Rust 基础
github.com/ocnyang

1. 变量绑定

- 不可变
  - `let x = 5;`
- 可变
  - `let mut x = 6;`
- 定义后未使用的变量，会有警告，若要消除警告可以以下划线命名。
- 常量
  - `const MAX_POINTS: u32 = 100_000;`
  - 编译时，确定值
- 解构赋值(模式匹配)
  - `let (a, mut b): (bool,bool) = (true, false);`

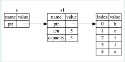
  -
    ```
    let (a, b, c, d, e);
    (a, b) = (1, 2);
    [c, .., d, _] = [1, 2, 3, 4, 5];
    Struct { e, .. } = Struct { e: 5 };
    ```
- 变量遮蔽
  - 后面声明的变量名，可以和前面已声明过的变量名重复，且类型可以不相同；只是后面在使用时，使用的皆是最后一次声明的变量值。

2. 数据类型

- 基本类型
  - 数值类型：有符号整数 (i8, i16, i32(默认), i64, isize)、
无符号整数 (u8, u16, u32, u64, usize) 、
浮点数 (f32, f64(默认))、以及有理数、复数
字 符 串：字符串字面量和字符串切片 &str
布尔类型：true 和 false
字符类型：表示单个 Unicode 字符，存储为 4 个字节
单元类型：即 () ，其唯一的值也是 ()
  - 变量的在编译期是确定的，如果无法推导出来时，需要显式标明
    - `let guess: i32 = "42".parse().expect("Not a number!");`
    - `let guess = "42".parse::<i32>().expect("Not a number!");`
  - 几种类型标注方式
    ```
    let a: i32 = 315;
    let b = 3i32;
    let c = 4.0_f32;
    ```
  - 整型注意溢出问题，debug 模式会监测溢出，--release 会忽略监测
显示处理溢出的方法：
wrapping_*（wrapping_add）、checked_*、overflowing_*、saturating_*、
  - 浮点数陷阱：
计算机并不能严格的（近似值）表示浮点数，
所以：浮点数的相等性比较不靠谱。
可以考虑：`(0.1_f64 + 0.2 - 0.3).abs() < 0.00001`
  - NaN：表示计算得到的数学上未定义的结果
    ```
    let x = (-42.0_f32).sqrt();
      if x.is_nan() {
        println!("未定义的数学行为")
      }
    ```
  - 序列：用来生产连续的数字（或字符）
    ```
    1..5 // 不包含 5
    1..=5 // 包含 5
    ```
  - 类型转换：`3i32 as f32;`
类型转换必须是显式的，不会自动转换
  - 高精度、有理数和复数、货币相关场景，可以使用 num 库
- 字符、布尔、单元类型
  - char:
Rust 的字符不仅仅包括 ASCII，还包括所有的 Unicode（中文、日文、韩文、emoji 表情）
占用 4 个字节
`let x = '中'`
  - bool: true \ false 占用一个字节
  - () : 表示单元类型，
可以理解为 Java 的 void、Kotlin 的 Unit、Go 的 struct{}
表示不关注具体的值，函数的空返回值，也可作为map 的值。
不占用任何内存
- 语句和表达式（忽略）
  - 函数体采用最后一行的表达式来作为返回值
  - `let x = 5;` 这是语句，不能作为返回值
  - `x+1` 这是表达式可以作为返回值
  - 简单来分辨就是在编译时后面可不可以不带 ; 的
let 是表达式已是实验功能。
（一般纯函数语言，比如 Elixir，赋值语句[匹配]也有返回值，也就是说不做这两个概念的区分）
- 函数
  - 函数名和变量名使用蛇形命名法：`fn add_two() -> {}`
  - 默认返回值为最后一条表达式；也可以使用 return 提前返回
  - 没有返回值的函数，默认是返回 ()
  - 永不返回的发散函数 !
    - 用作程序崩溃的函数
      ```
      fn dead_end() -> ! {
        panic!("你已经到了穷途末路，崩溃吧！");
      }
      ```

    -
      ```
      fn forever() -> ! {
        loop {
          //...
        };
      }
      ```
  - 函数的类型表示方式：fn(i32) -> i32
    - 函数变量
      ```
      let v = {}; // v 的类型为()，即没有返回值
      let v = {9}; // v 的类型为 i32
      以上这种是类型为：代码块最后一行的返回值

      let v = ||{}; // v 的类型为 fn()
      let v = |6|{9}; // v 的类型为 fn(i32)->i32
      以上这种是定义的匿名函数；

      let v = {||()}; // v 的类型为 fn()
      ```

3. 所有权和借用

- 所有权
  - 1. Rust 中每一个值都被一个变量所拥有，该变量被称为值的所有者；
2. 一个值同时只能被一个变量所拥有，或者说一个值只能拥有一个所有者；
3. 当所有者 (变量) 离开作用域范围时，这个值将被丢弃 (drop)；
  - 对于基本类型、仅包含基本类型的元组、不可变引用 &T，是可以 Copy 的，也就是说他们转移的是值，不涉及到所有权（所有权不被转移）问题。
- 借用/引用
  - 

    获取变量的引用，称之为借用.
引用 &x; 解引用 *x;
不可变引用，只是允许使用值，但不会转移所有权
- 可变引用
  - 可变引用的原则，是不能同时存在两个所有权。
即不能同时存在两个可变引用；
也不能同时存在不可变引用和可变引用。
（所有权借用时，不再用时即自动归还。）
所以只要保证时间线上不同时存在两个

    -
      ```
      let mut mm = String::from("ssss");
      let nn = &mut mm;
      println!("{}", *nn);
      println!("{}", mm);
      // 可以
      ```

    -
      ```
      let mut mm = String::from("ssss");
      let nn = &mut mm;
      println!("{}", mm);
      println!("{}", *nn);
      // 不可以
      ```

4. 复合类型

- 切片 slice
  - 切片是对集合的部分引用，
引用内容包括开始下标+长度。
  - String 的切片，类型标识 &str

    -
      ```
      let s = String::from("hello");

      let slice = &s[0..2];
      let slice = &s[..2];

      let len = s.len();
      let slice = &s[4..len];
      let slice = &s[4..];

      let slice = &s[..];
      ```

    -
      ```
      let s = "中国人";
      let a = &s[0..2];
      println!("{}",a);

      // 会崩溃，因为中文在 UTF-8 中占用三个字节
      ```
  - 集合的切片。例如数组的切片类型为 &[i32]

    -
      ```
      let a = [1, 2, 3, 4, 5];

      let slice = &a[1..3];
      ```
  - 字符串字面量是切片（&str，不可变引用）：
`let s: &str = "Hello, world!";`
- 字符串
- 元组
  - 多种类型组合到一起，长度是固定的，元素顺序也是固定的。常用于函数返回多个值

    -
      ```
      let x: (i32, f64, u8) = (500, 6.4, 1);

      let five_hundred = x.0;
      ```

    -
      ```
      let tup = (500, 6.4, 1);
      // 模式匹配结构
      let (x, y, z) = tup;
      ```
- 结构体
  - 初始化时，每个字段都要初始化；
初始化时不要求顺序；

    -
      ```
      struct User {
          active: bool,
          username: String,
          email: String,
          sign_in_count: u64,
      }
      ```

    -
      ```
      let email = String::from("");
      let user = User {
          email, // 变量名一致
          username: String::from("z3"),
          active: true,
          sign_in_count: 1,
      };
      ```
  - 更新字段语法 简化

    -
      ```
      let user2 = User {
          email: String::from("a@b.c"),
          ..user
      };
      ```
    - 老对象中的非 Copy 字段的所有权会转移到新对象中；
老对象的 Copy 字段可以继续获取使用，
非 Copy 字段不能再使用
  - 元组结构体
(不关心字段名字)

    -
      ```
      struct Color(i32, i32, i32);
      struct Point(i32, i32, i32);

      let black = Color(0, 0, 0);
      let origin = Point(0, 0, 0);
      使用：
      let Color(r,g,b) = black;
      let red = black.0;
      ```
  - 单元结构体
(不关心内容，关心行为)

    -
      ```
      struct AlwaysEqual;
      impl SomeTrait for AlwaysEqual {
          ...
      }
      ```
  - 打印
    - 使用 #[derive(Debug)] 标记后；
`println!("rect1 is {:?}", rect1);`
    - {:#?} 来替代 {:?}
可以打印立体日志
    - `dbg!(&rect1);`
打印立体日志的同时，携带信息
- 枚举
  - 定义
    ```
    enum PokerSuit {
      Spades,
      Hearts,
    }
    ```
    使用
    ```
    let heart = PokerSuit::Hearts;
    ```
  - 携带值
    ```
    enum Message {
        Quit,
        Move { x: i32, y: i32 },
