

Chapter 1

Rust融合了多种编程范式。Rust最初被设计为系统编程语言，但随着发展，它已经成为了一门更加通用的编程语言，包括系统软件、网络服务、桌面应用程序、嵌入式等。

Rust的 **借用检查器(borrow checker)** 是它区别与其他基于C的语言的关键。它通过强制单一所有权原则促进安全编码。

函数式编程

函数式编程指的是以函数为基础构建块，在函数式编程中，函数被视为一等公民，你可以在任何通常使用变量的地方使用函数。

函数式编程的好处：

- 更加灵活
 - 更加透明
 - 不变性，它通过消除函数内的常见副作用，使程序更容易维护
-

面向表达式编程

Rust也是一种支持表达式编程的语言，在表达式编程中，大多数操作是能返回一个值的表达式，而不是什么都不返回的语句。

- **表达式** 就是一个返回值且只产生很小副作用或没有的一个或多个操作
- **语句** 语句不返回值，但它们可以改变程序状态，

面向表达式编程的好处是：

- 没有副作用，程序更容易维护
- 更加透明，表达式的值完全由其接口定义
- 易于测试，因为表达式是基于接口的
- 更容易编写文档，在没有副作用的情况下，表达式甚至可以替代文档
- 更易于组合

面向模式编程

专业的Rust编程通常涉及大量的模式，Rust使用match进行模式匹配。。与C++/Java中的switch语句不同(字符串和整型表达式的单维度模式匹配)，在Rust中模式匹配进一步扩展到用户自定义类型和序列的实例。

面向模式匹配编程有几个好处：

- 表达力强，能够将复杂代码简化为更简单的表达式
- 更全面，面向模式是对面向表达式编程的补充
- 更可靠，支持穷举模式匹配，这样更可靠且不容易出错。

特性

一、安全性

- 不可变性是默认行为，以防止意外更改
- 强制检查生命周期的正确性，以防类型悬挂引用等反模式
- 以引用的方式安全使用指针
- 对于大小动态变化的资源，使用资源获取即初始化(RAI)策略进行可靠内存管理

二、所有权

所有权通过单一所有者原则保证安全的内存访问。这一个原则将变量指派给单一所有者，而且拥有不会有超过一个所有者。

三、生命周期

生命周期是Rust的一个重要特性，用于避免程序对已失效的数据进行访问，在Rust中，引用本质上也是指针。如果不加约束可以通过悬挂引用访问当已经被释放的内存区域，从而导致程序崩溃。Rust通过生命周期解决了这个问题。借用检查器用来对生命周期的正确性进行检查，如果有错误的生命周期，则会编译错误，而生命周期标注是当生命周期存在歧义时开发者需要显式标注的。

四、无畏并发

当程序从顺序过渡到并发时，通常会进行一个称为强化的过程，以确保多线程代码有安全运行环境，其中一个标准步骤是移除作为共享数据的全局变量，有了无畏并发后就不需要这个强化过程了。

五、零成本抽象

这是Rust中的一个重要特性，是Rust中各个特性都遵循的原则，即无必要就应避免在运行时带来性能上的损失。

Rust术语

- crate, 在rust中，crate是最基本的编译单元，常见分类有
 - 可执行crate：一个独立于其他create运行的可执行二进制文件
 - 库crate：一个库，包含一组功能，用于其他create使用，不能独立运行
 - 外部crate：一个外部依赖项。比如crate A引用crate B，但B和A不在同一个包内，那么crate B就是外部crate。或者说依赖。
- 包 一个包由多个create(包括可执行和库)构成
- 模块 类似于其他语言的命名空间，可以使用模块在一个create中创建层次化的程序结构，mod也有助于避免名称冲突
- Cargo 在Rust中有几个Cargo相关部分
 - Cargo.toml 描述了包，包括依赖项，版本，作者，描述，等等
 - Cargo.lock 描述了包依赖项和特定版本，Cargo会自动生成这个文件
- .rs Rust源文件的扩展名

Rust工具

- Rustup Rust安装程序，按照Rust以及整个工具链
- cargo 多功能工具，主要功能是包管理，辅助功能是编译代码、格式化源代码和创建新的crate等

```
cargo new --lib mylib
```

