

# Понимание и реализация смарт-указателя Arc и мьютекса на Rust

28.07.2024


Rust — язык системного программирования с акцентом на безопасности, многопоточности, производительности. В этом руководстве рассмотрим два примитива многопоточности Rust: **Arc** и **Mutex**.

При написании многопоточного Rust рано или поздно встречаются типы **Arc** и **Mutex**.

**Mutex** применяется во многих языках, а вот **Arc** вряд ли найдется где-то еще, кроме Rust. Нельзя полностью понять эти концепции, не связав их с моделью владения Rust. Эта статья — мой подход к пониманию **Arc** и **Mutex** в Rust.

Когда в многопоточной среде обмениваются данными, обычно передают их как сообщения или совместно используют память. В условиях многопоточности передача сообщений, например, по каналам предпочтительнее, но из-за модели владения различия в безопасности и корректности в Rust не так велики, как в других языках. То есть в безопасном Rust гонки данных невозможны. Поэтому основной критерий при выборе между передачей сообщений и совместным использованием памяти на Rust — удобство, а не безопасность.

РЕКЛАМА




trust-dubai-emirates.c

**Элитные  
квартиры  
в Дубае  
от 200 000 \$**

Узнать больше

РЕКЛАМА




bs.jewelry

**Серьги  
с муассанитом  
из США**

Узнать больше

РЕКЛАМА




iworld.com

**Оформить  
гражданство  
Армении. За 6  
месяцев.**

Подать заявку

РЕКЛАМА



oneandonlyresorts.co

**Курорт One&Only  
One Za'abeel  
в Дубае:  
роскошный отдых**

Забронировать

Если выбрать для обмена данными совместное использование памяти, быстро обнаруживается, что без **Arc** и **Mutex** здесь мало что делается. **Arc** — **умный указатель для совместного, безопасного использования потоками значения**. **Mutex** — **обертка над другим типом** для безопасной изменяемости в потоках. Чтобы полностью понять эти концепции, рассмотрим модель владения.

## Владение на Rust

Вот характеристики модели владения Rust:

- у значения имеется только один владелец;
- общих неизменяемых ссылок на значение может быть несколько;
- изменяемая ссылка на значение может быть только одна.



```

use std::thread::spawn;

#[derive(Debug)]
struct User {
    name: String
}

fn main() {
    let user = User { name: "sam".to_string() };

    spawn(move || {
        println!("Hello from the first thread {}", user.name);
    }).join().unwrap();
}

```

Пока все хорошо, программа компилируется с выводом сообщения. Добавим второй поток, также с доступом к экземпляру `user` :

```

fn main() {
    let user = User { name: "sam".to_string() };

    let t1 = spawn(move || {
        println!("Hello from the first thread {}", user.name);
    });

    let t2 = spawn(move || {
        println!("Hello from the second thread {}", user.name);
    });

    t1.join().unwrap();
    t2.join().unwrap();
}

```

С этим кодом получаем такую ошибку:

```

error[E0382]: use of moved value: `user.name`
--> src/main.rs:15:20
   |
11 |     let t1 = spawn(move || {
   |                     ^^^^^^ value moved into closure here
12 |         println!("Hello from the first thread {}", user.name);
   |                                     ^^^^^^^^^ variable mov
...
15 |     let t2 = spawn(move || {
   |                     ^^^^^^ value used here after move
16 |         println!("Hello from the second thread {}", user.name);
   |                                     ^^^^^^^^^ use occurs d
   |
   = note: move occurs because `user.name` has type `String`, which does not i

```

Что нужно компилятору? Ошибка здесь такая: `use of moved value` («Использование перемещенного значения `user.name`»). Компилятором даже указываются конкретные места, где возникает проблема. Сначала перемещаем значение в первый поток, затем пытаемся во второй.

