Rust | The Difference Between .clone() and .to_owned()

July 25, 2022 by Andres Reales

Chances are you recently came across the definition of the ToOwned trait to realize it is *a generalization of the Clone* trait? If so, what is the difference between the two traits? more specifically, what is the difference between `.clone()` and `.to_owned()` if they work the same?

> *A generalization of Clone to borrowed data.*
>
> *https://doc.rust-lang.org/std/borrow/trait.ToOwned.html#tymethod.to_owned*

The difference between `.clone()` and `.to_owned()` occurs when applying either of the two methods on slices such as string slice `&str` or arrays with undefined capacity like `&[i8]` or `&[u32]`, in their borrowed state (`&T`):

- **`.clone()` generates a duplicate of types such as`&str` or `&[u8]` with the same type in its borrowed state (`&T`). This means using `.clone()` on `&str` and `&[u8]` will generate `&str` and `[u8]` respectively.**

```
let str = "a"; // type &str
let cloned_str = str.clone(); // type &str

let array:&[u8] = &[1, 2, 3];
 // type &[u8]
let cloned_array = array.clone(); type &[u8]
```

- **`.to_owned()` generates a duplicate of types such as `&str` or `&[u8]` with types that have ownership. This means using `.to_owned()` on `&str` and `&[u8]` will generate a `String` and a `Vec<u8>` respectively.**

```
let array:&[u8] = &[1, 2, 3];
let cloned_array = array.to_owned();
```

You probably had to read twice the previous answer to understand the difference between `.clone()` and `.to_owned()`. In fact, it might still not be clear the difference.

Don’t worry.

This article will explain with examples the difference between `.clone()` and `.to_owned()` methods. You will get an overall understanding of how each method works. Then, you will understand why `.to_owned()` generates a `String` from a `&str` .

Table of Contents

Understanding more about the `.clone()` method

As the name suggests, the `.clone()` method generates a duplicate of an object similar to the `Copy` trait. **The difference between `Copy` and `Clone` is that `Copy` duplicates bits stored in the stack, while `Clone` might involve copying heap data**, which could or not result in a more expensive operation.

Accessible with the `Clone` trait

The `.clone()` method is available to all types deriving the `Clone` trait. By default, all the primitive types (`str`, `i8`,`u8`, `char`,`array`, `bool`, etc), have access to the `.clone()` method.

```
    let cloned_borrowed_number = borrowed_number.clone();

    let array: [&str; 3] = ["a", "b", "c"];
    let cloned_array = array.clone();

    let borrowed_array: &[&str; 3] = &["a", "b", "c"];
    let cloned_borrowed_array = borrowed_array.clone();

    let string: String = String::from("Hello, world!");
    let cloned_string = string.to_owned();
```

If you are defining a struct and want to have access the `.clone()` method, you must make sure:

1. Every field in the struct is clonable
2. To Add the `Clone` derive attribute

To make this clear, if you create a `MyObject` struct like the following:

```
struct MyObject {
    first_name: String,
    last_name: String,
    age: u8,
}

impl MyObject {
    pub const fn new() -> MyObject {
        MyObject {
            first_name: String::new(),
            last_name: String::new(),
            age: 0,
        }
    }
}
```

and later in your code you decide to create a variable storing an `MyObject` data type,

```
let my_object = MyObject::new();
```

you won't have access to the `.clone()` method. Hence, `my_object` variable won't have `.clone()` method and it will throw the following error if you attempt to use it.

```
// this will throw the following error:
// error[E0599]: no method named `clone` found for struct
let cloned_my_object = my_object.clone();
```

To solve this issue, adding the `Clone` derive in the *struct* `MyObject`

```
#[derive(Clone)]
struct MyObjectOne {
    first_name: String,
    last_name: String,
    age: u8,
}
```

will give access to the .clone() method:

```
let my_object = MyObject::new();
let cloned_my_object = my_object.clone(); // this will wo
```

Generating a duplicate with the same type

The `.clone()` generates a duplicate of an object `T` with the same type `T`, meaning if

- a is `u8`, the duplicate will be a `u8`
- a is `String`, the duplicate will be a `String`
- a is `&[&str]`, the duplicate will be a `&[&str]`
- a is `MyObject`, the duplicate will be a `MyObject`

```
let number:u8 = 10; // type u8
let cloned_number = number.clone(); // type u8

let array: &[&str] = &["a", "b", "c"]; // type &[&str
let cloned_array = array.clone(); // type &[&str]

let string: String = String::from("Hello, world!"); /
let cloned_string = string.to_owned(); // type String

let my_object = MyObject::new(); // type MyObject
let cloned_my_object = my_object.clone(); // type MyO
```

It is possible to duplicate values with known size at compile time in their borrowed state `&T` to generate a duplicate in the owned state `T` using the `.clone()` method.

```
let borrowed_number: &u8 = &10; // type &u8
let cloned_borrowed_number = borrowed_number.clone(); /

let borrowed_array: &[&str; 3] = &["a", "b", "c"]; // t
let cloned_borrowed_array = borrowed_array.clone(); //

let string: &String = &String::from("Hello, world!"); /
let cloned_string = string.clone(); // type String
```

