01 初见Rust

Rust工具链

```
rustup update stable
rustup self uninstall
rustup doc //打开本地文档
```

```
cargo new hello
cargo run //先调用了cargo build
cargo clean //清除存在的编译结果
cargo build //默认debug
cargo build --release
cargo check //检查语法错误
```

```
rustc main.rs
./main
```

宏

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

函数

```rust
fn gcd(mut x: u64, mut y: u64) -> u64 {
    assert!(x != 0 && y != 0);
    while y != 0 {
        if x > y {
            let t = x; //类型推断（let t: u64 = x）
            x = y;
            y = t;
        }
        y %= x;
    }
    x
}
```

测试函数

- 仅用于测试，正常情况无需编译

```rust
#[test] //称为attribute
fn test_gcd() {
    assert_eq!(gcd(14, 15), 1);
}
```

```
cargo test
```

命令行参数

```
use std::env;
use std::str::FromStr;

fn main() {
    let mut a = Vec::new();
    for arg in env::args().skip(1) { //获得参数，忽略第一个
        a.push(u64::from_str(&arg).expect("error parsing argument"));
    }

    if a.len() == 0 {
        eprintln!("Usage: gcd NUMBER ..."); //写入标准错误
        std::process::exit(1);
    }

    let mut d = a[0];
    for i in &a[1..] {
        d = gcd1(d, *i);
    }

    println!("{:?} 的最大公约数是 {}", a, d);
}
```

- std：a crate
- env：a module in std
- FromStr：a trait in std::str

```
pub trait FromStr {
    type Err;
    fn from_str(s: &str) -> Result<Self, Self::Err>;
}
```

- Vec：a struct in std::vec；re-exported by std::prelude
- new：a method in Vec

Rust语言之怪现象

- 变量缺省不可被修改，但总可以被覆盖

```
let x = 5;
let x = x + 1;
{
    let x = x * 2;
    //x = 12
}
//x = 6
```

- 代码块具有值

```
let y = {
    let z = x * x;
    z / 2
}
```

02 基础类型

整数类型

- 无符号整数：u8，u16，u32，u64，u128，usize（内存地址宽度）
- 有符号整数：i8，i16，i32，i64，i128，isize（内存地址宽度）

```
123u8, -1680isize
0x0fffffi32, -0o1670isize, 0b1100u8
0x_0f_ffff_i32, -0o_1_670_isize
```

- 推断出多种可能类型，如果i32是其中一种则为i32，否则编译器报错
- ASCII字符字面量

```
b'A' //=65u8
b'\x41' //=65u8
b'\'', b'\\', b'\n'
```

- as类型转换

```
assert_eq!(1000_i16 as u8, 232_u8);
assert_eq!(-10_i8 as u16, 65526_u16); //符号位高位填补
```

- 附着方法

```
2_u16.pow(4)
u16::pow(2, 4)

(-4_i32).abs()
i32::abs(-4)

0b1001_u8.count_ones()
u8::count_ones(0b1001)
```

- 函数调用优先级高于一元前缀操作符优先级

```
assert_eq!((-4_i32).abs(), 4);
assert_eq!(-4_i32.abs(), -4);
```

整数运算溢出

```
let mut num = 1;
loop {
    num *= 2;
}
//debug模式下panic
```

- checked operations

```
loop {
    num = num.checked_mul(2).expect("overflowed");
}
//dubug和release模式下
```

```
assert_eq!(10_u8.checked_add(2), Some(30));
assert_eq!((-128_i8).checked_div(-1), None);
//None.expect("xxx"): 导致panic, 输出xxx
//Some(num).expect("xxx"): 返回num
```

- wrapping operations

  与release模式下的缺省溢出行为完全相同

```
assert_eq!(500_u16.wrapping_mul(500), 53392);
```

- overflowing operations

  wrapping operation下的计算结果

```
assert_eq!(255_u8.overflowing_sub(2), (253, false));
assert_eq!(5_u16.overflowing_shl(17), (10, true)); //17 % 16 = 1
```

- saturating operations

  类型表示范围内距离结果最近的值

  除、取余、位移没有saturating operations

```
assert_eq!(32760_i16.saturating_add(10), 32767);
```

浮点数类型

- f32、f64
- 无法推断则默认为f64

```
31415.926e-4f64
f64::MAX
f64::MIN
f64::INFINITY
f64::NEG_INFINITY
f64::NAN
```

```
println!("{}", f64::NAN == f64::NAN); //false
println!("{}", f64::is_nan(f64::NAN)); //true
```

```
let a = 0.0 / 0.0; //NAN
let b = f64::sqrt(-1.0); //NAN
```

