PV281: Programování v Rustu

Obsah

Generika

Umožňuje obecnou definici pro různé typy položek u struktur, výčtů nebo metod.

Generika u struktur

```
struct Point<T> {
    x: T,
    y: T,
}

fn main() {
    let integer = Point { x: 5, y: 10 };
    let float = Point { x: 1.0, y: 4.0 };
}
```

Generika u struktur (více typů)

```
struct Point<T, U> {
    x: T,
    y: U,
}

fn main() {
    let both_integer = Point { x: 5, y: 10 };
    let both_float = Point { x: 1.0, y: 4.0 };
    let integer_and_float = Point { x: 5, y: 4.0 };
}
```

Generika u výčtu

```
enum Option<T> {
    Some(T),
    None,
}

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

Generika u metod

```
struct Point<T, U> {
    x: T,
    y: U,
}

impl<T, U> Point<T, U> {
    fn mixup<V, W>(self, other: Point<V, W>) -> Point<T, W> {
        Point {
            x: self.x,
            y: other.y,
        }
    }
}

fn main() {
    let p1 = Point { x: 5, y: 10.4 };
    let p2 = Point { x: "Hello", y: 'c' };

    let p3 = p1.mixup(p2);

    println!("p3.x = {}, p3.y = {}", p3.x, p3.y);
}
```

Generika u metod

```
fn largest<T: PartialOrd + Copy>(list: &[T]) -> T {
    let mut largest = list[0];

    for &item in list {
        if item > largest {
            largest = item;
        }
    }

    largest
}
```

Generika u metod

```
fn some_function<T, U>(t: &T, u: &U) -> i32
    where T: Display + Clone,
          U: Clone + Debug
{
    // ...
}
```

Smart Pointery

Pro připomenutí

1. Máme stack, který se uklízí automaticky na konci funkce
2. Máme heap, kde máme velké objekty, objekty s neznámou velikostí za překladu a data, která nám musí přežít dlouho
3. Heap se uklízí. V jiných jazycích to dělá garbage collector; v Rustu instrukce pro čištění dodá překladač

Typy pointerů

Pointer - základní odkaz na místo v paměti. Všechna omezení jsou definovaná borrow checkerem.

Smart Pointer - chování odpovídá základnímu pointeru, ale dle varianty má další metadata a schopnosti.

Smart Pointer

Smart Pointer implementuje trait Deref a Drop. Z toho je jasné, že je to struktura. Je vlastníkem dat, na která odkazuje.

Pozn.: String je smart pointer. Datové struktury jsou smart pointery.

Deref coercion

= automatická dereference parametrů funkcí/metod

```
use std::ops::Deref;

impl<T> Deref for MyBox<T> {
    type Target = T;

    fn deref(&self) -> &Self::Target {
        &self.0
    }
}
```

Drop

```
struct CustomSmartPointer {
    data: String,
}

impl Drop for CustomSmartPointer {
    fn drop(&mut self) {
        println!("Dropping CustomSmartPointer with data `{}`!", self.data);
    }
}

fn main() {
    let c = CustomSmartPointer {
        data: String::from("my stuff"),
    };
    let d = CustomSmartPointer {
        data: String::from("other stuff"),
    };
    println!("CustomSmartPointers created.");
}
```

Varianty Smart Pointerů

1. Box<T>
2. Cell<T>
3. RefCell<T>
4. Rc<T>
5. Ref<T>
6. RefMut<T>

Box<T>

Je nejjednoduššího smart pointerem. Ukládá data na haldě (nikoli na zásobníku) i ve chvíli, kdy je jejich velikost známá za překladu. Samotný pointer může být na zásobníku, ale data ne.

Hodí se nám ve chvíli, kdy potřebujeme udělat rekurzivní struktury.

Box<T>