Если посмотреть на правила владения, в этом нет ничего удивительного. **У значения имеется только один владелец.** В текущей версии кода нужно с помощью `move` переместить значение, которое планируется использовать, в первый поток, поэтому в другой поток значение переместить нельзя. Владение им уже поменялось. Но мы же не меняем данные, поэтому может быть **несколько общих ссылок**:

```
fn main() {
    let user = User { name: "sam".to_string() };

    let t1 = spawn(|| {
        println!("Hello from the first thread {}", &user.name);
    });

    let t2 = spawn(|| {
        println!("Hello from the second thread {}", &user.name);
    });

    t1.join().unwrap();
    t2.join().unwrap();
}
```

Здесь в замыканиях потоков удалили ключевое слово `move`, потоками неизменяемо заимствуется значение `user`. То есть получается общая ссылка, представленная амперсандом. С этим кодом получаем:

```
error[E0373]: closure may outlive the current function, but it borrows `user.n
--> src/main.rs:15:20
|
15 |     let t2 = spawn(|| {
|                   ^^ may outlive borrowed value `user.name`
16 |         println!("Hello from the first thread {}", &user.name);
|                                                     ----- `user.name`
|
note: function requires argument type to outlive `static`
--> src/main.rs:15:14
|
15 |     let t2 = spawn(|| {
|     _____^
16 | |         println!("Hello from the second thread {}", &user.name);
17 | |     });
| |     _____^
help: to force the closure to take ownership of `user.name` (and any other ref
|
```



```
15 |         let t2 = spawn(move || {
    |                               +++++
```

Теперь ошибкой указывается на то, что замыкание способно «пережить» функцию. Иными словами, компилятором Rust не гарантируется, что замыкание в потоке завершится до функции `main()`.

Структура `user` заимствуется потоками, но владение ею остается за функцией `main`. В этом сценарии, если функция `main` завершается, структура `user` выходит из области видимости, и память удаляется. Поэтому, если таким образом делиться значением с потоками, поток попытается считать освобожденную память. Это неопределенное поведение, и оно, конечно, нежелательно.

В примечании также говорится: чтобы избежать заимствования, переменную `user` можно переместить в поток, но из этого сценария мы и идем, нет смысла в него возвращаться. Здесь имеется два простых решения, одно из них — `Arc`, но рассмотрим сначала другое.

## Потоки области видимости

Потоки области видимости — это функционал, доступный из отличного крейта `crossbeam` или как экспериментальная ночная функция на Rust. Воспользуемся `crossbeam`, но API обеих версий очень похожи.

Добавив `crossbeam = "0.8"` в зависимости `Cargo.toml`, получаем бесперебойный рабочий код:

```
use crossbeam::scope;

#[derive(Debug)]
struct User {
    name: String,
}

fn main() {
    let user = User {
        name: "sam".to_string(),
    };

    scope(|s| {
        s.spawn(|_| {
            println!("Hello from the first thread {}", &user.name);
        });

        s.spawn(|_| {
            println!("Hello from the second thread {}", &user.name);
        });
    })
    .unwrap();
}
```



Все потоки, созданные в области видимости, гарантированно завершаются до завершения замыкания `scope`. То есть, прежде чем замыкание выйдет из области видимости, потоки объединяются в ожидании завершения. Благодаря этому компилятор «знает», что ни одно из заимствований не «переживет» владельца.

Интересно, что для человека обе эти программы допустимы: в версии, отвергаемой Rust, мы объединяем оба потока до завершения функции `main()`, поэтому делиться значением `user` с потоками на самом деле безопасно. Такое в Rust случается. Невозможно написать компилятор для приема всех допустимых программ, альтернатива — **суперстрогий компилятор, которым отклоняются все недопустимые**. *Потоки в области видимости созданы специально для этого — писать код, принимаемый компилятором.*

Но, как бы ни были хороши потоки области видимости, использовать их не всегда возможно. Например, при написании асинхронного кода. Вернемся к первому решению.

## Аrc в помощь

`Arc` — это умный указатель для обмена данными между потоками, расшифровывается как *atomic reference counter*, то есть атомарный подсчет ссылок.

Фактически задача `Arc` — обернуть значение, которым мы пытаемся поделиться, и быть указателем на него. Этим `Arc` отслеживаются все копии указателя, и по выходе последнего указателя из области видимости безопасно освобождается память.

Вот как `Arc` решается описанная выше проблема:

```
use std::thread::spawn;
use std::sync::Arc;

#[derive(Debug)]
struct User {
    name: String
}

fn main() {
    let user_original = Arc::new(User { name: "sam".to_string() });

    let user = user_original.clone();
    let t1 = spawn(move || {
        println!("Hello from the first thread {}", user.name);
    });

    let user = user_original.clone();
    let t2 = spawn(move || {
        println!("Hello from the first thread {}", user.name);
    });

    t1.join().unwrap();
    t2.join().unwrap();
}
```



Рассмотрим подробнее.