- rustc Rust语言编译器，可以将Rust源文件(.rs)编译成可执行文件或库

```
rustc source.rs
```

- rustdoc 文档生成器，将Rust源文件中的文档注释编译成帮助文档，并将其输出为HTML格式
- Clippy 综合测试工具
- Rustfmt 将源代码转化为符合Rust风格指南的格式
- rust-analyzer Rust语言服务器，提供智能提示、错误检查、代码补全等功能

基础语法

本章节涵盖 Rust 编程语言的基础语法知识，包括变量、字符串、控制台输入输出和循环结构等内容。

chapter_2

引用

引用的作用是借用所指向内存位置中的值。引用是一种安全的指针，为确保安全性，Rust为引用施加了各种规则，而裸指针没有这些约束。

- 引用必须是非空的
- 底层值必须是有效的类型
- 引用是有生命周期的
- 引用的使用有一些特殊的限制，包括不限于所有权机制

对于引用，执行运算前会先解引用

```
fn main() {  
    let ref1 = &15;  
    let ref2 = &20;  
    let value1 = ref1+10;  
    let value2 = ref1*ref2;  
    println!("{} {}", value1, value2);  
}
```

如果不自动解引用，那么将这么写，可读性会变差

```
fn main() {  
    let ref1 = &15;  
    let ref2 = &20;  
    let value1 = *ref1+10;  
    let value2 = *ref1**ref2;  
    println!("{} {}", value1, value2);  
}
```

需要注意的是 `==` 比较的是引用处的值，而不是内存地址，如果你想要比较实际的内存地址，则可以使用 `std::ptr` 模块中的 `eq` 函数，其参数是要比较的引用

```
fn main() {  
    let num_of_eggs = 10;  
    let num_of_pizza = 10;  
  
    let eggs = &num_of_eggs;  
    let pizza = &num_of_pizza;  
  
    // 比较的值  
    println!("{}", eggs==pizza);  
    // 比较的地址  
    println!("{}", ptr::eq(eggs, pizza));  
}
```

逻辑运算

二元逻辑运算符是惰性运算符。只有在需要对整体表达式求值时才会进行求值。例如，仅当左操作符为真时，才会对`&&`运算符右操作数求值，如果`&&`运算符左操作数为假，右操作数就不会被求值，这种现象被称为短路。相比之下，`&`和`|`是不进行短路求值的

chapter_3

str

- str类型是一种原生类型，str类型具有切片的所有特性，包括无固定大小和只读。
- 因为str就是一个切片，通常会借用一个str，即&str。
- str类型有两个字段组成，指向字符串数据的指针和长度。

字符串字面量定义在("..")内，是程序整个生命周期内都存在的str值。因此字符串字面量的生命周期是静态的，表示为 `&'static str`

字符串

- 字符串(String)类型定义在Rust标准库中，是基于特化的向量实现的，由字符值组成。
- 字符串和向量一样是可变的也是可增长的。
- 字符串String类型包含3个字段：指向底层数组的指针、长度、容量。
 - 底层数组是分配的内存空间
 - 长度是按照utf-8编码占据的字节数
 - 容量是底层分配给数组空间的长度

创建字符串实例

- 使用String::from("str")和str::to_string()，将str转成String

```
fn main() {
    let str1 = String::from("hello world1");
    let str2 = "hello world2".to_string();

    println!("{} {}", str1, str2);
}
```

- 也可以通过new构造函数为String创造一个新的空字符串，然后将字符串追加进里面，前提是可变

```
fn main() {
    let mut new_str = String::new();
    new_str.push_str("hello world3");
    println!("{}", new_str);
}
```

如前所述，字符串是一种特殊的字符值向量，你甚至可以直接从向量创建一个字符串。

```
fn main() {
    let v = vec![65, 114, 107, 97, 110, 115, 97, 115];

    let str = String::from_utf8(v).unwrap();
    println!("{}", str);
}
```

在这个示例中，“Arkansas”这个码值被存储到一个向量v内，例如码值65对应的字符A，通过from_utf函数将这个向量转换成一个字符串

字符串长度

一个给定的unicode字符的长度是多少？这个问题看似简单实际上很复杂。首先，这取决于你是指字符串的字符个数还是字节。一个utf-8字符可以用1-4个字节来描述，ascii字符，在unicode中是1字节。然而位于代码空间其他位置的字符大小可能是多个字节