```
enum List {
   Cons(i32, Box<List>),
   Nil,
}

use List::{Cons, Nil};

fn main() {
let list = Cons(1, Box::new(Cons(2, Box::new(Cons(3, Box::new(Nil))))));
}
```

Cell<T>

Umožňuje mutaci uvnitř imutabilní struktury. Má funkce get a set. Get vrací kopii dat.

Je vhodná pro menší data - z důvodů vracení kopie.

Je pro jednovláknové použití. Vícevláknové alternativy budou příště.

Cell<T>

```rust
use std::cell::Cell;

struct SomeStruct {
    regular_field: u8,
    special_field: Cell<u8>,
}

fn main() {
    let my_struct = SomeStruct {
        regular_field: 0,
        special_field: Cell::new(1),
    };

    let new_value = 100;

    // ERROR: `my_struct` is immutable
    // my_struct.regular_field = new_value;

    // WORKS: although `my_struct` is immutable, `special_field` is a `Cell`,
    // which can always be mutated
    my_struct.special_field.set(new_value);
    assert_eq!(my_struct.special_field.get(), new_value);
}
```

RefCell<T>

Dává sdílený přístup k T, ale narozdíl od Cell je kontrolovaná za běhu. Má metody, která vrací mutovatelnou nebo nemutovatelnou referenci. Musíme si sami napsat kontrolu, jestli se povedlo získat referenci.

Pokud neexistuje žádná reference můžeme dostat mutovatelnou i nemutovatelnou reference.
Pokud existuje mutovatelná reference, tak nejde získat jiná.
Pokud existuje nemutovatelná reference, tak můžeme získat další nemutovatelnou referenci.

RefCell<T>

```
fn main() {
    use std::cell::RefCell;
    let container = RefCell::new(11);
    {
        let _c = container.borrow();
        // You may borrow as immutable as many times as you want,...
        assert!(container.try_borrow().is_ok());
        // ...but cannot borrow as mutable because
        // it is already borrowed as immutable.
        assert!(container.try_borrow_mut().is_err());
    }
    // After the first borrow as mutable...
    let _c = container.borrow_mut();
    // ...you cannot borrow in any way.
    assert!(container.try_borrow().is_err());
    assert!(container.try_borrow_mut().is_err());
}
```

Rc<T>

Pokud potřebujeme více vlastníku, tak můžeme využít reference counting. Pokud existuje jakýkoliv odkaz, tak data nejsou uvolněna.

T v Rc je imutabilní. Pokud chceme, aby se obsah dal měnit, tak musíme použít kombinace s Cell nebo RefCell.

Pokud tvoříme cyklické vazby (například odkazování mezi potomkem a rodičem stromu) můžeme využít alternativu Weak, které není vlastníkem dat a ta mohou být odstraněna. Jinak bychom mohli způsobit memory leak.

Rc<T>

```
enum List {
    Cons(i32, Rc<List>),
    Nil,
}

use crate::List::{Cons, Nil};
use std::rc::Rc;

fn main() {
    let a = Rc::new(Cons(5, Rc::new(Cons(10, Rc::new(Nil)))));
    println!("count after creating a = {}", Rc::strong_count(&a));
    let b = Cons(3, Rc::clone(&a));
    println!("count after creating b = {}", Rc::strong_count(&a));
    {
        let c = Cons(4, Rc::clone(&a));
        println!("count after creating c = {}", Rc::strong_count(&a));
    }
    println!("count after c goes out of scope = {}", Rc::strong_count(&a));
}
```

Rc<T> s RefCell<U>

```
#[derive(Debug)]
enum List {
    Cons(Rc<RefCell<i32>>, Rc<List>),
    Nil,
}

use crate::List::{Cons, Nil};
use std::cell::RefCell;
use std::rc::Rc;

fn main() {
    let value = Rc::new(RefCell::new(5));

    let a = Rc::new(Cons(Rc::clone(&value), Rc::new(Nil)));

    let b = Cons(Rc::new(RefCell::new(3)), Rc::clone(&a));
    let c = Cons(Rc::new(RefCell::new(4)), Rc::clone(&a));

