免费电子书

学习

Rust

Free unaffiliated eBook created from
Stack Overflow contributors.

#rust

You can share this PDF with anyone you feel could benefit from it, downloaded the latest version from: rust

It is an unofficial and free Rust ebook created for educational purposes. All the content is extracted from Stack Overflow Documentation, which is written by many hardworking individuals at Stack Overflow. It is neither affiliated with Stack Overflow nor official Rust.

Use the content presented in this book at your own risk; it is not guaranteed to be correct nor accurate, please send your feedback and corrections to info@zzzprojects.com

1: Rust

Rust。 Rust。

1.17.0	2017427
1.16.0	2017316
1.15.1	201729
1.15.0	201722
1.14.0	20161222
1.13.0	20161110
1.12.1	○○
1.12.0	2016930
1.11.0	2016818
1.10.0	201677
1.9.0	2016526
1.8.0	2016414
1.7.0	201633
1.6.0	2016121
1.5.0	
1.4.0	20151029
1.3.0	2015917
1.2.0	201587
1.1.0	2015625
1.0.0	2015515

Beta

1.18.0	201768

Examples

println

println! print! printf。 。 {}

```
// No substitution -- the simplest kind of format string
println!("Hello World");
// Output: Hello World

// The first {} is substituted with a textual representation of
// the first argument following the format string. The second {}
// is substituted with the second argument, and so on.
println!("{} {} {}", "Hello", true, 42);
// Output: Hello true 42
```

println!println!true“true” {}Display traittext。 Rust“”。 4242。

Display [i32] Vec<i32> Option<&str> 。 RustDebug{:?}。 “ Debug。 ”


```
// No substitution -- the simplest kind of format string
println!("Hello World");
// Output: Hello World

// The first {} is substituted with a textual representation of
// the first argument following the format string. The second {}
// is substituted with the second argument, and so on.
println!("{} {} {}", "Hello", true, 42);
// Output: Hello true 42
```


Debugenable #


```
// No substitution -- the simplest kind of format string
println!("Hello World");
// Output: Hello World

// The first {} is substituted with a textual representation of
// the first argument following the format string. The second {}
// is substituted with the second argument, and so on.
println!("{} {} {}", "Hello", true, 42);
// Output: Hello true 42
```



```
// No substitution -- the simplest kind of format string
println!("Hello World");
// Output: Hello World

// The first {} is substituted with a textual representation of
// the first argument following the format string. The second {}
// is substituted with the second argument, and so on.
println!("{} {} {}", "Hello", true, 42);
// Output: Hello true 42
```

println!

```
// No substitution -- the simplest kind of format string
println!("Hello World");
// Output: Hello World

// The first {} is substituted with a textual representation of
// the first argument following the format string. The second {}
// is substituted with the second argument, and so on.
println!("{} {} {}", "Hello", true, 42);
// Output: Hello true 42
```

Rustformat!format!◦ println!format!


```
// No substitution -- the simplest kind of format string
println!("Hello World");
// Output: Hello World

// The first {} is substituted with a textual representation of
// the first argument following the format string. The second {}
// is substituted with the second argument, and so on.
println!("{} {} {}", "Hello", true, 42);
// Output: Hello true 42
```


```
// use Write trait that contains write() function
use std::io::Write;

fn main() {
    std::io::stdout().write(b"Hello, world!\n").unwrap();
}
```

- std::io::Write◦ std::io::stdout()◦
- Write::write() &[u8] b"<string>" ◦ Write::write()Result<usize, IoError> ◦
- Result::unwrap() Result<usize, IoError> -> usize ◦

RustHello Worldhello.rs hello.rs

```
fn main() {
    println!("Hello World!");
}
```

main◦ ◦ println! ◦

Rust

```
fn main() {
    println!("Hello World!");
}
```

LinuxMacOS

```
fn main() {
    println!("Hello World!");
}
```

Windows

```
fn main() {
    println!("Hello World!");
}
```

Rust - Windows*Unix*`$` symbol

```
$ curl https://sh.rustup.rs -sSf | sh
```

Rust。 。

*Linux*Arch Linux `rustup`。 *Unix*`rustc`*`cargo`*`rustup`*。

Rust

UNIXWindows*Rust*

```
$ curl https://sh.rustup.rs -sSf | sh
```

Rust。

Rust*Cargo* *Rust*。

```
$ curl https://sh.rustup.rs -sSf | sh
```

*Rust*Cargo 。

Cargo 。

`rustc`。

Rust https://riptutorial.com/zh-CN/rust/topic/362/rust

2: Drop Trait - Rust

Drop Trait◦ `mem::forget`◦

◦ Abort on Panic◦

https //doc.rust-lang.org/book/drop.html

Examples

```
use std::ops::Drop;

struct Foo(usize);

impl Drop for Foo {
    fn drop(&mut self) {
        println!("I had a {}", self.0);
    }
}
```

```
use std::ops::Drop;

#[derive(Debug)]
struct Bar(i32);

impl Bar {
    fn get<'a>(&'a mut self) -> Foo<'a> {
        let temp = self.0; // Since we will also capture `self` we..
                           // ..will have to copy the value out first
        Foo(self, temp) // Let's take the i32
    }
}

struct Foo<'a>(&'a mut Bar, i32); // We specify that we want a mutable borrow..
                                  // ..so we can put it back later on

impl<'a> Drop for Foo<'a> {
    fn drop(&mut self) {
        if self.1 < 10 { // This is just an example, you could also just put..
                         // ..it back as is
            (self.0).0 = self.1;
        }
    }
}

fn main() {
    let mut b = Bar(0);
    println!("{:?}", b);
    {
        let mut a : Foo = b.get(); // `a` now holds a reference to `b`..
        a.1 = 2;                   // .. and will hold it until end of scope
    }                              // .. here

    println!("{:?}", b);
    {
```

```
        let mut a : Foo = b.get();
        a.1 = 20;
    }
    println!("{:?}", b);
}
```

Drop。

Rc。

`println!("Dropping StructName: {:?}", self);`Drop`println!("Dropping StructName: {:?}", self);`。

Drop Trait - Rust https://riptutorial.com/zh-CN/rust/topic/7233/drop-trait-----rust

3: GUI

RustGUI。 。 rust-gtk 。 “”

Examples

Gtk +

Gtk`Cargo.toml`

```
[dependencies]
gtk = { git = "https://github.com/gtk-rs/gtk.git" }
```


```
[dependencies]
gtk = { git = "https://github.com/gtk-rs/gtk.git" }
```


GtkBoxGtk +GtkEntry

```
extern crate gtk;

use gtk::prelude::*;
use gtk::{Window, WindowType, Label, Entry, Box as GtkBox, Orientation};

fn main() {
    if gtk::init().is_err() {
        println!("Failed to initialize GTK.");
        return;
    }

    let window = Window::new(WindowType::Toplevel);

    window.connect_delete_event(|_,_| {gtk::main_quit(); Inhibit(false) });

    window.set_title("Stackoverflow. example");
    window.set_default_size(350, 70);
    let label = Label::new(Some("Some text"));

    // Create a VBox with 10px spacing
    let bx = GtkBox::new(Orientation::Vertical, 10);
    let entry = Entry::new();

    // Connect "activate" signal to anonymous function
    // that takes GtkEntry as an argument and prints it's text
    entry.connect_activate(|x| println!("{}",x.get_text().unwrap()));

    // Add our label and entry to the box
    // Do not expand or fill, zero padding
    bx.pack_start(&label, false, false, 0);
    bx.pack_start(&entry, false, false, 0);
    window.add(&bx);
    window.show_all();
    gtk::main();
}
```

GUI https://riptutorial.com/zh-CN/rust/topic/7169/gui

4: PhantomData

Examples

PhantomData

`PhantomData`Struct。

```
use std::marker::PhantomData;

struct Authenticator<T: GetInstance> {
    _marker: PhantomData<*const T>, // Using `*const T` indicates that we do not own a T
}

impl<T: GetInstance> Authenticator<T> {
    fn new() -> Authenticator<T> {
        Authenticator {
            _marker: PhantomData,
        }
    }

    fn auth(&self, id: i64) -> bool {
        T::get_instance(id).is_some()
    }
}

trait GetInstance {
    type Output; // Using nightly this could be defaulted to `Self`
    fn get_instance(id: i64) -> Option<Self::Output>;
}

struct Foo;

impl GetInstance for Foo {
    type Output = Self;
    fn get_instance(id: i64) -> Option<Foo> {
        // Here you could do something like a Database lookup or similarly
        if id == 1 {
            Some(Foo)
        } else {
            None
        }
    }
}

struct User;

impl GetInstance for User {
    type Output = Self;
    fn get_instance(id: i64) -> Option<User> {
        // Here you could do something like a Database lookup or similarly
        if id == 2 {
            Some(User)
        } else {
            None
        }
```

```
    }
}

fn main() {
    let user_auth = Authenticator::<User>::new();
    let other_auth = Authenticator::<Foo>::new();

    assert!(user_auth.auth(2));
    assert!(!user_auth.auth(1));

    assert!(other_auth.auth(1));
    assert!(!other_auth.auth(2));

}
```

PhantomData https://riptutorial.com/zh-CN/rust/topic/7226/phantomdata

5: rustup

rustup◦

Examples

```
curl https://sh.rustup.rs -sSf | sh
```

◦

```
curl https://sh.rustup.rs -sSf | sh
```

```
curl https://sh.rustup.rs -sSf | sh
```

rustup https://riptutorial.com/zh-CN/rust/topic/8942/rustup

6: SERDE

SERDE**SER** ialization JSONXML。 SerdeJSONYAMLTOMLBSONPickleXML。

Examples

↔JSON

main.rs

```
extern crate serde;
extern crate serde_json;

// Import this crate to derive the Serialize and Deserialize traits.
#[macro_use] extern crate serde_derive;

#[derive(Serialize, Deserialize, Debug)]
struct Point {
    x: i32,
    y: i32,
}

fn main() {
    let point = Point { x: 1, y: 2 };

    // Convert the Point to a packed JSON string. To convert it to
    // pretty JSON with indentation, use `to_string_pretty` instead.
    let serialized = serde_json::to_string(&point).unwrap();

    // Prints serialized = {"x":1,"y":2}
    println!("serialized = {}", serialized);

    // Convert the JSON string back to a Point.
    let deserialized: Point = serde_json::from_str(&serialized).unwrap();

    // Prints deserialized = Point { x: 1, y: 2 }
    println!("deserialized = {:?}", deserialized);
}
```

Cargo.toml

```
extern crate serde;
extern crate serde_json;

// Import this crate to derive the Serialize and Deserialize traits.
#[macro_use] extern crate serde_derive;

#[derive(Serialize, Deserialize, Debug)]
struct Point {
    x: i32,
```

```
    y: i32,
}

fn main() {
    let point = Point { x: 1, y: 2 };

    // Convert the Point to a packed JSON string. To convert it to
    // pretty JSON with indentation, use `to_string_pretty` instead.
    let serialized = serde_json::to_string(&point).unwrap();

    // Prints serialized = {"x":1,"y":2}
    println!("serialized = {}", serialized);

    // Convert the JSON string back to a Point.
    let deserialized: Point = serde_json::from_str(&serialized).unwrap();

    // Prints deserialized = Point { x: 1, y: 2 }
    println!("deserialized = {:?}", deserialized);
}
```

```
extern crate serde;
extern crate serde_json;

macro_rules! enum_str {
    ($name:ident { $($variant:ident($str:expr), )* }) => {
        #[derive(Clone, Copy, Debug, Eq, PartialEq)]
        pub enum $name {
            $($variant,)*
        }

        impl ::serde::Serialize for $name {
            fn serialize<S>(&self, serializer: S) -> Result<S::Ok, S::Error>
                where S: ::serde::Serializer,
            {
                // Serialize the enum as a string.
                serializer.serialize_str(match *self {
                    $( $name::$variant => $str, )*
                })
            }
        }

        impl ::serde::Deserialize for $name {
            fn deserialize<D>(deserializer: D) -> Result<Self, D::Error>
                where D: ::serde::Deserializer,
            {
                struct Visitor;

                impl ::serde::de::Visitor for Visitor {
                    type Value = $name;

                    fn expecting(&self, formatter: &mut ::std::fmt::Formatter) ->
::std::fmt::Result {
                        write!(formatter, "a string for {}", stringify!($name))
                    }

                    fn visit_str<E>(self, value: &str) -> Result<$name, E>
                        where E: ::serde::de::Error,
                    {
                        match value {
                            $( $str => Ok($name::$variant), )*
```

```
                            _ => Err(E::invalid_value(::serde::de::Unexpected::Other(
                                &format!("unknown {} variant: {}", stringify!($name), value)
                            ), &self)),
                        }
                    }
                }

                // Deserialize the enum from a string.
                deserializer.deserialize_str(Visitor)
            }
        }
    }
}

enum_str!(LanguageCode {
    English("en"),
    Spanish("es"),
    Italian("it"),
    Japanese("ja"),
    Chinese("zh"),
});

fn main() {
    use LanguageCode::*;
    let languages = vec![English, Spanish, Italian, Japanese, Chinese];

    // Prints ["en","es","it","ja","zh"]
    println!("{}", serde_json::to_string(&languages).unwrap());

    let input = r#" "ja" "#;
    assert_eq!(Japanese, serde_json::from_str(input).unwrap());
}
```

camelCase

```
extern crate serde;
extern crate serde_json;
#[macro_use] extern crate serde_derive;

#[derive(Serialize)]
struct Person {
    #[serde(rename="firstName")]
    first_name: String,
    #[serde(rename="lastName")]
    last_name: String,
}

fn main() {
    let person = Person {
        first_name: "Joel".to_string(),
        last_name: "Spolsky".to_string(),
    };

    let json = serde_json::to_string_pretty(&person).unwrap();

    // Prints:
    //
    //    {
    //      "firstName": "Joel",
```

```
    //      "lastName": "Spolsky"
    //    }
    println!("{}", json);
}
```

```
extern crate serde;
extern crate serde_json;
#[macro_use] extern crate serde_derive;

#[derive(Deserialize, Debug)]
struct Request {
    // Use the result of a function as the default if "resource" is
    // not included in the input.
    #[serde(default="default_resource")]
    resource: String,

    // Use the type's implementation of std::default::Default if
    // "timeout" is not included in the input.
    #[serde(default)]
    timeout: Timeout,

    // Use a method from the type as the default if "priority" is not
    // included in the input. This may also be a trait method.
    #[serde(default="Priority::lowest")]
    priority: Priority,
}

fn default_resource() -> String {
    "/".to_string()
}

/// Timeout in seconds.
#[derive(Deserialize, Debug)]
struct Timeout(u32);
impl Default for Timeout {
    fn default() -> Self {
        Timeout(30)
    }
}

#[derive(Deserialize, Debug)]
enum Priority { ExtraHigh, High, Normal, Low, ExtraLow }
impl Priority {
    fn lowest() -> Self { Priority::ExtraLow }
}

fn main() {
    let json = r#"
        [
          {
            "resource": "/users"
          },
          {
            "timeout": 5,
            "priority": "High"
          }
        ]
    "#;

    let requests: Vec<Request> = serde_json::from_str(json).unwrap();
```

```
    // The first request has resource="/users", timeout=30, priority=ExtraLow
    println!("{:?}", requests[0]);

    // The second request has resource="/", timeout=5, priority=High
    println!("{:?}", requests[1]);
}
```

```
extern crate serde;
extern crate serde_json;
#[macro_use] extern crate serde_derive;

use std::collections::BTreeMap as Map;

#[derive(Serialize)]
struct Resource {
    // Always serialized.
    name: String,

    // Never serialized.
    #[serde(skip_serializing)]
    hash: String,

    // Use a method to decide whether the field should be skipped.
    #[serde(skip_serializing_if="Map::is_empty")]
    metadata: Map<String, String>,
}

fn main() {
    let resources = vec![
        Resource {
            name: "Stack Overflow".to_string(),
            hash: "b6469c3f31653d281bbbfa6f94d60fea130abe38".to_string(),
            metadata: Map::new(),
        },
        Resource {
            name: "GitHub".to_string(),
            hash: "5cb7a0c47e53854cd00e1a968de5abce1c124601".to_string(),
            metadata: {
                let mut metadata = Map::new();
                metadata.insert("headquarters".to_string(),
                                "San Francisco".to_string());
                metadata
            },
        },
    ];

    let json = serde_json::to_string_pretty(&resources).unwrap();

    // Prints:
    //
    //    [
    //      {
    //        "name": "Stack Overflow"
    //      },
    //      {
    //        "name": "GitHub",
    //        "metadata": {
    //          "headquarters": "San Francisco"
    //        }
```

```
    //      }
    //    ]
    println!("{}", json);
}
```

Serialize

```
impl<K, V> Serialize for MyMap<K, V>
    where K: Serialize,
          V: Serialize
{
    fn serialize<S>(&self, serializer: S) -> Result<S::Ok, S::Error>
        where S: Serializer
    {
        let mut state = serializer.serialize_map(Some(self.len()))?;
        for (k, v) in self {
            state.serialize_entry(k, v)?;
        }
        state.end()
    }
}
```

```
// A Visitor is a type that holds methods that a Deserializer can drive
// depending on what is contained in the input data.
//
// In the case of a map we need generic type parameters K and V to be
// able to set the output type correctly, but don't require any state.
// This is an example of a "zero sized type" in Rust. The PhantomData
// keeps the compiler from complaining about unused generic type
// parameters.
struct MyMapVisitor<K, V> {
    marker: PhantomData<MyMap<K, V>>
}

impl<K, V> MyMapVisitor<K, V> {
    fn new() -> Self {
        MyMapVisitor {
            marker: PhantomData
        }
    }
}

// This is the trait that Deserializers are going to be driving. There
// is one method for each type of data that our type knows how to
// deserialize from. There are many other methods that are not
// implemented here, for example deserializing from integers or strings.
// By default those methods will return an error, which makes sense
// because we cannot deserialize a MyMap from an integer or string.
impl<K, V> de::Visitor for MyMapVisitor<K, V>
    where K: Deserialize,
          V: Deserialize
{
    // The type that our Visitor is going to produce.
    type Value = MyMap<K, V>;

    // Deserialize MyMap from an abstract "map" provided by the
    // Deserializer. The MapVisitor input is a callback provided by
    // the Deserializer to let us see each entry in the map.
```

```
    fn visit_map<M>(self, mut visitor: M) -> Result<Self::Value, M::Error>
        where M: de::MapVisitor
    {
        let mut values = MyMap::with_capacity(visitor.size_hint().0);

        // While there are entries remaining in the input, add them
        // into our map.
        while let Some((key, value)) = visitor.visit()? {
            values.insert(key, value);
        }

        Ok(values)
    }

    // As a convenience, provide a way to deserialize MyMap from
    // the abstract "unit" type. This corresponds to `null` in JSON.
    // If your JSON contains `null` for a field that is supposed to
    // be a MyMap, we interpret that as an empty map.
    fn visit_unit<E>(self) -> Result<Self::Value, E>
        where E: de::Error
    {
        Ok(MyMap::new())
    }

    // When an unexpected data type is encountered, this method will
    // be invoked to inform the user what is actually expected.
    fn expecting(&self, formatter: &mut fmt::Formatter) -> fmt::Result {
        write!(formatter, "a map or `null`")
    }
}

// This is the trait that informs Serde how to deserialize MyMap.
impl<K, V> Deserialize for MyMap<K, V>
    where K: Deserialize,
          V: Deserialize
{
    fn deserialize<D>(deserializer: D) -> Result<Self, D::Error>
        where D: Deserializer
    {
        // Instantiate our Visitor and ask the Deserializer to drive
        // it over the input data, resulting in an instance of MyMap.
        deserializer.deserialize_map(MyMapVisitor::new())
    }
}
```

