1 Rust语言介绍

Rust语言其他的不多强调了，但要强调一点：性能、安全以及实用是Rust追求的根本目标

1.1 内存安全方案

1.1.1 Rust针对C语言的不足

禁止对空指针和悬垂指针解引用

读取未初始化的内存

缓冲区溢出

非法释放已经释放或未分配的指针

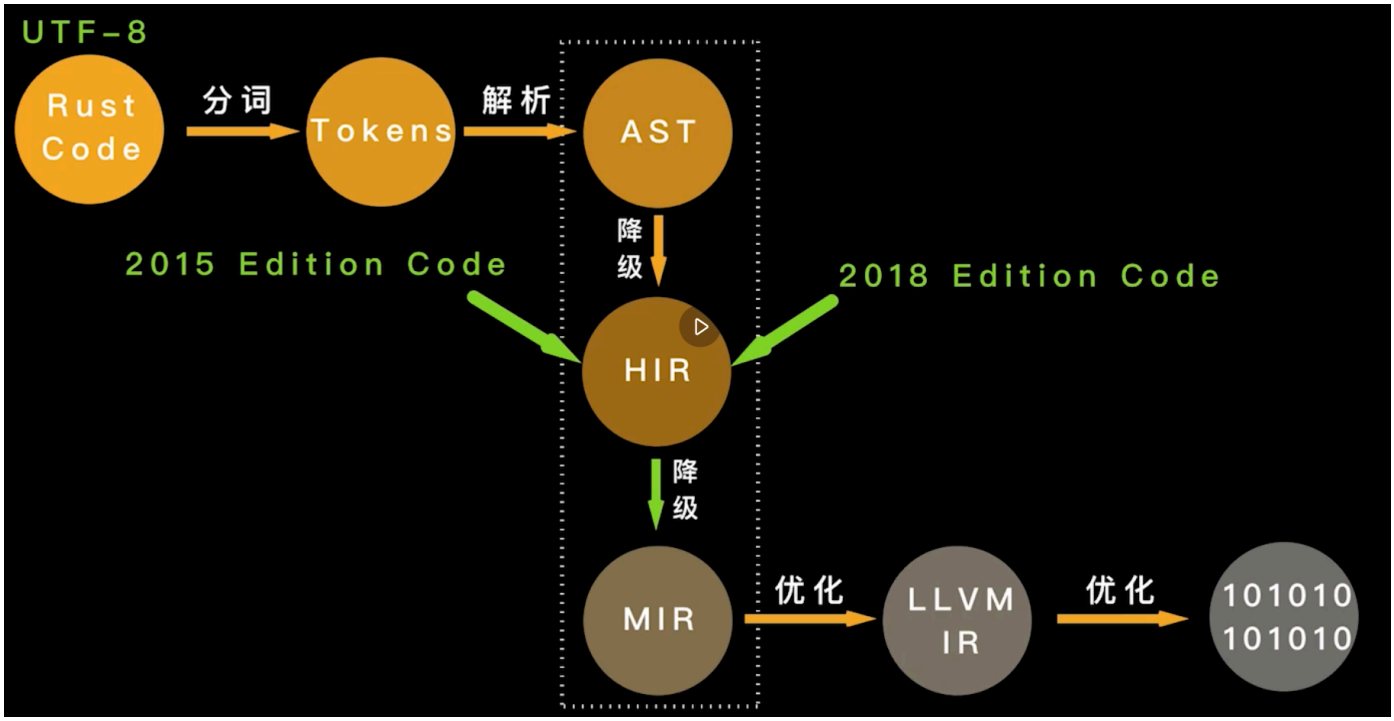
1.1.2 安全无缝的沟通C语言

通过C-ABI零成本和C语言打交道

划分了Safe Rust和Unsafe Rust

2 Rust语言核心原理及案例

2.1 Rust 编译过程



特别说明

大部分语言会将词条流解析到的抽象思维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，交由LLVM后端生成机器码