    *value.borrow_mut() += 10; // borrow_mut vrací RefMut, proto je tu třeba *
                               // tímto jde získat jednu mutující referenci a n nemutujících

    println!("a after = {:?}", a);
    println!("b after = {:?}", b);
    println!("c after = {:?}", c);
}
```

Ref<T>

```
use std::cell::{RefCell, Ref};

let c = RefCell::new((5, 'b'));
let b1: Ref<(u32, char)> = c.borrow();
let b2: Ref<u32> = Ref::map(b1, |t| &t.0);
assert_eq!(*b2, 5)
```

RefMut<T>

```
use std::cell::{RefCell, RefMut};

let c = RefCell::new((5, 'b'));
{
    let b1: RefMut<(u32, char)> = c.borrow_mut();
    let mut b2: RefMut<u32> = RefMut::map(b1, |t| &mut t.0);
    assert_eq!(*b2, 5);
    *b2 = 42;
}
assert_eq!(*c.borrow(), (42, 'b'));
```

Modularita

Demostrační struktura programu

```
crate
 └── front_of_house
     ├── hosting
     │   ├── add_to_waitlist
     │   └── seat_at_table
     └── serving
         ├── take_order
         ├── serve_order
         └── take_payment
```

Možná souborová struktura

```
.
└── front_of_house
    ├── mod.rs
    ├── hosting.rs
    └── serving
        └── mod.rs
```

Klíčové slovo mod

```
mod front_of_house {
    mod hosting {
        fn add_to_waitlist() {}

        fn seat_at_table() {}
    }

    mod serving {
        fn take_order() {}

        fn serve_order() {}

        fn take_payment() {}
    }
}
```

Odkaz do jiného modulu

```
mod front_of_house {
    pub mod hosting {
        pub fn add_to_waitlist() {}
    }
}

pub fn eat_at_restaurant() {
    // Absolute path
    crate::front_of_house::hosting::add_to_waitlist();

    // Relative path
    front_of_house::hosting::add_to_waitlist();
}
```

Veřejné struktury

```
mod back_of_house {
    pub struct Breakfast {
        pub toast: String,
        seasonal_fruit: String,
    }

    impl Breakfast {
        pub fn summer(toast: &str) -> Breakfast {
            Breakfast {
                toast: String::from(toast),
                seasonal_fruit: String::from("peaches"),
            }
        }
    }
}

pub fn eat_at_restaurant() {
    // Order a breakfast in the summer with Rye toast
    let mut meal = back_of_house::Breakfast::summer("Rye");
    // Change our mind about what bread we'd like
    meal.toast = String::from("Wheat");
    println!("I'd like {} toast please", meal.toast);

    // The next line won't compile if we uncomment it; we're not allowed
    // to see or modify the seasonal fruit that comes with the meal
    // meal.seasonal_fruit = String::from("blueberries");
```

Veřejné enumy

```
mod back_of_house {
    pub enum Appetizer {
        Soup,
        Salad,
    }
}

pub fn eat_at_restaurant() {
    let order1 = back_of_house::Appetizer::Soup;
    let order2 = back_of_house::Appetizer::Salad;
}
```

Klíčové slovo use

Use s absolutní cestou

```
mod front_of_house {
    pub mod hosting {
        pub fn add_to_waitlist() {}
    }
}

use crate::front_of_house::hosting;

pub fn eat_at_restaurant() {
    hosting::add_to_waitlist();
    hosting::add_to_waitlist();
    hosting::add_to_waitlist();
}
```

Use s relativní cestou

```
mod front_of_house {
    pub mod hosting {
        pub fn add_to_waitlist() {}
    }
}

use self::front_of_house::hosting; // <- tady je rozdil

pub fn eat_at_restaurant() {
    hosting::add_to_waitlist();
    hosting::add_to_waitlist();
    hosting::add_to_waitlist();
}
```

Use pro funkci

