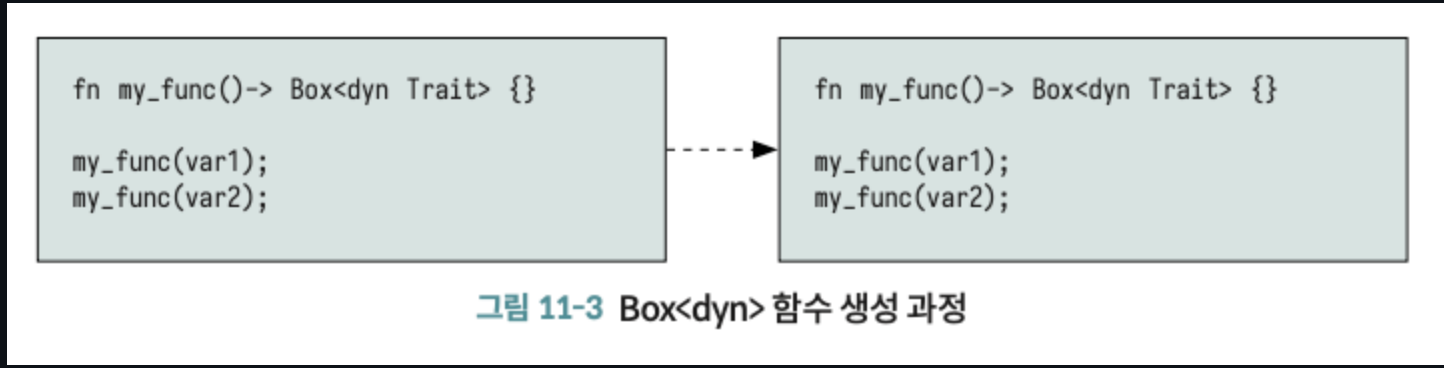

그림 11-3 Box<dyn> 함수 생성 과정

`Rc<T>`

만일 하나의 값에 여러 개의 소유자를 정말로 가지고 싶다면 어떻게 할까요?

`Rc<T>`는 레퍼런스 카운팅(Reference counting)이라는 뜻으로, 스마트 포인터 중 하나

마지막 순간까지

- 프로그램의 여러 부분이 읽을 수 있도록 힙에 일부 데이터를 할당하고, 컴파일 시점에 데이터가 마지막으로 사용되는 지점을 알 수 없을 때 `Rc<T>` 타입을 사용
- `Rc<T>`는 값의 소유권을 가지고 있는 변수가 몇 개인지를 계속 확인하고 있다가, 값을 소유하고 있는 변수가 전부 사라지면 값을 메모리에서 삭제

여기서 `clone`을 사용하면, 실제로 값이 복사되는 것이 아니라 `Rc`의 레퍼런스 카운트가 1 증가!

```
use std::rc::Rc;
fn main() {
    let origin = Rc::new(0);
    println!("Reference count: {}", Rc::strong_count(&origin));
    {
        let _dup1 = Rc::clone(&origin);
        println!("Reference count: {}", Rc::strong_count(&origin));
        {
            let _dup2 = &origin.clone();
            println!("Reference count: {}", Rc::strong_count(&origin));
        }
        println!("Reference count: {}", Rc::strong_count(&origin));
    }
    println!("Reference count: {}", Rc::strong_count(&origin));
    // origin drops here
}
```

실행 결과

```
Reference count: 1
Reference count: 2
Reference count: 3
Reference count: 2
Reference count: 1
```

레퍼런스를 사용하면 범위를 벗어난 값을 사용 불가

```rust
fn main() {
    let cloned;
    {
        let origin = "Rust".to_string();
    }
    cloned = &origin; // 🤯
    println!("{}", cloned);
}
```

실행 결과

```
error[E0425]: cannot find value `origin` in this scope
 --> src/main.rs:6:15
  |
6 |     cloned = &origin; // 🤯
  |              ^^^^^^
  |
```

`Rc<T>`를 사용하면 범위를 벗어난 값을 사용 가능

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

fn main() {
    let cloned;
    {
        let origin = Rc::new(1);
        cloned = origin.clone();
    }
    println!("{}", cloned); // 1
}
```

> `Rc<T>`는 멀티스레드 환경에서 동작하지 않습니다. 멀티스레드 환경에서는 `Arc<T>`를 사용해야 하며, 자세한 내용은 나중에 다루겠습니다.

Quiz

다음 코드가 컴파일되도록 코드를 수정해주세요.

