

Rust: Fearless Concurrency and Trivial Parallelism

Concurrency vs Parallelism

- ▶ Concurrency
 - ▶ Doing things at **different times** or **out of order**
 - ▶ Async IO
 - ▶ Interrupts
 - ▶ User Interface
- ▶ Parallelism
 - ▶ Doing multiple things **at the same time**
 - ▶ Multi-Threading
 - ▶ Multi-Processing
- ▶ Rust doesn't make much of a distinction here
- ▶ Because it deals with it the same way

Rc

- ▶ Reference Counted Pointer
- ▶ Takes ownership of the value
- ▶ Handles desctruction when ref-count becomes zero
- ▶ Only hands out **read-only** references usually
- ▶ **!Send + !Sync**: Not thread safe!

```
struct Rc<T> {  
    strong: usize,  
    weak: usize  
    value: T  
}
```

RefCell

- ▶ Run-time Borrow Checker
- ▶ Gives out **many readable XOR one writable references** to interior value
- ▶ References wrapped in a guard and can't escape
- ▶ **Send + !Sync**: Sendable, but no shared references

```
pub struct RefCell<T: ?Sized>
    borrow: Cell<BorrowFlag>,
    value: UnsafeCell<T>,
}
```

Callbacks

```
1 struct Callbacks {
2     on_click: Box<dyn FnMut()>,
3     on_dbl_click: Box<dyn FnMut()>
4 }
5
6 fn main() {
7     let mut click_string = String::new();
8
9     let callbacks = Callbacks {
10         on_click:
11             Box::new(|| { // This is a closure
12                 let click_string = &mut click_string; //
13                 click_string.push_str("Clicked! "); //
14             } ), // End of closure
15
16         on_dbl_click:
17             Box::new(|| { // This is a closure
18                 let click_string = &mut click_string; //
19                 click_string.push_str("Doubleclicked! "); //
20             } ), // End of closure
21     };
22
23 }
```

Callbacks

```
6 fn main() {
7     let mut click_string = String::new();
8
9     let callbacks = Callbacks {
10         on_click:
11         Box::new(|| { // This is a closure
12             value captured here
13             cast requires that `click_string` is borrowed for `static`
14             let click_string = &mut click_string; //
15             // `click_string` does not live long enough (borrowed value does not live long enough)
16             click_string.push_str("Clicked! "); //
17             // End of closure
18
19         on_dbl_click:
20         Box::new(|| { // This is a closure
21             value captured here
22             cast requires that `click_string` is borrowed for `static`
23             let click_string = &mut click_string; //
24             // `click_string` does not live long enough (borrowed value does not live long enough)
25             click_string.push_str("Doubleclicked! "); //
26             // End of closure
27         } ),
28     };
29 }
30
31 // `click_string` dropped here while still borrowed
```

Callbacks

```
6 fn main() {
7     let mut click_string = String::new();
8
9     let callbacks = Callbacks {
10         on_click:
11             Box::new(|_| { // This is a closure
12                 let click_string = &mut click_string; //
13                 click_string.push_str("Clicked! ") //
14             } ), // End of closure
15
16         on_dbl_click:
17             Box::new(|_| { // This is a closure
18                 value captured here
19                 cast requires that `click_string` is borrowed for `static`
20                 let click_string = &mut click_string; //
21                 // `click_string` does not live long enough (borrowed val-
22                 ue does not live long enough)
23                 click_string.push_str("Doubleclicked! "); //
24             } ), // End of closure
25     };
26 }
27 // `click_string` dropped here while still borrowed
```

Callbacks

```
1 struct Callbacks {
2     on_click: Box<dyn FnMut()>,
3     on_dbl_click: Box<dyn FnMut()>
4 }
5
6 fn main() {
7     let mut click_string = String::new();
8     move occurs because `click_string` has type `std::string::String`, which does not
    implement the `Copy` trait
9
10    Callbacks {
11        on_click:
12            Box::new(move || {                                // This is a closure
13                value moved into closure here
14                let click_string = &mut click_string;        //
15                variable moved due to use in closure
16                click_string.push_str("Clicked! ")            //
17            } ),                                              // End of closure
18
19        on_dbl_click:
20            Box::new(move || {                                // This is a closure
21                use of moved value: `click_string` (value used here after move)
22                let click_string = &mut click_string;        //
23                use occurs due to use in closure
24                click_string.push_str("Doubleclicked! ");    //
25            } ),                                              // End of closure
26    };
27 }
```