Сначала создаем значение `user` и оборачиваем его в `Arc`. Теперь оно сохраняется в памяти, а `Arc` всего лишь указатель.

При каждом клонировании копируется не значение `user`, а только ссылка. Клонировав `Arc`, мы перемещаем в каждый из потоков копию указателя. Благодаря `Arc` данные обмениваются независимо от времен жизни.

В этом примере создается три указателя на значение `user`: один при создании `Arc`, второй клонированием перед запуском первого потока, в который он и перемещается, а третий клонированием перед запуском второго потока — тоже перемещается в первый поток.

**Пока хоть один указатель активен, память в Rust не освободится.** Когда же завершаются потоки и функция `main`, все указатели `Arc` выходят из области видимости и удаляются. С последним из них удаляется и значение `user`.

## Send и Sync

Копнем глубже. Согласно документации, типы `Send` и `Sync` реализуются в `Arc`, только если реализуются и оборачиваемым типом. Чтобы разобраться, начнем с определения `Send` и `Sync`.

В *Rustonomicon* `Send` и `Sync` определяются так:

- Если тип безопасно отправляется в другой поток, это `Send`.
- Если тип безопасно обменивается между потоками, это `Sync`; `T` является `Sync`, только когда `&T` является `Send`.

Подробнее об этих типажах — в *Rustonomicon*, но попробуем разобраться самостоятельно.

`Send` и `Sync` — типажи-маркеры без реализованных методов, им ничего не требуется реализовывать. Компилятор уведомляется ими о возможности обмениваться типом или отправлять тип между потоками.

Начнем с `Send`, он попроще. Нельзя отправить в другой поток тип `!Send`, то есть не `Send`: нельзя отправить его по каналу или переместить в поток. Например, этот код не скомпилируется:

```
#![feature(negative_impls)]

#[derive(Debug)]
struct Foo {}

impl !Send for Foo {}

fn main() {
    let foo = Foo {};
    spawn(move || {
        dbg!(foo);
    });
}
```



```
});  
}
```

`Send` и `Sync` выводятся автоматически. Например, если все атрибуты типа являются `Send`, этот тип будет тоже `Send`. В коде экспериментальным функционалом `negative_impls` компилятору сообщается о намерении явно обозначить этот тип как `!Send`.

В итоге появляется ошибка:

```
`Foo` cannot be sent between threads safely
```

То же происходит при создании канала для отправки `foo` в поток. С `Arc` таким же образом вылетает та же ошибка. И то же справедливо для типа `!Sync`, поскольку `Arc` нужны оба типажа:

```
#![feature(negative_impls)]  
  
#[derive(Debug)]  
struct Foo {}  
impl !Send for Foo {}  
  
fn main() {  
    let foo = Arc::new(Foo {});  
    spawn(move || {  
        dbg!(foo);  
    });  
}
```

Но разве `Arc` не должен обернуть тип и предоставить больше возможностей? Верно, но с `Arc` тип не делается потокобезопасным как по волшебству. Почему? Покажем в подробном примере в конце статьи, а пока продолжим изучать применение этих типов.

Мы знаем теперь, что благодаря `Arc` потоки независимо от времен жизни обмениваются ссылками на типы, которые являются `Send + Sync`. Ведь это не обычная ссылка, а умный указатель.

## Изменение данных с помощью Mutex

Мьютексы во многих языках рассматриваются как семафоры. Создавая мьютексный объект, мы защищаем с помощью `mutex` конкретную часть или части кода. Так что защищаемое место одновременно доступно только одному потоку.

В Rust `Mutex` — это скорее обертка. **Доступ к базовому значению предоставляется ею только после блокировки мьютекса.** Обмен значения между потоками упрощается этим `Mutex` с помощью `Arc`.