```
struct Node {
    value: i32,
    next: Option<Box<Node>>,
}

fn main() {
    let mut head1 = Node {
        value: 1,
        next: None,
    };
    let node1 = Node {
        value: 2,
        next: None,
    };
    head1.next = Some(Box::new(node1));

    let mut head2 = Node {
        value: 3,
        next: None,
    };
    head2.next = Some(Box::new(node1)); // 😲

    println!("{} {}", head1.value, head1.next.unwrap().value);
    println!("{} {}", head2.value, head2.next.unwrap().value);
}
```

정답

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

struct Node {
    value: i32,
    next: Option<Rc<Node>>, // ✨
}

fn main() {
    let mut head1 = Node {
        value: 1,
        next: None,
    };
    let node1 = Rc::new(Node { // ✨
        value: 2,
        next: None,
    });
    head1.next = Some(Rc::clone(&node1));  // ✨

    let mut head2 = Node {
        value: 3,
        next: None,
    };
    head2.next = Some(Rc::clone(&node1));  // ✨

    println!("{} {}", head1.value, head1.next.unwrap().value);
    println!("{} {}", head2.value, head2.next.unwrap().value);
}
```

`RefCell<T>`

`Rc<T>`의 한계

- `Rc<T>`를 사용하면 프로그램의 여러 부분에서 읽기 전용으로 데이터를 공유 가능
- 하지만 불변 레퍼런스를 통해 값을 공유하기 때문에, 공유받은 값을 변경할 수 없음

```rust
use std::rc::Rc;
struct Owner {
    name: String,
    tools: Vec<Tool>,
}
struct Tool {
    name: String,
    owner: Rc<Owner>,
}
```

```rust
pub fn main() {
    let indo = Rc::new(Owner {
        name: "indo".to_string(),
        tools: vec![],
    });
    let plier = Tool {
        name: "plier".to_string(),
        owner: indo.clone(),
    };
    let wrench = Tool {
        name: "wrench".to_string(),
        owner: indo.clone(),
    };
    indo.tools.push(plier);
    indo.tools.push(wrench);
    for tool in indo.tools.iter() {
        println!("{} is owned by {}",
            tool.name, tool.owner.name);
    }
}
```

실행 결과

```
error[E0596]: cannot borrow data in an `Rc` as mutable
  --> src/main.rs:23:5
   |
23 |     indo.tools.push(plier);
   |     ^^^^^^^^^^ cannot borrow as mutable
   |
   = help: trait `DerefMut` is required to modify through a dereference,
   but it is not implemented for `Rc<Owner>`

error[E0596]: cannot borrow data in an `Rc` as mutable
  --> src/main.rs:24:5
   |
24 |     indo.tools.push(wrench);
   |     ^^^^^^^^^^ cannot borrow as mutable
   |
   = help: trait `DerefMut` is required to modify through a dereference,
   but it is not implemented for `Rc<Owner>`
```

⚠️ 에러가 발생한 이유가 `indo` 자체가 불변이기 때문이 아니라는 점에 주의

`RefCell`은 `Rc`와 비슷하게 어떤 값을 힙 영역에 저장하지만, 내부 값을 변경 가능

```
use std::{cell::RefCell, rc::Rc};
struct Owner {
    name: String,
    tools: RefCell<Vec<Tool>>,
}
```

```
pub fn main() {
    let indo = Rc::new(Owner { // ⚠️ 가변이 아님에 주의!
        name: "indo".to_string(),
        tools: RefCell::new(vec![]),
    });
    let plier = Tool {
        name: "plier".to_string(),
        owner: indo.clone(),
    };
    let wrench = Tool {
        name: "wrench".to_string(),
        owner: indo.clone(),
    };
    indo.tools.borrow_mut().push(plier); // 가변 소유권 대여
    indo.tools.borrow_mut().push(wrench);
    for tool in indo.tools.borrow().iter() { // 불변 소유권 대여
        println!("{} is owned by {}", tool.name, tool.owner.name);
    }
}
```

실행 결과

```
plier is owned by indo
wrench is owned by indo
```

내부 가변성(Interiror mutability)

`RefCell<T>`가 불변이어도 내부의 값은 가변으로 사용 가능

```
indo.tools.borrow_mut().push(Rc::clone(&pliers));
```

불변 소유권 대여도 가능

```
indo.tools.borrow().iter()
```

⚠️ 내부 가변성 사용시 주의

- 일반적인 소유권 규칙과 마찬가지로, 가변 소유권은 단 한번만 대여 가능
- 런타임 시간에 소유권이 확인되기 때문에 컴파일이 되지만 런타임 에러 발생

컴파일되지만 런타임 에러가 발생!

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

struct Owner {
    name: String,
    tools: RefCell<Vec<Tool>>,
}

struct Tool {
    owner: Rc<Owner>,
}
```

```rust
pub fn main() {
    let indo = Rc::from(Owner {
        name: "indo".to_string(),
        tools: RefCell::new(vec![]),
    });
    let pliers = Rc::from(Tool {
        owner: Rc::clone(&indo),
    });
    let wrench = Rc::from(Tool {
        owner: indo.clone(),
    });

    let mut borrow_mut_tools1 = indo.tools.borrow_mut();
    let mut borrow_mut_tools2 = indo.tools.borrow_mut(); // 😱
}
```

실행 결과

```
thread 'main' panicked at 'already borrowed: BorrowMutError', src/main.rs:25:44
```

`Rc<RefCell<T>>`

벡터로 소유권이 이동한 `alice`의 소유권을 다시 벡터로 대여할 수 없음

