

# Параллелизм в Rust


# Явные потоки

Явный поток в Rust - это механизм, который позволяет запустить параллельную или конкурентную работу. Он предоставляет возможность выполнения кода в отдельном потоке, а не в главном потоке исполнения программы.

Модуль `std::thread` является частью стандартной библиотеки Rust и предоставляет функциональность для работы с явными потоками. Он позволяет создавать, управлять и совместно использовать потоки.

# Создание потока

Для создания нового явного потока в Rust используется метод `spawn` из модуля `std::thread`. Пример создания нового потока:



```
1 fn main() {  
2     let (tx, rx) = mpsc::channel();  
3     let handle = thread::spawn(move || {  
4         let data = "Привет из отдельного потока!";  
5         tx.send(data).unwrap();  
6     });  
7     let received = rx.recv().unwrap();  
8     println!("{}", received);  
9     handle.join().unwrap();  
10 }
```

# Проблемы

Поскольку потоки могут работать одновременно, нет чёткой гарантии, определяющей порядок выполнения частей вашего кода в разных потоках. Это может привести к таким проблемам, как:


- Состояния гонки, когда потоки обращаются к данным, либо ресурсам, несогласованно.
- Взаимные блокировки, когда два потока ожидают друг друга, не позволяя тем самым продолжить работу каждому из потоков.
- Ошибки, которые случаются только в определённых ситуациях, которые трудно воспроизвести и, соответственно, трудно надёжно исправить.

# Пример

Когда основной поток программы на Rust завершается, все порождённые потоки закрываются, независимо от того, завершили они работу или нет

## Console:

hi number 1 from the main thread!  
hi number 1 from the spawned thread!  
hi number 2 from the main thread!  
hi number 2 from the spawned thread!  
hi number 3 from the main thread!  
hi number 3 from the spawned thread!  
hi number 4 from the main thread!  
hi number 4 from the spawned thread!



```
1 fn main() {  
2     thread::spawn(|| {  
3         for i in 1..10 {  
4             println!("hi number {} from the spawned thread!", i);  
5             thread::sleep(Duration::from_millis(1));  
6         }  
7     });  
8  
9     for i in 1..5 {  
10        println!("hi number {} from the main thread!", i);  
11        thread::sleep(Duration::from_millis(1));  
12    }  
13 }
```

# Ожидание завершения явного потока

Ожидание завершения явного потока с помощью метода `join` позволяет программе дождаться окончания выполнения потока перед продолжением работы. Если поток завершился успешно, метод `join` возвращает `Result<()>`. Если произошла ошибка, возвращается `Result>`, который можно обработать с помощью `unwrap` или других методов обработки ошибок.

# Ожидание завершения явного потока

В примере мы вызвали метод `panic!` внутри потока, что приводит к необработанной ошибке. Мы обработали ошибку, вызывая метод `join` и проверяя результат на наличие ошибки в блоке `if let Err(error)`.

```
use std::thread;

fn main() {
    let handle = thread::spawn(|| {
        // Код, выполняющийся в новом потоке
        panic!("Что-то пошло не так!");
    });

    let result = handle.join();
    if let Err(error) = result {
        println!("Произошла ошибка: {:?}", error);
    }
}
```

# Механизм "Result"

Механизм "Result" в Rust используется для работы с ошибками и исключительными ситуациями. Он позволяет явно обрабатывать возможные ошибки, а не просто вызывать "panic" и завершать выполнение программы.

```
enum Result<T, E> {  
    Ok(T),  
    Err(E),  
}
```



# Передача данных

Данные могут передаваться между явными потоками с использованием механизма синхронизации, такого как мьютексы или каналы. Мьютексы позволяют контролировать доступ к данным из нескольких потоков, а каналы предоставляют средство обмена сообщениями между потоками.



```
1 fn main() {
2     let (tx, rx) = mpsc::channel();
3     let handle = thread::spawn(move || {
4         let data = "Привет из отдельного потока!";
5         tx.send(data).unwrap();
6     });
7     let received = rx.recv().unwrap();
8     println!("{}", received);
9     handle.join().unwrap();
10 }
```

# move замыкания

- Замыкание получает из окружения права владения на используемые им значения, таким образом передавая права владения этими значениями от одного потока к другому



```
1 fn main() {  
2     let v = vec![1, 2, 3];  
3  
4     let handle = thread::spawn(move || {  
5         println!("Here's a vector: {:?}", v);  
6     });  
7  
8     handle.join().unwrap();  
9 }
```

# Каналы

- Канал состоит из двух половин: передатчика и приёмника.
- Канал считается **закрытым** , если либо передающая, либо принимающая его половина уничтожена.
- Функция **mpsc::channel** возвращает кортеж: **tx** и **rx**
- Передатчик имеет метод **send** , который принимает значение, которое мы хотим отправить. Метод **send** возвращает тип **Result<T, E>**
- Получатель имеет два важных метода: **recv** и **try\_recv**.
- **recv** блокирует выполнение основного потока и ждёт, пока данные не будут переданы по каналу
- **try\_recv** не блокирует, а сразу возвращает результат **Result<T, E>**: значение **Ok**, содержащее сообщение, если оно доступно или значение **Err**, если никаких сообщений не поступило

```
1 fn main() {
2     let (tx, rx) = mpsc::channel();
3
4     thread::spawn(move || {
5         let val = String::from("hi");
6         tx.send(val).unwrap();
7     });
8
9     let received = rx.recv().unwrap();
10    println!("Got: {}", received);
11 }
```

# Создание нескольких отправителей путём клонирования передатчика

Создаём несколько потоков, которые отправляют значения одному и тому же получателю.

