Need holistic explanation about Rust's cell and reference counted types

Asked 5 years, 3 months ago   Modified 4 years, 3 months ago   Viewed 9k times

91

There are several wrapper types in the Rust standard library:

- The cells in the `std::cell` module: `Cell` and `RefCell`
- The reference-counted wrappers, like `Rc` and `Arc`.
- The types in the `std::sync` module: `Mutex` or `AtomicBool` for example

As I understand it, these are wrappers which provide further possibilities than a simple reference. While I understand some basics, I cannot see the whole picture.

What do they do exactly? Do cells and reference-counted families provide orthogonal or similar features?

rust

Share Edit Follow

edited Jun 5, 2018 at 13:40
Shepmaster
351k 77 992 1244

asked Aug 14, 2017 at 12:23
Boiethios
34.2k 14 123 173

5 Please describe *what* you don't understand. The documentation you've linked to has had hundreds (or thousands) of people read it, and likely tens to hundreds of people contribute to it. What can we possibly say that would be any different from what is already said, especially if you don't tell us what you don't understand? – Shepmaster Aug 14, 2017 at 12:32

1 @Shepmaster The explanations in the documentation speak about *interior mutability*. This concept is unclear for me. Maybe my question has been unclear too. – Boiethios Aug 14, 2017 at 12:37

3 A quick internet search found a chapter in the book about interior mutability, a blog post, and a SO question. Perhaps now that you know what you are unclear about, you can perform some research and update your question. – Shepmaster Aug 14, 2017 at 12:48

Updated link to the latest version of the book: doc.rust-lang.org/book/ch15-05-interior-mutability.html – Tim Keating May 13, 2021 at 16:03

2 Answers

Sorted by: Highest score (default)

170

There are two essential concepts in Rust:

- Ownership,
- Mutability.

The various pointer types (`Box` , `Rc` , `Arc` ed with *Ownership*: they allow controlling whether there is a single or multiple owners for a single object.

Follow this question to receive notifications

On the other hand, the various cells (`Cel` `utex` , `RwLock` , `AtomicXXX`) are concerned with *Mutability*.

The founding rule of Rust's safety is **Aliasing XOR Mutability**. That is, an object can only be safely mutated if there is no outstanding reference to its interior.

This rule is generally enforced at compile time by the *borrow checker*:

- if you have a `&T`, you cannot also have a `&mut T` to the same object in scope,
- if you have a `&mut T`, you cannot also have any reference to the same object in scope.

However, sometimes, this is not flexible enough. Sometimes you DO need (or want) the ability to have multiple references to the same object and yet mutate it. Enter the **cells**.

The idea of `Cell` and `RefCell` is to permit mutability in the presence of aliasing *in a controlled manner*:

- `Cell` prevents the formation of reference to its interior, avoiding dangling references,
- `RefCell` shifts the enforcement of **Aliasing XOR Mutability** from compile time to runtime.

This functionality is sometimes described as providing **interior mutability**, that is where an object which otherwise looks immutable from the outside (`&T`) can actually be mutated.

When this mutability extends across multiple threads, you will instead use `Mutex`, `RwLock` or `AtomicXXX`; they provide the same functionality:

- `AtomicXXX` are just `Cell`: no reference to the interior, just moving in/out,
- `RwLock` is just `RefCell`: can obtain references to the interior through *guards*,
- `Mutex` is a simplified version of `RwLock` which does not distinguish between a read-only guard and write guard; so conceptually similar to a `RefCell` with only a `borrow_mut` method.

If you come from a C++ background:

- `Box` is `unique_ptr`,
- `Arc` is `shared_ptr`,
- `Rc` is a non thread-safe version of `shared_ptr`.

And the cells provide a similar functionality as `mutable`, except with additional guarantees to avoid aliasing issues; think of `Cell` as `std::atomic` and `RefCell` as a non thread-safe version of `std::shared_mutex` (which throws instead of blocking if the lock is taken).

Share Edit Follow

edited Aug 14, 2017 at 18:04

answered Aug 14, 2017 at 12:48
Matthieu M.
276k 48 434 702

2 Thank you *so* much, you answer exactly to all my perplexity about "complicated" types in Rust. The comparison with C++ is perfect, also. – Boiethios Aug 14, 2017 at 13:09

3 @Boiethios: Yes, that's exactly that. Each reference/pointer is "an alias" to the real data (a different name for the same "person" is what an alias is, after all), and by extension in software we talk about aliasing when there are multiple aliases at a given time. – Matthieu M. Aug 14, 2017 at 13:23

3 The first edition of the book had a chapter on this. It's one of the ones that was lost in the rewrite, which I think is unfortunate -- maybe it should be made into an appendix or something – trent Aug 14, 2017 at 15:47

7 If **Aliasing XOR Mutability** is the rule for safety, that implies that **(not Aliasing) AND (not Mutability)** is unsafe? But surely, if there is no aliasing AND no mutability then that is totally safe? I think you want something else than XOR there. – Lii Aug 14, 2017 at 16:43

13 Maybe the slogan should use NAND instead of XOR :P – Rufflewind Aug 14, 2017 at 20:25

Thanks to Matthieu's good answer, here is a diagram to help people to find the wrapper they need:

38

```
+-----------+
| Ownership |
+--+--------+                              +================+
   |                         +-Static----->| T              |(1)
   |                         |             +================+
   |                         |
   |                         |             +================+
   |          +-----------+  | Local    Val| Cell<T>        |(1)
   +-Unique-->| Borrowing +--+-Dynamic---->|----------------|
   |          +-----------+  |          Ref| RefCell<T>     |(1)
   |                         |             +================+
   |                         |
   |                         |             +================+
   |                         | Threaded    | AtomicT        |(2)
   |                         +-Dynamic---->|----------------|
   |                                       | Mutex<T>       |(1)
   |                                       | RwLock<T>      |(1)
   |                                       +================+
   |
   |
   |                                       +================+
   |                         +-No--------->| Rc<T>          |
   |                         |             +================+
   | Locally  +-----------+  |
   +-Shared-->| Mutable?  +--+             +================+
   |          +-----------+  |          Val| Rc<Cell<T>>    |
   |                         +-Yes-------->|----------------|
   |                                    Ref| Rc<RefCell<T>> |
   |                                       +================+
   |
   |
   |                                       +================+
   |                         +-No--------->| Arc<T>         |
   |                         |             +================+
   | Shared   +-----------+  |
   +-Between->| Mutable?  +--+             +================+
     Threads  +-----------+  |             | Arc<AtomicT>   |(2)
                             +-Yes-------->|----------------|
                                           | Arc<Mutex<T>>  |
                                           | Arc<RwLock<T>> |
                                           +================+
```

1. In those cases, `T` can be replaced with `Box<T>`
2. Use `AtomicT` when `T` is a `bool` or a number

To know if you should use `Mutex` or `RwLock`, see this related question.

Share Edit Follow

edited Jul 30, 2018 at 8:35
Peter Hall
47.6k 11 114 184

answered Jun 5, 2018 at 9:14
Boiethios
34.2k 14 123 173