```
mod front_of_house {
    pub mod hosting {
        pub fn add_to_waitlist() {}
    }
}

use crate::front_of_house::hosting::add_to_waitlist;

pub fn eat_at_restaurant() {
    add_to_waitlist();
    add_to_waitlist();
    add_to_waitlist();
}
```

Use pro více modulů/členů

```
use std::io::{self, Write};
```

Glob operátor

```
use std::collections::*;
```

super

```
fn function() {
    println!("called `function()`");
}

mod cool {
    pub fn function() {
        println!("called `cool::function()`");
    }
}

mod my {
    fn function() {
        println!("called `my::function()`");
    }

    mod cool {
        pub fn function() {
            println!("called `my::cool::function()`");
        }
    }

    //
```

super

```
mod my {
    // ...

    pub fn indirect_call() {
        // Let's access all the functions named `function` from this scope!
        print!("called `my::indirect_call()`, that\n> ");

        // The `self` keyword refers to the current module scope - in this case `my`.
        // Calling `self::function()` and calling `function()` directly both give
        // the same result, because they refer to the same function.
        self::function();
        function();

        // We can also use `self` to access another module inside `my`:
        self::cool::function();

        // The `super` keyword refers to the parent scope (outside the `my` module).
        super::function();

        // This will bind to the `cool::function` in the *crate* scope.
        // In this case the crate scope is the outermost scope.
        {
            use crate::cool::function as root_function;
            root_function();
        }
    }
}

fn main() {
    my::indirect_call();
```

Externí závislost

Závislosti mohou být checknuté za překladu a přilinkované za běhu. V Rustu 2015 bylo nutné použít extern crate. Dnes to už potřeba není.

```
extern crate pcre;

extern crate std; // equivalent to: extern crate std as std;

extern crate std as ruststd; // linking to 'std' under another name

extern crate hello_world; // if hyphen in package name then it is replaced with an underscore
                          // crate name cannot have hyphen

extern crate foo as _; // when only linked and not referenced
```

Workspace

Pokud tvoříme projekt, který má více samostatně použitelných knihoven, ale chceme se na ně odkazovat bez nutnosti publikování, tak můžeme použít workspace.

```
├── Cargo.lock
├── Cargo.toml
├── add-one
│   ├── Cargo.toml
│   └── src
│       └── lib.rs
├── adder
│   ├── Cargo.toml
│   └── src
│       └── main.rs
└── target
```

Cargo.toml workspacu

```
[workspace]

members = [
    "adder",
    "add-one",
]
```

Nová binárka do workspacu

```
cargo new adder
```

Nová libka do workspacu

```
cargo new add-one --lib
```

Závislost na libce v binárce

```
[dependencies]

add-one = { path = "../add-one" }
```

Následné spuštění

```
cargo run -p adder
```

Připomenutí konvencí pro binárky

```
foo
├── Cargo.toml
└── src
    ├── main.rs
    └── bin
        └── my_other_bin.rs
```


Testy

Testy

Jednotkové testy jsou v souboru s implementací. Integrační testy jsou ve složce tests.

Spouštíme je přes

```
cargo test
```

Panika v testu

Zfailuje test.

```
#[cfg(test)]
mod tests {
    #[test]
    fn exploration() {
        assert_eq!(2 + 2, 4);
    }

    #[test]
    fn another() {
        panic!("Make this test fail");
    }
}
```

Makro assert

```
#[derive(Debug)]
struct Rectangle {
    width: u32,
    height: u32,
}

impl Rectangle {
    fn can_hold(&self, other: &Rectangle) -> bool {
        self.width > other.width && self.height > other.height
    }
}

#[cfg(test)]
mod tests {
    use super::*;

    #[test]
    fn larger_can_hold_smaller() {
        let larger = Rectangle {
            width: 8,
            height: 7,
        };
        let smaller = Rectangle {
            width: 5,
            height: 1,
        };

        assert!(larger.can_hold(&smaller));
    }
```

assert_eq!