Функция `clone` - получим новый передатчик, который мы сможем передать первому порождённому потоку. Исходный передатчик мы передадим второму порождённому потоку.

Итого: 2 потока, каждый из которых отправляет разные сообщения одному получателю

```
1 fn main() {  
2     let (tx, rx) = mpsc::channel();  
3     let tx1 = tx.clone();  
4     thread::spawn(move || {  
5         let vals = vec![  
6             String::from("hi"),  
7             String::from("from"),  
8             String::from("the"),  
9             String::from("thread"),  
10        ];  
11        for val in vals {  
12            tx1.send(val).unwrap();  
13            thread::sleep(Duration::from_secs(1));  
14        }  
15    });  
16    thread::spawn(move || {  
17        let vals = vec![  
18            String::from("more"),  
19            String::from("messages"),  
20            String::from("for"),  
21            String::from("you"),  
22        ];  
23        for val in vals {  
24            tx.send(val).unwrap();  
25            thread::sleep(Duration::from_secs(1));  
26        }  
27    });  
28    for received in rx {  
29        println!("Got: {}", received);  
30    }  
31 }
```

# Мьютексы

- Мьютексы - это средство для синхронизации доступа к общим данным между несколькими потоками выполнения.
- Они являются объектами синхронизации, которые позволяют только одному потоку получить доступ к защищаемым данным в определенный момент времени.
- Мьютексы работают по принципу владения и аренды данных. Они позволяют получить мутабельную ссылку (изменяемую ссылку) на данные только одному потоку в определенный момент времени. При этом остальным потокам доступ к данным будет запрещен до тех пор, пока не будет снята блокировка.
- Мьютексы в Rust применяются для предотвращения состояния гонки, когда несколько потоков пытаются одновременно изменять общие данные. Использование мьютексов позволяет управлять доступом к данным таким образом, чтобы только один поток мог изменять их в конкретный момент времени.

# Правила использования мьютексов

- Перед тем как попытаться получить доступ к данным необходимо получить блокировку.
- Когда вы закончили работу с данными, которые защищает мьютекс, вы должны разблокировать данные, чтобы другие потоки могли получить блокировку.

# Разделение Mutex<T> между множеством потоков

- Мы стартуем 10 потоков и каждый из них увеличивает значение счётчика на 1, поэтому счётчик изменяется от 0 до 10
- Ошибка: не можем передать counter во владение нескольким потокам  $\Rightarrow$  нужно использовать **Rc<T>** (**Arc<T>**) для множественного владения

```
1 fn main() {
2     let counter = Mutex::new(0);
3     let mut handles = vec![];
4
5     for _ in 0..10 {
6         let handle = thread::spawn(move || {
7             let mut num = counter.lock().unwrap();
8
9             *num += 1;
10        });
11        handles.push(handle);
12    }
13
14    for handle in handles {
15        handle.join().unwrap();
16    }
17
18    println!("Result: {}", *counter.lock().unwrap());
19 }
```

## Разрешение передачи во владение между потоками с помощью Send

- Две концепции многопоточности: **std::marker** типы **Sync** и **Send**
- **Send** указывает, что владение типом реализующим **Send**, может передаваться между потоками
- Почти каждый тип Rust является типом **Send**,
- Исключение: **Rc<T>**, т.к. если вы клонировали значение **Rc<T>** и попытались передать владение клоном в другой поток, оба потока могут обновить счётчик ссылок одновременно



# Разрешение доступа из нескольких потоков с Sync

- Sync указывает, что на тип реализующий **Sync** можно безопасно ссылаться из нескольких потоков
- Любой тип **T** является типом **Sync**, если **&T** является типом **Send**, что означает что ссылку можно безопасно отправить в другой поток
- Примитивные типы являются типом Sync, а типы полностью скомбинированные из типов Sync, также являются Sync типом.

# Распараллеливание данных

- В Rust существуют различные концепции и инструменты для распараллеливания данных. Некоторые из них включены в стандартную библиотеку языка, а другие предоставляются дополнительными крейт-библиотеками.
- Поддержка многопоточности в стандартной библиотеке Rust и в дополнительных крейт-библиотеках
- В стандартной библиотеке Rust существует поддержка многопоточности с использованием типов данных, таких как `Mutex`, `RwLock` и `Atomic`.

# Параллельные коллекции в Rust

- `std::sync::mpsc` (Multiple Producer, Single Consumer) используется для обмена сообщениями между несколькими потоками. Она предоставляет каналы для отправки сообщений от нескольких потоков к одному потоку-получателю.
- `std::sync::Arc` (Atomic Reference Counted) - это синхронизированный умный указатель, который позволяет нескольким потокам иметь доступ к одному и тому же значению безопасно. Он использует подсчет ссылок для определения времени жизни объекта.
- `std::sync::Mutex` - это блокировка, которая позволяет только одному потоку иметь доступ к данным в определенный момент времени. При обращении к данным он блокирует остальные потоки, пока первый поток не закончит свою работу.

# Параллельные коллекции в Rust

- `rayon` - это библиотека для параллельного программирования в Rust, которая предоставляет параллельные версии стандартных методов коллекций, таких как `iter()` и `map()`. Она автоматически разбивает коллекции на части и выполняет операции на разных частях параллельно, используя все доступные ядра процессора.
- `crossbeam` - еще одна параллельная библиотека в Rust, которая предоставляет параллельные версии некоторых стандартных коллекций, таких как `Vec` и `HashMap`. Эта библиотека также предоставляет более низкоуровневые примитивы для синхронизации, блокировки и обмена данными между потоками.

Спасибо за внимание!

