Why lifetime check differs between Vec::as_slice and array's coercence?

Asked 1 year, 3 months ago

Active 1 year, 3 months ago

Viewed 59 times

2

Minimum code:

```
fn foo() {
    let vector = vec![1u8, 2u8];
    let a = &vector.as_slice()[0];
    drop(vector);
    let b = a;
}

fn bar() {
    let array = [1u8, 2u8];
    let a = &array[0];
    drop(array);
    let b = a;
}
```

When compiling the above codes, `foo` can't compile while `bar` can.

In function `foo`, `a` borrows the `vector`, so after I drop `vector`, I can't access `a` any more.

However, I think the situation is same for `bar`, but `bar` can successfully pass the compilation. I don't know why.

rust lifetime

asked Sep 24 '20 at 6:53

Evian

831 • 4 • 15

2 Answers

3

This is not specific to vectors and arrays. As soon as Rust can prove that a value is a constant known at compile time, it considers it has having a static lifetime (and drop in this case doesn't have any effect since the value will last forever). See this similar question.

I extended your example with a simple constant put in a variable (*binding*) and a boxed version; this shows the exact same distinction as you noticed between vector and array (trivial constant vs dynamically allocated value).

```
fn test_vec() {
    let src = vec![10u8, 20u8];
    let a = &src[0];
    // drop(src); // rejected by compiler
    let b = a;
    println!("{:?}", b);
}

fn test_array() {
    let src = [10u8, 20u8];
    let a = &src[0];
    drop(src);
    let b = a;
    println!("{:?}", b);
}

fn test_box() {
    let src = Box::new(10u8);
    let a = &src;
    // drop(src); // rejected by compiler
    let b = a;
    println!("{:?}", b);
}

fn test_var() {
    let src = 10u8;
    let a = &src;
    drop(src);
    let b = a;
    println!("{:?}", b);
}

fn main() {
    test_vec();
    test_array();
```

Share
Improve this answer
Follow

answered Sep 24 '20 at 7:20

prog-fh
8,988 ● 1 ● 5 ● 24

Add a comment

2

An immutable array literal has static lifetime. Dropping the local variable `array` doesn't really do anything in this case – the data is mapped into memory when the binary is loaded, and it will stay there for the runtime of the program, so the reference is still vaild. You just can't access the memory via the `array` variable anymore.

The vector case is very different. Creating a new vector dynamically allocates memory on the heap at runtime, and `drop()` will release that memory again, so the reference becomes invalid.

Share
Improve this answer
Follow

answered Sep 24 '20 at 7:20

Sven Marnach
511k ● 113 ● 891 ● 798

Add a comment

Your Answer

Post Your Answer

Not the answer you're looking for? Browse other questions tagged rust lifetime or ask your own question.

The Overflow Blog

- Sequencing your DNA with a USB dongle and open source code
- Don't push that button: Exploring the software that flies SpaceX rockets and...

Featured on Meta

- Providing a JavaScript API for userscripts
- Congratulations to the 59 sites that just left Beta

Linked

Why can I return a reference to a local literal but not a variable?

Related

What are the differences between Rust's `String` and `str`?

Why can't I store a value and a reference to that value in the same struct?

What is the difference between iter and into_iter?

Why does the closure take ownership of the vector here?

Problems with rust lifetime specifier

Rust lifetime in Vec<&T> : convoluted syntax

mutable borrow starts here in previous iteration of loop

Hot Network Questions

- Naruto fighting game with Hulk and Homer Simpson?
- Is it acceptable to omit "about" in this sentence? "I love everything (about) math."
- Which direction/constellation is the James Webb is headed towards
- How to force Mathematica to do infinite-precision calculations?
- Does saying "Keep it up" put me in an authoritative position?

more hot questions

Question feed