- Ascii是单字节大小
- 希腊字符是2字节大小
- 中文字符是3字节大小
- 表情符号是4字节大小

对于ascii,字节长度和字符数量是相同的。而对于其他字符集可能会有所不同。len函数返回字符串中的字节数

```
fn main() {
    let str = "你好世界";
    println!("{} {}", str.len());
}
```

理应打印的是4,实际上却是12 要获取字符串中字符的数量，可以首先使用chars返回字符串字符的迭代器，然后在迭代器上调用count方法来统计字符数量

```
fn main() {
    let str = "你好世界";
    let size = str.chars().count();
    println!("{} {}", size);
}
```

扩展字符串

你可以扩展一个字符串String的值，但你不能扩展str类型的值。下面提供了几个方法

- push 对于String追加char值
- push_str 对于String追加一个str

```
fn main() {
    let mut str = String::new();
    str.push_str("hello world");
    str.push('!');
    println!("{}" , str);
}
```

- insert
- insert_str

有时候，你可能不仅仅想在字符串末尾追加新内容，而是想将新内容插入到已有字符串中间的某个位置。

```
fn main() {
    let mut characters = "ac".to_string();
    characters.insert(1, 'b');
    println!("{}" , characters);

    let mut numbers = "one three".to_string();
    numbers.insert_str(4, "two");
    println!("{}" , numbers);
}
```

字符串容量

作为特化的向量，String具有一个底层数组和一个容量。

- 底层数组是存储字符串字符的空间，容量是底层数组的总大小，而长度则是字符串当前占用的大小。
- 当长度超过容量时，底层数组必须重新分配并进行扩展，当底层数组重新分配发生时，会有性能损失。因此避免不必要的重新分配可以提供程序的性能。

```
fn main() {
    let mut str = '我'.to_string();

    println!("容量:{} 长度:{}" , str.capacity() , str.len());

    str.push('是');
    println!("容量:{} 长度:{}" , str.capacity() , str.len());

    str.push('谁');
    println!("容量:{} 长度:{}" , str.capacity() , str.len());
}
```

上述例子将字符转换为字符串，然后每次追加一个字符，都引起字符串str的扩容，发生了两次重新分配，对应的3->8->16

如果一开始就能预估需要多大的字符值数组，那么在前面例子就可以给出更高效的写法，通过 `with_capacity()`可以在创建字符串时手动指定容量大小

```
fn main() {
    let mut str = String::with_capacity(12);
    str.push('我');
    str.push('是');
    str.push('谁');
    println!("容量:{} 长度:{} 内容:{}", str.capacity(), str.len(), str);
}
```

访问字符串的值

我们知道，字符串本质上就是字符值数组，所以通过数组下标索引来访问吗？

```
fn main() {
    let str = "你好世界".to_string();
    let ch = str[1];
}
```

然后你将会得到类似这样的错误

```
error[E0277]: the type `str` cannot be indexed by `'{integer}'`
```

虽然错误本身是正确的，但没有解释清楚根本原因。实际上，根本问题是：对字符串使用索引进行访问是存在歧义的，我们无法确定索引值究竟对应的字节位置还是字符位置。没有解决这一歧义，继续这种操作会被编译器认为是不安全的。因此，在Rust中直接通过索引来访问字符串中的字符是明确禁止的。你可以通过字符串切片来访问String中的字符，起始索引和结束索引来表示字节位置，切片的结果是一个str

```
string[startIndex..endIndex]
```

示例

```
fn main() {
    let str = "你好世界".to_string();
    let ch = &str[3..=5];
    println!("{} {}", ch);
}
```

字符串位置如下图所示

字节	0	1	2	3	4	5	6	7	8	9	10	11
你好世界	你		好		世		界					

当尝试获取字符串切片的前两个字符，但是切片的位置是不正确的，就会引发一个panic示例

```
fn main() {
    let str = "你好世界".to_string();
    let ch = &str[0..8];
    println!("{}", ch);
}
```

输出

```
byte index 8 is not a char boundary; it is inside '世' (bytes 6..9) of `你好世界`
```

在获取字符串切片之前，你可以手动调用str类型的is_char_boundary方法来确定给定索引位置是否与字符边界的开始对齐

```
fn main() {
    let str = "你好世界".to_string();
    println!("{}", str.is_char_boundary(0)); // true
    println!("{}", str.is_char_boundary(1)); // false
}
```