- 常量

```
use std::f64::consts::PI;

println!("{}", PI);
println!("{:/20}", PI); //20位, 不够则补0
```

工具函数size_of

```
println!("{}", std::mem::size_of::<isize>());
```

布尔类型

- 1字节
- as操作符可将bool转为整数，无法将数字转为bool

```
assert_eq!(false as i32, 0);
assert_eq!(true as f32, 1); //error
assert_eq!(1 as bool, true); //error
```

字符类型

- 4字节
- 字面量

```
'\xHH' //ASCII
'\u{HHHHHH}' //Unicode
```

- char转换为整数

```
assert_eq!('👍' as u32, 0x1f44d);
```

- u8转换为char

```
assert_eq!(97_u8 as char, 'a');
assert_eq!(97_u16 as char, 'a'); //error
```

- u32转换为char

```
use std::char

assert_eq!(char::from_u32(0x2764), Some(❤));
assert_eq!(char::from_u32(0x110000), None);
```

元组类型

```
let t = (1, false, 0.1); //类型推断
let t1: (i64, bool, f32) = (t.1, t.2, t.3);

println!("{:?}", t);

let i: usize = 1;
pritln!("{}", t[i]); //error
```

```
let t = (1, false, ); //最后一个值后可添加逗号

let t1: (i32, ) = (1 + 1, ); //1元组最后必须添加逗号

let t0: () = (); //0元组不能出现逗号
```

- 作为函数返回值

```
fn f(x: i32, y: i32) -> (i32, i32) {}

let (x, y) = f(1, 2);
```

```
fn f(x : i32) {} //实际返回0-tuple
```

指针类型

- 引用（Reference)

```
let v: i32 = 123;
let r = &v;
let r1: &i32 = &v;

let v1: i32 = *r; //去引用（dereference）
*r = 456; //error，只读

println!("{:p}", r); //地址
println!("{}", r); //123
```

```
let mut v: i32 = 123;
let r = &mut v;
*r = 456;
```

- Box

  值存在堆中

```
let v = 123;
let b = Box::new(v);
let mut b1: Box<i32> = Box::new(v);
*b1 = 456;
```

- raw pointer

```
let mut x = 10;
let ptr_x = &mut x as *mut i32;
let y = Box::new(20);
let ptr_y = &*y as *const i32;

unsafe {
    *ptr_x += *ptr_y;
}
```

数组、向量、切片

- array 数组

  缺省被放置在栈中

```
let a: [u32; 5] = [0, 1, 2, 3, 4];
let b = [true, false, true];
let c = [0; 100];
```

- vector 向量

  自身在栈中，元素在堆中

```
let v: Vec<i32> = vec![];
let v = vec![1, 2, 3];
let v = vec![0; 10];
v.insert(3, 10);
v.remove(1);
```

```
let mut vec: Vec<u16> = Vec::with_capacity(10);
for i in 0..10 {
    vec.push(i);
}
vec.push(10);
assert!(vec.capacity() >= 11); //内存重分配
```

```
let mut v = vec![];
v.push(1);
assert_eq!(v.pop(), Some(1));
assert_eq!(v.pop(), None);
```

- slice 切片

```
let mut a = [0, 1, 2, 3];
let s = &a[0..2]; //[0, 2)

let s = &a; //不是切片
let s: &[u16] = &a; //是切片
let s = &a[..];
let s = &a[1..];
let s = &a[..3];
```

```
fn f(s: &[u16]) {}

let a = [0, 1, 2];
let v = vec![0, 1, 2];
f(&a);
f(&v);
```

```
//slice上附着的所有方法都适用于array和vector
v.sort();
v.reverse();
```

字符串类型

```
fn main() {
    let s = "\"hello world\"";

    let s = "hello
    world"; //包含每一行前面空格

    let s = "hello\
    world"; //一行

    let s = r"\"; //停止转义操作，无法放置"字符
    let s = r##"\"##; //可以放置"字符
}
```

- 内存中采用UTF-8编码，不同字符编码长度可能不同
- 两种类型字符串：String（特殊的Vec<u8>）、&str（特殊的&[u8]）

```
let v = "hello".to_string();
let v = String::from("world");

let s = &v[1..4];
let l = "hello world"; //类型&str，所引用字符串在内存的只读区域中
```

```
let s = format!("hello {}", "world");
let s = format!("x = {x}", x = 1);
```

```
assert_eq!(["hello", "world"].concat(), "helloworld");
assert_eq!([[1, 2], [3, 4]].concat(), [1, 2, 3, 4]);
assert_eq!(["hello", "world"].join(" "), "hello world");
assert_eq!([[1, 2], [3, 4]].join(&0), [1, 2, 0, 3, 4]);
assert_eq!([[1, 2], [3, 4]].join(&[0, 0][..]), [1, 2, 0, 0, 3, 4]);
```

```
assert_eq!("中文".len, 6);
assert_eq!("中文".chars().count(), 2);
assert_eq!("English".len(), 7);
assert_eq!("English".chars().count(), 7);
```