Generates a duplicate of `&T` as `T` for scalar types and tuples

Rust has four primary scalar types:

- Integers
- Floating-point numbers
- Booleans
- Characters

They represent a single value. This means all integers (`12`), floating-point numbers (`3.4`), booleans (`true`,`false`), and characters (`'a'`, `'z'`) have the same value no matter how many times you use them. It is because of this, that it makes it possible to generate an owned duplicate from a borrowing state. In other words, going from `&T` to `T`.

```
let integer: &u8 = &1; // type is &u8
let cloned_integer = integer.clone(); // type is u8

let floating_number = &2.3; // type is &f64
let cloned_floating_number = floating_number.clone(); //

let boolean = &true; // type is &boolean
let cloned_boolean = boolean.clone(); // type is boolean

let character = &'a'; // type is &char
let cloned_character = character.clone(); // type is char
```

Generates a duplicate of `&[T]` as `[T]` for arrays with a defined length

The `.clone()` method can duplicate of array when the original array has ownership, meaning going from `[T]` to `[T]`:

```
let owned_array: [i32; 3] = [1, 2, 3];
let cloned_owned_array = owned_array.clone(); // type is
```

However, `.clone()` can also go from `&[T]` to `[T]` as long as the array capacity is defined.

```
let borrowed_array: &[i32; 3] = &[1, 2, 3];
let cloned_borrowed_array = borrowed_array.clone(); // typ
```

The array capacity will also be implicitly defined even when you don't assign a type definition (`&[T]`). In the following example, the duplicate generated is `[T]` from `[T]`.

```
&[1, 2, 3]; // it will implicitly define the type as &[i32;
rray = borrowed_array.clone(); // type is [i32; 3]
```

However, if you define the type is of variable is an array without explicitly defining the capacity, `.clone()` will still generate a duplicate. However, this new duplicate will be still in a borrowing state. Hence, it will go from `&[T]` to `&[T]`.

The `.clone()` not go from `&[T]` to `[T]` on arrays without a specified capacity

If you define the type of variable is an array without explicitly defining its capacity, `.clone()` will still generate a duplicate. However, this new duplicate will be still in a borrowing state. Hence, it will go from `&[T]` to `&[T]`.

```
let numbers: &[i32] = &[1, 2, 3];
let cloned_numbers = numbers.clone(); // type is [&i32]

let strs: &[&str] = &["asfa", "saf", "asfas"];
let cloned_strs = strs.clone(); // type is &[&str]
```

Understanding more about the `.to_owned()` method

As the Rust documentation states, the `.to_owned()` method is a generalization of the `Clone` trait to borrowed data. This means, `.to_owned()` **generates a duplicate value.** Hence, if you apply `.to_owned()` on `T` , the duplicate will be `T` .

```
let number:u8 = 10; // type u8
let to_owned_number = number.to_owned(); // type u8

let string: String = String::from("Hello, world!"); // ty
let to_owned_string = string.to_owned(); // type String

let my_object = MyObject::new(); // type MyObject
let to_owned_my_object = my_object.to_owned(); // type My
```

One key aspect of `.to_owned()` is that this method is capable of constructing owned data from any borrow of a given type. This means, if you apply `.to_owned()` on any `&T`, the duplicate will be `T`.

```
let number: &u8 = &10; // type u8
let cloned_number = number.to_owned(); // type u8

let string: &String = &String::from("Hello, world!"); //
let cloned_string = string.to_owned(); // type String

let my_object: &MyObject = &MyObject::new(); // type &My(
let cloned_my_object = my_object.to_owned(); // type MyOb
```

The `.to_owned()` method generates a `String` from `&str`

One question that comes for many Rust developers is: *How come the* `.clone()` *method generates a similar data type when cloning a string slice reference*

(`&str`) and the `.to_owned()` method generates a `String`?

To answer this question, you must understand the concepts of ownership and borrowing. The string slice reference `&str` is in its borrowed state. That means, that whichever variable stores the string slice reference does not own the value.

In the previous section you learned the `.clone()` method can still convert from `&T` to `T` where `T` could be a primitive type such as a boolean or a number or a custom struct such as `MyObject`. What is the difference with a `str`?

The string slice is not a primitive type but a sequence of characters. In other words, an array of characters `[T]`. A characteristic of the string slice is that doesn't have ownership. The `.clone()` method attempts to make it possible to go from borrowed to owned, but that doesn't mean always happens. As you know, that's the case of the string slice reference `&str` .

Hence, the `.clone()` method does generate a duplicate of a string slice reference without ownership. That's where the `.to_owned()` method differs from `.clone()`.

The `.to_owned()` "generalizes" the `Clone` trait to construct data from borrow of a give type. This implies that the core functionality of `to_owned()` is to make sure the duplicate value will always have ownership, even if this means having to allocate the values in a data type different from the original.