Vec

◦ ◦

```
extern crate serde;
extern crate serde_json;
#[macro_use] extern crate serde_derive;
use serde::{de, Deserialize, Deserializer};

use std::cmp;
use std::fmt;
use std::marker::PhantomData;

#[derive(Deserialize)]
```

```
struct Outer {
    id: String,

    // Deserialize this field by computing the maximum value of a sequence
    // (JSON array) of values.
    #[serde(deserialize_with = "deserialize_max")]
    // Despite the struct field being named `max_value`, it is going to come
    // from a JSON field called `values`.
    #[serde(rename(deserialize = "values"))]
    max_value: u64,
}

/// Deserialize the maximum of a sequence of values. The entire sequence
/// is not buffered into memory as it would be if we deserialize to Vec<T>
/// and then compute the maximum later.
///
/// This function is generic over T which can be any type that implements
/// Ord. Above, it is used with T=u64.
fn deserialize_max<T, D>(deserializer: D) -> Result<T, D::Error>
    where T: Deserialize + Ord,
          D: Deserializer
{
    struct MaxVisitor<T>(PhantomData<T>);

    impl<T> de::Visitor for MaxVisitor<T>
        where T: Deserialize + Ord
    {
        /// Return type of this visitor. This visitor computes the max of a
        /// sequence of values of type T, so the type of the maximum is T.
        type Value = T;

        fn expecting(&self, formatter: &mut fmt::Formatter) -> fmt::Result {
            write!(formatter, "a sequence of numbers")
        }

        fn visit_seq<V>(self, mut visitor: V) -> Result<T, V::Error>
            where V: de::SeqVisitor
        {
            // Start with max equal to the first value in the seq.
            let mut max = match visitor.visit()? {
                Some(value) => value,
                None => {
                    // Cannot take the maximum of an empty seq.
                    let msg = "no values in seq when looking for maximum";
                    return Err(de::Error::custom(msg));
                }
            };

            // Update the max while there are additional values.
            while let Some(value) = visitor.visit()? {
                max = cmp::max(max, value);
            }

            Ok(max)
        }
    }

    // Create the visitor and ask the deserializer to drive it. The
    // deserializer will call visitor.visit_seq if a seq is present in
    // the input data.
    let visitor = MaxVisitor(PhantomData);
```

```
    deserializer.deserialize_seq(visitor)
}

fn main() {
    let j = r#"
        {
          "id": "demo-deserialize-max",
          "values": [
            256,
            100,
            384,
            314,
            271
          ]
        }
    "#;

    let out: Outer = serde_json::from_str(j).unwrap();

    // Prints "max value: 384"
    println!("max value: {}", out.max_value);
}
```

`SerializeDeserialize`Serde。 `T: SerializeT` `T: DeserializeT: DeserializeT` `T: Deserialize`。 。 Serde。

```
extern crate serde;
extern crate serde_json;
#[macro_use] extern crate serde_derive;

use serde::de::{self, Deserialize, Deserializer};

use std::fmt::Display;
use std::str::FromStr;

#[derive(Deserialize, Debug)]
struct Outer<'a, S, T: 'a + ?Sized> {
    // When deriving the Deserialize impl, Serde would want to generate a bound
    // `S: Deserialize` on the type of this field. But we are going to use the
    // type's `FromStr` impl instead of its `Deserialize` impl by going through
    // `deserialize_from_str`, so we override the automatically generated bound
    // by the one required for `deserialize_from_str`.
    #[serde(deserialize_with = "deserialize_from_str")]
    #[serde(bound(deserialize = "S: FromStr, S::Err: Display"))]
    s: S,

    // Here Serde would want to generate a bound `T: Deserialize`. That is a
    // stricter condition than is necessary. In fact, the `main` function below
    // uses T=str which does not implement Deserialize. We override the
    // automatically generated bound by a looser one.
    #[serde(bound(deserialize = "Ptr<'a, T>: Deserialize"))]
    ptr: Ptr<'a, T>,
}

/// Deserialize a type `S` by deserializing a string, then using the `FromStr`
/// impl of `S` to create the result. The generic type `S` is not required to
/// implement `Deserialize`.
fn deserialize_from_str<S, D>(deserializer: D) -> Result<S, D::Error>
    where S: FromStr,
```

```
          S::Err: Display,
          D: Deserializer
{
    let s: String = try!(Deserialize::deserialize(deserializer));
    S::from_str(&s).map_err(|e| de::Error::custom(e.to_string()))
}

/// A pointer to `T` which may or may not own the data. When deserializing we
/// always want to produce owned data.
#[derive(Debug)]
enum Ptr<'a, T: 'a + ?Sized> {
    Ref(&'a T),
    Owned(Box<T>),
}

impl<'a, T: 'a + ?Sized> Deserialize for Ptr<'a, T>
    where Box<T>: Deserialize
{
    fn deserialize<D>(deserializer: D) -> Result<Self, D::Error>
        where D: Deserializer
    {
        let box_t = try!(Deserialize::deserialize(deserializer));
        Ok(Ptr::Owned(box_t))
    }
}

fn main() {
    let j = r#"
        {
          "s": "1234567890",
          "ptr": "owned"
        }
    "#;

    let result: Outer<u64, str> = serde_json::from_str(j).unwrap();

    // result = Outer { s: 1234567890, ptr: Owned("owned") }
    println!("result = {:?}", result);
}
```

SerializeDeserialize

Rustimpl。 `Deref` `Deref`SerializeDeserialize。

```
use serde::{Serialize, Serializer, Deserialize, Deserializer};
use std::ops::Deref;

// Pretend this module is from some other crate.
mod not_my_crate {
    pub struct Data { /* ... */ }
}

// This single-element tuple struct is called a newtype struct.
struct Data(not_my_crate::Data);

impl Serialize for Data {
    fn serialize<S>(&self, serializer: S) -> Result<S::Ok, S::Error>
        where S: Serializer
    {
```

```
        // Any implementation of Serialize.
    }
}

impl Deserialize for Data {
    fn deserialize<D>(deserializer: D) -> Result<Self, D::Error>
        where D: Deserializer
    {
        // Any implementation of Deserialize.
    }
}

// Enable `Deref` coercion.
impl Deref for Data {
    type Target = not_my_crate::Data;
    fn deref(&self) -> &Self::Target {
        &self.0
    }
}

// Now `Data` can be used in ways that require it to implement
// Serialize and Deserialize.
#[derive(Serialize, Deserialize)]
struct Outer {
    id: u64,
    name: String,
    data: Data,
}
```

SERDE https://riptutorial.com/zh-CN/rust/topic/1170/serde

7: TCP

Examples

TCPecho

std :: net :: TcpListener。“”。 。

```
use std::thread;
use std::net::{TcpListener, TcpStream, Shutdown};
use std::io::{Read, Write};

fn handle_client(mut stream: TcpStream) {
    let mut data = [0 as u8; 50]; // using 50 byte buffer
    while match stream.read(&mut data) {
        Ok(size) => {
            // echo everything!
            stream.write(&data[0..size]).unwrap();
            true
        },
        Err(_) => {
            println!("An error occurred, terminating connection with {}",
stream.peer_addr().unwrap());
            stream.shutdown(Shutdown::Both).unwrap();
            false
        }
    } {}
}

fn main() {
    let listener = TcpListener::bind("0.0.0.0:3333").unwrap();
    // accept connections and process them, spawning a new thread for each one
    println!("Server listening on port 3333");
    for stream in listener.incoming() {
        match stream {
            Ok(stream) => {
                println!("New connection: {}", stream.peer_addr().unwrap());
                thread::spawn(move|| {
                    // connection succeeded
                    handle_client(stream)
                });
            }
            Err(e) => {
                println!("Error: {}", e);
                /* connection failed */
            }
        }
    }
    // close the socket server
    drop(listener);
}
```

```
use std::thread;
use std::net::{TcpListener, TcpStream, Shutdown};
use std::io::{Read, Write};
```

```
fn handle_client(mut stream: TcpStream) {
    let mut data = [0 as u8; 50]; // using 50 byte buffer
    while match stream.read(&mut data) {
        Ok(size) => {
            // echo everything!
            stream.write(&data[0..size]).unwrap();
            true
        },
        Err(_) => {
            println!("An error occurred, terminating connection with {}",
stream.peer_addr().unwrap());
            stream.shutdown(Shutdown::Both).unwrap();
            false
        }
    } {}
}

fn main() {
    let listener = TcpListener::bind("0.0.0.0:3333").unwrap();
    // accept connections and process them, spawning a new thread for each one
    println!("Server listening on port 3333");
    for stream in listener.incoming() {
        match stream {
            Ok(stream) => {
                println!("New connection: {}", stream.peer_addr().unwrap());
                thread::spawn(move|| {
                    // connection succeeded
                    handle_client(stream)
                });
            }
            Err(e) => {
                println!("Error: {}", e);
                /* connection failed */
            }
        }
    }
    // close the socket server
    drop(listener);
}
```

TCP https://riptutorial.com/zh-CN/rust/topic/1350/tcp

8:

Rust`unsafe` ◦

Examples

◦ ◦

```
use std::cell::UnsafeCell;
use std::sync::Arc;
use std::thread;

// `UnsafeCell` is a zero-cost wrapper which informs the compiler that "what it
// contains might be shared mutably." This is used only for static analysis, and
// gets optimized away in release builds.
struct RacyUsize(UnsafeCell<usize>);

// Since UnsafeCell is not thread-safe, the compiler will not auto-impl Sync for
// any type containig it. And manually impl-ing Sync is "unsafe".
unsafe impl Sync for RacyUsize {}

impl RacyUsize {
    fn new(v: usize) -> RacyUsize {
        RacyUsize(UnsafeCell::new(v))
    }

    fn get(&self) -> usize {
        // UnsafeCell::get() returns a raw pointer to the value it contains
        // Dereferencing a raw pointer is also "unsafe"
        unsafe { *self.0.get() }
    }

    fn set(&self, v: usize) { // note: `&self` and not `&mut self`
        unsafe { *self.0.get() = v }
    }
}

fn main() {
    let racy_num = Arc::new(RacyUsize::new(0));

    let mut handlers = vec![];
    for _ in 0..10 {
        let racy_num = racy_num.clone();
        handlers.push(thread::spawn(move || {
            for i in 0..1000 {
                if i % 200 == 0 {
                    // give up the time slice to scheduler
                    thread::yield_now();
                    // this is needed to interleave the threads so as to observe
                    // data race, otherwise the threads will most likely be
                    // scheduled one after another.
                }

                // increment by one
                racy_num.set(racy_num.get() + 1);
            }
```

```
        }));
    }

    for th in handlers {
        th.join().unwrap();
    }

    println!("{}", racy_num.get());
}
```

`10000` 10×1000。

。 。 /。

“”Rust。

https://riptutorial.com/zh-CN/rust/topic/6018/

9:

RustOS 。

rust-lang / rfcs。

RFC https //github.com/rust-lang/rfcs/issues/1368

Examples

chan

OSaltough 。

BurntSushi / chan。

```
#[macro_use]
extern crate chan;
extern crate chan_signal;

use chan_signal::Signal;

fn main() {
    // Signal gets a value when the OS sent a INT or TERM signal.
    let signal = chan_signal::notify(&[Signal::INT, Signal::TERM]);
    // When our work is complete, send a sentinel value on `sdone`.
    let (sdone, rdone) = chan::sync(0);
    // Run work.
    ::std::thread::spawn(move || run(sdone));

    // Wait for a signal or for work to be done.
    chan_select! {
        signal.recv() -> signal => {
            println!("received signal: {:?}", signal)
        },
        rdone.recv() => {
            println!("Program completed normally.");
        }
    }
}

fn run(_sdone: chan::Sender<()>) {
    println!("Running work for 5 seconds.");
    println!("Can you send a signal quickly enough?");
    // Do some work.
    ::std::thread::sleep_ms(5000);

    // _sdone gets dropped which closes the channel and causes `rdone`
    // to unblock.
}
```

nix。

nix crateUNIX Rust API 。

```
use nix::sys::signal;

extern fn handle_sigint(_:i32) {
    // Be careful here...
}

fn main() {
    let sig_action = signal::SigAction::new(handle_sigint,
                                            signal::SockFlag::empty(),
                                            signal::SigSet::empty());
    signal::sigaction(signal::SIGINT, &sig_action);
}
```

Tokio

tokiotokio。 。

```
extern crate futures;
extern crate tokio_core;
extern crate tokio_signal;

use futures::{Future, Stream};
use tokio_core::reactor::Core
use tokio_signal::unix::{self as unix_signal, Signal};
use std::thread::{self, sleep};
use std::time::Duration;
use std::sync::mpsc::{channel, Receiver};

fn run(signals: Receiver<i32>) {
    loop {
        if let Some(signal) = signals.try_recv() {
            eprintln!("received {} signal");
        }
        sleep(Duration::from_millis(1));
    }
}

fn main() {
    // Create channels for sending and receiving signals
    let (signals_tx, signals_rx) = channel();

    // Execute the program with the receiving end of the channel
    // for listening to any signals that are sent to it.
    thread::spawn(move || run(signals_rx));

    // Create a stream that will select over SIGINT, SIGTERM, and SIGHUP signals.
    let signals = Signal::new(unix_signal::SIGINT, &handle).flatten_stream()
        .select(Signal::new(unix_signal::SIGTERM, &handle).flatten_stream())
        .select(Signal::new(unix_signal::SIGHUP, &handle).flatten_stream());

    // Execute the event loop that will listen for and transmit received
    // signals to the shell.
    core.run(signals.for_each(|signal| {
        let _ = signals_tx.send(signal);
        Ok(())
    })).unwrap();
```

```
}
```

https://riptutorial.com/zh-CN/rust/topic/3995/

10:

。

- ABC//ABC
- AB//AB.
- A//, A
- //

Examples

Rust。

```
// Tuples in Rust are comma-separated values or types enclosed in parentheses.
let _ = ("hello", 42, true);
// The type of a tuple value is a type tuple with the same number of elements.
// Each element in the type tuple is the type of the corresponding value element.
let _: (i32, bool) = (42, true);
// Tuples can have any number of elements, including one ..
let _: (bool,) = (true,);
// .. or even zero!
let _: () = ();
// this last type has only one possible value, the empty tuple ()
// this is also the type (and value) of the return value of functions
// that do not return a value ..
let _: () = println!("hello");
// .. or of expressions with no value.
let mut a = 0;
let _: () = if true { a += 1; };
```

`match if letlet。 match`

```
fn foo(x: (&str, isize, bool)) {
    match x {
        (_, 42, _) => println!("it's 42"),
        (_, _, false) => println!("it's not true"),
        _ => println!("it's something else"),
    }
}
```

`if let`

```
fn foo(x: (&str, isize, bool)) {
    match x {
        (_, 42, _) => println!("it's 42"),
        (_, _, false) => println!("it's not true"),
        _ => println!("it's something else"),
    }
}
```

`let` -deconstruction

```
fn foo(x: (&str, isize, bool)) {
    match x {
        (_, 42, _) => println!("it's 42"),
        (_, _, false) => println!("it's not true"),
        _ => println!("it's something else"),
    }
}
```

.nn

```
let x = ("hello", 42, true);
assert_eq!(x.0, "hello");
assert_eq!(x.1, 42);
assert_eq!(x.2, true);
```


```
let x = ("hello", 42, true);
assert_eq!(x.0, "hello");
assert_eq!(x.1, 42);
assert_eq!(x.2, true);
```


-
-

(1, "Hello")i32&str2(i32, &'static str) ◦

```
let tuple =  (1, "Hello");
println!("First element: {}, second element: {}", tuple.0, tuple.1);
```


```
let tuple =  (1, "Hello");
println!("First element: {}, second element: {}", tuple.0, tuple.1);
```

0:(() ◦ ->◦ *Rust“type”* ◦

1 (a,) 1◦ (a)a ◦

(1, "Hello",) ◦

Rust ◦ 12◦ ◦

◦

```
// It's possible to unpack tuples to assign their inner values to variables
let tup = (0, 1, 2);
// Unpack the tuple into variables a, b, and c
let (a, b, c) = tup;

assert_eq!(a, 0);
assert_eq!(b, 1);

// This works for nested data structures and other complex data types
let complex = ((1, 2), 3, Some(0));
```

```
let (a, b, c) = complex;
let (aa, ab) = a;

assert_eq!(aa, 1);
assert_eq!(ab, 2);
```

https://riptutorial.com/zh-CN/rust/topic/3941/

11:

- const IDENTIFIERtype = constexpr;
- static [mut] IDENTIFIERtype = expr;
- lazy_static {static ref IDENTIFIERtype = expr; }

- `const`。
- `static`。
- `static mut`unsafe。
- std :: sync :: Mutex
- `lazy_static`。 `thread_local`LocalKey<T>

Examples

`const`。

```
const DEADBEEF: u64 = 0xDEADBEEF;

fn main() {
    println("{:X}", DEADBEEF);
}
```


```
const DEADBEEF: u64 = 0xDEADBEEF;

fn main() {
    println("{:X}", DEADBEEF);
}
```


`static`。

```
static HELLO_WORLD: &'static str = "Hello, world!";

fn main() {
    println("{}", HELLO_WORLD);
}
```


```
static HELLO_WORLD: &'static str = "Hello, world!";

fn main() {
    println("{}", HELLO_WORLD);
}
```


lazy_static

`lazy_static`。 HashMap。

Cargo.toml

```
[dependencies]
lazy_static = "0.1.*"
```

main.rs

```
[dependencies]
lazy_static = "0.1.*"
```

◦ ◦

```
use std::cell::RefCell;
use std::thread;

thread_local! {
    static FOO: RefCell<f32> = RefCell::new(1.0);
}

// When this macro expands, `FOO` gets type `thread::LocalKey<RefCell<f32>>`.
//
// Side note: One of its private member is a pointer to a function which is
// responsible for returning the thread-local object. Having all its members
// `Sync` [0], `LocalKey` is also implicitly `Sync`.
//
// [0]: As of writing this, `LocalKey` just has 2 function-pointers as members

fn main() {
    FOO.with(|foo| {
        // `foo` is of type `&RefCell<f64>`
        *foo.borrow_mut() = 3.0;
    });

    thread::spawn(move|| {
        // Note that static objects do not move (`FOO` is the same everywhere),
        // but the `foo` you get inside the closure will of course be different.
        FOO.with(|foo| {
            println!("inner: {}", *foo.borrow());
        });
    }).join().unwrap();

    FOO.with(|foo| {
        println!("main: {}", *foo.borrow());
    });
}
```

```
use std::cell::RefCell;
use std::thread;

thread_local! {
    static FOO: RefCell<f32> = RefCell::new(1.0);
}

// When this macro expands, `FOO` gets type `thread::LocalKey<RefCell<f32>>`.
//
// Side note: One of its private member is a pointer to a function which is
// responsible for returning the thread-local object. Having all its members
// `Sync` [0], `LocalKey` is also implicitly `Sync`.
//
```

```
// [0]: As of writing this, `LocalKey` just has 2 function-pointers as members

fn main() {
    FOO.with(|foo| {
        // `foo` is of type `&RefCell<f64>`
        *foo.borrow_mut() = 3.0;
    });

    thread::spawn(move|| {
        // Note that static objects do not move (`FOO` is the same everywhere),
        // but the `foo` you get inside the closure will of course be different.
        FOO.with(|foo| {
            println!("inner: {}", *foo.borrow());
        });
    }).join().unwrap();

    FOO.with(|foo| {
        println!("main: {}", *foo.borrow());
    });
}
```

mut_staticmut

static mut ◦

◦

```
#[macro_use]
extern crate lazy_static;
extern crate mut_static;

use mut_static::MutStatic;

pub struct MyStruct { value: usize }

impl MyStruct {
    pub fn new(v: usize) -> Self{
        MyStruct { value: v }
    }
    pub fn getvalue(&self) -> usize { self.value }
    pub fn setvalue(&mut self, v: usize) { self.value = v }
}

lazy_static! {
    static ref MY_GLOBAL_STATE: MutStatic<MyStruct> = MutStatic::new();
}

fn main() {
    // Here, I call .set on the MutStatic to put data inside it.
    // This can fail.
    MY_GLOBAL_STATE.set(MyStruct::new(0)).unwrap();
    {
        // Using the global state immutably is easy...
        println!("Before mut: {}",
                 MY_GLOBAL_STATE.read().unwrap().getvalue());
    }
    {
         // Using it mutably is too...
```

```
         let mut mut_handle = MY_GLOBAL_STATE.write().unwrap();
         mut_handle.setvalue(3);
         println!("Changed value to 3.");
    }
    {
        // As long as there's a scope change we can get the
        // immutable version again...
        println!("After mut: {}",
                 MY_GLOBAL_STATE.read().unwrap().getvalue());
    }
    {
        // But beware! Anything can change global state!
        foo();
        println!("After foo: {}",
                 MY_GLOBAL_STATE.read().unwrap().getvalue());
    }

}

// Note that foo takes no parameters
fn foo() {
    let val;
    {
        val = MY_GLOBAL_STATE.read().unwrap().getvalue();
    }
    {
        let mut mut_handle =
            MY_GLOBAL_STATE.write().unwrap();
        mut_handle.setvalue(val + 1);
    }
}
```

```
#[macro_use]
extern crate lazy_static;
extern crate mut_static;

use mut_static::MutStatic;

pub struct MyStruct { value: usize }

impl MyStruct {
    pub fn new(v: usize) -> Self{
        MyStruct { value: v }
    }
    pub fn getvalue(&self) -> usize { self.value }
    pub fn setvalue(&mut self, v: usize) { self.value = v }
}

lazy_static! {
    static ref MY_GLOBAL_STATE: MutStatic<MyStruct> = MutStatic::new();
}

fn main() {
    // Here, I call .set on the MutStatic to put data inside it.
    // This can fail.
    MY_GLOBAL_STATE.set(MyStruct::new(0)).unwrap();
    {
        // Using the global state immutably is easy...
        println!("Before mut: {}",
                 MY_GLOBAL_STATE.read().unwrap().getvalue());
```

```
    }
    {
         // Using it mutably is too...
         let mut mut_handle = MY_GLOBAL_STATE.write().unwrap();
         mut_handle.setvalue(3);
         println!("Changed value to 3.");
    }
    {
        // As long as there's a scope change we can get the
        // immutable version again...
        println!("After mut: {}",
                 MY_GLOBAL_STATE.read().unwrap().getvalue());
    }
    {
        // But beware! Anything can change global state!
        foo();
        println!("After foo: {}",
                 MY_GLOBAL_STATE.read().unwrap().getvalue());
    }

}

// Note that foo takes no parameters
fn foo() {
    let val;
    {
        val = MY_GLOBAL_STATE.read().unwrap().getvalue();
    }
    {
        let mut mut_handle =
            MY_GLOBAL_STATE.write().unwrap();
        mut_handle.setvalue(val + 1);
    }
}
```

