

1 Rust语言介绍

Rust语言其他方面无需多言，只要强调一点就是它的目标：性能、安全以及实用

Rust 采百家之长，从 C++ 学习并强化了 move 语义和 RAII，从 Cyclone 借鉴和发展了生命周期，从 Haskell 吸收了函数式编程和类型系统等

1.1 内存安全方案

1.1.1 Rust针对C语言的不足

1. 禁止对空指针和悬垂指针解引用（rust中引用属于一等公民，受所有权、借用检查以及生命周期约束，不会有空指针和悬垂指针，更别说解引用了）

空指针指的是指向了不存在的数据

悬垂指针指的是原本指向的数据被释放掉了

2. 读取未初始化的内存（rust中必须先初始化才能读取）
3. 非法释放已经释放或未分配的指针（rust释放使用的是drop trait）
4. 缓冲区溢出（? ? ?，有空了需要学一点C语言和C++）

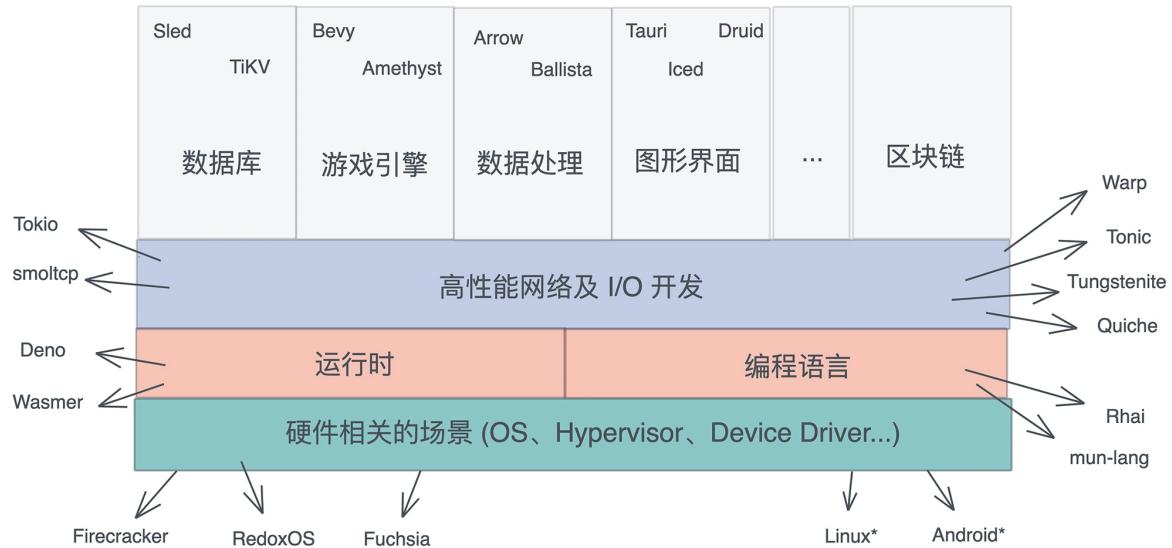
Rust本质上是限制了对指针使用的行为，就像React一样，

1.1.2 安全无缝地沟通C语言

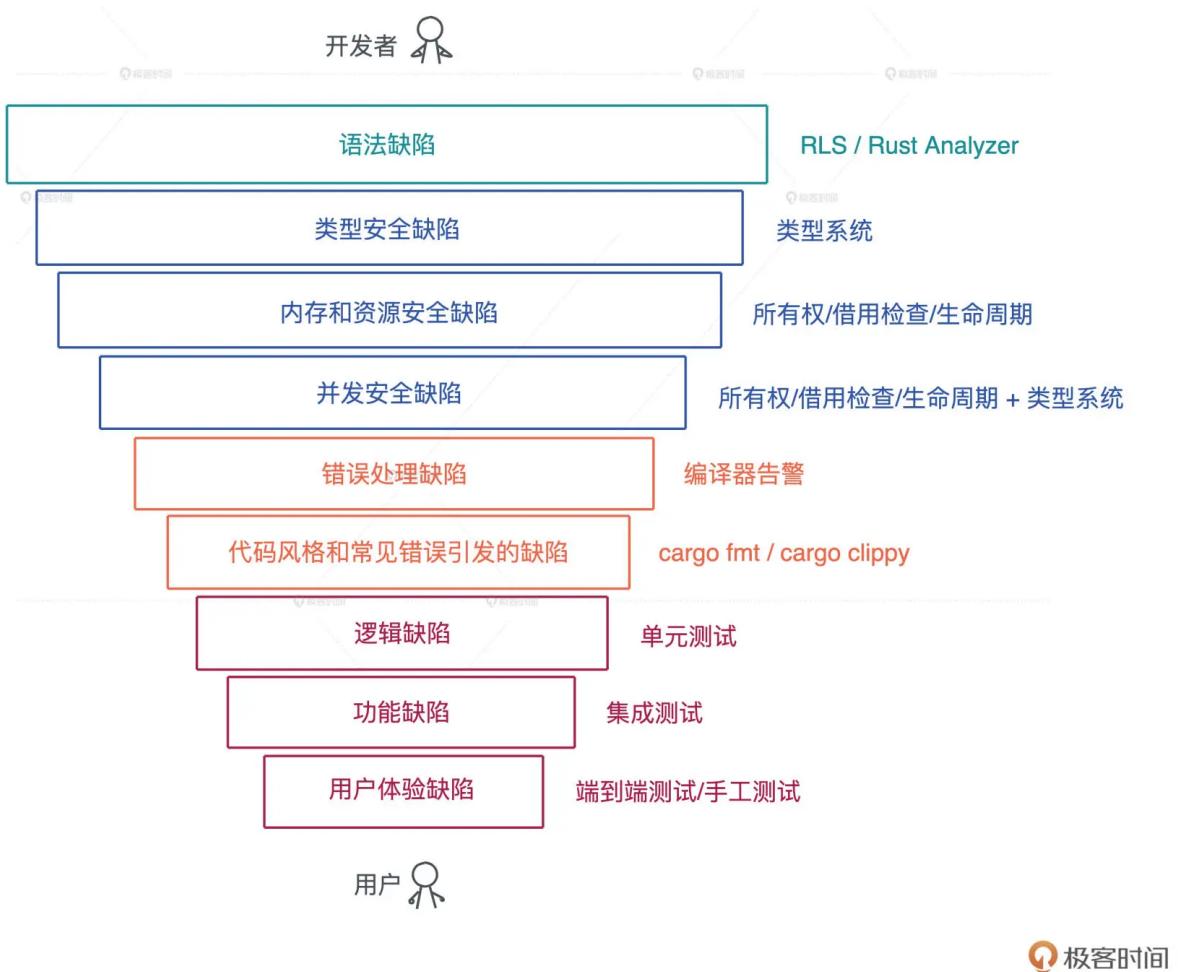
通过C-ABI零成本和C语言打交道

划分了Safe Rust和Unsafe Rust

1.2 Rust应用场景



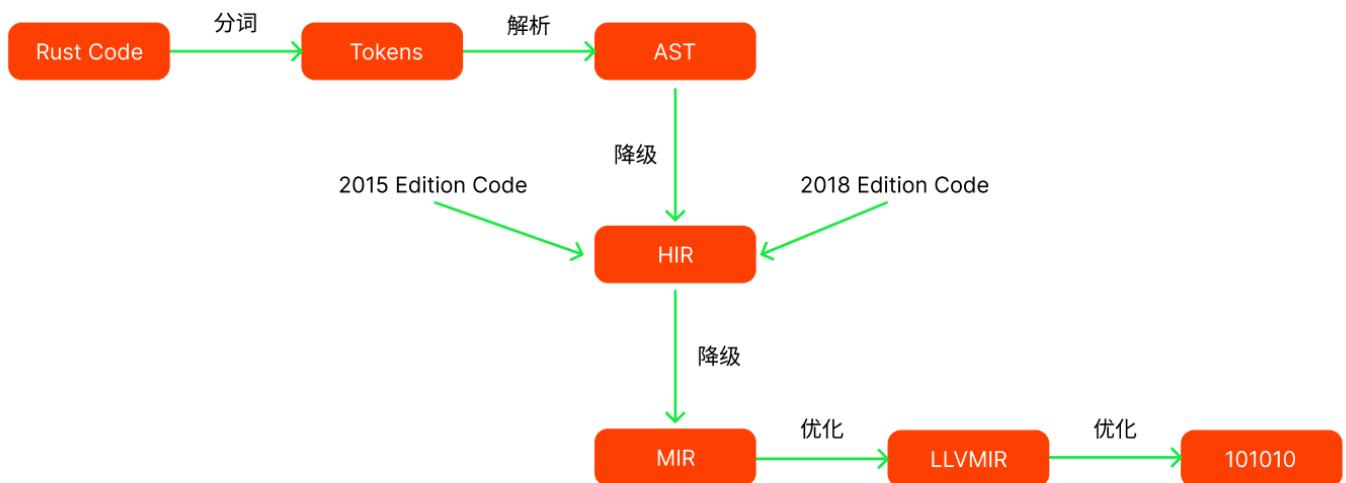
1.3 通用内存安全解决方案



2 Rust语言基础

2.1 Rust 语言编译

2.1.1 编译过程



2.1.2 Rust与其它语言编译比较

大部分语言会将词条流解析到的抽象语法树直接转为机器码，但是rust会将其转为高级中间语言以及中级中间语言、LLVM中间语言，然后再交由LLVM后端生成机器码。各级中间语言承担的功能：

1. 高级中间语言：类型检查、方法查找
2. 中级中间语言：借用检查、代码生成、泛型单态化、优化等工作
3. Rust语言版次差异在到达中级中间语言时就会消除

2.2 Rust 词法结构

词法结构对于任何一种语言来说都非常重要，因为它不光是构成语言的必要部分，而且也关乎到语言如何解析和编译。在rust中，词法结构中的词条还涉及元编程

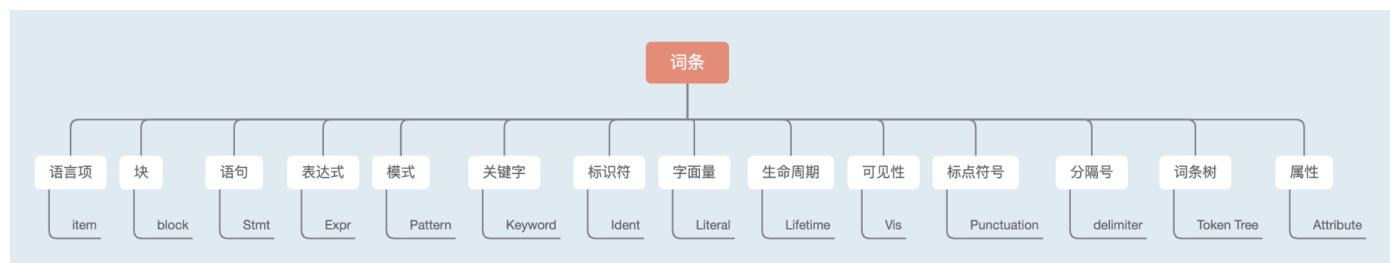
2.2.1 六大词法结构

1. 关键字：严格关键字、保留字、弱关键字
2. 标识符：不以数字开头的ASCII字符注释

```
let name = "name";
let _100 = "number";
let math_grade = 150;

println!("{} , {} , {}" , name , _100 , math_grade)
```

3. 注释：Rust可以使用注释直接生成文档，非常友好。
4. 空白：空白不表示任何含义，如换行等
5. 词条：词条在写宏的时候非常有用（它是宏的关键词，需要熟悉并深刻理解词条才能编写宏代码），Rust语言有14个词条



6. 路径：Rust中路径有三种用途，模块路径、方法调用和泛型类型指定

```
pub fn main() {

    /// 1.模块路径
    ///

    pub mod a {
        fn foo() {
            println!("a")
        }
    }

    pub mod b {
        pub mod c {
```

```
    pub fn foo() {
        super::super::foo();
        self::super::super::foo();
    }
}

}

a::b::c::foo();

/// 2.方法调用
///
struct S;

impl S {
    fn correlation_function(){
        println!("correlation function");
    }
}

trait T1 {
    fn method1() {
        println!("method1");
    }
}

impl T1 for S {}

trait T2 {
    fn method2() {
        println!("method2")
    }
}

impl T2 for S {}

// 注意: 调用方法有两种情况
// 两个trait中的方法相同时使用完全限定无歧义调用
<S as T1>::method1();
<S as T2>::method2();

// 其他情况下, 调用关联函数和方法的方式相同
S::correlation_function();
S::method1();

/// 3.泛型函数-turbofish操作符, rust也会支持局部类型推断
///

// 将0到9收集到vec中,默认类型是i32, 但是可以指定为u64
```

```

let vec0 = (0..10).collect::<Vec<_>>();
let vec1 = (0..10).collect::<Vec<u64>>();
println!("{:?}", vec1);

// 开辟一个容量为1024的u8Vec
let vec2 = Vec::<u8>::with_capacity(1024);

println!("{:?}", vec2);

// 类型推断

fn main() {
    let numbers = vec![1, 2, 3, 4, 5, 6, 7, 8, 9, 10];

    let even_numbers = numbers
        .into_iter()
        .filter(|n| n % 2 == 0)
        .collect::<Vec<_>>();

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

```

2.3 Rust 语法骨架

Rust语法骨架只包含三类元素

1. 属性：行属性和块属性

以# 或者 #! 开头

2. 分号：行分隔符

以 ; 结尾

3. 花括号：块分隔符

以 } 结尾

2.4 Rust表达式

在Rust中，一切皆表达式,它是以分号 ; 和花括号 {} 进行区分，而不是以循环、匹配等条件作为区分

一切皆表达式可以引申为一切皆类型，因为表达式都有值，而值都有类型

let / fn / static / const 是一些定性语句

2.4.1 表达式分类：按语法骨架

其中作为Rust骨架的分号和花括号构成了Rust语言中两种最基本的表达式

1. 分号表达式：值的类型是单元类型，它实际上是一个空元组。如：

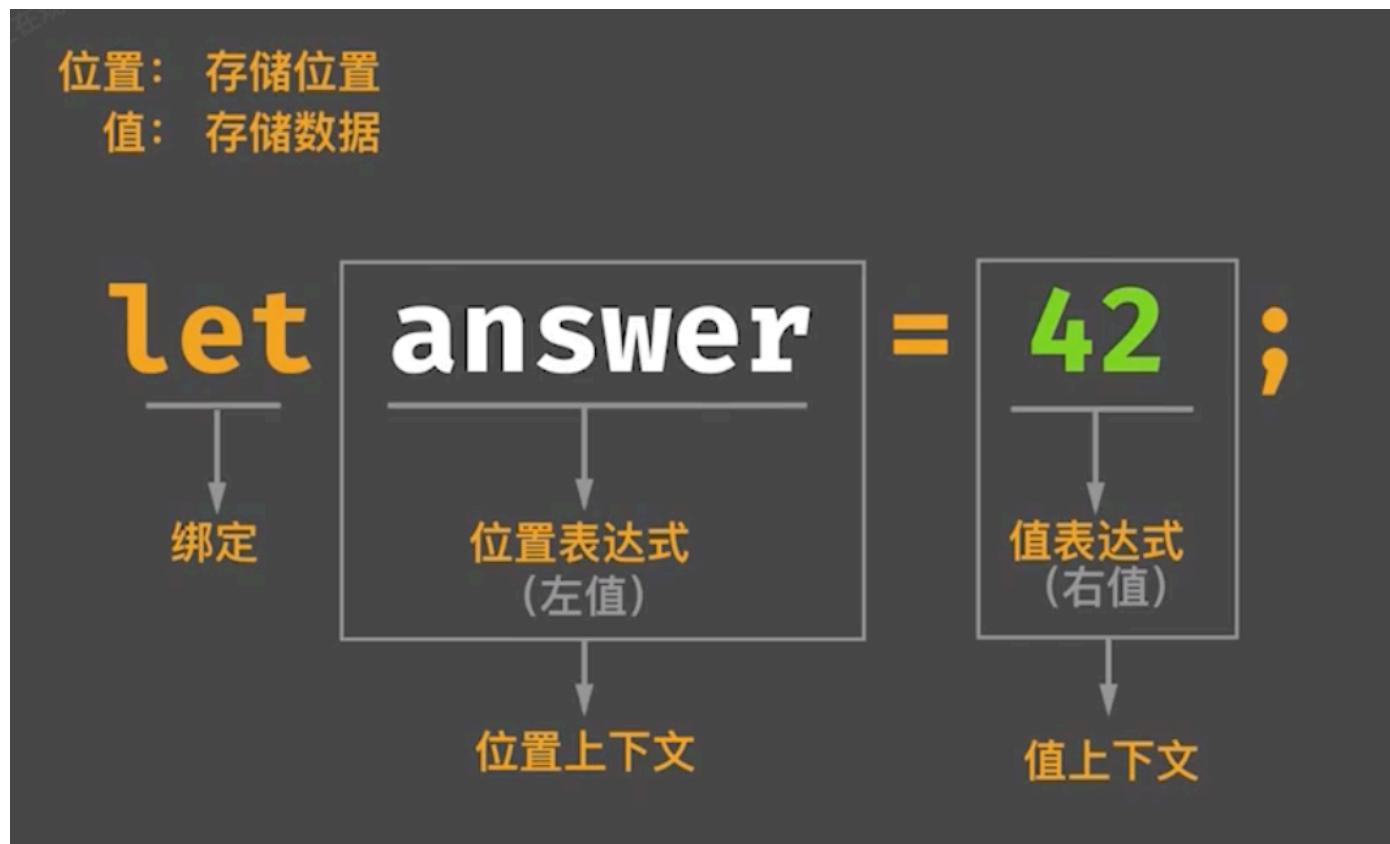
```
; -> ()  
let expr: &str = "hello";
```

块表达式：值的类型是块中最后一个表达式的值。当块中最后一行为一个值时，块表达式的值为该值，类型是该值的类型。如：

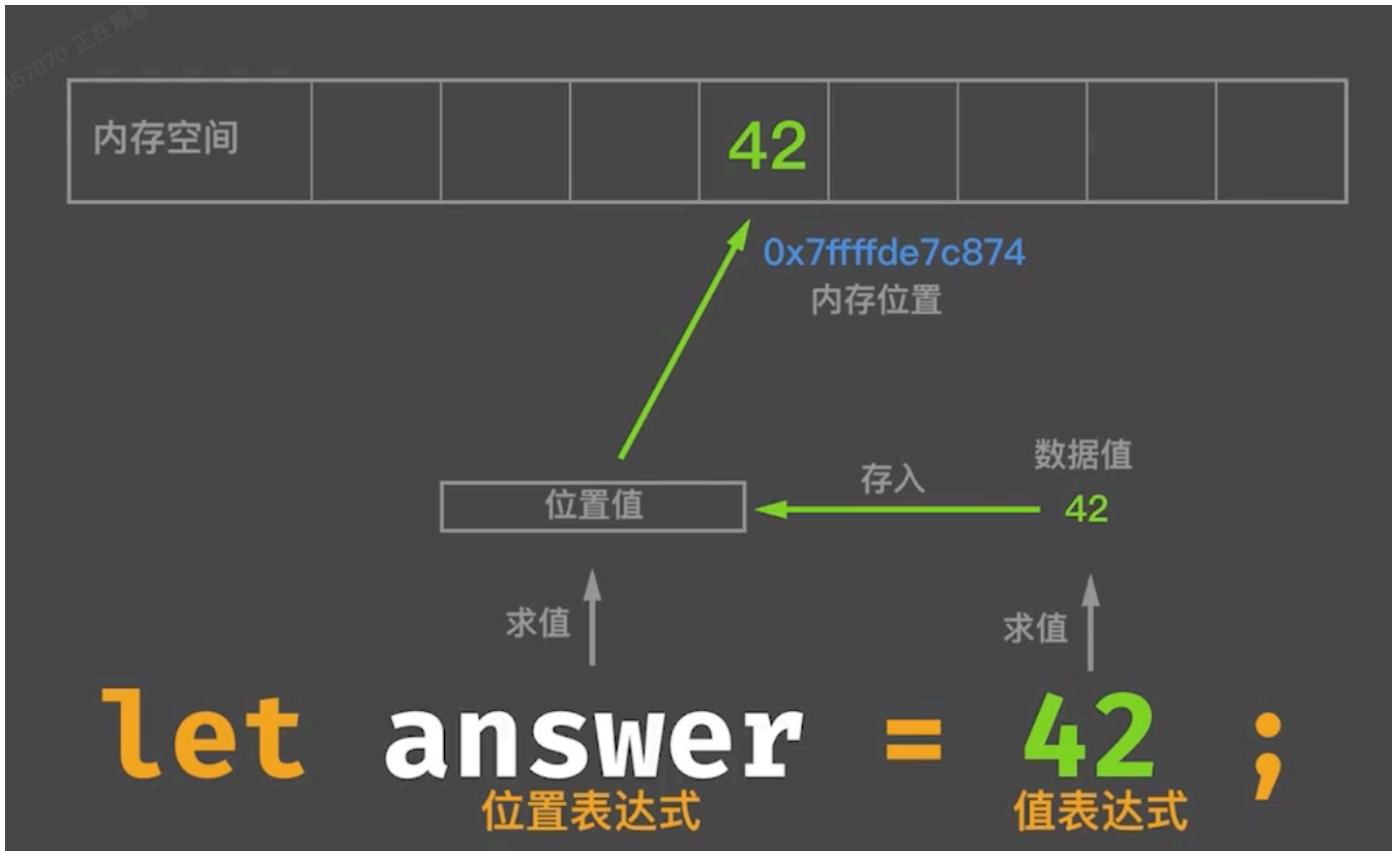
```
let a: () = {  
    let expr = "hello";  
};  
  
let b: &str = {  
    let expr = "hello";  
    expr  
};
```

2.4.2 表达式分类：按内存管理

1. 对于数据和变量的关系，变量处于位置区域，数据处于值区域，位置存储位置，值存储数据，二者以等号为界
2. 位置表达式：位置，存储位置
3. 值表达式：值，存储数据



4. 表达式背后的内存管理



静态变量初始化：比如：static mut A:u32 = 0; 解引用表达式：形如*expr 数组索引表达式：形如expr[expr] 字段表达式：形如expr.field 括号位置表达式：(expr)

2.4.2.1 位置上下文

```
// 1. 赋值和复合赋值操作左侧
let mut a = 1;
a += 1;

// 2. 一元借用和解引用操作数所在区域
let a = &mut 7;
*a = 42;
// 二元借用 b:&&mut i32
let b = &a;

// 3. 字段表达式操作数所在位置
struct A {
    name: &'static str,
}
let a = A { name: "Alice" };
a.name; //位置上下文
```

```

// 4.数组索引表达式操作数所在区域
let mut arr = [1, 2, 3];
let b = &mut arr;
arr[1];

// 5.任意隐式借用操作数所在区域
let mut v = vec![1, 2, 3];
v.push(4);

// 6.let 初始化
let a: i32;

// 7.if let/while let/match 的匹配表达式所在的区域
let dish = ("ham", "eggs");
if let ("bacon", b) = dish {
    // ("bacon",b) 就是位置上下文
    println!("bacon is served {}", b);
} else {
    println!("No bacon will be served")
}

//match (位置上下文) / while let (位置上下文) 同理

// 结构体更新语法中的base表达式
struct Point3d {
    x: i32,
    y: i32,
    z: i32,
}

let mut base = Point3d { x: 1, y: 2, z: 3 };
let y_ref = &mut base.y;

Point3d {
    y: 0,
    z: 10,
    ..base
};

```

2.4.3 所有权语义在表达式上的体现



2.4.3.1 位置表达式的移动

```
let stack_a = 42;
let stack_b = stack_a; // 位置表达式到值上下文中，发生了copy

stack_a;

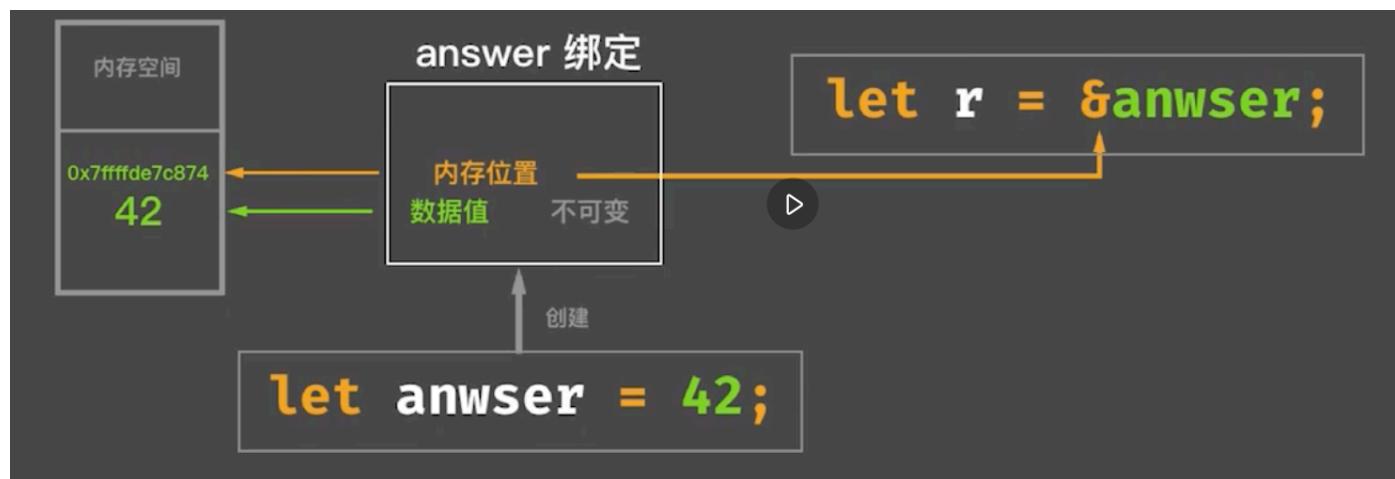
let heap_a = "hello".to_string();
let heap_b = heap_a; // 位置表达式到值上下文中，发生了move

//      heap_a; error
```

2.4.4 不可变与可变

由于所有权机制，一个内存地址只能有一个绑定

1. 不可变绑定与可变绑定
2. 不可变引用（共享引用）与可变引用（独占引用）



3. Rust中与C语言一样的`*mut T`和`*const T`只能在Unsafe Rust中使用，Rust中的原生指针没有所有权语义

2.5 编译期计算

编译期计算 (CTFE)：编译期函数求值，最先由Lisp/Cpp语言支持

2.5.1 Rus编译期计算方式

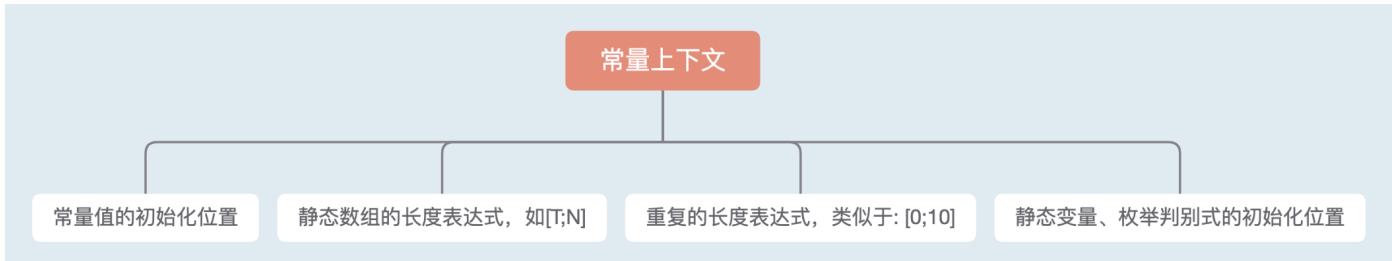
过程宏 + Build脚本 (build.rs)：类型计算、生成代码等，但是无法在宏代码和普通代码之间共享代码

类似Cpp中的constexpr的CTFE功能：分为两类：常量函数和常量泛型

2.5.1.1 常量表达式和常量上下文

在编译期对常量表达式进行求值

```
const AN: i32 = 1000; //常量表达式
```



1. 常量上下文是编译器唯一进行常量求值的地方
2. 编译期计算默认是对开发者透明的，但是了解这部分的知识能够让你对底层更有sense
3. 与常量计算相对应地一个知识点：常量传播，它是编译器的一种优化，防止运行时重复计算

2.5.1.2 常量安全

1. 理论上，Rust中的大部分表达式都可以用作常量表达式，但目前只支持常量函数，元组结构体，元组的值
2. 并不是所有常量表达式都可以用在常量上下文：比如某个数组的长度依赖于磁盘中文件内容的长度。因为编译期求值必须得到确定结果，当文件发生变化时就无法保证确定性
3. 因此rust引入了常量函数解决常量安全问题

```
// 1. 常量函数
const fn gcd(a: u32, b: u32) -> u32 {
    match (a, b) {
        (x, 0) | (0, x) => x, // 
        (x, y) if x % 2 == 0 && y % 2 == 0 => 2 * gcd(x / 2, y / 2),
        (x, y) | (y, x) if x % 2 == 0 => gcd(x / 2, y / 2),
        (x, y) if x < y => gcd((y - x) / 2, x),
        (x, y) => gcd((x - y) / 2, y),
    }
}

const GCD: u32 = gcd(21, 7);

println!("{}:", GCD);
```

2.5.1.3 编译期计算如何实现

Rust编译器内置了MIR解释器：Miri，它会执行中级中间语言中const上下文中的const代码，从而实现编译期计算
一个小知识点：无限循环用loop而不是while true

2.5.1.4 常量泛型

Rust中静态数组是二等公民，长度不同类型不同，我们无法使用统一的命名所有数组

```
// 可以定义泛型结构体
pub struct S<T, N> {
    x: T,
    y: N,
}

// 但是无法定义泛型数组
// let arr = [T; N]; // 不支持
```

问题：如何定义一个泛型静态数组，等到真正填充数据的时候，再决定数组中元素的类型以及长度？

问题的核心是：在未初始化数据结构的情况下在分配内存空间

解决方案：

1. 使用泛型结构体声明泛型参数 `T` 和常量泛型 `N`
2. 使用核心库中的联合体 `MaybeUninit` 包裹泛型参数占位
3. 用于给泛型生成一个未初始化的示例，并再构建一个泛型结构体，泛型参数分别是类型 `T` 和常量泛型。
`MaybeUninit` 用来占位

解决方案的核心是：先分配内存空间，再初始化数据结构

```
#![feature(min_const_generics)]
use core::mem::MaybeUninit;

#[derive(Debug)]
pub struct ArrayVec<T, const N: usize> {
    items: [MaybeUninit<T>; N], // 先分配内存
    length: usize,
}

fn main() {
    println!();

    let av = ArrayVec {
        items: [MaybeUninit::uninit(); 3], // 再声明数据结构
        length: 10,
    };

    println!("{:?}", av)
}
```

```
// 打印结果:  
ArrayVec {  
    items: [  
        core::mem::maybe_uninit::MaybeUninit<u32>,  
        core::mem::maybe_uninit::MaybeUninit<u32>,  
        core::mem::maybe_uninit::MaybeUninit<u32>,  
    ],  
    length: 10,  
}
```

常量泛型目前只支持

1. 一个简单的常量泛型参数，比如 `const N:usize`
2. 可以在不依赖任何类型或常量参数的常量上下文中使用表达式

保留的问题：什么时候使用常量泛型呢

```
// array_chunks 方法是基于常量泛型对数组进行分割处理  
  
let data = [1, 2, 3, 4, 5, 6];  
let sum1 = data.array_chunks().map(|&[x, y]| x * y).sum::<i32>();  
let sum2 = data.array_chunks().map(|&[x, y, z]| x * y * z).sum::<i32>();  
assert_eq!(sum1, (1 * 2) + (3 * 4) + (5 * 6));  
assert_eq!(sum2, (1 * 2 * 3) + (4 * 5 * 6));  
  
println!("{} , {}", sum1, sum2);
```

2.6 Rust 类型系统

2.6.1 类型系统目标

1. 保证内存安全
2. 保证一致性
3. 表达明确的语义
4. 零成本抽象表达能力

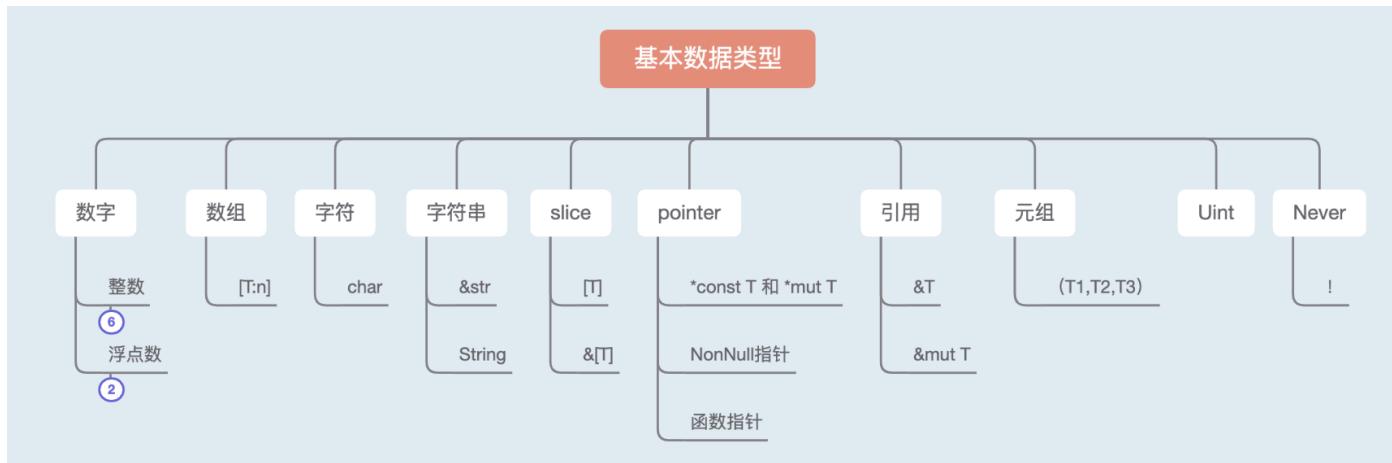
2.6.2 Rust如何实现目标

类型：在rust中，一切皆类型

行为：Rust使用trait规范了类型的行为

2.6.3 Rust数据类型

2.6.3.1 基本数据类型



特别说明

1. `usize`和`isize`有符号和无符号指针大小类型，指针一般和计算机字长相等，32位处理器：4字节，64位处理器：8字节
2. 布尔值可以转数字，但是反过来不可以
3. 数组在Rust中是二等公民，长度不同，类型不同。等常量泛型稳定后可以晋升统一的 $[T;N]$ 类型
4. 字符，rust中的`char`是unicode标量，占四个字节,对应于Rust中的`u32`类型。并且可以方便的转换为utf8编码的字节序列。字节序列的每一个元素是1个字节。注意对应`u32`类型，并不代表所占字节是4字节，所占字节仍然遵从unicode规则
5. 补充知识：