Вот пример:



```

use std::time::Duration;
use std::{thread, thread::sleep};
use std::sync::{Arc, Mutex};

struct User {
    name: String
}

fn main() {
    let user_original = Arc::new(Mutex::new(User { name: String::from("sam") } )

    let user = user_original.clone();
    let t1 = thread::spawn(move || {
        let mut locked_user = user.lock().unwrap();
        locked_user.name = String::from("sam");
        // После того как «locked_user» выйдет из области видимости, мьютекс с
        // Чтобы разблокировать его явно, применяется
        // «drop(locked_user)».
    });

    let user = user_original.clone();
    let t2 = thread::spawn(move || {
        sleep(Duration::from_millis(10));

        // Выведется «Hello sam»
        println!("Hello {}", user.lock().unwrap().name);
    });

    t1.join().unwrap();
    t2.join().unwrap();
}

```

Разберем этот код. В первой строке функции `main()` создается экземпляр структуры `User`, оборачиваемый в `Mutex` и `Arc`. С `Arc` указатель легко копируется, поэтому мьютекс задействуется потоками совместно. После того как мьютекс блокируется, базовое значение используется исключительно этим потоком. В следующей строке это значение меняется. Мьютекс разблокируется, как только защищенная, блокированная часть кода выходит из области видимости, или удаляется вручную с помощью `drop(locked_user)`.

Во втором потоке через 10 мс ожидания выводится название, обновленное в первом потоке. На этот раз блокировка выполняется в одной строке, поэтому мьютекс удаляется в том же операторе.

Необходимо упомянуть также о методе `unwrap()`, вызываемом после `lock()`. В `Mutex` стандартной библиотеки заложено понятие об отравлении. Если поток «паникует» при заблокированном мьютексе, нельзя определить, остается ли значение внутри `Mutex` валидным. Поэтому поведение по умолчанию — возвращение ошибки, а не защищенной части кода. Причем этим `Mutex` возвращается вариант `Ok()` с обернутым значением в качестве аргумента либо ошибка. Подробнее об этом — в документации.





В целом оставлять методы `unwrap()` в производственном коде не рекомендуется, но в случае с `Mutex` это рабочая стратегия: если мьютекс отравлен, состояние приложения бывает недопустимым, тогда работа приложения аварийно завершается.

Интересно и вот что: пока тип внутри `Mutex` является `Send`, мьютекс будет также и `Sync`. Ведь мьютексом обеспечивается доступ к базовому значению только для одного потока, поэтому совместное использование `Mutex` безопасно для потоков.

## Mutex : добавление Sync к типу Send

Напомним: чтобы `Arc` стал `Send + Sync`, ему нужен базовый тип `Send + Sync`. А вот, чтобы `Mutex` стал `Send`, требуется только базовый тип `Send`. То есть с `Mutex` тип `!Sync` становится `Sync`, обменивается между потоками, а также изменяется.

## Mutex без Arc

Что, если использовать `Mutex` без `Arc`? Подумайте, что означает то, что `Mutex` — это `Send + Sync` для типов `Send`?

Очень похоже на то, что это означает для типа `Arc`. Если применять что-то вроде потоков области видимости, `Mutex` обходится без `Arc`:

```
use crossbeam::scope;
use std::{sync::Mutex, thread::sleep, time::Duration};

#[derive(Debug)]
struct User {
    name: String,
}

fn main() {
    let user = Mutex::new(User {
        name: "sam".to_string(),
    });

    scope(|s| {
        s.spawn(|_| {
            user.lock().unwrap().name = String::from("psaa");
        });

        s.spawn(|_| {
            sleep(Duration::from_millis(10));

            // выводится «Hello psaa»
            println!("Hello {}", user.lock().unwrap().name);
        });
    })
    .unwrap();
}
```



В этой программе достигается та же цель: доступ к значению позади мьютекса получается в двух отдельных потоках, но мьютексы используются ими совместно по ссылке, и без `Arc`. Опять же, это не всегда возможно, например, в асинхронном коде, поэтому `Mutex` очень часто применяется вместе с `Arc`.

## Заключение

Мы изучили типы `Arc` и `Mutex` в Rust. `Arc`, как правило, используется при невозможности обмена данными между потоками с помощью обычных ссылок. Для изменения данных, которыми обмениваются потоки, применяется `Mutex`. Если же при этом мьютекс не разделяется посредством ссылок, используется `Arc<Mutex<...>>`.

## Бонус: почему Arc нужен тип Sync?

Вернемся к вопросу о том, почему `Arc` нужно, чтобы базовый тип являлся и `Send`, и `Sync`, помечался как `Send` и `Sync`. Концовку статьи можете пропустить, поскольку `Arc` и `Mutex` в коде не особо востребованы. Но для понимания типажей-маркеров она придется кстати.

Возьмем в качестве примера `Cell`, которым обортывается другой тип и обеспечивается внутренняя изменяемость, то есть возможность изменять значение внутри неизменяемой структуры. Тип `Cell` — `Send`, но это `!Sync`.