字符串里的字符

字符串由字符组成，一个很有用的操作是迭代每一个字符。比如对每个字符执行某种操作、对字符进行编码、统计字符数量、或者搜索并删除包含字母e的所有单词等等。字符串的chars方法会返回一个字符串迭代器，方便我们遍历访问字符串中的每一个字符。

```
fn main() {
    let str = "你好世界".to_string();
    for ch in str.chars() {
        println!("{}", ch);
    }
}
```

也可以使用迭代器的nth方法读取某个位置的一个字符

```
fn main() {
    let str = "你好世界".to_string();
    let ch1 = str.chars().nth(1).unwrap();
    println!("{}", ch1);

}
```

格式化字符串

在需要`&str`的场合，可以借用字符串的`&String`来代替。这时`String`会继承`str`的所有方法，这种隐式转换的原理就是`String`类型为`str`实现了`Deref trait`，这种自动转换行为被称为`Deref`强制转换，但反过来，从`str`转换为`String`类型是不允许的

```
fn foo(str: &str) {
    println!("{}", str);
}
fn main() {
    let str = "hello".to_string();
    foo(&str);
}
```

格式化字符串

如果你需要创建格式化的字符串，那么可以使用`format!()`宏，这个宏与`println!()`的用法类似，不同的是，它返回一个格式化后的字符串，而不是直接输出。

```
fn main() {
    let left = 5;
    let right = 10;
    let result = format!("{}+{}={}", left, right, left+right);
    println!("{}", result);
}
```

实用函数

字符串类型实现了丰富的方法，可以方便对字符串进行各种处理。以下是一些常用函数。

- **clear** :清除一个字符串但不减少当前容量，如果需要，你可以通过`shrink_to_fit`方法来减少容量

```
fn clear(&mut self)
```

- **contains** :在字符串中查找一个模式字符串，如果找到则返回`true`

```
fn contains<'a, P>(&'a self, pat: P)->bool
```

- **ends_with** :判断字符串是否以给定模式字符串结尾，如果是则返回true

```
fn ends_with<'a, P>(&'a self, pat: P)->bool
```

- **eq_ignore_ascii_case** :以不区分大小写的方式比较两个字符串，如果相同则返回true

```
fn eq_ignore_ascii_case(&self, other: &str)->bool
```

- **replace** :替换字符串中的模式，返回修改后的字符串

```
fn replace<'a, P>(&'a self, from: P, to: &str);
```

- **split** :用给定的分隔符将字符串拆分，返回一个迭代器以遍历拆分后得到的字符串数组

```
fn split<'a, P>(&'a self, pat: P)->Split<'a, P>
```

- **start_with** :判断字符串是否以给定模式字符串开始，如果是则返回true

```
fn starts_with<&'a, P>(&'a self, pat: P)->bool
```

- **to_uppercase** :将字符串转换为大写

```
fn to_uppercase(&self)->String
```

chapter_4

控制台命令行程序适合很多场景，如记录事件日志、配置应用程序、接收用户输入、访问开发者工具等等。它是GUI的一种可替代方案。

Rust提供了多种与控制台进行交互的读写操作，我们从最常用的`println!`和`print!`宏开始。

输出(`println!`宏)

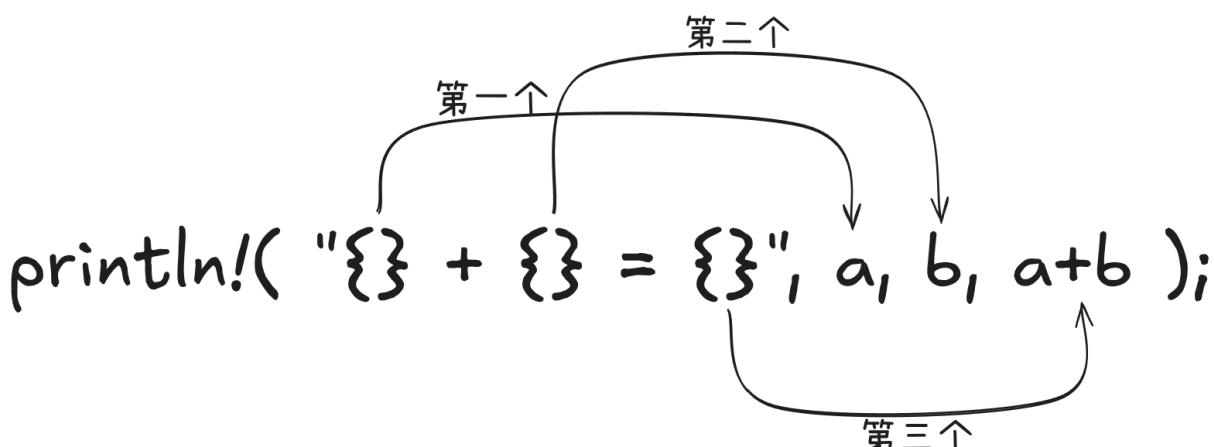
- `println!`和`print!`宏用于在控制台显示信息。这两个宏将格式化的字符串插入输出流(`stdout`)中。区别是`println!()`会在格式化输出的结尾加上一个换行符`\n`。
- `println!`和`print!`宏是可变参数的，意味着可以有多个数量的参数。
- 每个宏的第一个参数是一个格式化字符串也是一个字符串字面量。格式化字符串可以包含占位符`{}`，宏的其余参数必须依次替换占位符所对应的位置。
- `print!`宏至少必须有一个参数；而`println!`可以没有，仅仅表示打印换行。