5.1 ASCII 字符集（英文字符集）：使用一个字节存储，共计128个字符

5.2 GBK:汉字字符集，占两个字节，共计两万多个，编码第一位是1

5.3 Unicode字符集：编码方案：uft-8，可变长编码方案，共分为四个长度区：1字节，2字节，3字节，4字节

汉字：1110xxxx 10xxxxxx 10xxxxxx，英文数字一个字节

```
let tao = '道';

let tao_u32 = tao as u32;
println!("{}" , tao_u32); // 字符'道'对应的u32的值
println!("U+{:x}" , tao_u32); // 道的Unicode 字符编码
println!("{}" , tao.escape_unicode()); // 道转译后的Unicode 码点

let a = char::from(65);
println!("{}" , a);

//转换16进制的码点值
if let Some(c1) = std::char::from_u32(0x9053) {
    println!("{}" , c1)
}
if let Some(c2) = std::char::from_u32(36947) {
```

```
    println!("{}", c2)
}

// 并不是所有的数字都是有效的Unicode标量值
if let Some(c3) = std::char::from_u32(129010101) {
    println!("{}", c3)
} else {
    println!("invalid code")
}

use std::str;
// 将utf-8序列转换为字符串
let tao = str::from_utf8(&[0xE9u8, 0x81u8, 0x93u8]).unwrap();
println!("tao:{}", tao);

// 通过16进制码位转换为字符串
let tao = String::from("\u{9053}");
println!("{}", tao);
let unicode_x = 0x9053;
let utf_x_hex = 0xe98193;
let utf_x_bin = 0b11101001100000011001011;

println!("unicode_x: {:?}", unicode_x);
println!("utf_x_hex: {:?}", utf_x_hex);
println!("utf_x_bin: {:?}", utf_x_bin);

// 特殊字符
// 码位可能不同,但字节大小一样
// 长度可能不同,但值的大小一样

let e = 'é'; // 和 let e = 'é'; 不一样, 前者是两个unicode 码点, 后者是1个
             // let e = 'é';
let f = 'e';

let g = "é";
let h = "e";

println!("e utf-8 bytes: {}", e.len_utf8()); // 占2个字节
println!("f utf-8 bytes: {}", f.len_utf8()); // 占1个字节

println!("e value size: {}", std::mem::size_of_val(&e)); // 4字节
println!("f value size: {}", std::mem::size_of_val(&f)); // 4字节

println!("g utf-8 bytes: {}", g.len()); // 2字节
println!("h utf-8 bytes: {}", h.len()); // 1字节

println!("g value size: {}", std::mem::size_of_val(&g)); // 16字节
println!("h value size: {}", std::mem::size_of_val(&g)); // 16字节
```

```
// emoji 只能是字符串
let s = String::from("love: ❤");
println!("emoji {}", s)
```

实现的 trait 有 Copy、Clone 等，Clone 是深拷贝，堆栈内存一起拷贝

clone 方法的接口是 &self，这在绝大多数场合下都是适用的，我们在 clone 一个数据时只需要有已有数据的只读引用。但对 Rc 这样在 clone() 时维护引用计数的数据结构，clone() 过程中会改变自己，所以要用 Cell 这样提供内部可变性的结构来进行改变

6. 字符串，rust 中的字符串有非常多的类型，从根本上讲是为了适应不同的场景，如下：

在 Rust 中，字符串比较复杂，涉及底层内存管理知识

```
// 类型是 &str，字符串切片的引用，胖指针（指针和数据长度），原属数据存放在静态存储区
let s_static_memory = "hello";

// 不可以使用未知大小的静态存储区的原始字符串
// let s = *s_static_memory;

// 类型是 String，字符串的引用，智能指针（指针、容量和数据长度），原属数据存放在堆上
let s_heap_memory = String::from("hello");

// 不可以使用未知大小的堆上原始字符串
// let s = *s_heap_memory;
```

Rust 中每一个字符串都是一个 UTF-8 字节序列，实际上是一个“Vec”动态数组

6.1 两种常见类型：str（字符串切片）和 String

Rust 中没有内含正则引擎，日常字符串操作通过它本身的一些方法来完成字符、字节和字符串之间的转换。还有一些定位、搜索、匹配、去除空白等方法。可以在多线程中安全的使用

6.2 String 为什么有容量，因为它基于数组

Pattern 相关的 trait 提供了同名函数不同参数的功能，可以重点看看

6.3 其他类型：

1. Cstr/Cstring 与 C 语言打交道
2. OsStr/OsString 与操作系统打交道
3. Path/PathBuf 与路径打交道

标准库导读三原则

1. 类型自身介绍
2. 类型自身实现的方法
3. 类型实现的 trait

7 指针类型

两种原始指针：*mut T 和 *const T

NonNull指针：替代*mut T，非空，并遵循生命周期类型协变规则

函数指针：指向代码而非数据，可以用于直接调用函数

8 引用与指针之别

1. 引用不为空
2. 拥有生命周期
3. 受借用检查器保护，不会发生悬垂指针等问题
4. 访问受限，只能解引用到它引用的数据类型，不能用做它用

8 元组

Rust中唯一的异构序列

长度不同类型不同

单元类型的唯一实例等价于空元组

当元组只有一个元素时需要在元素后加逗号，以做区分

```
// 类型是元组: (i32,)  
let a = (42,);  
// 类型是 i32  
let b = (42);
```

9 Never类型

代表不可能返回值的计算类型

在类型理论中叫底类型，不包含任何值，但是可以合一到任何其他类型。用! 表示（目前还未稳定）

2.6.3.2 自定义复合类型

2.6.3.2.1 结构体

2.6.3.2.1.1 结构体种类

```
// 1.具名结构体  
struct Point {  
    x:f32,  
    y:f32  
}  
  
// 2.元组结构体,常用于包装基本数据类型以扩展功能  
struct Pair(i32,i32);  
// 当元组结构体只包含一个类型时，称为NewType模式  
// 如下对u32进行包装，表示分数  
struct Score(u32);  
  
impl Score {  
    fn pass(&self) -> bool {
```

```

        self.0 >= 60
    }
}

let s = Score(59);
assert_eq!(s.pass(), false);

// 3.单元结构体,实例就是它自身, 0大小
struct Uint;

let point = Point { x: 3.0, y: 4.0 };
let pair = Pair(1, 1);
let uint = Uint;

assert_eq!(point.x, 3.0);
assert_eq!(pair.0, 1);

```

2.6.3.2.1.2 内存对齐方式

```

// 推断结构体占12字节
// #[repr(C)] //使用属性不让编译器自动优化布局
struct A {
    a: u8,    // 占1字节,按照4字节对齐, 补3
    b: u32,   // 占4字节, 补0
    c: u16,   //占2字节, 补2
}

// 实际优化,字段重排
struct B {
    b: u32, // 计算机按照字节寻址, 指令是字节的整数倍
    c: u16,
    d: u8,
}

println!("{:?}", std::mem::size_of::<A>());
println!("{:?}", std::mem::size_of::<B>());

```

2.6.3.2.2 枚举

枚举在Rust下是一个标签联合体，大小是标签的大小加上最大类型的长度。enum的最大长度是最大类型的长度+8
常见的类型长度

Type	T	Option<T>	Result<T, io::Error>
u8	1	2	24
f64	8	16	24
&u8	8	8	24
Box<u8>	8	8	24
&[u8]	16	16	24
String	24	24	32
Vec<u8>	24	24	32
HashMap<String, String>	48	48	56
E	56	56	64

对于 Option 结构而言，它的 tag 只有两种情况：0 或 1，tag 为 0 时，表示 None，tag 为 1 时，表示 Some。tag 占 1 个字节。64 位 CPU 对对齐时 8 字节

rust 如何优化的，当 tag 后的类型是引用类型时，tag 为 0，其它情况下为 1

特殊的枚举类型 Cow，就像 Option 一样，在返回数据的时候，提供了一种可能：要么返回一个借用的数据（只读），要么返回一个拥有所有权的数据（可写），? 代表放松问号之后的约束，?Sized 代表用可变大小的类型，Rust 默认泛型参数都是 Sized 的

```
pub enum Cow<'a, B: ?Sized + 'a> where B: ToOwned, // early bound
{
    // 借用的数据
    Borrowed(&'a B),
    // 拥有的数据
    Owned(<B as ToOwned>::Owned), // 在 rust 中，子类型可以强强制转换为父类型
}
```

late boud：逐步添加约束，可以让约束只出现在它不得不出现的地方，这样代码的灵活性最大

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

// 定义一个带有泛型参数 R 的 reader，此处我们不限制 R
struct MyReader<R> {
    reader: R,
    buf: String,
}

// 实现 new 函数时，我们不需要限制 R
impl<R> MyReader<R> {
    pub fn new(reader: R) -> Self {
        Self {
            reader,
            buf: String::with_capacity(1024),
        }
    }
}
```

```

}

// 定义 process 时，我们需要用到 R 的方法，此时我们限制 R 必须实现 Read trait
impl<R> MyReader<R>
where
    R: Read,
{
    pub fn process(&mut self) -> Result<usize> {
        self.reader.read_to_string(&mut self.buf)
    }
}

fn main() {
    // 在 windows 下，你需要换个文件读取，否则会出错
    let f = File::open("/etc/hosts").unwrap();
    let mut reader = MyReader::new(BufReader::new(f));

    let size = reader.process().unwrap();
    println!("total size read: {}", size);
}

```

2.6.3.3 容器类型

系统相关

容器类型

原生类型

I/O 抽象：

- TcpStream / TcpListener: TCP Socket
- UdpSocket: UDP socket
- SocketAddr: socket 地址
- File: 文件
- Metadata: 文件元数据
- PathBuf / Path: 文件路径
- OsString / OsStr: 操作系统字符串
- Thread: 线程
- JoinHandle: 线程句柄

并发抽象：

- AtomicXXX: 并发原语
- Mutex<T> / RwLock<T>: 共享内存
- CondVar: 同步原语
- Channel<T>: 并发通道

特定容器：

- Option<T>: 表达有值或无值
- Result<T, E>: 表达正确或错误结果
- Cell<T> / RefCell<T>: 单线程下内部可变性
- Rc<T> / Arc<T>: 单线程/多线程引用计数
- Cow<a, B>: 写时克隆
- Box<T>: 分配到堆内存

集合容器：

- Vec<T>: T 的连续列表
- String: 字符的连续列表
- VecDeque<T>: 循环缓冲区
- LinkedList<T>: 双向链表
- BinaryHeap<T>: 二叉堆(最大堆)
- HashMap<K, V> / BTreeMap<K, V>: 哈希表 / 有序哈希表
- HashSet<T> / BTreeSet<T>: 哈希集 / 有序哈希集

基本类型：

- bool
- i8 / i16 / i32 / i64 / isize
- u8 / u16 / u32 / u64 / usize
- f32 / f64

指针和引用：

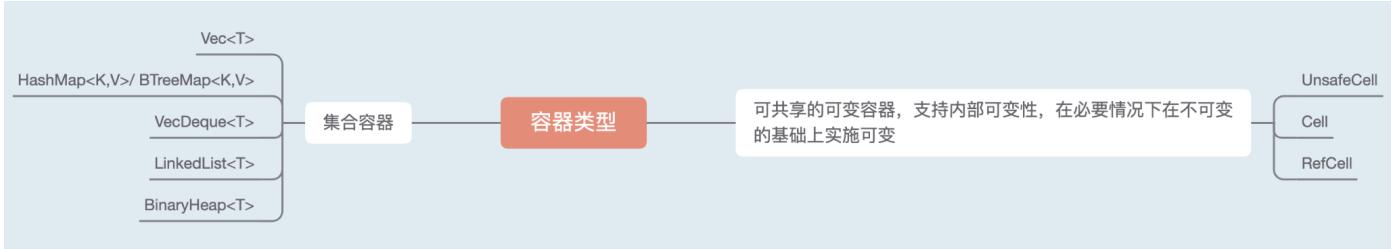
- *const T / *mut T
- &T / &mut T

集合容器：

- slice
- str
- array

特定容器：

- tuple



切片

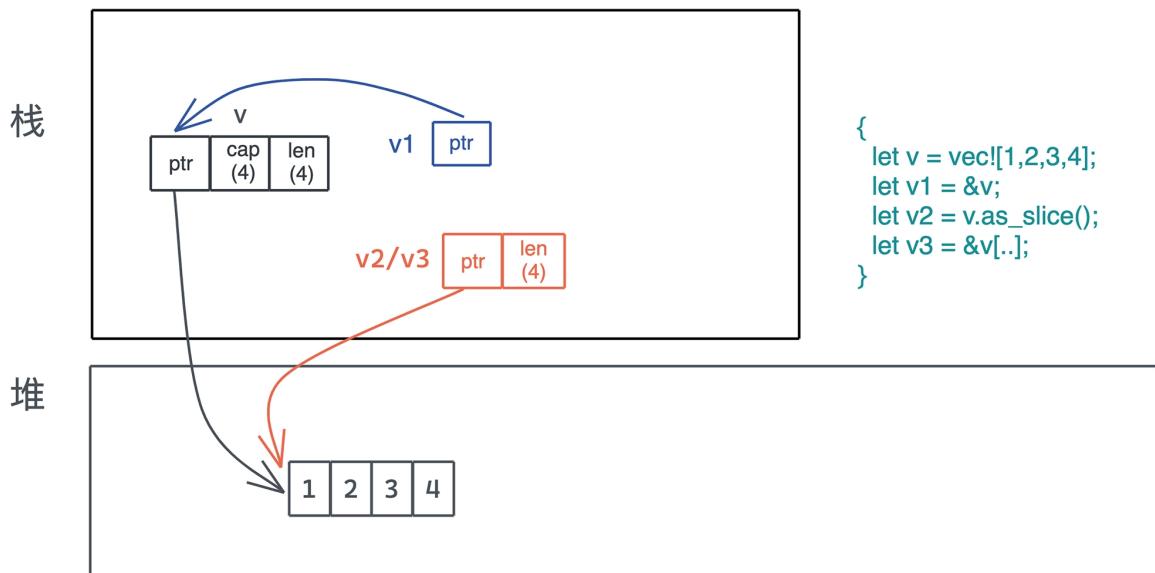
三种形式，切片之于具体的数据结构，就像数据库中的视图于表，可以把它看成一种工具，让我们可以统一访问行为相同、结构类似但有些许差异的类型

```
&[T]: 表示一个只读的切片引用
&mut [T]: 表示一个可写的切片引用
Box<[T]>: 一个在堆上分配的切片。
```

```
fn main() {
    let arr = [1, 2, 3, 4, 5];
    let vec = vec![1, 2, 3, 4, 5];
    let s1 = &arr[..2];
    let s2 = &vec[..2];
    println!("s1: {:?}", s1, s2);

    // &[T] 和 &[T] 是否相等取决于长度和内容是否相等
    assert_eq!(s1, s2);
    // &[T] 可以和 Vec<T>/[T;n] 比较，也会看长度和内容
    assert_eq!(&arr[..], vec);
    assert_eq!(&vec[..], arr);
}
```

array 和 vector，虽然是不同的数据结构，一个放在栈上，一个放在堆上，但它们的切片是类似的；而且对于相同内容数据的相同切片，比如 `&arr[1..3]` 和 `&vec[1..3]`，这两者是等价的，并且切片和对应的数据结构可以直接比较，它们之间实现了 `PartialEq` trait



极客时间

支持切片的具体数据类型，可以根据需要，解引用转换为切片类型。比如Vec 和 [T;n] 会转化为 &[T].这是因为 Vec 实现了 Deref trait，而 array 内建了到 &[T] 的解引用

```

use std::fmt;
fn main() {
    let v = vec![1, 2, 3, 4];

    // Vec 实现了 Deref, &Vec<T> 会被自动解引用为 &[T], 符合接口定义
    print_slice(&v);
    // 直接是 &[T], 符合接口定义
    print_slice(&v[..]);

    // &Vec<T> 支持 AsRef<[T]>
    print_slicel(&v);
    // &[T] 支持 AsRef<[T]>
    print_slicel(&v[..]);
    // Vec<T> 也支持 AsRef<[T]>
    print_slicel(v);

    let arr = [1, 2, 3, 4];
    // 数组虽没有实现 Deref, 但它的解引用就是 &[T]
    print_slice(&arr);
    print_slice(&arr[..]);
    print_slicel(&arr);
    print_slicel(&arr[..]);
    print_slicel(arr);
}

```

```
// 注意下面的泛型函数的使用
fn print_slice<T: fmt::Debug>(s: &[T]) {
    println!("{:?}", s);
}

fn print_slice1<T, U>(s: T)
where
    T: AsRef<[U]>,
    U: fmt::Debug,
{
    println!("{:?}", s.as_ref());
}
```

通过解引用，这几个和切片有关的数据结构都会获得切片的所有能力，包括：binary_search、chunks、concat、contains、start_with、end_with、group_by、iter、join、sort、split、swap 等一系列丰富的功能

切片和迭代器 Iterator

迭代器可以说是切片的孪生兄弟。切片是集合数据的视图，而迭代器定义了对集合数据的各种各样的访问操作

通过切片的 iter() 方法，我们可以生成一个迭代器，对切片进行迭代

看一个例子：对 Vec 使用 iter() 方法，并进行各种 map / filter / take 操作

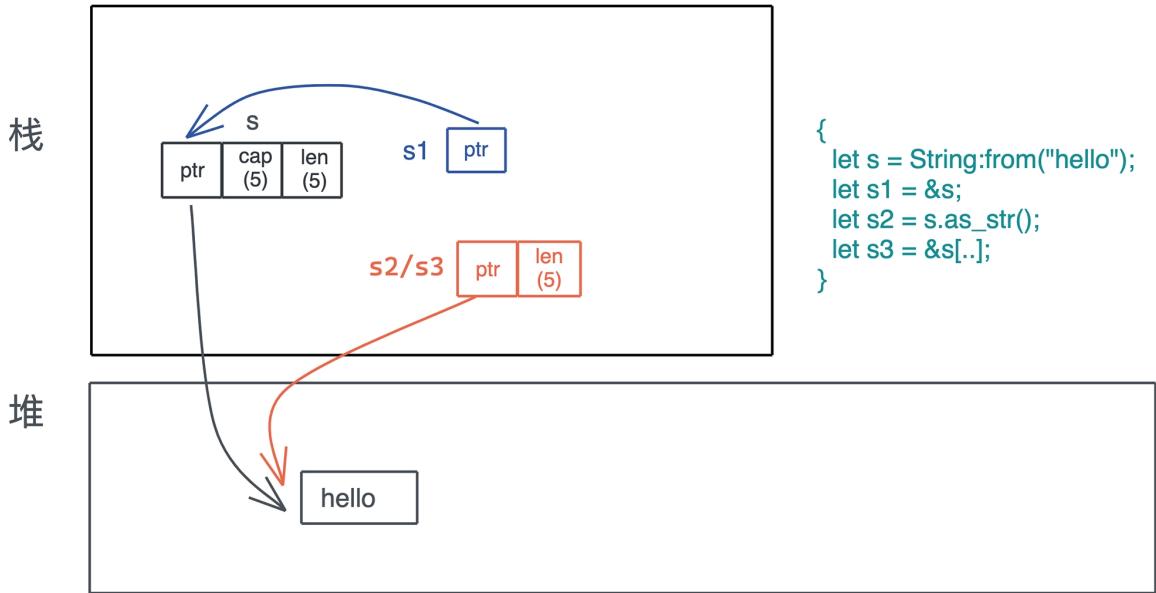
```
fn main() {
    // 这里 Vec<T> 在调用 iter() 时被解引用成 &[T]，所以可以访问 iter()
    let result = vec![1, 2, 3, 4]
        .iter()
        .map(|v| v * v)
        .filter(|v| *v < 16)
        .take(1)
        .collect::<Vec<_>>();

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

Iterator 大部分方法都返回一个实现了 Iterator 的数据结构，所以可以这样一路链式下去，在 Rust 标准库中，这些数据结构被称为 Iterator Adapter。

在 collect() 执行的时候，它实际试图使用 FromIterator 从迭代器中构建一个集合类型，这会不断调用 next() 获取下一个数据；
此时的 Iterator 是 Take，Take 调自己的 next()，也就是它会调用 Filter 的 next()；
Filter 的 next() 实际上调用自己内部的 iter 的 find()，此时内部的 iter 是 Map，find() 会使用 try_fold()，它会继续调用 next()，也就是 Map 的 next()；
Map 的 next() 会调用其内部的 iter 取 next() 然后执行 map 函数。而此时内部的 iter 来自 vec。

特殊的切片 &str，String是一个特殊的Vec,在其之上做切片也是&str



极客时间

String 在解引用时会转换成&str

字符列表和字符串的区别：

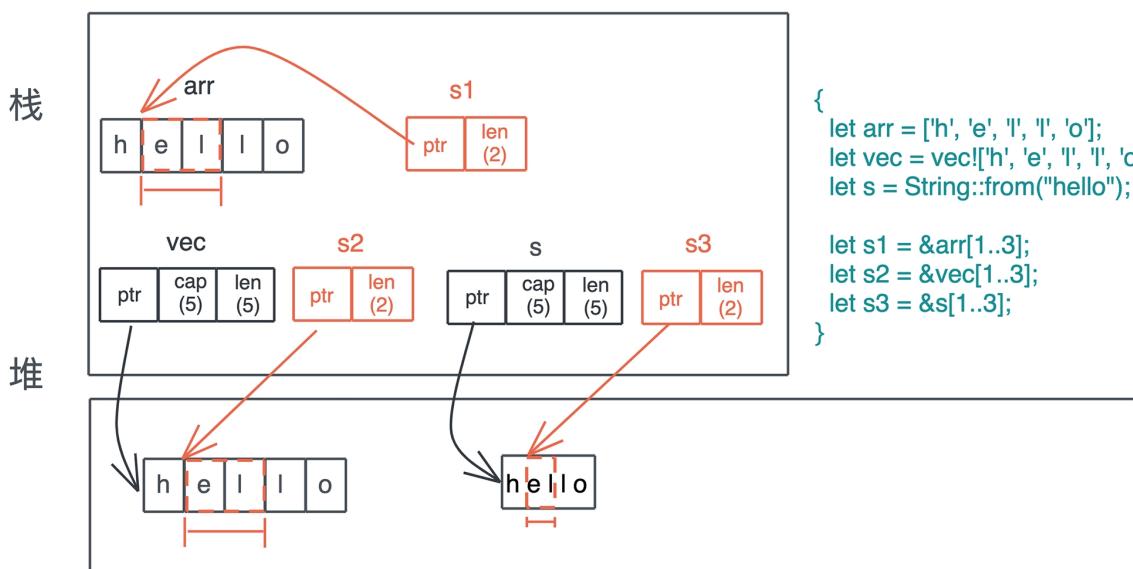
```

use std::iter::FromIterator;

fn main() {
    let arr = ['h', 'e', 'l', 'l', 'o'];
    let vec = vec!['h', 'e', 'l', 'l', 'o'];
    let s = String::from("hello");
    let s1 = &arr[1..3];
    let s2 = &vec[1..3];
    // &str 本身就是一个特殊的 slice
    let s3 = &s[1..3];
    println!("s1: {:?}", s1, s2: {:?}", s3: {:?}", s1, s2, s3);

    // &[char] 和 &[char] 是否相等取决于长度和内容是否相等
    assert_eq!(s1, s2);
    // &[char] 和 &str 不能直接对比，我们把 s3 变成 Vec<char>
    assert_eq!(s2, s3.chars().collect::<Vec<_>>());
    // &[char] 可以通过迭代器转换成 String, String 和 &str 可以直接对比
    assert_eq!(String::from_iter(s2), s3);
}

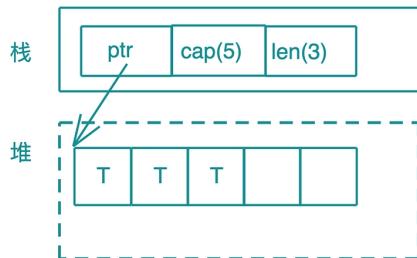
```



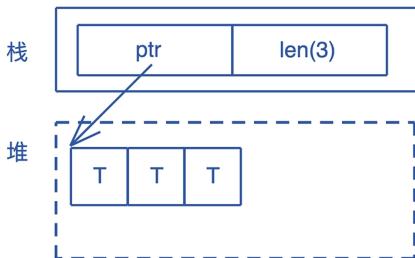
极客时间

Box<[T]> 和切片的引用 &[T] 也很类似：它们都是在栈上有一个包含长度的胖指针，指向存储数据的内存位置。区别是：Box<[T]> 只会指向堆，&[T] 指向的位置可以是栈也可以是堆；此外，Box<[T]> 对数据具有所有权，而 &[T] 只是一个借用

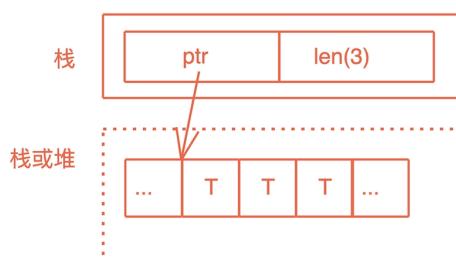
Vec<T>



Box<[T]>



切片引用：&[T] / &mut [T]



极客时间

```

use std::ops::Deref;

fn main() {
    let mut v1 = vec![1, 2, 3, 4];
    v1.push(5);
    println!("cap should be 8: {}", v1.capacity());

    // 从 Vec<T> 转换成 Box<[T]>, 此时会丢弃多余的 capacity
    let b1 = v1.into_boxed_slice();
    let mut b2 = b1.clone();

    let v2 = b1.into_vec();
    println!("cap should be exactly 5: {}", v2.capacity());

    assert!(b2.deref() == v2);

    // Box<[T]> 可以更改其内部数据, 但无法 push
    b2[0] = 2;
    // b2.push(6);
    println!("b2: {:?}", b2);

    // 注意 Box<[T]> 和 Box<[T; n]> 并不相同
    let b3 = Box::new([2, 2, 3, 4, 5]);
    println!("b3: {:?}", b3);

    // b2 和 b3 相等, 但 b3.deref() 和 v2 无法比较
    assert!(b2 == b3);
    // assert!(b3.deref() == v2);
}

```

Vec 可以通过 `into_boxed_slice()` 转换成 `Box<[T]>`, `Box<[T]>` 也可以通过 `into_vec()` 转换回 Vec

```

use std::ops::Deref;

fn main() {
    let mut v1 = vec![1, 2, 3, 4];
    v1.push(5);
    println!("cap should be 8: {}", v1.capacity());

    // 从 Vec<T> 转换成 Box<[T]>, 此时会丢弃多余的 capacity
    let b1 = v1.into_boxed_slice();
    let mut b2 = b1.clone();

    let v2 = b1.into_vec();
    println!("cap should be exactly 5: {}", v2.capacity());

    assert!(b2.deref() == v2);
}

```

```

// Box<[T]> 可以更改其内部数据, 但无法 push
b2[0] = 2;
// b2.push(6);
println!("b2: {:?}", b2);

// 注意 Box<[T]> 和 Box<[T; n]> 并不相同
let b3 = Box::new([2, 2, 3, 4, 5]);
println!("b3: {:?}", b3);

// b2 和 b3 相等, 但 b3.deref() 和 v2 无法比较
assert!(b2 == b3);
// assert!(b3.deref() == v2);
}

```

所以, 当我们需要在堆上创建固定大小的集合数据, 且不希望自动增长, 那么, 可以先创建 Vec, 再转换成 Box<[T]>

2.6.3.3.1 共享容器

访问方式	数据	不可变借用	可变借用
单一所有权	T	&T	&mut T
单线程	Rc<T>	&Rc<T>	无法得到可变借用
	Rc<RefCell<T>>	v.borrow()	v.borrow_mut()
共享所有权	Arc<T>	&Arc<T>	无法得到可变借用
	Arc<Mutex<T>>	v.lock()	v.lock()
		v.read()	v.write()



内部可变性: 本质是把原始指针*mut 给开发者, 外部可变性是通过mut显式声明。

- 与继承式可变相对应 (继承式可变, 前面声明了一个不可变, 紧接着又声明了可变)
- 由核心原语UnsafeCell提供支持, UnsafeCell是Rust中唯一可以把不可变引用转为可变指针的方法
- 基于UnsafeCell,提供了Cell和RefCell, 在运行时可变借用未声明成mut的变量

```

### 容器Cell、RefCell、UnsafeCell
### 1. 容器Cell: 通过移进移出值来实现内部可变性
```
use std::cell::Cell;

```

```

struct Foo {
 x: u32,
 y: Cell<u32>, // 包裹实现了copy trait的类型
 z: Cell<String>, // 包裹未实现copy trait的类型
}

// 初始化一个不可变实例
let foo = Foo {
 x: 1,
 y: Cell::new(3),
 z: Cell::new("hello".to_string()),
};

assert_eq!(1, foo.x);
assert_eq!(3, foo.y.get());
// 没有实现copy的类型无法使用get方法获取内部值,可以看到Cell容器是通过移进移出值来实现内部可变性的
// assert_eq!("hello".to_string(), foo.z.get());

// 改变不可变实例
foo.y.set(100);
println!("y: {:?}", foo.y.get());
foo.z.set("world".to_string());
// 未实现copy的类型不可以使用get获取,但是可以使用into_inner获取
println!("z: {:?}", foo.z.into_inner());
// 实现了copy的类型既可以使用get获取,也可以使用into_inner获取
println!("y: {:?}", foo.y.into_inner());
```
#### 2. 容器RefCell: 通过borrow_mut实现可变性
// 主要是应用于一些未实现copy trait类型, 通过borrow获取值, 有运行时开销
```
use std::cell::RefCell;
// 使用vec! 宏创建不可变的动态可增长数组
let vec = vec![1, 2, 3, 4];
// vec.push(5); // 不能往不可变的数组中增加元素

let ref_vec = RefCell::new(vec); //包裹变长数组
println!("{:?}", ref_vec.borrow()); // 不可变借用打印
ref_vec.borrow_mut().push(5); // 可变借用改变
println!("{:?}", ref_vec.borrow()); // 不可变借用打印
```
#### 3. 容器UnsafeCell 是上述两种容器的底层实现

```

```

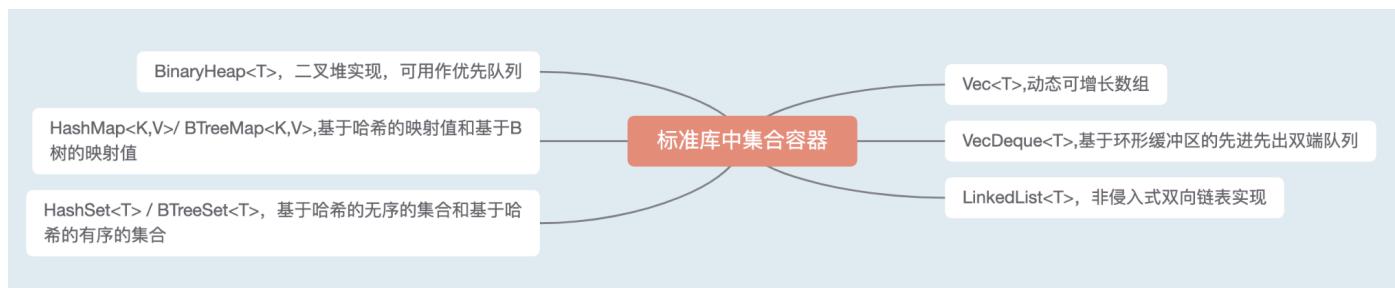
let data = RefCell::new(1);
    // 绕过了非词法作用域的检查
    // 在运行时检查
{
    //对值进行可变借用
    let mut v = data.borrow_mut();

    *v += 1;
}

```

2.6.3.3.2 集合容器

以一个 Vec 为例，当你使用完堆内存目前的容量后，还继续添加新的内容，就会触发堆内存的自动增长。有时候，集合类型里的数据不断进进出出，导致集合一直增长，但只使用了很小部分的容量，内存的使用效率很低，所以你要考虑使用，比如 shrink_to_fit 方法，来节约对内存的使用。



1. Vec 标准库导读

自身的方法：转换、排序、二分搜索、组合链接（join）、交换、追加等，

实现的trait：Default

```

pub trait Default: Sized {
    fn default() -> Self;
}

```

Rust的内存分配器可以自定义；Vec内部是一个结构体，还介绍了容量和重新分配的概念。按照预分配的成倍增加。不会自动缩减。存放于堆上。如果相对存放的位置进行优化，可以使用rust- smallvec库

集合容器：collection。什么时候用哪些？性能，迭代器，容量管理（手工使用缩减）、entry模式（连续插入）

2. LinkedList标准库导读

增删改查，node存储数据非侵入式，侵入式的不存储数据。建议尽量使用动态数组和双端队列

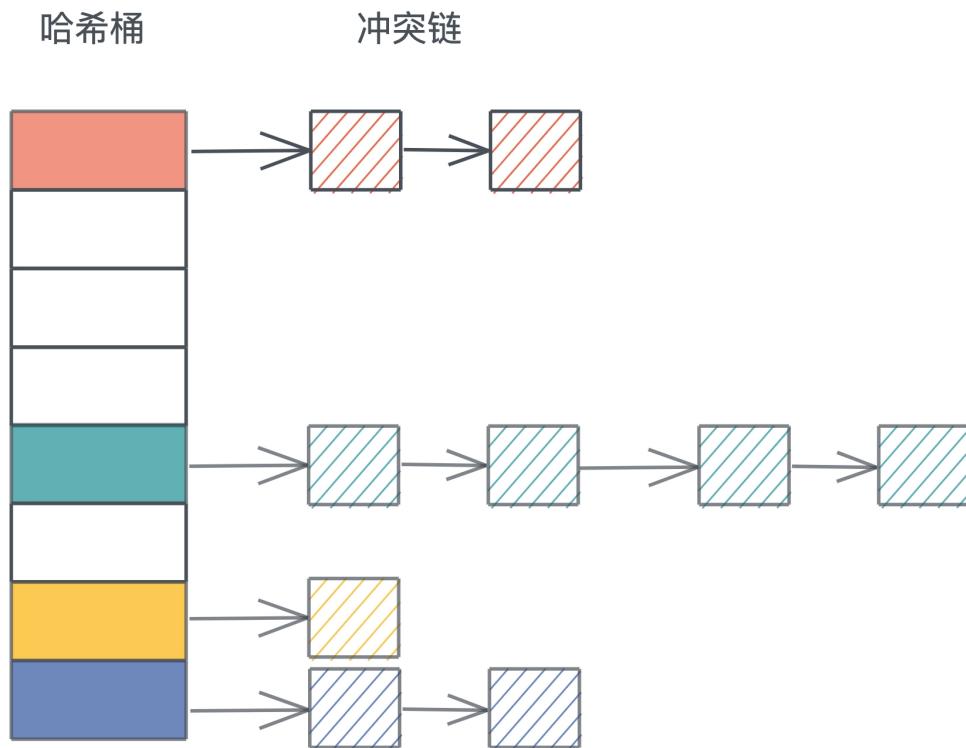
3. HashMap标准库导读

基于二次探查和SIMD查找，数据级的并行，就是单指令多数据查找

一般对哈希表的要求，哈希值如何产生，如何避免哈希冲突。Rust哈希算法默认是siphash，可以实现Hasher trait 替换哈希算法，如FnvHasher，默认可以抵抗HashDos攻击。如何解决哈希碰撞，现在是Google的SwissTable实现，和C++持平。以前用的是Robinhood，但他们都基于二次探查

哈希表如何解决哈希冲突：链地址法和开放寻址法

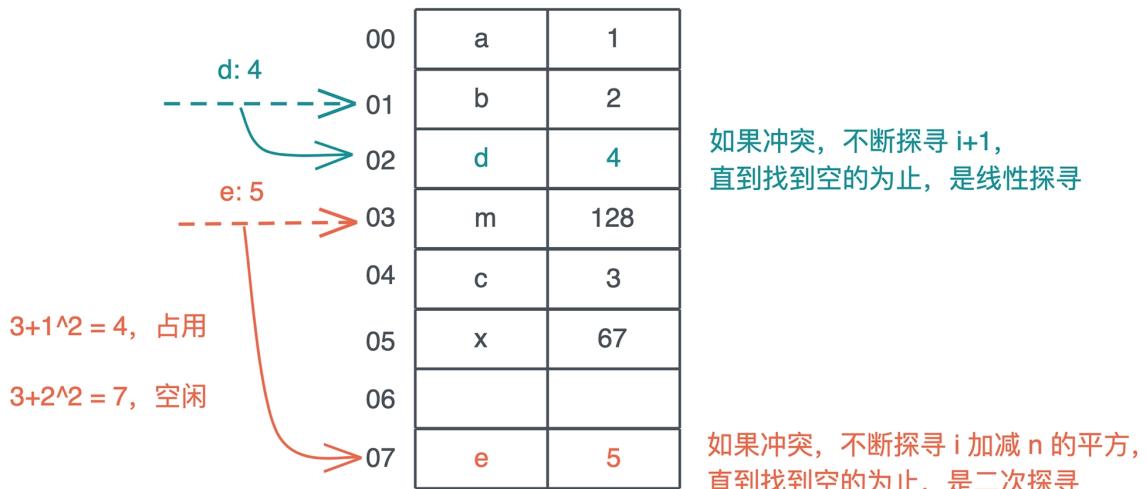
链地址法，我们比较熟悉，就是把落在同一个哈希上的数据用单链表或者双链表连接起来。这样在查找的时候，先找到对应的哈希桶（hash bucket），然后再在冲突链上挨个比较