Rust。 `foo()`。 。

。 - MutStatic。

https://riptutorial.com/zh-CN/rust/topic/1244/

12:

- [featureasm] //asm
- asm<template><output><input><clobbers><options>//“NOP”“ADDeax4”。

Examples

asm

Rust。 `asm!`

```
 #![feature(asm)]
```

`asm!unsafe`

```
 #![feature(asm)]
```

`x86` 。 ARM。

```
#![feature(asm)]

// Any valid x86 code is valid for x86_64 as well. Be careful
// not to write x86_64 only code while including x86 in the
// compilation targets!
#[cfg(any(target_arch = "x86", target_arch = "x86_64"))]
fn do_nothing() {
    unsafe {
        asm!("NOP");
    }
}

#[cfg(not(any(target_arch = "x86", target_arch = "x86_64"))]
fn do_nothing() {
    // This is an alternative implementation that doesn't use any asm!
    // calls. Therefore, it should be safe to use as a fallback.
}
```

```
#![feature(asm)]

#[cfg(any(target_arch="x86", target_arch="x86_64"))]
fn subtract(first: i32, second: i32) {
   unsafe {
        // Output values must either be unassigned (let result;) or mutable.
        let result: i32;
        // Each value that you pass in will be in a certain register, which
        // can be accessed with $0, $1, $2...
        //
        // The registers are assigned from left to right, so $0 is the
        // register containing 'result', $1 is the register containing
        // 'first' and $2 is the register containing 'second'.
        //
        // Rust uses AT&T syntax by default, so the format is:
```

```
        // SUB source, destination
        // which is equivalent to:
        // destination -= source;
        //
        // Because we want to subtract the first from the second,
        // we use the 0 constraint on 'first' to use the same
        // register as the output.
        // Therefore, we're doing:
        // SUB second, first
        // and getting the value of 'first'

        asm!("SUB $2, $0" : "=r"(result) : "0"(first), "r"(second));
        println!("{}", result);
    }
}
```

LLVM`rustc`LLVM。

https://riptutorial.com/zh-CN/rust/topic/6998/

13:

- raw_ptr =pointee* const //
- raw_mut_ptr =mut pointee* mut *type* //
- let deref = * raw_ptr //

- 。
- Rust`&my_object`T`my_object *const T*mut T`
- Box

Examples

```
// Let's take an arbitrary piece of data, a 4-byte integer in this case
let some_data: u32 = 14;

// Create a constant raw pointer pointing to the data above
let data_ptr: *const u32 = &some_data as *const u32;

// Note: creating a raw pointer is totally safe but dereferencing a raw pointer requires an
// unsafe block
unsafe {
    let deref_data: u32 = *data_ptr;
    println!("Dereferenced data: {}", deref_data);
}
```

```
Dereferenced data: 14
```

```
// Let's take a mutable piece of data, a 4-byte integer in this case
let mut some_data: u32 = 14;

// Create a mutable raw pointer pointing to the data above
let data_ptr: *mut u32 = &mut some_data as *mut u32;

// Note: creating a raw pointer is totally safe but dereferencing a raw pointer requires an
// unsafe block
unsafe {
    *data_ptr = 20;
    println!("Dereferenced data: {}", some_data);
}
```

```
Dereferenced data: 20
```

null

Rust。

```
use std::ptr;

// Create a const NULL pointer
let null_ptr: *const u16 = ptr::null();

// Create a mutable NULL pointer
```

```
let mut_null_ptr: *mut u16 = ptr::null_mut();
```

CRust◦

```
// Take a regular string slice
let planet: &str = "Earth";

// Create a constant pointer pointing to our string slice
let planet_ptr: *const &str = &planet as *const &str;

// Create a constant pointer pointing to the pointer
let planet_ptr_ptr: *const *const &str = &planet_ptr as *const *const &str;

// This can go on...
let planet_ptr_ptr_ptr = &planet_ptr_ptr as *const *const *const &str;

unsafe {
    // Direct usage
    println!("The name of our planet is: {}", planet);
    // Single dereference
    println!("The name of our planet is: {}", *planet_ptr);
    // Double dereference
    println!("The name of our planet is: {}", **planet_ptr_ptr);
    // Triple dereference
    println!("The name of our planet is: {}", ***planet_ptr_ptr_ptr);
}
```

```
The name of our planet is: Earth◦
```

Rust◦

```
use std::ptr;

// Create some data, a raw pointer pointing to it and a null pointer
let data: u32 = 42;
let raw_ptr = &data as *const u32;
let null_ptr = ptr::null() as *const u32;

// the {:p} mapping shows pointer values as hexadecimal memory addresses
println!("Data address: {:p}", &data);
println!("Raw pointer address: {:p}", raw_ptr);
println!("Null pointer address: {:p}", null_ptr);
```

```
use std::ptr;

// Create some data, a raw pointer pointing to it and a null pointer
let data: u32 = 42;
let raw_ptr = &data as *const u32;
let null_ptr = ptr::null() as *const u32;

// the {:p} mapping shows pointer values as hexadecimal memory addresses
println!("Data address: {:p}", &data);
println!("Raw pointer address: {:p}", raw_ptr);
println!("Null pointer address: {:p}", null_ptr);
```

https://riptutorial.com/zh-CN/rust/topic/7270/

14:

Examples

`i8` `i16` `i32` `i64` `isize`

`u8` `u16` `u32` `u64` `usize`

`45` ◦ `45u8` `u8` ◦

`isize``usize`◦ 323264

`f32``f64` ◦

`2.0` `f64`

`f32` `f32` `2.0f32` ◦

`bool` `true``false`◦

`char` `'x'` ◦ Unicode3 `'` `'\u{3f}'` `'\u{1d160}'` ◦

https://riptutorial.com/zh-CN/rust/topic/8705/

15:

RustPythonargparse◦ clap◦

- std :: env; //env
- let args = env :: args; //argsArgs◦

Examples

std :: env :: args

std::env::args()◦ ArgsVec ◦

```
use std::env;

fn main() {
    for argument in env::args() {
        if argument == "--help" {
            println!("You passed --help as one of the arguments!");
        }
    }
}
```

Vec

```
use std::env;

fn main() {
    for argument in env::args() {
        if argument == "--help" {
            println!("You passed --help as one of the arguments!");
        }
    }
}
```

```
use std::env;

fn main() {
    for argument in env::args() {
        if argument == "--help" {
            println!("You passed --help as one of the arguments!");
        }
    }
}
```

◦ ◦

std::env::args()◦ clap◦

clap ◦

CLI。

```
clap ◦ cargo rungit push ◦ ◦ --verbose --message "Hello, world"
```

```
extern crate clap;
use clap::{Arg, App, SubCommand};

fn main() {
    let app = App::new("Foo Server")
        .about("Serves foos to the world!")
        .version("v0.1.0")
        .author("Foo (@Example on GitHub)")
        .subcommand(SubCommand::with_name("run")
            .about("Runs the Foo Server")
            .arg(Arg::with_name("debug")
                .short("D")
                .about("Sends debug foos instead of normal foos.")))

    // This parses the command-line arguments for use.
    let matches = app.get_matches();

    // We can get the subcommand used with matches.subcommand(), which
    // returns a tuple of (&str, Option<ArgMatches>) where the &str
    // is the name of the subcommand, and the ArgMatches is an
    // ArgMatches struct:
    // https://docs.rs/clap/2.13.0/clap/struct.ArgMatches.html

    if let ("run", Some(run_matches)) = app.subcommand() {
        println!("Run was used!");
    }
}
```

https://riptutorial.com/zh-CN/rust/topic/7015/

16: FFI

- [linkname =“snappy”] //

  extern {...} //

Examples

libc

`libc` crate' feature gated 'Rust。

```
#![feature(libc)]
extern crate libc;
use libc::pid_t;

#[link(name = "c")]
extern {
    fn getpid() -> pid_t;
}

fn main() {
    let x = unsafe { getpid() };
    println!("Process PID is {}", x);
}
```

FFI https://riptutorial.com/zh-CN/rust/topic/6140/-ffi-

17:

Rust `String` `&str` ◦ Rust◦

Examples

```
fn main() {
    // Statically allocated string slice
    let hello = "Hello world";

    // This is equivalent to the previous one
    let hello_again: &'static str = "Hello world";

    // An empty String
    let mut string = String::new();

    // An empty String with a pre-allocated initial buffer
    let mut capacity = String::with_capacity(10);

    // Add a string slice to a String
    string.push_str("foo");

    // From a string slice to a String
    // Note: Prior to Rust 1.9.0 the to_owned method was faster
    // than to_string. Nowadays, they are equivalent.
    let bar = "foo".to_owned();
    let qux = "foo".to_string();

    // The String::from method is another way to convert a
    // string slice to an owned String.
    let baz = String::from("foo");

    // Coerce a String into &str with &
    let baz: &str = &bar;
}
```

`String::newString::with_capacity◦` ◦ `String::with_capacityString::with_capacity` ◦

```
fn main() {
    let english = "Hello, World!";

    println!("{}", &english[0..5]); // Prints "Hello"
    println!("{}", &english[7..]);  // Prints "World!"
}
```

`&◦` ◦ `println!◦`

```
fn main() {
    let english = "Hello, World!";

    println!("{}", &english[0..5]); // Prints "Hello"
    println!("{}", &english[7..]);  // Prints "World!"
}
```

icelandic[5] ◦

```
let strings = "bananas,apples,pear".split(",");
```

split◦

```
let strings = "bananas,apples,pear".split(",");
```

Iterator::collectVec “”◦

```
let strings = "bananas,apples,pear".split(",");
```

```
// all variables `s` have the type `String`
let s = "hi".to_string();  // Generic way to convert into `String`. This works
                           // for all types that implement `Display`.

let s = "hi".to_owned();   // Clearly states the intend of obtaining an owned object

let s: String = "hi".into();       // Generic conversion, type annotation required
let s: String = From::from("hi");  // in both cases!

let s = String::from("hi");  // Calling the `from` impl explicitly -- the `From`
                             // trait has to be in scope!

let s = format!("hi");       // Using the formatting functionality (this has some
                             // overhead)
```

format!() ◦

\

```
let a = "foobar";
let b = "foo\
         bar";

// `a` and `b` are equal.
assert_eq!(a,b);
```

concat!concat!

```
let a = "foobar";
let b = "foo\
         bar";

// `a` and `b` are equal.
assert_eq!(a,b);
```

https://riptutorial.com/zh-CN/rust/topic/998/

18:

The Rust Programming Language The Book。

Examples

。

```
/// Computes `a + b * c`. If any of the operation overflows, returns `None`.
fn checked_fma(a: u64, b: u64, c: u64) -> Option<u64> {
    let product = match b.checked_mul(c) {
        Some(p) => p,
        None => return None,
    };
    let sum = match a.checked_add(product) {
        Some(s) => s,
        None => return None,
    };
    Some(sum)
}
```

`match`

```
/// Computes `a + b * c`. If any of the operation overflows, returns `None`.
fn checked_fma(a: u64, b: u64, c: u64) -> Option<u64> {
    let product = match b.checked_mul(c) {
        Some(p) => p,
        None => return None,
    };
    let sum = match a.checked_add(product) {
        Some(s) => s,
        None => return None,
    };
    Some(sum)
}
```

`try_opt!(expression)` 3

```
/// Computes `a + b * c`. If any of the operation overflows, returns `None`.
fn checked_fma(a: u64, b: u64, c: u64) -> Option<u64> {
    let product = match b.checked_mul(c) {
        Some(p) => p,
        None => return None,
    };
    let sum = match a.checked_add(product) {
        Some(s) => s,
        None => return None,
    };
    Some(sum)
}
```

`try_opt!`。 - 。

macro_rules!macro_rules!

```
/// Computes `a + b * c`. If any of the operation overflows, returns `None`.
fn checked_fma(a: u64, b: u64, c: u64) -> Option<u64> {
    let product = match b.checked_mul(c) {
        Some(p) => p,
        None => return None,
    };
    let sum = match a.checked_add(product) {
        Some(s) => s,
        None => return None,
    };
    Some(sum)
}
```

。

Rust Playground

HashSet

```
// This example creates a macro `set!` that functions similarly to the built-in
// macro vec!

use std::collections::HashSet;

macro_rules! set {
    ( $( $x:expr ),* ) => {  // Match zero or more comma delimited items
        {
            let mut temp_set = HashSet::new();  // Create a mutable HashSet
            $(
                temp_set.insert($x); // Insert each item matched into the HashSet
            )*
            temp_set // Return the populated HashSet
        }
    };
}

// Usage
let my_set = set![1, 2, 3, 4];
```

```
macro_rules! sum {
    ($base:expr) => { $base };
    ($a:expr, $($rest:expr),+) => { $a + sum!($($rest),+) };
}
```

sum!(1, 2, 3)sum!(1, 2, 3)

```
macro_rules! sum {
    ($base:expr) => { $base };
    ($a:expr, $($rest:expr),+) => { $a + sum!($($rest),+) };
}
```

- 64

```
macro_rules! sum {
    ($base:expr) => { $base };
    ($a:expr, $($rest:expr),+) => { $a + sum!($($rest),+) };
}
```

-
-

64

```
macro_rules! sum {
    ($base:expr) => { $base };
    ($a:expr, $($rest:expr),+) => { $a + sum!($($rest),+) };
}
```

```
/// The `sum` macro may be invoked in two ways:
///
///     sum!(iterator)
///     sum!(1234, iterator)
///
macro_rules! sum {
    ($iter:expr) => {    // This branch handles the `sum!(iterator)` case
        $iter.fold(0, |a, b| a + *b)
    };
//   ^ use `;` to separate each branch
    ($start:expr, $iter:expr) => {  // This branch handles the `sum!(1234, iter)` case
        $iter.fold($start, |a, b| a + *b)
    };
}

fn main() {
    assert_eq!(10, sum!([1, 2, 3, 4].iter()));
    assert_eq!(23, sum!(6, [2, 5, 9, 1].iter()));
}
```

-

`$e:expr` `expr` ◦ `$e`◦ Rust◦

ident		`x` `foo`
path		`std::collection::HashSet` `Vec::new`
ty		`i32` `&T` `Vec<(char, String)>`
expr		`2+2` `f(42)` `if true { 1 } else { 2 }`
pat		`_` `c @ 'a' ... 'z'` `(true, &x)` `Badger { age, .. }`
stmt		`let x = 3` `return 42`
block		`{ foo(); bar(); }` `{ x(); y(); z() }`

item		fn foo() {} struct Bar; use std::io;
meta		cfg!(windows) doc="comment"
tt		+ foo 5 [?!(???)]

```
/// comment#[doc="comment"]。
```

```
macro_rules! declare_const_option_type {
    (
        $(#[$attr:meta])*
        const $name:ident: $ty:ty as optional;
    ) => {
        $(#[$attr])*
        const $name: Option<$ty> = None;
    }
}

declare_const_option_type! {
    /// some doc comment
    const OPT_INT: i32 as optional;
}

// The above will be expanded to:
#[doc="some doc comment"]
const OPT_INT: Option<i32> = None;
```

“”。Rust。

expr stmt	=> , ;
ty path	=> , = | ; : > [{ as where
pat	=> , = | if in
ident block item meta tt	

```
macro_rules! declare_const_option_type {
    (
        $(#[$attr:meta])*
        const $name:ident: $ty:ty as optional;
    ) => {
        $(#[$attr])*
        const $name: Option<$ty> = None;
    }
}

declare_const_option_type! {
    /// some doc comment
    const OPT_INT: i32 as optional;
}
```

```
// The above will be expanded to:
#[doc="some doc comment"]
const OPT_INT: Option<i32> = None;
```

```
    #[macro_export]
//  ^~~~~~~~~~~~~~~ Think of it as `pub` for macros.
    macro_rules! my_macro { (..) => {..} }
```

```
    #[macro_export]
//  ^~~~~~~~~~~~~~~ Think of it as `pub` for macros.
    macro_rules! my_macro { (..) => {..} }
```

。

log_syntax

```
#![feature(log_syntax)]

macro_rules! logged_sum {
    ($base:expr) => {
        { log_syntax!(base = $base); $base }
    };
    ($a:expr, $($rest:expr),+) => {
        { log_syntax!(a = $a, rest = $($rest),+); $a + logged_sum!($($rest),+) }
    };
}

const V: u32 = logged_sum!(1, 2, 3);
```

stdout

a = 1= 2,3
a = 2= 3
base = 3

-

```
#![feature(log_syntax)]

macro_rules! logged_sum {
    ($base:expr) => {
        { log_syntax!(base = $base); $base }
    };
    ($a:expr, $($rest:expr),+) => {
        { log_syntax!(a = $a, rest = $($rest),+); $a + logged_sum!($($rest),+) }
    };
}

const V: u32 = logged_sum!(1, 2, 3);
```

stdout

```
#![feature(log_syntax)]

macro_rules! logged_sum {
    ($base:expr) => {
        { log_syntax!(base = $base); $base }
    };
    ($a:expr, $($rest:expr),+) => {
        { log_syntax!(a = $a, rest = $($rest),+); $a + logged_sum!($($rest),+) }
    };
}

const V: u32 = logged_sum!(1, 2, 3);
```

C`gccclang -E`。

https://riptutorial.com/zh-CN/rust/topic/1031/

19:

- fn function <'a>x'a Type
- struct Struct <'a> {x'a Type}
- enum Enum <'a> {Variant'a Type}
- impl <'a> Struct <'a> {fn x <'a>self - >'a Type {self.x}}
- impl <'a> Trait <'a>
- impl <'a><'a>
- `fn function<F>(f: F) where for<'a> F: FnOnce(&'a Type)`
- `struct Struct<F> where for<'a> F: FnOnce(&'a Type) { x: F }`
- `enum Enum<F> where for<'a> F: FnOnce(&'a Type) { Variant(F) }`
- `impl<F> Struct<F> where for<'a> F: FnOnce(&'a Type) { fn x(&self) -> &F { &self.x } }`

- Rust。 。
- `'static。 &'static str 。`

Examples

```
fn foo<'a>(x: &'a u32) {
    // ...
}
```

`foo'a x'a 。`

```
fn foo<'a>(x: &'a u32) {
    // ...
}
```

。

```
fn foo<'a>(x: &'a u32) {
    // ...
}
```

。

```
fn foo<'a>(x: &'a u32) {
    // ...
}
```

。

```
fn foo<'a>(x: &'a u32) {
    // ...
}
```

```
struct Struct<'a> {
```

```
    x: &'a u32,
}
```

Struct'a x&u32'a◦

Impl

```
impl<'a> Type<'a> {
    fn my_function(&self) -> &'a u32 {
        self.x
    }
}
```

Type'a my_function()'aTypeself.x

```
fn copy_if<F>(slice: &[i32], pred: F) -> Vec<i32>
    where for<'a> F: Fn(&'a i32) -> bool
{
    let mut result = vec![];
    for &element in slice {
        if pred(&element) {
            result.push(element);
        }
    }
    result
}
```