Вот пример:

```
use std::cell::Cell;

struct User {
    age: Cell<usize>
}

fn main() {
    let user = User { age: Cell::new(30) };

    user.age.set(36);

    // выведется «Age: 36»
    println!("Age: {}", user.age.get());
}
```

`Cell` полезен в некоторых ситуациях, но не потокобезопасен, то есть это `!Sync`. Если значение, обернутое в `cell`, каким-то образом обменивается между потоками, то же место в памяти изменяется из двух потоков:

```
// этот пример не скомпилируется, «Cell» — это «!Sync», поэтому
// «Arc» будет «!Sync» и «!Send»
use std::cell::Cell;

struct User {
    age: Cell<usize>
}
```



```
fn main() {
    let user_original = Arc::new(User { age: Cell::new(30) });

    let user = user_original.clone();
    std::thread::spawn(move || {
        user.age.set(2);
    });

    let user = user_original.clone();
    std::thread::spawn(move || {
        user.age.set(3);
    });
}
```

Такой код чреват неопределенным поведением. Поэтому **Arc** не рабочий с любыми типами, кроме **Send** и **Sync**. Но **Cell** — это **Send**, поэтому отправляется между потоками. Дело в том, что отправкой или перемещением значение не делается доступным более чем из одного потока, такой поток всегда только один. Как только значение перемещается в другой, предыдущему потоку оно уже не принадлежит. Учитывая это, мы всегда можем изменить **Cell** локально.

## Бонус: зачем Arc нужен тип

А нет ли у **Arc** типажа **Send** и для **!Send**? **Rc** — один из типов Rust, который является **!Send**. В отличие от **Arc**, он не атомарный **Rc**, расширяется лишь до счетчика ссылок и при практически той же роли **Arc** используется только в одном потоке. Он не обменивается и даже не перемещается между потоками, посмотрим почему:

```
// этот код не скомпилируется, Rc является «!Send» и «!Sync»
use std::rc::Rc;

fn main() {
    let foo = Rc::new(1);

    let foo_clone = foo.clone();
    std::thread::spawn(move || {
        dbg!(foo_clone);
    });

    let foo_clone = foo.clone();
    std::thread::spawn(move || {
        dbg!(foo_clone);
    });
}
```

Этот пример не компилируется, потому что **Rc** — **!Sync + !Send**. Его внутренний счетчик не атомарный, поэтому обмен им между потоками чреват неточным подсчетом ссылок.

Если же в **Arc** типы **!Send** сделаются **Send**:



```

use std::rc::Rc;
use std::sync::Arc;

#[derive(Debug)]
struct User {
    name: Rc<String>,
}

unsafe impl Send for User {}
unsafe impl Sync for User {}

fn main() {
    let foo = Arc::new(User {
        name: Rc::new(String::from("drogus")),
    });

    let foo_clone = foo.clone();
    std::thread::spawn(move || {
        let name = foo_clone.name.clone();
    });

    let foo_clone = foo.clone();
    std::thread::spawn(move || {
        let name = foo_clone.name.clone();
    });
}

```

Теперь пример компилируется, только не делайте так в коде. Здесь определяется структура `User` с `Rc` внутри. Поскольку `Send` и `Sync` выводятся автоматически, а `Rc` — `!Send` + `!Sync`, структура `User` тоже `!Send` + `!Sync`, но компилятору явно указывается обозначить ее по-другому, в данном случае `Send` + `Sync` с синтаксисом `unsafe impl`.

Теперь видно, что пойдет не так, если разрешить в `Arc` перемещение типов `!Send` между потоками. В примере клоны `Arc` перемещаются в отдельные потоки, после чего ничто не мешает клонировать тип `Rc`. А поскольку тип `Rc` не потокобезопасный, это чревато неточным подсчетом ссылок. Следовательно, память освободится слишком рано либо не освободится вовсе, хотя и должна.

Читайте также:

- [Изучаем Rust. Поточковая передача tar-архива](#)
- [Ошибки в Rust: формула](#)
- [Борьба с веб-скрейперами с помощью Rust](#)

Читайте нас в [Telegram](#), [VK](#) и [Дзен](#)

---

Перевод статьи [nollan fabianski: Understanding and Implementing Rust's Arc and Mutex](#)