By generating a `String` from a string slice reference `&str` , `to_owned()` meats the criteria of providing ownership. As you should remember, a `String` value can have ownership. If you look at the definition of the `String`, you will find it is a struct based on a vector.

```
pub struct String {
    vec: Vec<u8>
}
```

The `.to_owned()` method generates a `Vector` from a reference array with undefined capacity

There is another interesting case of using the `.to_owned()` method results in a duplicate of a different type. This happens when applying `.to_owned()` to reference arrays with undefined capacity, such as,

- `&[i8]`
- `&[i16]`
- `&[i32]`
- `&[i64]`
- `&[i128]`
- `&[u8]`
- `&[u16]`
- `&[u32]`
- `&[u64]`
- `&[u128]`

generates a vector

- `Vec<i8>`
- `Vec<i16>`
- `Vec<i32>`
- `Vec<i64>`
- `Vec<i128>`
- `Vec<u8>`
- `Vec<u16>`
- `Vec<u32>`
- `Vec<u64>`
- `Vec<u128>`

respectively.

```
let array_i16: &[i16] = &[1, 2, 3];
let cloned_array_i16 = array_i16.to_owned(); // type is \

let array_i32: &[i32] = &[1, 2, 3];
let cloned_array_i32 = array_i32.to_owned(); // type is \

let array_i64: &[i64] = &[1, 2, 3];
let cloned_array_i64 = array_i64.to_owned(); // type is \

let array_i128: &[i128] = &[1, 2, 3];
let cloned_array_i128 = array_i128.to_owned(); // type is

let array_u8: &[u8] = &[1, 2, 3];
let cloned_array_u8 = array_u8.to_owned(); // type is Vec

let array_u16: &[u16] = &[1, 2, 3];
let cloned_array_u16 = array_u16.to_owned(); // type is \

let array_u32: &[u32] = &[1, 2, 3];
let cloned_array_u32 = array_u32.to_owned(); // type is \

let array_u64: &[u64] = &[1, 2, 3];
let cloned_array_u64 = array_u64.to_owned(); // type is \

let array_u128: &[u128] = &[1, 2, 3];
let cloned_array_u128 = array_u128.to_owned(); // type is
```

Conclusion

In this article, you learned the difference between `.clone()` and `.to_owned()` despite them being similar in their functionality. The key difference to remember is that while `.clone()` attempts to go from borrowed to owned state, that is not always possible as `Clone` works only from `&T` to `T`.

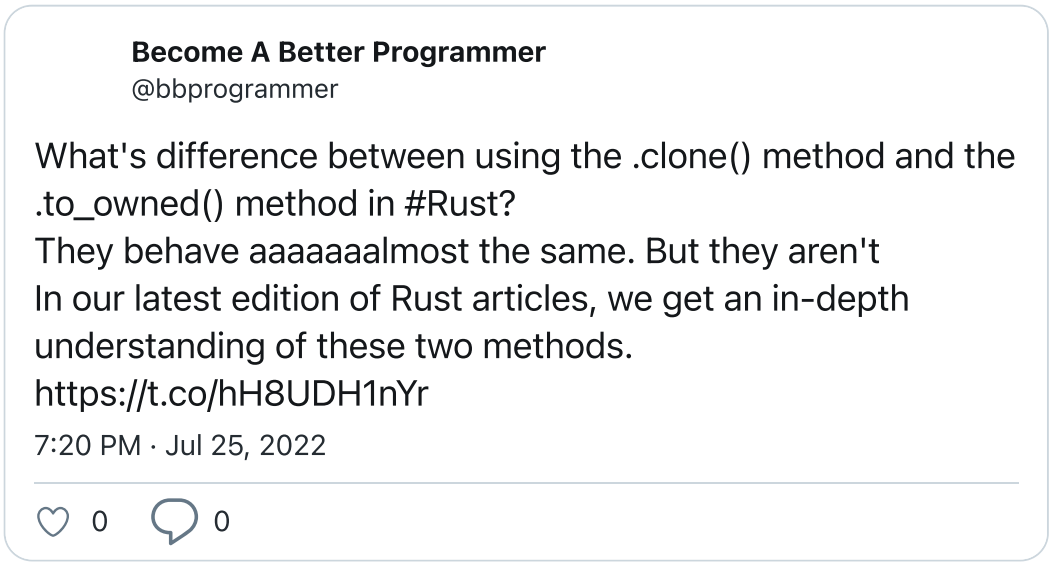
On the other hand, `to_owned()` makes sure the duplicate value has ownership even if that means having to use a different data type such as a `String` or a

`Vec<u8>` when generating a duplicate for a `&str` or a `&[u8]` .

Was this a tough concept, right?

Indeed it is, and it takes a decent amount of time to understand concepts in Rust. Hopefully, this article helped you have more clarity on these two methods (`.clone()` and `to_owned()`).

Let me know what you thought of this article by replying us in Twitter of Become a Better Programmer or to my personal account.



Rust
< How to Build a File Upload Rest API in Node.js and Express?
> How to Check for an Empty Object in TypeScript/JavaScript

About Us

Andrés Reales is the founder of Become a Better Programmer blogs and tutorials and Senior Full-Stack Software Engineer. With the purpose of helping others succeed in the always-evolving world of programming, Andrés gives back to the community by sharing his experiences and teaching his programming skillset gained over his years as a professional programmer.