 极客时间

缺点是缓存不友好

开放寻址法把整个哈希表看做一个大数组，不引入额外的内存，当冲突产生时，按照一定的规则把数据插入到其它空闲的位置。比如线性探寻（linear probing）在出现哈希冲突时，不断往后探寻，直到找到空闲的位置插入而二次探查，理论上是在冲突发生时，不断探寻哈希位置加减 n 的二次方，找到空闲的位置插入



极客时间

枚举在rust中相当于一个接口

方法：和动态数组差不多，实现trait: Extend，没有实现Drop，因为内部使用了算法hashbrown，实现了drop，涉及数据并行。还需要关注一个设计模式，entry，entry返回一个枚举（占位和空缺两种状态），非常聪明

Rust集合容器为什么没有统一的接口（trait）：缺乏功能泛型关联类型GAT的支持

哈希表的数据结构

```
use hashbrown::hash_map as base;

#[derive(Clone)]
pub struct RandomState {
    k0: u64,
    k1: u64,
}

pub struct HashMap<K, V, S = RandomState> {
    base: base::HashMap<K, V, S>,
}
```

HashMap 的基本使用方法

```
use std::collections::HashMap;

fn main() {
    let mut map = HashMap::new();
    explain("empty", &map);
```

```

map.insert('a', 1);
explain("added 1", &map);

map.insert('b', 2);
map.insert('c', 3);
explain("added 3", &map);

map.insert('d', 4);
explain("added 4", &map);

// get 时需要使用引用，并且也返回引用
assert_eq!(map.get(&'a'), Some(&1));
assert_eq!(map.get_key_value(&'b'), Some((&'b', &2)));

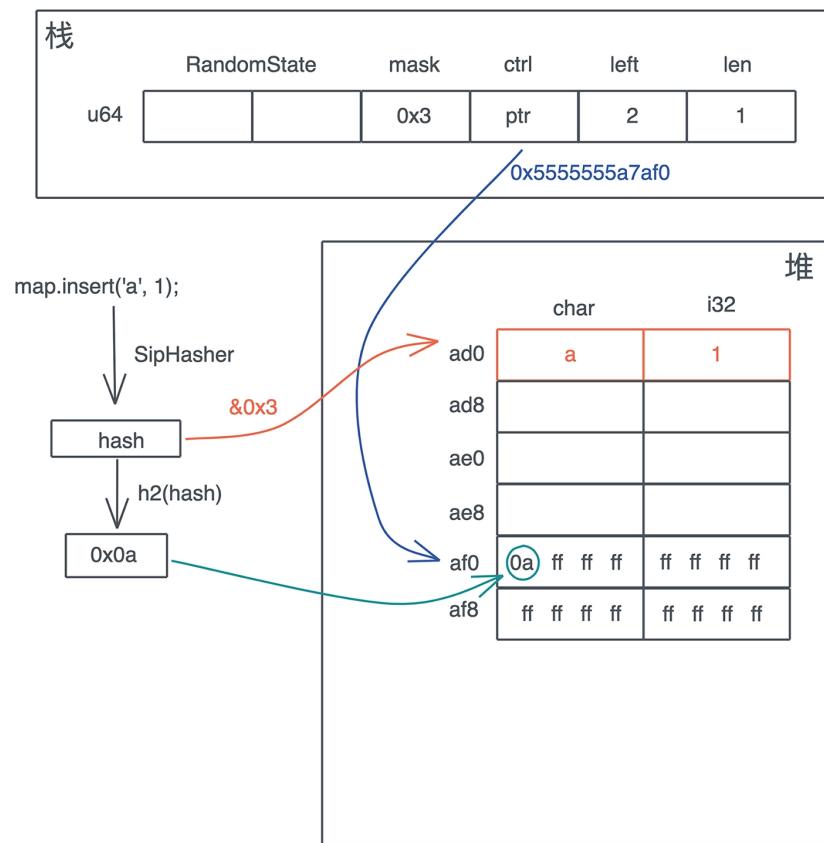
map.remove(&'a');
// 删除后就找不到了
assert_eq!(map.contains_key(&'a'), false);
assert_eq!(map.get(&'a'), None);
explain("removed", &map);
// shrink 后哈希表变小
map.shrink_to_fit();
explain("shrinked", &map);
}

fn explain<K, V>(name: &str, map: &HashMap<K, V>) {
    println!("{}: len: {}, cap: {}", name, map.len(), map.capacity());
}

```

当 HashMap::new() 时，它并没有分配空间，容量为零，随着哈希表不断插入数据，它会以 2 的幂减一的方式增长，最小是 3。当删除表中的数据时，原有的表大小不变，只有显式地调用 shrink_to_fit，才会让哈希表变小

内存布局



极客时间

哈希表会按幂扩容

删除一个值，并不需要实际清除内存，只需要将它的 ctrl byte 设回 0xff

让自定义的数据结构做 Hash key

```

use std::{
    collections::{hash_map::DefaultHasher, HashMap},
    hash::{Hash, Hasher},
};

// 如果要支持 Hash, 可以用 #[derive(Hash)], 前提是每个字段都实现了 Hash
// 如果要能作为 HashMap 的 key, 还需要 PartialEq 和 Eq
#[derive(Debug, Hash, PartialEq, Eq)]
struct Student<'a> {
    name: &'a str,
    age: u8,
}

impl<'a> Student<'a> {
    pub fn new(name: &'a str, age: u8) -> Self {
        Self { name, age }
    }
}

```

```

fn main() {
    let mut hasher = DefaultHasher::new();
    let student = Student::new("Tyr", 18);
    // 实现了 Hash 的数据结构可以直接调用 hash 方法
    student.hash(&mut hasher);
    let mut map = HashMap::new();
    // 实现了 Hash / PartialEq / Eq 的数据结构可以作为 HashMap 的 key
    map.insert(student, vec!["Math", "Writing"]);
    println!("hash: 0x{:x}, map: {:?}", hasher.finish(), map);
}

```

HashSet / BTreemap / BTreeset

简单确认元素是否在集合中，使用HashSet，存放无序集合，定义直接是HashMap<k, ()>

```

use hashbrown::hash_set as base;

pub struct HashSet<T, S = RandomState> {
    base: base::HashSet<T, S>,
}

pub struct HashSet<T, S = DefaultHashBuilder, A: Allocator + Clone = Global> {
    pub(crate) map: HashMap<T, (), S, A>,
}

```

BTreeset存放有序集合

```

use std::collections::BTreemap;

fn main() {
    let map = BTreemap::new();
    let mut map = explain("empty", map);

    for i in 0..16usize {
        map.insert(format!("Tyr {}", i), i);
    }

    let mut map = explain("added", map);

    map.remove("Tyr 1");

    let map = explain("remove 1", map);

    for item in map.iter() {
        println!("{:?}", item);
    }
}

// BTreemap 结构有 height, node 和 length

```

```
// 我们 transmute 打印之后，再 transmute 回去
fn explain<K, V>(name: &str, map: BTreeMap<K, V>) -> BTreeMap<K, V> {
    let arr: [usize; 3] = unsafe { std::mem::transmute(map) };
    println!(
        "{}: height: {}, root node: 0x{:x}, len: 0x{:x}",
        name, arr[0], arr[1], arr[2]
    );
    unsafe { std::mem::transmute(arr) }
}
```

如果你想让自定义的数据结构可以作为 BTreeMap 的 key，那么需要实现 PartialOrd 和 Ord，这两者的关系和 PartialEq / Eq 类似，PartialOrd 也没有实现自反性。同样的，PartialOrd 和 Ord 也可以通过派生宏来实现

	get(i)	insert(i)	remove(i)	append
HashMap	O(1)~	O(1)~*	O(1)~	N/A
BTreeMap	O(log(n))	O(log(n))	O(log(n))	O(n+m)
Vec	O(1)	O(n-i)*	O(n-i)	O(m)*
VecDeque	O(1)	O(min(i, n-i))*	O(min(i, n-i))	O(m)*
LinkedList	O(min(i, n-i))	O(min(i, n-i))	O(min(i, n-i))	O(1)



2.6.3.4 泛型

在Rust中,泛型是零成本的，因为会在编译期就单态化（在实际调用的位置生成具体类型相关的代码），也叫静态分发

单态化的坏处是编译速度很慢，一个泛型函数，编译器需要找到所有用到的不同类型，一个个编译，所以 Rust 编译代码的速度总被人吐槽，另一个重要因素是宏。还有一个问题：因为单态化，代码以二进制分发会损失泛型的信息

```
fn foo<T>(x: T) -> T {
    x
}
fn main() {
    assert_eq!(foo(1), 1);
    assert_eq!(foo("hello"), "hello");
}

// 上述的函数会单态化为两个不同参数类型的函数
```

```

fn foo_1(x: i32) -> i32 {
    x
}

fn foo_2(x: &'static str) -> &'static str {
    x
}

foo_1(1);
foo_2("2");

// Rust根据上下文有一定的推断能力，但是推断不出来时需要手工通过turbofish指定

// foo(1) 等价于 foo::<i32>(1);
// foo("hello") 等价于 foo::<&'static str>("hello");
}

```

2.6.3.5 特定类型

特定类型是指专门有特殊用途的类型，Rust中有两种

1. PhantomData, 幻影类型：一般用于Unsafe rust的安全抽象或者占位。

PhantomData<T>不包含任何实际数据，只用来记录类型信息

通常在 Rust 的一些底层编程中，我们需要知道一个数据结构中存储的类型，但不需要存储该类型的 actual data。这种情况下，可以使用PhantomData<T>。

使用PhantomData<T>可以帮助我们保持某些类型的类型安全，同时不需要存储任何实际数据。这有助于减少内存使用，因为我们不需要分配任何内存来存储类型相关的信息。

总的来说，PhantomData<T>是一种辅助工具，帮助我们在不存储实际数据的情况下，知道某些数据结构中存储的数据的类型。

```

use std::marker::PhantomData;

#[derive(Debug)]
struct Container<T, U> {
    data: T,
    marker: PhantomData<U>,
}

impl<T, U> Container<T, U> {
    fn new(data: T) -> Container<T, U> {
        Container {
            data,
            marker: PhantomData,
        }
    }
}

```

```
fn main() {
    // 我们知道结构体的第二个字段是u32，但是它的值是多少我们并不在意
    let _container: Container<i32, u32> = Container::new(42);
    println!("{}: {}", _container)
}
```

2. Pin, 固定类型：为了支持异步开发特意引进，防止被引用的值发生移动的类型

Pin<T>是 Rust 编程语言中的一种数据类型，它是一种指针类型，用于表示不可移动的指针。

在 Rust 中，通常情况下，对象的地址可能随着时间的推移而发生变化。但是，有时我们希望一个对象的地址保持不变，这时就可以使用Pin<T>类型。

使用Pin<T>类型可以防止 Rust 自动重新分配内存并改变指针的指向。这样，可以在编写需要一个不可移动的指针的代码时，更方便地管理内存。

总的来说，Pin<T>是一种特殊的指针类型，用于保证一个对象的地址不变，从而在某些场景下更方便地管理内存。

```
use std::pin::Pin;
use std::mem::MaybeUninit;

struct MyStruct {
    data: u32,
}

fn main() {
    let mut x = MaybeUninit::uninit();
    let x = unsafe { x.as_mut_ptr() };
    let x = unsafe { Pin::new_unchecked(x) };
    x.as_ref().write(MyStruct { data: 42 });
    println!("{}: {}", x.data);
}
```

使用 MaybeUninit 类型来创建一个未初始化的内存空间，然后将其转换为指针。最后，使用 Pin::new_unchecked 将该指针包装在 Pin<T> 类型中。通过使用 Pin<T>，可以保证该指针指向的内存空间不会发生变化，从而避免了内存安全问题。

在 Rust 中，指针指向的内存空间通常在以下情况下会发生变化：

重新分配内存：当对象的大小或类型发生变化时，Rust 可能会重新分配内存以存储该对象。这会导致原来指向该对象的指针指向不同的内存空间。

移动：在 Rust 中，对象通常是不可移动的，因此其地址不变。但是，当移动该对象时，其指针的指向将发生变化。

释放内存：当没有对象再引用一个对象时，Rust 会释放该对象占用的内存。这样，指向该对象的指针将不再指向有效的内存空间。

2.7 类型的行为

2.7.1 trait

1. trait 含义

本质上是定义了公共的方法，以便达到某个目的。任何类型想要达到某个目的，有两种方式，一种是自己定义方法去实现，另一种就是接入到trait系统中来，实现trait中一定定义好签名的方法。第二种会让代码更清楚明了和有约束性

2. trait实现

trait中也可以定义默认实现和定义关联类型（一般是返回值类型中的错误类型）

```
//单个类型的解析
let four: u32 = "4".parse().unwrap();
println!("{}", four);

// 元组结构体的解析
// 解析思路是先拿到结构体中的数字，然后使用from_str转化
use std::str::FromStr;
#[derive(Debug, PartialEq)]
struct Point(i32, i32);

#[derive(Debug, PartialEq, Eq)]
struct ParsePointError;

// 使用trait 提供的公共的方法来解析
// trait中有个方法是from_str，参数是字符串切片，返回值是目标类型实例
impl FromStr for Point {
    type Err = ParsePointError;
    fn from_str(s: &str) -> Result<Self, Self::Err> {
        // 实现过程因类型而异
        let (x, y) = s
            .strip_prefix('(')
            .and_then(|s| s.strip_suffix(')'))
            .and_then(|s| s.split_once(','))
            .ok_or(ParsePointError)?;
        let x_fromstr = x.parse::<i32>().map_err(|_| ParsePointError)?;
        let y_fromstr = y.parse::<i32>().map_err(|_| ParsePointError)?;

        // Ok()中包含了实例
        Ok(Point(x_fromstr, y_fromstr))
    }
}

let p = "(1,2)".parse::<Point>();
assert_eq!(p.unwrap(), Point(1, 2))
```

3. trait是一种特设多态（意思是一个接口多个实现，多个类型可以实现同一个trait，Go没有泛型支持）

Ad-hoc多态：一个接口多个实现

4. trait掌控了类型的行为逻辑

例如把一个变量赋值给另一个变量时，默认情况下时发生move语义，也就是发生所有权转移，原来的变量不再有数据的所有权

但是由于Copy trait的存在，凡是实现了Copy trait的类型，在发生上述行为时，所有权没有发生转移，而是为新的变量重新拷贝了一份数据（发生在栈上）

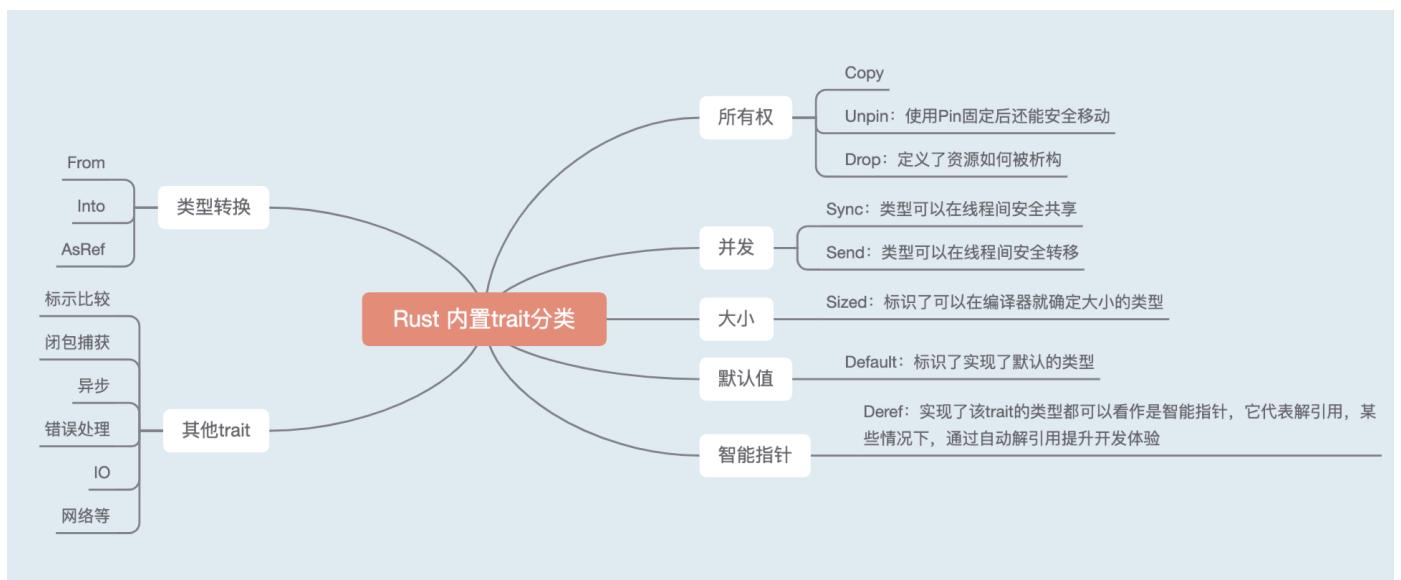
```
pub trait Copy: Clone {} // Copy trait 只是一个标记trait，虽然没有任何行为，但它可以用作 trait bound 来进行类型安全检查
```

5. trait 理论来源

Rust类型系统遵循的是仿射类型理论，即系统中用于标识内存等资源，最多只能被使用一次。Copy trait在整个逻辑的推理中起了很大作用

还有在rust编译器内使用了一个叫做chalk的trait系统，它是一个类似于逻辑编程语言Prolog的一个逻辑推理引擎

6. trait 分类



2.8 函数与闭包

参数为闭包

```
pub fn spawn<F, T>(f: F) -> JoinHandle<T>
where
    F: FnOnce() -> T,
    F: Send + 'static,
    T: Send + 'static,
```

F: FnOnce() → T, 表明 F 是一个接受 0 个参数、返回 T 的闭包。FnOnce 我们稍后再说。F: Send + 'static, 说明闭包 F 这个数据结构, 需要静态生命周期或者拥有所有权, 并且它还能被发送给另一个线程。T: Send + 'static, 说明闭包 F 返回的数据结构 T, 需要静态生命周期或者拥有所有权, 并且它还能被发送给另一个线程

2.8.1 函数与函数项

2.8.1.1 函数

函数的签名都是显式的

函数有三种类型：自由函数、关联函数和方法

函数自身是一种类型，值就是对应的代码

Rust语言中函数是一等公民，可以在函数间进行传递，也称高阶函数

2.8.1.2 函数项

```
struct A(i32, i32);
impl A {
    // 2. 关联函数
    fn sum(x: i32, y: i32) -> i32 {
        x + y
    }

    // 3. 方法
    fn math(&self) -> i32 {
        Self::sum(self.0, self.1) // 关联函数调用使用比目鱼符号
    }

    // 关联函数
    fn function_item(x: i32) -> i32 {
        x
    }
}

let a = A(1, 2);
let x = a.math();
let y = A::sum(1, 3);

// 1. 函数项构造：类型::函数/方法名构建函数项以及自由函数的直接赋值
// 2. 函数项类型：如 fn sum(i32,i32)-> i32, 就是函数签名, 同trait中的方法签名一样
let add = A::sum; // Fn item 类型
let add_math = A::math; // Fn item 类型

// 3. 函数项的使用：作为函数调用
assert_eq!(add(1, 2), A::sum(1, 2));
assert_eq!(add_math(&a), a.math());

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

```
// 4. 函数项类型本质：0大小类型，会在类型中记录函数信息
// 好处：优化函数调用

// 5. 同函数项类型一样的其他类型构造器：枚举体和单元结构体

// 5.1 函数项类型
let fn_item = A::function_item;

// 等价于
// fn function_item(_1:i32)->i32 /* */

// 5.1 枚举体
enum Color {
    R(i32),
    G(i32),
    B(i32),
}
// 等价于
// fn Color::R(_1: i32) -> Color /* */
// fn Color::G(_1: i32) -> Color /* */
// fn Color::B(_1: i32) -> Color /* */

// 5. 单元结构体
struct UintStruct(i32, i32);

// 等价于
// fn UintStruct(_1: i32, _2: i32) -> UintStruct /* */

// 6 函数项默认实现的 trait
// Copy/Clone/Send/Sync/Fn/FnMut/FnOnce

// 7 函数项可以作为函数参数（函数项可以当做变量）：函数项隐式转换为函数指针

// 定义一个类型别名作为返回值的类型（RGB是三元组的类型别名）
type RGB = (i16, i16, i16);

// 自由函数
fn color(s: &str) -> RGB {
    (1, 1, 1)
}

// 参数类型是函数指针类型的自由函数
fn show(c: fn(&str) -> RGB) {
    println!("{}:{}:", c("black"))
}

// 将函数变为函数项
let rgb = color;
```

```

// 将函数项显式转换为函数指针
let c: fn(&str) -> RGB = rgb;

// 函数指针作为另一个函数参数
show(c);
// 函数指针作为另一个函数参数, 隐式转换
show(rgb);

println!("the size of fn item {:?}" , std::mem::size_of_val(&rgb)); // 0
println!("the size of fn pointer {:?}" , std::mem::size_of_val(&c)); // 8

// 8 结论
// 8.1 函数项类型可以显式转换为函数指针类型, 也可以隐式转换, 但是因为携带了指针的信息, 所以要占用额外的空间
// 8.2 尽量使用函数项类型而不是函数指针, 以享受零大小类型的优化 (直接用, 不要作为参数传递)

```

2.8.1.3 函数名

函数名是一种表达式, 表达式的值是函数的相关信息, 比如类型名、参数类型名、生命周期等, 它的类型是函数项类型, 它是0大小类型

2.8.2 闭包

2.8.2.1 闭包和函数

函数只能使用传入的参数以及定义的局部变量, 无法捕获环境变量, 闭包可以

1. 闭包对环境变量的使用仍然遵循所有权机制
2. 闭包可以与函数指针互通
3. 闭包在作为函数返回值时要使用impl trait语法
4. 闭包可以捕获环境变量
5. 闭包的大小跟参数、局部变量都无关, 只跟捕获的变量有关
6. 闭包是存储在栈上, 并且除了捕获的数据外, 闭包本身不包含任何额外函数指针指向闭包的代码

```

fn counter(i: i32) -> impl FnMut(i32) -> i32 {
    // 1. 闭包与所有权
    // 闭包使用move关键字把环境变量所有权转移到闭包内
    // 具体执行copy还是move语义需要看具体的类型
    let s1 = "hello".to_string();
    move |s2: &str| s1 + s2;
    // println!("{}" , s1); // 不可用, move语义

    // 2. 闭包类型与函数指针类型
    // 某闭包类型: |i32| -> i32, 同函数指针非常相似
    // 某函数指针类型: fn(i32) -> i32

    // 3. 闭包与函数指针互通 (闭包作为参数)

    type RGB = (i32, i32, i32);
}

```

```

fn show(c: fn(&str) -> RGB) {
    println!("{}: {:?}", c("black"));
}

// 定义闭包: 类型 | &str | -> (i32, i32, i32), 实现了 `Fn(&str)-> RGB` trait
let c = |s: &str| (1, 2, 3);
show(c);

// 4. 闭包作为返回值
// 因为闭包是基于Trait实现的, 所以闭包作为返回值时使用的是impl trait语法
// 返回值是i32 trait的类型, 其中 FnMut(i32)->i32 这一整块作为一个trait, 属于静态分发
// impl FnMut(i32) -> i32 代表返回的是一个实现了FnMut(i32)
let closure = move |n| n + i;
closure

}

let mut f = counter(21);
assert_eq!(42, f(21))

```

如果不使用 move 转移所有权, 闭包会引用上下文中的变量, 这个引用受借用规则的约束, 所以只要编译通过, 那么闭包对变量的引用就不会超过变量的生命周期, 没有内存安全问题。

如果使用 move 转移所有权, 上下文中的变量在转移后就无法访问, 闭包完全接管这些变量, 它们的生命周期和闭包一致, 所以也不会有内存安全问题。

2.8.2.2 闭包实现原理

1. Rust闭包的实现与所有权机制在语义上保持了统一。闭包的三种使用场景与所有权语义三件套相匹配
2. 闭包实际上是编译器的语法糖, 也就是说, 当创建一个闭包时, 编译器会解析闭包, 并且生成一个匿名结构体, 该结构体有个泛型变量, 主要用于存储捕获的自由变量

```

// 请将下列模块属性放置在执行文件顶部
#![feature(unboxed_closures, fn_traits)]
// 按使用场景

// 1. 未捕捉环境变量 对应所有权
let c1 = || println!("hello");
c1();

// 等价于创建了一个闭包结构体, 并未闭包结构体实现了 call_once方法
// 对闭包的调用实际上是对相应trait中的方法进行调用, 但使用的名字不同, 类似在使用函数项一样
// 注意call_once方法的第一个参数是self, 代表它会消耗结构体, 需要拥有所有权

struct Closure1<T> {
    env_var: T,
}

/*
### 标准库 FnOnce trait的定义

```

```

pub trait FnOnce<Args>
where
    Args: Tuple, {
    type Output;
    extern "rust-call" fn call_once(mut self, args: Args) -> Self::Output;
}
*/

```

// 为类型实现trait

```

impl<T> FnOnce<()> for Closure1<T> {
    type Output = ();
    extern "rust-call" fn call_once(self, args: ()) -> () {
        println!("hello");
    }
}

```

// 调用

```

let c1 = Closure1 { env_var: () };
c1.call_once();

```

// 2. 可修改环境变量 对应可变借用 &mut T

```

let mut arr = [1, 2, 3];
let mut c2 = |i| {
    arr[0] = i;
    println!("{}: {}", i, arr)
};

```

```

c2(100);

```

// 等价于

// 继承式的实现实际上是所有权一致性的体现

// 闭包实例至少需要一个消耗自身的方法

```

struct Closure2 {
    env_var: [i32; 3],
}

```

```

/*
### 标准库 FnOnce trait的定义
pub trait FnOnce<Args> {
    type Output;
    extern "rust-call" fn call_once(mut self, args: Args) -> Self::Output;
}
*/

```

// 为类型实现 FnOnce trait

```

impl FnOnce<(i32,)> for Closure2 {
    type Output = ();
}

```

```

    extern "rust-call" fn call_once(mut self, args: (i32,)) -> () {
        self.env_var[0] = args.0;
        println!("{:?}", self.env_var);
    }
}

/*
### 标准库 FnMut trait的定义
pub trait FnMut<Args>:FnOnce<Args> {
where
    Args:Tuple, {
        extern "rust-call" fn call_mut(&mut self, args: Args) -> Self::Output;
    }
}
*/

// 为类型实现 FnMut trait
impl FnMut<(i32,)> for Closure2 {
    extern "rust-call" fn call_mut(&mut self, args: (i32,)) -> () {
        self.env_var[0] = args.0;
        println!("{:?}", self.env_var);
    }
}

// 调用

let arr2 = [1, 2, 3];
let mut c2 = Closure2 { env_var: arr2 };
c2.call_mut((0,)); //可变引用调用
c2.call_once((1,)); //消耗式调用

// 3. 未修改环境变量 对应不可变借用 &T
let answer = 42;
let c3 = || {
    println!("{:?}", answer);
};

// 等价于

struct Closure3 {
    env_var: i32,
}

/*
### 标准库 FnOnce trait的定义
pub trait FnOnce<Args>
where
    Args:Tuple, {

```

```
    type Output;
    extern "rust-call" fn call_once(mut self, args: Args) -> Self::Output;
}
*/
// 为类型实现 FnOnce trait
impl FnOnce<()> for Closure3 {
    type Output = ();
    extern "rust-call" fn call_once(mut self, args: ()) -> () {
        println!("{}:?", self.env_var);
    }
}

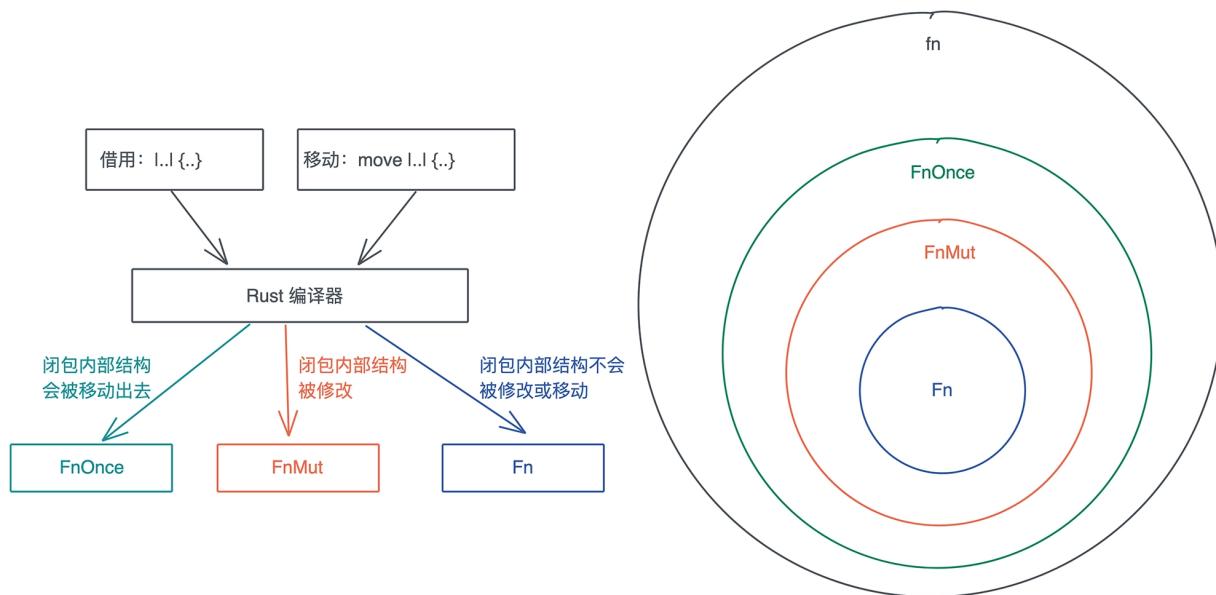
/*
### 标准库 FnMut trait的定义
pub trait FnMut<Args>:FnOnce<Args> {
where
    Args:Tuple, {
        extern "rust-call" fn call_mut(&mut self, args: Args) -> Self::Output;
    }
}
// 为类型实现 FnMut trait
impl FnMut<()> for Closure3 {
    extern "rust-call" fn call_mut(&mut self, args: ()) -> () {
        println!("{}:?", self.env_var);
    }
}

/*
### 标准库 Fn trait的定义
pub trait Fn<Args>:FnMut<Args>
where
    Args:Tuple, {
        extern "rust-call" fn call(&self, args: Args) -> Self::Output;
    }
}
impl Fn<()> for Closure3 {
    extern "rust-call" fn call(&self, args: ()) -> () {
        println!("{}:?", self.env_var);
    }
}

let mut c3 = Closure3 { env_var: 42 };
c3.call(); // 不可变引用
c3.call_mut(); //可变引用
c3.call_once(); //消耗式调用
```

2.8.2.3 闭包的类型

1. 没有捕获变量，则实现 FnOnce
2. 修改捕获变量，则实现 FnMut
3. 未改捕获变量，则实现 Fn



2.8.2.4 特殊情况

1. 编译器会把FnOnce当成fn(T)函数指针区看待
2. Fn/FnMut/FnOnce 关系依次继承，对应所有权语义三件套
3. 唯一不可变借用

FnOnce 只能调用一次；
FnMut 允许在执行时修改闭包的内部数据，可以执行多次；
Fn 不允许修改闭包的内部数据，也可以执行多次。

2.8.2.5 逃逸闭包和非逃逸闭包

```
```  
// 逃逸闭包
fn c_mut() -> impl FnMut(i32) -> [i32; 3] {
 let mut arr = [1, 2, 5];
 move |n| {
 arr[2] = n;
 arr
 }
}
```

```

let i = 42;

let mut arr_closure = c_mut();
println!("{:?}", arr_closure(i));

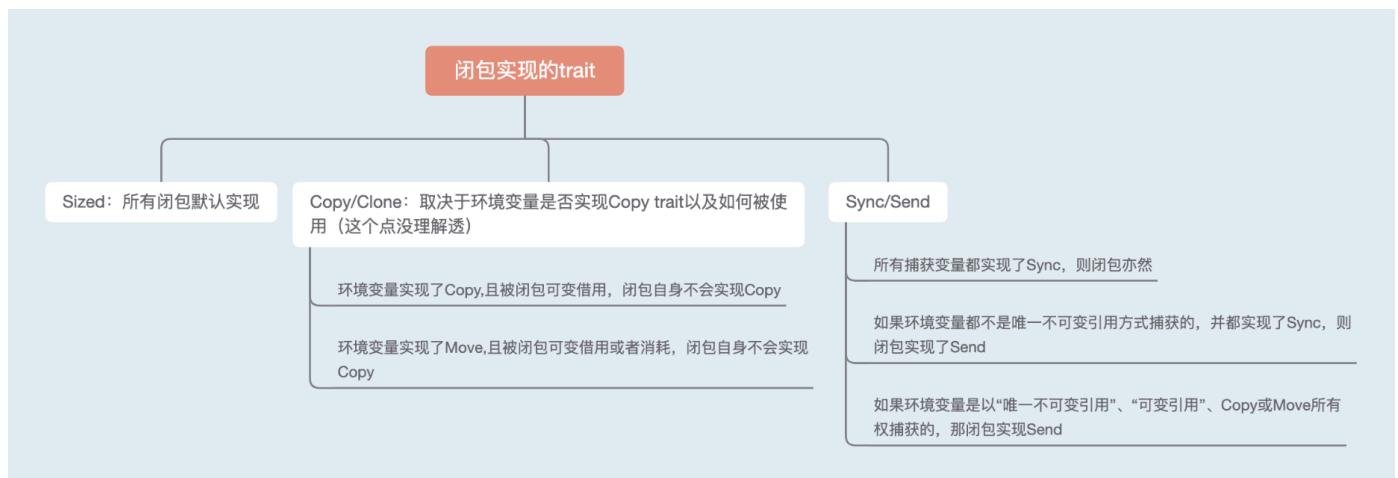
// 被捕获类型不支持Copy,无法返回闭包,主要是为了防止悬垂引用

/*
fn c_mut2() -> impl for<'a> FnMut(&'a str) -> String {
 // 当闭包捕获了未实现Copy trait 的类型时,无法返回
 let mut s = "hello".to_string();
 move |i| {
 s += i;
 s
 }
}
*/

```

### 2.8.2.6 闭包实现的trait

我们已知闭包会生成匿名结构体，那默认实现了哪些trait呢



只有当闭包的捕获列表中的所有变量都实现了 `Clone` 和 `Copy` 时，闭包才会实现这两个 trait

```

// 闭包自身实现了Fn Copy trait
fn foo<F: Fn() + Copy>(f: F) {
 f()
}

let s = "hello".to_owned();

// 不可变借用
let f = || {
 println!("{} ", s);
};

foo(f);

```

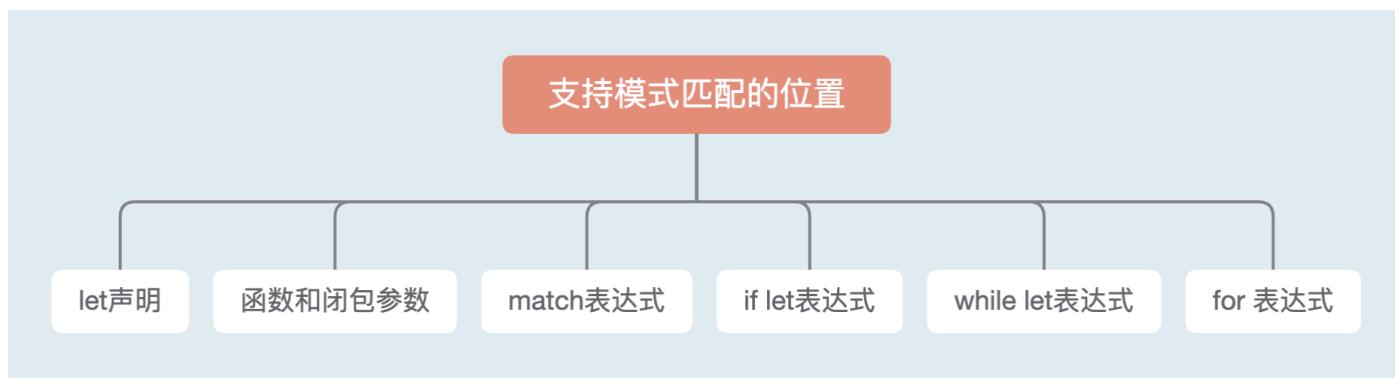
```
// 消耗
let g = move || {
 println!("{}", s);
};

//foo(g); // 未实现copy trait
```