```
pub fn add_two(a: i32) -> i32 {
    a + 2
}

#[cfg(test)]
mod tests {
    use super::*;

    #[test]
    fn it_adds_two() {
        assert_eq!(4, add_two(2)); // máme i assert_ne
    }
}
```

vlastní zprávy při failu

```
pub fn greeting(name: &str) -> String {
    String::from("Hello!")
}

#[cfg(test)]
mod tests {
    use super::*;

    #[test]
    fn greeting_contains_name() {
        let result = greeting("Carol");
        assert!(
            result.contains("Carol"),
            "Greeting did not contain name, value was `{}`",
            result
        );
    }
}
```

testování paniky

```
pub struct Guess {
    value: i32,
}

impl Guess {
    pub fn new(value: i32) -> Guess {
        if value < 1 || value > 100 {
            panic!("Guess value must be between 1 and 100, got {}.", value);
        }

        Guess { value }
    }
}

#[cfg(test)]
mod tests {
    use super::*;

    #[test]
    #[should_panic]
    fn greater_than_100() {
        Guess::new(200);
    }
}
```

result jako výsledek testu

```
#[cfg(test)]
mod tests {
    #[test]
    fn it_works() -> Result<(), String> {
        if 2 + 2 == 4 {
            Ok(())
        } else {
            Err(String::from("two plus two does not equal four"))
        }
    }
}
```

Dokumentace

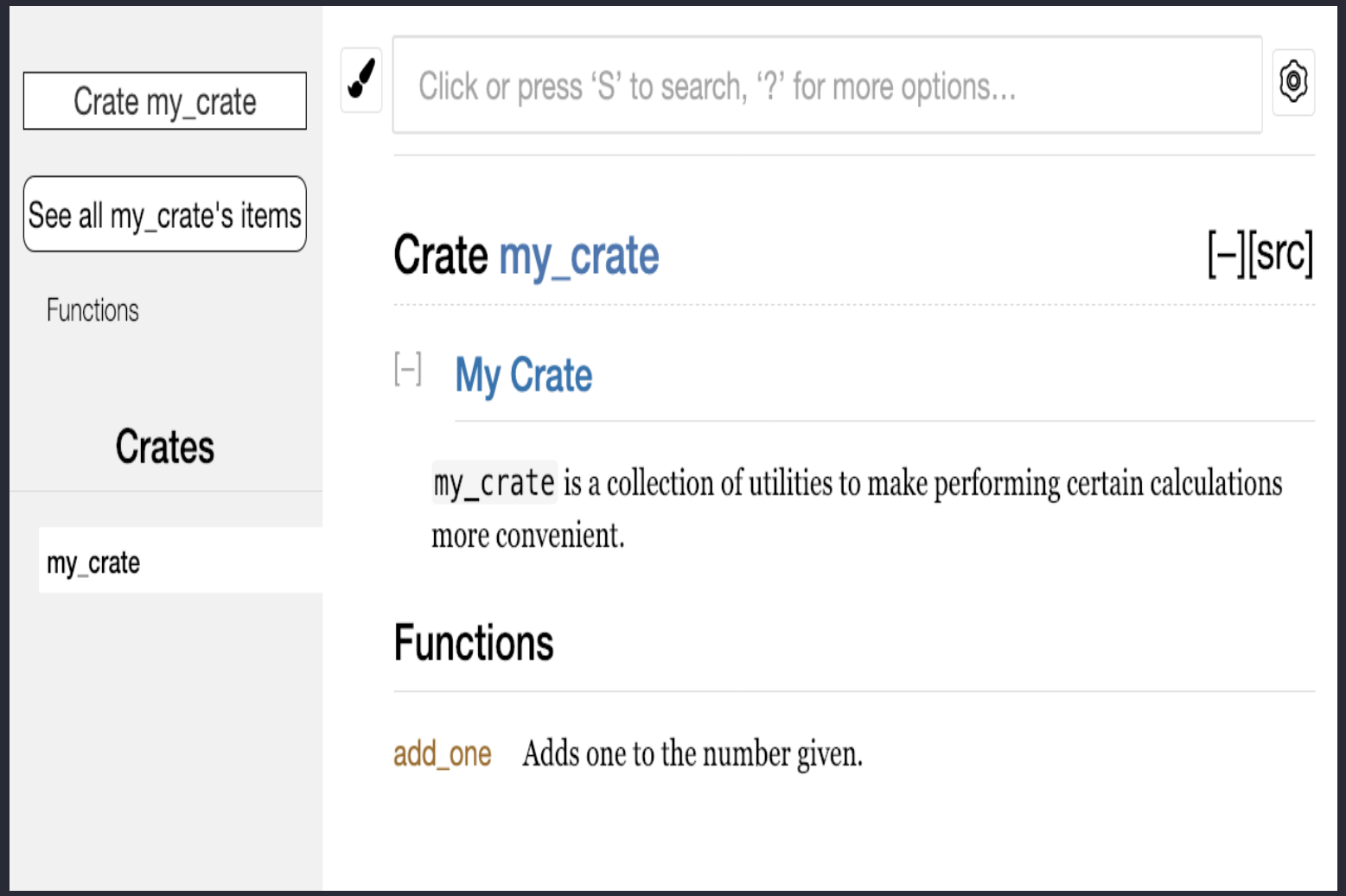
Dokumentační komentáře