高级中间语言：类型检查、方法查找

中级中间语言：借用检查、优化、代码生成、泛型单态化等工作

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

关键字：严格关键字、保留字、弱关键字

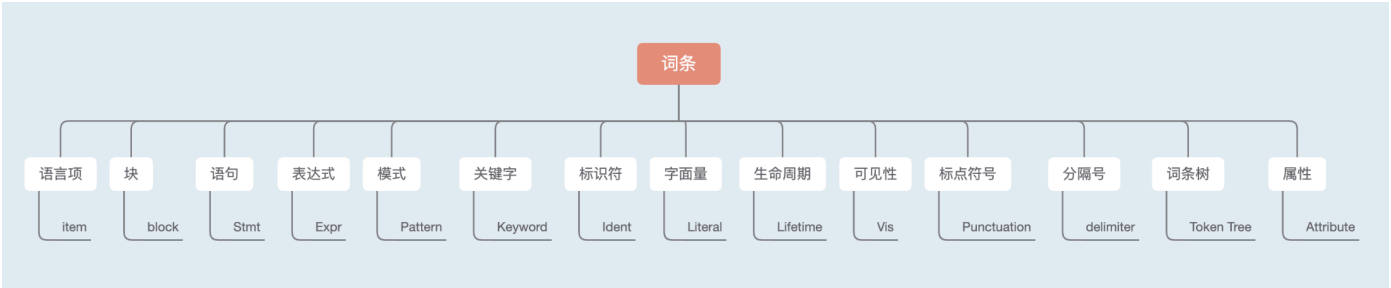
标识符：不以数字开头的ASCII字符注释

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

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

注释：Rust可以使用注释直接生成文档，非常友好

空白：空白不表示任何含义，如换行等

词条：词条在写宏的时候非常有用




```
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
///

// 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);
}
```