## 2.9 模式匹配

模式匹配是一种结构性的解构与构造的语义相对。Rust 的模式匹配是一个很重要的语言特性，被广泛应用在状态机处理、消息处理和错误处理中

### 2.9.1 模式匹配位置



### 2.9.2 模式匹配的两种类型

1. 可辩驳
2. 不可辩驳

```
// 1. let 声明中的匹配
struct Point {
 x: i32,
 y: i32,
}

let (a, b) = (1, 2);

let Point { x, y } = Point { x: 3, y: 4 };

assert_eq!(1, a);
assert_eq!(2, b);
assert_eq!(3, x);
assert_eq!(4, y);

// 2. 函数与闭包参数

fn sum(x: String, ref y: String) -> String {
 x + y
}
```

```

}

let s = sum("1".to_owned(), "2".to_owned());
assert_eq!(s, "12".to_owned());

// 辅助理解 ref

{
 let a = 42;
 let ref b = a;
 let c = &a;

 assert_eq!(b, c);

 let mut a = [1, 2, 3];
 let ref mut b = a;

 b[0] = 0;

 assert_eq!(a, [0, 2, 3])
}

// 3. match 表达式

fn check_option(opt: Option<i32>) {
 match opt {
 Some(p) => println!("has value {:?}", p),
 None => println!("has no value"),
 }
}

/*
fn hand_result(res: i32) -> Result<i32, dyn Error> {
 do_something(res)?;

 // 问号等价于

 match do_something(res) {
 Ok(o) => Ok(o),
 Err(e) => return SomeError(e),
 }
}

let arr = [1, 2, 3];
match arr {
 [1, ..] => "start with one",
 [a, b, c] => "not start with one",
};

```

```

let v = vec![1, 2, 3];
match v[...] {
 [a, b] => "not match",
 [a, b, c] => "matched",
 _ => "",
};

// if let 表达式

let x: &Option<i32> = &Some(3);

// 编译器自动使用ref
if let Some(y) = x {
 y;
}

```

## 2.10 智能指针

智能指针是一个表现行为很像指针的数据结构，但除了指向数据的指针外，它还有元数据以提供额外的处理能力。智能指针一定是一个胖指针，但胖指针不一定是一个智能指针。比如 `&str` 就只是一个胖指针，它有指向堆内存字符串的指针，同时还有关于字符串长度的元数据

除了 `String`，在之前的课程中我们遇到了很多智能指针，比如用于在堆上分配内存的 `Box` 和 `Vec`、用于引用计数的 `Rc` 和 `Arc`。很多其他数据结构，如 `PathBuf`、`Cow<'a, B>`、`MutexGuard`、`RwLockReadGuard` 和 `RwLockWriteGuard` 等也是智能指针

### 2.10.1 在堆上分配内存：Box

从语义上Rust的类型分为值语义和指针语义。存储在栈上的就是值语义，在语义层面上就是一种值。动态字符串和动态数组会在运行时增长，它们实际上属于指针语义，传递时传递的是存储在栈上的指针而不是全部数据

`Box`是Safe Rust 中唯一的堆内存分配方式，在使用 `Box` 分配堆内存的时候要注意，`Box::new()` 是一个函数，所以传入它的数据会出现在栈上，再移动到堆上，非常大的结构时就容易出问题

```
pub struct Box<T: ?Sized, A: Allocator = Global>(Unique<T>, A)
```

```

pub struct Unique<T: ?Sized> {
 pointer: *const T,
 // NOTE: this marker has no consequences for variance, but is necessary
 // for dropck to understand that we logically own a `T`.
 //
 // For details, see:
 // https://github.com/rust-lang/rfcs/blob/master/text/0769-sound-generic-
 drop.md#phantom-data
 _marker: PhantomData<T>,
}

```

```

let x: Box<i32> = Box::new(42);
// 通过解引用来获取所包裹的值，指针都可以解引用
let y = *x;

assert_eq!(y, 42)

```

## 2.10.2 Box 内存管理机制

借鉴了Cpp的RAII, Box实现了Drop trait。当变量离开作用域时，自动调用析构函数（drop函数）销毁值

```

// 标准库中的drop实现，编译器的行为
/*
unsafe impl<#[may_dangle] T: ?Sized> Drop for Box<T> {
 fn drop(&mut self) {
 FIXME:Do nothing, drop is currently performed by compiler
 }
}
*/

```

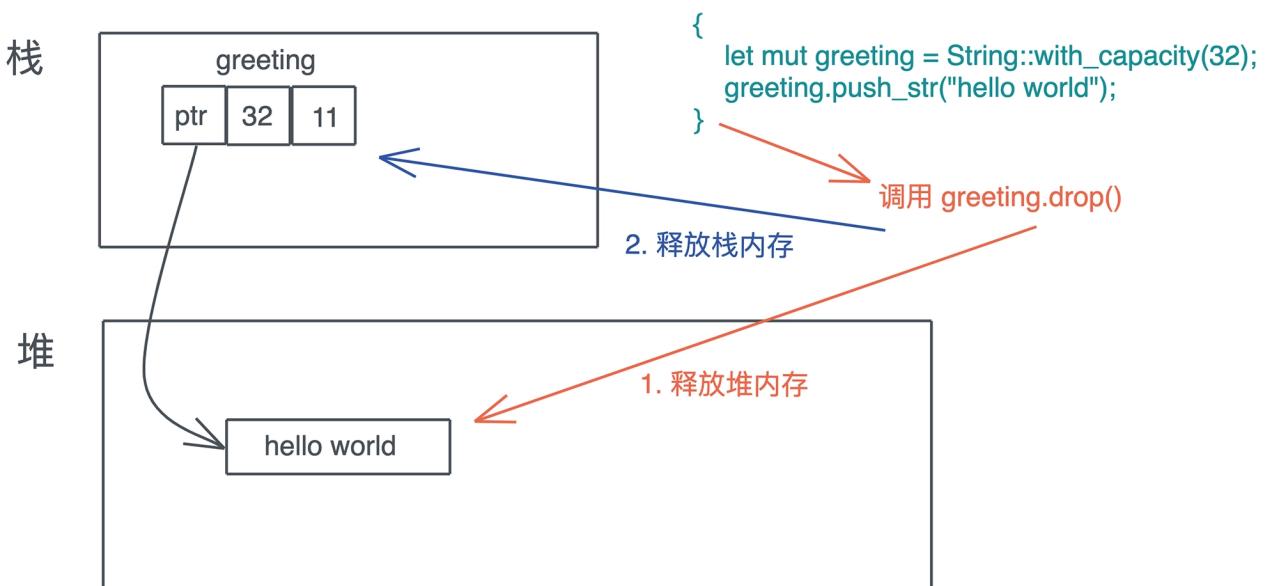
Rust中的值如何被销毁的？

```

pub trait Drop {
 fn drop(&mut self);
}

```

这里用到了Drop trait。Drop trait类似面向对象编程中的析构函数，当一个值要被释放，它的Drop trait会被调用。比如下面的代码，变量greeting是一个字符串，在退出作用域时，其drop()函数被自动调用，释放堆上包含“hello world”的内存，然后再释放栈上的内存：



结构体在调用 drop() 时，会依次调用每一个域的 drop() 函数，如果域又是一个复杂的结构或者集合类型，就会递归下去，直到每一个域都释放干净

释放堆内存，简单的调用Drop trait

但是可以为自定义类型手动实现Drop trait，让值的释放更受控。它还可以释放任何资源，比如 socket、文件、锁等等。

```
use std::fs::File;
use std::io::prelude::*;

fn main() -> std::io::Result<()> {
 let mut file = File::create("foo.txt")?;
 file.write_all(b"hello world")?;
 Ok(())
}
```

什么时候需要Drop trait

第一种是希望在数据结束生命周期的时候做一些事情，比如记日志

第二种是需要对资源回收的场景。编译器并不知道你额外使用了哪些资源，也就无法帮助你 drop 它们。比如说锁资源的释放，在 MutexGuard 中实现了 Drop 来释放锁资源

```
impl<T: ?Sized> Drop for MutexGuard<'_, T> {
 #[inline]
 fn drop(&mut self) {
 unsafe {
 self.lock.poison.done(&self.poison);
 self.lock.inner.raw_unlock();
 }
 }
}
```

注意的是，Copy trait 和 Drop trait 是互斥的，两者不能共存，当你尝试为同一种数据类型实现 Copy 时，也实现 Drop，编译器就会报错。这其实很好理解：Copy 是按位做浅拷贝，那么它会默认拷贝的数据没有需要释放的资源；而 Drop 恰恰是为了释放额外的资源而生的

|      |                |           |
|------|----------------|-----------|
| 检查时间 | 编译时            | 运行时       |
| 检查效果 | 高效，但不灵活        | 灵活，但有额外负担 |
| 检查位置 | 栈              | 堆         |
| 检查机制 | borrow checker | 引用计数      |



## 2.10.3 智能指针

在Rust中，trait决定了类型的行为。所以智能指针和Deref trait、Drop trait相关

二者都实现或者实现其一都是智能指针，所以智能指针在Rust中有两种语义，自动解引用（提升开发体验）和自动管理内存（安全无忧）

只实现Deref trait：拥有指针语义，Deref赋予了类型的指针行为，通常在Rust中代表了Move语义，基本是分配在堆上的数据

只实现Drop trait：拥有内存自动管理机制，Deref赋予了类型的析构行为

### 2.10.3.1 智能指针与Deref trait

```
// 1. 自动解引用 点调用操作
// 自定义一个类型
#[derive(Copy, Clone)]
struct User {
 name: &'static str,
}

impl User {
 fn name(&self) {
 println!(":{}{}", self.name);
 }
}

// 调用

let u = User { name: "Alex" };
// 原来的调用方式

println!("{}", u.name);
```

```
// 使用自定义的智能指针包裹
let y = MySP::new(u);

// 包裹后的调用方式
// 这里智能指针实际上自动进行了解引用，获取了里面的值，然后用值进行关联函数调用

println!("{}", y.name);
// 手动解引用
let z = *y;

println!("{}", z.name);

// 结论：使用类型直接调用字段 = 智能指针解引用调用 = 手动解引用调用

// 2. 自动解引用 函数参数
fn takes_str(s: &str) {
 println!("{}", s);
}

let s = String::from("hello");
// String 也是一个智能指针，它包裹了 str
// 自动解引用为原始类型str后要再加&

// 调用
takes_str(&s);

// 标准库中为String类型实现了Deref trait
/*
impl ops::Deref for String {
 type Target = str;

 #[inline]
 fn deref(&self) -> &str {
 unsafe { str::from_utf8_unchecked(&self.vec) }
 }
}
 */

// 自动解引用需要注意的地方
// 使用*x 解引用等价于 * (x.deref())

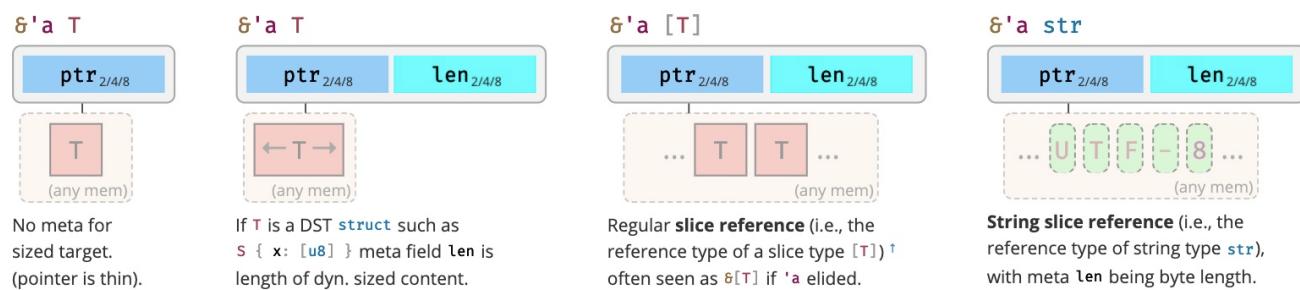
let s = Box::new("world");
let ref_s1 = *s;
let ref_s2 = *(s.deref());

assert_eq!(ref_s1, ref_s2);

// 自动解引用等价于 x.deref()
```

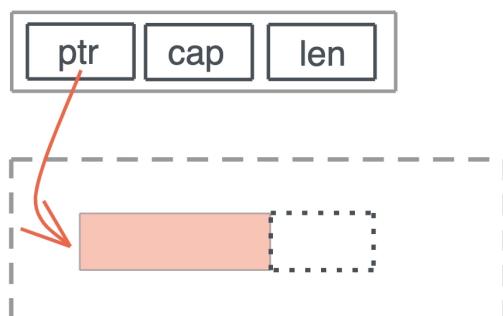
## 2.10.3.2 标准库中的智能指针

Rust类型备忘清单：<https://cheats.rs/#data-layout>

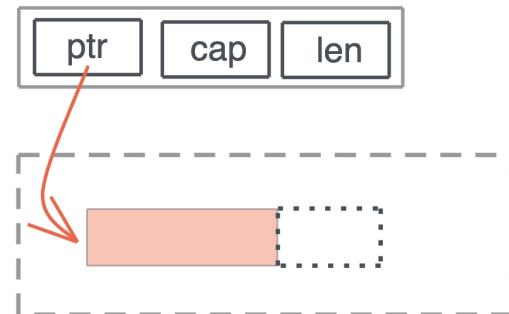


Vec是胖指针

String(Vec<u8>)



Vec<T>



极客时间

Rc<T>和Arc<T>(共享所有权容器，实际内部有个引用计数器，分别用在单线程和多线程)：Drop and Deref，实现的目的和Box<T>相同

标准库中的智能指针

HashMap<K,V>:Drop

Box<T>: Drop and Deref

Vec<T>和String: Drop and Deref

```
// 标准库中给泛型T实现的 Deref trait
/*
impl<T: ?Sized> const Deref for &T {
 type Target = T;

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

impl<T: ?Sized> !DerefMut for &T {
```

```

** 在日常开发中非常实用
** 当我们拥有可变引用 T 时如果还想使用 T，则可以自动解引用，比如点调用
impl<T: ?Sized> const Deref for &mut T {
 type Target = T;

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

```

## 2.10.4 Cow<'a, B>

Cow 是 Rust 下用于提供写时克隆（Clone-on-Write）的一个智能指针，它跟虚拟内存管理的写时复制（Copy-on-write）有异曲同工之妙：包裹一个只读借用，但如果调用者需要所有权或者需要修改内容，那么它会 clone 借用的数据

```

pub enum Cow<'a, B> where B: 'a + ToOwned + ?Sized {
 Borrowed(&'a B),
 Owned(<B as ToOwned>::Owned),
}

```

何时用？如果 Cow<'a, B> 中的 Owned 数据类型是一个需要在堆上分配内存的类型，如 String、Vec 等，还能减少堆内存分配的次数。相对于栈内存的分配释放来说，堆内存的分配和释放效率要低很多，其内部还涉及系统调用和锁，减少不必要的堆内存分配是提升系统效率的关键手段

使用案例：

```

use std::borrow::Cow;

use url::Url;
fn main() {
 let url = Url::parse("https://tyr.com/rust?
page=1024&sort=desc&extra=hello%20world").unwrap();

 let mut pairs = url.query_pairs();
 assert_eq!(pairs.count(), 3);

 let (mut k, v) = pairs.next().unwrap();

 println!("Key: {}, value: {}", k, v);

 print_pairs((k, v));

 print_pairs(pairs.next().unwrap());

 print_pairs(pairs.next().unwrap());
}

```

```

}

fn print_pairs(pairs: (Cow<str>, Cow<str>)) {
 println!(
 "key: {}, value: {}",
 show_pairs(pairs.0),
 show_pairs(pairs.1)
);
}

fn show_pairs(cow: Cow<str>) -> String {
 match cow {
 Cow::Borrowed(v) => format!("Borrowed :{}", v),
 Cow::Owned(v) => format!("Owned :{}", v),
 }
}

```

其它第三方库对Cow的支持

```

use serde::Deserialize;
use std::borrow::Cow;

#[derive(Debug, Deserialize)]
struct User<'input> {
 #[serde(borrow)]
 name: Cow<'input, str>,
 age: u8,
}

fn main() {
 let input = r#"{"name": "Tyr", "age": 18}"#;
 let user: User = serde_json::from_str(input).unwrap();

 match user.name {
 Cow::Borrowed(x) => println!("borrowed {}", x),
 Cow::Owned(x) => println!("owned {}", x),
 }
}

```

## 2.10.5 MutexGuard

它不但通过 Deref 提供良好的用户体验，还通过 Drop trait 来确保，使用到的内存以外的资源在退出时进行释放

```

pub fn lock(&self) -> LockResult<MutexGuard<'_, T>> {
 unsafe {
 self.inner.raw_lock();
 MutexGuard::new(self)
 }
}

```

```

// 这里用 must_use, 当你得到了却不使用 MutexGuard 时会报警
#[must_use = "if unused the Mutex will immediately unlock"]
pub struct MutexGuard<'a, T: ?Sized + 'a> {
 lock: &'a Mutex<T>,
 poison: poison::Guard,
}

impl<T: ?Sized> Deref for MutexGuard<'_, T> {
 type Target = T;

 fn deref(&self) -> &T {
 unsafe { &*self.lock.data.get() }
 }
}

impl<T: ?Sized> DerefMut for MutexGuard<'_, T> {
 fn deref_mut(&mut self) -> &mut T {
 unsafe { &mut *self.lock.data.get() }
 }
}

impl<T: ?Sized> Drop for MutexGuard<'_, T> {
 #[inline]
 fn drop(&mut self) {
 unsafe {
 self.lock.poison.done(&self.poison);
 self.lock.inner.raw_unlock();
 }
 }
}

```

当 MutexGuard 结束时，Mutex 会做 unlock，这样用户在使用 Mutex 时，可以不必关心何时释放这个互斥锁

```

use lazy_static::lazy_static;
use std::borrow::Cow;
use std::collections::HashMap;
use std::sync::{Arc, Mutex};
use std::thread;
use std::time::Duration;

```

```

// lazy_static 宏可以生成复杂的 static 对象
lazy_static! {
 // 一般情况下 Mutex 和 Arc 一起在多线程环境下提供对共享内存的使用
 // 如果你把 Mutex 声明成 static, 其生命周期是静态的, 不需要 Arc
 static ref METRICS: Mutex<HashMap<Cow<'static, str>, usize>> =
 Mutex::new(HashMap::new());
}

fn main() {
 // 用 Arc 来提供并发环境下的共享所有权 (使用引用计数)
 let metrics: Arc<Mutex<HashMap<Cow<'static, str>, usize>>> =
 Arc::new(Mutex::new(HashMap::new()));
 for _ in 0..32 {
 let m = metrics.clone();
 thread::spawn(move || {
 let mut g = m.lock().unwrap();
 // 此时只有拿到 MutexGuard 的线程可以访问 HashMap
 let data = &mut *g;
 // Cow 实现了很多数据结构的 From trait,
 // 所以我们可以用 "hello".into() 生成 Cow
 let entry = data.entry("hello".into()).or_insert(0);
 *entry += 1;
 // MutexGuard 被 Drop, 锁被释放
 });
 }

 thread::sleep(Duration::from_millis(100));

 println!("metrics: {:?}", metrics.lock().unwrap());
}

```

MutexGuard 不允许 Send, 只允许 Sync

```

impl<T: ?Sized> !Send for MutexGuard<'_, T> {}
unsafe impl<T: ?Sized + Sync> Sync for MutexGuard<'_, T> {}

```

类似 MutexGuard 的智能指针有很多用途。比如要创建一个连接池, 你可以在 Drop trait 中, 回收 checkout 出来的连接, 将其再放回连接池

实现自己的智能指针

```

use std::{fmt, ops::Deref, str};

const MINI_STRING_MAX_LEN: usize = 30;

// MyString 里, String 有 3 个 word, 供 24 字节, 所以它以 8 字节对齐
// 所以 enum 的 tag + padding 最少 8 字节, 整个结构占 32 字节。

```

```

// MiniString 可以最多有 30 字节（再加上 1 字节长度和 1字节 tag），就是 32 字节。
struct MiniString {
 len: u8,
 data: [u8; MINI_STRING_MAX_LEN],
}

impl MiniString {
 // 这里 new 接口不暴露出去，保证传入的 v 的字节长度小于等于 30
 fn new(v: impl AsRef<str>) -> Self {
 let bytes = v.as_ref().as_bytes();
 // 我们在拷贝内容时一定要使用字符串的字节长度
 let len = bytes.len();
 let mut data = [0u8; MINI_STRING_MAX_LEN];
 data[..len].copy_from_slice(bytes);
 Self {
 len: len as u8,
 data,
 }
 }
}

impl Deref for MiniString {
 type Target = str;

 fn deref(&self) -> &Self::Target {
 // 由于生成 MiniString 的接口是隐藏的，它只能来自字符串，所以下面这行是安全的
 str::from_utf8(&self.data[..self.len as usize]).unwrap()
 // 也可以直接用 unsafe 版本
 // unsafe { str::from_utf8_unchecked(&self.data[..self.len as usize]) }
 }
}

impl fmt::Debug for MiniString {
 fn fmt(&self, f: &mut fmt::Formatter<'_>) -> fmt::Result {
 // 这里由于实现了 Deref trait，可以直接得到一个 &str 输出
 write!(f, "{}", self.deref())
 }
}

#[derive(Debug)]
enum MyString {
 Inline(MiniString),
 Standard(String),
}

// 实现 Deref 接口对两种不同的场景统一得到 &str
impl Deref for MyString {
 type Target = str;
}

```

```
fn deref(&self) -> &Self::Target {
 match *self {
 MyString::Inline(ref v) => v.deref(),
 MyString::Standard(ref v) => v.deref(),
 }
}

impl From<&str> for MyString {
 fn from(s: &str) -> Self {
 match s.len() > MINI_STRING_MAX_LEN {
 true => Self::Standard(s.to_owned()),
 _ => Self::Inline(MiniString::new(s)),
 }
 }
}

impl fmt::Display for MyString {
 fn fmt(&self, f: &mut fmt::Formatter<'_>) -> fmt::Result {
 write!(f, "{}", self.deref())
 }
}

fn main() {
 let len1 = std::mem::size_of::<MyString>();
 let len2 = std::mem::size_of::<MiniString>();
 println!("Len: MyString {}, MiniString {}", len1, len2);

 let s1: MyString = "hello world".into();
 let s2: MyString = "这是一个超过了三十个字节的很长很长的字符串".into();

 // debug 输出
 println!("s1: {:?}, s2: {:?}", s1, s2);
 // display 输出
 println!(
 "s1: {}({} bytes, {} chars), s2: {}({} bytes, {} chars)",
 s1,
 s1.len(),
 s1.chars().count(),
 s2,
 s2.len(),
 s2.chars().count()
);
 // MyString 可以使用一切 &str 接口, 感谢 Rust 的自动 Deref
 assert!(s1.ends_with("world"));
 assert!(s2.starts_with("这"));
}
```

## 2.11 迭代器

迭代器是一个与循环相关的概念。Rust 的循环和大部分语言都一致，支持死循环 loop、条件循环 while，以及对迭代器的循环 for。循环可以通过 break 提前终止，或者 continue 来跳到下一轮循环。Rust 支持分支跳转、模式匹配、错误跳转和异步跳转

### 2.11.1 迭代器trait

迭代器和Rust中的集合类型密切相关

1. 是设计模式中的一种行为模式
2. 与集合使用，在不暴露集合底层的情况下遍历集合元素
3. 将集合的遍历行为抽象为单独的迭代对象（将行为抽象为对象）

迭代器分为外部迭代器和内部迭代器，for循环实际上是外部迭代器的一个语法糖。

在执行过程中，IntoIterator 会生成一个迭代器，for 循环不断从迭代器中取值，直到迭代器返回 None 为止。因而，for 循环实际上只是一个语法糖，编译器会将其展开使用 loop 循环对迭代器进行循环访问，直至返回 None。

```
// 迭代器trait
trait Iterator {
 type Item;

 fn next(&mut self) -> Option<Self::Item>;
}

// 外部迭代器语法糖for循环，相当于迭代器的next方法。for 循环可以用于任何实现了 IntoIterator trait 的数据结构。
// Vec实现了迭代器trait
let v = vec![1, 2, 3, 4, 5];
{
 // 使用into_iter方法获得迭代器
 let mut _iterator = v.into_iter();
 loop {
 // match 匹配每一次的迭代结果
 match _iterator.next() {
 Some(i) => {
 println!("{} ", i);
 }
 None => break,
 }
 }
}

// 使用for循环遍历
let v = vec![1, 2, 3, 4, 5];
for i in 0..v.len() {
 println!("{} ", v[i]);
}
```

```

// 自定义的内部迭代器（不是主要的模式）
trait InIterator<T: Copy> {
 // 指定约束是为了把闭包作为参数传递
 fn each<F: Fn(T) -> T>(&mut self, f: F);
}

impl<T: Copy> InIterator<T> for Vec<T> {
 fn each<F: Fn(T) -> T>(&mut self, f: F) {
 let mut i = 0;
 while i < self.len() {
 self[i] = f(self[i]);
 i += 1;
 }

 // 等价于
 // for i in 0..self.len() {
 // self[i] = f(self[i]);
 // }
 }
}

let mut v = vec![1, 2, 3];
v.each(|i| i * 3);
assert_eq!([3, 6, 9], &v[..3])

```

## 2.11.2 标准库导读

为集合类型实现迭代器时只需要实现next方法

迭代器有三种类型：iter () &T, iter\_mut () &mut T, into\_iter () T, 对应所有权三种语义

迭代器适配器模式：允许在迭代的时候以不同的方式迭代：如map变迭代边映射，还有take和filter, chain。把原来的迭代器进行封装

迭代器trait：扩展，消费，两头迭代，FromIterator（和消费者配合）等

## 2.11.3 第三方库

ertools

## 2.12 模块

1. 语法集合
2. 模块是一种软件设计思想，降低耦合，便于维护
3. Rust中模块用于分割代码

在rust中模块可以使用mod关键字定义，也可默认使用单个文件作为模块

同级模块使用crate

父级模块使用super

包外模块之间使用包名

模块的可见性自定义

## 2.12.1 模块与属性

```
#[path = "foo.rs"]
mod c
// 找 c.rs

// 找inline/inner.rs
mod inline {
 #[path = "other.rs"]
 mod inner;
}

// 找路径thread_files/local_data.rs
#[path = "thread_files"]
mod thread {
 #[path = "tls.rs"]
 mod local_data
}
```

## 2.13 包管理器Cargo

```
[package]
name = "from-principle-to-practice"
version = "0.1.0"
edition = "2021"

See more keys and their definitions at https://doc.rust-
lang.org/cargo/reference/manifest.html

[dependencies]
```

1. 在rust中， package/crate都是指包， crate时编译单元
2. package包含多个crate
3. crate是实际的编译单元
4. codegen-uint: 每个crate在编译时默认被LLVM IR切割为16份， 方便并行编译

## 2.13.1 Cargo工作

没有依赖地狱问题

基本包结构



```
└── Cargo.lock
└── Cargo.toml
src/
└── lib.rs
└── main.rs
└── bin/
 ├── named-executable.rs
 ├── another-executable.rs
 └── multi-file-executable/
 └── main.rs
 └── some_module.rs
benches/
└── large-input.rs
└── multi-file-bench/
 └── main.rs
 └── bench_module.rs
examples/
└── simple.rs
└── multi-file-example/
 └── main.rs
 └── ex_module.rs
tests/
└── some-integration-tests.rs
└── multi-file-test/
 └── main.rs
```

## └── test\_module.rs

### 2.13.2 toml 配置文件

语义明显、无歧义的配置文件格语言格式

```
这是一个 TOML 文档

title = "TOML 示例"

[owner]
name = "Tom Preston-Werner"
dob = 1979-05-27T07:32:00-08:00

[database]
enabled = true
ports = [8000, 8001, 8002]
data = [["delta", "phi"], [3.14]]
temp_targets = { cpu = 79.5, case = 72.0 }

[servers]

[servers.alpha]
ip = "10.0.0.1"
role = "前端"

[servers.beta]
ip = "10.0.0.2"
role = "后端"
```

### 2.13.3 Cargo 命令与工具介绍

```
cargo check 静态检查当前crate及其依赖项
cargo build 静态检查和编译
cargo run 检查+构建+执行
cargo clean 清除构建文件
cargo doc 生成文档
```

常用工具

```
cargo fix 修复warning
cargo add
cargo audit 维护漏洞数据库，检查以来漏洞
cargo clippy 静态分析坏代码
cargo fmt 格式化代码
cargo expand 展开宏
```

更多内容请查看Cargo book

## 3 Rust语言核心

基本概念

内存—栈、堆 ✓

数据—程序操作的对象：值、类型、指针、引用

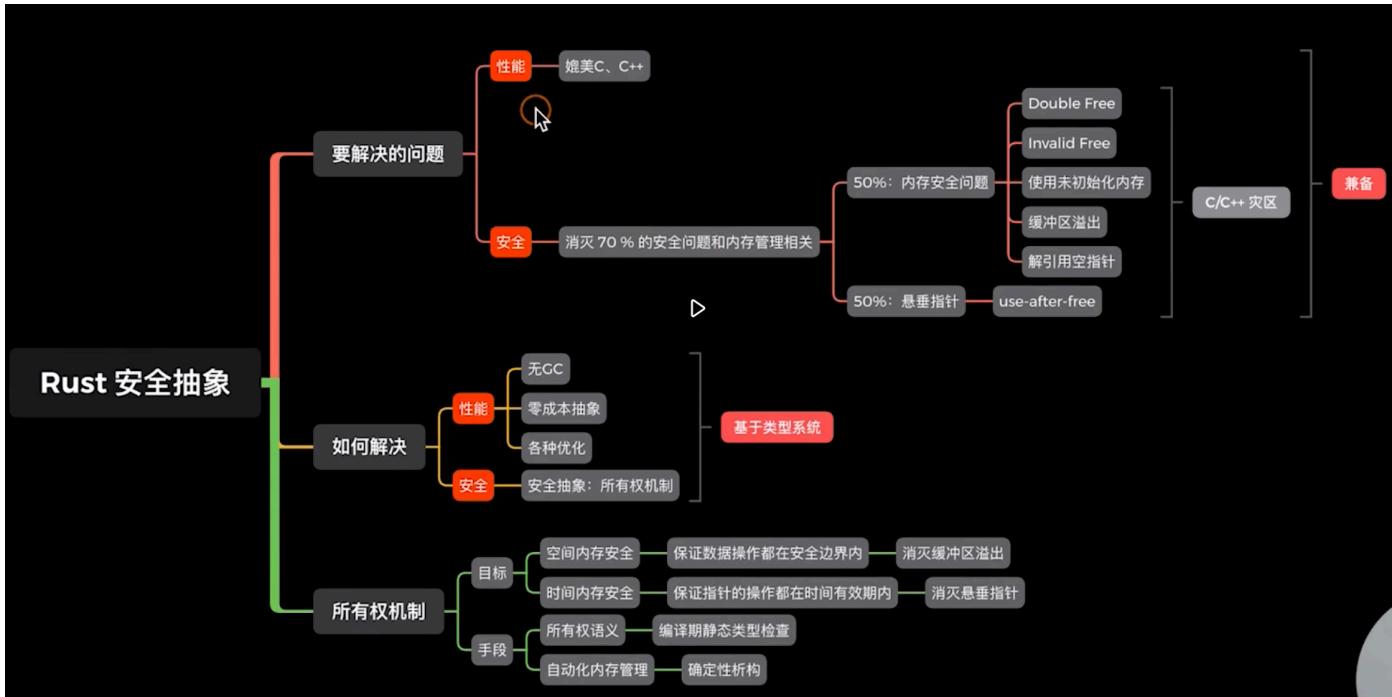
代码—程序运行的主体：函数、方法、闭包、接口、虚表

运行方式—程序的执行效率：并发、并行、同步、异步

编程范式—提升代码的质量：泛型编程

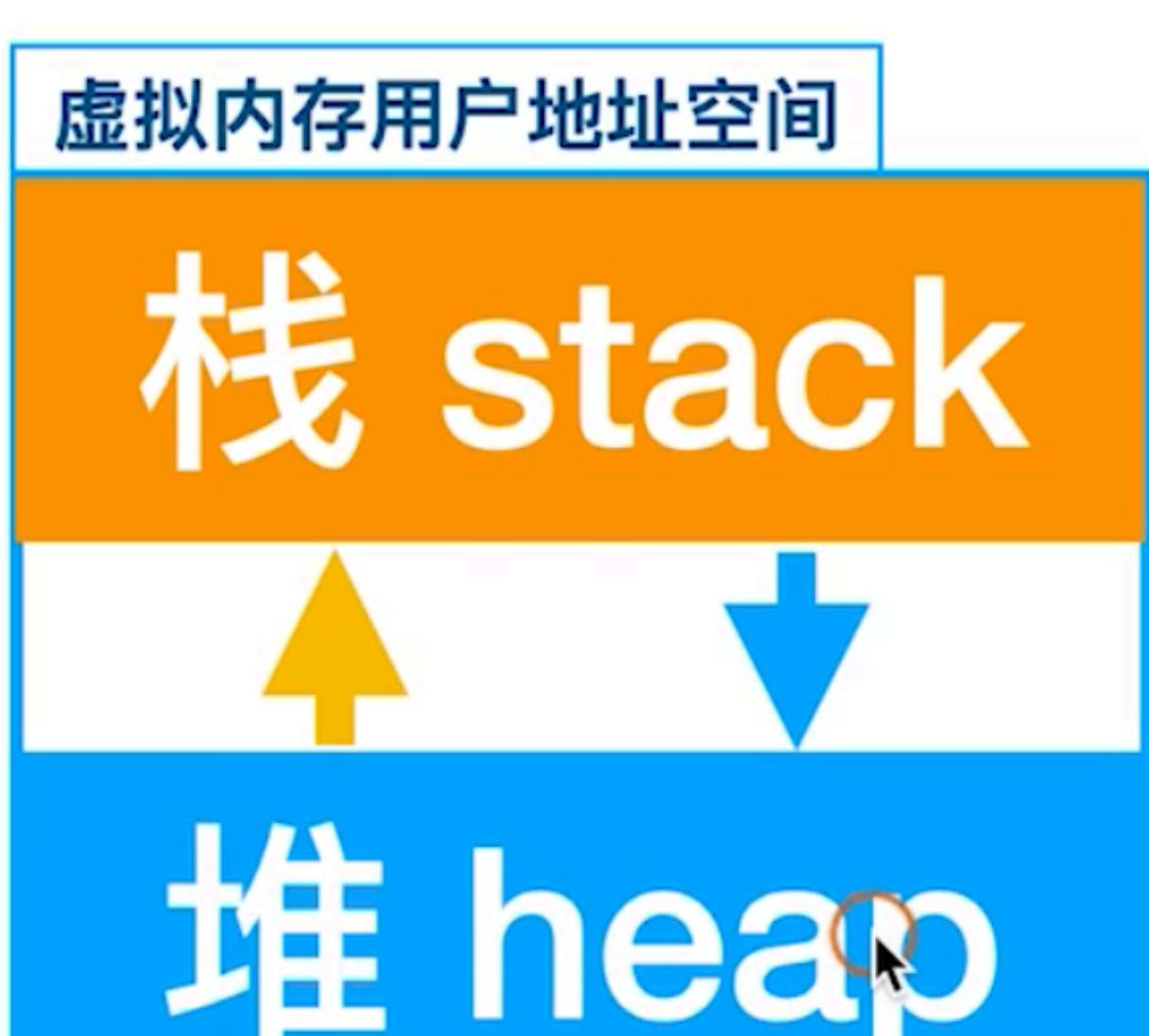
### 3.1 Rust语言架构

#### 1. 安全抽象和范式抽象



2. 类型系统: 保证程序安全
3. 资源管理 (内存管理)

### 3.1.1 虚拟地址空间



The diagram illustrates the memory layout of a process. It consists of three main horizontal sections. The top section is grey and contains the text 'BSS' above '数据段' (Data Segment). The middle section is white and contains the text '代码段' (Code Segment). The bottom section is blue. The entire diagram is enclosed in a light grey border.