Callbacks

```
3 struct Callbacks {
4     on_click: Box<dyn FnMut(>,>,
5     on_dbl_click: Box<dyn FnMut(>,>
6 }
7
8 fn main() {
9     let mut click_string = Rc::new(String::new());
10    // Clone Rc-pointer, increases ref-count
11    let mut dbl_click_string = click_string.clone();
12
13    Callbacks {
14        on_click:
15            Box::new(move || {
16                let click_string = &mut click_string;
17                click_string.push_str("Clicked! ");
18                cannot borrow data in a `&` reference as mutable (cannot borrow as mutable)
19            } ),
20        on_dbl_click:
21            Box::new(move || {
22                let click_string = &mut dbl_click_string;
23                click_string.push_str("Doubleclicked! ");
24            } ),
25    };
26
27 }
```

Callbacks

```
5     on_click: Box<dyn FnMut()>,
6     on_dbl_click: Box<dyn FnMut()>
7 }
8
9 fn main() {
10     let click_string: Rc<RefCell<String>> = Rc::new(RefCell::new(String::new()));
11     // Clone Rc-pointer, increases ref-count
12     let dbl_click_string = click_string.clone();
13
14     Callbacks {
15         on_click:
16         Box::new(move || {
17             // borrow_mut() returns a wrapped mutable reference
18             let mut string_ref: RefMut<_> = click_string.borrow_mut();
19             string_ref.push_str("Clicked! ");
20             // reference wrapper dropped here, RefCell usable again
21         } ),
22
23         on_dbl_click:
24         Box::new(move || {
25             // borrow_mut() returns a wrapped mutable reference
26             let mut string_ref: RefMut<_> = dbl_click_string.borrow_mut();
27             string_ref.push_str("Doubleclicked! ");
28             // reference wrapper dropped here, RefCell usable again
29         } ),
30     };
31
32 }
```

Send/Sync Traits

- ▶ Marker Traits
- ▶ Send: Allows **moving** values to other threads
- ▶ Sync: Allows **sharing** references with other threads
- ▶ Most types are actually Send + Sync!

Sharing & Locking

Arc and Mutex

- ▶ Arc: Atomically counted Rc
 - ▶ Like Rc, can be cloned without copying data
 - ▶ Like Rc, only allows read access
 - ▶ Unlike Rc, can be moved and shared among threads
 - ▶ Send + Sync
- ▶ Mutex: Atomically locked RefCell
 - ▶ Like RefCell, takes ownership of the contained value
 - ▶ Like RefCell, ensures exclusive access to data
 - ▶ Unlike RefCell, doesn't panic when locking a locked value
 - ▶ Unlike RefCell, can be moved and shared among threads
 - ▶ Send + Sync

Channels

- ▶ Sending values to other parts of your code
- ▶ Sender **never blocks**, unless you use synchronous channels
- ▶ Receiver **always blocks** as long as Senders exist
- ▶ Great for simple event systems when combined with enums
- ▶ The stdlib mpsc channels allow one Receiver and multiple Senders

Channels

```
1 use std::sync::mpsc::channel;
2
3
4 fn main() {
5     let (tx, rx) = channel();
6
7     std::thread::spawn(move || {
8         for i in 0..100 {
9             tx.send(i).ok();
10        }
11    });
12
13    std::thread::spawn(move || {
14        for number in rx.iter() {
15            println!("{}", number);
16        }
17    });
18 }
```

Channels

```
7  std::thread::spawn(move || {
8      for i in 0..100 {
9          tx.send(i).ok();
10     }
11 });
12
13 std::thread::spawn(move || {
14     for number in rx.iter() {
15         println!("{}", number);
16     }
17 });
18 }
```