格式化字符串

占位符`{}`保留给实现了`Display trait`的公共类型。标准库中很多原生类型(整数、浮点数)，都视为公共类型，它们实现了`Display`这个`trait`。比如自定义的结构体(`struct`)，可能并未实现`Display trait`，无法直接使用占位符`{}`进行格式化。

```
fn main() {
    let a = 10;
    let b = 20;
    println!("{} + {} = {}", a, b, a+b);
}
```

格式字符串`"{} + {} = {}"`中的占位符`"{}"`将会被后面的结果替换

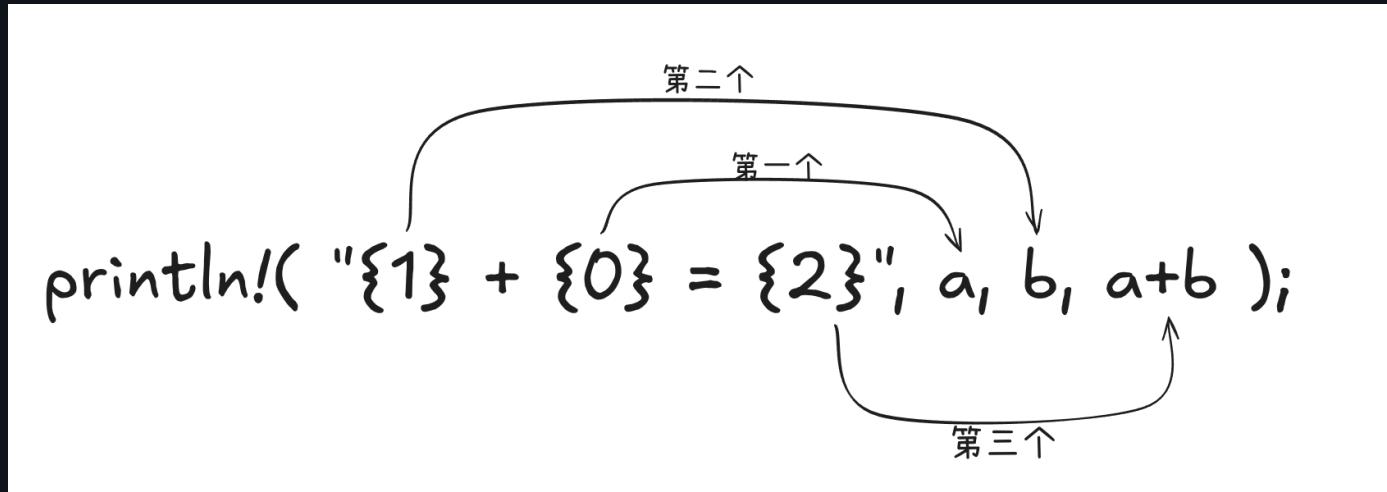


位置参数

你也可以使用索引来指示占位符 "`{}`" 中的位置参数，如 "`{index}`"。`index`是从0开始的，且是一个`usize`类型

```
fn main() {
    let a = 15;
    let b = 20;
    println!("{}+{}={}", a, b, a+b);
}
```