BSS  
数据段

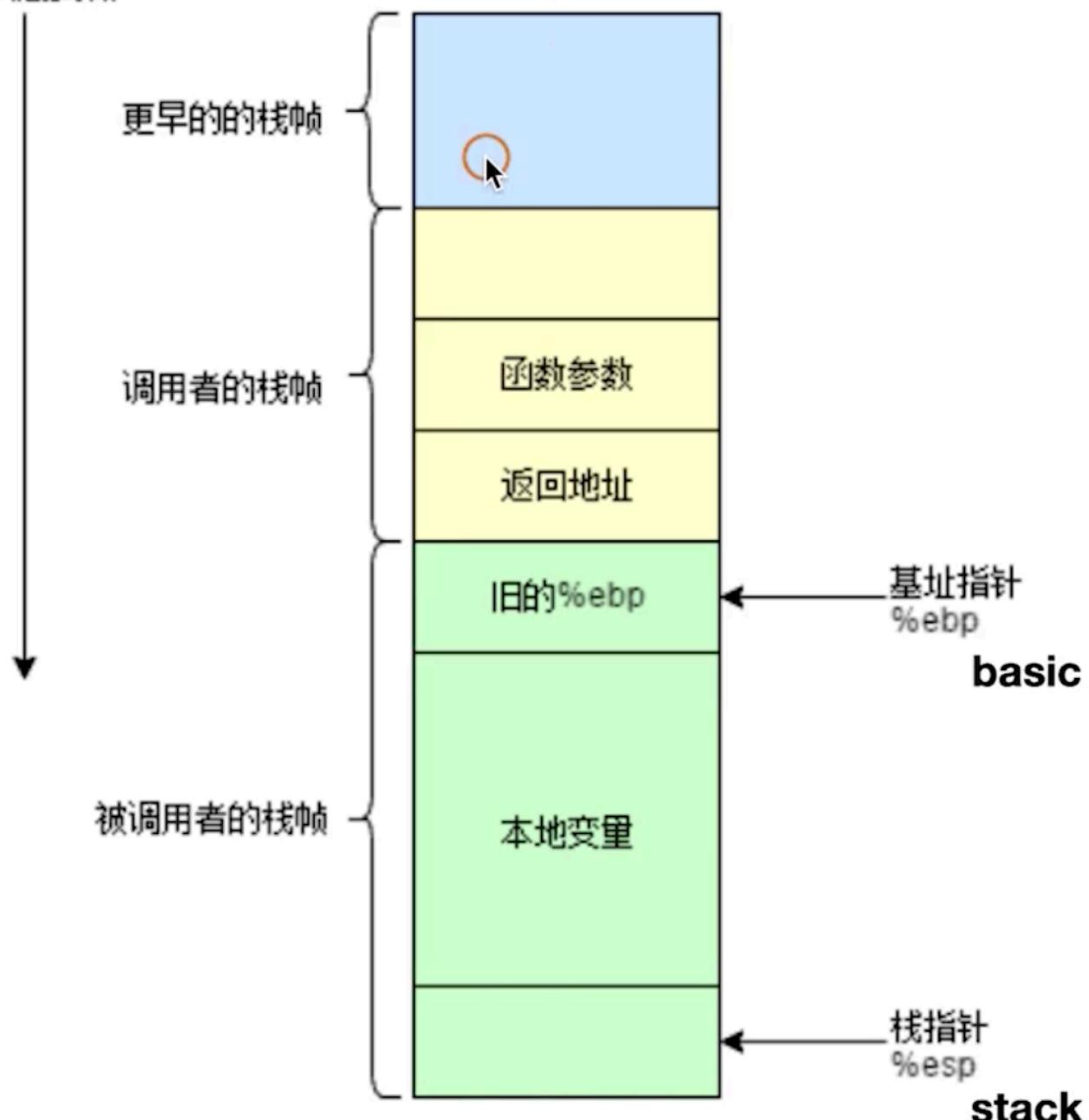
代码段

### 3.1.2 函数调用栈

避免栈溢出



## 地址从高到低



## 函数调用栈实例

```
let answer = "42";
let no_answer = answer;
println!("{:?}", answer); //可用

let answer = String::from("42");
let no_answer = answer;
```

## 中级中间语言

```
// MIR
// 函数调用栈
// 运行结束时,最后一个会先被清除
```

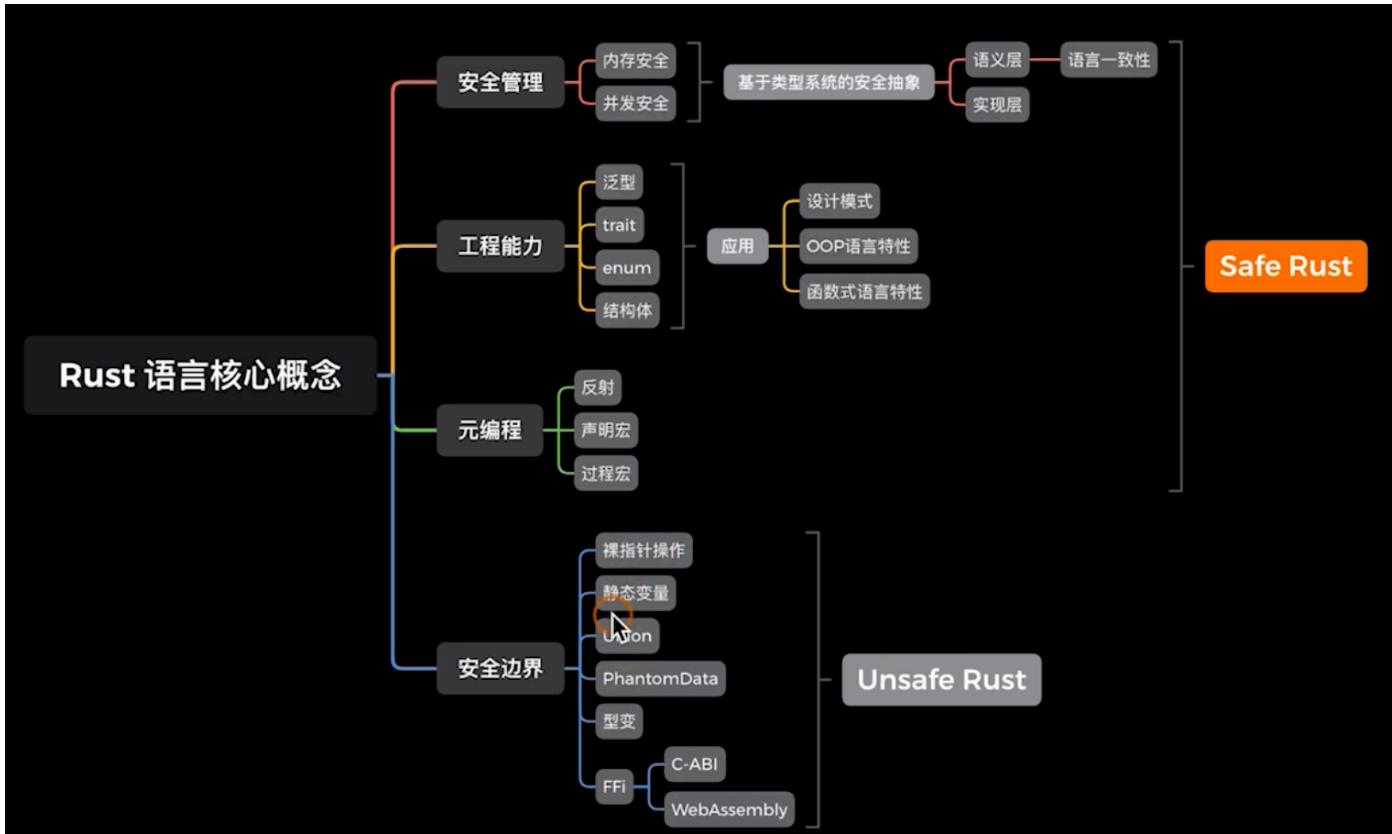
```
// 先进后出
/*
let _1: &str;
scope 1 {
 debug answer => _1;
 let _2: &str;
 scope 2 {
 debug no_answer => _2;
 let _3: std::string::String;
 scope 3 {
 debug answer => _3;
 let _4: std::string::String;
 scope 4 {
 debug no_answer => _4;
 }
 }
 }
}
```

### 3.1.3 Rust与其它语言内存管理区别

1. C: 纯手工管理 (缺乏安全抽象模型)
2. C++: 手工管理 + 确定性析构 (缺乏安全抽象模型)
3. GC语言: 垃圾回收 (性能差)
4. Rust语言: 考虑性能, 借鉴Cpp的RAII资源管理方式, 考虑安全: 增加所有权语义

## 3.2 Rust核心概念

### 3.2.1 核心概念



### 3.2.2 要掌握的内容

1. 掌握所有权语义
2. 领略Rust的工程能力
3. 掌握元编程能力
4. 正确认识Unsafe Rust

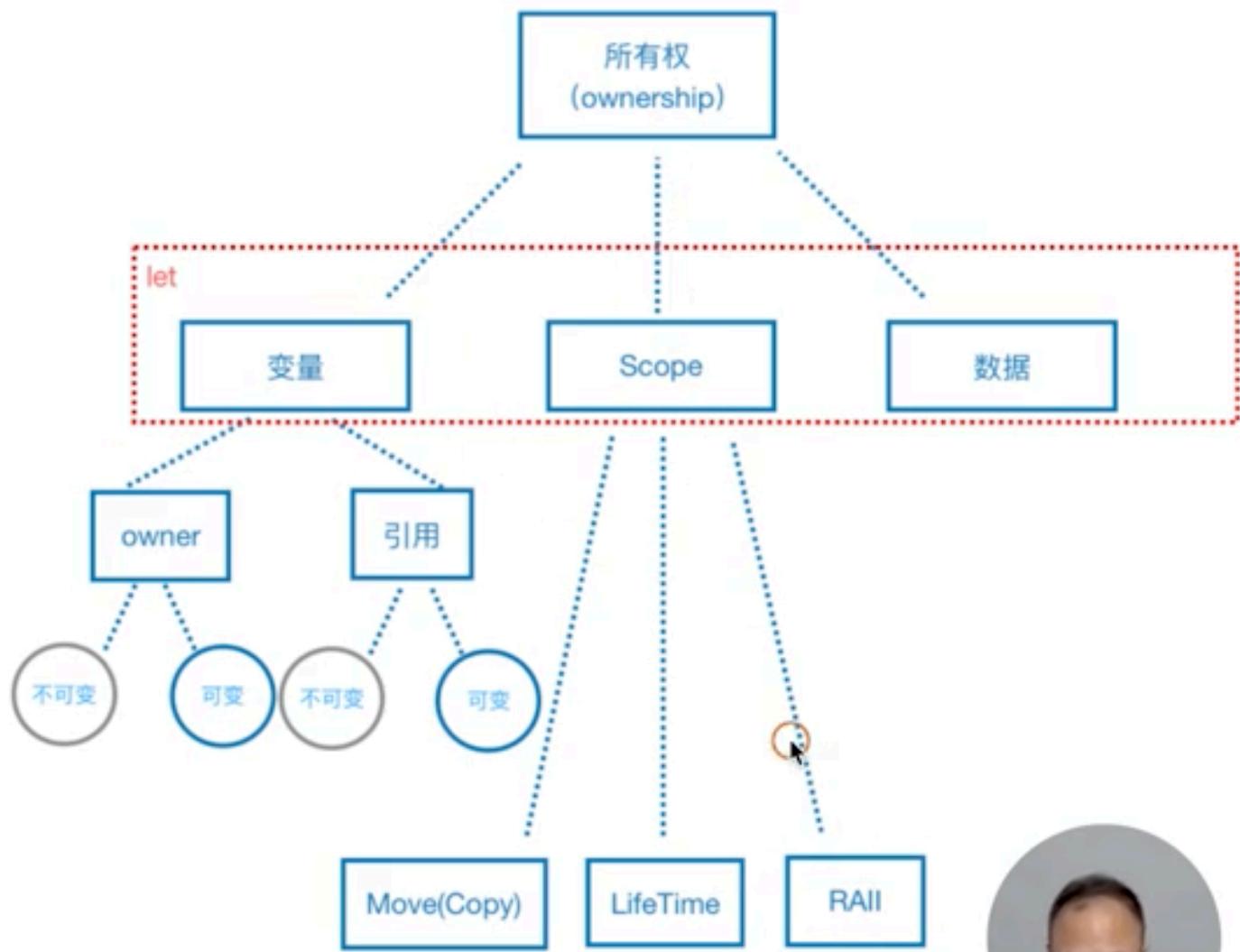
## 3.3 内存安全：所有权

### 3.3.1 语义模型

有两种: Copy和Clone

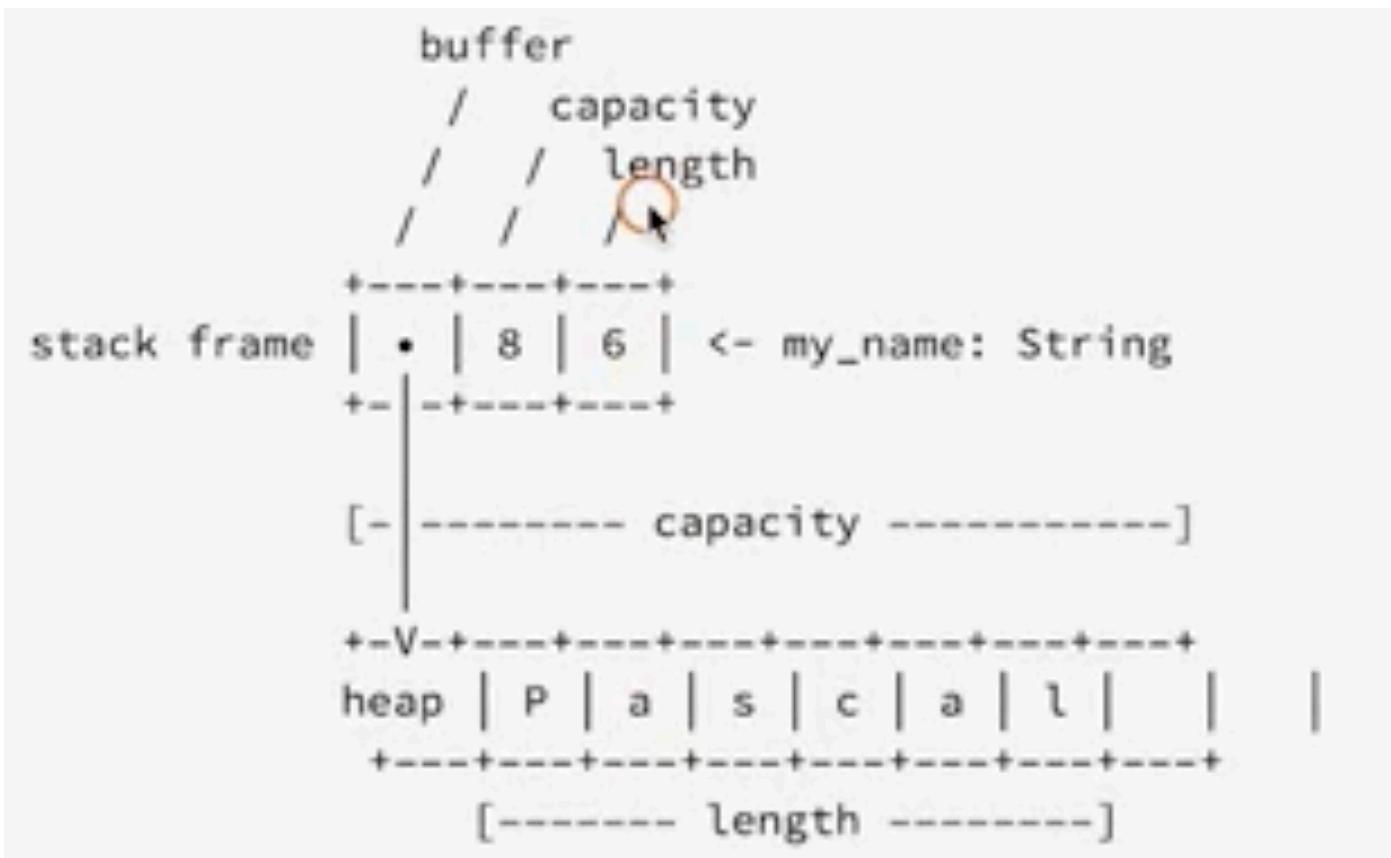
当声明一个变量时，这个变量会拥有所有权，绑定一段生命周期以及绑定一个数据，这个变量是所有权的拥有者，它可以被使用（所有权转移）或者借用（使用权转移）。当它进入到新的scope时是move或者拷贝，引用的话受原变量声明周期的约束，RALL内存管理机制通过Scope（有所有权的变量才有权利管理释放内存）管理内存

```
let answer = 42;
```

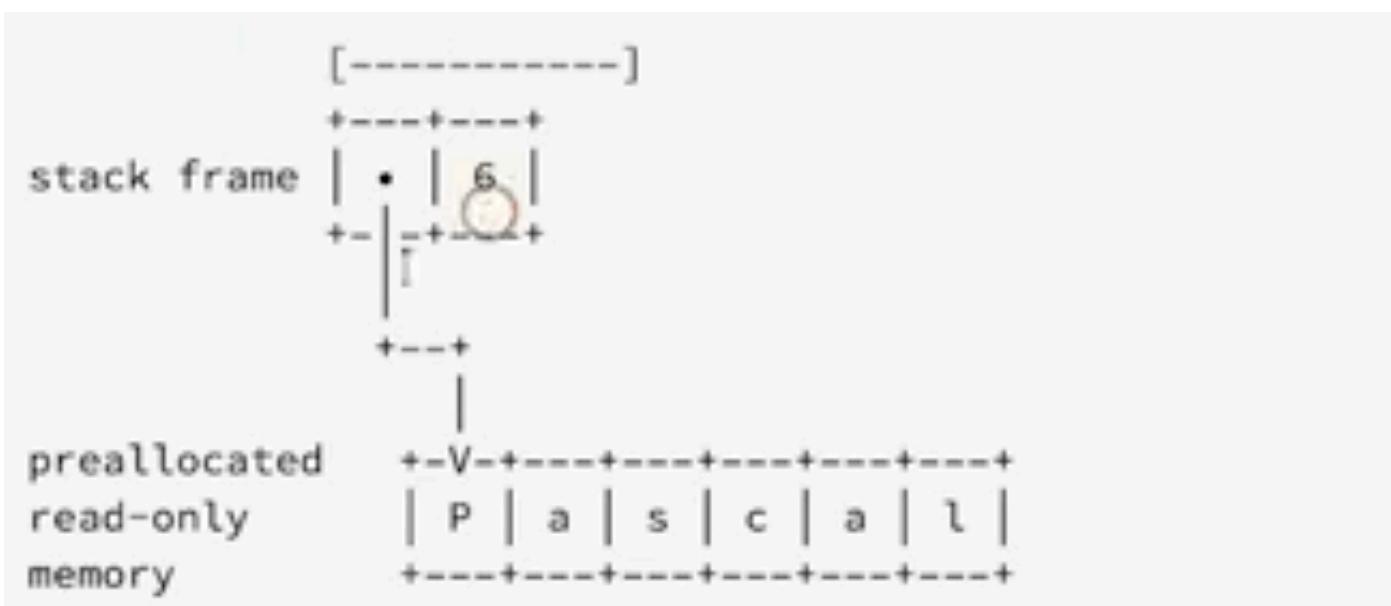


### 3.3.1.1 Copy和Copy trait

String的结构



&str的结构



基础数据类型：基本都实现了Copy trait

自定义类型：结构体不会实现Copy，需要手动通过派生宏实现，并且同时需要Clone trait；当结构体内部的成员类型没有实现copy时，结构体也不能实现Copy，枚举同理

注意：&mut T 没有实现Copy类型，&T实现了Copy

### 3.3.1.2 Move与析构

move的本质是把变量进行了未初始化标记而不是立刻丢弃

不同的情况下，变量析构的顺序可能不同，本质上是和内存安全相关的

## 3.3.2 借用检查

### 3.2.1.1 词法作用域和非词法作用域

学习词法作用域和非词法作用域借用检查

非词法作用域检查颗粒度更细，在mir层级

```
// 1 词法作用域
// 一个函数的块表达式对应一个栈帧 stack frame
// 栈帧的特点是函数调用完会自动清空
// 词法作用域对应栈帧
// 基本上词法作用域等于生命周期
let mut v = vec![];
v.push(1);

{
 // println!("{:?}", v[0]);
 v.push(2);
}
// mir 中每一个scope都代表一个词法作用域
/*
scope 1 {
 debug v => _1; // in scope 1 at src/main.rs:3:5: 3:10
}
*/
// 2 非词法作用域 NLL: 案例 1

// Rust语言编译过程
// text -> tokens -> ast -> hir -> mir -> llvm ir -> llvm
// 在可变借用的作用域内不允许在开辟的子生命周期中执行可变借用
let mut v: Vec<i32> = vec![];
let vv = &v;

{
 // println!("{:?}", v[0]);
 // v.push(2); // 不允许可变借用
}

vv;
// 2 非词法作用域 NLL: 案例 2
// 替换问好
```

```

let s = "ab?c?d";

// 把字符串转成字符切片
let mut chars = s.chars().collect::<Vec<char>>();

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

for i in 0..s.len() {
 // 这里不可以用可变借用
 let mut words = ('a'..'z').into_iter();
 println!("{}?", words);

 if chars[i] == '?' {
 // 获取左边和右边的字符
 let left = if i == 0 { None } else { Some(chars[i - 1]) };
 let right = if i == s.len() - 1 {
 None
 } else {
 Some(chars[i + 1])
 };

 // 在26个字母中寻找不等于左边也不等于右边的字母进行替换
 chars[i] = words
 .find(|&w| Some(w) != left && Some(w) != right)
 .unwrap();
 }
}

// 将字符收集转换为字符串
let s = chars.into_iter().collect::<String>();
println!("{}?", s)

```

### 3.3.1.2 生命周期参数

生命周期参数，描述的是参数和参数之间、参数和返回值之间的关系，并不改变原有的生命周期。生命周期标注的目的是，在参数和返回值之间建立联系或者约束。调用函数时，传入的参数的生命周期需要大于等于（outlive）标注的生命周期

当每个函数都添加好生命周期标注后，编译器，就可以从函数调用的上下文中分析出，在传参时，引用的生命周期，是否和函数签名中要求的生命周期匹配。如果不匹配，就违背了“引用的生命周期不能超出值的生命周期”，编译器就会报错

规则：

1. 所有引用类型的参数都有独立的生命周期 'a 、 'b 等。
2. 如果只有一个引用型输入，它的生命周期会赋给所有输出。
3. 如果有多个引用类型的参数，其中一个是 self，那么它的生命周期会赋给所有输出。

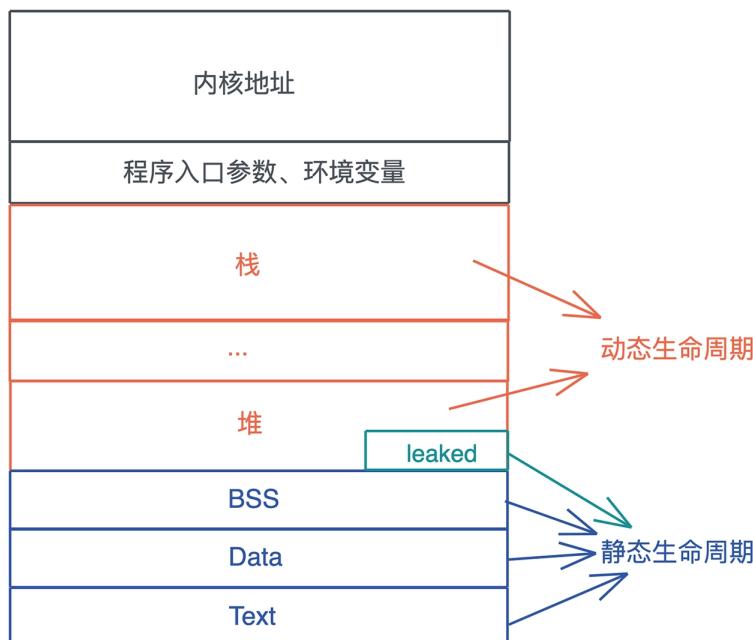
规则 3 适用于 trait 或者自定义数据类型

分配在堆和栈上的内存有其各自的作用域，它们的生命周期是动态的

注意：当一个函数参数是引用类型，并且跟内部的局部变量做了相关运算后，再返回时也是同样的引用类型，这时候生命周期参数就比较直观了，为了避免返回局部变量而出现悬垂指针

全局变量、静态变量、字符串字面量、代码等内容，在编译时，会被编译到可执行文件中的 BSS/Data/RoData/Text 段，然后在加载时，装入内存。因而，它们的生命周期和进程的生命周期一致，所以是静态的。

所以，函数指针的生命周期也是静态的，因为函数在 Text 段中，只要进程活着，其内存一直存在。



1. 目的：为了避免出现悬垂指针
2. 晚限定与早限定

生命周期参数一般出现在函数参数的传递过程中以及自定义类型声明时

有两种方式：晚限定和早限定，早限定是一种更普遍的用法，尤其是实现trait或者关联函数时，不用在每个函数签名处声明生命周期参数

总结：

late bound：在具体调用时才自动生成具体的生命周期参数实例，不可以手动指定，编译器会检查本地变量

early bound：可以指定生命周期参数，会让编译器只检查参数类型以及生命周期参数，不检查本地变量

### 3. trait 对象中的生命周期参数

```
trait Foo<'a> {}

struct FooImpl<'a> {
 s: &'a [u32],
}

impl<'a> Foo<'a> for FooImpl<'a> {}
```

```

// trait 对象必须使用 Box 包裹
// 任何实现了某个 trait 的类型，它的实例都是 trait 对象
// trait 对象默认为静态生命周期，当作为返回值时，需要手动“缩短”（指定生命周期参数，如 'a)

// fn foo<'a, 'b: 'a>(s: &'a [u32]) -> Box<dyn Foo<'a> + 'a> { // 第一种写法
fn foo<'a>(s: &'a [u32]) -> Box<dyn Foo<'a> + 'a> {
 // 第二种写法
 Box::new(FooImpl { s: s })
}

```

#### 4. 高阶生命周期参数

```

use std::fmt::Debug;
trait DosSomething<T> {
 fn do_something(&self, value: T);
}

impl<'a, T: Debug> DosSomething<T> for &'a usize {
 fn do_something(&self, value: T) {
 println!("{}: {:?}", value);
 }
}

// 高阶生命周期，高阶限定，for 语法，是一种 late bound
// fn foo<'a>(b: Box<dyn DosSomething<&'a usize>>) { 改动前
fn foo<'a>(b: Box<dyn for<'f> DosSomething<&'f usize>>) {
 // 不在当前作用域判断
 let s: usize = 10;
 b.do_something(&s) // 在 do something 函数作用域判断
}

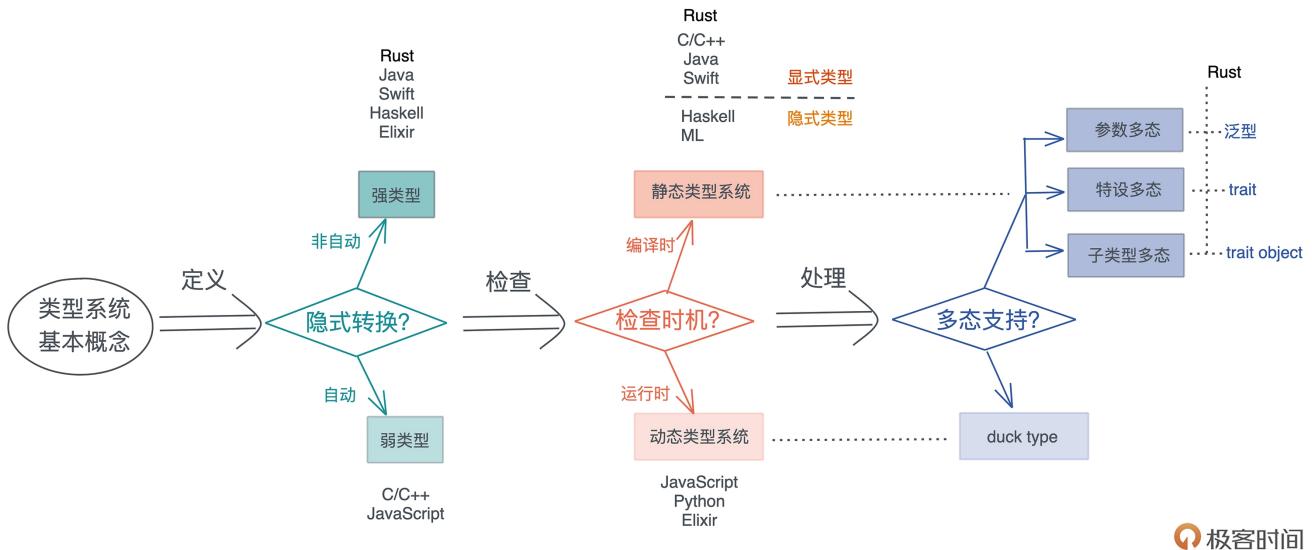
let x = Box::new(&2usize);
foo(x)

```

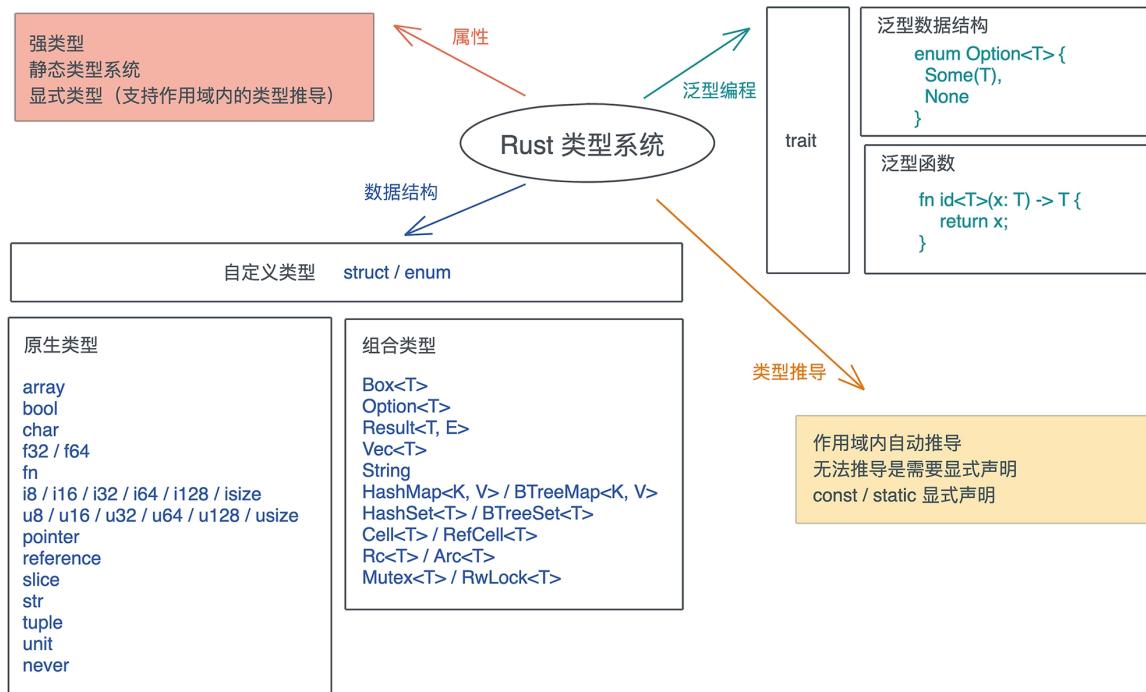
注意：并不是所有的生命周期都是在当前作用域判断的

5. 闭包生命周期参数
6. trait 对象中的生命周期参数

## 3.3.2 类型系统



极客时间



极客时间

里氏替换原则简单说就是子类型对象可以在程序中代替父类型对象。它是运行时多态的基础。所以如果要支持运行时多态，以及动态分派、后期绑定、反射等功能，编程语言需要支持动态类型系统

按类型定义、检查以及检查时能否被推导出来，Rust 是强类型 + 静态类型 + 显式类型

类型系统完全是一种工具，编译器在编译时对数据做静态检查，或者语言在运行时对数据做动态检查的时候，来保证某个操作处理的数据是开发者期望的数据类型。类型系统其实就是，对类型进行定义、检查和处理的系统

Rhs类型，是对值的区分，它包含了值在内存中的长度、对齐以及值可以进行的操作等信息

在类型系统中，多态是一个非常重要的思想，它是指在使用相同的接口时，不同类型的对象，会采用不同的实现

对于动态类型系统，多态通过鸭子类型（duck typing）实现；而对于静态类型系统，多态可以通过参数多态（parametric polymorphism）、特设多态（adhoc polymorphism）和子类型多态（subtype polymorphism）实现。

参数多态是指，代码操作的类型是一个满足某些约束的参数，而非具体的类型。特设多态是指同一种行为有多个不同实现的多态。比如加法，可以  $1+1$ ，也可以是 “abc” + “cde”、matrix1 + matrix2、甚至 matrix1 + vector1。在面向对象编程语言中，特设多态一般指函数的重载。

子类型多态是指，在运行时，子类型可以被当成父类型使用。在 Rust 中，参数多态通过泛型来支持、特设多态通过 trait 来支持、子类型多态可以用 trait object 来支持。

Rust 编译器遵循类型理论：仿射类型：它是一种子结构类型系统。意义：资源最多只能被使用一次。

Rust 类型系统有两种语义：移动语义（默认）复制语义（该类型必须实现 Copy trait：数据能够被安全的复制）

为什么实现 Copy 必须先实现 Clone，它是编译器的行为，开发者再实现无用。

```
pub trait Copy: Clone { }
```

哪些是移动语义？在运行时动态增长的类型，也就是说需要动态分配内存。

Copy 本质上是按位复制，并且不可以被重载，clone 隐式调用，可以显式实现和调用。

一个特殊的例子，原生指针是 Copy 的。

```
// raw pointer is Copy
is_copy::<*const String>();
is_copy::<*mut String>();
```

可变引用和非固定大小的数据结构没有实现 copy。

一些组合类型

| 类型                                      | 介绍                                                 | 示例                                                                                        |
|-----------------------------------------|----------------------------------------------------|-------------------------------------------------------------------------------------------|
| <code>Box&lt;T&gt;</code>               | 分配在堆上的类型 T                                         | <code>let v: Box&lt;i32&gt; = Box::new(1);</code>                                         |
| <code>Option&lt;T&gt;</code>            | T 要么存在，要么为 None                                    | <code>Some(42)</code><br><code>None</code>                                                |
| <code>Result&lt;T, E&gt;</code>         | 要么成功 <code>Ok(T)</code> , 要么失败 <code>Err(E)</code> | <code>Ok(42)</code><br><code>Err(ConnectionError::TooMany)</code>                         |
| <code>Vec&lt;T&gt;</code>               | 可变列表，分配在堆上                                         | <code>let mut arr = vec![1, 2, 3];</code>                                                 |
| <code>String</code>                     | 字符串                                                | <code>let s = String::from("hello");</code>                                               |
| <code>HashMap&lt;K, V&gt;</code>        | 哈希表                                                | <code>let map: HashMap&lt;&amp;str, &amp;str&gt; =</code><br><code>HashMap::new();</code> |
| <code>HashSet&lt;T&gt;</code>           | 集合                                                 | <code>let set: HashSet&lt;u32&gt; = HashSet::new();</code>                                |
| <code>RefCell&lt;T&gt;</code>           | 为 T 提供内部可变性的智能指针                                   | <code>let v = RefCell::new(42);</code><br><code>let mut borrowed = v.borrow_mut();</code> |
| <code>Rc&lt;T&gt; / Arc&lt;T&gt;</code> | 为 T 提供引用计数的智能指针                                    | <code>let v = Rc::new(42);</code><br><code>let v1 = Arc::new(42);</code>                  |