Fni32◦

```
fn copy_if<F>(slice: &[i32], pred: F) -> Vec<i32>
    where for<'a> F: Fn(&'a i32) -> bool
{
    let mut result = vec![];
    for &element in slice {
        if pred(&element) {
            result.push(element);
        }
    }
    result
}
```

```
fn copy_if<F>(slice: &[i32], pred: F) -> Vec<i32>
    where for<'a> F: Fn(&'a i32) -> bool
{
    let mut result = vec![];
    for &element in slice {
        if pred(&element) {
            result.push(element);
        }
    }
    result
}
```

element'a◦

◦ for<'a> ◦

```
fn copy_if<F>(slice: &[i32], pred: F) -> Vec<i32>
    where for<'a> F: Fn(&'a i32) -> bool
{
    let mut result = vec![];
    for &element in slice {
        if pred(&element) {
            result.push(element);
        }
    }
    result
}
```

◦

Fn*◦

◦

https://riptutorial.com/zh-CN/rust/topic/2074/

20:

- loop { *block* } //
- { *block* }
- let *pattern* = *expr* { *block* }
- *expr*{} // exprIntoIterator
- //
- break //
- ' *label* loop { *block* }
- ' *label* while *condition* { *block* }
- ' *label* while let *pattern* = *expr* { *block* }
- ' *label* for *expr* { *block* }*pattern*
- continue' *//label*
- *break'label //label*

Examples

Rust4。。

```
let mut x = 0;
loop {
    if x > 3 { break; }
    println!("{}", x);
    x += 1;
}
```

```
let mut x = 0;
loop {
    if x > 3 { break; }
    println!("{}", x);
    x += 1;
}
```

loopwhile true

while let◦

```
let mut x = 0;
loop {
    if x > 3 { break; }
    println!("{}", x);
    x += 1;
}
```

loopmatch

```
let mut x = 0;
loop {
    if x > 3 { break; }
    println!("{}", x);
    x += 1;
}
```

Rust for“”IntoIterator ◦

```
let mut x = 0;
loop {
    if x > 3 { break; }
    println!("{}", x);
    x += 1;
}
```

while let

```
let mut x = 0;
loop {
    if x > 3 { break; }
    println!("{}", x);
    x += 1;
}
```

0..4IteratorRange ◦ into_iter()for◦ forIntoIterator ◦

BasicsforIntoIterator

```
let vector = vec!["foo", "bar", "baz"]; // vectors implement IntoIterator
for val in vector {
    println!("{}", val);
}
```

```
let vector = vec!["foo", "bar", "baz"]; // vectors implement IntoIterator
for val in vector {
    println!("{}", val);
}
```

vectorfor vector◦ IntoIterator::into_iter self ◦

IntoIterator&Vec<T>&mut Vec<T> &T&mut T vector

```
let vector = vec!["foo", "bar", "baz"]; // vectors implement IntoIterator
for val in vector {
    println!("{}", val);
}
```

val&&str vectorVec<&str> ◦

breakcontinue◦ ◦

break

```
for x in 0..5 {
    if x > 2 { break; }
    println!("{}", x);
}
```

```
for x in 0..5 {
    if x > 2 { break; }
    println!("{}", x);
}
```

---

continue

```
for x in 0..5 {
    if x > 2 { break; }
    println!("{}", x);
}
```

```
for x in 0..5 {
    if x > 2 { break; }
    println!("{}", x);
}
```

---

break◦ breakcontinue◦ 'outer◦

```
for x in 0..5 {
    if x > 2 { break; }
    println!("{}", x);
}
```

```
for x in 0..5 {
    if x > 2 { break; }
    println!("{}", x);
}
```

i > 1 --◦

◦ <>&◦

https://riptutorial.com/zh-CN/rust/topic/955/

21:

`struct`“”。 。

。

- trait Trait {fn method... - > ReturnType; ...}
- trait TraitBound {fn method... - > ReturnType; ...}
- impl Trait for Type {fn method... - > ReturnType {...} ...}
- impl <T> TTBounds {fn method... - > ReturnType {...} ...}

- 。 JavaOO。 Rust。

Examples

```
trait Speak {
    fn speak(&self) -> String;
}
```

```
trait Speak {
    fn speak(&self) -> String;
}
```

。

```
fn generic_speak<T: Speak>(speaker: &T) {
    println!("{0}", speaker.speak());
}

fn main() {
    let person = Person {};
    let dog = Dog {};

    generic_speak(&person);
    generic_speak(&dog);
}
```

Rust`DogPersongeneric_speak`。 **Monomorphization** 。

```
fn generic_speak<T: Speak>(speaker: &T) {
    println!("{0}", speaker.speak());
}

fn main() {
    let person = Person {};
    let dog = Dog {};

    generic_speak(&person);
    generic_speak(&dog);
```

```
}
```

generic_speak vtablespeak()。 。

&SpeakBox<Speak> 。

- 。
- 。

```
trait GetItems {
    type First;
//  ^~~~ defines an associated type.
    type Last: ?Sized;
//          ^~~~~~~~ associated types may be constrained by traits as well
    fn first_item(&self) -> &Self::First;
//                           ^~~~~~~~~~~ use `Self::` to refer to the associated type
    fn last_item(&self) -> &Self::Last;
//                         ^~~~~~~~~~ associated types can be used as function output...
    fn set_first_item(&mut self, item: Self::First);
//                                     ^~~~~~~~~~~  ... input, and anywhere.
}
```

FPGA

```
trait GetItems {
    type First;
//  ^~~~ defines an associated type.
    type Last: ?Sized;
//          ^~~~~~~~ associated types may be constrained by traits as well
    fn first_item(&self) -> &Self::First;
//                           ^~~~~~~~~~~ use `Self::` to refer to the associated type
    fn last_item(&self) -> &Self::Last;
//                         ^~~~~~~~~~ associated types can be used as function output...
    fn set_first_item(&mut self, item: Self::First);
//                                     ^~~~~~~~~~~  ... input, and anywhere.
}
```

TGetItems T::First。

```
trait GetItems {
    type First;
//  ^~~~ defines an associated type.
    type Last: ?Sized;
//          ^~~~~~~~ associated types may be constrained by traits as well
    fn first_item(&self) -> &Self::First;
//                           ^~~~~~~~~~~ use `Self::` to refer to the associated type
    fn last_item(&self) -> &Self::Last;
//                         ^~~~~~~~~~ associated types can be used as function output...
    fn set_first_item(&mut self, item: Self::First);
//                                     ^~~~~~~~~~~  ... input, and anywhere.
```

```
}
```

```
trait GetItems {
    type First;
//  ^~~~ defines an associated type.
    type Last: ?Sized;
//           ^~~~~~~~ associated types may be constrained by traits as well
    fn first_item(&self) -> &Self::First;
//                           ^~~~~~~~~~~ use `Self::` to refer to the associated type
    fn last_item(&self) -> &Self::Last;
//                          ^~~~~~~~~~ associated types can be used as function output...
    fn set_first_item(&mut self, item: Self::First);
//                                     ^~~~~~~~~~~  ... input, and anywhere.
}
```

```
trait GetItems {
    type First;
//  ^~~~ defines an associated type.
    type Last: ?Sized;
//           ^~~~~~~~ associated types may be constrained by traits as well
    fn first_item(&self) -> &Self::First;
//                           ^~~~~~~~~~~ use `Self::` to refer to the associated type
    fn last_item(&self) -> &Self::Last;
//                          ^~~~~~~~~~ associated types can be used as function output...
    fn set_first_item(&mut self, item: Self::First);
//                                     ^~~~~~~~~~~  ... input, and anywhere.
}
```


```
trait Speak {
    fn speak(&self) -> String {
        String::from("Hi.")
    }
}
```

impl◦


```
trait Speak {
    fn speak(&self) -> String {
        String::from("Hi.")
    }
}
```


◦

◦

◦

DerefMutDeref

```
pub trait DerefMut: Deref {
    fn deref_mut(&mut self) -> &mut Self::Target;
```

```
}
```

`DerefMutDerefTarget` ◦

- ……
- `&DerefMut&Deref`

- `'static`
-

◦

Static Dispatch◦

```
fn mammal_speak<T: Person + Dog>(mammal: &T) {
    println!("{0}", mammal.speak());
}

fn main() {
    let person = Person {};
    let dog = Dog {};

    mammal_speak(&person);
    mammal_speak(&dog);
}
```

https://riptutorial.com/zh-CN/rust/topic/1313/

22:

Rust `panic!`。 `Result`panic!。

。 RAIIDrop。 ;RAII。

Examples

Rust`Result<T, E>` `Err(E)`panic! 。

。 Rust。 `pushVec`

```
pub fn push(&mut self, value: T) {
    // This will panic or abort if we would allocate > isize::MAX bytes
    // or if the length increment would overflow for zero-sized types.
    if self.len == self.buf.cap() {
        self.buf.double();
    }
    ...
}
```

Rust。

。 Abort。

```
pub fn push(&mut self, value: T) {
    // This will panic or abort if we would allocate > isize::MAX bytes
    // or if the length increment would overflow for zero-sized types.
    if self.len == self.buf.cap() {
        self.buf.double();
    }
    ...
}
```

`Cargo.toml` 。

```
pub fn push(&mut self, value: T) {
    // This will panic or abort if we would allocate > isize::MAX bytes
    // or if the length increment would overflow for zero-sized types.
    if self.len == self.buf.cap() {
        self.buf.double();
    }
    ...
}
```

https://riptutorial.com/zh-CN/rust/topic/6895/

23:

Rust。 。

- xT = ... // x
- xmut T = ... // x
- let _ =mut foo; //foo
- let _ =foo; //foo
- _ = foo; //foo

- Rust1.0; 20155~。 。

Examples

Rust。 。 。

```
let owned = String::from("hello");
// since we own the value, we may let other variables borrow it
let immutable_borrow1 = &owned;
// as all current borrows are immutable, we can allow many of them
let immutable_borrow2 = &owned;
// in fact, if we have an immutable reference, we are also free to
// duplicate that reference, since we maintain the invariant that
// there are only immutable references
let immutable_borrow3 = &*immutable_borrow2;
```

ERROR

```
let owned = String::from("hello");
// since we own the value, we may let other variables borrow it
let immutable_borrow1 = &owned;
// as all current borrows are immutable, we can allow many of them
let immutable_borrow2 = &owned;
// in fact, if we have an immutable reference, we are also free to
// duplicate that reference, since we maintain the invariant that
// there are only immutable references
let immutable_borrow3 = &*immutable_borrow2;
```

。

```
let owned = String::from("hello");
// since we own the value, we may let other variables borrow it
let immutable_borrow1 = &owned;
// as all current borrows are immutable, we can allow many of them
let immutable_borrow2 = &owned;
// in fact, if we have an immutable reference, we are also free to
// duplicate that reference, since we maintain the invariant that
// there are only immutable references
let immutable_borrow3 = &*immutable_borrow2;
```

Rust 。

```
{
    let x = String::from("hello"); //             +
    // ...                                        :
    let y = String::from("hello"); //      +      |
    // ...                                 :      :
    foo(x) // x is moved                   |      = x's lifetime
    // ...                                 :
} //                                       = y's lifetime
```

```
{
    let x = String::from("hello"); //             +
    // ...                                        :
    let y = String::from("hello"); //      +      |
    // ...                                 :      :
    foo(x) // x is moved                   |      = x's lifetime
    // ...                                 :
} //                                       = y's lifetime
```

。

```
fn foo(x: &String) {
    // foo is only authorized to read x's contents, and to create
    // additional immutable references to it if it so desires.
    let y = *x; // ERROR, cannot move when not owned
    x.push_str("foo"); // ERROR, cannot mutate with immutable reference
    println!("{}", x.len()); // reading OK
    foo(x); // forwarding reference OK
}
```

foo

```
fn foo(x: &String) {
    // foo is only authorized to read x's contents, and to create
    // additional immutable references to it if it so desires.
    let y = *x; // ERROR, cannot move when not owned
    x.push_str("foo"); // ERROR, cannot mutate with immutable reference
    println!("{}", x.len()); // reading OK
    foo(x); // forwarding reference OK
}
```

&&mut 。 foox 。

```
fn foo(x: &String) {
    // foo is only authorized to read x's contents, and to create
    // additional immutable references to it if it so desires.
    let y = *x; // ERROR, cannot move when not owned
    x.push_str("foo"); // ERROR, cannot mutate with immutable reference
    println!("{}", x.len()); // reading OK
    foo(x); // forwarding reference OK
}
```

Rust Copy trait。 Copy。 。 Rust bool usize f64Copy 。

```
let x: isize = 42;
let xr = &x;
```

```
let y = *xr; // OK, because isize is Copy
// both x and y are owned here
```

VecString Copy

```
let x: isize = 42;
let xr = &x;
let y = *xr; // OK, because isize is Copy
// both x and y are owned here
```

https://riptutorial.com/zh-CN/rust/topic/4395/

24:

Rust`std::thread`channelsatomicsParallelism。 。

Examples

```
use std::thread;

fn main() {
    thread::spawn(move || {
        // The main thread will not wait for this thread to finish. That
        // might mean that the next println isn't even executed before the
        // program exits.
        println!("Hello from spawned thread");
    });

    let join_handle = thread::spawn(move || {
        println!("Hello from second spawned thread");
        // To ensure that the program waits for a thread to finish, we must
        // call `join()` on its join handle. It is even possible to send a
        // value to a different thread through the join handle, like the
        // integer 17 in this case:
        17
    });

    println!("Hello from the main thread");

    // The above three printlns can be observed in any order.

    // Block until the second spawned thread has finished.
    match join_handle.join() {
        Ok(x) => println!("Second spawned thread returned {}", x),
        Err(_) => println!("Second spawned thread panicked")
    }
}
```

。 - 0,1...9

```
use std::thread;
use std::sync::mpsc::channel;

fn main() {
    // Create a channel with a sending end (tx) and a receiving end (rx).
    let (tx, rx) = channel();

    // Spawn a new thread, and move the receiving end into the thread.
    let join_handle = thread::spawn(move || {
        // Keep receiving in a loop, until tx is dropped!
        while let Ok(n) = rx.recv() { // Note: `recv()` always blocks
            println!("Received {}", n);
        }
    });

    // Note: using `rx` here would be a compile error, as it has been
    // moved into the spawned thread.
```

```
    // Send some values to the spawned thread. `unwrap()` crashes only if the
    // receiving end was dropped before it could be buffered.
    for i in 0..10 {
        tx.send(i).unwrap(); // Note: `send()` never blocks
    }

    // Drop `tx` so that `rx.recv()` returns an `Err(_)`.
    drop(tx);

    // Wait for the spawned thread to finish.
    join_handle.join().unwrap();
}
```

- HTTPFTP。 - Heisenbugs。。

```
// Session Types aren't part of the standard library, but are part of this crate.
// You'll need to add session_types to your Cargo.toml file.
extern crate session_types;

// For now, it's easiest to just import everything from the library.
use session_types::*;

// First, we describe what our client thread will do. Note that there's no reason
// you have to use a client/server model - it's just convenient for this example.
// This type says that a client will first send a u32, then quit. `Eps` is
// shorthand for "end communication".
// Session Types use two generic parameters to describe the protocol - the first
// for the current communication, and the second for what will happen next.
type Client = Send<u32, Eps>;
// Now, we define what the server will do: it will receive as u32, then quit.
type Server = Recv<u32, Eps>;

// This function is ordinary code to run the client. Notice that it takes
// ownership of a channel, just like other forms of interthread communication -
// but this one about the protocol we just defined.
fn run_client(channel: Chan<(), Client>) {
    let channel = channel.send(42);
    println!("The client just sent the number 42!");
    channel.close();
}

// Now we define some code to run the server. It just accepts a value and prints
// it.
fn run_server(channel: Chan<(), Server>) {
    let (channel, data) = channel.recv();
    println!("The server received some data: {}", data);
    channel.close();
}

fn main() {
    // First, create the channels used for the two threads to talk to each other.
    let (server_channel, client_channel) = session_channel();

    // Start the server on a new thread
    let server_thread = std::thread::spawn(move || {
        run_server(server_channel);
    });

    // Run the client on this thread.
```

```
    run_client(client_channel);

    // Wait for the server to finish.
    server_thread.join().unwrap();
}
```

main。

```
// Session Types aren't part of the standard library, but are part of this crate.
// You'll need to add session_types to your Cargo.toml file.
extern crate session_types;

// For now, it's easiest to just import everything from the library.
use session_types::*;

// First, we describe what our client thread will do. Note that there's no reason
// you have to use a client/server model - it's just convenient for this example.
// This type says that a client will first send a u32, then quit. `Eps` is
// shorthand for "end communication".
// Session Types use two generic parameters to describe the protocol - the first
// for the current communication, and the second for what will happen next.
type Client = Send<u32, Eps>;
// Now, we define what the server will do: it will receive as u32, then quit.
type Server = Recv<u32, Eps>;

// This function is ordinary code to run the client. Notice that it takes
// ownership of a channel, just like other forms of interthread communication -
// but this one about the protocol we just defined.
fn run_client(channel: Chan<(), Client>) {
    let channel = channel.send(42);
    println!("The client just sent the number 42!");
    channel.close();
}

// Now we define some code to run the server. It just accepts a value and prints
// it.
fn run_server(channel: Chan<(), Server>) {
    let (channel, data) = channel.recv();
    println!("The server received some data: {}", data);
    channel.close();
}

fn main() {
    // First, create the channels used for the two threads to talk to each other.
    let (server_channel, client_channel) = session_channel();

    // Start the server on a new thread
    let server_thread = std::thread::spawn(move || {
        run_server(server_channel);
    });

    // Run the client on this thread.
    run_client(client_channel);

    // Wait for the server to finish.
    server_thread.join().unwrap();
}
```