        Write(String),
        ChangeColor(i32, i32, i32),
    }
    // 可以携带任何类型，包括结构体、枚举
    ```
  - 同一化类型
⚠️ 不懂

    -
      ```
      enum Websocket {
        Tcp(Websocket<TcpStream>),
        Tls(Websocket<native_tls::TlsStream<TcpStream>>),
      }
      ```
  - Option 枚举

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

    -
      ```
      let some_number = Some(5);
      let some_string = Some("a string");

      let absent_number: Option<i32> = None;
      ```

    -
      ```
      fn plus_one(x: Option<i32>) -> Option<i32> {
          match x {
              None => None,
              Some(i) => Some(i + 1),
          }
      }
      ```
- 数组
  - 固定数组 array
类型相同；长度固定；
线性排列；存储栈上；

    -
      ```
      let a = [1, 2, 3, 4, 5];
      // 显式类型
      let a: [i32; 5] = [1, 2, 3, 4, 5];
      // 某个值重复 N 次
      let a = [3; 5];
      // 访问
      let first = a[0];
      ```
  - 非基础类型数组

    -
      ```
      let array = [String::from("rust is good!"); 8];
      // 会报错，因为 String 是不可 Copy 的
      ```

    -
      ```
      let array: [String; 8] = std::array::from_fn(|_i| String::from("rust is good!"));
      // 正确写法
      ```
  - 数值切片

    -
      ```
      let a: [i32; 5] = [1, 2, 3, 4, 5];

      let slice: &[i32] = &a[1..3];
      ```
    - 数组类型：[i32;5]
切片类型：&[i32]
切片创建代价小
  - 数组类型：[值类型;长度]
    - [i32;2] 和 [i32;3] 是不同的数组类型，当数组作为参数可以体现；
    - &[i32] 表示任意长度的数组切片；
    - 如果想表示任意长度的数组，可以通过 const 泛型实现；
- 动态数组 Vector
  - 类型用 Vec<T> 表示，其实就是其他语言中的 List。
长度不固定，存储相同类型，占用内存连续，堆存储。
  - 创建方式
    - `let v: Vec<i32> = Vec::new();`

    -
      ```
      let mut v = Vec::new();
      v.push(1);
      ```
    - `let v = vec![1, 2, 3];`
    - 创建固定长度：
`Vec::with_capacity(capacity)`
// 减少了长度的变更，性能略好
  - vec! 是宏语句，所以
vec![] 也可以使用 vec!() 、vec!{} 相互等价
  - 读取元素
    - 通过下标：
      ```
      let v = vec![1, 2, 3, 4, 5];

      let third: &i32 = &v[2];
      ```
    - 通过 get:
      ```
      match v.get(2) {
          Some(third) => println!("{third}"),
          None => println!("根本没有！"),
      }
      ```
    - 读取/借用多个元素时，遵从所有权规则：
同一个时间节点只能存在一个，不交叉。
    - 遍历：
      ```
      let v = vec![1, 2, 3];
      for i in &v {
          println!("{i}");
      }
      ```
      可变遍历：
      ```
      let mut v = vec![1, 2, 3];
      for i in &mut v {
          *i += 10
      }
      ```
  - 存储不同类型的元素
    - 枚举方式：
      ```
      #[derive(Debug)]
      enum IpAddr {
          V4(String),
          V6(String)
      }
      fn main() {
          let v = vec![
              IpAddr::V4("127.0.0.1".to_string()),
              IpAddr::V6("::1".to_string())
          ];

          for ip in v {
              show_addr(ip)
          }
      }

      fn show_addr(ip: IpAddr) {
          println!("{:?}",ip);
      }
      ```
    - 特征对象方式（更灵活，推荐）：
      ```
      trait IpAddr {
          fn display(&self);
      }

      struct V4(String);
      impl IpAddr for V4 {
          fn display(&self) {
              println!("ipv4: {:?}",self.0)
          }
      }
      struct V6(String);
      impl IpAddr for V6 {
          fn display(&self) {
              println!("ipv6: {:?}",self.0)
          }
      }

      fn main() {
          let v: Vec<Box<dyn IpAddr>> = vec![
              Box::new(V4("127.0.0.1".to_string())),
              Box::new(V6("::1".to_string())),
          ];

          for ip in v {
              ip.display();
          }
      }
      ```
  - 排序
    - 稳定的排序 sort 和 sort_by；
非稳定排序 sort_unstable 和 sort_unstable_by；
    - 浮点数排序：
`vec.sort_unstable_by(|a, b| a.partial_cmp(b).unwrap());`
- 哈希映射
  - 类型 HashMap<K,V> ；
长度动态，K/V 都要求类型相同，堆存储。
  - 创建方式

    -
      ```
      let mut my_gems = HashMap::new();
      my_gems.insert("红宝石", 1);
      ```
    - 创建固定长度：
`HashMap::with_capacity(capacity)`
    - 元组的动态数组转换：
      ```
      let teams_list = vec![("中国队".to_string(), 100),];
      let teams_map: HashMap<_,_> = teams_list.into_iter().collect();
      ```
  - 遵从所有权规则：
    ```
    let name = String::from("Sunface");
    let age = 18;

    let mut handsome_boys = HashMap::new();
    handsome_boys.insert(name, age);
    // name 将不在可用，因为它的所有权已转到 HashMap
    ```
  - 查询

    -
      ```
      let mut scores = HashMap::new();

      scores.insert(String::from("Blue"), 10);
      scores.insert(String::from("Yellow"), 50);

      let team_name = String::from("Blue");
      let score: Option<&i32> = scores.get(&team_name);
      // get 方法返回一个 Option<&i32> 类型：当查询不到时，会返回一个 None，查询到时返回 Some(&i32)
      // &i32 是对 HashMap 中值的借用，如果不使用借用，可能会发生所有权的转移
      // 若想获取值类型：let score: i32 = scores.get(&team_name).copied().unwrap_or(0);
      ```
    - 遍历
`for (key, value) in &scores {}`
  - 更新

    -
      ```
      // 覆盖已有的值
      let old = scores.insert("Blue", 20);

      // 查询对应的值，若不存在则插入新值
      let v = scores.entry("Yellow").or_insert(5);
      ```
    - 统计单词出现次数
      ```
      let text = "hello world wonderful world";

      let mut map = HashMap::new();
      // 根据空格来切分字符串(英文单词都是通过空格切分)
      for word in text.split_whitespace() {
          let count = map.entry(word).or_insert(0);
          *count += 1;
      }
      ```
  - 可选高性能哈希函数库；
默认用的：SipHash；

5. 流程控制

- if else

  -
    ```
    if n % 4 == 0 {
      ...
    } else if n % 3 == 0 {
      ...
    }
    ```
- for

  -
    ```
    for i in 1..=5 {...}

    for item in &container {...}

    for item in &mut collection {...}

    for (i, v) in a.iter().enumerate() {...}

    for _ in 0..10 {...}
    ```

