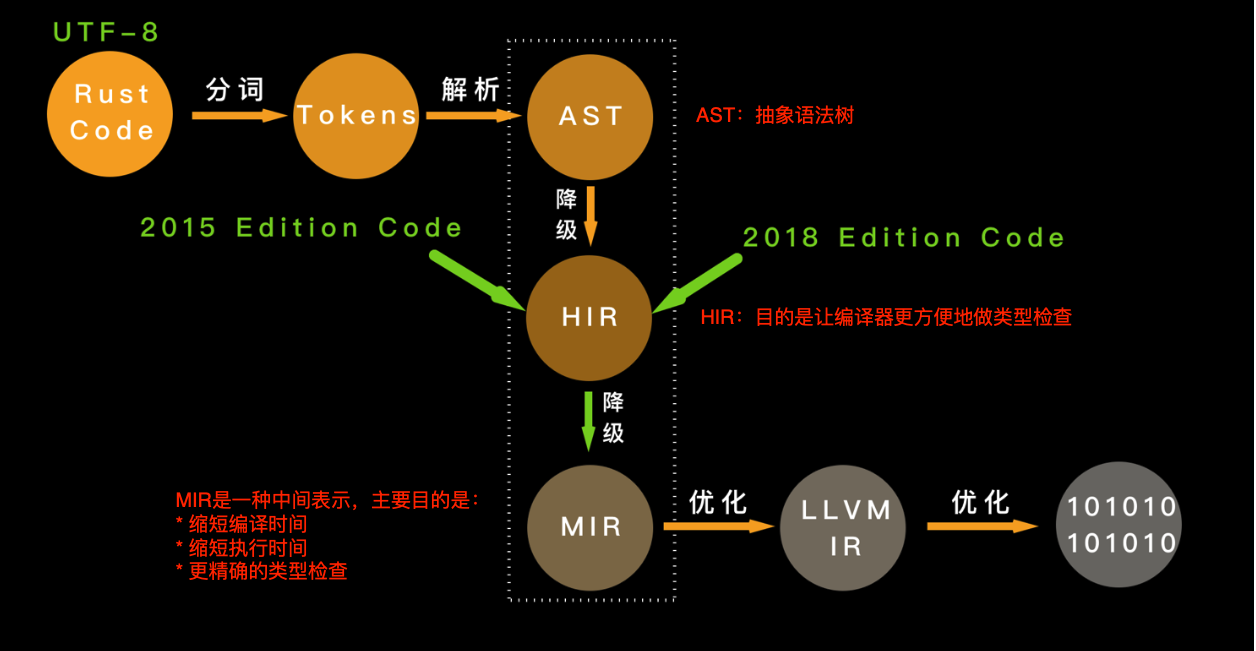
Rust笔记

Rust实战课

词法结构

Rust编译过程：



面向表达式的语言

* Rust:面向表达式的语言 / 求值规则

1. 分号表达式返回值是永远为自身的单元(Unit)类型:「()」。

2. 分号表达式只有在块表达式最后一行才会进行求值，其他时候只作为「连接符」存在。

3. 块表达式只对其最后一行表达式进行求值。

* 流程控制也是表达式
* 作业:

1. Rust 语法中还有很多种类的表达式，课堂里没有一一列举，你可以在课下把这些表达式都走查一遍。

2. 操作符都是表达式，它们都有优先级，你是否可以自行了解一下这些操作符表达式的优先级列表呢

Rust语言的编译期计算

* + - 编译期计算 / Rust 支持两种方式:

过程宏 + Build 脚本(build.rs)

类似于 Cpp 中 constexpr 的 CTFE 功能

* + - 编译期计算 / Rust 的 CTFE

常量函数(const fn)

常量泛型(const generic)

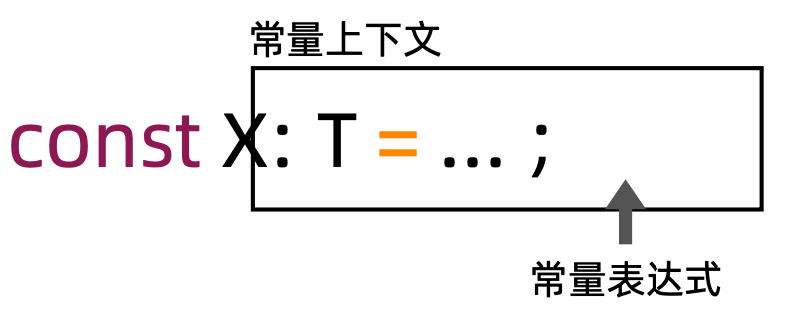
* + - 常量上下文(const context)包含:

1. 常量值初始化位置

2. 静态数组的长度表达式，[T; N]

3. 重复的长度表达式，类似于 :[ 0; 10]

4. 静态变量、枚举判别式的初始化位置



* + - 谨记:

1. 常量上下文是编译器唯一运行进行编译期求值的地方

2. 在非常量上下文的地方，常量表达式不一定会在编译期求值

* + - 常量传播和编译期计算是不同的:

1. 常量传播是编译器的一种优化。

2. 常量传播并不能改变程序的任何行为，并且对开发者是隐藏的。

3. 编译期计算则是指编译时执行的代码，必须知道其结果，才能继续执行。

* + - 常量函数(const fn) / 常量安全(Const Safe)

1. Rust 里的大部分表达式都可用作常量表达式。

2. 并不是所有常量表达式都可以用在常量上下文。 3. 编译期求值必须得到一个确定性的结果。

* + - 常量函数(const fn)/ 常量安全子类型系统

1. 普通的 fn 关键字定义的函数，是Safe Rust 主类型系统保证安全。

2. const fn 定义的函数，是 Safe Rust 主类型系统下有一个专门用于 常量计算的子类型系统来保证常量安全。

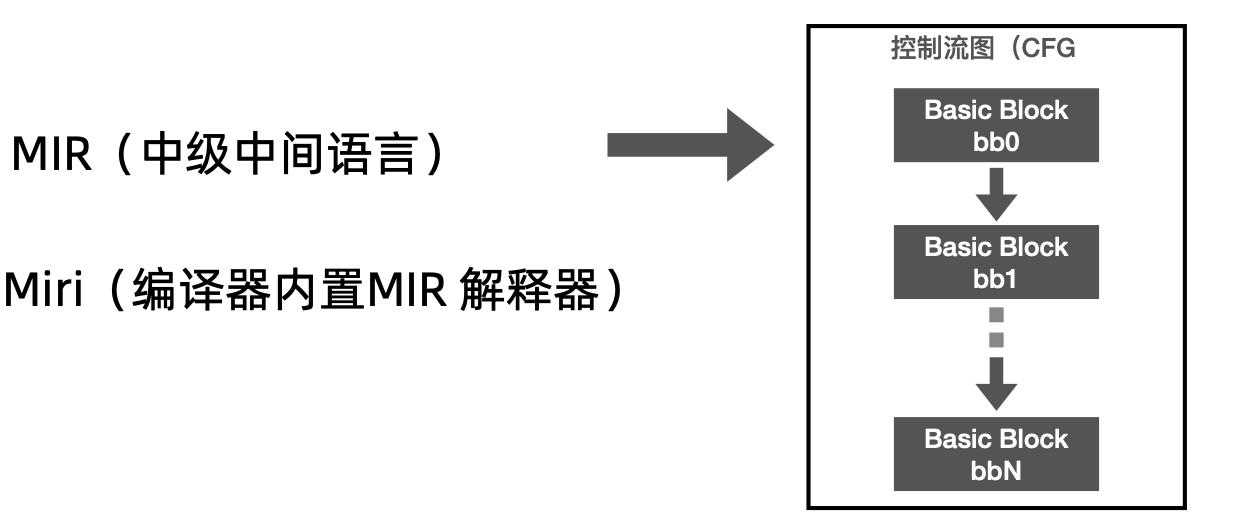
* + - 常量函数(const fn)/ 常量上下文可接受的常量表达式

1. const fn 函数

2. 元组结构体

3. 元组的值

* + - 常量函数(const fn)/ 编译期计算如何实现



* + - 常量函数(const fn)/ while true vs loop

想使用无限循环的时候，建议你使用 loop，而非 while true。

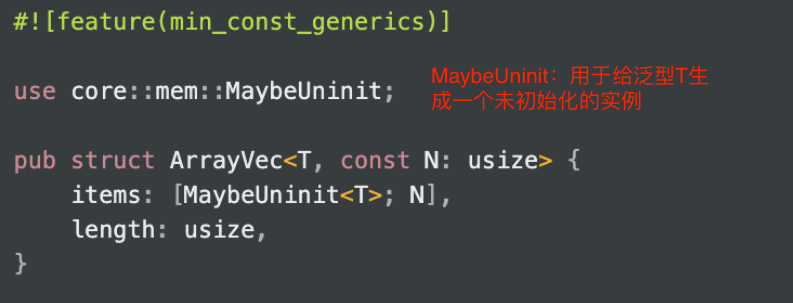
在while true 中未对变量初始化会得到编译期警告，即使加上了 #[allow(while\_true)] 属性。

Rust 编译器为什么不识别 while true?