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

2.3 Rust 语法骨架

Rust语法骨架只包含三种元素

属性：行属性和块属性

分号：行分隔符

花括号：块分隔符

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

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

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：

```
; -> ()
```

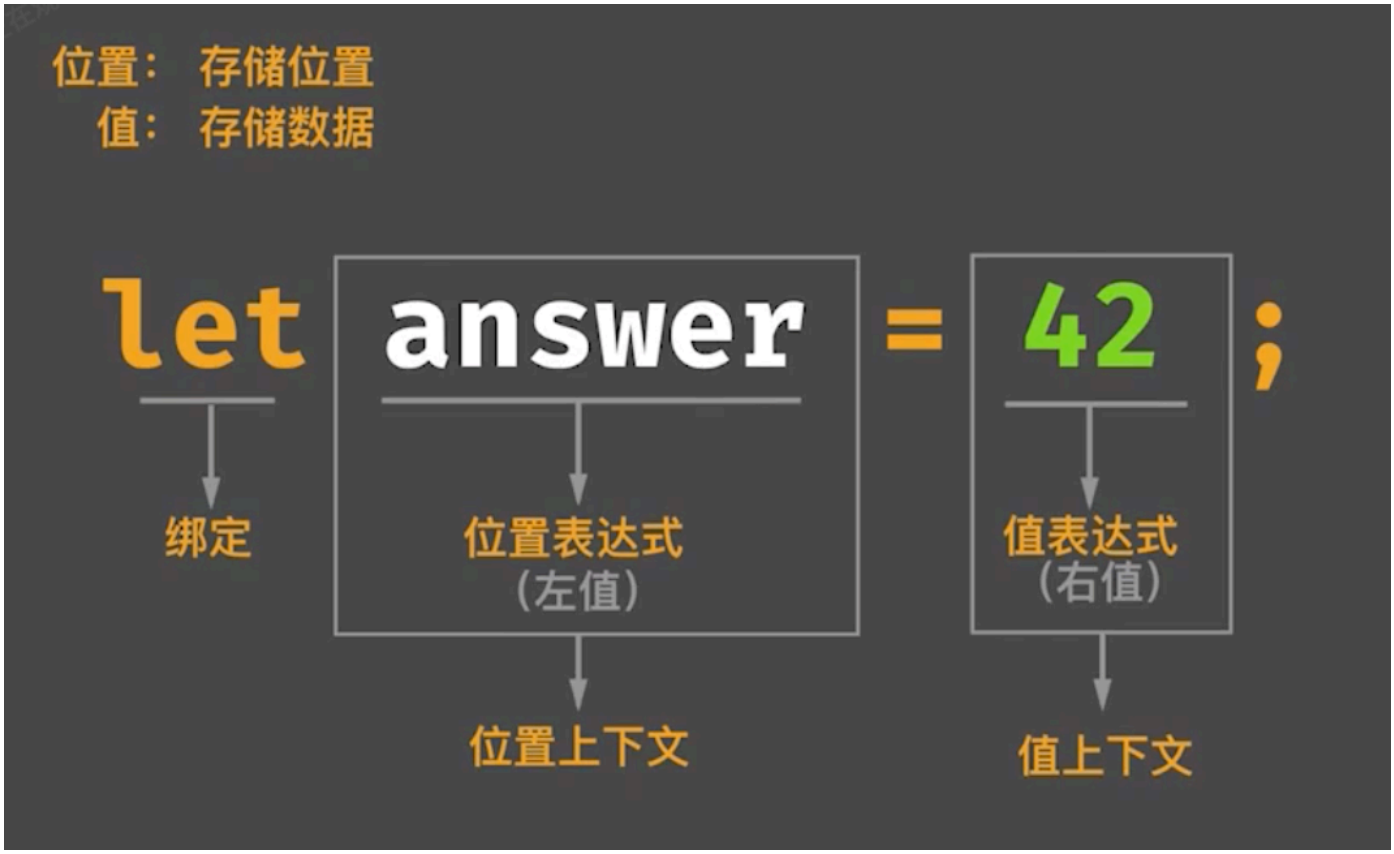
块表达式：值的类型是块中最后一个表达式的值。当块中最后一行为一个值时，块表达式的值为该值，类型是该值的类型。如：

```
{
  let s = "string"; -> ()
  s -> &str
}
```