。

recv◦ ◦ SendRecv“” - Recv ◦ Eps◦

- ◦ session_typesclose ◦ - ◦ ◦

SendRecv - Offer RecVar◦ session_types GitHub◦ ◦

◦ /◦ Rust5 **Acquire Release AcqRel** “Acquire-for-loadRelease-for-store”;**SeqCst** ◦ “”“”“”“”◦

```
use std::cell::UnsafeCell;
use std::sync::atomic::{AtomicUsize, Ordering};
use std::sync::{Arc, Barrier};
use std::thread;

struct UsizePair {
    atom: AtomicUsize,
    norm: UnsafeCell<usize>,
}

// UnsafeCell is not thread-safe. So manually mark our UsizePair to be Sync.
// (Effectively telling the compiler "I'll take care of it!")
unsafe impl Sync for UsizePair {}

static NTHREADS: usize = 8;
static NITERS: usize = 1000000;

fn main() {
    let upair = Arc::new(UsizePair::new(0));

    // Barrier is a counter-like synchronization structure (not to be confused
    // with a memory barrier). It blocks on a `wait` call until a fixed number
    // of `wait` calls are made from various threads (like waiting for all
    // players to get to the starting line before firing the starter pistol).
    let barrier = Arc::new(Barrier::new(NTHREADS + 1));

    let mut children = vec![];

    for _ in 0..NTHREADS {
        let upair = upair.clone();
        let barrier = barrier.clone();
        children.push(thread::spawn(move || {
            barrier.wait();

            let mut v = 0;
            while v < NITERS - 1 {
                // Read both members `atom` and `norm`, and check whether `atom`
                // contains a newer value than `norm`. See `UsizePair` impl for
                // details.
                let (atom, norm) = upair.get();
                if atom > norm {
                    // If `Acquire`-`Release` ordering is used in `get` and
                    // `set`, then this statement will never be reached.
                    println!("Reordered! {} > {}", atom, norm);
                }
                v = atom;
            }
        }));
    }

    barrier.wait();
```

```
    for v in 1..NITERS {
        // Update both members `atom` and `norm` to value `v`. See the impl for
        // details.
        upair.set(v);
    }

    for child in children {
        let _ = child.join();
    }
}

impl UsizePair {
    pub fn new(v: usize) -> UsizePair {
        UsizePair {
            atom: AtomicUsize::new(v),
            norm: UnsafeCell::new(v),
        }
    }

    pub fn get(&self) -> (usize, usize) {
        let atom = self.atom.load(Ordering::Relaxed); //Ordering::Acquire

        // If the above load operation is performed with `Acquire` ordering,
        // then all writes before the corresponding `Release` store is
        // guaranteed to be visible below.

        let norm = unsafe { *self.norm.get() };
        (atom, norm)
    }

    pub fn set(&self, v: usize) {
        unsafe { *self.norm.get() = v };

        // If the below store operation is performed with `Release` ordering,
        // then the write to `norm` above is guaranteed to be visible to all
        // threads that "loads `atom` with `Acquire` ordering and sees the same
        // value that was stored below". However, no guarantees are provided as
        // to when other readers will witness the below store, and consequently
        // the above write. On the other hand, there is also no guarantee that
        // these two values will be in sync for readers. Even if another thread
        // sees the same value that was stored below, it may actually see a
        // "later" value in `norm` than what was written above. That is, there
        // is no restriction on visibility into the future.

        self.atom.store(v, Ordering::Relaxed); //Ordering::Release
    }
}
```

x86。 。 Wikipedia。

RwLocks。

RwLock。

```
use std::time::Duration;
use std::thread;
use std::thread::sleep;
use std::sync::{Arc, RwLock };
```

```
fn main() {
    // Create an u32 with an inital value of 0
    let initial_value = 0u32;

    // Move the initial value into the read-write lock which is wrapped into an atomic
reference
    // counter in order to allow safe sharing.
    let rw_lock = Arc::new(RwLock::new(initial_value));

    // Create a clone for each thread
    let producer_lock = rw_lock.clone();
    let consumer_id_lock = rw_lock.clone();
    let consumer_square_lock = rw_lock.clone();

    let producer_thread = thread::spawn(move || {
        loop {
            // write() blocks this thread until write-exclusive access can be acquired and
retuns an
            // RAII guard upon completion
            if let Ok(mut write_guard) = producer_lock.write() {
                // the returned write_guard implements `Deref` giving us easy access to the
target value
                *write_guard += 1;

                println!("Updated value: {}", *write_guard);
            }

            // ^
            // |   when the RAII guard goes out of the scope, write access will be dropped,
allowing
            // +~  other threads access the lock

            sleep(Duration::from_millis(1000));
        }
    });

    // A reader thread that prints the current value to the screen
    let consumer_id_thread = thread::spawn(move || {
        loop {
            // read() will only block when `producer_thread` is holding a write lock
            if let Ok(read_guard) = consumer_id_lock.read() {
                // the returned read_guard also implements `Deref`
                println!("Read value: {}", *read_guard);
            }

            sleep(Duration::from_millis(500));
        }
    });

    // A second reader thread is printing the squared value to the screen. Note that readers
don't
    // block each other so `consumer_square_thread` can run simultaneously with
`consumer_id_lock`.
    let consumer_square_thread = thread::spawn(move || {
        loop {
            if let Ok(lock) = consumer_square_lock.read() {
                let value = *lock;
                println!("Read value squared: {}", value * value);
            }
```

```
            sleep(Duration::from_millis(750));
        }
    });

    let _ = producer_thread.join();
    let _ = consumer_id_thread.join();
    let _ = consumer_square_thread.join();
}
```

```
use std::time::Duration;
use std::thread;
use std::thread::sleep;
use std::sync::{Arc, RwLock };

fn main() {
    // Create an u32 with an inital value of 0
    let initial_value = 0u32;

    // Move the initial value into the read-write lock which is wrapped into an atomic
reference
    // counter in order to allow safe sharing.
    let rw_lock = Arc::new(RwLock::new(initial_value));

    // Create a clone for each thread
    let producer_lock = rw_lock.clone();
    let consumer_id_lock = rw_lock.clone();
    let consumer_square_lock = rw_lock.clone();

    let producer_thread = thread::spawn(move || {
        loop {
            // write() blocks this thread until write-exclusive access can be acquired and
retuns an
            // RAII guard upon completion
            if let Ok(mut write_guard) = producer_lock.write() {
                // the returned write_guard implements `Deref` giving us easy access to the
target value
                *write_guard += 1;

                println!("Updated value: {}", *write_guard);
            }

            // ^
            // |   when the RAII guard goes out of the scope, write access will be dropped,
allowing
            // +~  other threads access the lock

            sleep(Duration::from_millis(1000));
        }
    });

    // A reader thread that prints the current value to the screen
    let consumer_id_thread = thread::spawn(move || {
        loop {
            // read() will only block when `producer_thread` is holding a write lock
            if let Ok(read_guard) = consumer_id_lock.read() {
                // the returned read_guard also implements `Deref`
                println!("Read value: {}", *read_guard);
            }

            sleep(Duration::from_millis(500));
```

```
        }
    });

    // A second reader thread is printing the squared value to the screen. Note that readers
don't
    // block each other so `consumer_square_thread` can run simultaneously with
`consumer_id_lock`.
    let consumer_square_thread = thread::spawn(move || {
        loop {
            if let Ok(lock) = consumer_square_lock.read() {
                let value = *lock;
                println!("Read value squared: {}", value * value);
            }

            sleep(Duration::from_millis(750));
        }
    });

    let _ = producer_thread.join();
    let _ = consumer_id_thread.join();
    let _ = consumer_square_thread.join();
}
```

https://riptutorial.com/zh-CN/rust/topic/1222/

25:

Examples

◦

◦ [T; N]T N◦

[4u64, 5, 6][u64; 3] ◦ 56u64◦

```
fn main() {
    // Arrays have a fixed size.
    // All elements are of the same type.
    let array = [1, 2, 3, 4, 5];

    // Create an array of 20 elements where all elements are the same.
    // The size should be a compile-time constant.
    let ones = [1; 20];

    // Get the length of an array.
    println!("Length of ones: {}", ones.len());

    // Access an element of an array.
    // Indexing starts at 0.
    println!("Second element of array: {}", array[1]);

    // Run-time bounds-check.
    // This panics with 'index out of bounds: the len is 5 but the index is 5'.
    println!("Non existant element of array: {}", array[5]);
}
```

Rust23121 ◦

RustRFC1657 ◦ ◦ 32◦ ◦

◦

◦

```
fn main() {
    // Create a mutable empty vector
    let mut vector = Vec::new();

    vector.push(20);
    vector.insert(0, 10); // insert at the beginning
```

```
    println!("Second element of vector: {}", vector[1]); // 20

    // Create a vector using the `vec!` macro
    let till_five = vec![1, 2, 3, 4, 5];

    // Create a vector of 20 elements where all elements are the same.
    let ones = vec![1; 20];

    // Get the length of a vector.
    println!("Length of ones: {}", ones.len());

    // Run-time bounds-check.
    // This panics with 'index out of bounds: the len is 5 but the index is 5'.
    println!("Non existant element of array: {}", till_five[5]);
}
```

[T] T◦

◦ *str* ◦

◦ ◦ *strString [T]Vec<T>* ◦

```
fn main() {
    let vector = vec![1, 2, 3, 4, 5, 6, 7, 8];
    let slice = &vector[3..6];
    println!("length of slice: {}", slice.len()); // 3
    println!("slice: {:?}", slice); // [4, 5, 6]
}
```

https://riptutorial.com/zh-CN/rust/topic/5004/-

26: I / O.

Examples

```
use std::fs::File;
use std::io::Read;

fn main() {
    let filename = "src/main.rs";
    // Open the file in read-only mode.
    match File::open(filename) {
        // The file is open (no error).
        Ok(mut file) => {
            let mut content = String::new();

            // Read all the file content into a variable (ignoring the result of the
operation).
            file.read_to_string(&mut content).unwrap();

            println!("{}", content);

            // The file is automatically closed when is goes out of scope.
        },
        // Error handling.
        Err(error) => {
            println!("Error opening file {}: {}", filename, error);
        },
    }
}
```

```
use std::fs::File;
use std::io::{BufRead, BufReader};

fn main() {
    let filename = "src/main.rs";
    // Open the file in read-only mode (ignoring errors).
    let file = File::open(filename).unwrap();
    let reader = BufReader::new(file);

    // Read the file line by line using the lines() iterator from std::io::BufRead.
    for (index, line) in reader.lines().enumerate() {
        let line = line.unwrap(); // Ignore errors.
        // Show the line and its number.
        println!("{}. {}", index + 1, line);
    }
}
```

```
use std::env;
use std::fs::File;
use std::io::Write;

fn main() {
    // Create a temporary file.
    let temp_directory = env::temp_dir();
    let temp_file = temp_directory.join("file");
```

```
    // Open a file in write-only (ignoring errors).
    // This creates the file if it does not exist (and empty the file if it exists).
    let mut file = File::create(temp_file).unwrap();

    // Write a &str in the file (ignoring the result).
    writeln!(&mut file, "Hello World!").unwrap();

    // Write a byte string.
    file.write(b"Bytes\n").unwrap();
}
```

Vec

```
use std::fs::File;
use std::io::Read;

fn read_a_file() -> std::io::Result<Vec<u8>> {
    let mut file = try!(File::open("example.data"));

    let mut data = Vec::new();
    try!(file.read_to_end(&mut data));

    return Ok(data);
}
```

`std::io::Result<T>Result<T, std::io::Error>`◦

`try!()`◦

`read_to_end()std::io::Read` trait`use` d◦

`read_to_end()`◦ ◦

I / O. https://riptutorial.com/zh-CN/rust/topic/1307/i---o-

27:

Rust。 lint。

- ///
- //
- cargo doc。
- cargo doc --open。
- cargo doc -p CRATE。
- cargo doc --no-deps。
- 。

“Rust Book”。

-
-
-
-
-
-

Examples

`missing_docs`/。 `lib.rs`

```
#![warn(missing_docs)]
```

lint

```
#![warn(missing_docs)]
```

`missing_docs`

```
#![warn(missing_docs)]
```

。

Rust。 。

```
mod foo {
    //! Inner documentation comments go *inside* an item (e.g. a module or a
```

```
    //! struct). They use the comment syntax //! and must go at the top of the
    //! enclosing item.
    struct Bar {
        pub baz: i64
        //! This is invalid. Inner comments must go at the top of the struct,
        //! and must not be placed after fields.
    }
}
```

```
mod foo {
    //! Inner documentation comments go *inside* an item (e.g. a module or a
    //! struct). They use the comment syntax //! and must go at the top of the
    //! enclosing item.
    struct Bar {
        pub baz: i64
        //! This is invalid. Inner comments must go at the top of the struct,
        //! and must not be placed after fields.
    }
}
```

```
/// In documentation comments, you may use **Markdown**.
/// This includes `backticks` for code, *italics* and **bold**.
/// You can add headers in your documentation, like this:
/// # Notes
/// `Foo` is unsuitable for snafucating. Use `Bar` instead.
struct Foo {
    ...
}
```

```
/// In documentation comments, you may use **Markdown**.
/// This includes `backticks` for code, *italics* and **bold**.
/// You can add headers in your documentation, like this:
/// # Notes
/// `Foo` is unsuitable for snafucating. Use `Bar` instead.
struct Foo {
    ...
}
```


cargo test◦ “”◦

```
extern crate mycrate;
```

```
/// ```
/// use mycrate::foo::Bar;
/// ```
```

no_run

```
/// ```
/// use mycrate::foo::Bar;
/// ```
```

```
/// ```
/// use mycrate::foo::Bar;
```

```
/// ```
```

https://riptutorial.com/zh-CN/rust/topic/4865/

28: IO

-rsRust。

FutureStream 。

Examples

oneshot

Future。 futures::sync::oneshotfutures::oneshot

```
extern crate futures;

use std::thread;
use futures::Future;

fn expensive_computation() -> u32 {
    // ...
    200
}

fn main() {
    // The oneshot function returns a tuple of a Sender and a Receiver.
    let (tx, rx) = futures::oneshot();

    thread::spawn(move || {
        // The complete method resolves a values.
        tx.complete(expensive_computation());
    });

    // The map method applies a function to a value, when it is resolved.
    let rx = rx.map(|x| {
        println!("{}", x);
    });

    // The wait method blocks current thread until the value is resolved.
    rx.wait().unwrap();
}
```

IO https://riptutorial.com/zh-CN/rust/topic/8595/io

29:

- mod *modname* ; //*modname* .rs*modname* /mod.rs
- mod *modname* { *block* }

Examples

```
- example.rs (root of our modules tree, generally named lib.rs or main.rs when using Cargo)
- first.rs
- second/
  - mod.rs
  - sub.rs
```

```
- example.rs (root of our modules tree, generally named lib.rs or main.rs when using Cargo)
- first.rs
- second/
  - mod.rs
  - sub.rs
```

`example.rs`

```
- example.rs (root of our modules tree, generally named lib.rs or main.rs when using Cargo)
- first.rs
- second/
  - mod.rs
  - sub.rs
```

`secondexample.rsexample first first.rs`

`second/mod.rs`

```
- example.rs (root of our modules tree, generally named lib.rs or main.rs when using Cargo)
- first.rs
- second/
  - mod.rs
  - sub.rs
```

[path]

Rust`#[path]`。 。

```
#[path="../path/to/module.rs"]
mod module;
```

`use`

`use`use。

`use`crate root。 。

```
use std::fs::File;
```

◦ std::fs::File::open()Rust◦

```
use std::fs::File;
```

std::…

```
use std::fs::File;
```

selfsuperuse

```
use std::fs::File;
```

use◦ super

```
fn x() -> u8 {
    5
}

mod example {
    use super::x;

    fn foo() {
        println!("{}", x());
    }
}
```

super""super◦

```
yourproject/
    Cargo.lock
    Cargo.toml
    src/
        main.rs
        writer.rs
```

main.rs

```
yourproject/
    Cargo.lock
    Cargo.toml
    src/
        main.rs
        writer.rs
```

writer.rs

```
yourproject/
    Cargo.lock
    Cargo.toml
    src/
```

```
        main.rs
        writer.rs
```

◦

01.

```
fn main() {
  greet();
}

fn greet() {
  println!("Hello, world!");
}
```

02. -

```
fn main() {
  greet();
}

fn greet() {
  println!("Hello, world!");
}
```

03. -

mod◦ ◦

```
fn main() {
  greet();
}

fn greet() {
  println!("Hello, world!");
}
```

```
fn main() {
  greet();
}

fn greet() {
  println!("Hello, world!");
}
```

mod◦

```
fn main() {
  greet();
}

fn greet() {
  println!("Hello, world!");
}
```

```
fn main() {
  greet();
}

fn greet() {
  println!("Hello, world!");
}
```

04. -

◦ mod.rs◦ ◦

```
fn main() {
  greet();
}

fn greet() {
  println!("Hello, world!");
}
```

```
fn main() {
  greet();
}

fn greet() {
  println!("Hello, world!");
}
```

```
fn main() {
  greet();
}

fn greet() {
  println!("Hello, world!");
}
```

```
fn main() {
  greet();
}

fn greet() {
  println!("Hello, world!");
}
```

```
fn main() {
  greet();
}

fn greet() {
  println!("Hello, world!");
}
```

05. - `self`

```
fn main() {
```

```
  greet();
}

fn greet() {
  println!("Hello, world!");
}
```

06. - `super`

1.

```
fn main() {
  greet();
}

fn greet() {
  println!("Hello, world!");
}
```

2. /

```
fn main() {
  greet();
}

fn greet() {
  println!("Hello, world!");
}
```

07. - `use`

1.

```
fn main() {
  greet();
}

fn greet() {
  println!("Hello, world!");
}
```

2. crate

```
fn main() {
  greet();
}

fn greet() {
  println!("Hello, world!");
}
```

https://riptutorial.com/zh-CN/rust/topic/2528/

30:

- _ //[1]
- *ident* //*ident*[1]
- *ident* @ *pat* //
- ref *ident* // [1]
- ref mut *ident* // [1]
- *pat* //*pat*[1]
- mut *pat* //[1]
- *CONST* //
- *Struct* { *field1* *field2* } //[1]
- *EnumVariant* //
- *EnumVariant* *pat1* *pat2* //
- *EnumVariant* *pat1* *pat2* .. *patn* //
- *pat1* *pat2* //[1]
- *pat1* *pat2* .. *patn* //[1]
- *lit* //charnumeric typesbooleanstring
- *pat1* ... *pat2* //charnumeric

field_namefield_name : *pattern* ◦

```
let Point { x, y } = p;
// equivalent to
let Point { x: x, y: y } = p;

let Point { ref x, ref y } = p;
// equivalent to
let Point { x: ref x, y: ref y } = p;
```

1

Examples

@

```
struct Badger {
    pub age: u8
}

fn main() {
    // Let's create a Badger instances
    let badger_john = Badger { age: 8 };

    // Now try to find out what John's favourite activity is, based on his age
    match badger_john.age {
        // we can bind value ranges to variables and use them in the matched branches
        baby_age @ 0...1 => println!("John is {} years old, he sleeps a lot", baby_age),
        young_age @ 2...4 => println!("John is {} years old, he plays all day", young_age),
```

```
        adult_age @ 5...10 => println!("John is {} years old, he eats honey most of the time",
adult_age),
        old_age => println!("John is {} years old, he mostly reads newspapers", old_age),
    }
}
```

```
struct Badger {
    pub age: u8
}

fn main() {
    // Let's create a Badger instances
    let badger_john = Badger { age: 8 };

    // Now try to find out what John's favourite activity is, based on his age
    match badger_john.age {
        // we can bind value ranges to variables and use them in the matched branches
        baby_age @ 0...1 => println!("John is {} years old, he sleeps a lot", baby_age),
        young_age @ 2...4 => println!("John is {} years old, he plays all day", young_age),
        adult_age @ 5...10 => println!("John is {} years old, he eats honey most of the time",
adult_age),
        old_age => println!("John is {} years old, he mostly reads newspapers", old_age),
    }
}
```

```
// Create a boolean value
let a = true;

// The following expression will try and find a pattern for our value starting with
// the topmost pattern.
// This is an exhaustive match expression because it checks for every possible value
match a {
  true => println!("a is true"),
  false => println!("a is false")
}
```

```
// Create a boolean value
let a = true;

// The following expression will try and find a pattern for our value starting with
// the topmost pattern.
// This is an exhaustive match expression because it checks for every possible value
match a {
  true => println!("a is true"),
  false => println!("a is false")
}
```

_|

```
// Create a boolean value
let a = true;

// The following expression will try and find a pattern for our value starting with
// the topmost pattern.
// This is an exhaustive match expression because it checks for every possible value
match a {
```

```
  true => println!("a is true"),
  false => println!("a is false")
}
```

```
// Create a boolean value
let a = true;

// The following expression will try and find a pattern for our value starting with
// the topmost pattern.
// This is an exhaustive match expression because it checks for every possible value
match a {
  true => println!("a is true"),
  false => println!("a is false")
}
```

|

```
enum Colour {
    Red,
    Green,
    Blue,
    Cyan,
    Magenta,
    Yellow,
    Black
}

enum ColourModel {
    RGB,
    CMYK
}

// let's take an example colour
let colour = Colour::Red;

let model = match colour {
    // check if colour is any of the RGB colours
    Colour::Red | Colour::Green | Colour::Blue => ColourModel::RGB,
    // otherwise select CMYK
    _ => ColourModel::CMYK,
};
```

`if` guards

```
// Let's imagine a simplistic web app with the following pages:
enum Page {
  Login,
  Logout,
  About,
  Admin
}

// We are authenticated
let is_authenticated = true;

// But we aren't admins
let is_admin = false;
```

```
let accessed_page = Page::Admin;

match accessed_page {
    // Login is available for not yet authenticated users
    Page::Login if !is_authenticated => println!("Please provide a username and a password"),

    // Logout is available for authenticated users
    Page::Logout if is_authenticated => println!("Good bye"),

    // About is a public page, anyone can access it
    Page::About => println!("About us"),

    // But the Admin page is restricted to administators
    Page::Admin if is_admin => println!("Welcome, dear administrator"),

    // For every other request, we display an error message
    _ => println!("Not available")
}
```

“”。

/

---

if let

matchif。

```
if let Some(x) = option {
    do_something(x);
}
```

```
if let Some(x) = option {
    do_something(x);
}
```

---

else。

```
if let Some(x) = option {
    do_something(x);
}
```

```
if let Some(x) = option {
    do_something(x);
}
```