U:--- channels.rs Bot L19 (Rust FlyC cargo God)

[~/rede/snippets](#)# ./channels

-> 0

[~/rede/snippets](#)#

-> 0

U:%*- *ansi-term* All L2 (Term: char run)

Channels

```
1 use std::sync::mpsc::channel;
2
3
4 fn main() {
5     let (tx, rx) = channel();
6
7     std::thread::spawn(move || {
8         for i in 0..100 {
9             tx.send(i).ok();
10        }
11    });
12
13    let handle = std::thread::spawn(move || {
14        for number in rx.iter() {
15            println!("{}", number);
16        }
17    });
18
19    handle.join().ok();
20 }
```


Channels

```
10     }
11 });
12
13 let handle = std::thread::spawn(move || {
14     for number in rx.iter() {
15         println!("{}", number);
16     }
17 });
18
19 handle.join().ok();
20 }
```

U:--- channels.rs Bot L21 (Rust FlyC cargo God)

89
90
91
92
93
94
95
96
97
98
99

[\[~/rede/snippets\]](#)#

-> 0

U:%*- *ansi-term* Bot L13 (Term: char run)

Channels

```
1 use std::sync::mpsc::channel;
2
3
4 fn main() {
5     let (tx, rx) = channel();
6     let tx2 = tx.clone();
7
8     std::thread::spawn(move || {
9         for i in 0..100 {
10             tx.send(i).ok();
11         }
12     });
13
14     std::thread::spawn(move || {
15         for i in (0..100).rev() {
16             tx2.send(i).ok();
17         }
18     });
19
20     let handle = std::thread::spawn(move || {
21         for number in rx.iter() {
22             println!("{}", number);
23         }
24     });
25
26     handle.join().ok();
27 }
```

Channels

```
5  let (tx, rx) = channel();
6  let tx2 = tx.clone();
7
8  std::thread::spawn(move || {
9      for i in 0..100 {
10         tx.send(i).ok();
11     }
12 });
13
14 std::thread::spawn(move || {
15     for i in (0..100).rev() {
16         tx2.send(i).ok();
17     }
18 });
```

U:--- channels.rs 10% L16 (Rust FlyC cargo God)

4
3
94
2
95
1
96
0
97
98
99

[~/rede/snippets]#

-> 0

U:%*- *ansi-term* Bot L58 (Term: char run)

rayon/crossbeam

- ▶ `par_iter`
 - ▶ Can parallelize most iterations by just replacing `.iter()` with `.par_iter()`
 - ▶ Be careful about small loops, there's overhead due to locking/synchronization!
- ▶ Thread Pools
 - ▶ Global and local pools
- ▶ Scoped Threads
 - ▶ Share values on the stack with threads
 - ▶ Readable references in multiple threads
 - ▶ Writeable reference in one thread

Scoped Threads

```
1 fn main() {
2     let primes = [2,3,5,7,9,11,13,15]; // Some data on the stack
3
4     let handle = std::thread::spawn(|| {
5         let found_primes = (0..100)
6             .into_iter()
7             .filter(|num| {
8                 primes.contains(num) // Access stack data
9             }).collect::<Vec<usize>>();
10
11         println!("{:?}", found_primes);
12     });
13
14     let handle2 = std::thread::spawn(|| {
15         let found_primes = (0..100).rev()
16             .into_iter()
17             .filter(|num| {
18                 primes.contains(num) // Access stack data
19             }).collect::<Vec<usize>>();
20
21         println!("{:?}", found_primes);
22     });
23
24     handle.join(); // Wait until thread is done, so primes isn't dropped
25     handle2.join(); // Wait until thread is done, so primes isn't dropped
26 } // <- Dropping primes here
```