## 一些原生类型

| 类型                        | 介绍                                                         | 示例                                                                                                 |
|---------------------------|------------------------------------------------------------|----------------------------------------------------------------------------------------------------|
| array                     | 数组，固定大小的同构序列，[T; N]                                        | [u32; 16]                                                                                          |
| bool                      | 布尔值                                                        | true、false                                                                                         |
| char                      | utf-8 字符                                                   | a'、'♥'                                                                                             |
| f32/f64                   | 浮点数                                                        | 0f32、3.1415926                                                                                     |
| fn                        | 函数指针                                                       | fn(&str) -> usize                                                                                  |
| i8/i16/i32/i64/i128/isize | 有符号整数                                                      | 0i32、1024i128                                                                                      |
| u8/u16/u32/u64/u128/usize | 无符号整数                                                      | 0u8、1024                                                                                           |
| pointer                   | 裸指针，*const T、*mut T。<br>裸指针在解引用时是不安全的。                     | let x = 42;<br>let mut y = 24;<br>let raw = &x as *const i32;<br>let raw_mut = &mut y as *mut i32; |
| reference                 | 引用，&T、&mut T                                               | let x = 42;<br>let mut y = 24;<br>let ref = &x;<br>let ref_mut = &mut y;                           |
| slice                     | 切片，动态大小的连续序列，用[T]表述。<br>一般使用其引用 &[T]、&mut [T] 或者 Box<[T]>。 | let boxed: Box<i32> = Box::new([1,2,3]);<br>let slice = &boxed;                                    |
| str                       | 字符串切片，一般使用其引用 &str、&mut str。                               | let s: &str = "hello world";                                                                       |
| tuple                     | 元组，固定大小的异构序列，表述为 (T, U, ...)                               | ("Hello", 1, false)                                                                                |
| unit                      | 也就是 () 类型，表示没有值                                            | let a = ();<br>let result = Ok(());<br>fn hello() {} 等价于 fn hello() -> () {}                       |
| never                     | ! 类型，表示类型无法产生任何值。<br>目前还是实验特性，详情见 Rust 文档                  | 永远不会出错：Result<T, !><br>一个循环要么出错退出，要么永不返回：<br>Result<!, ConnectionError>                            |

 极客时间

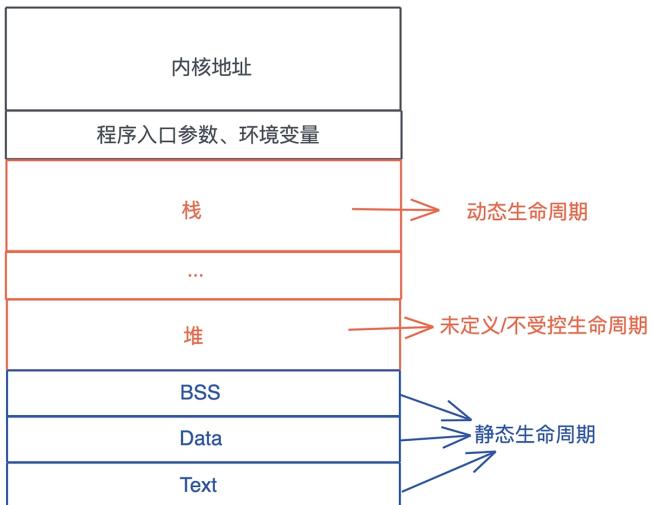
### 3.3.3 内存管理

栈内存“分配”和“释放”都很高效，在编译期就确定好了，因而它无法安全承载动态大小或者生命周期超出帧存活范围外的值。所以，我们需要运行时可以自由操控的内存，也就是堆内存，来弥补栈的缺点

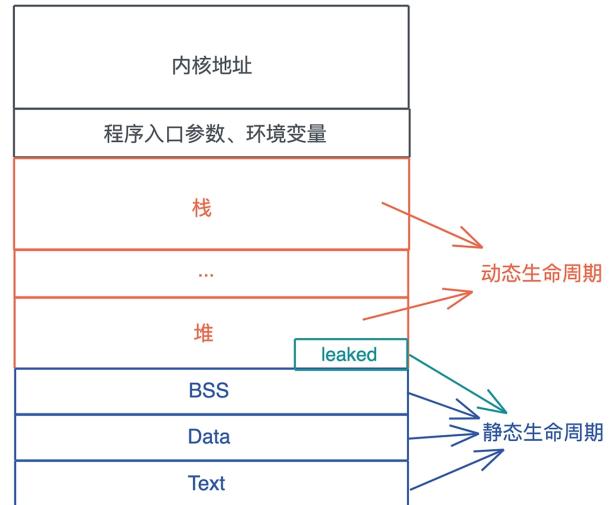
堆内存足够灵活，然而堆上数据的生命周期也比较难管理

但是，大部分堆内存的需求在于动态大小，小部分需求是更长的生命周期。所以Rust默认将堆内存的生命周期和使用它的栈内存的生命周期绑在一起，并留了个小口子 leaked 机制，让堆内存存在需要的时候，可以有超出帧存活期的生命周期

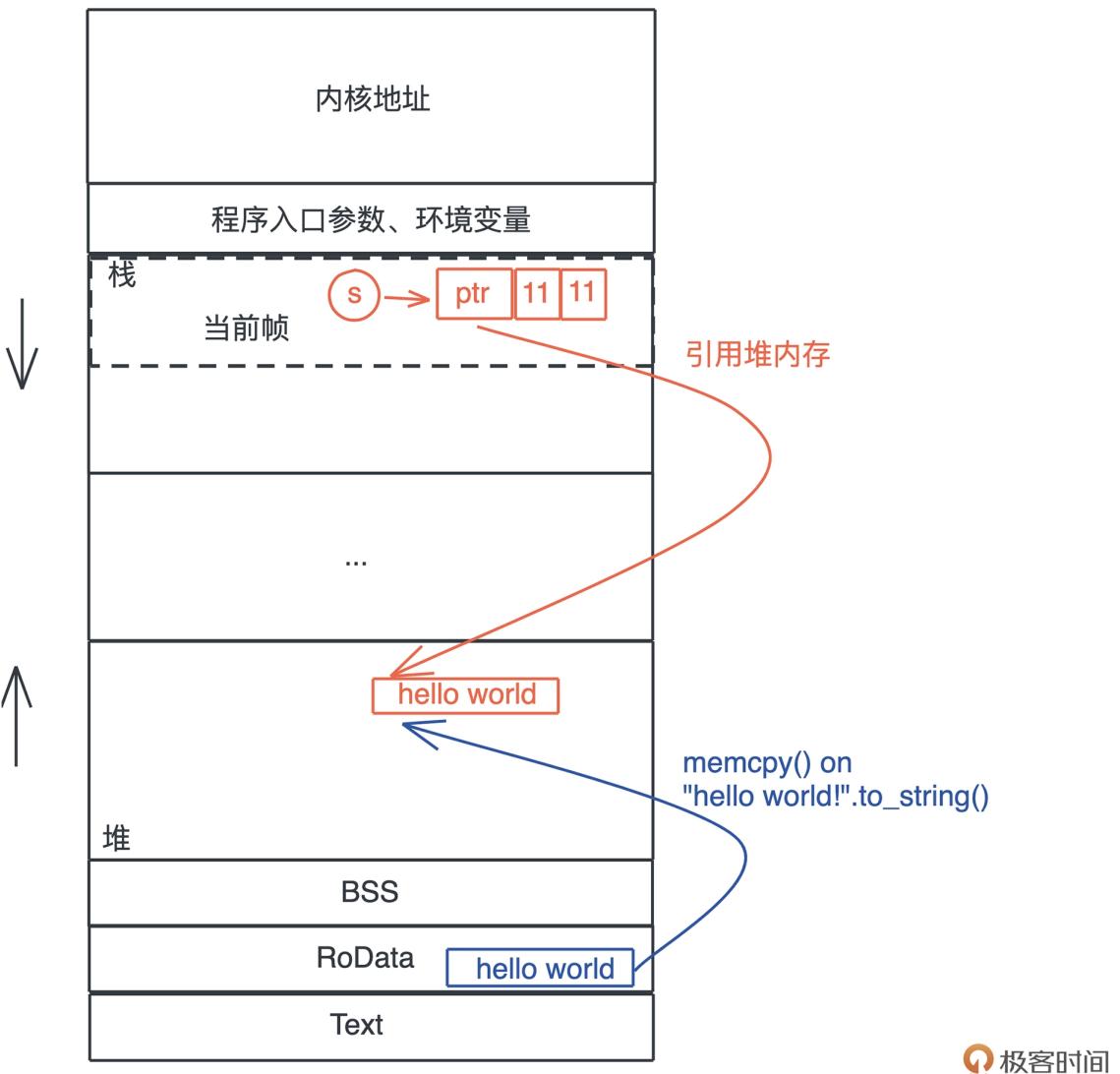
## 大部分编程语言



## Rust



以“hello world”串常量（string literal）为例，在编译时被存入可执行文件的 .RODATA 段（GCC）或者 .RDATA 段（VC++），然后在程序加载时，获得一个固定的内存地址



1. 数据默认存储到栈上，堆上内存分配使用libc提供的malloc函数，其内部会请求操作系统的调用来分配内存，系统调用的代价是昂贵，需要尽可能避免频繁的使用malloc()
2. 利用栈来自动管理堆内存（结合函数调用栈来理解，当栈针被清除时，自动调用析构函数Drop，堆上的数据也被清空）shulu
3. 除了动态大小的内存需要分配到堆上外，动态生命周期的内存要分配在堆上（在不同调用栈之间共享数据）。带来的问题：堆内存泄漏和越界、悬垂指针，分别是1.2大内存安全问题

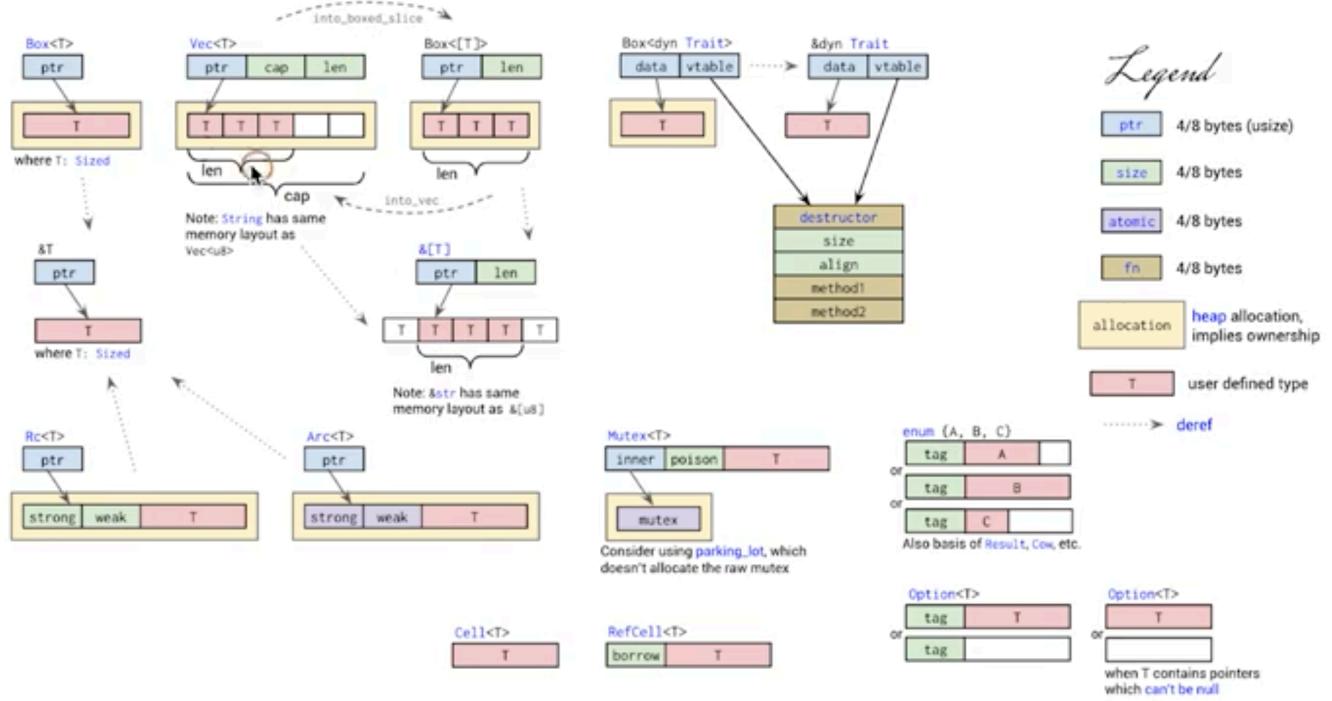
Box也叫做装箱类型，栈上会保留指针

Vec:确定性析构

Box: trait 对象在栈上，保留了数据指针和虚表指针

Rc和Arc引用计数的容器：可以共享所有权，强指针有所有权。锁和容器也类似（不是说强弱指针）

枚举：相当于每个枚举值前面都有tag



### 3.3.4 引用（借用）

一个作用域内，只允许有一个活跃的可变引用

#### 引用规则

可变引用和不可变引用不能共存统同一作用域

借用本质上指的是所有权的借用，共享所有权。可以把它看作是一个受控指针（被借用者可以看作是内存位置），但是它是安全的，经过Rust编译器安全检查的。安全检查包括一些行为，比如可变与不可借用/使用等。Safe Rust中，引用永远是指向有效的数据

Rust中的借用和引用是一个意思，并且是一等公民，和其他类型地位相等

关于裸指针（没有安全的外衣）

Rust中的引用传递是传值，不像java那样，引用是数据对象的别名

对借用的可能指向已经释放的数据这样的内存安全问题，对引用增加了生命周期约束：借用不能超过值的生存期。并且，可变借用是不Copy的。逻辑完美

### 3.3.5 共享

1. Rust中的Clone trait在语义上表示：所有权共享
2. 包含两种：一种是深拷贝，另一种是引用计数。但是二者共用一个clone trait

引用计数容器Rc和Arc以及同步所和互斥锁（Mutex和RwLock）

Arc 内部的引用计数使用了 Atomic Usize，而非普通的 usize。从名称上也可以感觉出来，Atomic Usize 是 usize 的原子类型，它使用了 CPU 的特殊指令，来保证多线程下的安全。

如果我们要在多线程中，使用内部可变性，Rust 提供了 Mutex 和 RwLock。

```
let a = Rc::new(1);
let b = a.clone();
let c = a.clone();

// 如下三个变量地址是一样的 (clone只会增加引用计数)

println!("{:p}", a);
println!("{:p}", b);
println!("{:p}", c);

let a = Arc::new(1);
let b = a.clone();
let c = a.clone();

// 如下三个变量地址是一样的

println!("{:p}", a);
println!("{:p}", b);
println!("{:p}", c);

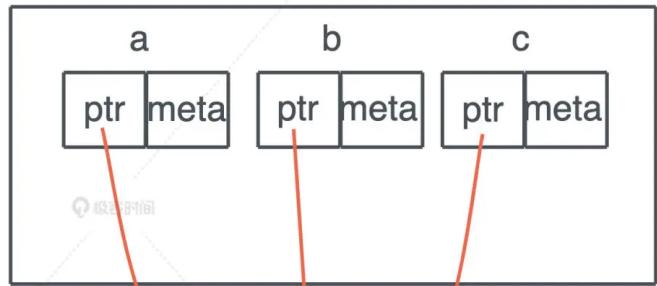
// 如下三个变量地址是不一样的

let l = String::from("hello Rust");
let m = l.clone();
let n = l.clone();

println!("{:p}", l.as_ptr());
println!("{:p}", m.as_ptr());
println!("{:p}", n.as_ptr());
```

示意图：但是为什么出现了三个所有者指向同一块数据的情况呢？从编译器的角度看，每个变量都有一个Rc/Arc，所以不违反所有权规则。其实通过地址我们发现并不是三个变量，而是一个变量被标记了三次

栈

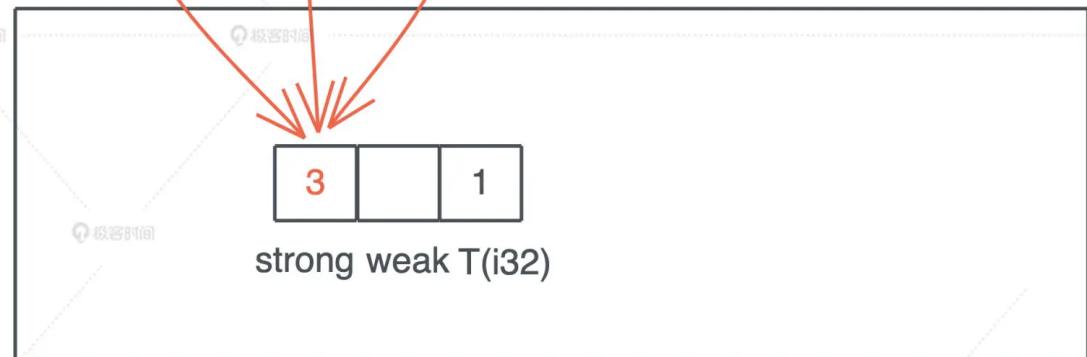


{

```
let a = Rc::new(1);
let b = a.clone();
let c = a.clone();
```

}

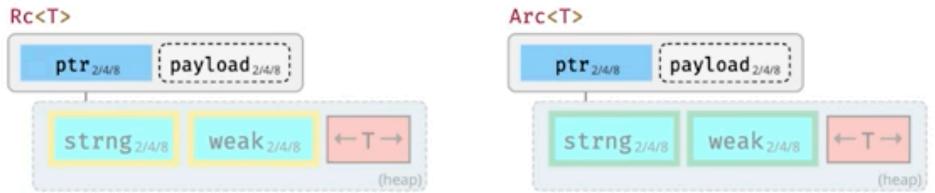
堆



极客时间

极客时间

## 引用计数容器



Needs to be held in Arc to be shared between threads, always Send and Sync. Consider using parking\_lot instead (faster, no heap usage).

## 多线程需要加锁



Box::leak () 机制：提供了创建不受栈内存控制的堆内存。它相当于撕开了一个口子，允许内存泄漏，Rc/Arc指向的堆内存可以绕过编译器检查，然后再使用引用计数在合适的时机结束内存的生命周期（想定多长就多长）。其他也能绕过编译器检查的机制：Box::into\_raw() / ManualDrop。其它情况下，堆内存的生命周期会和其栈内存的生命周期绑定在一起。来确保引用的生命周期不超出值的生命周期

如果值创建在局部，生命周期就是动态的，生命周期约定用'a,'b小写字母来表示

总结：静态检查，靠编译器保证代码符合所有权规则；动态检查，通过 Box::leak 让堆内存拥有不受限的生命周期，然后在运行过程中，通过对引用计数的检查，保证这样的堆内存最终会得到释放。

## 2.3.6 特殊情况

1. 有向无环图 (DAG) : 某个节点可能有两个或者两个以上的节点指向它

```
use std::rc::Rc, sync::Arc;

type NODE = Rc<Node>;

#[derive(Debug)]
struct Node {
 id: usize,
 // 用Rc把节点包裹住，想创建多少个指向它的引用都行
 downstream: Option<NODE>,
}

impl Node {
 pub fn new(id: usize) -> Node {
 Node {
 id,
 downstream: None,
 }
 }

 pub fn update_downstream(&mut self, downstream: NODE) {
 self.downstream = Some(downstream);
 }

 pub fn get_downstream(&self) -> Option<NODE> {
 self.downstream.as_ref().map(|v| v.clone())
 }
}

fn main() {
 let mut node1 = Node::new(1);
 let mut node2 = Node::new(2);
 let mut node3 = Node::new(3);

 let node4 = Node::new(4);

 node3.update_downstream(Rc::new(node4));
 node1.update_downstream(Rc::new(node3));
 node2.update_downstream(node1.get_downstream().unwrap());

 println!("node1: {:?}", node1);
 println!("node2: {:?}", node2);
}
```

使用RefCell获得内部可变性

## 2. 多线程共享变量

此时Rust所有权静态检查无法处理，Rust提供了运行时动态检查来满足这些特殊场景。也就是前一节所提到的Rc和Arc这两个容器（智能指针）

注意：堆是唯一可以让动态创建的数据被到处使用的内存

这也是Rust处理很多问题的思路：编译时，处理大部分使用场景，保证安全性和效率；运行时，处理无法在编译时处理的场景，会牺牲一部分效率，提高灵活性。后续讲到静态分发和动态分发也会有体现，这个思路很值得我们借鉴

## 3.4 线程安全：线程和并发

Mutex是互斥量，获得互斥量的线程对数据独占访问，RwLock是读写锁，获得写锁的线程对数据独占访问，但当没有写锁的时候，允许有多个读锁。读写锁的规则和Rust的借用规则非常类似

很多拥有高并发处理能力的编程语言，会在用户程序中嵌入一个M:N的调度器，把M个并发任务，合理地分配在N个CPU core上并行运行，让程序的吞吐量达到最大。

### 3.4.1 本地线程

本地线程也叫内核线程，由操作系统来调度。

并发：同时应对很多事情的能力

并行：同时执行很多事情的能力

Rust使用了强大的类型系统以及两个专用的trait来在编译期时就发现并发安全问题

```
// Rust 中的线程

// 时间间隔
let duration = std::time::Duration::from_millis(30000);

println!("main thread ");

use std::thread;

// 使用 thread
let handle = thread::spawn(move || {
 println!("sub thread 1");

 let handle2 = thread::spawn(move || {
 println!("sub thread 2");
 thread::sleep(duration)
 });
 handle2.join().unwrap();
 thread::sleep(duration)
});

handle.join().unwrap();
```

```
thread::sleep(duration)

// rust 并不保证线程之间的引用之间的生命周期关系
// rust线程由操作系统调度
```

### 3.4.2 线程间共享数据

如果一个类型 T: Send，那么 T 在某个线程中的独占访问是线程安全的；如果一个类型 T: Sync，那么 T 在线程间的只读共享是安全的。一个类型 T 满足 Sync trait，当且仅当 &T 满足 Send trait

如果一个类型 T 实现了 Send trait，意味着 T 可以安全地从一个线程移动到另一个线程，也就是说所有权可以在线程间移动

如果一个类型 T 实现了 Sync trait，则意味着 &T 可以安全地在多个线程中共享。一个类型 T 满足 Sync trait，当且仅当 &T 满足 Send trait

基本原生类型都支持 sync 和 send，不支持的有：

裸指针 \*const T / \*mut T。它们是不安全的，所以既不是 Send 也不是 Sync

UnsafeCell 不支持 Sync。也就是说，任何使用了 Cell 或者 RefCell 的数据结构不支持 Sync

引用计数 Rc 不支持 Send 也不支持 Sync。所以 Rc 无法跨线程

```
pub fn spawn<F, T>(f: F) -> JoinHandle<T>
where
 F: FnOnce() -> T,
 F: Send + 'static,
 T: Send + 'static,
```

'static 意思是闭包捕获的自由变量必须是一个拥有所有权的类型，或者是一个拥有静态生命周期的引用；Send 意思是，这些被捕获自由变量的所有权可以从一个线程移动到另一个线程

1. 手动实现必要的trait：共享借用和所有权类型的数据
2. 使用第三方库 crossbeam：共享借用 / 可变借用
3. 使用Arc和Mutex

```
// 在线程间共享数据

// 案例 1 通过借用检查，消除数据竞争
use std::thread;
let mut v = vec![1, 2, 3, 4];
// thread::spawn(move || v.push(5)); // v 只能使用1次，无法使用for 循环迭代加入元素

// 借用规则要求可变借用只能有一次，避免了数据竞争（多个线程同时使用 v ）
// for i in 0..10 {
// thread::spawn(move || v.push(i));
// }

// 案例 2 通过函数来传递数据，也不被允许
// 线程中没法传递引用，因为不知道线程执行顺序
```

```

// 如果线程封装在函数中，不知道函数会被在哪里调用以及调用多少次

// fn inner_func(vref: &mut Vec<u32>) {
// std::thread::spawn(move || vref.push(3));
// }

// 案例 4 只读也不能通过函数传递吗？ 不能，可能存在悬垂指针

// fn inner_func(vref: &Vec<u32>) {
// std::thread::spawn(move || println!("{}: {:?}", vref));
// }

// 案例 5 如何在线程间传递引用

// 5.1 不使用第三方库的实现（加'static'类型）
use std::fmt;
struct Foo {
 string: String,
 v: Vec<f64>,
}

impl fmt::Display for Foo {
 fn fmt(&self, f: &mut fmt::Formatter<'_>) -> fmt::Result {
 write!(f, "{}: {:?}", self.string, self.v)
 }
}

// 封装到函数中
// 但是存在约束条件
fn test<T: Send + Sync + fmt::Display + 'static>(val: T) {
 thread::spawn(move || println!("{}: {:?}", val));
}

test("hello"); // &'static str
test(String::from("hello")); // String，因为它是所有权的数据，与程序同生同灭
test(5); // i32

// 内部的数据是由所有权的，所以也可以作为参数传递
let foo = Foo {
 string: String::from("hello"),
 v: vec![1.2, 2.2, 3.2, 42.2],
};
test(foo);
// test(foo); 不能使用第二次

use std::time::Duration;
thread::sleep(Duration::new(1, 0));

// 5.2 使用第三方库crossbeam 的实现

```

```

// crossbeam::scope 共享数据
use crossbeam;
let mut vec = vec![1, 2, 3, 4, 5];

crossbeam::scope(|scope| {
 // scope 出来的子线程会在主线程关闭之前回收
 // 保证不会出现悬垂指针
 scope.spawn(move |_| {
 for e in &vec {
 println!("{}:?", e);
 }
 });
})
.expect("a child thread panicked");

let mut v = vec![1, 2, 3, 4, 5];

crossbeam::scope(|scope| {
 // scope 出来的子线程会在主线程关闭之前回收
 // 不出现数据竞争
 for e in &mut v {
 scope.spawn(move |_| thread::sleep(Duration::from_secs(1)));
 }
})
.expect("a child thread panicked");

use std::sync::{Arc, Mutex};
// 5.3 也可以使用Arc和Mutex实现共享所有权
let v = Arc::new(Mutex::new(vec![1, 2, 3]));

// 每次都克隆一个
for i in 0..3 {
 let cloned_v = v.clone();
 thread::spawn(move || {
 cloned_v.lock().unwrap().push(i);
 });
}

```

### 3.4.3 构建无悔的并发系统

并发编程需要注意的三点：

原子性：保证操作是原子的

可见性：保证数据是同步的

顺序性：保证操作的顺序是正确的

并发编程的方式：

同步锁和无锁编程

锁带来的问题

性能：无锁编程可以最大化减少线程上下文切换、线程等待

死锁：引入无锁编程就不会产生死锁

无锁编程主要依靠原子类型，性能上并不总是优于锁编程

无锁编程和计算机组成密切相关：现代计算机一般都是多核三级缓存，带来缓存一致性问题；CPU指令重排；编译器指令重排。用内存屏障解决问题

内存屏障允许开发者在编写代码时在需要的地方加入它：内存屏障是指一种操作，它确保在该操作之前的内存访问完成，并且在该操作之后的内存访问不会在该操作之前执行。这有助于在多线程环境中维护内存的一致性和避免数据竞争。

CPU有四种屏障

内存模型：获取语义和释放语义

#### 1. 多线程并发

使用channel 和 condvar 模拟并行组件。Rust 只保证语言层面的安全，逻辑层面的安全并不保证

并发模型的最佳默认模式：事件循环 (event-loop)

### 3.4.4 无锁并发

#### 2. 无锁并发

原子类型：原子布尔值和数字，都提供了Ordering内存顺序：5种顺序，和LLVM以及C++20一致

原子类型还分硬件架构，ARM上的Linux没有原子类型

Rust提供了条件编译

内存顺序

```
pub enum Ordering {

 Relaxed, 原子类型只保证原子操作，不指定内存顺序（不指定内存屏障）
 Release, 当前线程内的所有写操作，对于其他对这个原子变量进行acquire得线程可见
 Acquire, 可以保证读到所有在Release之前发生的写入
 AcqRel, 对读取和写入施加acquire-release 语义，无法被重排
 SeqCst,
}
```

原子类型提供的方法：使用支持硬件的指令和方法

ABA问题

可以关注的库

### 3.5 trait 和泛型

特殊的trait默认值 Global，它是Rust默认的全局分配器，在程序运行时分配和释放内存

A 这个参数有默认值 Global，它是 Rust 默认的全局分配器，这也是为什么 Vec 虽然有两个参数，使用时都只需要用 T。

```
pub struct Vec<T, A: Allocator = Global> {
 buf: RawVec<T, A>,
 len: usize,
}

pub struct RawVec<T, A: Allocator = Global> {
 ptr: Unique<T>,
 cap: usize,
 alloc: A,
}
```

让一个类型拥有方法有两种方式：自定义关联函数（使用其他类型作为泛型约束），为其实现 trait

特殊的泛型参数 Rhs 代表加号右边的值，在 add 方法中默认是 Self，也就是如果不提供泛型参数，左值和右值相同类型

```
pub trait Add<Rhs = Self> {
 type Output;
 #[must_use]
 fn add(self, rhs: Rhs) -> Self::Output;
}
```

等价写法

```
fn name(animal: impl Animal) -> &'static str { animal.name() }
```

```
fn name<T: Animal>(animal: T) -> &'static str;
```

注意区别特设多态、子类型多态

### 3.5.1 trait

#### trait 四种作用

接口

类型标记

泛型限定

抽象类型

接口也是一种多态

作为泛型的限定

```
// trait 作为泛型限定
use std::string::ToString;

fn print<T: ToString>(v: T) {
 println!("{}", v.to_string());
}
```

抽象类型 (trait object) : 因为trait中包含了很多方法，在运行时都化作trait对象。用一个trait 对象可以表示同样实现了 trait的多种类型

我们无法在编译期给定一个具体类型，所以我们要有一种手段，告诉编译器，此处需要并且仅需要任何实现了该 trait的数据类型。在 Rust 里，这种类型叫 Trait Object，表现为 &dyn Trait 或者 Box,这里的dyn仅仅是用来区分 trait 类型还是普通类型

```
pub fn format(input: &mut String, formatters: Vec<&dyn Formatter>) {
 for formatter in formatters {
 formatter.format(input);
 }
}
```

trait object 是动态分派

```
pub trait Formatter {
 fn format(&self, input: &mut String) -> bool;
}

struct MarkdownFormatter;
impl Formatter for MarkdownFormatter {
 fn format(&self, input: &mut String) -> bool {
 input.push_str("\nformatted with Markdown formatter");
 true
 }
}

struct RustFormatter;
impl Formatter for RustFormatter {
 fn format(&self, input: &mut String) -> bool {
 input.push_str("\nformatted with Rust formatter");
 true
 }
}

struct HtmlFormatter;
impl Formatter for HtmlFormatter {
```

```

fn format(&self, input: &mut String) -> bool {
 input.push_str("\nformatted with HTML formatter");
 true
}

pub fn format(input: &mut String, formatters: Vec<&dyn Formatter>) {
 for formatter in formatters {
 formatter.format(input);
 }
}

fn main() {
 let mut text = "Hello world!".to_string();
 let html: &dyn Formatter = &HtmlFormatter;
 let rust: &dyn Formatter = &RustFormatter;
 let formatters = vec![html, rust];
 format(&mut text, formatters);

 println!("text: {}", text);
}

```

trait 有两种分发类型：静态分发（单态化）：生成具体类型的函数

静态分发还有一种语法：impl trait

```

// 静态分发: impl trait

use std::fmt::Display;

// 返回一个实现了 Display trait 的类型
fn make_value<T: Display>(index: usize) -> impl Display {
 match index {
 0 => "Hello,World",
 1 => "Hello,world (1)",
 _ => panic!(),
 }
}

println!("{}", make_value::<&'static str>(0));
println!("{}", make_value::<&'static str>(1))

```

trait与生命周期

```

// trait 与生命周期
// fn make_debug<T>(_: T) -> impl std::fmt::Debug {
// 42u8

```

```

// }

// late bound
fn make_debug<'a, T: 'static>(_: &'a T) -> impl std::fmt::Debug {
 42u8
}

fn test() -> impl std::fmt::Debug {
 let value = "value".to_string();
 make_debug(&value)
}

```

## 3.5.2 trait 对象

是动态分发的一种

Any是Rust中仅有的一种自省机制，相当于反射机制。因为rust是编译型语言，所以作用有限，智能识别static（不能是引用类型），在运行时反射。

```

// 实现了Any trait 的类型到具体类型的转换
use std::fmt::Debug;

// 当函数参数是string时，可以转换为具体类型，否则什么都不干
fn log<T: Any + Debug>(value: &T) {
 let value_any = value as &dyn Any; // 先转为trait 对象
 match value_any.downcast_ref::<String>() {
 // 转为 String
 Some(as_string) => {
 println!("String ({}): {}", as_string.len(), as_string)
 }
 None => println!("{:?}", value),
 }
}

fn do_work<T: Any + Debug>(value: &T) {
 log(value)
}

let my_string = "hello world".to_string();
do_work(&my_string);
let my_i8 = 100;
do_work(&my_i8);

```

TypeId是全局唯一，当程序重新启动会发生变化

trait 对象：也是一组方法的集合。当我们在运行时想让某个具体类型，只表现出某个 trait 的行为，可以通过将其赋值给一个 dyn T，无论是 &dyn T，还是 Box，还是 Arc，都可以，这里，T 是当前数据类型实现的某个 trait。此时，原有的类型被抹去，Rust 会创建一个 trait object，并为其分配满足该 trait 的 vtable

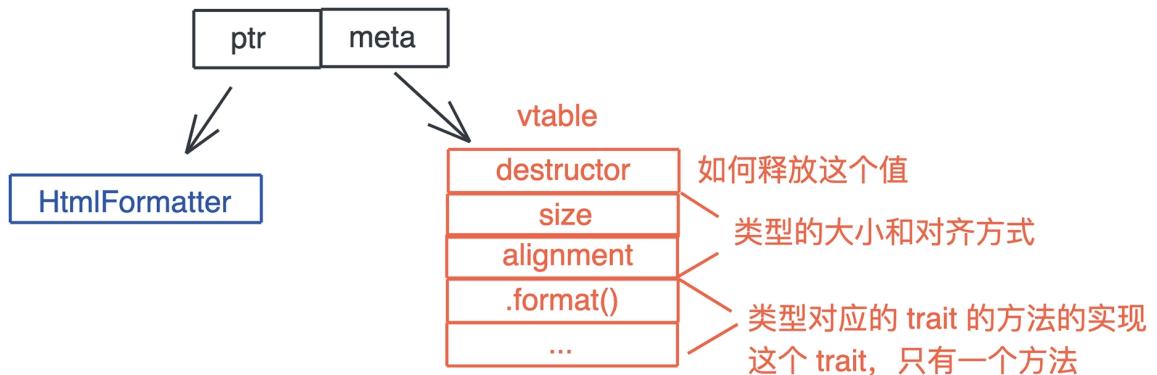
```

let mut text = "Hello world!".to_string();

let formatter: &dyn Formatter = &HtmlFormatter; // 使用 &HtmlFormatter 赋值给
 // &Formatter 创建一个 trait object

formatter.format(&mut text); // 调用 trait 的方法

```



在编译 dyn T 时, Rust 会为使用了 trait object 类型的 trait 实现, 生成相应的 vtable, 放在可执行文件中 (一般在 TEXT 或 RODATA 段) :