1. 要考虑:while (constexpr == true) 的情况。

2. 使用 #[allow(while\_true)] 属性在某些情况下 允许使用 while true。

* + - 常量泛型(const generic) / 是什么?



* + - 常量泛型(const generic) / 类型理论

常量泛型是一种依赖类型(Depended Type)。

因为，数组 [T; N] 的类型，最终是要依赖于 N 的具体值来确定。

* + - 常量泛型(const generic) / 目前存在的缺陷

1. 目前仅限于整数原生类型，包括有符号和无符号整数类型， 布尔值和字符还不允许使用复合类型和自定义类型，也不允 许使用引用，这就意味着不允许使用字符串。

2. 常量泛型参数目前只支持两种表达式:

一个简单的常量泛型参数，比如 代码示例中的 `const N: usize`。

可以在不依赖于任何类型或常量参数的常量上下文中使用的表达式。

上面说的缺陷可能会在后续的版本中修复。

* + - 小作业:

1. 使用最新的 Rust 稳定版，尝试编写几个使用 if 、macth、loop等表达式的常量函数，也可以尝试一下在常量函数里使用 println 宏进行打印，看看会你会发现什么?

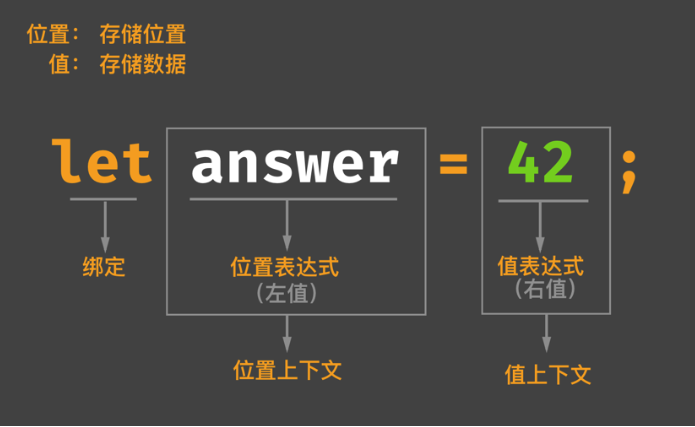
2. 在 crates.io 中搜索到 const-sha1 库，去看它的源码实现，看它的内部是如何应用 const fn 函数的。

3. 思考标准库 中 std::vec::Vec 的 new 方法，为什么是常量函数?这样做有什么好处?

|  |
| --- |
| /// 创建一个新的空 `Vec<T>`。  ///  /// 直到将元素压入 vector 为止，vector 才会分配。  ///  /// # Examples  ///  /// ```  /// # #![allow(unused\_mut)]  /// let mut vec: Vec<i32> = Vec::new();  /// ```  #[inline]  #[rustc\_const\_stable(feature = "const\_vec\_new", since = "1.39.0")]  #[stable(feature = "rust1", since = "1.0.0")]  #[must\_use]  pub const fn new() -> Self {  Vec { buf: RawVec::NEW, len: 0 }  } |

4. 继续完善 ArrayVec 示例，参考: https://github.com/Michael-F-Bryan/const- arrayvec

从表达式的分类角度来看 Rust 的变量绑定和引用



* + - Rust 中表达式的分类 / 位置表达式

包括:

1. 静态变量初始化。 比如:static mut LEVELS: u32 = 0;

2. 解引用表达式。形如 \*expr

3. 数组索引表达式。形如 expr[expr]

4. 字段表达式。 形如 expr. field

5. 以及，加上括号的位置表达式，形如 ( expr)

