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

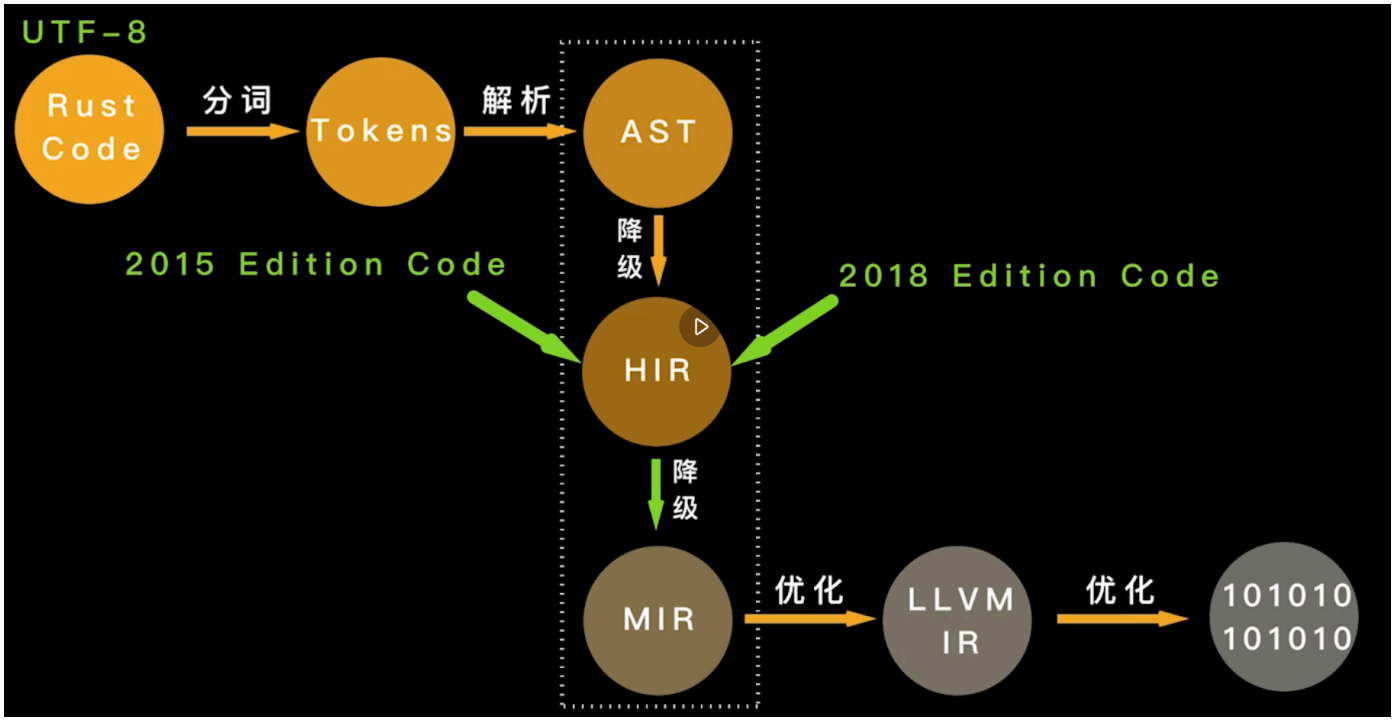
1.1.2 安全无缝的沟通C语言

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

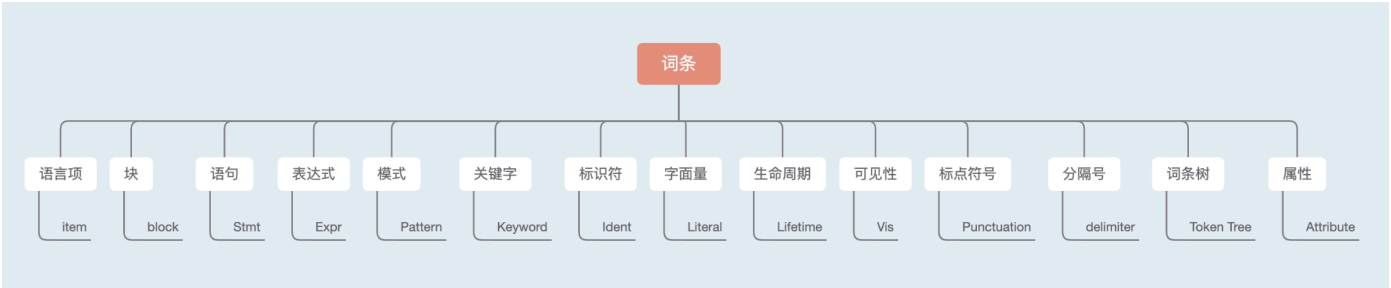
```
let name = "name";
let _100 = "number";
let math_grade = 150;