  -
    ```
    for i in 0..collection.len() {
      let item = collection[i];
    }
    ```
    不推荐，会有性能损耗，也易出问题

-
  ```
  continue; // 跳出本次循环
  break; // 跳出、结束整个循环
  ```
- while
  ```
  while n <= 5  {
      ...
      n = n + 1;
  }
  ```
- loop
  ```
  let result = loop {
      counter += 1;

      if counter == 10 {
          break counter * 2;
      }
  };
  // 要靠自己判断条件，然后 break 结束
  ```

6. 方法、泛型、特征

- 方法
  - 定义：和结构体是分开的；
    - 方法的第一个参数是自身：&self ；
    - 方法名可以与字段名相同，常用于私有字段的 getter；
    - 方法可以有多个参数；
    - 不携带 &self 参数的方法，称为 关联函数，比如常用于构造函数的 new ；
    - 可以使用多个 impl 定义，便于代码组织不同方法；
    - 枚举、特征（trait）都可以拥有方法；
    ```
    pub struct Rectangle {
        width: u32,
        height: u32,
    }

    impl Rectangle {
        pub fn new(width: u32, height: u32) -> Self {
            Rectangle { width, height }
        }
        pub fn width(&self) -> u32 {
            return self.width;
        }
    }
    ```
  - self、&self、&mut self 皆可使用；
同样遵从所有权规则；
使用 self 转移所有权的方式少见，常用来转换成新对象使用，而旧对象不再允许使用；
  - 可见行：pub 关键字来决定模型、类型、函数、方法是公开的；默认情况下是私有的；
- 泛型
  - Rust 的泛型可以理解为它是在编译期时，对使用的地方（调用）进行的代码替换；所以它是零成本，不会牺牲性能，但牺牲了编译速度；
  - 类型限制
    ```
    fn add<T: std::ops::Add<Output = T>>(a:T, b:T) -> T {
        a + b
    }
    ```
  - 枚举泛型特例
    - 空值类型
      ```
      enum Option<T> {
          Some(T),
          None,
      }
      ```
    - 结果是否正确
      ```
      enum Result<T, E> {
          Ok(T),
          Err(E),
      }
      ```
  - 对结构体的具体泛型类型实现方法
    ```
    结构体：Point<T>
    impl Point<f32> {
        fn distance_from_origin(&self) -> f32 {
            ...
        }
    }
    ```
  - const (值)泛型
    ```
    任意长度的数组
    fn display_array<T: std::fmt::Debug, const N: usize>(arr: [T; N]) {
        println!("{:?}", arr);
    }
    ```
  - const 泛型表达式 ⚠
- 特征 Trait
  - 类似其他语言中的接口
(和 Elixir 中的协议基本相同)
    - 定义
      ```
      pub trait Summary {
          fn summarize(&self) -> String;
      }
      ```
    - 实现方式（Post 是结构体）
      ```
      impl Summary for Post {
          fn summarize(&self) -> String {
              ...
          }
      }
      ```
  - 孤儿规则：如果想要为类型 A 实现特征 T，那么 A 或者 T 至少有一个是在当前作用域中定义的。
  - 默认实现：可以在定义特征时提供默认的实现内容，然后 impl Summary for Post {}
  - 特征作为函数参数
    ```
    pub fn notify(item: &impl Summary) {
        println!("Breaking news! {}", item.summarize());
    }
    // 这其实是一种语法糖，实际上等同于下面的 泛型约束
    ```
  - 特征作为泛型约束
    ```
    pub fn notify<T: Summary>(item: &T) {
        println!("Breaking news! {}", item.summarize());
    }
    ```
    ```
    pub fn notify(item1: &impl Summary, item2: &impl Summary) {}
    // 两个参数类型可以不同

    pub fn notify<T: Summary>(item1: &T, item2: &T) {}
    // 两个参数上相同类型
    ```
  - 多重约束
    ```
    方式一：
    pub fn notify(item: &(impl Summary + Display)) {}
    方式二：
    pub fn notify<T: Summary + Display>(item: &T) {}
    ```
  - Where 约束
    ```
    多重约束的另一种写法：
    fn some_function<T, U>(t: &T, u: &U) -> i32
        where T: Display + Clone,
                  U: Clone + Debug
    {}
    ```
  - 使用特征约束有条件地实现方法或特征
    ```
    use std::fmt::Display;
    struct Pair<T> {
        x: T,
        y: T,
    }

    impl<T: Display + PartialOrd> Pair<T> {
        fn cmp_display(&self) {
            ...
        }
    }
    // 只有满足这两个特征的泛型实例，才能使用这个方法
    ```
  - 有条件地实现特征
    ```
    impl<T: Display> ToString for T {
        // --snip--
    }
    ```
    ```
    其实也可以无条件实现特征，这样所有的类型都可以调用：
    impl<T> ToString for T {
        // --snip--
    }
    // 此实现后，不支持共存其他类型的实现，所以无意义
    ```
  - 特征作为返回值
    ```
    fn returns_summarizable() -> impl Summary {
        Weibo {...}
    }
    // 但这里有一个限制，你不能在内部通过判断返回不同的类型（两种不同的实现了特征的类型）
    ```
  - 通过 derive 派生特征
    ```
    #[derive(Debug)]
    #[derive(Copy)]
    ```
- 特征对象

7. 模式匹配

- match

  -
    ```
    let dire = Direction::South;// menu
    match dire {
        Direction::East => println!("East"),
        Direction::North | Direction::South => {
            println!("South or North");
        },
        Direction::Quarter(state) => {
            println!("{:?}!", state);
        },
        _ => println!("West"),
    };
    // 必须穷尽所有可能
    // | 匹配多个选项
    // 可以取值
    // _ 或 other 匹配剩下的可能项
    // 1..=5 可以匹配1到5
    ```

  -
    ```
    守卫条件
    let num = Some(4);

    match num {
        Some(x) if x < 5 => println!("less than five:
    {}", x),
        Some(x) => println!("{}", x),
        None => (),
    }
    ```

  -
    ```
    @绑定
    let x = 60;
    let y = false;

    match x {
        10 | 11 => println!("x is 10, 11"),
        1..=9 => println!("x is in range [1, 9]"),
        id @ 12..=20 => println!("x in range [12, 20], id is {}", id),
        n if !y => println!("y is false, x is: {}", n),
        _ => println!("nothing match"),
    }
    ```

  -
    ```
    let values = vec![1, 2, 3];

    {
        let result = match IntoIterator::into_iter(values) {
            mut iter => loop {
                match iter.next() {
                    Some(x) => { println!("{}", x); },
                    None => break,
                }
            },
        };
        result
    }

    ps: 感觉越来越类似 elixir 的概念了，魅力十足的模式匹配
    ```
- if let 匹配一个条件

  -
    ```
    let v = Some(3u8);
    if let Some(3) = v {
        println!("three");
    }
    // 可以结合 else
    ```

  -
    ```
    if let Some(i) = v {
        // 用来取值
    } else if ... {
    } else {
    }
    ```
- let
  ```
  let x = 1;
  let option_x = Some(x);
  (刚学时没有这个概念，大概是 rust 后面更新的)
  这里可以理解为 rust 已经和 elixir 理念相同，
  即没有赋值语句，所有的 = 都是模式匹配，
  如果左侧的变量无值，会将右值匹配到左侧。
  ```
- while let
  ```
  let mut nums = vec![1,2,3];
  while let Some(num) = nums.pop() {
      println!("num in while let is : {}", num)
  }
  ```
- matches! 宏
  ```
  let foo = 'f';
  assert!(matches!(foo, 'A'..='Z' | 'a'..='z'));

  let bar = Some(4);
  assert!(matches!(bar, Some(x) if x > 2));
  // ⚠ 第二种不理解
  ```