如图



当然，你也可以混用非位置型 "`{}`" 和位置型 "`{index}`" 的占位符。但是，非位置型参数占位符 "`{}`" 会被优先求值

```
fn main() {
    let (first, second, third, tourth) = (1, 2, 3, 4);
    let result = first+second+third+tourth;
    println!("{} + {} + {} + {} = {}", first, second, third, tourth, result);
}
```

变量参数

格式字符串中的占位符也可以引用变量。变量参数是一种位置参数，引用的变量必须在作用于内，并且是可见的。

```
fn main() {
    let (first, second, third, tourth) = (1, 2, 3, 4);
    let result = first+second+third+tourth;
    println!("{} + {} + {} + {} = {}", first, second, third, tourth, result);
}
```

命名参数

print!/println!宏内也可以使用命名参数。语法是name=value。这个参数可以在格式化字符串中的占位符中使用。

```
fn main() {
    let (first, second, third, fourth) = (1, 2, 3, 4);
    println!("{} + {} + {} + {} = {}", first, second, third, fourth, first+second+third+fourth);
}
```

填充、对齐和精度

在格式化字符串中，你可以控制占位符的填充、对齐方式和数字精度。只需要在占位符后的冒号处添加格式规范即可调整对应的属性，比如`{:format}`。

你可以使用`:width`来设置占位符的填充和列宽。在列中的对齐方式：

- 数字值默认是右对齐
- 字符串默认是左对齐 你可以使用以下特殊字符重写这些默认的对齐方式：
 - `>`：右对齐
 - `<`：左对齐
 - `^`：居中对齐

```
fn main() {
    let numbers = [("one", 10), ("two", 2000), ("three", 400)];
    println!("{}{:>10}","Text", "Value");
    println!("{}{:>10}","====", "====");
    for (k, v) in numbers {
        println!("{}{:<10}", k, v);
    }
}
```

对于浮点数，你可以在占位符`{}`中添加精度，用以控制小数点后显示的位数(并非四舍五入)。语法是`padding.precision`。如果没有填充宽度设置，直接使用`.precision`即可。需要注意的是，对于整型，`precision`参数会被忽略。

```
fn main() {
    let (float1, float2) = (3.14159, 1.2);
    println!("Result: {:.10.3}{:.10.2}", float1, float2);
}
```

你可以使用\$字符参数化精度或宽度。

基本写法

```
fn main() {
    let int = 43;
    let float = 3.14159;
    println!("{:5}", int);
    println!(" {:.3}", float);
}
```

这里5和3是写死的。

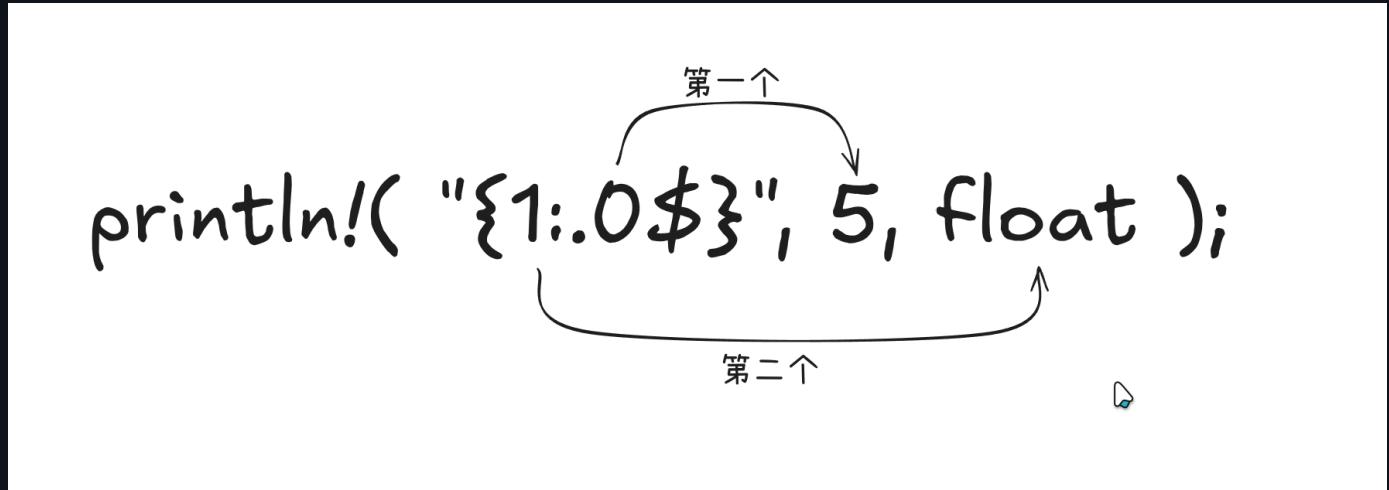
参数化宽度/精度(用\$)

```
fn main() {
    let width = 10;
    let precision = 5;
    let float = 3.1415926;
    println!("Result: {:width$.precision$}", float);
}
```