Scoped Threads

```
1 fn main() {
2     let primes = [2,3,5,7,9,11,13,15]; // Some data on the stack
3
4     let handle = std::thread::spawn(|| {
5         // to force the closure to take ownership of
6         // `primes` (and any other referenced variables), use the `move` keyword: `move`
7         // but it borrows `primes`, which is owned by the current function (may outlive
8         // borrowed value `primes`)
9         // function requires argument type to outlive `static`
10        let found_primes = (0..100)
11            .into_iter()
12            .filter(|num| {
13                primes.contains(num) // Access stack data
14            })
15            .collect::<Vec<usize>>();
16
17        println!("{:?}", found_primes);
18    });
19
20    let handle2 = std::thread::spawn(|| {
21        let found_primes = (0..100).rev()
22            .into_iter()
23            .filter(|num| {
24                primes.contains(num) // Access stack data
```

Scoped Threads

```
1 fn main() {
2     let primes = [2,3,5,7,9,11,13,15]; // Some data on the stack
3     let primes_ref = &primes;
4     `primes` does not live long enough (borrowed value does
not live long enough)
5     let primes_ref2 = &primes;
6     let handle = std::thread::spawn(move || {
7         argument requires that `primes` is borrowed for `static`
8         let found_primes = (0..100)
9             .into_iter()
10            .filter(|num| {
11                primes_ref.contains(num) // Access stack data
12            }).collect::<Vec<usize>>();
13        println!("{:?}", found_primes);
14    });
15
16    let handle2 = std::thread::spawn(move || {
17        let found_primes = (0..100).rev()
18            .into_iter()
19            .filter(|num| {
20                primes_ref2.contains(num) // Access stack data
21            }).collect::<Vec<usize>>();
22
23        println!("{:?}", found_primes);
```

Scoped Threads

```
1 use rayon;
2
3 fn main() {
4     let primes = [2,3,5,7,9,11,13,15]; // Some data on the stack
5
6     rayon::scope(|s| {
7         s.spawn(|_| {
8             let found_primes = (0..100)
9                 .into_iter()
10                 .filter(|num| {
11                     primes.contains(num) // Access stack data
12                 }).collect::<Vec<usize>>();
13
14             println!("{:?}", found_primes);
15         });
16
17         s.spawn(|_| {
18             let found_primes = (0..100).rev()
19                 .into_iter()
20                 .filter(|num| {
21                     primes.contains(num) // Access stack data
22                 }).collect::<Vec<usize>>();
23
24             println!("{:?}", found_primes);
25         });
26     });
```


Scoped Threads

```
19         .into_iter()
20         .filter(|num| {
21             primes.contains(num) // Access stack data
22         }).collect::<Vec<usize>>();
23
24         println!("{:?}", found_primes);
25     });
26 }
27 // Rayon automatically joins all threads
28 } // <- Dropping primes here
```

Bot L27 (Rust FlyC cargo God)

```
[~/talk/snippets]# cargo run --bin scoped-threads -> 0
    Finished dev [unoptimized + debuginfo] target(s) in 0.03s
    Running `target/debug/scoped-threads`
[15, 13, 11, 9, 7, 5, 3, 2]
[2, 3, 5, 7, 9, 11, 13, 15]
[~/talk/snippets]# -> 0
```

Scoped Threads

```
1 use rayon;
2 use rayon::prelude::*;
3
4 fn main() {
5     let primes = [2,3,5,7,9,11,13,15]; // Some data on the stack
6
7     rayon::scope(|s| {
8         s.spawn(|_| {
9             let found_primes = (0..100_usize)
10                .into_par_iter()
11                .filter(|num| {
12                    primes.contains(num) // Access stack data
13                }).collect::<Vec<usize>>();
14
15             println!("{:?}", found_primes);
16         });
17
18         s.spawn(|_| {
19             let found_primes = (0..100_usize)
20                .into_par_iter()
21                .rev()
22                .filter(|num| {
23                    primes.contains(num) // Access stack data
24                }).collect::<Vec<usize>>();
25
26             println!("{:?}", found_primes);
```