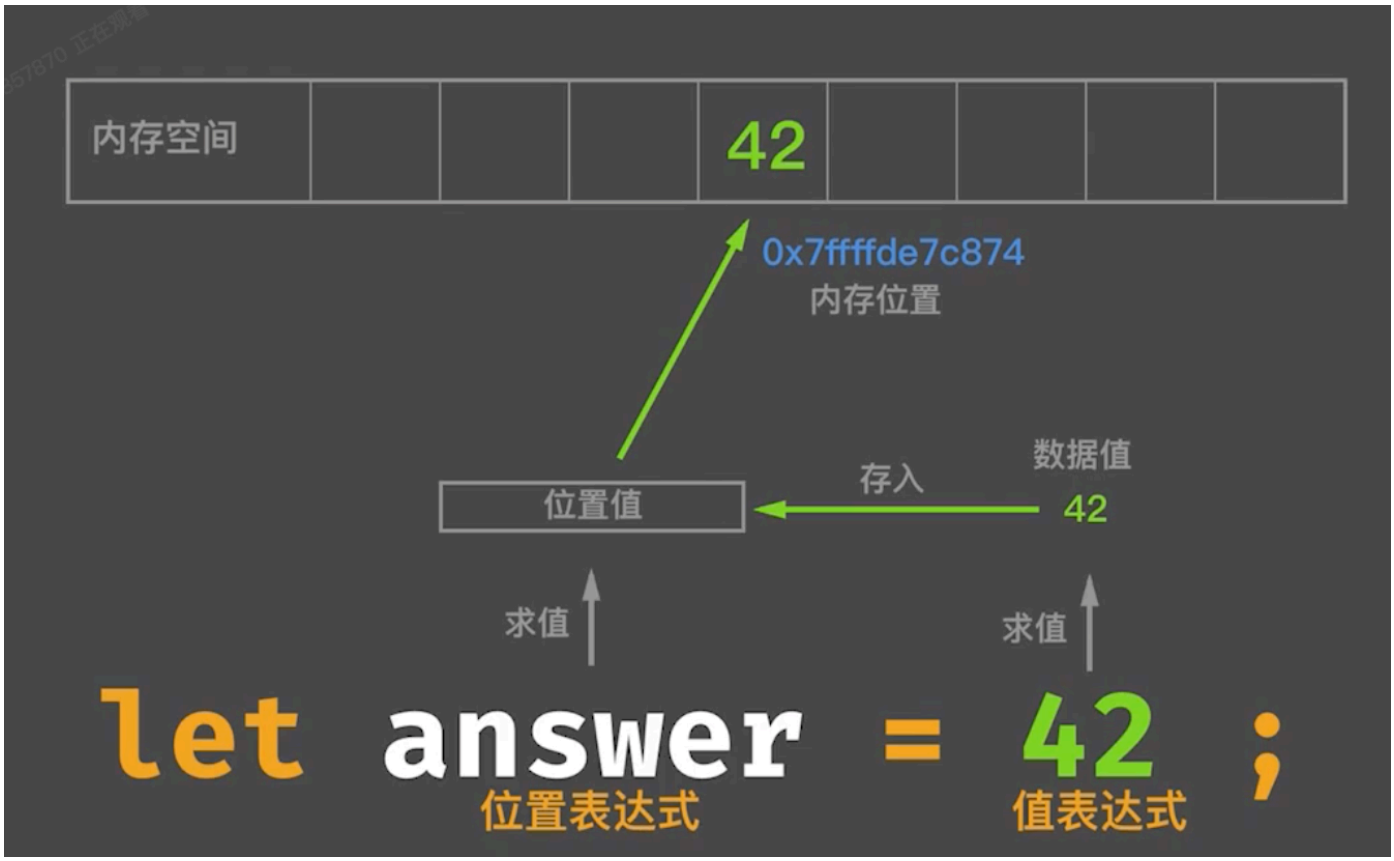
2.4.2 表达式分类：按内存管理

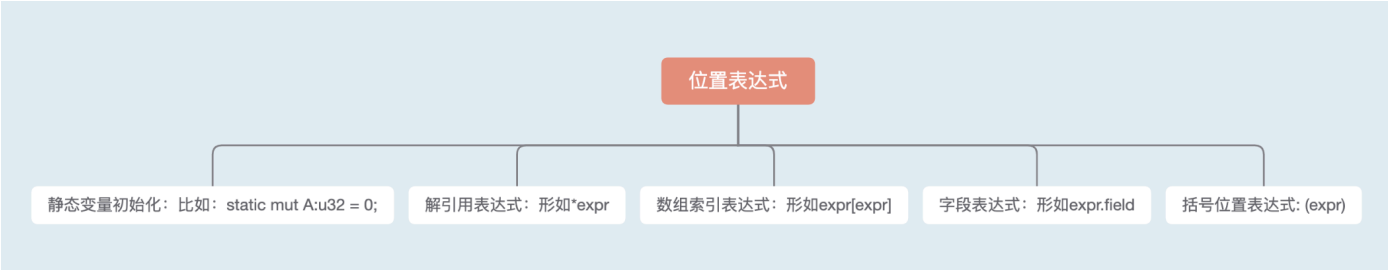
位置表达式：位置，存储位置，二者以等号为界

值表达式：值，存储数据


位置：存储位置
值：存储数据
let
answer
=
42
;
绑定
位置表达式
（左值）
值表达式
（右值）
位置上下文
值上下文

表达式背后的内存管理


内存空间
42
0x7ffffde7c874
内存位置
数据值
存入
位置值
42
求值
求值
let
answer
=
42
;
位置表达式
值表达式



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

```
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

不可变绑定与可变绑定

不可变引用（共享引用）与可变引用（独占引用）

Rust中与C语言一样的*mut T和 *const T 只能在Unsafe Rust中使用

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

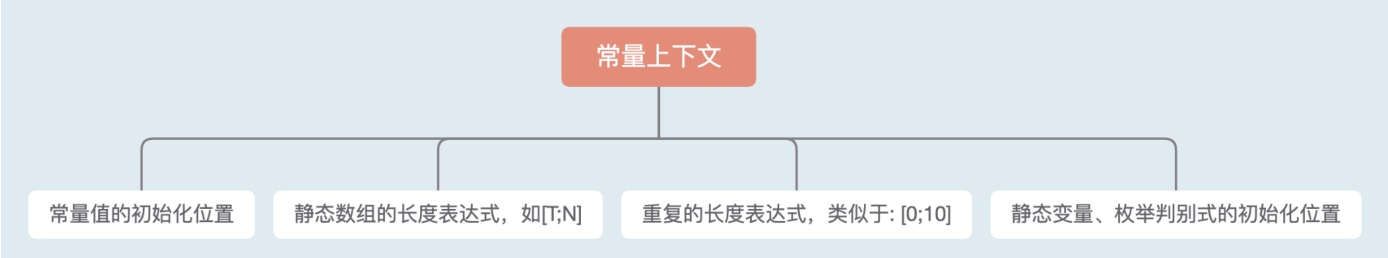
过程宏 + Build脚本（build.rs）：类型计算、生成代码等，但是无法在宏代码和普通代码之间共享代码

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
```



特别说明

常量上下文是编译器唯一进行常量求值的地方

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

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，目前支持常量函数，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
3. 编译期求值必须得到确定结果，当文发生变化时确定性就无法保证了

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

特别说明

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命名所有数组

为了解决这个问题需要使用核心库中的联合体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

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

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

fn main() {
    println!();

    let av = ArrayVec {
        items: [MaybeUninit::<u32>::uninit(); 3],
        length: 10,
    };