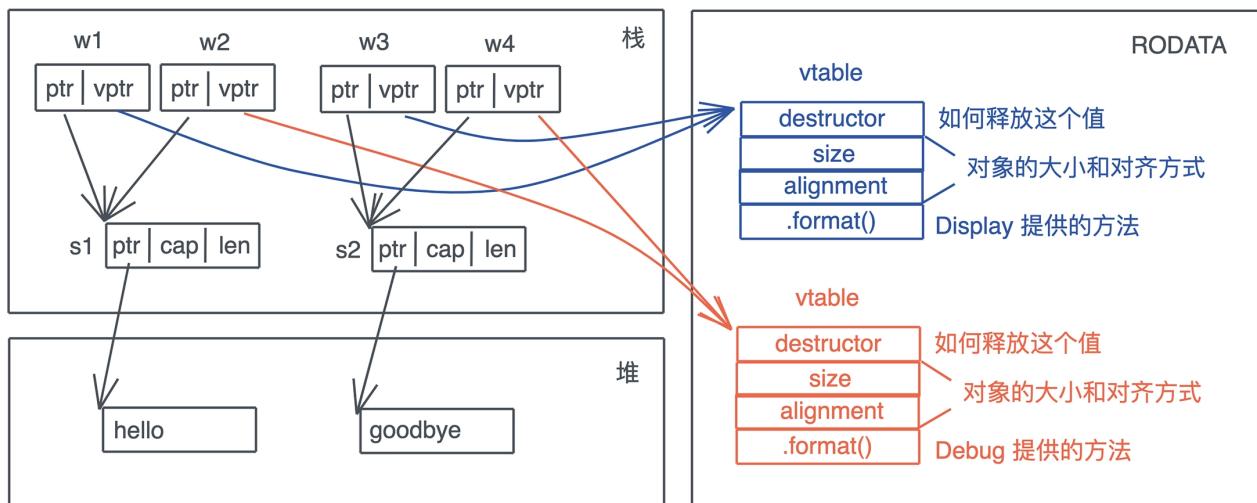
```

let s1 = String::from("hello");
let s2 = String::from("goodbye");

// Display / Debug trait object for s1
let w1: &dyn Display = &s1;
let w2: &dyn Debug = &s1;

// Display / Debug trait object for s2
let w3: &dyn Display = &s2;
let w4: &dyn Debug = &s2;

```



这样，当 trait object 调用 trait 的方法时，它会先从 vptr 中找到对应的 vtable，进而找到对应的方法来执行。

使用 trait object 的好处是，当在某个上下文中需要满足某个 trait 的类型，且这样的类型可能有很多，当前上下文无法确定会得到哪一个类型时，我们可以用 trait object 来统一处理行为。和泛型参数一样，trait object 也是一种延迟绑定，它让决策可以延迟到运行时，从而得到最大的灵活性

trait object 把决策延迟到运行时，带来的后果是执行效率的打折。在 Rust 里，函数或者方法的执行就是一次跳转指令，而 trait object 方法的执行还多一步，它涉及额外的内存访问，才能得到要跳转的位置再进行跳转，执行的效率要低一些

此外，如果要把 trait object 作为返回值返回，或者要在线程间传递 trait object，都免不了使用 Box 或者 Arc，会带来额外的堆分配的开销

```
&dyn Trait or Box<dyn Trait>
```

```
use core::any::{Any,TypeId};
use std::sync::Arc;

// 模拟类
// 类的实例相当于trait 对象
struct Class {
 name: String,
 type_id: TypeId,
}

impl Class {
 fn new<T: 'static>() -> Self {
 Class {
 name: std::any::type_name::<T>().to_string(),
 type_id: TypeId::of::<T>(),
 }
 }
}

struct Instance {
 inner: Arc<dyn Any>, //相当于 Box<T>
}

impl Instance {
 fn new(obj: impl Any) -> Self {
 Self {
 inner: Arc::new(obj),
 }
 }
}

fn instance_of(&self, class: &Class) -> bool {
 self.inner.as_ref().type_id() == class.type_id
}
```

```

struct Foo {};
struct Bar {};

let foo_class = Class::new::<Foo>();
let bar_class = Class::new::<Bar>();

let foo_instance = Instance::new(Foo {});

assert!(foo_instance.instance_of(&foo_class));
assert!(!foo_instance.instance_of(&bar_class));

```

### 3.5.2.1 泛型和trait 对象实现模版方法

多个类型实现同一个trait

代表项目：actix-extras

### 3.5.2.2 trait对象的本质

Trait Object 的底层逻辑就是胖指针。其中，一个指针指向数据本身，另一个则指向虚函数表（vtable）。所以，Rust 里的 Trait Object 没什么神秘的，它不过是我们熟知的 C++ / Java 中 vtable 的一个变体而已

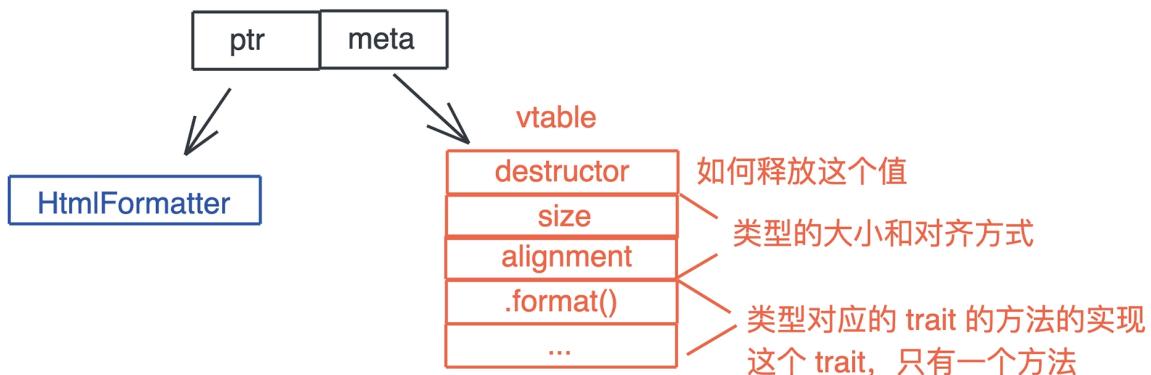
```

let mut text = "Hello world!".to_string();

let formatter: &dyn Formatter = &HtmlFormatter; // 使用 &HtmlFormatter 赋值给
 // &Formatter 创建一个 trait object

formatter.format(&mut text); // 调用 trait 的方法

```



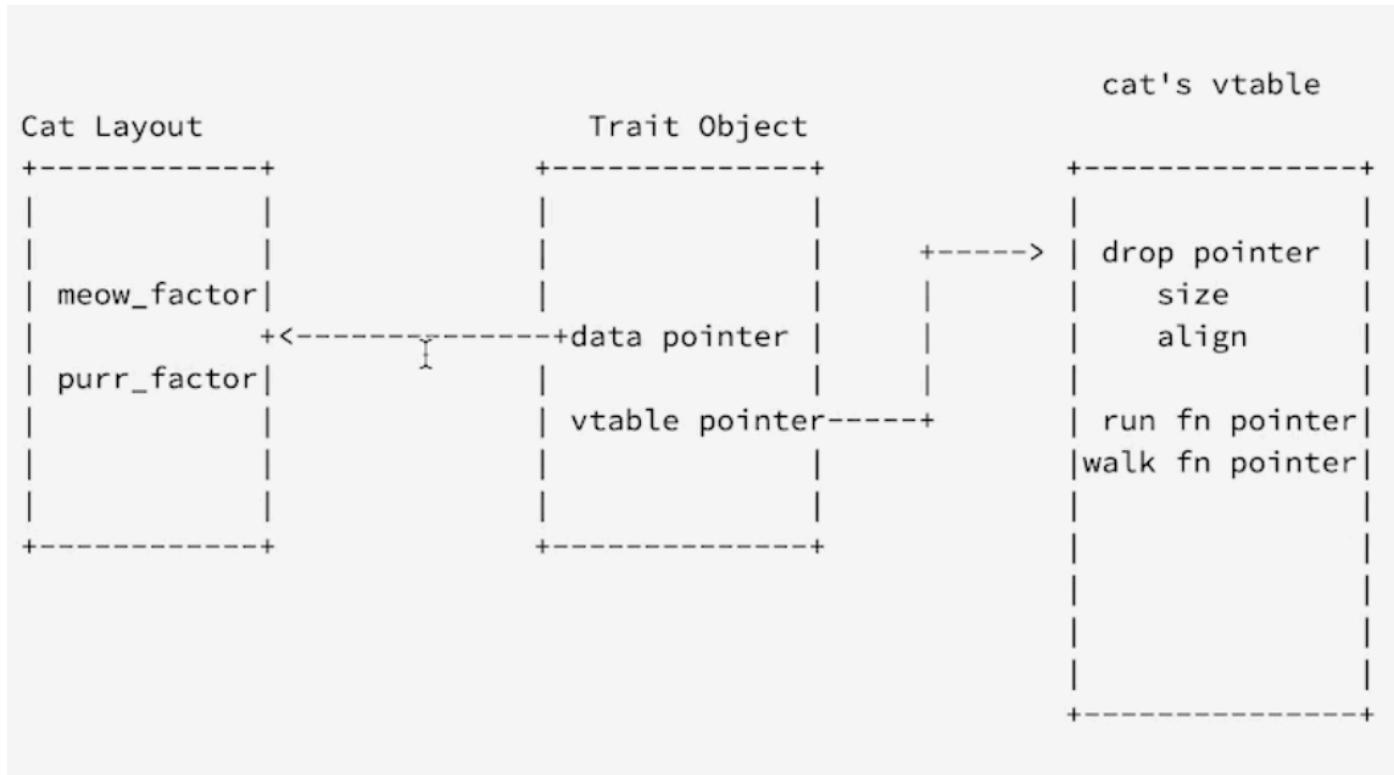
 极客时间

在运行时，一旦使用了关于接口的引用，变量原本的类型被抹去，我们无法单纯从一个指针分析出这个引用具备什么样的能力。因此，在生成这个引用的时候，我们需要构建胖指针，除了指向数据本身外，还需要指向一张涵盖了这个接口所支持方法的列表。这个列表，就是我们熟知的虚表（virtual table）。

trait定义了共同的行为

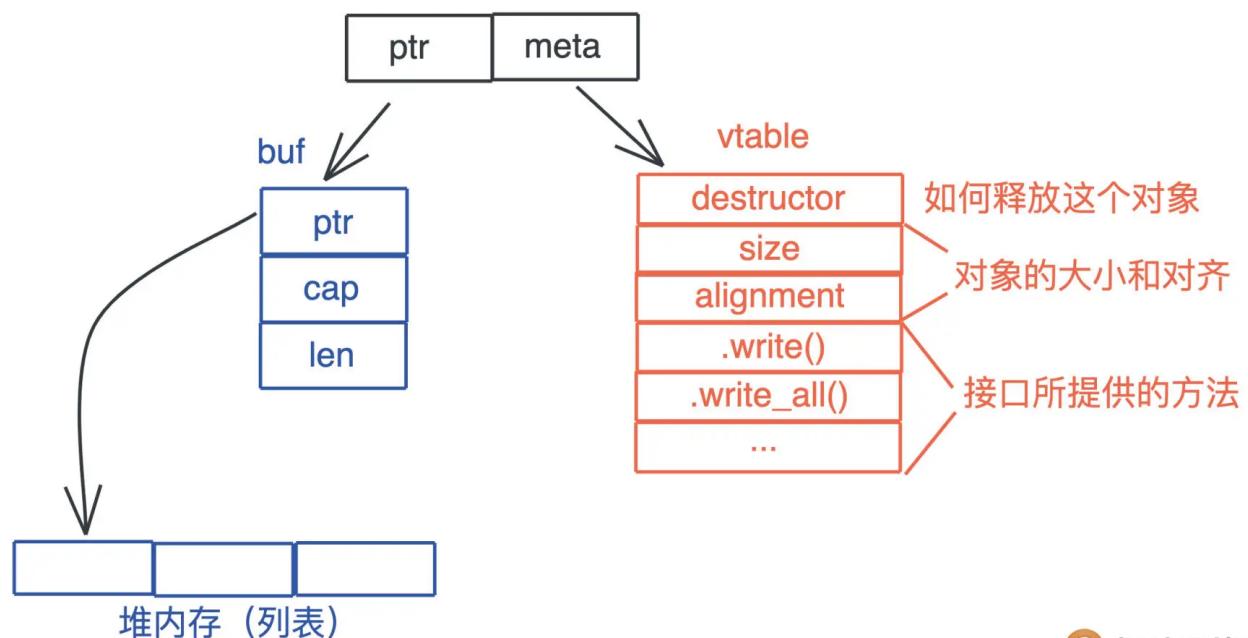
vtable存的是函数指针集

trait 对象本质上是一个虚表



```
use std::io::Write;

let mut buf: Vec<u8> = vec![];
let writer: &mut Write = &mut buf; // 构建一个带有虚表的 trait object
```



### 3.5.2.3 trait 对象安全的本质

当多个类型实现了trait中的方法时，返回类型实例无法确定。对象安全就要确保方法能被安全的调用

编译器如何确保对象安全？如果trait能实现自己就是对象安全的

如果 trait 所有的方法，返回值是 Self 或者携带泛型参数，那么这个 trait 就不能产生 trait object。如果一个 trait 只有部分方法返回 Self 或者使用了泛型参数，那么这部分方法在 trait object 中不能调用。

```
trait StarkFamily {
 fn last_name(&self) -> &'static str;
 fn totem(&self) -> &'static str;
}

trait TullyFamily {
 fn territory(&self) -> &'static str;
}

trait Children {
 fn new(first_name: &'static str) -> Self
 where
 Self: Sized;

 fn first_name(&self) -> &'static str;
}

impl StarkFamily for dyn Children {
 fn last_name(&self) -> &'static str {
 "Stark"
 }

 fn totem(&self) -> &'static str {
 "Wolf"
 }
}

impl TullyFamily for dyn Children {
 fn territory(&self) -> &'static str {
 "Riverrun City"
 }
}

struct People {
 first_name: &'static str,
}

impl Children for People {
 fn new(first_name: &'static str) -> Self
 where
```

```

 Self: Sized,
}

 println!("hello,{:?}", first_name);
 People {
 first_name: first_name,
 }
}
fn first_name(&self) -> &'static str {
 self.first_name
}

fn fully_name(person: Box<dyn Children>) {
 println!(
 "--- Winter is coming, the lone {:?} dies, the packs lives ---",
 person.totem()
);

 let full = format!("{} {}", person.first_name(), person.last_name());
 println!("I'm {:?}", full);

 println!("My mother come from {:?}", person.territory());
}

let sansa = People::new("Sansa");
let aray = People::new("Aray");

let starks = Box::new(sansa);
fully_name(starks);

let starks = Box::new(aray);
fully_name(starks)

```

维护了两个虚表，safe\_table,nosafe\_vatble,where Self:sized,nosafe\_vtable

### 3.5.2.4 使用Enum 替代trait

当trait对象无法保证安全时的替代方案

trait 对象性能比较差，因为它在运行时，想要提高性能可以转为enum

```

// 类型不同，行为相同，通过trait实现
trait KnobControl {
 fn set_position(&mut self, value: f64);
 fn get_value(&self) -> f64;
}

struct LinearKnob {
 position: f64,
}

```

```

struct LogarithmicKnob {
 position: f64,
}

impl KnobControl for LinearKnob {
 fn set_position(&mut self, value: f64) {
 self.position = value
 }
 fn get_value(&self) -> f64 {
 self.position
 }
}

impl KnobControl for LogarithmicKnob {
 fn set_position(&mut self, value: f64) {
 self.position = value
 }

 fn get_value(&self) -> f64 {
 (self.position + 1.).log2()
 }
}

// 通过enum实现
// 将类型抽象到枚举体中

enum Knob {
 Linear(LinearKnob),
 Logarithmic(LogarithmicKnob),
}

impl KnobControl for Knob {
 fn set_position(&mut self, value: f64) {
 match self {
 Knob::Linear(inner_knob) => inner_knob.set_position(value),
 Knob::Logarithmic(inner_knob) => inner_knob.set_position(value),
 }
 }

 fn get_value(&self) -> f64 {
 match self {
 Knob::Linear(inner_knob) => inner_knob.get_value(),
 Knob::Logarithmic(inner_knob) => inner_knob.get_value(),
 }
 }
}

```

```

use core::ops::Add;
// 类型不同，行为相同，通过trait实现
trait KnobControl<T: Add + Add<Output = T> + Copy> {
 fn set_position(&mut self, value: T);
 fn get_value(&self, p: T) -> T;
}

struct LinearKnob<T: Add + Add<Output = T> + Copy> {
 position: T,
}

struct LogarithmicKnob<T: Add + Add<Output = T> + Copy> {
 position: T,
}

impl<T: Add + Add<Output = T> + Copy> KnobControl<T> for LinearKnob<T> {
 fn set_position(&mut self, value: T) {
 self.position = value
 }
 fn get_value(&self, p: T) -> T {
 self.position
 }
}

impl<T: Add + Add<Output = T> + Copy> KnobControl<T> for LogarithmicKnob<T> {
 fn set_position(&mut self, value: T) {
 self.position = value
 }

 fn get_value(&self, p: T) -> T {
 self.position + p
 }
}

// 通过enum实现
// 将类型抽象到枚举体中

enum Knob<T: Add + Add<Output = T> + Copy> {
 Linear(LinearKnob<T>),
 Logarithmic(LogarithmicKnob<T>),
}

impl<T: Add + Add<Output = T> + Copy> KnobControl<T> for Knob<T> {
 fn set_position(&mut self, value: T) {
 match self {
 Knob::Linear(inner_knob) => inner_knob.set_position(value),
 Knob::Logarithmic(inner_knob) => inner_knob.set_position(value),
 }
 }
}

```

```

fn get_value(&self, value: T) -> T {
 match self {
 Knob::Linear(inner_knob) => inner_knob.get_value(value),
 Knob::Logarithmic(inner_knob) => inner_knob.get_value(value),
 }
}
}
}

```

### 基本 trait

可以为数据结构定义抽象的接口  
使用 self 可以访问实现 trait 的数据结构

```

pub trait Write {
 fn write(&mut self, buf: &[u8]) -> Result<usize>;
 ...
}

```

### 带关联类型的 trait

trait 内部可以定义和这个 trait 关联的类型

```

pub trait Iterator {
 type Item;
 fn next(&mut self) -> Option <Self::Item>;
 ...
}

```

### 动态分派

使用 trait object

```

pub trait Formatter {
 fn format(&self, input: &mut String) -> bool;
}
pub fn format(input: &mut String, formatters: Vec<dyn Formatter>) {
 for formatter in formatters {
 formatter.format(input);
 }
}

```

### 静态分派

使用泛型函数

```

trait Animal {
 fn name(&self) -> &'static str;
}
fn name(animal: impl Animal) -> &'static str {
 animal.name()
}

```

### 泛型 trait

trait 可以有一个甚至多个泛型参数

```

pub trait From<T> {
 fn from(T) -> Self;
}

pub trait Service <Request> {
 type Response;
 type Error;
 // Future 类型受 Future trait 约束
 type Future: Future;
 fn poll_ready(
 &mut self,
 cx: &mut Context<'_>
) -> Poll<Result<(), Self::Error>>;
 fn call(&mut self, req: Request) -> Self::Future;
}

```

## 3.5.2.5 trait 覆盖实现

Rust trait中的方法不允许覆盖实现

但是可以使用trait 对象实现

### 3.5.2.6 trait 与 Self: Sized

什么时候需要用到。Rust中所有类型， 默认都是T: Sized

```
// trait 中有默认实现时
// 并且默认实现的函数体中包含Self
trait WithConstructor {
 fn build(param: usize) -> Self
 where
 Self: Sized;
 fn new(param: usize) -> Self
 where
 Self: Sized,
 {
 Self::build(0)
 }

 fn t(&self);
}

struct A;

impl WithConstructor for A {
 fn t(&self) {
 println!("hello");
 }
 fn build(param: usize) -> Self
 where
 Self: Sized,
 {
 A
 }
}

let a = &A;
a.t()
```

### 3.5.2.7 trait 对象与Box

```
trait Test {
 fn foo(&self);

 fn works(self: Box<Self>) {
 println!("hello");
 }

 fn fails(self: Box<Self>)
 // where
 // Self: Sized, //限定了被调用,关闭; ? Sized 在类型声明时使用
```

```

 {
 self.foo();
 }
}

struct Concrete;

impl Concrete {
 fn hello(&self) {
 println!("hello");
 }
}

impl Test for Concrete {
 fn foo(&self) {
 ()
 }
 fn works(self: Box<Self>) {
 self.hello();
 }
 // 没有实现fails
}
}

let concrete: Box<dyn Test> = Box::new(Concrete);
// concrete.fails();
concrete.works();

```

### 3.5.2.8 标记 trait

Sized / Send / Sync / Unpin / Copy

```

pub unsafe auto trait Send {}
pub unsafe auto trait Sync {}

```

auto 意味着编译器会在合适的场合，自动为数据结构添加它们的实现，如果开发者手工实现这两个 trait，要自己为它们的安全性负责

类型转换 trait From / Into / AsRef / AsMut

对值类型的转换和对引用类型的转换，Rust 提供了两套不同的 trait：值类型到值类型的转换：From / Into / TryFrom / TryInto 引用类型到引用类型的转换：AsRef / AsMut

**From / Into**

```
pub trait From<T> {
 fn from(T) -> Self;
}

pub trait Into<T> {
 fn into(self) -> T;
}
```

```
// 实现 From 会自动实现 Into, 所以需要的时候, 不要去实现 Into, 只要实现 From 就好
impl<T, U> Into<U> for T where U: From<T> {
 fn into(self) -> U {
 U::from(self)
 }
}
```

From 和 Into 还是自反的：把类型 T 的值转换成类型 T，会直接返回

```
// From (以及 Into) 是自反的
impl<T> From<T> for T {
 fn from(t: T) -> T {
 t
 }
}
```

```
use std::net::{IpAddr, Ipv4Addr, Ipv6Addr};

fn print(v: impl Into<IpAddr>) {
 println!("{:?}", v.into());
}

fn p<T: Into<IpAddr>>(v: T) {
 println!("{:?}", v.into());
}

fn main() {
 let v4: Ipv4Addr = "2.2.2.2".parse().unwrap();
 let v6: Ipv6Addr = "::1".parse().unwrap();

 // IPAddr 实现了 From<[u8; 4], 转换 IPv4 地址
 print([1, 1, 1, 1]);
 // IPAddr 实现了 From<[u16; 8], 转换 IPv6 地址
 print([0xfe80, 0, 0, 0, 0xaede, 0x48ff, 0xfe00, 0x1122]);
 // IPAddr 实现了 From
 p(v4);
 // IPAddr 实现了 From
 p(v6);
}
```

注意，如果你的数据类型在转换过程中有可能出现错误，可以使用 TryFrom 和 TryInto，它们的用法和 From / Into 一样，只是 trait 内多了一个关联类型 Error，且返回的结果是 Result<T, Self::Error>

## AsRef / AsMut

```
pub trait AsRef<T> where T: ?Sized {
 fn as_ref(&self) -> &T;
}

pub trait AsMut<T> where T: ?Sized {
 fn as_mut(&mut self) -> &mut T;
}
```

```
enum Language {
 Rust,
 TS,
 Elixir,
 Haskell,
}

// 把一种类型转为引用，实际上就是转为另一种类型
impl AsRef<str> for Language {
 fn as_ref(&self) -> &str {
 match self {
 Language::Rust => "Rust",
 Language::TS => "TypeScript",
 Language::Elixir => "Elixir",
 Language::Haskell => "Haskell",
 }
 }
}

fn print_ref(v: impl AsRef<str>) {
 println!("{}: {}", v, v.as_ref());
}

fn main() {
 let rust = Language::TS;

 print_ref("hello world");

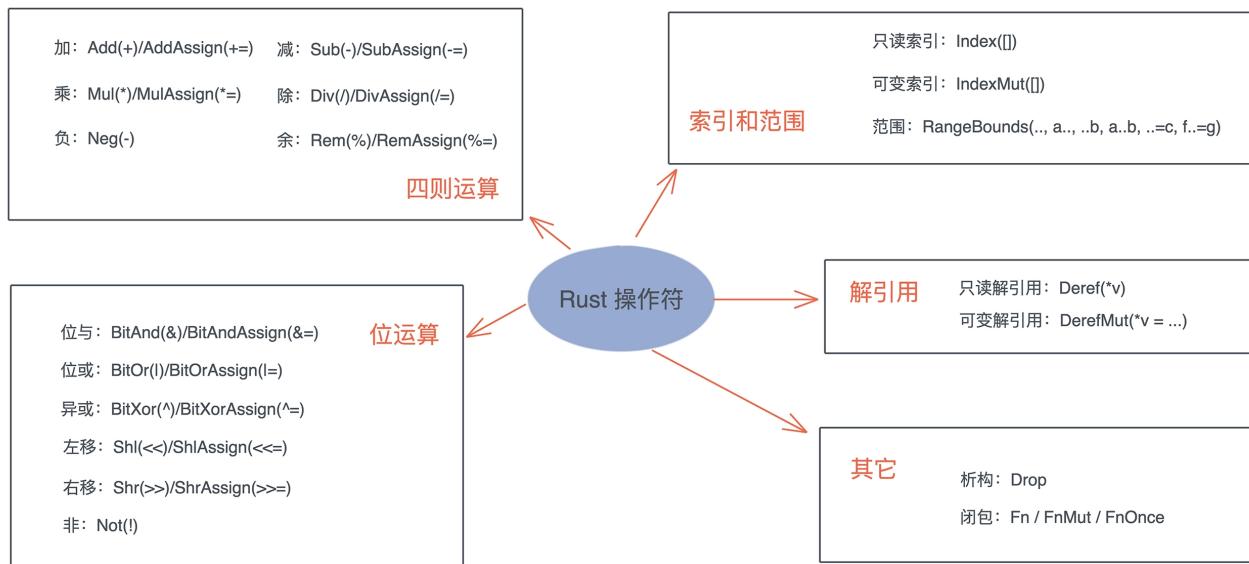
 print_ref("hello world".to_string());

 let r = rust.as_ref();
 println!("{}: {}", r, r);

 print_ref(rust)
}
```

如果你的代码出现 `v.as_ref().clone()` 这样的语句，也就是说你要对 `v` 进行引用转换，然后又得到了拥有所有权的值，那么你应该实现 `From`，然后做 `v.into()`

## 操作符相关：Deref / DerefMut



```
pub trait Deref {
 // 解引用出来的结果类型
 type Target: ?Sized;
 fn deref(&self) -> &Self::Target;
}

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

```
use std::ops::{Deref, DerefMut};

#[derive(Debug)]
struct Buffer<T>(Vec<T>);

impl<T> Buffer<T> {
 fn new(v: impl Into<Vec<T>>) -> Self {
 Self(v.into())
 }
}

impl<T> Deref for Buffer<T> {
 type Target = [T];
 fn deref(&self) -> &Self::Target {
```

```

 &self.0
 }
}

impl<T> DerefMut for Buffer<T> {
 fn deref_mut(&mut self) -> &mut Self::Target {
 &mut self.0
 }
}

fn main() {
 let mut buf = Buffer::new([1, 3, 2, 4, 9, 17, 200, 83, 21]);
 buf.sort();

 println!("buf: {:?}", buf)
}

```

其他: Debug / Display / Default

```

pub trait Debug {
 fn fmt(&self, f: &mut Formatter<'_>) -> Result<(), Error>;
}

pub trait Display {
 fn fmt(&self, f: &mut Formatter<'_>) -> Result<(), Error>;
}

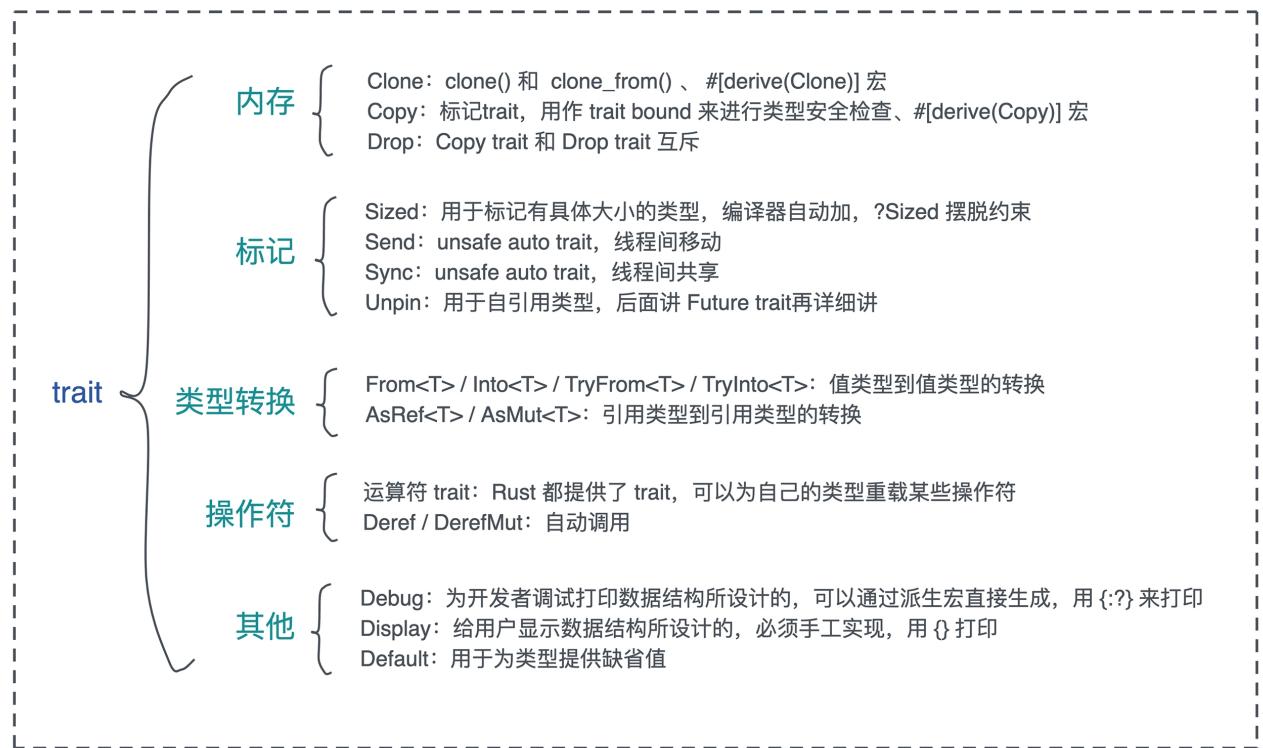
```

```

pub trait Default {
 fn default() -> Self;
}

```

Default trait 用于为类型提供缺省值。它也可以通过 derive 宏 #[derive(Default)]



trait 是行为的延迟绑定。我们可以在不知道具体要处理什么数据结构的前提下，先通过 trait 把系统的很多行为约定好。这也是为什么开头解释标准 trait 时，频繁用到了“约定.....行为”

### 3.5.3 泛型参数使用场景

#### 使用泛型参数延迟数据结构的绑定

```

/// Service 数据结构
pub struct Service<Store = MemTable> { // 指定了缺省值，使用时可以不提供泛型参数，直接使用缺省值，但也可以指定
 inner: Arc<ServiceInner<Store>>,
}

```

#### 使用泛型参数和幽灵数据（PhantomData）提供额外类型

PhantomData，声明数据结构中不直接使用，但在实现过程中需要用到的类型。它被广泛用在处理，数据结构定义过程中不需要，但是在实现过程中需要的泛型参数

```

use std::marker::PhantomData;

#[derive(Debug, Default, PartialEq, Eq)]
pub struct Identifier<T> {
 inner: u64,
 _tag: PhantomData<T>,
}

```

```

#[derive(Debug, Default, PartialEq, Eq)]
pub struct User {
 id: Identifier<Self>,
}

#[derive(Debug, Default, PartialEq, Eq)]
pub struct Product {
 id: Identifier<Self>,
}

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

 #[test]
 fn id_should_not_be_the_same() {
 let user = User::default();
 let product = Product::default();

 // 两个 id 不能比较，因为他们属于不同的类型
 // assert_ne!(user.id, product.id);

 assert_eq!(user.id.inner, product.id.inner);
 }
}

```

使用泛型参数让同一个数据结构对同一个 trait 可以拥有不同的实现。

### 使用泛型参数来提供多个实现

```

use std::marker::PhantomData;

#[derive(Debug, Default)]
pub struct Equation<IterMethod> {
 current: u32,
 _method: PhantomData<IterMethod>,
}

// 线性增长
#[derive(Debug, Default)]
pub struct Linear;

// 二次增长
#[derive(Debug, Default)]
pub struct Quadratic;

impl Iterator for Equation<Linear> {
 type Item = u32;
}

```

```

fn next(&mut self) -> Option<Self::Item> {
 self.current += 1;
 if self.current >= u32::MAX {
 return None;
 }

 Some(self.current)
}
}

impl Iterator for Equation<Quadratic> {
 type Item = u32;

 fn next(&mut self) -> Option<Self::Item> {
 self.current += 1;
 if self.current >= u16::MAX as u32 {
 return None;
 }

 Some(self.current * self.current)
 }
}

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

 #[test]
 fn test_linear() {
 let mut equation = Equation::<Linear>::default();
 assert_eq!(Some(1), equation.next());
 assert_eq!(Some(2), equation.next());
 assert_eq!(Some(3), equation.next());
 }

 #[test]
 fn test_quadratic() {
 let mut equation = Equation::<Quadratic>::default();
 assert_eq!(Some(1), equation.next());
 assert_eq!(Some(4), equation.next());
 assert_eq!(Some(9), equation.next());
 }
}

```

## 高级用法

如果想要实现返回值中带泛型参数，不能用impl trait，但是可以用 trait object

```
pub trait Storage {
 ...
 /// 遍历 HashTable, 返回 kv pair 的 Iterator
 fn get_iter(&self, table: &str) ->
 Result<Box<dyn Iterator<Item = Kvpair>>, KvError>;
}
```

但是使用 trait Object 是有代价的，除了有一次额外的堆分配之外，还有动态分配会带来一定的性能损失  
如何处理复杂的泛型参数

## 3.6 Rust语言编程范式

Rust支持面向对象语言的一些特性，也支持函数式语言的特性。函数式style:

1. 默认不可变，但是rust可变
2. 支持递归，但rust不支持尾递归优化（推荐递归而不是优化）
3. 函数式一等公民，有限的高阶函数支持
4. 和类型/积类型

Rust语言式混合范式

### 3.6.1 面向编译器编程

洋葱模型：编译器->核->标准库->第三方库

## 3.7 Rust 错误处理

Rust是基于返回值的错误机制

Rust整体的错误机制

1. 类型系统保证函数契约
2. 断言用于防御
3. Option消除空指针失败
4. Result<T,E> 传播错误
5. Panic恐慌

### 3.7.1 消除失败

1. 类型系统保证函数契约

```
// 1 类型系统保证函数契约
fn sum(a: i32, b: i32) -> i32 {
 a + b
}

// sum(1u32, 2u32) 违反函数契约
```

## 2. 断言用于防御

```
// 2 断言用于防御

fn extend_vec(v: &mut Vec<i32>, i: i32) {
 assert!(v.len() == 5);
 v.push(i)
}

let mut vec = vec![1, 2, 3];
extend_vec(&mut vec, 4);
extend_vec(&mut vec, 5);
assert_eq!(5, vec[4]);
extend_vec(&mut vec, 6); // panic
```

## 3.7.2 错误处理：Option

分层错误处理：Option 有无，Result 对错

Option提供了一些方法可以方便操作，如map

```
// 3 Option
let maybe_some_string = Some(String::from("hello, world!"));
let maybe_some_len = maybe_some_string.map(|s| s.len());
assert_eq!(maybe_some_len, Some(13))
```