- mutable String

```
let mut s = String::from("hello");
s.push(' ');
s.push_str("world");
s.insert(5, ' ');
s.insert_str(11, "!!");
```

```
let mut s = String::from("中文");
s.push('串');
s.insert(1, 'E'); //error
```

```
let mut z = String::from("English");
z[0] = 'e'; //error
```

- mutable &str

```
let mut z = String::from("English");
let s = &mut z[0..3];
println!("{}", s.make_ascii_lowercase());
```

- 比较操作符

```
let a = "Dog".to_lowecase() == "dog"; //true
let a = "Dog" != "dog"; //true
let a = "Dog" > "dog"; //false
```

```
let s0 = "th\u{e9}"; //thé
let s1 = "the\u{301}"; //thé
println!("{}", s0 == s1); //false
```

- 其他常用方法

  当在String上调用&str上的方法时编译器会自动把String转换为&str

```
println!("{}", "Hello, world!".contains("world")); //true
println!("{}", "Hello, world!".replace("world", "dog")); //Hello, dog!
println!("{}", " Hello  \n  ".trim() = "Hello"); //true

for word in "Hello world and dog".split("") {
    println!("{}", word);
}

```

- Byte String

  本质是[u8; N]

  String literal的各种语法都适用于Byte String（Raw Byte String的前缀的br）

  String和&str上的方法不适用于Byte String

```
let s = b"GET";
assert_eq!(s, &[b'G', b'E', b'T']);
```

类型别名

```
type Bytes = Vec<u8>;
let a: Bytes = vec![0, 1, 2];
```

用户自定义类型

- struct

```
struct Image {
    size: (usize, usize),
    pixels: Vec<u32>
}
impl Image {
    //type-associated function
    fn new(w: usize, h: usize) -> Image {
        Image {
            pixels: vec![0; w * h];
            size: (w, h);
        }
    }
    //value-associated function
    fn sizes(&self) -> (usize, usize) {
        self.size
    }
}

let image = Image {
    pixels: vec![0; width * height],
    size: (width, height)
};
```

- enum

```
#[derive(PartialEq)]
enum Ordering {
    Less,
    Equal,
    Greater
}
fn cmp(a: i32, b: i32) -> Ordering {
    if a < b {
        Ordering::Less
    }
    else if a > b {
        Ordering::Greater
    }
    else {
        Ordering::Equal
    }
}
impl Ordering {
    fn is_eq(self) -> bool {
        if self == Ordering::Equal {
            true
        }
        else {
            false
        }
    }
}
```

```
#[derive(PartialEq)]
enum Color {
    RGB(u8, u8, u8),
    Gray(u8)
}
impl Color {
    fn is_gray(&self) -> bool {
        match self {
            Color::Gray(_) => true,
            Color::RGB(a, b, c) =>
```

```
                if a == b && b == c {
                    true
                }
                else {
                    false
                }
        }
    }
}
```

```
//std::option::Option
pub enum Option<T> {
    None,
    Some(T),
}
fn divide(x: f64, y: f64) -> Option<f64> {
    if y == 0.0 {
        None
    }
    else {
        Some(x / y)
    }
}
```

Rust关于Memory的若干基本概念

- Value

  Type + Byte Representation

  Independent of where the value is stored

- Place

  A location in memory that can hold a value

  can be on stack, the heap, ...

- Variable

  A place on the stack

  a named value slot on the stack

- Pointer

  A value that holds the address of a place

  That is, a pointer points to a place

```
let x = 5;
let v = &x;
//Value: 5k, &x
//Place: x, v
//Variable: x, v
//Pointer: &x

//let x = vec![0, 1, 2];
//let y = &x[1..3];
```

03 所有权与所有权转移

示例

- C++

```
std::string *ptr;
{
    std::string s = "Hello world";
    ptr = &s;
}
//无法访问变量s
std::cout << *ptr; //可以访问到s原本空间
```

- Rust

```
let ptr: &String;
{
    let s = String::from("Hello world");
    ptr = &s; //error
}
//无法访问变量s
println!("{}", ptr);
```