8. 错误处理

- panic
  - 触发方式：
    - 程序运行出现错误；
    - 主动触发 panic!("crash");
  - 触发时，对调用栈展开，
通过下面方式运行查看错误详细信息：
RUST_BACKTRACE=1 cargo run
  - 在配置里设置 panic 时退出方式：
默认：栈展开，打印错误信息；
直接终止：在 Cargo.toml 文件中配置，(减少执行文件大小)
    ```
    [profile.release]
    panic = 'abort'
    ```
  - 终止逻辑
在 main 线程，程序终止；
在子线程中，只终止子线程；
  - 函数返回 Result 处理
    - upwrap()
如果返回成功就取出值；
如果失败就直接 panic;
`let f = File::open("hello.txt").unwrap();`
    - expect("Fail info")
和 upwrap 相同，只是 panic 时会打印出参数；
    - 自定义解决
      ```
      let f = File::open("hello.txt");

      let f = match f {
          Ok(file) => file,
          Err(error) => match error.kind() {
              ErrorKind::NotFound => match File::create("hello.txt") {
                  Ok(fc) => fc,
                  Err(e) => panic!("creating error: {:?}", e),
              },
              other_error => panic!("opening error: {:?}", other_error),
          },
      };
      ```
- ?
  - 如果失败，就直接返回（向上传播）Error
如果成功，就取出值

? 上抛错误时，会对错误类型自动转换

    -
      ```
      fn read_username_from_file() ->
      Result<String, io::Error> {
          let mut f = File::open("hello.txt")?;
          let mut s = String::new();
          f.read_to_string(&mut s)?;
          Ok(s)
      }
      ```

    -
      ```
      简化（链式调用）
      fn read_username_from_file() -> Result<String, io::Error> {
          let mut s = String::new();
          File::open("hello.txt")?.read_to_string(&mut s)?;
          Ok(s)
      }
      ```

    -
      ```
      上面等价于：
      fn read_username_from_file() ->
      Result<String, io::Error> {
          // 打开文件，f是`Result<文件句
      柄,io::Error>`
          let f = File::open("hello.txt");

          let mut f = match f {
              // 打开文件成功，将file句柄赋值给f
              Ok(file) => file,
              // 打开文件失败，将错误返回(向上传播)
              Err(e) => return Err(e),
          };
          // 创建动态字符串s
          let mut s = String::new();
          // 从f文件句柄读取数据并写入s中
          match f.read_to_string(&mut s) {
              // 读取成功，返回Ok封装的字符串
              Ok(_) => Ok(s),
              // 将错误向上传播
              Err(e) => Err(e),
          }
      }
      ```
  - 传播 Option.None
和传播 Result.Error 相同
  - main 函数返回 () 不能使用 ?
    ```
    解决方式，main 函数还可以这样写：
    fn main() -> Result<(), Box<dyn Error>> {
        let f = File::open("hello.txt")?;

        Ok(())
    }
    ```

9. 生命周期

```
fn longest<'a>(x: &'a str, y: &str) -> &'a str {
    x
}
```

10. 注释&文档

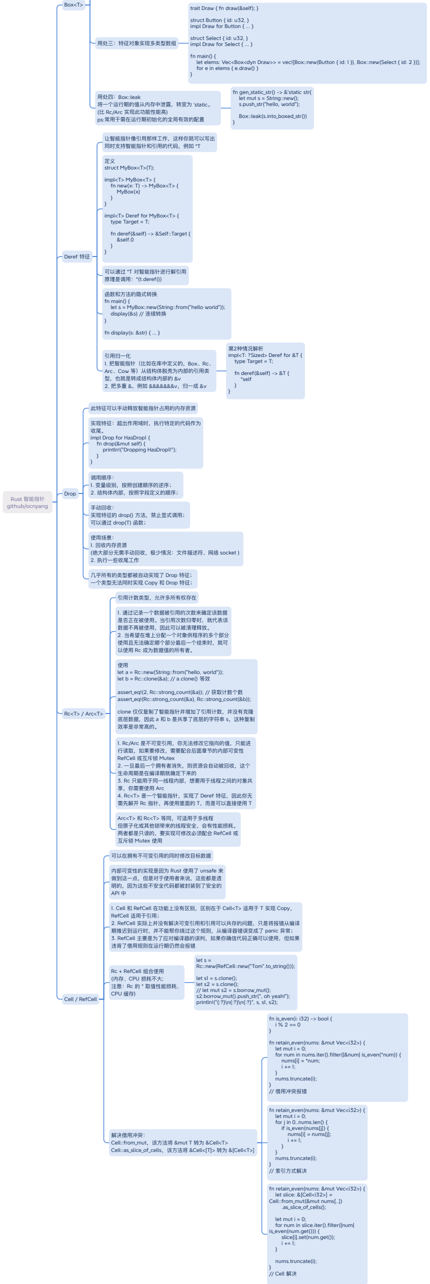
- 代码注释
  - 行注释 //
  - 块注释 /*...*/
- 文档注释
  - 行注释 ///
  - 块注释 /**...**/
- 包、模块注释
  - 行注释 //!
  - 块注释 /*!...*/
- 文档
  - cargo doc 可以直接生成 HTML 文件，放入 target/doc 目录下
cargo doc --open 可以生产后自动打开网页
- 文档格式，支持 markdown
- 文档常用标题[约定惯例]
  - markdown 标题 #XXX
  - Panics：函数可能会出现的异常状况，这样调用函数的人就可以提前规避
  - Errors：描述可能出现的错误及什么情况会导致错误，有助于调用者针对不同的错误采取不同的处理方式
  - Safety：如果函数使用 unsafe 代码，那么调用者就需要注意一些使用条件，以确保 unsafe 代码块的正常工作
- 注释代码跳转

  -
    ```
    /// [`Receiver<T>`]   [`std::future`].
    ///
    ///  [`std::future::Future`] [`Self::recv()`].
    ```

  -
    ```
    同名项，可以标示类型
    /// 跳转到结构体  [`Foo`](struct@Foo)
    pub struct Bar;

    /// 跳转到同名函数 [`Foo`](fn@Foo)
    pub struct Foo {}

    /// 跳转到同名宏 [`foo!`]
    pub fn Foo() {}

    #[macro_export]
    macro_rules! foo {
      () => {}
    }
    ```
- 文档搜索别名
  ```
  #[doc(alias = "x")]
  #[doc(alias = "big")]
  pub struct BigX;

  #[doc(alias("y", "big"))]
  pub struct BigY;
  ```

- 可以将值分配到堆上，然后在栈上保留一个智能指针指向堆值。

-
  ```
  let a: Box<i32> = Box::new(3);
  println!("a = {}", a);// Deref 隐式自动解引用
  let b: i32 = *a + 3; // 必须显式解引用取值
  ```
- 用处一：避免栈上数据的拷贝
(所有权的转移是通过复制栈中指针到新变量，废弃旧指针变量)
  ```
  let arr = Box::new([0; 1000]);
  let arr1 = arr;
  ```
- 用处二：将动态大小类型变为 Sized 固定大小类型
  ```
  递归类型
  enum List {
      Cons(i32, Box<List>),
      Nil,
  }
  ```



Rust 智能指针
github/ocnyang

Box<T>

用处三：特征对象实现多类型数组

```
trait Draw { fn draw(&self); }

struct Button { id: u32, }
impl Draw for Button { ... }

struct Select { id: u32, }
impl Draw for Select { ... }

fn main() {
    let elems: Vec<Box<dyn Draw>> = vec![Box::new(Button { id: 1 }), Box::new(Select { id: 2 })];
    for e in elems { e.draw() }
}
```