```
// 返回值类型都是Option可以使用链式调用，不需要一个个unwrap处理
fn double(val: f64) -> f64 {
 val * 2.
}

fn square(val: f64) -> f64 {
 val.powi(2 as i32)
}

fn inverse(val: f64) -> f64 {
 val * -1.
}

fn log(val: f64) -> Option<f64> {
```

```

 match val.log2() {
 x if x.is_normal() => Some(x),
 _ => None,
 }
 }

fn sqrt(val: f64) -> Option<f64> {
 match val.sqrt() {
 x if x.is_normal() => Some(x),
 _ => None,
 }
}

let number = 20.;
let result = Option::from(number)
 .map(inverse)
 .map(double)
 .map(inverse)
 .and_then(log)
 .map(square)
 .and_then(sqrt);
match result {
 Some(x) => println!("x was {:?}", x),
 None => println!("this failed"),
}

```

map方法接受一个泛型参数，返回一个实现了FnOnce 闭包类型

### 3.7.3 Result

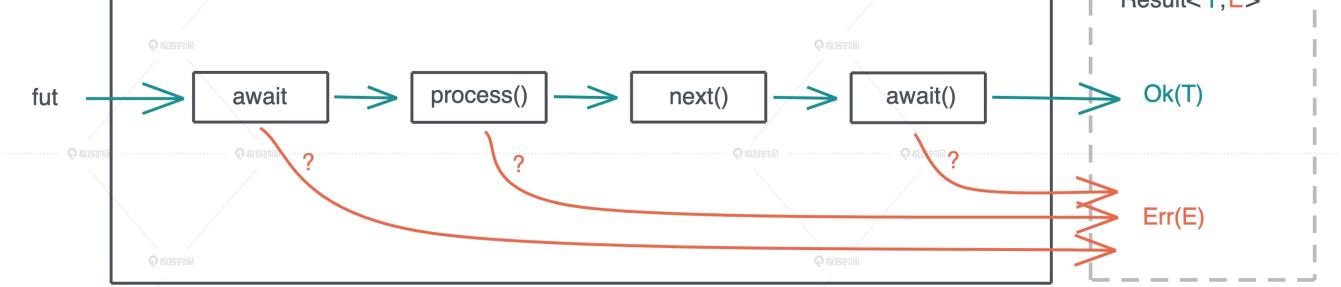
result的错误需要处理，当直接使用unwrap时，如果结果是Err，会发生Panic

什么样的Error才算，实现了Error trait，自定义必须实现该trait

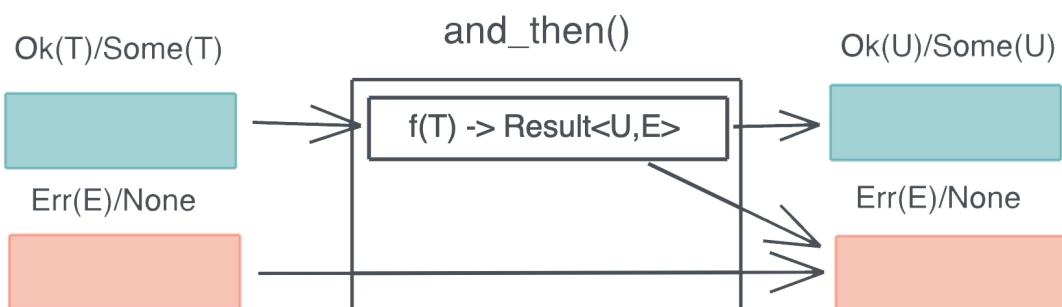
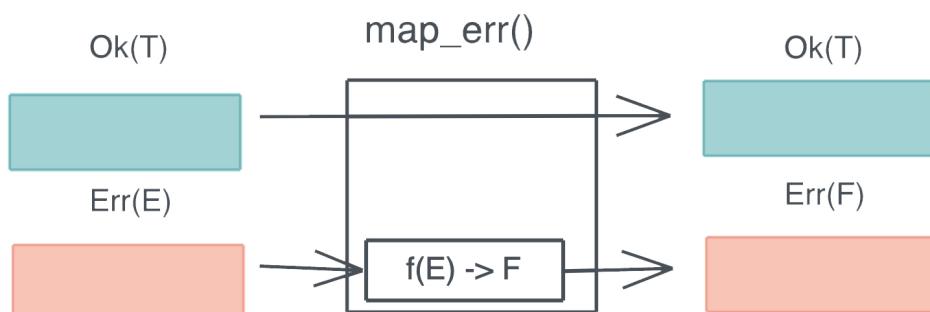
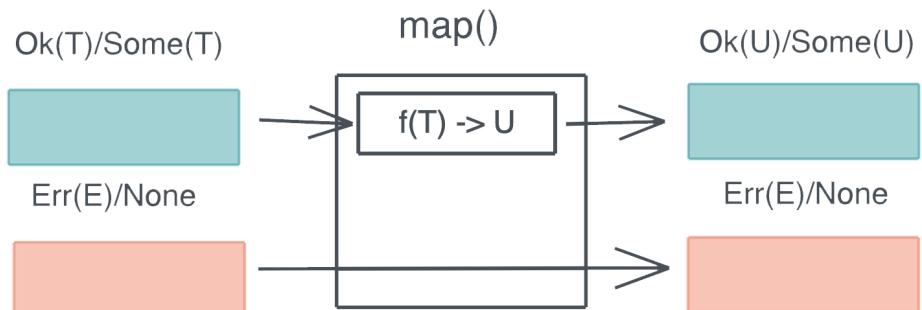
引用第三方库

使用?：如果你只想传播错误，不想就地处理，可以用?操作符

虽然?操作符使用起来非常方便，但你要注意在不同的错误类型之间是无法直接使用的，需要实现From trait 在二者之间建立起转换的桥梁，这会带来额外的麻烦



Rust 还为 Option 和 Result 提供了大量的辅助函数，如 map / map\_err / and\_then



```

Ok(data)
 .and_then(validate)
 .and_then(process)
 .map(transform)
 .and_then(store)
 .map_error(...)


```

执行流程



Option 和 Result 可以互换

### 3.7.4 Panic

Panic的两种类型: unwinding (栈展开) aborting (中止) 无法恢复

资源超过分配直接aborting

错误传播

如果想让错误传播, 可以把所有的 unwrap() 换成 ? 操作符, 并让 main() 函数返回一个 Result

```

fut
 .await?
 .process()?;
 .next();
 .await?;
}


```

Error trait 和错误类型的转换:

Result 里 E 是一个代表错误的数据类型。为了规范这个代表错误的数据类型的行为, Rust 定义了 Error trait

```

pub trait Error: Debug + Display {
 fn source(&self) -> Option<&(dyn Error + 'static)> { ... }
 fn backtrace(&self) -> Option<&Backtrace> { ... }
 fn description(&self) -> &str { ... }
 fn cause(&self) -> Option<&dyn Error> { ... }
}


```

Thiserror 和 anyhow已经简化了

```

use thiserror::Error;
#[derive(Error, Debug)]
#[non_exhaustive]
pub enum DataStoreError {
 #[error("data store disconnected")]
 Disconnect(#[from] io::Error),
 #[error("the data for key `{}` is not available")]
 Redaction(String),
 #[error("invalid header (expected {expected:?}, found {found:?}))")]
 InvalidHeader {
 expected: String,
 found: String,
 },
 #[error("unknown data store error")]
 Unknown,
}

```

如果你在撰写一个 Rust 库，那么 thiserror 可以很好地协助你对这个库里所有可能发生的错误进行建模。而 anyhow 实现了 anyhow::Error 和任意符合 Error trait 的错误类型之间的转换，让你可以使用 ? 操作符，不必再手工转换错误类型。

作为一名严肃的开发者，建议在开发前，先用类似 thiserror 的库定义好你项目中主要的错误类型，并随着项目的深入，不断增加新的错误类型，让系统中所有的潜在错误都无所遁形

## 3.8 元编程

### 3.8.1 反射

Any: Rust 中唯一反射，运行时反射

因为 Rust 是编译型语言，没有在运行时提供很多的反射功能。并且只有 `'static` 的类型才能支持动态运行时反射。

```

// case 1 反射
fn log<T: Any + Debug>(value: &T) {
 // 将具体类型转换为 trait 对象
 let value_any = value as &dyn Any;

 // 反射，判断类型，也叫自省
 match value_any.downcast_ref::<String>() {
 Some(as_string) => println!("string ({})：{}", as_string.len(), as_string),
 None => {
 println!("{:?}", value)
 }
 }
}

fn do_work<T: Any + Debug>(value: &T) {
 log(value)
}

```

```
}

let my_string = "hello world".to_string();
do_work(&my_string);
let my_i8: i8 = 100;
do_work(&my_i8);

// 反射如何实现
// pub trait Any: 'static {
// pub fn type_id(&self) -> TypeId;
// }

// 为dyn Any实现了 fn is<T:Any>(&self) -> bool;方法
// 他也是线程安全的

// case 2

use std::any::Any;

trait Foo: Any {
 fn as_any(&self) -> &dyn Any;
}

impl<T: Any> Foo for T {
 fn as_any(&self) -> &dyn Any {
 self
 }
}

#[derive(Debug)]
struct Bar {}

#[derive(Debug)]
struct Baz {}

impl PartialEq for dyn Foo {
 fn eq(&self, other: &dyn Foo) -> bool {
 let me = self.as_any();
 let you = other.as_any();

 if me.is::() && you.is::() {
 true
 } else {
 false
 }
 }
}

let bar = Bar {};
let baz = Baz {};
```

```

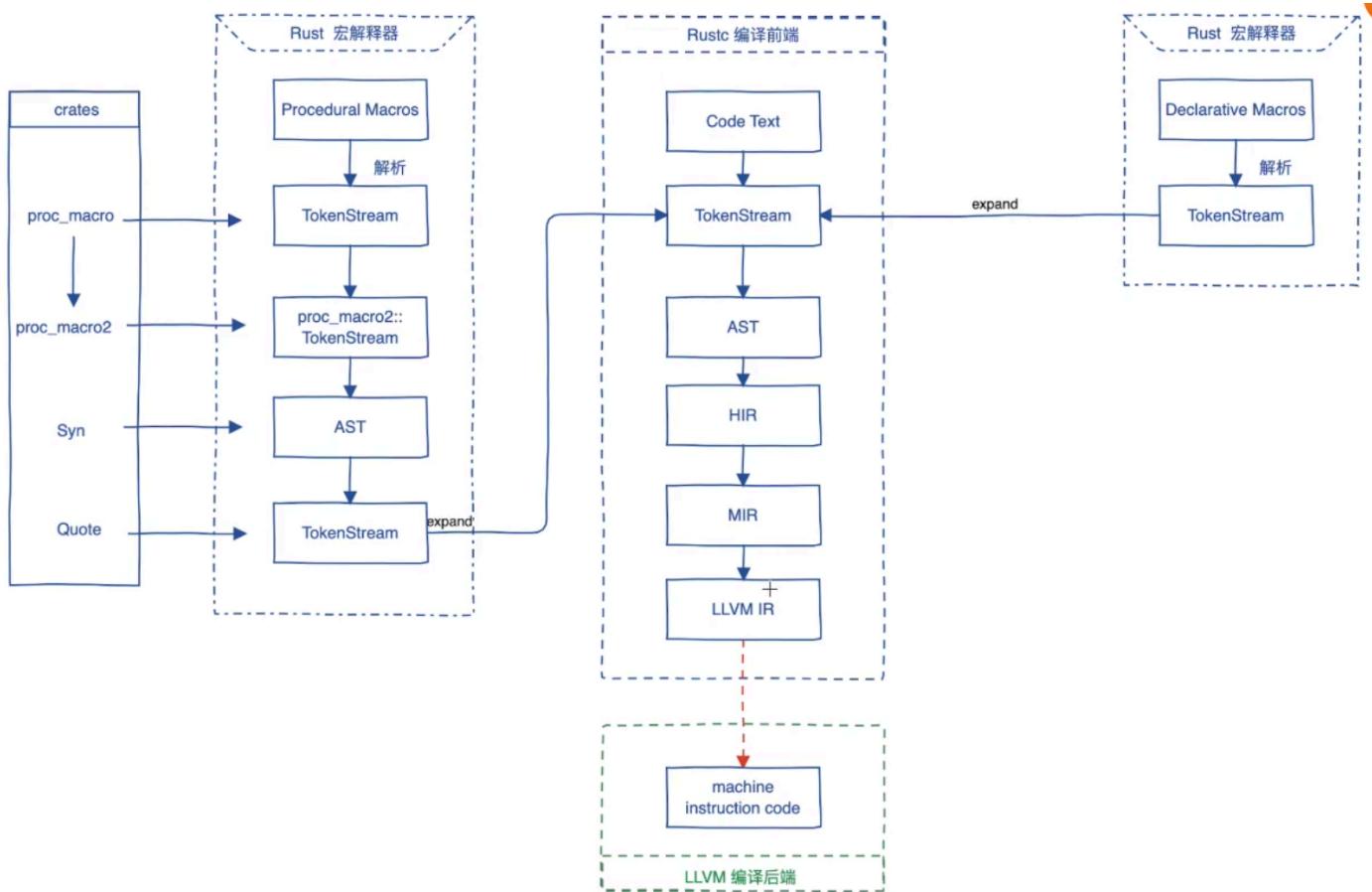
let foo1 = &bar;
let foo2 = &baz;

println!("{:?}", foo1);
println!("{:?}", foo2);

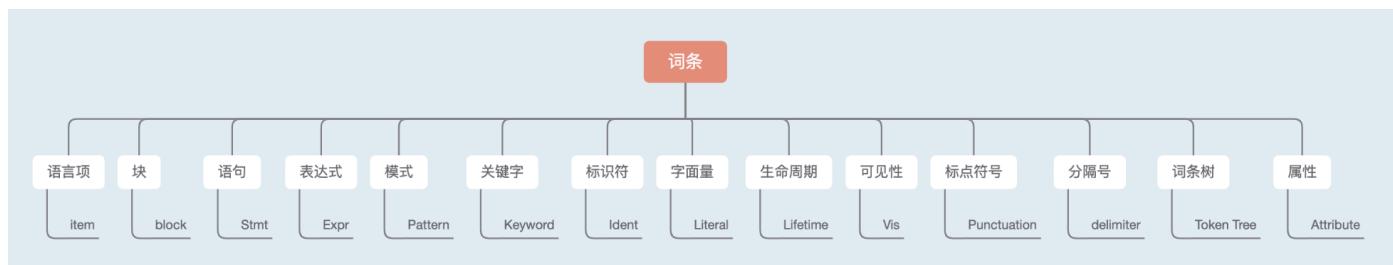
```

## 2.8.2 宏

宏是代码生成的一种技术，在此之前需要先理解rust编译过程



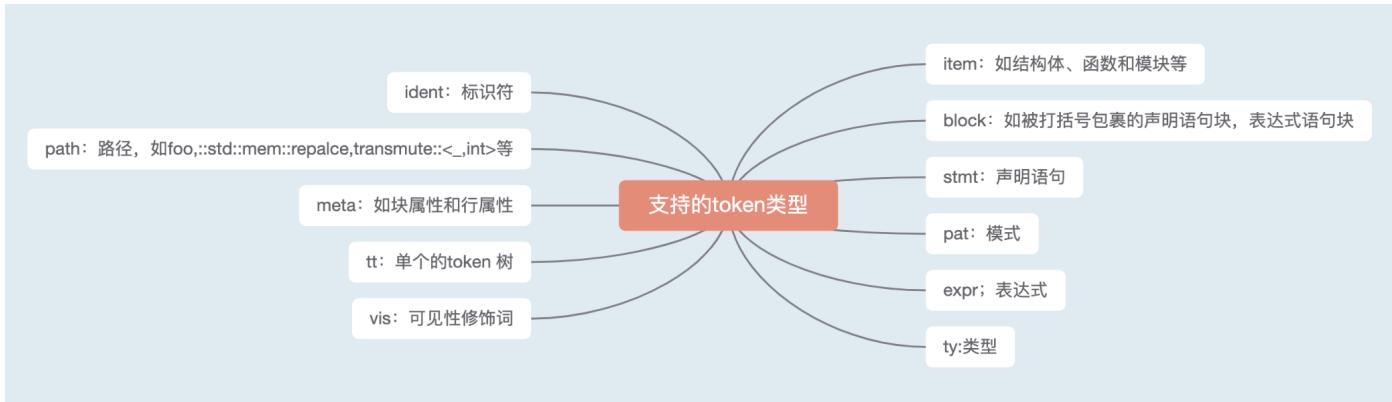
元编程也叫DSL，Domain Special Language



### 2.8.2.1 声明宏

把宏展开为TokenStream。只做替换几乎不做计算。如果是Token匹配，就是声明宏。

声明宏也是在分词阶段进行正则表达式的一种匹配



```
// 声明宏: 传入两个表达式, 正则匹配表达式
// 入参
// 替换
macro_rules! unless {
 ($arg:expr => $branch:expr) => {
 // 自定义自己的语言
 if !$arg {
 $branch
 }
 };
}

fn cmp(a: i32, b: i32) {
 unless!(a > b => println!("{} < {}", a, b))
}

let (a, b) = (1, 3);
cmp(a, b)
```

## 2.8.2.2 过程宏

### 1. derive 宏

更加复杂, 在Tokenstream上又构建了自己的AST, 为了更强大的计算

以serde库为例

过程宏有三种: 一种是类似于声明宏那样的函数调用, 第二种: 派生宏, 第三种: 属性宏

派生宏原理: 把结构体解析为词条流, 使用宏派生宏陪里面专门定义的词条处理的方法, 然后结合自定义的AST来处理

```
#[derive(Serialize, Deserialize)]
#[serde(deny_unknown_fields)]
struct S {
 #[serde(default)]
 f: i32,
}
```

如何实现过程宏？离不开过程宏三件套：syn(ast) quote(ast转为词条流) proc-macro2

proc-macro2 库：使用仅限于过程宏

syn（依赖于proc-macro2）和quote：配合使用，syn是把proc-macro2的TokenStream 转为AST，quote是再转回去。相当于这两个库配合又再加了一层

syn提供了一些数据结构：其实是语法树

过程宏的实例：Bang宏实现原理

宏一般独立一个crate，几乎可以做任何事情》宏代码调试工具：darling，可以在宏代码打log，cargo expand展开查看错误

第三方有哪些好用的宏代码

Derive-new 和 derive-more

过程宏的逻辑：解析->匹配模版->组装模版->输出为TokenStream

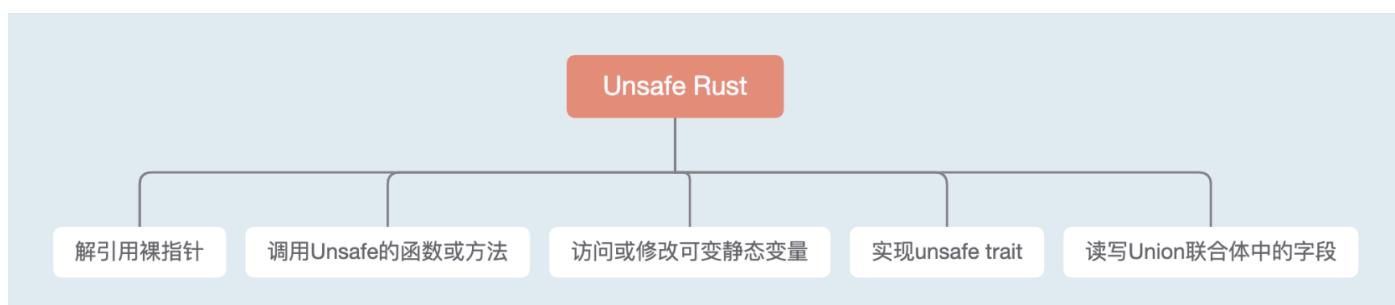
## 2. 属性宏

语法相对来说更加自由：案例：log-derive;rocket

## 3.9 Unsafe Rust

是Rust的超集，Unsafe rust也是有安全检查的

以下几种情况Rust不会提供任何安全检查



解引用裸指针：\*const T 和 \*mut T 两种指针类型，因为其和C语言中的指针十分相近，所以叫原生指针

原生指针的特点：

1. 不保证指向合法内存，如空指针
2. 不能像智能指针那样，自动清理内存
3. 没有生命周期的概念，编译器不会对其进行借用检查
4. 不能保证线程安全

```
// 解引用静态变量
static mut COUNTER: u32 = 0;
let inc = 3;

unsafe {
 COUNTER += inc;
 println!("Counter: {}", COUNTER);
}
```

safe rust构建于unsafe rust之上，凭什么safe?

官方保证：1. unsafe 在调用时注明安全边界；2. 实现了形式化验证；3. 安全数据库

### 3.9.1 安全抽象

从指针到引用。从不安全抽象为安全

### 3.9.2 Drop检查

### 3.9.3 型变

协变和逆变

### 3.9.4 NonNull和

NonNull 协变

## 4 异步编程



## 4.1 同步I/O模型

### 4.1.1 同步和异步

关注的是消息通信机制（调用者视角）

同步：发出一个调用，在没有得到结果之前不返回

异步：发出一个调用，在没有得到结果之前返回

## 4.1.2 阻塞和非阻塞

关注的是程序等待调用结果的状态（被调用者的视角）

阻塞：在调用结果返回之前，线程被挂起

非阻塞：在调用结果之前，线程不会被挂起

阻塞和系统调用有关

## 4.1.3 同步阻塞

数据输入阶段可以分为两段：数据准备和数据拷贝（数据从网卡接收，应用程序想从内核中读取数据（通过系统系统调用看数据有没有准备好））

等待数据准备好的阶段：可以是阻塞的和非阻塞的（轮询数据是否准备好）

数据拷贝阶段：把数据从内核缓冲区拷贝至应用程序缓冲区（用户态缓冲区），同步I/O下永远阻塞

现在常用的是IO多路复用：同步：数据等待和数据拷贝，第二阶段永远阻塞。多路复用也是一种同步IO模型。实现一个线程监视多个文件句柄

## 4.1.4 同步I/O 和异步I/O之别

异步I/O模型会把数据的准备和拷贝过程看作一个整体，整个过程都由内核来完成，不存在阻塞和非阻塞之说，它关注什么时候完成

## 4.1.5 I/O 多路复用

它是一种不同I/O模型，实现一个线程可以监视多个文件句柄

支持I/O多路复用的系统调用有select/pselect/poll/epoll。本质都是同步I/O，因为数据拷贝都是阻塞的，通过select/epoll来判断数据是否准备好，即判断可读可写状态

## 4.2 异步I/O模型

Rust编程模型下的异步包括同步I/O（应用进程不参与数据的拷贝，拷贝工作由内核完成）和异步I/O（特指linux）

异步非阻塞框架都是基于epoll

实际上就是一个I/O多路复用，但是可以设置为非阻塞，即在数据准备阶段可以是非阻塞的

## 4.2 epoll 和 io\_uring

### 4.2.1 epoll

是一个同步的多路复用，实际上是一种事件通知机制，具体包括：

三个函数：

1. epoll\_create, 内核产生一个epoll实例数据结构，并返回一个epfd
2. epoll\_ctl: 将被监听的描述符添加到红黑树或者从红黑树中删除或者对监听事件进行修改（epoll\_ctl内部（内核缓存区）提供的红黑树可以支持百万并发连接，添加删除非常快，可以用它来管理socket）

3. epoll\_wait: 阻塞等待注册的事件发生，返回事件的数目，并将触发事件的数目写到events数组之中（通过双向链表）

两种触发机制：

1. 水平触发机制：缓冲区只要有数据就触发读写，epoll默认工作方式。select/poll只支持该方式
2. 边缘触发机制：缓冲区空或者满的状态才触发读写，nginx使用该方式，避免频繁重复读写

如何解决惊群问题：

当多个进程/线程调用epoll\_wait时会阻塞等待，当内核触发可读写事件，所有进程/线程都会响应，但实际上只有一个进程才处理这些事件。Linux 4.5 通过引入EPOLLEXCLUSIVE标识来保证一个事件发生时只有一个线程会被唤醒，以避免惊群问题。

## 4.2.2 io\_uring

io\_uring是真正的异步I/O模型

原理：用户态和内核共享两个环形缓存区，一个是提交队列，另外一个是完成队列。省了系统调用。已经实现了零拷贝，两个阶段都是异步（无阻塞状态，进程发起数据准备调用后就可以做其他事情，直到数据准备好）。rust也支持，但是用的最多的还是epoll

## 4.3 事件驱动编程模型

处理IO复用的编程模型相当复杂，为了简化编程，提出了反应器模式和主动器模式

Reactor模式：应对同步I/O，被动的事件分离和分发模型。服务等待请求事件的到来，再通过不受阶段的同步处理事件，从而做出反应

Preactor模式：对应异步I/O，主动的事件分离和分发模型。允许多个任务并发执行，吞吐量很高；并可执行耗时长的任务（任务间不受影响）

三种实现方式

1. 单线程模式：accept()、read()、write()以及connect()都在同一线程
2. 工作者线程池模式：非I/O操作就交给线程池处理
3. 多线程模式：主Reactor (master) 负责网络监听，子Reactor (worker) 读写网络数据

读写操作流程

1. 应用注册读写就绪事件和相关联的事件处理器
2. 事件分离器等待事件发生
3. 当发生读写就绪事件，事件分离器调用已注册的事件处理器
4. 事件处理器执行读写操作

参与者

1. 描述符：操作系统提供的资源，识别socket等
2. 同步事件多路分离器：开启事件循环，等待事件发生，封装了多路复用函数select/poll/epoll等
3. 事件处理器，提供了回调函数，用于描述与应用程序相关的某个事件的操作
4. 具体的事件处理器，事件处理器接口的具体实现，使用描述符来识别事件和程序提供的服务
5. Reactor管理器，事件处理器的调度核心，分离每个事件，调度事件管理器，调用具体的函数处理某个事件

## 4.4 epoll代码实践

---

使用三个系统调用函数。安卓，Linux都用。使用Reactor分发处理。epoll只支持linux

## 4.5 Reactor 代码实践

---

事件驱动编程模型。读写，注册事件

## 4.6 MiniMio代码实践

---

跨平台抽象，mio库

Linux和win有不同的系统抽象。抽象一个selector去选择不同的平台

## 4.7 Mio代码实践

---

### 4.7.1 epoll接口

它是一个生产环境下的库

udp: poll::new 系统调用；轮询；建立UDP链接；处理等

### 4.7.2 其他代码

Waker;唤醒

io\_source 实现了Source trait

对不同的平台底层进行抽象

## 4.8 异步编程模型

---

### 4.8.1 与其他语言相比

1. Rust只提供零成本的异步编程抽象而不内置运行时，运行时可以替换如tokio
2. 基于Generator实现的Future，在future的基础上提供async/await语法糖，本质是一个状态机
3. Node.js依赖于V8运行时，其async/await建立在Promise抽象机制（范式）上，Go内置了运行时，提供了协程
4. Rust只提供了async/await 以及Future（基于语言层面），运行时在语言之外，可以根据不同的场景更换运行时

### 4.8.2 为什么需要异步

1. 对极致性能的追求
2. 对编程体验的追求

## 4.8.3 异步编程模型的发展阶段

1. callback (回掉地狱)
2. Promise/Future (会产生很多内嵌Future)
3. async/await: 拥有了和同步代码的一致体验

## 4.8.4 如何理解异步任务

异步任务可以看作是一种绿色线程

异步任务的行为是模仿线程来抽象

1. 线程在进程内，进异步任务在线程内
2. 线程可以被调度切换（Linux默认抢占），异步任务也可以被调度（协作式而非抢占式）。区别在于，异步任务只在用户态没有线程的上下文切换开销
3. 线程和异步任务都有上下文信息
4. 线程和异步任务之间都可以通信
5. 线程和异步任务之间都会有竞争

整个异步编程的概念，包括异步语法、异步运行时都是围绕如何建立这种[绿色线程]抽象而成的

## 4.8.5 Future

Future代表一个异步计算，就像Option一样。在很多支持异步的语言中，Promise也叫Future / Delay / Deferred等。

```
pub trait Future {
 type Output;

 fn poll(self: Pin<&mut Self>, cx: &mut Context<'_>) -> Poll<Self::Output>;
}
```

```
pub enum Poll<T> {
 Ready(T),
 Pending,
}
```

通过poll方法获得值有没有准备好。而std::task就是最终要创建的绿色线程，调度器要自己实现

Future是惰性求值，需要创建异步运行时计算。通过Trait Wake实现这种唤醒机制

```
pub trait Wake {
 fn wake(self: Arc<Self>);

 fn wake_by_ref(self: &Arc<Self>) { ... }
}
```

Future以及上述trait还有async和await是rust提供的最小化的定义，用于异步编程。future-rs实现了更完整的异步运行时

async 定义了一个可以并发执行的任务，而 await 则触发这个任务并发执行。大多数语言中，async/await 是一个语法糖（syntactic sugar），它使用状态机将 Promise 包装起来，让异步调用的使用感觉和同步调用非常类似，也让代码更容易阅读。

## 4.8.2 编写异步echo服务

1. 建立tcp链接
2. 处理tcp 流：read /write
3. poll/select epoll

## 4.8.3 异步Task模型

调度线程中的协程就是运行时。

## 4.8.4 Waker实现

一个task可以看作是一个线程中的微线程

## 4.9 异步库源码导读

异步运行时的实现机制：Future channel 是task之间通讯

Pin异步运行时中相当于一个模版

Future 的流相当于异步迭代器

Future task

## 4.10 async/await 语法

async的两种用法：`async fn` 和 `async {}`

Await 将暂停函数执行。如果用锁的话尽量使用Future提供的锁

```
use std::future::Future;
// async 真正会返回 Future<Output = u8>, 而不是看上去的u8
async fn foo() -> u8 {
 5 // 去糖后是Future
}

// async 块用法, 返回 "impl Future<Output = u8>"
fn bar() -> impl Future<Output = u8> {
 async { // 块返回值
 let x = foo().await;
 x + 5
 }
}
```

async 生命周期

```
async fn foo (x:&u8) -> u8 {*x}
```

等价于

```
fn foo_expanded<'a>(x:&'a u8) -> impl Future<Output =u8> + 'a {
 async move {*x}
}
```

```
fn bad() -> impl Future<Output = u8> {
 let x = 5;
 borrow_x(&x) // 无法编译通过
}
```

```
fn good() -> impl Future<Output = u8> {
 async {
 let x = 5;
 borrow_x(&x) // 可以编译通过
 }
}
```

多个await以及move还有join

在多线程执行器中使用 await时，尽量使用futures::lock提供的锁，而不是标准库提供的锁，以避免死锁

async/await 本身是个语法糖吗，解糖以后是个future，一切都是围绕future来进行的

## 4.10.1 生成器

async / await 对应底层生成器为 resume/yield。yield是暂停点。和闭包的区别在于能暂停，底层实际上是一个状态机。和闭包底层也非常相似。

```
let mut gen = || {
 yield 1;
 yield 2;
 yield 3;
 return 4;
}

let c = Pin::new(&mut gen).resume(());
println!("{:?}", c);

let c = Pin::new(&mut gen).resume(());
println!("{:?}", c);

let c = Pin::new(&mut gen).resume(());
println!("{:?}", c);
```

可以把生成器当做迭代器用，高阶用法

生成器本质上是一个状态机，与future的相比：Generator可以变为Future

## 4.10.2 Pin与Unpin

Rust解决自引用，异步传递引用的安全性

是一种使用类型系统的解决方案。Pin防止得到可变借用乱用

## 4.11 no\_std异步生态

核心库一般是使用在wasm和嵌入式，这些场景一般没有堆分配。所以关于堆分配的一些集合找不到

运行时

1. async-std 异步的，专门处理异步io
2. tokio 最成熟的，生产级应用比较多
3. smol-rs 轻量的运行时 封装了很多底层的库
4. gloomio
5. bastion 目标是高可用的

## 4.12 实现异步缓存

Rust中的B树命中率更高。异步过程中构建组件等也是异步的。

多线程或者异步使用B树需要加锁,smol 实现了一些锁和屏障.B树和HashMap使用一样的。

区分同步和异步代码

如何清理过期缓存？redis：按照频率，定期删除策略

# 5 Rust异步Web框架

## 5.1 Rocket

充分的利用了

# 6 知名Rust项目

代码组织方式：Rust推荐整个项目使用多个crate构建

## 6.1 Rust

主要是编译器的实现

## 6.2 Wasmtime

字节码联盟维护的一个JIT的WebAssembly运行时，使用的编译器是Cranelift

## 6.3 Futures-rs

官方提供的一个运行时实现

## 6.4 async-std

标准库对async的实现

## 6.5 Tokio

比较成熟的异步运行时

## 6.6 Rocket

Web框架

## 6.7 Actix-web

Web框架

## 6.8 TiKV

数据库

# 7 常用crates

## 7.1 标准库

```
std::fs; // 处理文件
```

## 7.2 第三方库

```
html2md = "0.2.14" // 将文本转换为markdown
reqwest = {version ="0.11.14", features = ["blocking"]} // 一个http客户端, 类似于python中的request
reqwest = {version ="0.11.14", features =["json"] }
anyhow = "1.0.69" # 错误处理
clap = "4.1.6" # 命令行解析库
colored = "2.0.0" #命令终端多色彩显示
jsonxf = "1.1.1" # JSON pretty print 格式化
mime = "0.3.16" # 处理mime类型
tokio = "1.25.0" # 异步处理库
tokio = { version = "1.25.0", features = ["full"] } # 异步处理
axum = "0.6.7" # web服务器
base64 = "0.21.0" # 编解码
bytes = "1.4.0" # 处理字节流
image = "0.24.5" # 处理图片
lazy_static = "1.4.0" # 通过宏更方便的初始化静态变量
lru = "0.9.0" # LRU 缓存
percent-encoding = "2.2.0" # url 编码解码
photon-rs = "0.3.2" # 图片效果
```

```

prost = "0.11.6" # protobuf 处理
serde = { version = "1.0.152", features = ["derive"] } # 序列化和反序列化数据
tower = { version = "0.4.13", features = ["util", "timeout", "load-shed", "limit"] } # 服务处理中间件
tower-http = { version = "0.3.5", features = ["add-extension", "compression-full"] } # http中间件
tracing = "0.1.37" # 日志和追踪
tracing-subscriber = "0.3.16" # 日志和追踪
async-prost = "0.2.1" # 支持把 protobuf 封装成 TCP frame
futures = "0.3" # 提供 Stream trait

```

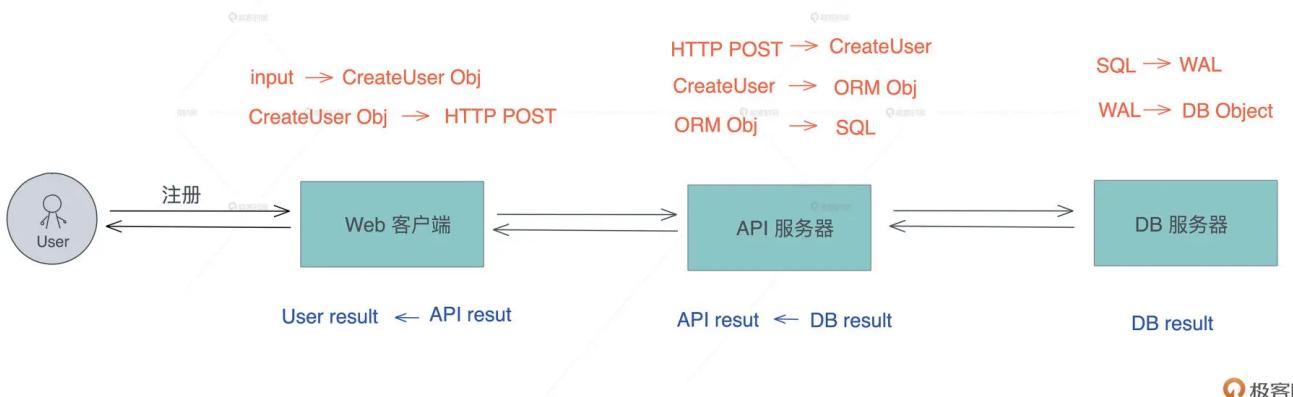
```

[build-dependencies]
prost-build = "0.11.6" # 编译 protobuf

```

## 8 Rust 实战

在开发项目时，核心之一就是做各种数据转换，也就是从一个API到另一个API，以一个前端到数据库的交互为案例，如下是整个过程：



Rust 标准库的 From/ TryFrom trait 即是服务于此目的，方便开发者编写出易于阅读、容易测试、维护简单的代码