---

while let

while。

```
if let Some(x) = option {
```

```
    do_something(x);
}
```

H+e+l+l+o+,+ +w+o+r+l+d+!+ ◦

loop {}match

```
if let Some(x) = option {
    do_something(x);
}
```

◦

```
struct Token {
  pub id: u32
}

struct User {
  pub token: Option<Token>
}


fn main() {
    // Create a user with an arbitrary token
    let user = User { token: Some(Token { id: 3 }) };

    // Let's borrow user by getting a reference to it
    let user_ref = &user;

    // This match expression would not compile saying "cannot move out of borrowed
    // content" because user_ref is a borrowed value but token expects an owned value.
    match user_ref {
        &User { token } => println!("User token exists? {}", token.is_some())
    }

    // By adding 'ref' to our pattern we instruct the compiler to give us a reference
    // instead of an owned value.
    match user_ref {
        &User { ref token } => println!("User token exists? {}", token.is_some())
    }

    // We can also combine ref with destructuring
    match user_ref {
        // 'ref' will allow us to access the token inside of the Option by reference
        &User { token: Some(ref user_token) } => println!("Token value: {}", user_token.id ),
        &User { token: None } => println!("There was no token assigned to the user" )
    }

    // References can be mutable too, let's create another user to demonstrate this
    let mut other_user = User { token: Some(Token { id: 4 }) };

    // Take a mutable reference to the user
    let other_user_ref_mut = &mut other_user;

    match other_user_ref_mut {
        // 'ref mut' gets us a mutable reference allowing us to change the contained value
directly.
        &mut User { token: Some(ref mut user_token) } => {
```

```
            user_token.id = 5;
            println!("New token value: {}", user_token.id )
        },
        &mut User { token: None } => println!("There was no token assigned to the user" )
    }
}
```

```
struct Token {
  pub id: u32
}

struct User {
  pub token: Option<Token>
}


fn main() {
    // Create a user with an arbitrary token
    let user = User { token: Some(Token { id: 3 }) };

    // Let's borrow user by getting a reference to it
    let user_ref = &user;

    // This match expression would not compile saying "cannot move out of borrowed
    // content" because user_ref is a borrowed value but token expects an owned value.
    match user_ref {
        &User { token } => println!("User token exists? {}", token.is_some())
    }

    // By adding 'ref' to our pattern we instruct the compiler to give us a reference
    // instead of an owned value.
    match user_ref {
        &User { ref token } => println!("User token exists? {}", token.is_some())
    }

    // We can also combine ref with destructuring
    match user_ref {
        // 'ref' will allow us to access the token inside of the Option by reference
        &User { token: Some(ref user_token) } => println!("Token value: {}", user_token.id ),
        &User { token: None } => println!("There was no token assigned to the user" )
    }

    // References can be mutable too, let's create another user to demonstrate this
    let mut other_user = User { token: Some(Token { id: 4 }) };

    // Take a mutable reference to the user
    let other_user_ref_mut = &mut other_user;

    match other_user_ref_mut {
        // 'ref mut' gets us a mutable reference allowing us to change the contained value
directly.
        &mut User { token: Some(ref mut user_token) } => {
            user_token.id = 5;
            println!("New token value: {}", user_token.id )
        },
        &mut User { token: None } => println!("There was no token assigned to the user" )
    }
}
```

https://riptutorial.com/zh-CN/rust/topic/1188/

31:

Rust/regexrust-lang-nursery ◦ regex◦

Examples

regexregexCargo.toml

```
[dependencies]
regex = "0.1"
```

regexregex::Regex

```
[dependencies]
regex = "0.1"
```

```
extern crate regex;
use regex::Regex;

fn main() {
    let rg = Regex::new(r"was (\d+)").unwrap();
    // Regex::captures returns Option<Captures>,
    // first element is the full match and others
    // are capture groups
    match rg.captures("The year was 2016") {
        // Access captures groups via Captures::at
        // Prints Some("2016")
        Some(x) => println!("{:?}", x.at(1)),
        None    => unreachable!()
    }

    // Regex::captures also supports named capture groups
    let rg_w_named = Regex::new(r"was (?P<year>\d+)").unwrap();
    match rg_w_named.captures("The year was 2016") {
        // Named capures groups are accessed via Captures::name
        // Prints Some("2016")
        Some(x) => println!("{:?}", x.name("year")),
        None    => unreachable!()
    }

}
```

```
extern crate regex;
use regex::Regex;

fn main() {
    let rg = Regex::new(r"(\d+)").unwrap();

    // Regex::replace replaces first match
    // from it's first argument with the second argument
    // => Some string with numbers (not really)
    rg.replace("Some string with numbers 123", "(not really)");
```

```
    // Capture groups can be accesed via $number
    // => Some string with numbers (which are 123)
    rg.replace("Some string with numbers 123", "(which are $1)");

    let rg = Regex::new(r"(?P<num>\d+)").unwrap();

    // Named capture groups can be accesed via $name
    // => Some string with numbers (which are 123)
    rg.replace("Some string with numbers 123", "(which are $num)");

    // Regex::replace_all replaces all the matches, not only the first
    // => Some string with numbers (not really) (not really)
    rg.replace_all("Some string with numbers 123 321", "(not really)");
}
```

https://riptutorial.com/zh-CN/rust/topic/7184/

32:

Examples

```
// Generic types are declared using the <T> annotation

struct GenericType<T> {
    pub item: T
}

enum QualityChecked<T> {
    Excellent(T),
    Good(T),
    // enum fields can be generics too
    Mediocre { product: T }
}
```

```
// explicit type declaration
let some_value: Option<u32> = Some(13);

// implicit type declaration
let some_other_value = Some(66);
```

◦ Result

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

```
// Only accept T and U generic types that also implement Debug
fn print_objects<T: Debug, U: Debug>(a: T, b: U) {
    println!("A: {:?} B: {:?}", a, b);
}

print_objects(13, 44);
// or annotated explicitly
print_objects::<usize, u16>(13, 44);
```

◦ std::ops::Add trait◦ where T: std::ops::Add<u32,Output=U>Add Tu32 U ◦


```
// Only accept T and U generic types that also implement Debug
fn print_objects<T: Debug, U: Debug>(a: T, b: U) {
    println!("A: {:?} B: {:?}", a, b);
}

print_objects(13, 44);
// or annotated explicitly
print_objects::<usize, u16>(13, 44);
```

Sized◦ ?Sized◦

◦

```
fn convert_values<T, U>(input_value: T) -> Result<U, String> {
  // Try and convert the value.
  // Actual code will require bounds on the types T, U to be able to do something with them.
}
```


```
fn convert_values<T, U>(input_value: T) -> Result<U, String> {
  // Try and convert the value.
  // Actual code will require bounds on the types T, U to be able to do something with them.
}
```


https://riptutorial.com/zh-CN/rust/topic/1801/

33:

Examples

```
fn to_test(output: bool) -> bool {
    output
}

#[cfg(test)] // The module is only compiled when testing.
mod test {
    use super::to_test;

    // This function is a test function. It will be executed and
    // the test will succeed if the function exits cleanly.
    #[test]
    fn test_to_test_ok() {
        assert_eq!(to_test(true), true);
    }

    // That test on the other hand will only succeed when the function
    // panics.
    #[test]
    #[should_panic]
    fn test_to_test_fail() {
        assert_eq!(to_test(true), false);
    }
}
```

cargo test ◦

lib.rs

```
pub fn to_test(output: bool) -> bool {
    output
}
```

tests/◦ tests/integration_test.rs

```
pub fn to_test(output: bool) -> bool {
    output
}
```

◦ benches/cargo bench ◦

llogiq.github.io

```
extern crate test;
extern crate rand;

use test::Bencher;
use rand::Rng;
use std::mem::replace;
```

```
#[bench]
fn empty(b: &mut Bencher) {
    b.iter(|| 1)
}

#[bench]
fn setup_random_hashmap(b: &mut Bencher) {
    let mut val : u32 = 0;
    let mut rng = rand::IsaacRng::new_unseeded();
    let mut map = std::collections::HashMap::new();

    b.iter(|| { map.insert(rng.gen::<u8>() as usize, val); val += 1; })
}

#[bench]
fn setup_random_vecmap(b: &mut Bencher) {
    let mut val : u32 = 0;
    let mut rng = rand::IsaacRng::new_unseeded();
    let mut map = std::collections::VecMap::new();

    b.iter(|| { map.insert((rng.gen::<u8>()) as usize, val); val += 1; })
}

#[bench]
fn setup_random_vecmap_cap(b: &mut Bencher) {
    let mut val : u32 = 0;
    let mut rng = rand::IsaacRng::new_unseeded();
    let mut map = std::collections::VecMap::with_capacity(256);

    b.iter(|| { map.insert((rng.gen::<u8>()) as usize, val); val += 1; })
}
```

https://riptutorial.com/zh-CN/rust/topic/961/

34:

Rustrustacean

Examples

Rust`Box::new`Box。

```
let boxed_int: Box<i32> = Box::new(1);
```

Boxes`Deref<Target=T>` 。

```
let boxed_vec = Box::new(vec![1, 2, 3]);
println!("{}", boxed_vec.get(0));
```

。

```
let boxed_vec = Box::new(vec![1, 2, 3]);
println!("{}", boxed_vec.get(0));
```

BoxRust。

```
// This gives an error!
enum List {
    Nil,
    Cons(i32, List)
}
```

Box。

```
// This gives an error!
enum List {
    Nil,
    Cons(i32, List)
}
```

BoxTRustList。

https://riptutorial.com/zh-CN/rust/topic/9341/

35:

- [associated_consts]
- const IDi32;

- 29646

Examples

```
// Must enable the feature to use associated constants
#![feature(associated_consts)]

use std::mem;

// Associated constants can be used to add constant attributes to types
trait Foo {
    const ID: i32;
}

// All implementations of Foo must define associated constants
// unless a default value is supplied in the definition.
impl Foo for i32 {
    const ID: i32 = 1;
}

struct Bar;

// Associated constants don't have to be bound to a trait to be defined
impl Bar {
    const BAZ: u32 = 5;
}

fn main() {
    assert_eq!(1, i32::ID);

    // The defined constant value is only stored once, so the size of
    // instances of the defined types doesn't include the constants.
    assert_eq!(4, mem::size_of::<i32>());
    assert_eq!(0, mem::size_of::<Bar>());
}
```

https://riptutorial.com/zh-CN/rust/topic/7042/

36:

- struct Foo {field1Type1field2Type2}
- let foo = Foo {field1Type1 :: newfield2Type2 :: new};
- struct BarType1Type2; //
- let _ = BarType1 :: newType2 :: new;
- Baz; //
- _ = Baz;
- Foo {field1..} = foo; //field1
- Foo {field1x..} = foo; //field1x
- foo2 = Foo {field1Type1 :: new.. foo}; //
- impl Foo {fn fiddleself{}} //Foo
- impl Foo {fn tweakmut self{}} //Foo
- impl Foo {fn doubleself{}} //Foo
- impl Foo {fn new{}} //Foo

Examples

Rust`structstruct`。

```
struct Foo {
    my_bool: bool,
    my_num: isize,
    my_string: String,
}
```

`struct my_bool my_nummy_string bool isizeString`。

Rust`struct`

```
struct Foo {
    my_bool: bool,
    my_num: isize,
    my_string: String,
}
```

`Bar bool isizeString` 。 *newtype* “”。 `type`; `Bar`。

`struct`

```
struct Foo {
    my_bool: bool,
    my_num: isize,
    my_string: String,
}
```

。 。

Rust`struct` - ◦ `pub`◦ `struct`◦ `structpub`

```
struct Foo {
    my_bool: bool,
    my_num: isize,
    my_string: String,
}
```

`struct`

```
struct Foo {
    my_bool: bool,
    my_num: isize,
    my_string: String,
}
struct Bar (bool, isize, String);
struct Baz;
```

```
struct Foo {
    my_bool: bool,
    my_num: isize,
    my_string: String,
}
struct Bar (bool, isize, String);
struct Baz;
```

.

```
struct Foo {
    my_bool: bool,
    my_num: isize,
    my_string: String,
}
struct Bar (bool, isize, String);
struct Baz;
```

```
struct Foo {
    my_bool: bool,
    my_num: isize,
    my_string: String,
}
struct Bar (bool, isize, String);
struct Baz;
```

Rust`struct`

```
struct Foo {
    my_bool: bool,
    my_num: isize,
    my_string: String,
}
struct Bar (bool, isize, String);
struct Baz;
```

Rust“”

```
struct Foo {
    my_bool: bool,
    my_num: isize,
    my_string: String,
}
struct Bar (bool, isize, String);
struct Baz;
```

structstructstructimpl

```
impl Foo {
    fn fiddle(&self) {
        // "self" refers to the value this method is being called on
        println!("fiddling {}", self.my_string);
    }
}

// ...
foo.fiddle(); // prints "fiddling hello"
```

&self herestruct Foofiddle。 &mut self

```
impl Foo {
    fn fiddle(&self) {
        // "self" refers to the value this method is being called on
        println!("fiddling {}", self.my_string);
    }
}

// ...
foo.fiddle(); // prints "fiddling hello"
```

self & 。 。 。 “”

```
impl Foo {
    fn fiddle(&self) {
        // "self" refers to the value this method is being called on
        println!("fiddling {}", self.my_string);
    }
}

// ...
foo.fiddle(); // prints "fiddling hello"
```

mutself self。 double。 implSelfimplFoo 。 。

“”structself。 struct

```
impl Foo {
    fn fiddle(&self) {
        // "self" refers to the value this method is being called on
        println!("fiddling {}", self.my_string);
    }
```

```
}

// ...
foo.fiddle(); // prints "fiddling hello"
```

◦ ◦ pub◦

◦ <>

```
struct Gen<T> {
    x: T,
    z: isize,
}

// ...
let _: Gen<bool> = Gen{x: true, z: 1};
let _: Gen<isize> = Gen{x: 42, z: 2};
let _: Gen<String> = Gen{x: String::from("hello"), z: 3};
```

```
struct Gen<T> {
    x: T,
    z: isize,
}

// ...
let _: Gen<bool> = Gen{x: true, z: 1};
let _: Gen<isize> = Gen{x: 42, z: 2};
let _: Gen<String> = Gen{x: String::from("hello"), z: 3};
```

```
struct Gen<T> {
    x: T,
    z: isize,
}

// ...
let _: Gen<bool> = Gen{x: true, z: 1};
let _: Gen<isize> = Gen{x: 42, z: 2};
let _: Gen<String> = Gen{x: String::from("hello"), z: 3};
```

struct

```
struct Gen<T> {
    x: T,
    z: isize,
}

// ...
let _: Gen<bool> = Gen{x: true, z: 1};
let _: Gen<isize> = Gen{x: 42, z: 2};
let _: Gen<String> = Gen{x: String::from("hello"), z: 3};
```

gxT◦

```
struct Gen<T> {
    x: T,
```

```
    z: isize,
}

// ...
let _: Gen<bool> = Gen{x: true, z: 1};
let _: Gen<isize> = Gen{x: 42, z: 2};
let _: Gen<String> = Gen{x: String::from("hello"), z: 3};
```

helloTfmt::Display Gen◦ Gen<(bool, isize)> hello ◦

structstruct

```
struct Gen<T> {
    x: T,
    z: isize,
}

// ...
let _: Gen<bool> = Gen{x: true, z: 1};
let _: Gen<isize> = Gen{x: 42, z: 2};
let _: Gen<String> = Gen{x: String::from("hello"), z: 3};
```

GenBxHash .x.hash() ◦ +◦

where<>

```
struct Gen<T> {
    x: T,
    z: isize,
}

// ...
let _: Gen<bool> = Gen{x: true, z: 1};
let _: Gen<isize> = Gen{x: 42, z: 2};
let _: Gen<String> = Gen{x: String::from("hello"), z: 3};
```

◦

struct

```
struct Gen<T> {
    x: T,
    z: isize,
}

// ...
let _: Gen<bool> = Gen{x: true, z: 1};
let _: Gen<isize> = Gen{x: 42, z: 2};
let _: Gen<String> = Gen{x: String::from("hello"), z: 3};
```

GenTimpl◦ impl

```
struct Gen<T> {
    x: T,
    z: isize,
}
```

```
// ...
let _: Gen<bool> = Gen{x: true, z: 1};
let _: Gen<isize> = Gen{x: 42, z: 2};
let _: Gen<String> = Gen{x: String::from("hello"), z: 3};
```

https://riptutorial.com/zh-CN/rust/topic/4583/

37:

Examples

. Rust. deref“tree”* s。 。

```
let mut name: String = "hello world".to_string();
// no deref happens here because push is defined in String itself
name.push('!');

let name_ref: &String = &name;
// Auto deref happens here to get to the String. See below
let name_len = name_ref.len();
// You can think of this as syntactic sugar for the following line:
let name_len2 = (*name_ref).len();

// Because of how the deref rules work,
// you can have an arbitrary number of references.
// The . operator is clever enough to know what to do.
let name_len3 = (&&&&&&&&&&&&name).len();
assert_eq!(name_len3, name_len);
```

std::ops::Deref trait。

```
let mut name: String = "hello world".to_string();
// no deref happens here because push is defined in String itself
name.push('!');

let name_ref: &String = &name;
// Auto deref happens here to get to the String. See below
let name_len = name_ref.len();
// You can think of this as syntactic sugar for the following line:
let name_len2 = (*name_ref).len();

// Because of how the deref rules work,
// you can have an arbitrary number of references.
// The . operator is clever enough to know what to do.
let name_len3 = (&&&&&&&&&&&&name).len();
assert_eq!(name_len3, name_len);
```

iterVec<T>[T]Vec<T> DerefTarget=[T]Vec<T>[T]*. 。

Deref

TU TDeref<Target=U> &T&U

```
fn foo(a: &[i32]) {
    // code
}

fn bar(s: &str) {
    // code
}
```

```
let v = vec![1, 2, 3];
foo(&v); // &Vec<i32> coerces into &[i32] because Vec<T> impls Deref<Target=[T]>

let s = "Hello world".to_string();
let rc = Rc::new(s);
// This works because Rc<T> impls Deref<Target=T> ∴ &Rc<String> coerces into
// &String which coerces into &str. This happens as much as needed at compile time.
bar(&rc);
```

DerefAsRef

```
fn work_on_bytes(slice: &[u8]) {}
```

Vec<T>[T; N]Deref<Target=[T]>

```
fn work_on_bytes(slice: &[u8]) {}
```

```
fn work_on_bytes(slice: &[u8]) {}
```

work_on_bytesas_ref()T [u8]◦

```
fn work_on_bytes(slice: &[u8]) {}
```

OptionDeref

```
use std::ops::Deref;
use std::fmt::Debug;

#[derive(Debug)]
struct RichOption<T>(Option<T>); // wrapper struct

impl<T> Deref for RichOption<T> {
    type Target = Option<T>; // Our wrapper struct will coerce into Option
    fn deref(&self) -> &Option<T> {
        &self.0 // We just extract the inner element
    }
}

impl<T: Debug> RichOption<T> {
    fn print_inner(&self) {
        println!("{:?}", self.0)
    }
}

fn main() {
    let x = RichOption(Some(1));
    println!("{:?}",x.map(|x| x + 1)); // Now we can use Option's methods...
    fn_that_takes_option(&x); // pass it to functions that take Option...
    x.print_inner() // and use it's own methods to extend Option
}

fn fn_that_takes_option<T : std::fmt::Debug>(x: &Option<T>) {
    println!("{:?}", x)
```

```
}
```

Deref

`DerefTDeref<Target=F>` `&T&F` F

```
fn f(x: &str) -> &str { x }
fn main() {
    // Compiler will coerce &&&&&&&str to &str and then pass it to our function
    f(&&&&&&&"It's a string");
}
```

`BoxArc` Deref

```
fn f(x: &str) -> &str { x }
fn main() {
    // Compiler will coerce &&&&&&&str to &str and then pass it to our function
    f(&&&&&&&"It's a string");
}
```

https://riptutorial.com/zh-CN/rust/topic/2574/

38: “1.1”

Rust 1.15Custom derive aka Macros 1.1。

`PartialEqDebug#[deriving(MyOwnDerive)]` 。 serde 。

Rust Book https //doc.rust-lang.org/stable/book/procedural-macros.html

Examples

helhelloworld

Cargo.toml

```
[package]
name = "customderive"
version = "0.1.1"

[lib]
proc-macro=true

[dependencies]
quote="^0.3.12"
syn="^0.11.4"
```

SRC / lib.rs

```
[package]
name = "customderive"
version = "0.1.1"

[lib]
proc-macro=true

[dependencies]
quote="^0.3.12"
syn="^0.11.4"
```

/ hello.rs

```
[package]
name = "customderive"
version = "0.1.1"

[lib]
proc-macro=true

[dependencies]
quote="^0.3.12"
syn="^0.11.4"
```

```
[package]
name = "customderive"
version = "0.1.1"

[lib]
proc-macro=true

[dependencies]
quote="^0.3.12"
syn="^0.11.4"
```

Cargo.toml

```
[package]
name = "customderive"
version = "0.1.0"
[lib]
proc-macro=true
```