除此之外的，都是值表达式。

* + - Rust 中表达式的分类 / 位置上下文

1. 除了赋值左侧的位置上下文之外，还有 「复合赋值操作」的左侧。

2. 一元「借用」和「解引用」操作中的操作数所在区域。

3. 字段表达式的操作数所在区域

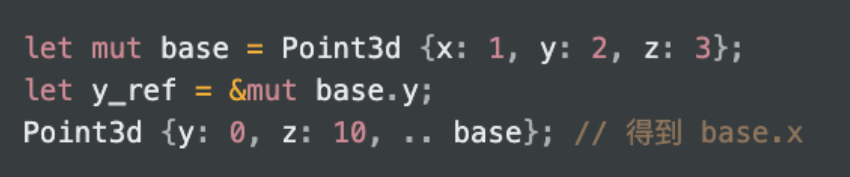
4. 数组索引表达式的操作数所在区域

5. 任意隐式借用操作数所在区域

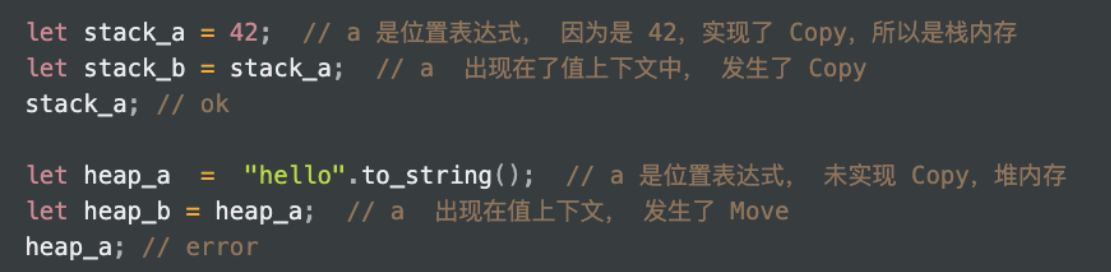
6. let 语句初始化

7. if let / while let/ match 的匹配表达式所在区域

8. 结构体更新语法中的 base 表达式(..操作符后面的操作数区域)



* + - 当位置表达式出现在了值上下文中



* + - 【补充】Copy语义
    1. 原生类型，包括函数、不可变引用和裸指针实现了 Copy；
    2. 数组和元组，如果其内部的数据结构实现了 Copy，那么它们也实现了 Copy；
    3. 可变引用没有实现 Copy；
    4. 非固定大小的数据结构，没有实现 Copy。
    - Rust 借鉴了函数式语言的不可变特性 包括:

不可变绑定与可变绑定

不可变引用与可变引用

理解：想要改变一个不可变绑定变量的值，可以使用变量遮蔽，也就是继承式可变，其实就是另一个变量，只是变量名相同。

* + - 【补充】可变引用与不可变引用的约束

为了保证内存安全，Rust 对可变引用的使用也做了严格的约束：

在一个作用域内，仅允许一个活跃的可变引用。所谓活跃，就是真正被使用来修改数据的可变引用，如果只是定义了，却没有使用或者当作只读引用使用，不算活跃。

在一个作用域内，活跃的可变引用（写）和只读引用（读）是互斥的，不能同时存在。

* + - 小作业:

1. 除了 let 赋值语句的值上下文之外，你还能找出哪些值上下文? 请写一些代码示例去尝试一下。

1.9 Rust 语法面面观:数据类型(上)

* + - 内容包括:

1. 基本数据类型

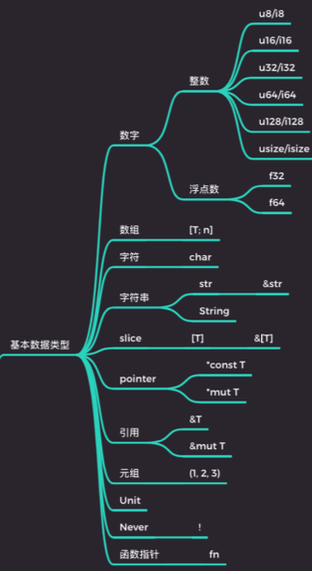
2. 自定义复合类型

3. 容器类型

4. 泛型

5. 特定类型

* + - 基本数据类型



* + - 基本数据类型 / 字符串

根据字符串使用场景设计类型

1. 字面量

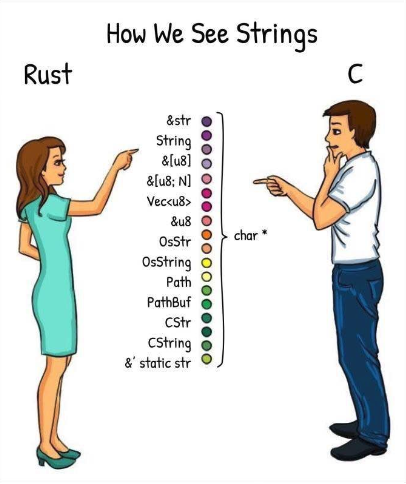
2. 动态可增长字符串

3. 从一段字符串中截取的片段

4. 字符串编码

5. FFi 中需要转换字符串到 C 或 OS 本地字符串

6. 遵循特定格式的文件路径



* + - 基本数据类型 / 字符串与切片

字符串为 UTF-8 编码的 u8 序列

1. str -> &str

2. [T] -> &[T], 比如 &[u8]