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

特别说明：常量泛型目前只支持

1. 一个简单的常量泛型参数，比如 `cosnt 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

保证内存安全

保证一致性

表达明确的语义

零成本抽象表达能力

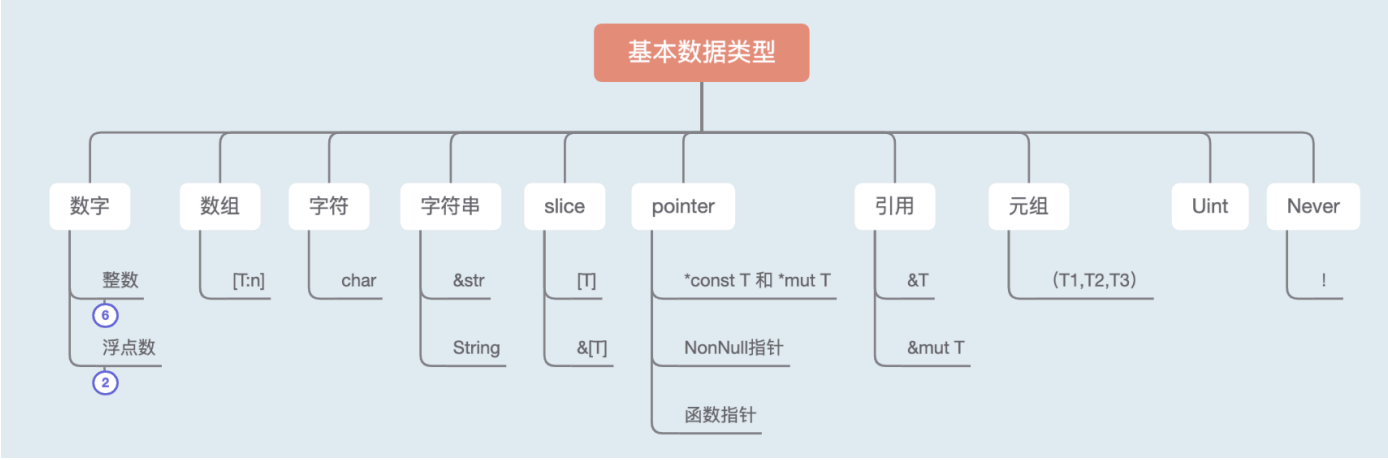
2.6.2 Rust如何实现目标

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]l 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

```
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 = 0b111010011100000011001011;

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

5. 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

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。可以在多线程种安全的使用

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

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

6 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

7 引用与指针之别

引用不为空

拥有生命周期

受借用检查器保护，不会发生悬垂指针等问题

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

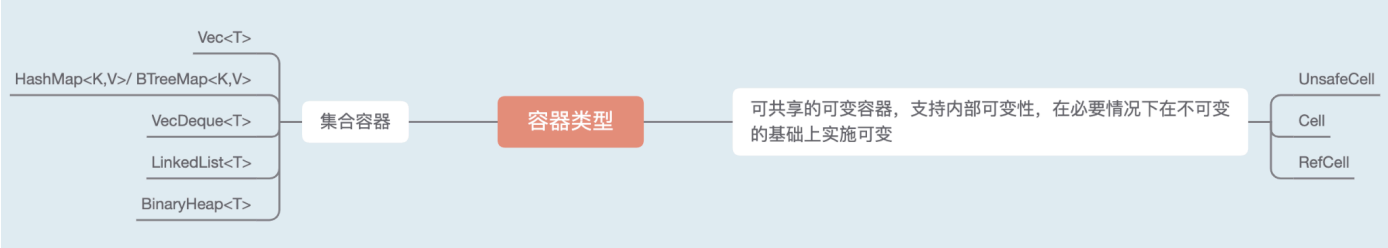
```
// 1.具名结构体
struct Point {
     x:f32,
     y:f32
}


// 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是，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 结构体内存对齐方式

```
// 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,
```

```
    c: u16,
    d: u8,
}

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

2.6.3.3 容器类型



共享容器

内部可变性：本质是把原始指针*mut 给开发者

1. 与继承式可变相对应
2. 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
3. 基于UnsafeCell,提供了Cell和RefCell

```
### 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
    }

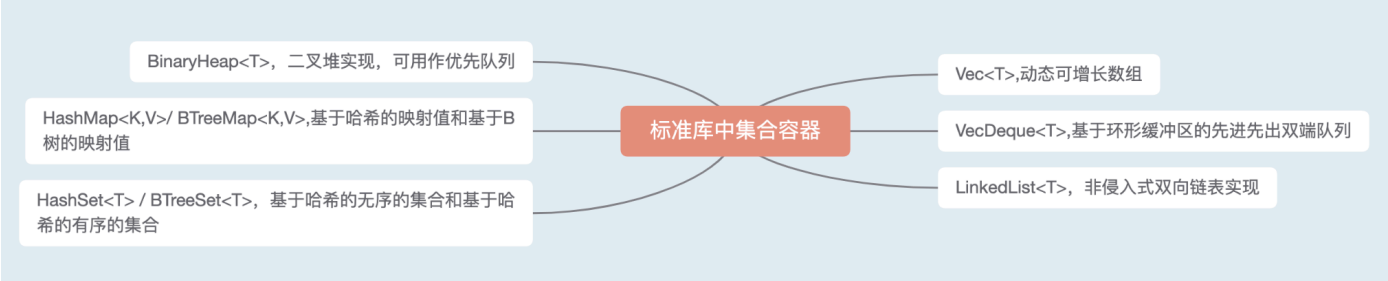
    // 初始化一个不可变实例
    let foo = Foo {
        x: 1,
        y: Cell::new(3),
        z: Cell::new("hello".to_string()),
    };