也可以和位置参数配合使用

```
fn main() {
    let float = 3.1415926;
    println!("{1:.0$}", 5, float);
}
```

如图



相应的，填充精度也可以参数化，这里不做过多演示。

进制

在 `print!` / `println!` 宏中，数值默认是以10进制表示的，如果你想使用其他进制，可以使用相关的字母标识符。

```
fn main() {
    println!("{:b}", 42); //二进制
    println!("{:o}", 42); //八进制
    println!("{}", 42); //十进制(默认)
    println!("{:x}", 42); //十六进制(小写)
    println!("{:X}", 42); // 十六进制(大写)
}
```

开发者友好

在使用 `println! / print!` 宏时，

- 对于已经实现了 `Display trait` 的类型，用 `{}` 占位符
- 对于已经实现了 `Debug trait` 的类型，用 `{:?}` 占位符

这种区分是站在用户友好的角度考虑，`{:?}` 格式通常注重开发者视角。

- 对于标准库中的原生类型同时实现了 `Display` 和 `Debug` 这两个 `trait`
- 但对于某些复杂类型，如数组和向量，只实现了 `Debug` 这个 `trait`

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

- 而用户自定义的类型通常不会自动实现 `Display` 或 `Debug` 这两个 `trait`，不过可以使用派生 (`derive`) 属性为自定义类型添加 `Debug` 的默认实现，之后在 `println! / print!` 宏调用时，可以使用 `{:?}` 占位符来输出

```
#[derive(Debug)]
struct Person<'a> {
    name: &'a str,
    age: i32,
}

fn main() {
    let person = Person {name: "张三", age: 43};
    println!("{:?}", person);
}
```

Rust还提供了 `{:#?}` 占位符，可以实现更优雅地输出。

```

#[derive(Debug)]
struct Person<'a> {
    name: &'a str,
    age: i32,
}

fn main() {
    let person = Person {name: "张三", age: 43};
    println!("{:?}", person);
}

```

输出(write!宏)

- `print!` 和 `println!` 可以将内容显示到标准输出。而 `write!` 更加灵活，可以将格式化的字符串输出到实现了 `fmt::Write trait` 或 `io::Write trait` 的不同目标上。
- `write!` 宏的参数包括目标值、格式化字符串和格式参数。
- 格式化后的字符串会被写入到目标中。因此目标需要是可变的，并且是一个借用值(`write!` 宏只需要临时使用目标，不需要拿走所有权) 例如 `Vec` 类型实现了 `std::io::Write trait`，因此可以在 `write!` 宏中作为目标使用。

```

use std::io::Write;

fn main() {
    let mut vec = vec![];

    write!(&mut vec, "{}", 10);
    write!(&mut vec, "{}", 11);

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

```

`write!`宏将值10和值11转为字符串。对于`Vec`类型来说，`write!`宏会将以utf-8的字符串按从前到后字节顺序插入此处为`vec!`类型。

在将Stdout(标准输出流，在`std::io`模块中，调用`stdout`方法获取标准输出目标)作为目标时，`write!`宏的行为类似于`print!`宏，

```

use std::io::Write;

fn main() {
    write!(&mut std::io::stdout(), "{}", "Hello, World!");
}

```

Display trait

格式化字符串的{}接收实现了 `Display trait` 的类型的参数，用于用户友好的视图。对于自定义的类型，可能需要为它们实现Display trait。.