```
/// Adds one to the number given.
///
/// # Examples
///
/// ```
/// let arg = 5;
/// let answer = my_crate::add_one(arg);
///
/// assert_eq!(6, answer);
/// ```
pub fn add_one(x: i32) -> i32 {
    x + 1
}
```

Vytvoření sekcí

```
//! # My Crate
//!
//! `my_crate` is a collection of utilities to make performing certain
//! calculations more convenient.

/// Adds one to the number given.
// --snip--
```


Crate my_crate
See all my_crate's items
Functions
Crates
my_crate
Click or press 'S' to search, '?' for more options...
Crate my_crate
[–][src]
[–]
My Crate
my_crate is a collection of utilities to make performing certain calculations more convenient.
Functions
add_one
Adds one to the number given.

Ukázka Dokumentace

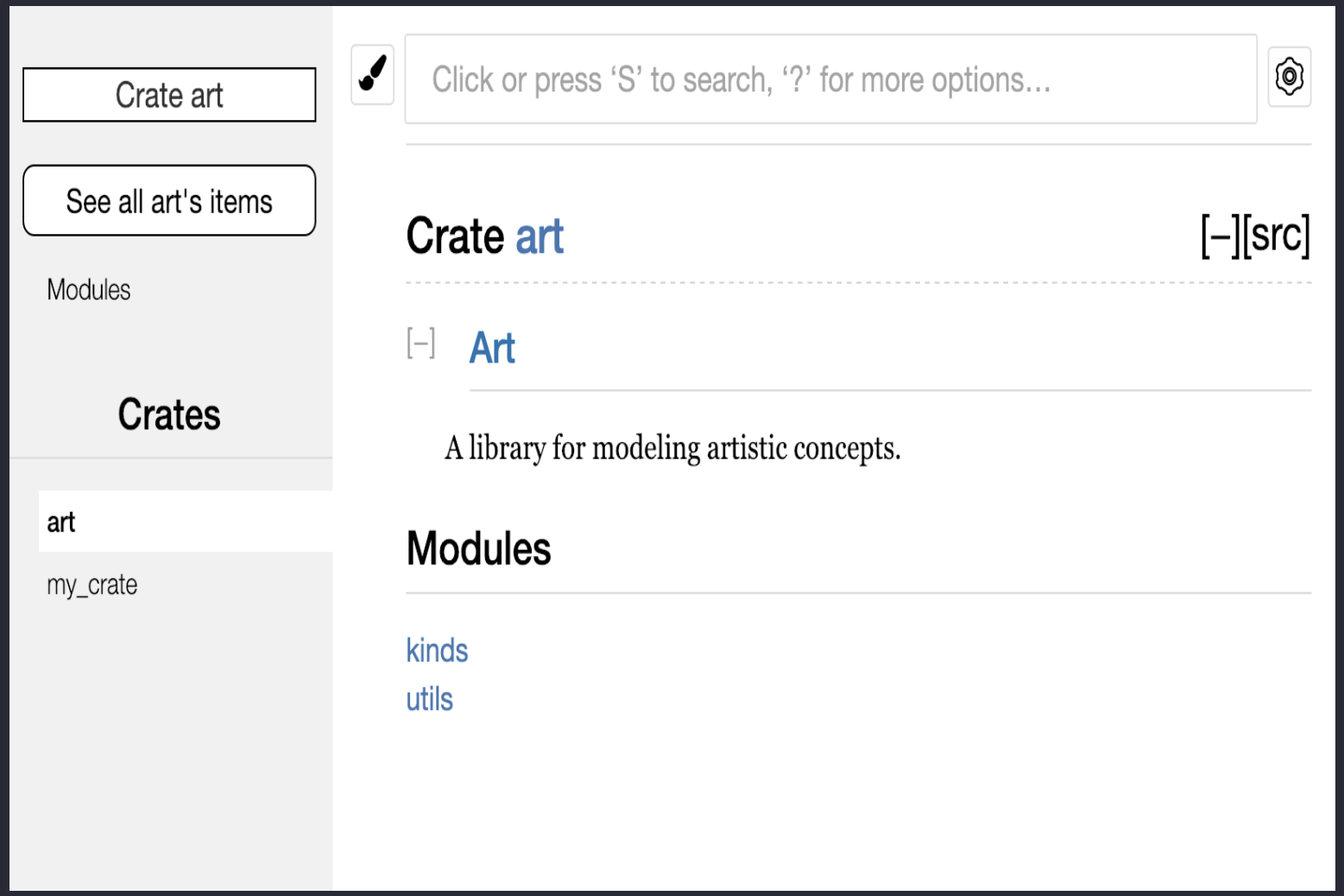
```
//! # Art
//!
//! A library for modeling artistic concepts.

pub mod kinds {
    /// The primary colors according to the RYB color model.
    pub enum PrimaryColor {
        Red,
        Yellow,
        Blue,
    }

    /// The secondary colors according to the RYB color model.
    pub enum SecondaryColor {
        Orange,
        Green,
        Purple,
    }
}

pub mod utils {
    use crate::kinds::*;

    /// Combines two primary colors in equal amounts to create
    /// a secondary color.
    pub fn mix(c1: PrimaryColor, c2: PrimaryColor) -> SecondaryColor {
        // --snip--
```


Crate art
See all art's items
Modules
Crates
art
my_crate
Click or press 'S' to search, '?' for more options...
Crate art
[–][src]
[–] Art
A library for modeling artistic concepts.
Modules
kinds
utils

Testování dokumentace

```
/// Finally, we print the sum of `x` and `y`:
///
/// ```
/// # let x = 5;
/// # let y = 6;
/// println!("{}", x + y);
/// assert_eq!(x + y, 11);
/// ```
```

? v doc testech

```
/// ```
/// use std::io;
/// let mut input = String::new();
/// io::stdin().read_line(&mut input)?;
/// # Ok::<(), io::Error>(())
/// ```
```

Poznámky k dokumentaci

Generujeme pomocí

```
cargo doc
```

Examply testujeme pomocí

```
cargo test --examples
```

Dokumentaci přes

```
cargo test --doc
```

Dotazy?

Děkuji za pozornost