用处四：Box::leak
将一个运行期的值从内存中泄露，转变为 'static。
(比 Rc/Arc 实现此功能性能高)
ps:常用于需在运行期初始化的全局有效的配置

```
fn gen_static_str() -> &'static str{
    let mut s = String::new();
    s.push_str("hello, world");

    Box::leak(s.into_boxed_str())
}
```

Deref 特征

让智能指针像引用那样工作，这样你就可以写出同时支持智能指针和引用的代码，例如 *T

定义

```
struct MyBox<T>(T);

impl<T> MyBox<T> {
    fn new(x: T) -> MyBox<T> {
        MyBox(x)
    }
}

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

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

可以通过 *T 对智能指针进行解引用
原理是调用：*(t.deref())

函数和方法的隐式转换

```
fn main() {
    let s = MyBox::new(String::from("hello world"));
    display(&s) // 连续转换
}

fn display(s: &str) { ... }
```

引用归一化
1. 把智能指针（比如在库中定义的，Box、Rc、Arc、Cow 等）从结构体脱壳为内部的引用类型，也就是转成结构体内部的 &v
2. 把多重 &，例如 &&&&&&&v，归一成 &v

第2种情况解析

```
impl<T: ?Sized> Deref for &T {
    type Target = T;

    fn deref(&self) -> &T {
        *self
    }
}
```

Drop

此特征可以手动释放智能指针占用的内存资源

实现特征：超出作用域时，执行特定的代码作为收尾。

```
impl Drop for HasDrop1 {
    fn drop(&mut self) {
        println!("Dropping HasDrop1!");
    }
}
```

调用顺序：
1. 变量级别，按照创建顺序的逆序；
2. 结构体内部，按照字段定义的顺序；

手动回收：
实现特征的 drop() 方法，禁止显式调用；
可以通过 drop(T) 函数；

使用场景：
1. 回收内存资源
(绝大部分无需手动回收，极少情况：文件描述符、网络 socket)
2. 执行一些收尾工作

几乎所有的类型都被自动实现了 Drop 特征；
一个类型无法同时实现 Copy 和 Drop 特征；

Rc<T> / Arc<T>

引用计数类型，允许多所有权存在

1. 通过记录一个数据被引用的次数来确定该数据是否正在被使用。当引用次数归零时，就代表该数据不再被使用，因此可以被清理释放。
2. 当希望在堆上分配一个对象供程序的多个部分使用且无法确定哪个部分最后一个结束时，就可以使用 Rc 成为数据值的所有者。

使用

```
let a = Rc::new(String::from("hello, world"));
let b = Rc::clone(&a); // a.clone() 等效

assert_eq!(2, Rc::strong_count(&a)); // 获取计数个数
assert_eq!(Rc::strong_count(&a), Rc::strong_count(&b));
```

clone 仅仅复制了智能指针并增加了引用计数，并没有克隆底层数据，因此 a 和 b 是共享了底层的字符串 s，这种复制效率是非常高的。

1. Rc/Arc 是不可变引用，你无法修改它指向的值，只能进行读取，如果要修改，需要配合后面章节的内部可变性 RefCell 或互斥锁 Mutex
2. 一旦最后一个拥有者消失，则资源会自动被回收，这个生命周期是在编译期就确定下来的
3. Rc 只能用于同一线程内部，想要用于线程之间的对象共享，你需要使用 Arc
4. Rc<T> 是一个智能指针，实现了 Deref 特征，因此你无需先解开 Rc 指针，再使用里面的 T，而是可以直接使用 T

Arc<T> 和 Rc<T> 等同，可适用于多线程
但原子化或其他锁带来的线程安全，会有性能损耗。
两者都是只读的，要实现可修改必须配合 RefCell 或互斥锁 Mutex 使用

Cell / RefCell

可以在拥有不可变引用的同时修改目标数据

内部可变性的实现是因为 Rust 使用了 unsafe 来做到这一点，但是对于使用者来说，这些都是透明的，因为这些不安全代码都被封装到了安全的 API 中

1. Cell 和 RefCell 在功能上没有区别，区别在于 Cell<T> 适用于 T 实现 Copy，RefCell 适用于引用；
2. RefCell 实际上并没有解决可变引用和引用可以共存的问题，只是将报错从编译期推迟到运行时，并不能帮你绕过这个规则，从编译器错误变成了 panic 异常；
3. RefCell 主要是为了应对编译器的误判，如果你确信代码正确可以使用，但如果违背了借用规则在运行期仍然会报错

Rc + RefCell 组合使用
(内存、CPU 损耗不大;
注意：Rc 的 * 取值性能损耗、CPU 缓存)

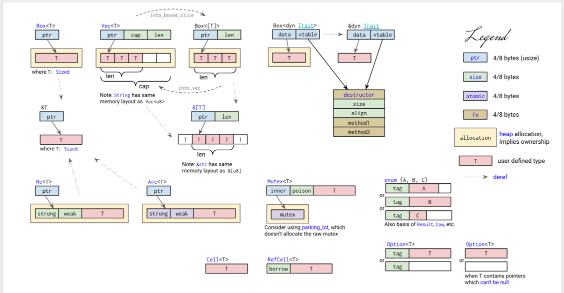
```
let s =
Rc::new(RefCell::new("Tom".to_string()));

let s1 = s.clone();
let s2 = s.clone();
// let mut s2 = s.borrow_mut();
s2.borrow_mut().push_str(", oh yeah!");
println!("{:?}\n{:?}\n{:?}", s, s1, s2);
```

解决借用冲突：
Cell::from_mut，该方法将 &mut T 转为 &Cell<T>
Cell::as_slice_of_cells，该方法将 &Cell<[T]> 转为 &[Cell<T>]

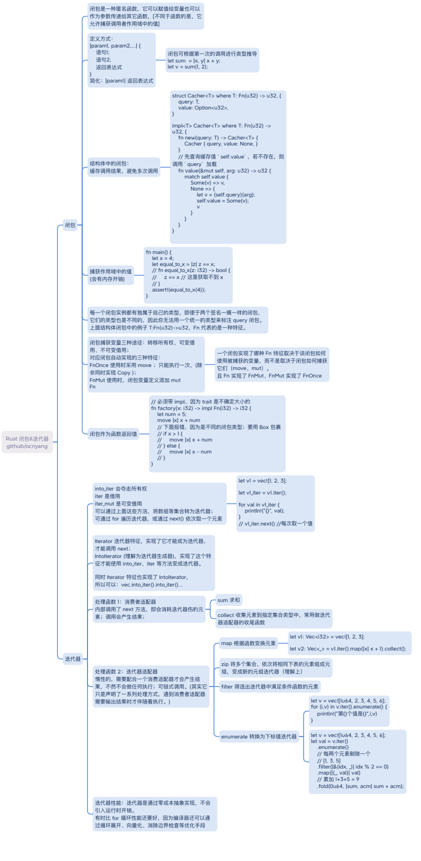
```
fn is_even(i: i32) -> bool {
    i % 2 == 0
}

fn retain_even(nums: &mut Vec<i32>) {
    let mut i = 0;
    for num in nums.iter().filter(|&num| is_even(*num)) {
        nums[i] = *num;
        i += 1;
    }
    nums.truncate(i);
}
// 借用冲突报错
```

```
fn retain_even(nums: &mut Vec<i32>) {
    let mut i = 0;
    for j in 0..nums.len() {
        if is_even(nums[j]) {
            nums[i] = nums[j];
            i += 1;
        }
    }
    nums.truncate(i);
}
// 索引方式解决
```

```
fn retain_even(nums: &mut Vec<i32>) {
    let slice: &[Cell<i32>] =
Cell::from_mut(&mut nums[..])
        .as_slice_of_cells();

    let mut i = 0;
    for num in slice.iter().filter(|num|
is_even(num.get())) {
        slice[i].set(num.get());
        i += 1;
    }

    nums.truncate(i);
}
// Cell 解决
```



Rust container cheat sheet, by Raph Levien, Copyright 2017 Google Inc., released under Creative Commons BY, 2017-04-21, version 0.0.4



Rust 闭包&迭代器
github/ocnyang

闭包

闭包是一种匿名函数，它可以赋值给变量也可以作为参数传递给其它函数，[不同于函数的是，它允许捕获调用者作用域中的值]

定义方式：