3. String -> Vec<u8>

* + - 基本数据类型 / 指针类型

三种指针类型:

1. 原始指针， \*mut T 和 \* const T。

2. NonNull 指针。 它是 Rust 语言建议的 \*mut T 指针的替代指针。 NonNull 指针是非空指针，并且是遵循生命周期类型协变规则。

3. 函数指针: 函数指针是指向代码的指针，而非数据。 你可以使用它直接调用函数。

* + - 基本数据类型 / 引用

两种引用类型:

1. &T 和 &mut T

2. 引用与指针的主要区别:

引用不可能为空

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

* 基本数据类型 / 元组

唯一的异构序列

1. 不同长度的元组是不同类型。

2. 单元类型的唯一实例等价于空元组。

3. 当元组只有一个元素的时候，要在元素末尾加逗号分隔 这是为了方便和括号操作符区分开来。

* 基本数据类型 / Never 类型

代表的是不可能返回值的计算类型

1. 类型理论中，叫做底类型，底类型不包含任何值，但它可以合一到任何其他类型。

2. Never类型用!号表示。

3. 目前还未稳定，但是在 Rust 内部已经在用了。

* 自定义复合类型

1. 结构体 Struct

2. 枚举体 Enum

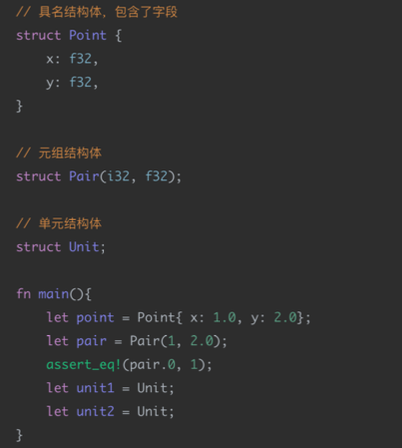
3. 联合体 Union

* 自定义复合类型 / 结构体 Struct

1. 具名结构体

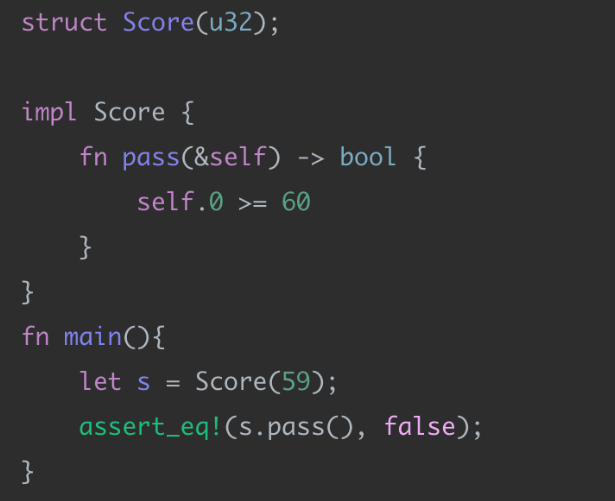
2. 元组结构体

3. 单元结构体：表示占位，不占据大小



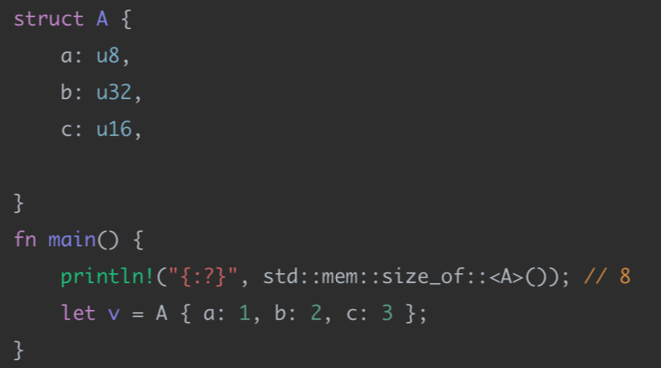
* 自定义复合类型 / NewType 模式

当元组结构体只包含一个成员时，如下所示，这里扩展了 u32 类型功能。