    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

集合容器



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的代码），也叫静态分发

```
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
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

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

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

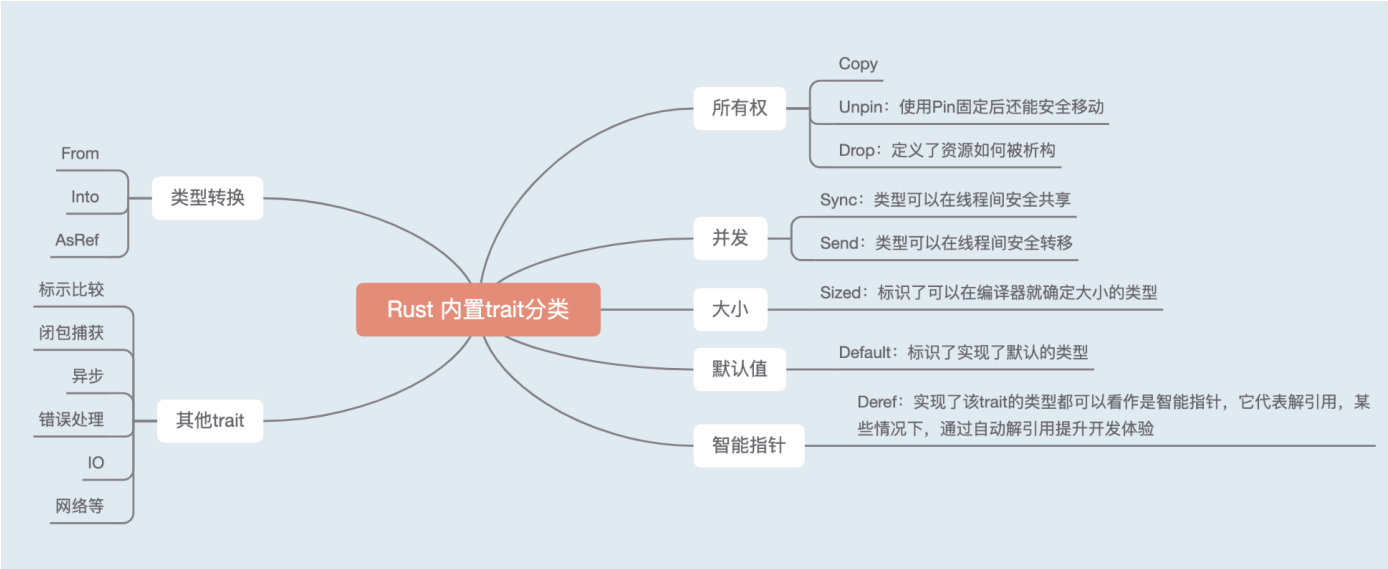
但是由于Copy trait的存在，凡是实现了Copy trait的类型，在发生上述行为时，所有权没有发生转移，而是为新的变量重新拷贝了一份数据（发生在栈上）

5. trait 理论来源

Rust类型系统遵循的时仿射类型理论，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

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