Scoped Threads

```
13         }).collect::<Vec<usize>>();
14
15         println!("{:?}", found_primes);
16     });
17
18     s.spawn(|_| {
19         let found_primes = (0..100_usize)
20             .into_par_iter()
21             .rev()
22             .filter(|num| {
23                 primes.contains(num) // Access stack data
24             }).collect::<Vec<usize>>();
25
26         println!("{:?}", found_primes);
```

```
-:--- scoped-threads.rs 40% L18 (Rust FlyC cargo God)
```

```
[~/talk/snippets]# cargo run --bin scoped-threads -> 0
```

```
    Finished dev [unoptimized + debuginfo] target(s) in 0.03s
```

```
    Running `target/debug/scoped-threads`
```

```
[2, 3, 5, 7, 9, 11, 13, 15]
```

```
[15, 13, 11, 9, 7, 5, 3, 2]
```

```
[~/talk/snippets]# -> 0
```

Hunter's Async type

- ▶ Why not use Futures?
 - ▶ Futures forget: Values can only be returned once
 - ▶ No asynchronous API to read directory contents and metadata
 - ▶ It seemed interesting and fun
- ▶ Advantages:
 - ▶ Easy to wrap any synchronous operation
 - ▶ Flexible ad-hoc mutation of value from anywhere, anytime
- ▶ Disadvantage: Not zero-cost

Async Type Definitions

```
1 // Final value is temporarily stored here
2 pub type AsyncValue<T> = Arc<Mutex<Result<T>>>>;
3
4 // The closure that computes the value we want
5 pub type AsyncValueFn<T> = FnOnce() -> Result<T> + Send + 'static;
6
7 // This closure runs when the value is ready
8 pub type OnReadyFn<T> = FnOnce(Result<&mut T, AError>)
9                             -> Result<()> + Send + 'static;
10
11 // This stores all the state (final values, closures, state, etc)
12 // When it's done running, the unnecessary Arcs cause a few bytes overhead :(
13 pub struct Async<T: Send + 'static>
14 {
15     // Value is moved here later, so further access doesn't require locking
16     pub value: AResult<T>,
17
18     async_value: AsyncValue<T>,
19     async_closure: Arc<Mutex<Option<Box<dyn AsyncValueFn<T>>>>>>,
20
21     // A Vec so that multiple closures can be registered
22     on_ready: Arc<Mutex<Vec<Box<dyn AsyncReadyFn<T>>>>>>,
23 }
```

Async Usage

```
1 fn read_dir_async() -> Result<Async<Files>, Error> {
2     let mut async_files = Async::new(|| {
3         // Can be slow!!
4         let files: Files = Files::read_dir("/foo")?;
5
6         Ok(files)
7     });
8
9     // Send event to main loop when ready, so it redraws screen
10    async_files.on_ready(|files| send_event_files_ready(&files.dir))?;
11
12    // Run in default thread pool
13    async_files.run_pooled(None)?;
14
15    // Return async dir entries
16    Ok(files)
17 }
18
19 fn draw_files(async_files: Async<Files>) -> Result<(), Error> {
20     // Optionally do this once to move async value out of Mutex
21     files.pull_async().ok();
22
23     let files: &Files = files.get()?;
24
25     for file in files.iter() {
26         println!("{}", file.name);
```

Async File Selection

```
1 // Run some program that prints out the path of a file to stdout
2 let path = run_external_selector()?;
3
4 // Create File structure from that path
5 let file = File::new_from_path(&path, None)?;
6
7 // Get the parent directory of that file
8 let dir = file.parent_as_file()?;
9
10 // Does some bookkeeping and asynchronously loads new directory
11 self.main_widget_goto(&dir)?;
12
13 // We don't want to wait for the directory to load here!
14
15
16 // This adds a new callback to be run when the directory is loaded
17 // Doesn't overwrite any previously registered callbacks
18 self.main_async_widget_mut()?
19     .widget
20     .on_ready(move |w, _| {
21         w?.select_file(&file);
22         Ok(())
23     })?;
24
25
26 // Execution goes on immediately
```

Other Optimizations

- ▶ 3-Phase directory loading
 1. Load file paths/names with `read_dir()`
 2. Load metadata for visible files
 3. Calculate number of files in visible directories
- ▶ Mostly Useless
 - ▶ Async bookmark saving
 - ▶ Async tag loading/saving
 - ▶ Async config loading