SRC / lib.rs

```
[package]
name = "customderive"
version = "0.1.0"
[lib]
proc-macro=true
```

/ hello.rs

```
[package]
name = "customderive"
version = "0.1.0"
[lib]
proc-macro=true
```

Cargo.toml

```
[package]
name = "gettersetter"
version = "0.1.0"
[lib]
proc-macro=true
[dependencies]
quote="^0.3.12"
syn="^0.11.4"
```

SRC / lib.rs

```
[package]
name = "gettersetter"
version = "0.1.0"
[lib]
proc-macro=true
[dependencies]
```

```
quote="^0.3.12"
syn="^0.11.4"
```

/ hello.rs

```
[package]
name = "gettersetter"
version = "0.1.0"
[lib]
proc-macro=true
[dependencies]
quote="^0.3.12"
syn="^0.11.4"
```

https //github.com/emk/accessors

“1.1” https://riptutorial.com/zh-CN/rust/topic/9104/--1-1-

39:

Rust `std` ◦ LLVMRust`std` `core` ◦

Examples

[no_std]

```
#![feature(start, libc, lang_items)]
#![no_std]
#![no_main]

// The libc crate allows importing functions from C.
extern crate libc;

// A list of C functions that are being imported
extern {
    pub fn printf(format: *const u8, ...) -> i32;
}

#[no_mangle]
// The main function, with its input arguments ignored, and an exit status is returned
pub extern fn main(_nargs: i32, _args: *const *const u8) -> i32 {
    // Print "Hello, World" to stdout using printf
    unsafe {
        printf(b"Hello, World!\n" as *const u8);
    }

    // Exit with a return status of 0.
    0
}

#[lang = "eh_personality"] extern fn eh_personality() {}
#[lang = "panic_fmt"] extern fn panic_fmt() -> ! { panic!() }
```

https://riptutorial.com/zh-CN/rust/topic/8344/

40:

/◦ Cargocrates.io◦ Cargo`cargo build` `cargo run`cargo `test` Rust◦

- cargo new crate_name [--bin]
- [--bin]
- [--release]
- [--release]
-
-
-
-
-
-
- cargo [un] install binary_crate_name
- crate_name
-
- api_key

- `cargo bench`◦

Examples

```
cargo new my-library
```

`my-library`Rust

```
cargo new my-library
```

`cargo test my-library`◦

```
cargo new my-library
```

`my-binary`

```
cargo new my-library
```

`cargo`Hello World`cargo run` ◦

`init`

```
cargo new my-library
```

--bin。 。

```
cargo build
```

--release。 。 。

```
cargo build
```

```
cargo test
```

```
cargo test
```

```
cargo test
```

shell“Hello world”Cargo

```
$ cargo new hello --bin
$ cd hello
$ cargo run
   Compiling hello v0.1.0 (file:///home/rust/hello)
     Running `target/debug/hello`
Hello, world!
```

src/main.rs。

crates.ioCargo“ *Crates.io* ”。

```
cargo package
cargo publish
```

Cargo.toml。 .gitignoreCargo.toml。

Crates.io

crates.ioGitHub;。

GitHubcrates.io“ *GitHub* ”crates.io。 crates.io。

API “ ”

```
cargo login abcdefghijklmnopqrstuvwxyz1234567890rust
```

/cargo。

API -

https://riptutorial.com/zh-CN/rust/topic/1084/

41:

- AsRefBorrow◦ Borrow AsRefAsRef◦
- From<A> for BInto<B> for A ◦
- From<A> for A◦

Examples

RustFrom trait◦ TypeATypeB

```
impl From<TypeA> for TypeB
```

TypeBTypeA◦ From

```
impl From<TypeA> for TypeB
```

AsRefAsMut

std::convert::AsRefstd::convert::AsMut◦ AB

```
impl AsRef<B> for A
```

a &A&B

```
impl AsRef<B> for A
```

a &mut A&mut B

◦ std::fs::File.open()

```
impl AsRef<B> for A
```

File.open()Path OsStr OsString str StringPathBuf AsRef<Path> ◦

BorrowBorrowMutToOwned

std::borrow::Borrow std::borrow::BorrowMut std::borrow::Borrowstd::borrow::BorrowMut◦ AB

```
impl Borrow<B> for A
```

BA std::collections::HashMap.get()Borrowget()AHashMap&B◦

---

std::borrow::ToOwned◦

AB

```
impl Borrow<B> for A
```

ATBorrow<T> BToOwned◦

DerefDerefMut

std::ops::Derefstd::ops::DerefMut traits*x ◦ AB

```
impl Deref<Target=B> for A
```

&A&B

```
impl Deref<Target=B> for A
```

&mut A&mut B

Deref DerefMut *deref coercion*&A &mut A &B &mut B ◦ String&str&String&str ◦

DerefMutDeref◦

AsRefDerefDerefMut ◦

https://riptutorial.com/zh-CN/rust/topic/2661/

42:

“”Rust。 `std::ops`。

Examples

+

+`std::ops::Add` trait。

```
pub trait Add<RHS = Self> {
    type Output;
    fn add(self, rhs: RHS) -> Self::Output;
}
```

-
-
- `Output`

3。

trait。

+

```
pub trait Add<RHS = Self> {
    type Output;
    fn add(self, rhs: RHS) -> Self::Output;
}
```

https://riptutorial.com/zh-CN/rust/topic/7271/

43:

Rust`Iterator`◦ `Vec<T>` ◦

Examples

```
//         Iterator  Adapter
//             |       |
let my_map = (1..6).map(|x| x * x);
println!("{:?}", my_map);
```


```
//         Iterator  Adapter
//             |       |
let my_map = (1..6).map(|x| x * x);
println!("{:?}", my_map);
```


- “”◦

◦


```
//         Iterator  Adapter
//             |       |
let my_map = (1..6).map(|x| x * x);
println!("{:?}", my_map);
```

```
//         Iterator  Adapter
//             |       |
let my_map = (1..6).map(|x| x * x);
println!("{:?}", my_map);
```


`find` `foldsum` ◦


```
//         Iterator  Adapter
//             |       |
let my_map = (1..6).map(|x| x * x);
println!("{:?}", my_map);
```

```
//         Iterator  Adapter
//             |       |
let my_map = (1..6).map(|x| x * x);
println!("{:?}", my_map);
```


```
fn is_prime(n: u64) -> bool {
    (2..n).all(|divisor| n % divisor != 0)
}
```

◦ `n`

```
fn is_prime(n: u64) -> bool {
```

```
    (2..n).all(|divisor| n % divisor != 0)
}
```

```
struct Fibonacci(u64, u64);

impl Iterator for Fibonacci {
    type Item = u64;

    // The method that generates each item
    fn next(&mut self) -> Option<Self::Item> {
        let ret = self.0;
        self.0 = self.1;
        self.1 += ret;

        Some(ret) // since `None` is never returned, we have an infinite iterator
    }

    // Implementing the `next()` method suffices since every other iterator
    // method has a default implementation
}
```

```
struct Fibonacci(u64, u64);

impl Iterator for Fibonacci {
    type Item = u64;

    // The method that generates each item
    fn next(&mut self) -> Option<Self::Item> {
        let ret = self.0;
        self.0 = self.1;
        self.1 += ret;

        Some(ret) // since `None` is never returned, we have an infinite iterator
    }

    // Implementing the `next()` method suffices since every other iterator
    // method has a default implementation
}
```

https://riptutorial.com/zh-CN/rust/topic/4657/

44:

`Option<T>`Rust。 C`null` `Option`'optionals'Haskell`Maybe` monad。 `Option`Rust。

Examples

Option

```
// The Option type can either contain Some value or None.
fn find(value: i32, slice: &[i32]) -> Option<usize> {
    for (index, &element) in slice.iter().enumerate() {
        if element == value {
            // Return a value (wrapped in Some).
            return Some(index);
        }
    }
    // Return no value.
    None
}

fn main() {
    let array = [1, 2, 3, 4, 5];
    // Pattern match against the Option value.
    if let Some(index) = find(2, &array) {
        // Here, there is a value.
        println!("The element 2 is at index {}.", index);
    }

    // Check if the result is None (no value).
    if let None = find(12, &array) {
        // Here, there is no value.
        println!("The element 12 is not in the array.");
    }

    // You can also use `is_some` and `is_none` helpers
    if find(12, &array).is_none() {
        println!("The element 12 is not in the array.");
    }
}
```

```
fn main() {
    let maybe_cake = Some("Chocolate cake");
    let not_cake = None;

    // The unwrap method retrieves the value from the Option
    // and panics if the value is None
    println!("{}", maybe_cake.unwrap());

    // The expect method works much like the unwrap method,
    // but panics with a custom, user provided message.
    println!("{}", not_cake.expect("The cake is a lie."));

    // The unwrap_or method can be used to provide a default value in case
    // the value contained within the option is None. This example would
    // print "Cheesecake".
```

```
    println!("{}", not_cake.unwrap_or("Cheesecake"));

    // The unwrap_or_else method works like the unwrap_or method,
    // but allows us to provide a function which will return the
    // fallback value. This example would print "Pumpkin Cake".
    println!("{}", not_cake.unwrap_or_else(|| { "Pumpkin Cake" }));

    // A match statement can be used to safely handle the possibility of none.
    match maybe_cake {
        Some(cake) => println!("{} was consumed.", cake),
        None       => println!("There was no cake.")
    }

    // The if let statement can also be used to destructure an Option.
    if let Some(cake) = maybe_cake {
        println!("{} was consumed.", cake);
    }
}
```

`T&Option<T>◦ as_ref()&Option<&T> ◦`

Rust◦ Option `&Option<T> -◦`

```
#[derive(Debug)]
struct Foo;

fn main() {
    let wrapped = Some(Foo);
    let wrapped_ref = &wrapped;

    println!("{:?}", wrapped_ref.unwrap()); // Error!
}
```

[--explain E0507]

`Option<T>◦ as_ref()&T`

```
#[derive(Debug)]
struct Foo;

fn main() {
    let wrapped = Some(Foo);
    let wrapped_ref = &wrapped;

    println!("{:?}", wrapped_ref.unwrap()); // Error!
}
```

mapand_thenOption

`mapOption◦`

```
fn main() {

    // We start with an Option value (Option<i32> in this case).
    let some_number = Some(9);
```

```
    // Let's do some consecutive calculations with our number.
    // The crucial point here is that we don't have to unwrap
    // the content of our Option type - instead, we're just
    // transforming its content. The result of the whole operation
    // will still be an Option<i32>. If the initial value of
    // 'some_number' was 'None' instead of 9, then the result
    //  would also be 'None'.
    let another_number = some_number
        .map(|n| n - 1) // => Some(8)
        .map(|n| n * n) // => Some(64)
        .and_then(|n| divide(n, 4)); // => Some(16)

    // In the last line above, we're doing a division using a helper
    // function (definition: see bottom).
    // 'and_then' is very similar to 'map', but allows us to pass a
    // function which returns an Option type itself. To ensure that we
    // don't end up with Option<Option<i32>>, 'and_then' flattens the
    // result (in other languages, 'and_then' is also known as 'flatmap').

    println!("{}", to_message(another_number));
    // => "16 is definitely a number!"

    // For the sake of completeness, let's check the result when
    // dividing by zero.
    let final_number = another_number
        .and_then(|n| divide(n, 0)); // => None

    println!("{}", to_message(final_number));
    // => "None!"
}

// Just a helper function for integer division. In case
// the divisor is zero, we'll get 'None' as result.
fn divide(number: i32, divisor: i32) -> Option<i32> {
    if divisor != 0  { Some(number/divisor) } else { None }
}

// Creates a message that tells us whether our
// Option<i32> contains a number or not. There are other
// ways to achieve the same result, but let's just use
// map again!
fn to_message(number: Option<i32>) -> String {
    number
        .map(|n| format!("{} is definitely a number!", n)) // => Some("...")
        .unwrap_or("None!".to_string()) // => "..."
}
```

https://riptutorial.com/zh-CN/rust/topic/1125/

45:

IronRustWebHyper。 Iron。

Examples

'Hello'

。

```
extern crate iron;

use iron::prelude::*;
use iron::status;

// You can pass the handler as a function or a closure. In this
// case, we've chosen a function for clarity.
// Since we don't care about the request, we bind it to _.
fn handler(_: &mut Request) -> IronResult<Response> {
    Ok(Response::with((status::Ok, "Hello, Stack Overflow")))
}

fn main() {
    Iron::new(handler).http("localhost:1337").expect("Server failed!")
}
```

`Iron` `expect`。 `http()` 。

`Cargo.toml`

```
[dependencies]
iron = "0.4.0"
```

`cargo build` CargoIron。

IronWeb。

Iron`Cargo.toml`。

```
[dependencies]
iron = "0.4.*"
```

Iron。 IronIron。 IronRouter。

```
[dependencies]
iron = "0.4.*"
```

。

```
[dependencies]
iron = "0.4.*"
```

`main()`。 Web。 。

```
[dependencies]
iron = "0.4.*"
```

。 Iron `Router`“”URL `"/"` `"/:query"` 。

“query”IronURL。

Iron`router`URL。 。

```
[dependencies]
iron = "0.4.*"
```

`handlerquery_handler` 。 URLURL。

URL`"query"`。

```
[dependencies]
iron = "0.4.*"
```

`localhost:3000localhost:3000`。 `"OK"`。

。

https://riptutorial.com/zh-CN/rust/topic/8060/

46:

RustRust。 。

RustGitHubrust-lang/rustrust-lang-nursery/fmt-rfcs。 rust-lang。

rustfmtclippy。 Cargo

```
cargo install clippy
cargo install rustfmt
```


```
cargo install clippy
cargo install rustfmt
```


Examples

```
// Lines should never exceed 100 characters.
// Instead, just wrap on to the next line.
let bad_example = "So this sort of code should never really happen, because it's really hard
to fit on the screen!";
```


```
// Lines should never exceed 100 characters.
// Instead, just wrap on to the next line.
let bad_example = "So this sort of code should never really happen, because it's really hard
to fit on the screen!";
```


。


```
// Lines should never exceed 100 characters.
// Instead, just wrap on to the next line.
let bad_example = "So this sort of code should never really happen, because it's really hard
to fit on the screen!";
```

```
// Lines should never exceed 100 characters.
// Instead, just wrap on to the next line.
let bad_example = "So this sort of code should never really happen, because it's really hard
to fit on the screen!";
```

```
// Lines should never exceed 100 characters.
// Instead, just wrap on to the next line.
let bad_example = "So this sort of code should never really happen, because it's really hard
to fit on the screen!";
```

```
// Lines should never exceed 100 characters.
// Instead, just wrap on to the next line.
let bad_example = "So this sort of code should never really happen, because it's really hard
to fit on the screen!";
```

```
// Lines should never exceed 100 characters.
// Instead, just wrap on to the next line.
let bad_example = "So this sort of code should never really happen, because it's really hard
to fit on the screen!";
```


```
// Lines should never exceed 100 characters.
// Instead, just wrap on to the next line.
let bad_example = "So this sort of code should never really happen, because it's really hard
to fit on the screen!";
```


```
// To reduce the amount of imports that users need, you should
// re-export important structs and traits.
pub use foo::Client;
pub use bar::Server;
```

crates`preludestd::io::prelude` ◦ `use std::io::prelude::*;use std::io::prelude::*;`

- `extern crate`
- `use`
  ◦
- `pub use`

```
// Structs use UpperCamelCase.
pub struct Snafucator {

}

mod snafucators {
    // Try to avoid 'stuttering' by repeating
    // the module name in the struct name.

    // Bad:
    pub struct OrderedSnafucator {

    }

    // Good:
    pub struct Ordered {

    }
}
```


```
// Structs use UpperCamelCase.
pub struct Snafucator {

}

mod snafucators {
    // Try to avoid 'stuttering' by repeating
    // the module name in the struct name.

    // Bad:
    pub struct OrderedSnafucator {

    }
```

```
    // Good:
    pub struct Ordered {

    }
}
```

```
// Structs use UpperCamelCase.
pub struct Snafucator {

}

mod snafucators {
    // Try to avoid 'stuttering' by repeating
    // the module name in the struct name.

    // Bad:
    pub struct OrderedSnafucator {

    }

    // Good:
    pub struct Ordered {

    }
}
```

```
// Structs use UpperCamelCase.
pub struct Snafucator {

}

mod snafucators {
    // Try to avoid 'stuttering' by repeating
    // the module name in the struct name.

    // Bad:
    pub struct OrderedSnafucator {

    }

    // Good:
    pub struct Ordered {

    }
}
```

```
// Structs use UpperCamelCase.
pub struct Snafucator {

}

mod snafucators {
    // Try to avoid 'stuttering' by repeating
    // the module name in the struct name.

    // Bad:
    pub struct OrderedSnafucator {
```

```
    }

    // Good:
    pub struct Ordered {

    }
}
```

```
// Structs use UpperCamelCase.
pub struct Snafucator {

}

mod snafucators {
    // Try to avoid 'stuttering' by repeating
    // the module name in the struct name.

    // Bad:
    pub struct OrderedSnafucator {

    }

    // Good:
    pub struct Ordered {

    }
}
```

```
// Structs use UpperCamelCase.
pub struct Snafucator {

}

mod snafucators {
    // Try to avoid 'stuttering' by repeating
    // the module name in the struct name.

    // Bad:
    pub struct OrderedSnafucator {

    }

    // Good:
    pub struct Ordered {

    }
}
```

```
// Structs use UpperCamelCase.
pub struct Snafucator {

}

mod snafucators {
    // Try to avoid 'stuttering' by repeating
    // the module name in the struct name.

    // Bad:
```

```
    pub struct OrderedSnafucator {

    }

    // Good:
    pub struct Ordered {

    }
}
```

TCP

- UpperCamelCase TcpClient
- snake_casetcp_client
- SCREAMING_SNAKE_CASETCP_CLIENT

```
// There should be one space after the colon of the type
// annotation. This rule applies in variable declarations,
// struct fields, functions and methods.

// GOOD:
let mut buffer: String = String::new();
// BAD:
let mut buffer:String = String::new();
let mut buffer : String = String::new();
```


```
// There should be one space after the colon of the type
// annotation. This rule applies in variable declarations,
// struct fields, functions and methods.

// GOOD:
let mut buffer: String = String::new();
// BAD:
let mut buffer:String = String::new();
let mut buffer : String = String::new();
```

```
// There should be one space after the colon of the type
// annotation. This rule applies in variable declarations,
// struct fields, functions and methods.

// GOOD:
let mut buffer: String = String::new();
// BAD:
let mut buffer:String = String::new();
let mut buffer : String = String::new();
```


https://riptutorial.com/zh-CN/rust/topic/4620/

47:

Rust`Result<T, E>`。 Panics。

The Rust Programming Language The Book

Examples

```
use std::io::{Read, Result as IoResult};
use std::fs::File;

struct Config(u8);

fn read_config() -> IoResult<String> {
    let mut s = String::new();
    let mut file = File::open(&get_local_config_path())
        // or_else closure is invoked if Result is Err.
        .or_else(|_| File::open(&get_global_config_path()))?;
    // Note: In `or_else`, the closure should return a Result with a matching
    //       Ok type, whereas in `and_then`, the returned Result should have a
    //       matching Err type.
    let _ = file.read_to_string(&mut s)?;
    Ok(s)
}

struct ParseError;

fn parse_config(conf_str: String) -> Result<Config, ParseError> {
    // Parse the config string...
    if conf_str.starts_with("bananas") {
        Err(ParseError)
    } else {
        Ok(Config(42))
    }
}

fn run() -> Result<(), String> {
    // Note: The error type of this function is String. We use map_err below to
    //       make the error values into String type
    let conf_str = read_config()
        .map_err(|e| format!("Failed to read config file: {}", e))?;
    // Note: Instead of using `?` above, we can use `and_then` to wrap the let
    //       expression below.
    let conf_val = parse_config(conf_str)
        .map(|Config(v)| v / 2) // map can be used to map just the Ok value
        .map_err(|_| "Failed to parse the config string!".to_string())?;

    // Run...

    Ok(())
}

fn main() {
    match run() {
        Ok(_) => println!("Bye!"),
        Err(e) => println!("Error: {}", e),
```

```
    }
}

fn get_local_config_path() -> String {
    let user_config_prefix = "/home/user/.config";
    // code to get the user config directory
    format!("{}/my_app.rc", user_config_prefix)
}

fn get_global_config_path() -> String {
    let global_config_prefix = "/etc";
    // code to get the global config directory
    format!("{}/my_app.rc", global_config_prefix)
}
```

```
use std::io::{Read, Result as IoResult};
use std::fs::File;

struct Config(u8);

fn read_config() -> IoResult<String> {
    let mut s = String::new();
    let mut file = File::open(&get_local_config_path())
        // or_else closure is invoked if Result is Err.
        .or_else(|_| File::open(&get_global_config_path()))?;
    // Note: In `or_else`, the closure should return a Result with a matching
    //       Ok type, whereas in `and_then`, the returned Result should have a
    //       matching Err type.
    let _ = file.read_to_string(&mut s)?;
    Ok(s)
}

struct ParseError;

fn parse_config(conf_str: String) -> Result<Config, ParseError> {
    // Parse the config string...
    if conf_str.starts_with("bananas") {
        Err(ParseError)
    } else {
        Ok(Config(42))
    }
}

fn run() -> Result<(), String> {
    // Note: The error type of this function is String. We use map_err below to
    //       make the error values into String type
    let conf_str = read_config()
        .map_err(|e| format!("Failed to read config file: {}", e))?;
    // Note: Instead of using `?` above, we can use `and_then` to wrap the let
    //       expression below.
    let conf_val = parse_config(conf_str)
        .map(|Config(v)| v / 2) // map can be used to map just the Ok value
        .map_err(|_| "Failed to parse the config string!".to_string())?;

    // Run...

    Ok(())
}

fn main() {
```

```
    match run() {
        Ok(_) => println!("Bye!"),
        Err(e) => println!("Error: {}", e),
    }
}

fn get_local_config_path() -> String {
    let user_config_prefix = "/home/user/.config";
    // code to get the user config directory
    format!("{}/my_app.rc", user_config_prefix)
}

fn get_global_config_path() -> String {
    let global_config_prefix = "/etc";
    // code to get the global config directory
    format!("{}/my_app.rc", global_config_prefix)
}
```

```
use std::io::{Read, Result as IoResult};
use std::fs::File;

struct Config(u8);

fn read_config() -> IoResult<String> {
    let mut s = String::new();
    let mut file = File::open(&get_local_config_path())
        // or_else closure is invoked if Result is Err.
        .or_else(|_| File::open(&get_global_config_path()))?;
    // Note: In `or_else`, the closure should return a Result with a matching
    //       Ok type, whereas in `and_then`, the returned Result should have a
    //       matching Err type.
    let _ = file.read_to_string(&mut s)?;
    Ok(s)
}

struct ParseError;

fn parse_config(conf_str: String) -> Result<Config, ParseError> {
    // Parse the config string...
    if conf_str.starts_with("bananas") {
        Err(ParseError)
    } else {
        Ok(Config(42))
    }
}

fn run() -> Result<(), String> {
    // Note: The error type of this function is String. We use map_err below to
    //       make the error values into String type
    let conf_str = read_config()
        .map_err(|e| format!("Failed to read config file: {}", e))?;
    // Note: Instead of using `?` above, we can use `and_then` to wrap the let
    //       expression below.
    let conf_val = parse_config(conf_str)
        .map(|Config(v)| v / 2) // map can be used to map just the Ok value
        .map_err(|_| "Failed to parse the config string!".to_string())?;

    // Run...

    Ok(())
```

```
}

fn main() {
    match run() {
        Ok(_) => println!("Bye!"),
        Err(e) => println!("Error: {}", e),
    }
}

fn get_local_config_path() -> String {
    let user_config_prefix = "/home/user/.config";
    // code to get the user config directory
    format!("{}/my_app.rc", user_config_prefix)
}

fn get_global_config_path() -> String {
    let global_config_prefix = "/etc";
    // code to get the global config directory
    format!("{}/my_app.rc", global_config_prefix)
}
```

docs ◦ ◦ error-chain ◦

```
use std::error::Error;
use std::fmt;
use std::convert::From;
use std::io::Error as IoError;
use std::str::Utf8Error;

#[derive(Debug)] // Allow the use of "{:?}" format specifier
enum CustomError {
    Io(IoError),
    Utf8(Utf8Error),
    Other,
}

// Allow the use of "{}" format specifier
impl fmt::Display for CustomError {
    fn fmt(&self, f: &mut fmt::Formatter) -> fmt::Result {
        match *self {
            CustomError::Io(ref cause) => write!(f, "I/O Error: {}", cause),
            CustomError::Utf8(ref cause) => write!(f, "UTF-8 Error: {}", cause),
            CustomError::Other => write!(f, "Unknown error!"),
        }
    }
}

// Allow this type to be treated like an error
impl Error for CustomError {
    fn description(&self) -> &str {
        match *self {
            CustomError::Io(ref cause) => cause.description(),
            CustomError::Utf8(ref cause) => cause.description(),
            CustomError::Other => "Unknown error!",
        }
    }

    fn cause(&self) -> Option<&Error> {
        match *self {
            CustomError::Io(ref cause) => Some(cause),
```

```
            CustomError::Utf8(ref cause) => Some(cause),
            CustomError::Other => None,
        }
    }
}

// Support converting system errors into our custom error.
// This trait is used in `try!`.
impl From<IoError> for CustomError {
    fn from(cause: IoError) -> CustomError {
        CustomError::Io(cause)
    }
}
impl From<Utf8Error> for CustomError {
    fn from(cause: Utf8Error) -> CustomError {
        CustomError::Utf8(cause)
    }
}
```

◦ std::error::Error

```
use std::error::Error;

let orig_error = call_returning_error();

// Use an Option<&Error>. This is the return type of Error.cause().
let mut err = Some(&orig_error as &Error);

// Print each error's cause until the cause is None.
while let Some(e) = err {
    println!("{}", e);
    err = e.cause();
}
```

Result<T, E>enum Ok(T)T Err(E)E◦

```
enum DateError {
    InvalidDay,
    InvalidMonth,
}

struct Date {
    day: u8,
    month: u8,
    year: i16,
}

fn validate(date: &Date) -> Result<(), DateError> {
    if date.month < 1 || date.month > 12 {
        Err(DateError::InvalidMonth)
    } else if date.day < 1 || date.day > 31 {
        Err(DateError::InvalidDay)
    } else {
        Ok(())
    }
}

fn add_days(date: Date, days: i32) -> Result<Date, DateError> {
```

```
    validate(&date)?; // notice `?` -- returns early on error
    // the date logic ...
    Ok(date)
}
```

?◦

Error ◦ ◦

```
enum DateError {
    InvalidDay,
    InvalidMonth,
}

struct Date {
    day: u8,
    month: u8,
    year: i16,
}

fn validate(date: &Date) -> Result<(), DateError> {
    if date.month < 1 || date.month > 12 {
        Err(DateError::InvalidMonth)
    } else if date.day < 1 || date.day > 31 {
        Err(DateError::InvalidDay)
    } else {
        Ok(())
    }
}

fn add_days(date: Date, days: i32) -> Result<Date, DateError> {
    validate(&date)?; // notice `?` -- returns early on error
    // the date logic ...
    Ok(date)
}
```

Option<T>◦ Option<T>◦ Result<T, E>◦ Result<T, E>◦ ◦

https://riptutorial.com/zh-CN/rust/topic/1762/

48: lambda

Examples

lambda

```
// A simple adder function defined as a lambda expression.
// Unlike with regular functions, parameter types often may be omitted because the
// compiler can infer their types
let adder = |a, b| a + b;
// Lambdas can span across multiple lines, like normal functions.
let multiplier = |a: i32, b: i32| {
    let c = b;
    let b = a;
    let a = c;
    a * b
};

// Since lambdas are anonymous functions, they can be called like other functions
println!("{}", adder(3, 5));
println!("{}", multiplier(3, 5));
```


```
// A simple adder function defined as a lambda expression.
// Unlike with regular functions, parameter types often may be omitted because the
// compiler can infer their types
let adder = |a, b| a + b;
// Lambdas can span across multiple lines, like normal functions.
let multiplier = |a: i32, b: i32| {
    let c = b;
    let b = a;
    let a = c;
    a * b
};

// Since lambdas are anonymous functions, they can be called like other functions
println!("{}", adder(3, 5));
println!("{}", multiplier(3, 5));
```


lambda。 lambdas。

```

// variable definition outside the lambda expression...
let lucky_number: usize = 663;

// but the our function can access it anyway, thanks to the closures
let print_lucky_number = || println!("{}", lucky_number);

// finally call the closure
print_lucky_number();
```


```

// variable definition outside the lambda expression...
let lucky_number: usize = 663;
```

```
// but the our function can access it anyway, thanks to the closures
let print_lucky_number = || println!("{}", lucky_number);

// finally call the closure
print_lucky_number();
```

Lambdas

```
// lambda expressions can have explicitly annotated return types
let floor_func = |x: f64| -> i64 { x.floor() as i64 };
```

lambda

lambda。

```
// This function takes two integers and a function that performs some operation on the two
arguments
fn apply_function<T>(a: i32, b: i32, func: T) -> i32 where T: Fn(i32, i32) -> i32 {
    // apply the passed function to arguments a and b
    func(a, b)
}

// let's define three lambdas, each operating on the same parameters
let sum = |a, b| a + b;
let product = |a, b| a * b;
let diff = |a, b| a - b;

// And now let's pass them to apply_function along with some arbitary values
println!("3 + 6 = {}", apply_function(3, 6, sum));
println!("-4 * 9 = {}", apply_function(-4, 9, product));
println!("7 - (-3) = {}", apply_function(7, -3, diff));
```

```
// This function takes two integers and a function that performs some operation on the two
arguments
fn apply_function<T>(a: i32, b: i32, func: T) -> i32 where T: Fn(i32, i32) -> i32 {
    // apply the passed function to arguments a and b
    func(a, b)
}

// let's define three lambdas, each operating on the same parameters
let sum = |a, b| a + b;
let product = |a, b| a * b;
let diff = |a, b| a - b;

// And now let's pass them to apply_function along with some arbitary values
println!("3 + 6 = {}", apply_function(3, 6, sum));
println!("-4 * 9 = {}", apply_function(-4, 9, product));
println!("7 - (-3) = {}", apply_function(7, -3, diff));
```

lambdas

lambdas。

```
// Box in the return type moves the function from the stack to the heap
fn curried_adder(a: i32) -> Box<Fn(i32) -> i32> {
    // 'move' applies move semantics to a, so it can outlive this function call
    Box::new(move |b| a + b)
}

println!("3 + 4 = {}", curried_adder(3)(4));
```

```
3 + 4 = 7
```

lambda https://riptutorial.com/zh-CN/rust/topic/1815/lambda

49:

Rust*rand* crate。 Rust *rand* crateRust。 rand crateRust。

RNG。 RNG`thread_rng`random。 RNGUnix`/dev/urandom` 32 KiB。

`OsRng` Unix`/dev/urandom`Windows`CryptGenRandom()` 。 。

Examples

Rand

crateCargo.toml。

```
[dependencies]
rand = "0.3"
```

crates.io`rand`。 。

```
[dependencies]
rand = "0.3"
```

。 。 。

```
[dependencies]
rand = "0.3"
```

。

```
[dependencies]
rand = "0.3"
```

Rand

`random` 。

```
fn main() {
    let tuple = rand::random::<(f64, char)>();
    println!("{:?}", tuple)
}
```

。 。

。

```
fn main() {
```

```
    let tuple = rand::random::<(f64, char)>();
    println!("{:?}", tuple)
}
```

https://riptutorial.com/zh-CN/rust/topic/8864/

50: Rust

Rust。

RustTraits。OO。。

Examples

Rust“”。Rust。。

Python

```
class Animal:
    def speak(self):
        print("The " + self.animal_type + " said " + self.noise)

class Dog(Animal):
    def __init__(self):
        self.animal_type = 'dog'
        self.noise = 'woof'
```

Rust。

```
class Animal:
    def speak(self):
        print("The " + self.animal_type + " said " + self.noise)

class Dog(Animal):
    def __init__(self):
        self.animal_type = 'dog'
        self.noise = 'woof'
```

。

“The {animal} say {noise}”。 `Speak` for `Animal`

```
class Animal:
    def speak(self):
        print("The " + self.animal_type + " said " + self.noise)

class Dog(Animal):
    def __init__(self):
        self.animal_type = 'dog'
        self.noise = 'woof'
```

。Python。 `Human Humanspeak`

```
class Animal:
    def speak(self):
        print("The " + self.animal_type + " said " + self.noise)
```

```
class Dog(Animal):
    def __init__(self):
        self.animal_type = 'dog'
        self.noise = 'woof'
```

Java

```
interface ShapeVisitor {
    void visit(Circle c);
    void visit(Rectangle r);
}

interface Shape {
    void accept(ShapeVisitor sv);
}

class Circle implements Shape {
    private Point center;
    private double radius;

    public Circle(Point center, double radius) {
        this.center = center;
        this.radius = radius;
    }

    public Point getCenter() { return center; }
    public double getRadius() { return radius; }

    @Override
    public void accept(ShapeVisitor sv) {
        sv.visit(this);
    }
}

class Rectangle implements Shape {
    private Point lowerLeftCorner;
    private Point upperRightCorner;

    public Rectangle(Point lowerLeftCorner, Point upperRightCorner) {
        this.lowerLeftCorner = lowerLeftCorner;
        this.upperRightCorner = upperRightCorner;
    }

    public double length() { ... }
    public double width() { ... }

    @Override
    public void accept(ShapeVisitor sv) {
        sv.visit(this);
    }
}

class AreaCalculator implements ShapeVisitor {
    private double area = 0.0;

    public double getArea() { return area; }

    public void visit(Circle c) {
        area = Math.PI * c.radius() * c.radius();
    }
```

```
    public void visit(Rectangle r) {
         area = r.length() * r.width();
    }
}

double computeArea(Shape s) {
    AreaCalculator ac = new AreaCalculator();
    s.accept(ac);
    return ac.getArea();
}
```

Rust。

```
interface ShapeVisitor {
    void visit(Circle c);
    void visit(Rectangle r);
}

interface Shape {
    void accept(ShapeVisitor sv);
}

class Circle implements Shape {
    private Point center;
    private double radius;

    public Circle(Point center, double radius) {
        this.center = center;
        this.radius = radius;
    }

    public Point getCenter() { return center; }
    public double getRadius() { return radius; }

    @Override
    public void accept(ShapeVisitor sv) {
        sv.visit(this);
    }
}

class Rectangle implements Shape {
    private Point lowerLeftCorner;
    private Point upperRightCorner;

    public Rectangle(Point lowerLeftCorner, Point upperRightCorner) {
        this.lowerLeftCorner = lowerLeftCorner;
        this.upperRightCorner = upperRightCorner;
    }

    public double length() { ... }
    public double width() { ... }

    @Override
    public void accept(ShapeVisitor sv) {
        sv.visit(this);
    }
}

class AreaCalculator implements ShapeVisitor {
```

```
    private double area = 0.0;

    public double getArea() { return area; }

    public void visit(Circle c) {
        area = Math.PI * c.radius() * c.radius();
    }

    public void visit(Rectangle r) {
         area = r.length() * r.width();
    }
}

double computeArea(Shape s) {
    AreaCalculator ac = new AreaCalculator();
    s.accept(ac);
    return ac.getArea();
}
```

```
interface ShapeVisitor {
    void visit(Circle c);
    void visit(Rectangle r);
}

interface Shape {
    void accept(ShapeVisitor sv);
}

class Circle implements Shape {
    private Point center;
    private double radius;

    public Circle(Point center, double radius) {
        this.center = center;
        this.radius = radius;
    }

    public Point getCenter() { return center; }
    public double getRadius() { return radius; }

    @Override
    public void accept(ShapeVisitor sv) {
        sv.visit(this);
    }
}

class Rectangle implements Shape {
    private Point lowerLeftCorner;
    private Point upperRightCorner;

    public Rectangle(Point lowerLeftCorner, Point upperRightCorner) {
        this.lowerLeftCorner = lowerLeftCorner;
        this.upperRightCorner = upperRightCorner;
    }

    public double length() { ... }
    public double width() { ... }

    @Override
    public void accept(ShapeVisitor sv) {
```

```
        sv.visit(this);
    }
}

class AreaCalculator implements ShapeVisitor {
    private double area = 0.0;

    public double getArea() { return area; }

    public void visit(Circle c) {
        area = Math.PI * c.radius() * c.radius();
    }

    public void visit(Rectangle r) {
         area = r.length() * r.width();
    }
}

double computeArea(Shape s) {
    AreaCalculator ac = new AreaCalculator();
    s.accept(ac);
    return ac.getArea();
}
```

Rust https://riptutorial.com/zh-CN/rust/topic/6737/rust

S. No		Contributors
1	Rust	Andy Hayden, ar-ms, Aurora0001, Community, Cormac O'Brien, D. Ataro, David Grayson, Eric Platon, gavinb, IceyEC, John, Jon Gjengset, Kellen, kennytm, Kevin Montrose, Lukabot, mmstick, Neikos, Pavel Strakhov, Shepmaster, Timidger, Tot Zam, Tshepang, Wolf, xfix, Yohaï Berreby
2	Drop Trait - Rust	Leo Tindall, Neikos
3	GUI	eddy, vaartis
4	PhantomData	Neikos
5	rustup	torkleyy
6	SERDE	Aurora0001, dtolnay, kennytm
7	TCP	E_net4
8		Aurora0001, John
9		Aurora0001, Jean Pierre Dudey, mmstick
10		adelarsq, Ameo, Aurora0001, John, Jon Gjengset, Matthieu M.
11		Cormac O'Brien, Jean Pierre Dudey, John, kennytm, Leo Tindall , mcarton
12		4444, Aurora0001
13		xea
14		John
15		Aurora0001
16	FFI	Aurora0001, John, Konstantin V. Salikhov
17		Arrem, Aurora0001, David Grayson, KokaKiwi, Lukas Kalbertodt , mcarton, rap-2-h, tmr232, Yos Riady
18		Aurora0001, kennytm, Matt Smith
19		antoyo, Cormac O'Brien, Jean Pierre Dudey, letmutx, xetra11
20		Andy Hayden, apopiak, Arrem, Aurora0001, JDemler, John, kennytm, Mario Carneiro, Matt Smith, Matthieu M., mcarton,

		Sanpi, Shepmaster, Timidger, Winger Sendon, YOU
21		a10y, adelarsq, Arrem, Aurora0001, Cormac O'Brien, Hauleth, John, kennytm, Leo Tindall, Matt Smith, Matthieu M., Mylainos, RamenChef, SplittyDev, tversteeg, xea
22		Aurora0001, Leo Tindall, Timidger
23		Aurora0001, Jon Gjengset
24		Aurora0001, John, Ruud, xea, zrneely
25		antoyo, Aurora0001, John, Matthieu M.
26	I / O.	antoyo, Kornel
27		Aurora0001, Cormac O'Brien, Hauleth
28	IO	KolesnichenkoDS
29		Aurora0001, Cormac O'Brien, Dumindu Madunuwan, John, KokaKiwi, Kornel, Lu.nemec, xetra11
30		adelarsq, Andy Hayden, aSpex, Aurora0001, Cormac O'Brien, Lukas Kalbertodt, mcarton, mnoronha, xea
31		Aurora0001, vaartis
32		Kornel, xea
33		Aurora0001, Cormac O'Brien, IceyEC, JDemler, Jean Pierre Dudey, kennytm, Lu.nemec, mcarton
34		BookOwl
35		Ameo, Aurora0001, Hauleth
36		4444, Jon Gjengset, letmutx
37		Aurora0001, John, kennytm, Kornel, Timidger, vaartis, Winger Sendon
38	“1.1”	Vi.
39		John, mmstick, SplittyDev
40		Arrem, Aurora0001, Bo Lu, Charlie Egan, Cormac O'Brien, David Grayson, Enigma, John, Lukas Kalbertodt
41		Cormac O'Brien, Matthieu M.

42		Aurora0001, John, Matthieu M.
43		Aurora0001, Chris Emerson, Hauleth, John, Lukas Kalbertodt, Matt Smith, Shepmaster
44		antoyo, Arrem, Aurora0001, fxlae, Kornel, letmutx, mcarton, Shepmaster
45		4444, Aurora0001, Phil J. Laszkowicz
46		Aurora0001, Cldfire, James Gilles, tversteeg
47		Cormac O'Brien, John, kennytm, Winger Sendon, xea
48	lambda	xea
49		Phil J. Laszkowicz
50	Rust	adelarsq, Aurora0001, Leo Tindall, Marco Alka, Matthieu M., s3rvac, Sorona, Timidger