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：

```
; -> ()
```

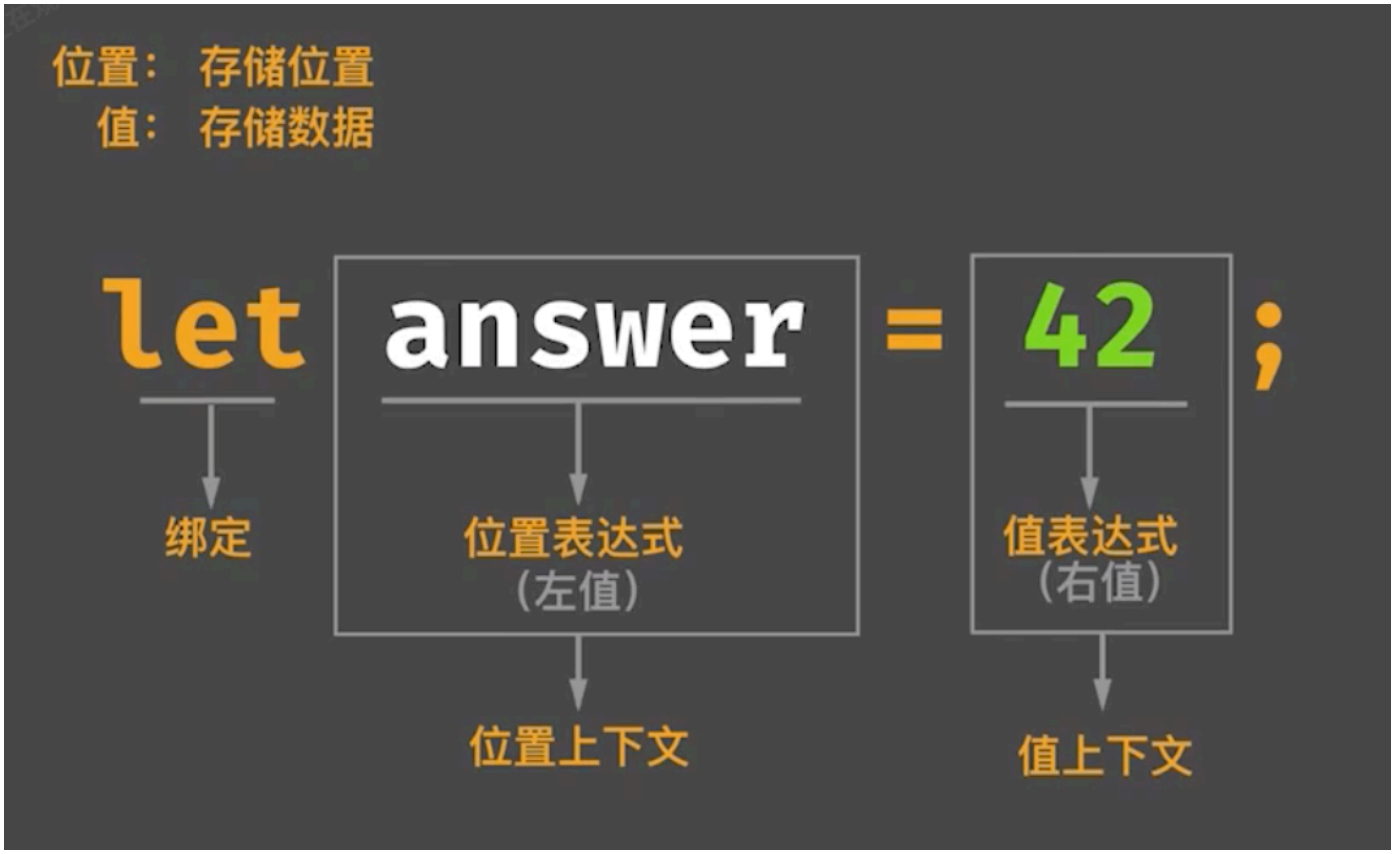
块表达式：值的类型是块中最后一个表达式的值。当块中最后一行为一个值时，块表达式的值为该值，类型是该值的类型。如：

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

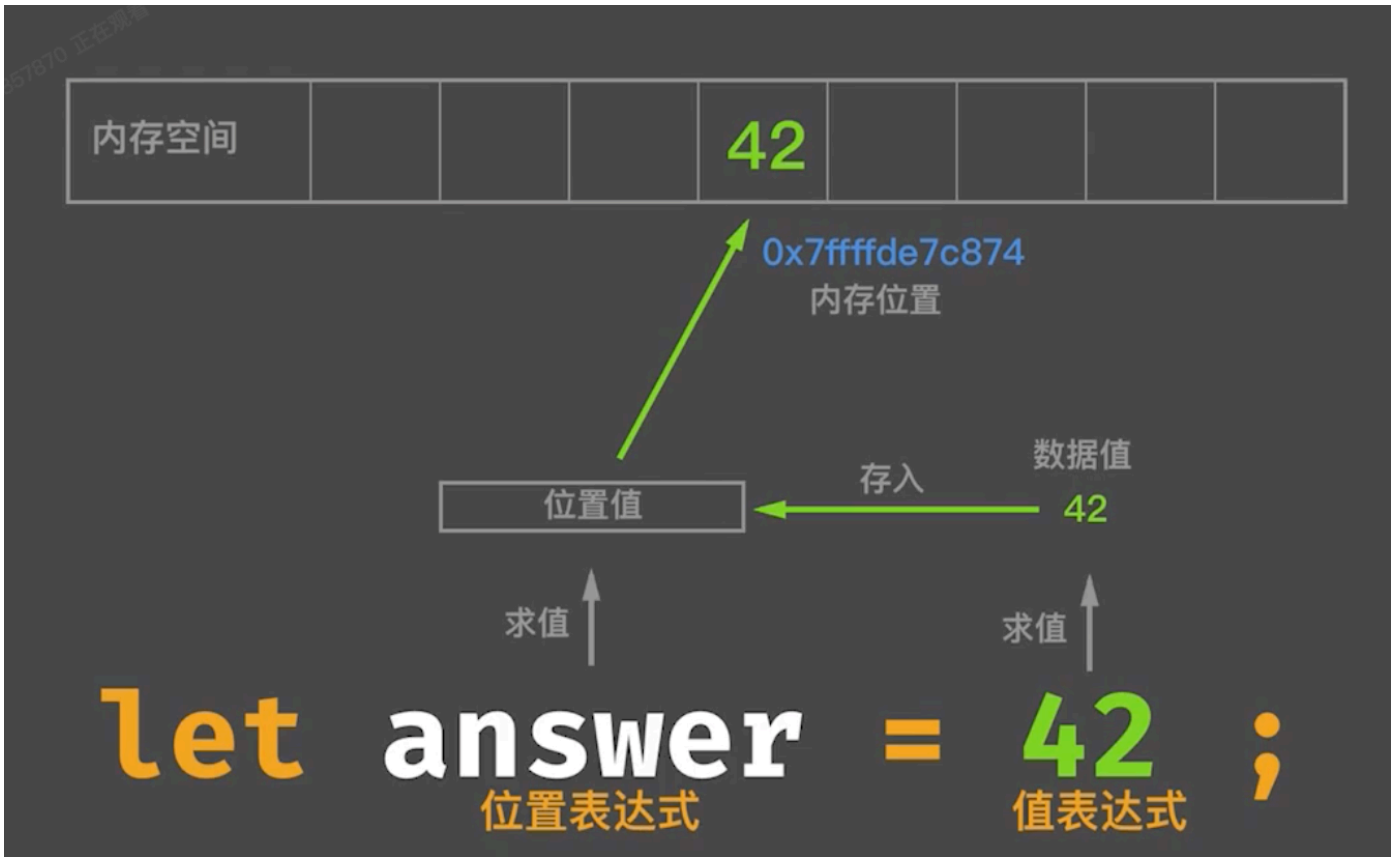
2.4.2 表达式分类：按内存管理