```
|param1, param2,...| {
    语句1;
    语句2;
    返回表达式
}
```

简化：|param1| 返回表达式

闭包可根据第一次的调用进行类型推导

```
let sum  = |x, y| x + y;
let v = sum(1, 2);
```

结构体中的闭包：
缓存调用结果，避免多次调用

```
struct Cacher<T> where T: Fn(u32) -> u32, {
    query: T,
    value: Option<u32>,
}

impl<T> Cacher<T> where T: Fn(u32) ->
u32, {
    fn new(query: T) -> Cacher<T> {
        Cacher { query, value: None, }
    }
    // 先查询缓存值 `self.value`，若不存在，则
调用 `query` 加载
    fn value(&mut self, arg: u32) -> u32 {
        match self.value {
            Some(v) => v,
            None => {
                let v = (self.query)(arg);
                self.value = Some(v);
                v
            }
        }
    }
}
```

捕获作用域中的值
(会有内存开销)

```
fn main() {
    let x = 4;
    let equal_to_x = |z| z == x;
    // fn equal_to_x(z: i32) -> bool {
    //     z == x // 这里获取不到 x
    // }
    assert!(equal_to_x(4));
}
```

每一个闭包实例都有独属于自己的类型，即使于两个签名一模一样的闭包，它们的类型也是不同的，因此你无法用一个统一的类型来标注 query 闭包。
上面结构体闭包中的例子 T:Fn(u32)->u32，Fn 代表的是一种特征。

闭包捕获变量三种途径：转移所有权、可变借用、不可变借用；
对应闭包自动实现的三种特征：
FnOnce 使用时采用 move ；只能执行一次，(除非同时实现 Copy)；
FnMut 使用时，闭包变量定义添加 mut
Fn

一个闭包实现了哪种 Fn 特征取决于该闭包如何使用被捕获的变量，而不是取决于闭包如何捕获它们（move、mut）。
且 Fn 实现了 FnMut、FnMut 实现了 FnOnce

闭包作为函数返回值

```
// 必须带 impl，因为 trait 是不确定大小的
fn factory(x: i32) -> impl Fn(i32) -> i32 {
    let num = 5;
    move |x| x + num
    // 下面报错，因为是不同的闭包类型；要用 Box 包裹
    // if x > 1 {
    //     move |x| x + num
    // } else {
    //     move |x| x - num
    // }
}
```

迭代器

into_iter 会夺走所有权
iter 是借用
iter_mut 是可变借用
可以通过上面这些方法，将数组等集合转为迭代器；
可通过 for 遍历迭代器，或通过 next() 依次取一个元素

```
let v1 = vec![1, 2, 3];

let v1_iter = v1.iter();

for val in v1_iter {
    println!("{}", val);
}
// v1_iter.next() //每次取一个值
```

Iterator 迭代器特征，实现了它才能成为迭代器，才能调用 next；
IntoIterator (理解为迭代器生成器)，实现了这个特征才能使用 into_iter，iter 等方法变成迭代器。

同时 Iterator 特征也实现了 IntoIterator，
所以可以：vec.into_iter().into_iter()...

处理函数 1：消费者适配器
内部调用了 next 方法，即会消耗迭代器伤的元素；调用会产生结果；

- sum 求和
- collect 收集元素到指定集合类型中，常用做迭代器适配器的收尾函数

处理函数 2：迭代器适配器
惰性的，需要配合一个消费适配器才会产生结果，不然不会做任何执行；可链式调用。(其实它只是声明了一系列处理方式，遇到消费者适配器需要输出结果时才伴随着执行。)

- map 根据函数变换元素

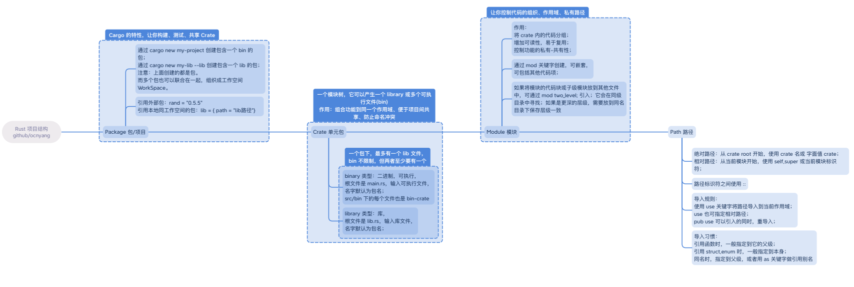
```
let v1: Vec<i32> = vec![1, 2, 3];

let v2: Vec<_> = v1.iter().map(|x| x + 1).collect();
```

- zip 将多个集合，依次将相同下表的元素组成元组，变成新的元组迭代器（理解上）
- filter 筛选出迭代器中满足条件函数的元素
- enumerate 转换为下标值迭代器

```
let v = vec![1u64, 2, 3, 4, 5, 6];
for (i,v) in v.iter().enumerate() {
    println!("第{}个值是{}",i,v)
}
```