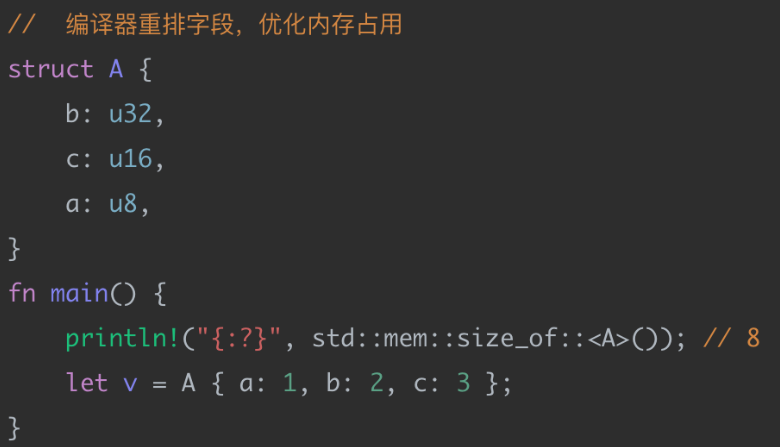


* 自定义复合类型 / 结构体内存布局

编译器会对结构体进行内存对齐，以便提升 CPU 的访问效率

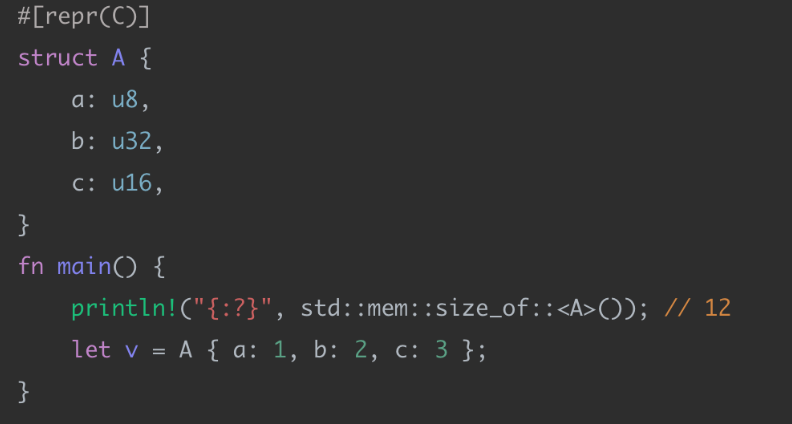


编译器重排后：



结构体A中会进行对齐，最大是b，占4个字节，a、c连在一起后共占4个字节，所以总共占8个字节。

使用内存布局属性可以指定 C 内存布局，阻止编译期重排。



* 自定义复合类型 / 枚举体与联合体内存布局

以枚举类型成员最大的对齐值为准

不需要为每个枚举值都对齐



对于enum，相当于是带有tag的unionl类型，这个tag会占一个字节。

enum A ：tag占一个字节，中间的元素不占字节，因此最终占一个字节。

enum E ：tag占一个字节，最大元素是 M(Box<u32>)，占8个字节，而tag对齐后占8个字节，因此总共占16个字节。

union U ：最大的是v，占8个字节，因此最终占8个字节。

* 容器类型



* 容器类型 / 内部可变性(interior mutability)

1. 与继承式可变相对应

2. 由可变性核心原语 UnsafeCell <T> 提供支持

3. 基于 UnsafeCell <T> 提供了 Cell<T> 和 RefCell<T>

* 泛型

所谓泛型，就是参数化类型，会进行静态分发。实现零成本抽象。

* 泛型/ turbofish 操作符

当类型推断失效，需要手工指定类型的时候使用。



* 特定类型

所谓特定类型，是指专门有特殊用途的类型

1. PhantomData <T>， 幻影类型。

一般用于 Unsafe Rust 的安全抽象。

2. Pin<T>， 固定类型。

为了支持异步开发而特意引进，防止被引用的值发生移动的类型。

* 小作业:

请自行阅读标准库文档，按课程中的类型分类，自己梳理一遍类型。

相信你能发现更细节的内容。

Rust 中的类型的行为

* Rust 内置 trait 分类