```
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))
```

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!("{:?}", 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

2.8.2.5 逃逸闭包和非逃逸闭包

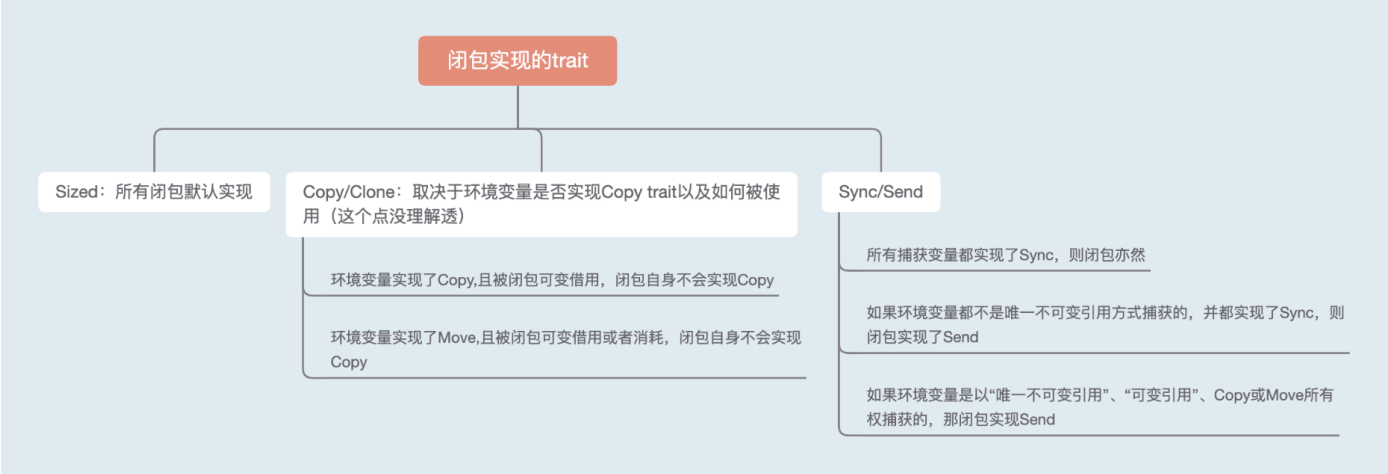
```
```
 // 逃逸闭包
    fn c_mut() -> impl FnMut(i32) -> [i32; 3] {
        let mut arr = [1, 2, 5];
        move |n| {
            arr[2] = n;
            arr
        }
    }

    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
```

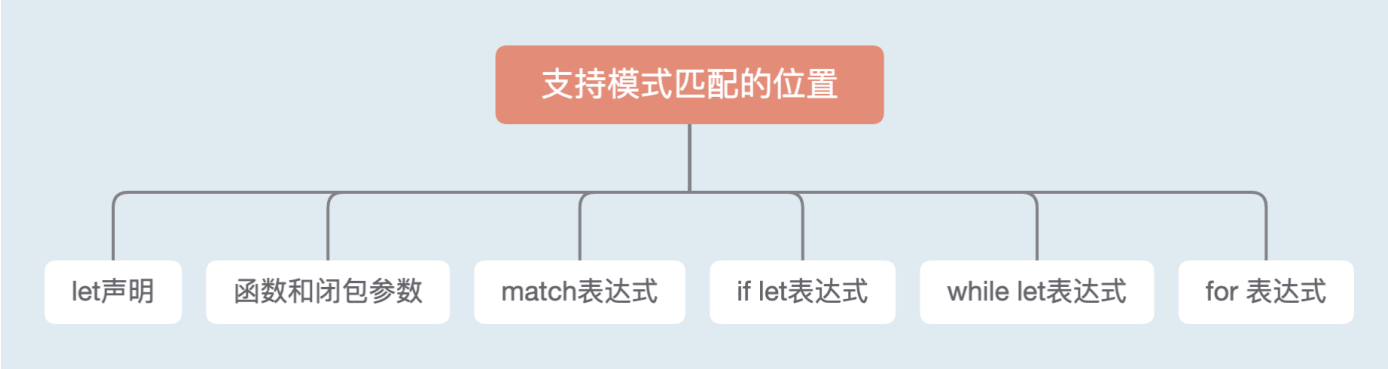



```
// 闭包自身实现了Fn Copy trait
    fn foo<F: Fn() + Copy>(f: F) {
        f()
    }

    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

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

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Saferust 中唯一的堆内存分配方式

```
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RALL,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
        }
    }
    */
```

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


```
     // 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
        }
    }
    */
```

3 Rust核心库

```
use core::mem::MaybeUninit;
```

4 Rust标准库

5 Rust第三方库

6 知名Rust项目