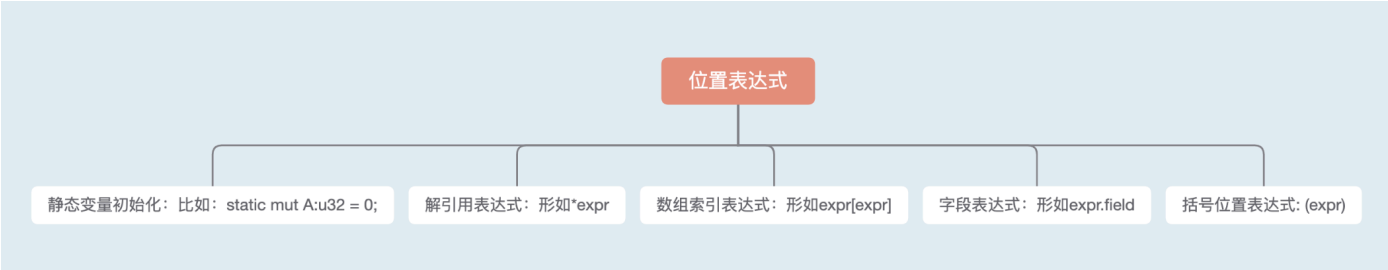
位置表达式：位置，存储位置，二者以等号为界

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

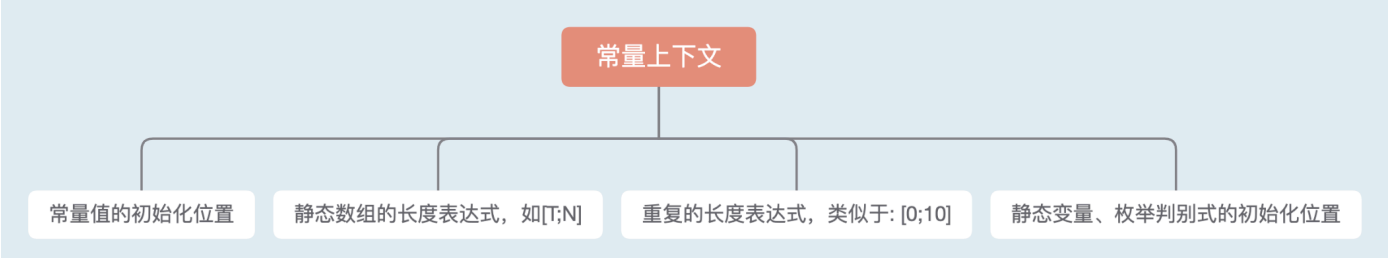
过程宏 + Build脚本（build.rs）：类型计算、生成代码等，但是无法在宏代码和普通代码之间共享代码

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