```
let v = vec![1u64, 2, 3, 4, 5, 6];
let val = v.iter()
    .enumerate()
    // 每两个元素剔除一个
    // [1, 3, 5]
    .filter(|&(idx, _)| idx % 2 == 0)
    .map(|(_, val)| val)
    // 累加 1+3+5 = 9
    .fold(0u64, |sum, acm| sum + acm);
```

迭代器性能：迭代器是通过零成本抽象实现，不会引入运行时开销。
有时比 for 循环性能还要好，因为编译器还可以通过循环展开、向量化、消除边界检查等优化手段



Rust 项目结构
github/ocnyang

Package 包/项目

Cargo 的特性，让你构建、测试、共享 Crate

通过 cargo new my-project 创建包含一个 bin 的包；
通过 cargo new my-lib --lib 创建包含一个 lib 的包；
注意：上面创建的都是包。
而多个包也可以联合在一起，组织成工作空间 WorkSpace。

引用外部包：rand = "0.5.5"
引用本地同工作空间的包：lib = { path = "lib路径"}

Crate 单元包

一个模块树，它可以产生一个 library 或多个可执行文件(bin)
作用：组合功能到同一个作用域、便于项目间共享、防止命名冲突

一个包下，最多有一个 lib 文件，bin 不限制，但两者至少要有一个

binary 类型：二进制，可执行，
根文件是 main.rs，输入可执行文件，
名字默认为包名；
src/bin 下的每个文件也是 bin-crate

library 类型：库，
根文件是 lib.rs，输入库文件，
名字默认为包名；

Module 模块

让你控制代码的组织、作用域、私有路径

作用：
将 crate 内的代码分组；
增加可读性，易于复用；
控制功能的私有-共有性；

通过 mod 关键字创建，可嵌套，
可包括其他代码项；

如果将模块的代码块或子级模块放到其他文件中，可通过 mod two_level; 引入；它会在同级目录中寻找；如果是更深的层级，需要放到同名目录下保存层级一致

Path 路径

绝对路径：从 crate root 开始，使用 crate 名或 字面值 crate；
相对路径：从当前模块开始，使用 self,super 或当前模块标识符；

路径标识符之间使用 ::

导入规则：
使用 use 关键字将路径导入到当前作用域；
use 也可指定相对路径；
pub use 可以引入的同时，重导入；

导入习惯：
引用函数时，一般指定到它的父级；
引用 struct,enum 时，一般指定到本身；
同名时，指定到父级，或者用 as 关键字做引用别名

Rust 并发编程
github/ocnyang

- 并发编程
  - 创建线程
    let handle = thread::spawn(||{/*并发执行代码*/});
    阻塞主线程等待创建的子线程执行结束
    handle.join().unwrap();
    - 注意点：
      - 主线程结束时，子线程会立即结束；
      - 休眠线程 thread::sleep(Duration::from_millis(1));
      - 在线程闭包中使用 move 转移外部值的所有权到线程中；
      - 线程的代码执行完，线程会自动结束；
      - 子线程中再创建子线程，外部子线程的结束不会中断内部子线程；
  - 线程屏障 Barrier
    可以让多个线程都执行到某个点后，
    再同时开始往后执行

    ```
    fn main() {
        let mut handles = Vec::with_capacity(6);
        let barrier = Arc::new(Barrier::new(6));

        for _ in 0..6 {
            let b = barrier.clone();
            handles.push(thread::spawn(move|| {
                println!("before wait");
                b.wait();// 6 个线程执行到这相互等待
                println!("after wait");
            }));
        }

        for handle in handles {
            handle.join().unwrap();
        }
    }
    ```
  - 线程局部变量：
    thread_local 宏可以初始化线程局部变量；
    在线程内部使用该变量的 with 方法获取变量值；
    每个线程中 相同局部变量 的值互不干扰；

    ```
    fn main() {
        thread_local!(static FOO: RefCell<u32> = RefCell::new(1));

        FOO.with(|f| {
            assert_eq!(*f.borrow(), 1);
            *f.borrow_mut() = 2;
        });

        // 每个线程开始时都会拿到线程局部变量的FOO的初始值
        let t = thread::spawn(move|| {
            FOO.with(|f| {
                assert_eq!(*f.borrow(), 1);
                *f.borrow_mut() = 3;
            });
        });

        // 等待线程完成
        t.join().unwrap();

        // 尽管子线程中修改为了3，我们在这里依然拥有main线程中
    的局部值：2
        FOO.with(|f| {
            assert_eq!(*f.borrow(), 2);
        });
    }
    ```

    ```
    // 在结构体中使用线程局部变量
    struct Foo;
    impl Foo {
        thread_local! {
            static FOO: RefCell<usize> = RefCell::new(0);
        }
    }

    fn main() {
        Foo::FOO.with(|x| println!("{:?}", x));
    }
    ```

    ```
    // 通过引用的方式使用
    thread_local! {
        static FOO: RefCell<usize> = RefCell::new(0);
    }
    struct Bar {
        foo: &'static LocalKey<RefCell<usize>>,
    }
    impl Bar {
        fn constructor() -> Self {
            Self {
                foo: &FOO,
            }
        }
    }
    ```
    - 更强大的三方库：thread-local
  - 条件控制线程的挂起和执行：
    条件变量经常和 Mutex 一起使用，可以让线程挂起，直到某个条件发生后再继续执行

    ```
    fn main() {
        let pair = Arc::new((Mutex::new(false), Condvar::new()));
        let pair2 = pair.clone();

        thread::spawn(move|| {
            let (lock, cvar) = &*pair2;
            let mut started = lock.lock().unwrap();
            println!("changing started");
            *started = true;
            cvar.notify_one();
        });

        let (lock, cvar) = &*pair;
        let mut started = lock.lock().unwrap();
        while !*started {
            started = cvar.wait(started).unwrap();
        }

        println!("started changed");
    }
    ```
  - 只被调用一次的函数

    ```
    static mut VAL: usize = 0;
    static INIT: Once = Once::new();

    fn main() {
        let handle1 = thread::spawn(move || {
            INIT.call_once(|| {
                unsafe {
                    VAL = 1;
                }
            });
        });

        let handle2 = thread::spawn(move || {
            INIT.call_once(|| {
                unsafe {
                    VAL = 2;
                }
            });
        });

        handle1.join().unwrap();
        handle2.join().unwrap();

        println!("{}", unsafe { VAL });
    }
    // call_once 执行初始化过程一次，且只执行一次
    ```
  - 线程同步：消息传递
    - 单发送者、单接收者

      ```
      fn main() {
          let (tx, rx) = mpsc::channel();
          thread::spawn(move || {// 使用 move 转移 tx 的所有权
              tx.send(1).unwrap();// 推导出发送类型为 i32
              // send() 当接收者被 drop 无人接收时会报错，实际开发要处理
          });

          // recv() 将阻塞当前线程，直到读取到值或通道被关闭；
          println!("收到：{}", rx.recv().unwrap());
      }
      ```

      1. 使用 try_recv() 方法将不阻塞线程，且尝试接收一次消息，没有消息将返回错误 Empty。
      2. 发送消息后子线程将结束，发送者将被 drop，接收者会报错误 Disconnected；
      3. 发送的数据的所有权将转移给接收端，发送端不能再使用；
      4. 使用 for msg in rx {} 将阻塞的进行循环接收，直到子线程结束 发送者被 drop 才继续。
    - 多发送者、单接收者

      ```
      fn main() {
          let (tx, rx) = mpsc::channel();
          let tx1 = tx.clone();
          thread::spawn(move || {
              tx.send(String::from("hi from raw tx")).unwrap();
          });

          thread::spawn(move || {
              tx1.send(String::from("hi from cloned tx")).unwrap();
          });

          // 所有的发送者都被 drop 后，循环才结束，才继续执行
          for received in rx {
              println!("Got: {}", received);
          }
      }
      ```

      5. 接收到的消息是无序的，因为多个线程创建的时机不确定；但消息通道是有序的，遵循先进先出原则。
    - 通道的同步和异步
      上面的例子，通道都是异步的，也就是说，发送者只管发送，不管接收者是否在接收。
      同步通道相反，发送消息说阻塞的，只有消息被接收后才解除阻塞。⚠
      let (tx, rx)= mpsc::sync_channel(0); // 其中 0 表示消息缓存条数
    - 关闭通道
      所有发送者被 drop 或者所有接收者被 drop 后，通道会自动关闭。
      编译期实现，无性能损耗。
      - 新手坑

        ```
        fn main() {
            let (send, recv) = mpsc::channel();
            let num_threads = 3;
            for i in 0..num_threads {
                let thread_send = send.clone();
                thread::spawn(move || {
                    thread_send.send(i).unwrap();
                    println!("thread {:?} finished", i);
                });
            }

            // 在这里drop send...

            for x in recv {
                println!("Got: {}", x);
            }
            println!("finished iterating");
        }
        // send 没有被 drop，通道无法关闭，所以一直阻塞在循环上
        ```
    - 传输多种类型的数据，可以通过枚举类型实现；
      多发送者，多接收者 crossbeam-channel；
      或更高的性能 flume ；采用三方库。
  - 线程同步：锁、 Condvar 和信号量
  - 线程同步：Atomic 原子操作与内存顺序
  - 基于 Send 和 Sync 的线程安全
- 异步编程
  - async/.await
    - 通过 async 标记的语法块会被转换成实现了 Future 特征的状态机。 与同步调用阻塞当前线程不同，当 Future 执行并遇到阻塞时，它会让出当前线程的控制权，这样其它的 Future 就可以在该线程中运行，这种方式完全不会导致当前线程的阻塞。
    - async 使用的两种方式
      (异步函数的返回值是一个 Future)

      ```
      async fn foo() {//异步函数
          ...
      }
      ```

      ```
      async { // 代码块
          ...
      }
      ```
    - sync 体和其他 Future 都是惰性的，需要执行器才会运行；
      非异步函数调用：执行器 block_on 会阻塞线程等待 Future 完成；
      异步函数中调用：执行器 .await 会异步的等待 Future 完成；