```
use std::cell::RefCell;

struct Person {
    name: String,
    age: u8,
}
fn main() {
    let alice = RefCell::new(Person {
        name: "Alice".to_string(),
        age: 30,
    });
    let people = vec![alice];
    for person in people {
        person.borrow_mut().age += 1;
    }
    println!("Alice is now {} years old", alice.borrow().age); //🤯
}
```

실행 결과

```
error[E0382]: borrow of moved value: `alice`
  --> src/main.rs:15:43
   |
7  |     let alice = RefCell::new(Person {
   |         ----- move occurs because `alice` has type `RefCell<Person>`,
                  which does not implement the `Copy` trait
...
11 |     let people = vec![alice];
   |                       ----- value moved here
...
15 |     println!("Alice is now {} years old", alice.borrow().age); //
   |                                           ^^^^^ value borrowed here after move
```

`Rc<RefCell<T>>`를 사용하면 여러 소유자가 값을 변경 가능

```
use std::cell::RefCell;
use std::rc::Rc;

fn main() {
    let alice = Rc::new(RefCell::new(Person {
        name: "Alice".to_string(),
        age: 30,
    }));
    let people = vec![alice.clone()]; // `Rc`를 복제해서 소유권을 공유
    for person in people {
        person.borrow_mut().age += 1; // 가변 소유권 대여
    }
    println!("Alice is now {} years old", alice.borrow().age);
}
```

실행 결과

```
Alice is now 31 years old
```

언제 무엇을

표 11-1 스마트 포인터

	Box<T>	Rc<T>	RefCell<T>
소유권	한 개	한 개를 공유	한 개를 공유
소유권 확인 시점	불변/가변 소유권을 컴파일 타임에 확인	불변 소유권을 컴파일 타임에 확인	불변/가변 소유권을 런타임에 확인
특징	스코프를 벗어나면 레퍼런스도 모두 삭제	레퍼런스가 존재한다면 스코프를 벗어나도 값이 유지됨	RefCell<T>가 불변이어도 내부의 값은 가변으로 사용 가능

> RefCell는 멀티스레드 코드에서는 작동하지 않는다는 점에 유의하세요! Mutex는 스레드에 안전한 RefCell<T의 버전이며, Mutex<T에 대해서는 나중에 설명하겠습니다.

Quiz

다음 코드가 컴파일되도록 `Wrapper` 타입과 `wrap` 함수를 완성합니다.

```
use std::cell::RefCell;
use std::fmt::Display;
use std::rc::Rc;
use std::vec::Vec;

type Wrapper<T> = 🔥;

fn wrap<T>(data: T) -> Wrapper<T> {
    🔥
}

#[derive(Debug)]
struct Node<T> {
    data: T,
    children: Vec<Wrapper<Node<T>>>,
}
```

```
impl<T: Display> Node<T> {
    fn add_child(&mut self, child: Wrapper<Node<T>>) {
        self.children.push(child);
    }
    fn new(data: T) -> Node<T> {
        Node {
            data,
            children: Vec::new(),
        }
    }
    fn depth_first(&self) {
        println!("node {}", self.data);
        for child in self.children.iter() {
            child.borrow().depth_first();
        }
    }
}

fn main() {
    let a = wrap(Node::new('A'));
    let b = wrap(Node::new('B'));
    let c = wrap(Node::new('C'));
    let d = wrap(Node::new('D'));
    a.borrow_mut().add_child(Rc::clone(&b));
    a.borrow_mut().add_child(Rc::clone(&c));
    b.borrow_mut().add_child(Rc::clone(&d));
    a.borrow_mut().depth_first();
}
```

정답

```rust
use std::cell::RefCell;
use std::fmt::Display;
use std::rc::Rc;
use std::vec::Vec;

type Wrapper<T> = Rc<RefCell<T>>;
fn wrap<T>(data: T) -> Wrapper<T> {
    Rc::new(RefCell::new(data))
}

#[derive(Debug)]
struct Node<T> {
    data: T,
    children: Vec<Wrapper<Node<T>>>,
}

impl<T: Display> Node<T> {
    fn add_child(&mut self, child: Wrapper<Node<T>>) {
        self.children.push(child);
    }
    fn new(data: T) -> Node<T> {
        Node {
            data,
            children: Vec::new(),
        }
    }
    fn depth_first(&self) {
        println!("node {}", self.data);
        for child in self.children.iter() {
            child.borrow().depth_first();
        }
    }
}

fn main() {
    let a = wrap(Node::new('A'));
    let b = wrap(Node::new('B'));
    let c = wrap(Node::new('C'));
    let d = wrap(Node::new('D'));
    a.borrow_mut().add_child(Rc::clone(&b));
    a.borrow_mut().add_child(Rc::clone(&c));
    b.borrow_mut().add_child(Rc::clone(&d));
    a.borrow_mut().depth_first();
}
```