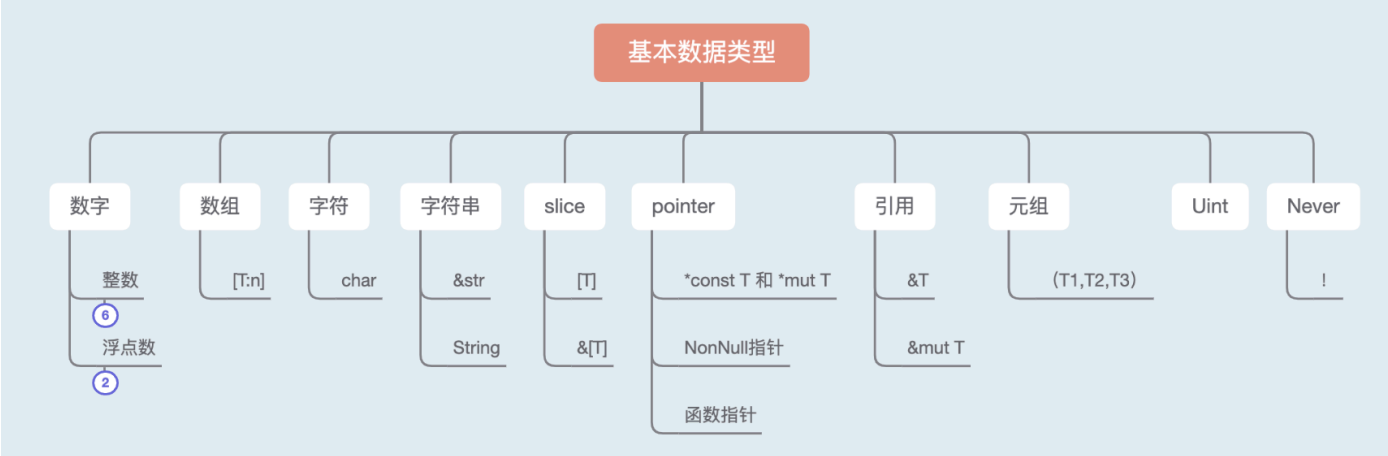
2.6.2 Rust如何实现目标

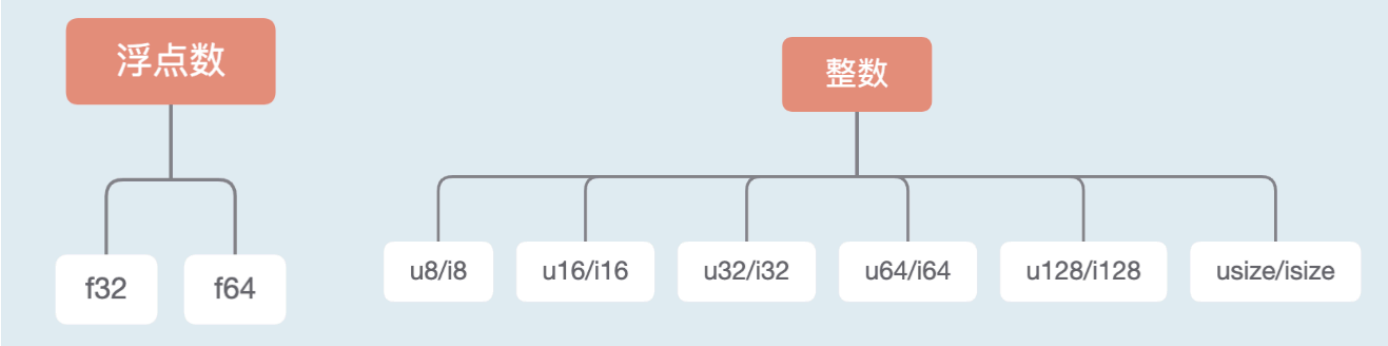
类型：在rust中，一切皆类型

trait：trait规范了类型的行为

2.6.3 Rust数据类型

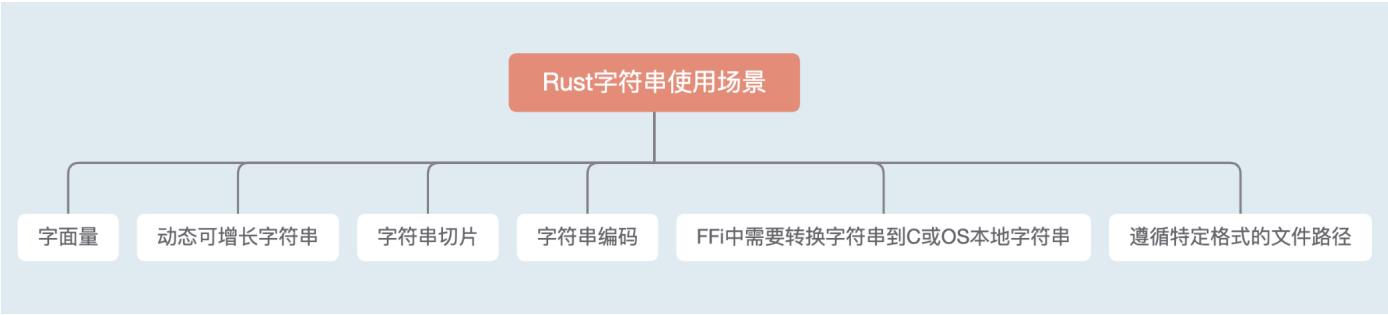
2.6.3.1 基本数据类型





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. rust中的char是unicode标量，占四个字节
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;
```

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>());
```

2.6.2.3 容器类型

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

2.6.2.4 泛型

2.6.2.5 特定类型

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