Rust中的所有权

- 在任意时刻

  1、一个值具有唯一一个所有者

  2、每一个变量，作为根节点，出现在一棵所有权关系树中

  3、当一个变量离开当前作用域后，它所有权关系树中的所有值都无法再被访问，其中所有存在堆中的值所占空间会被自动释放

- 扩展/软化措施

  1、所有权转移

  2、简单变量豁免

  3、引用计数指针类型

  4、borrow a ref to a value

所有权转移

- 对no-copy type的值，发生如下操作时

  1、赋给一个变量

  2、作为参数传入函数调用

  3、在函数调用中返回

```
let s = vec![1, 2, 3];
let t = s; //s栈空间的值拷贝到t的栈空间
let u = s; //error
```

- Python：赋值成本低（增加引用计数），内存管理成本高（运行时垃圾回收、循环引用难处理）

  C++：赋值成本高（深层复制），内存管理成本低

Rust：赋值成本低（近拷贝栈空间），内存管理成本低

```
let s = vec![1, 2, 3];
let t = s.clone(); //实现C++赋值行为
```

```
let mut s = String::from("abc");
s = String::from("def"); //原来堆空间释放
```

- 条件语句

  若变量有可能在某一个分支被剥夺所有权，即使运行没经过该分支也不能读该变量

```
let x = vec![1, 2, 3];
let c = 1;
if c < 0 {
    f(x);
}
println!("{:?}", x); //error
```

- 循环语句

```
let x = vec![1, 2, 3];
let mut len = x.len();
while len > 0 {
    f(x); //error
    len -= 1;
}
```

```
let mut x = vec![1, 2, 3];
let mut len = x.len();
while len > 0 {
    f(x);
    x = vec![1, 2, 3];
    len -= 1;
}
```

- 数组、向量、切片

  不允许仅通过赋值把某位置上元素的所有权转移

  多数情况不必转移所有权，取得元素的引用可能就足够

```
let mut v = vec![String::from("abc"), String::from("def"), String::from("ghi"),
String::from("jkl")];
let x = v[1]; //error
```

```
//从向量中，成本高
let e = v.remove(1);
println!("{:?}", v); //["abc", "ghi", "jkl"]
```

```
//从向量中
let e = v.swap_remove(1);
println!("{:?}", v); //["abc", "jkl", "ghi"]
```

```
//从向量中
let e = v.pop().expect("empty");
println!("{:?}", v); //["abc", "def", "ghi"]
```

```
//从向量/数组/切片中
let e = std::mem::replace(&mut v[1], String::from("dog"));
println!("{:?}", v); //["abc", "dog", "ghi", "jkl"]
```

```
//必须是具有缺省值的类型
let e = std::mem::take(&mut v[1]);
println!("{:?}", v); //["abc", "", "ghi", "jkl"];
```

```
//显示标注是否有值
let mut v = vec![Some(String::from("abc")), Some(String::from("def"))];
let e = std::mem::take(&mut v[1]);
println!("{:?}", v); //[Some("abc"), None]
println!("{:?}", e); //Some("def")
```

- 向量/数组的所有权转移给循环语句

```
for s in v {
    println!("{}", s);
    //不能读取v
}
//不能读取v
```

Copy Types

- 语言自带的所有数字类型（整数、浮点数），char/bool，若干其他类型，元素类型为Copy Type的数组，所有元素类型均为Copy Type的元组
- 用户自定义的数据类型缺省情况下都不是Copy Type

```
let n1 = 5;
let n2 = n1; //栈中新的空间
```

- Copy Types与自定义数据类型

  如果struct类型包含的所有分量类型都是Copy Type，那么可以通过attribute将该类型声明为Copy Type

```
struct C { x: u32 }
let l = C { x: 3 };
f(l);
println!("{}", l.x); //error
```

```
#[derive(Copy, Clone)]
struct C { x: u32 }
let l = C { x: 3 };
f(l);
println!("{}". l.x); //3
```

```
#[derive(Copy, Clone)]
struct C { x: u32, s: String }
let l = C { x: 3, s: String::from("dog") }; //error
f(l);
println!("{}". l.x);
```

共享所有权

```
use std::rc::Rc;

//可以不必写类型声明
let s: Rc<String> = Rc::new(String::from("dog"));
let t: Rc<String> = s.clone(); //Method-call syntax
let u: Rc<String> = Rc::clone(&s); //Fully qualified syntax

//可以在Rc<T>类型的值上直接调用T类型的值上的方法

println!("{}", RC::strong_count(&s)); //3

let t = 0;
let u = 1;

println!("{}", RC::strong_count(&s)); //3
```

- 被Rc拥有的值不可修改

```
let s = Rc::new(String::from("dog"));
s.push_str("!"); //error
```

- Rc：non-thread-safe，速度快

  Arc：thread-safe，速度慢
- 使用建议：始终用Rc，除非编译器告诉用Arc（多线程环境下使用Rc会被编译器检查出来）