  - Future
    - join! 宏
      可以同时并发的执行多个 future。

      ```
      async fn foo() {
          join!(foo_a(),foo_b());
      }
      ```
    - try_join!
      适用于返回 Result 的 future，
      如果子 future 其中一个返回了错误，会立即完成

      ```
      async fn foo_m() -> Result<String, String> {}
      async fn foo_n() -> Result<u32, String> {}
      async fn foo() -> Result<(String, u32), String> {
          let m = foo_m();
          let n = foo_n();
          try_join!(m,n)
      }
      ```

      ```
      // 如果多个 future 返回的错误类型不一样
      let m = foo_m().map_err(|i32| "err");
      // 可通过 map_err 转换一致
      ```
    - select!
      可以同时并发的执行多个 future，
      且在任意一个完成时进行响应并结束。

      ```
      async fn foo() {
          let m = foo_m().fuse();
          let n = foo_n().fuse();

          pin_mut!(m,n);
          select! {
              () = m => println!("m done!"),
              () = n => println!("n done!"),
          }
      }
      ```
      - select! 支持的两种分支
        default：如果上面判定的分支都没有完成，如果有 default 分支将立即返回；
        complete：只有当所有的判定分支都完成时才执行。
      - Unpin & FusedFuture

        首先，.fuse() 方法可以让 Future 实现 FusedFuture 特征， 而 pin_mut! 宏会为 Future 实现 Unpin 特征，这两个特征恰恰是使用 select 所必须的:

        1. Unpin，由于 select 不会通过拿走所有权的方式使用 Future，而是通过可变引用的方式去使用，这样当 select 结束后，该 Future 若没有被完成，它的所有权还可以继续被其它代码使用。
        2. FusedFuture 的原因跟上面类似，当 Future 一旦完成后，那 select 就不能再对其进行轮询使用。Fuse 意味着熔断，相当于 Future 一旦完成，再次调用 poll 会直接返回 Poll::Pending。

        只有实现了 FusedFuture，select 才能配合 loop 一起使用。假如没有实现，就算一个 Future 已经完成了，它依然会被 select 不停的轮询执行。
      - Steam 使用 select! 需要的特征是 FusedStream，
        也可以通过 .fuse() 实现，也可自己实现；
        但需要对其调用 .next() 或 .try_next()

        ```
        async fn add_two_streams(
            mut s1: impl Stream<Item = u8> + FusedStream + Unpin,
            mut s2: impl Stream<Item = u8> + FusedStream + Unpin,
        ) -> u8 {
            let mut total = 0;

            loop {
                let item = select! {
                    x = s1.next() => x,
                    x = s2.next() => x,
                    complete => break,
                };
                if let Some(next_num) = item {
                    total += next_num;
                }
            }

            total
        }
        ```
      - Fuse::terminated()
        可以使用它构建一个空的 Future，在循环运行中创建后再填充上

        ```
        async fn get_new_num() -> u8 { /* ... */ 5 }

        async fn run_on_new_num(_: u8) { /* ... */ }

        async fn run_loop(
            mut interval_timer: impl Stream<Item = ()> + FusedStream + Unpin,
            starting_num: u8,
        ) {
            let run_on_new_num_fut = run_on_new_num(starting_num).fuse();
            let get_new_num_fut = Fuse::terminated();
            pin_mut!(run_on_new_num_fut, get_new_num_fut);
            loop {
                select! {
                    () = interval_timer.select_next_some() => {
                        // 定时器已结束，若` get_new_num_fut` 没有在运行，就创建一个新的
                        if get_new_num_fut.is_terminated() {
                            get_new_num_fut.set(get_new_num().fuse());
                        }
                    },
                    new_num = get_new_num_fut => {
                        // 收到新的数字 -- 创建一个新的` run_on_new_num_fut` 并丢弃掉旧的
                        run_on_new_num_fut.set(run_on_new_num(new_num).fuse());
                    },
                    // 运行 ` run_on_new_num_fut`
                    () = run_on_new_num_fut => {},
                    // 若所有任务都完成，直接 ` panic`， 原因是 ` interval_timer` 应该连续不断的产生值，
                    // 而不是结束后，执行到 ` complete` 分支
                    complete => panic!("` interval_timer` completed unexpectedly"),
                }
            }
        }
        ```
      - 当某个 Future 有多个拷贝都需要同时运行时，
        可以使用 FuturesUnordered 类型。
        例子同上，但是它会将 run_on_new_num_fut 的每一个拷贝都运行到完成，而不是像之前那样一旦创建新的就终止旧的。

        ```
        async fn get_new_num() -> u8 { /* ... */ 5 }

        async fn run_on_new_num(_: u8) -> u8 { /* ... */ 5 }


        // 使用从 ` get_new_num` 获取的最新数字 来运行 ` run_on_new_num`
        //
        // 每当计时器结束后，` get_new_num` 就会运行一次，它会立即取消当前正在运行的` run_on_new_num` ，
        // 并且使用新返回的值来替换
        async fn run_loop(
            mut interval_timer: impl Stream<Item = ()> + FusedStream + Unpin,
            starting_num: u8,
        ) {
            let mut run_on_new_num_futs = FuturesUnordered::new();
            run_on_new_num_futs.push(run_on_new_num(starting_num));
            let get_new_num_fut = Fuse::terminated();
            pin_mut!(get_new_num_fut);
            loop {
                select! {
                    () = interval_timer.select_next_some() => {
                         // 定时器已结束，若 ` get_new_num_fut` 没有在运行，就创建一个新的
                        if get_new_num_fut.is_terminated() {
                            get_new_num_fut.set(get_new_num().fuse());
                        }
                    },
                    new_num = get_new_num_fut => {
                         // 收到新的数字 -- 创建一个新的 ` run_on_new_num_fut` (并没有像之前的例子那样丢弃掉旧值)
                        run_on_new_num_futs.push(run_on_new_num(new_num));
                    },
                    // 运行 ` run_on_new_num_futs` , 并检查是否有已经完成的
                    res = run_on_new_num_futs.select_next_some() => {
                        println!("run_on_new_num_fut returned {:?}", res);
                    },
                    // 若所有任务都完成，直接 ` panic` ， 原因是 ` interval_timer` 应该连续不断的产生值，而不是结束
                    //后，执行到 ` complete` 分支
                    complete => panic!("` interval_timer` completed unexpectedly"),
                }
            }
        }
        ```
  - 常见问题
    - async 语句块
      返回类型无法自动推导

      ```
      fn main() {
          let fut = async {
              foo_a().await?;
              foo_b().await?;
              // Ok(()) // 可能无法自动识别出返回类型
              Ok::<(), String>(()) // 采用此方式显式返回类型
          };
      }
      ```
    - 在 async 函数中使用非 Send 特征的变量
      理论上不允许，但是可以短暂使用。

      ```
      async fn foo_x() {
          NotSend::default(); // 可以
          let x = NotSend::default(); // 原来不可以，新版本可以
          {
              let x = NotSend::default(); // 可以
          }
          foo().await;
      }
      ```
    - 递归使用 async fn

      ```
      fn recursive() -> BoxFuture<'static, ()> {
          async move {
              recursive().await;
              recursive().await;
          }.boxed()
      }
      ```
    - 在特征中使用 async
      需要使用 async-trait
      有内存损耗

      ```
      #[async_trait]
      trait Advertisement {
          async fn run(&self);
      }

      struct Modal;

      #[async_trait]
      impl Advertisement for Modal {
          async fn run(&self) {
              self.render_fullscreen().await;
              for _ in 0..4u16 {
                  remind_user_to_join_mailing_list().await;
              }
              self.hide_for_now().await;
          }
      }

      struct AutoplayingVideo {
          media_url: String,
      }

      #[async_trait]
      impl Advertisement for AutoplayingVideo {
          async fn run(&self) {
              let stream = connect(&self.media_url).await;
              stream.play().await;

              // 用视频说服用户加入我们的邮件列表
              Modal.run().await;
          }
      }
      ```

Presented with xmind