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

- 第一个参数是&self
- 第二个参数是Formatter(格式化器)，是一个输出参数，包含类型的用户友好格式渲染 该函数返回一个Result类型，以指示函数是否成功。

```
use std::fmt::Display;

struct Person {
    name: &'static str,
    age: i32,
}

impl Display for Person {
    fn fmt(&self, formatter: &mut std::fmt::Formatter<'_>) -> std::fmt::Result {
        write!(f, "名字叫{}年龄是{}", self.name, self.age)
    }
}
fn main() {
    let p = Person { name: "Alice", age: 25 };

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

`write!`宏将格式化后的字符串放入到formatter参数中。有了Display的实现，我们自定义的结构可以与{}占位符一起使用。

Debug trait

{:?}占位符渲染开发者友好的格式，与{:?}一起使用的是实现了 `Debug trait` 的类型。 `Debug trait` 在`std::fmt`模块中定义。类似于Display，trait.Debug trait包含一个fmt方法

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

在默认情况下，自定义类型没有实现Debug trait.为了便于调试，我们可以为其实现Debug，这样我们就可以使用{:?}占位符来输出自定义结构。

```

use std::fmt::{Debug};

struct Person {
    name: &'static str,
    age: i32,
}

impl Debug for Person {
    fn fmt(&self, f: &mut std::fmt::Formatter<'_>) -> std::fmt::Result {
        write!(f, "人类 {{ 姓名: {}, 年龄: {} }}", self.name, self.age)
    }
}
fn main() {
    let p = Person { name: "Alice", age: 25 };

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

```

format!宏

我们主要关注了与格式化输出有关的两个宏：print!和println!。其实还有几个相关的宏用于创建格式化字符串。

- print! 将格式化字符串发送到标准输出。
- println! 将格式化字符串追加换行符后发送到标准输出。
- eprint! 将格式化字符串发送到标准错误
- eprintln! 将格式化字符串追加换行符后发送到标准错误。
- format! 创建一个格式化后的字符串。
- lazy_format! 创建一个延迟渲染的格式化字符串。

控制台读写

如果需要开发一个交互式的控制台应用程序，那么必然需要通过控制台与用户进行信息交换。Rust的标准库std::io模块提供了一些非常有用的方法。

- stdout 返回一个标准输出流的句柄，类型为Stdout
- stdin 返回一个标准输入流的句柄，类型为Stdin
- stderr 返回一个标准错误流的句柄，类型为Stderr 为了实现从控制台读取输入，Stdin实现了BufRead和Read这两个trait,提供了多个方法。
- read 将输入读入字节缓冲区
- read_line 将一行输入读入一个字符串缓冲区
- read_to_string 读取输入直到文件结束，并将文件内容存入一个字符串缓冲区。

```
use std::io::stdin;

fn main() {
    let mut name = String::new();
    println!("请输入你的名字");
    stdin().read_line(&mut name).unwrap_or_default();
    if name != "" {
        println!("你好 {}", name);
    } else {
        println!("请输入正确的名字");
    }
}
```

除了使用输出宏以外，还可以直接利用标准输出流的句柄在控制台显示信息。

1. 首先调用stdout获取标准输出流的句柄对象
2. 可以通过write_all方法来显示字节数据

```
use std::io::{stdout, Write};

fn main() {
    stdout().write_all(b"helloworld");
}
```

chapter_5

for 表达式 for循环是一种基于迭代器的循环结构。迭代器需要实现Iterator trait.然后可以使用for循环。不过像数组或向量这样的集合类型并不是迭代器，而是实现了IntoIterator trait,该trait定义了如何从某种类型转换为迭代器。for循环也可以直接作用于实现了IntoIterator的集合类型。

for循环

下面演示了在for循环中使用范围字面量

```
use std::ops::Range;

fn main() {
    let r = Range{start: 0, end: 3};
    for i in r {
        println!("{}", i);
    }
}
```

在迭代器中，有一个enumerate方法，返回一个包含两个字段的元组，分别是当前项的索引和值

```
fn main() {
    let iter = (1..=10).enumerate();
    for value in iter {
        println!("{}: {}", value.0, value.1);
    }
}
```

数组和向量不是迭代器，但是for-in会通过IntoIterator trait转换为迭代器。

```
fn main() {
    let vec = vec![1,2,3,4,5,6];
    for i in vec {
        println!("{}: {}", i, i);
    }
}
```

我们尝试在数组或向量上使用enumerate方法是行不通的，因为IntoIterator trait没有提供相应的方法。在使用enumerate之前，我们需要通过iter/iter_mut/into_iter方法将集合类型转换为迭代器。

- iter() 生成不可变引用(&T)
- iter_mut() 生成可变引用(&mut T)
- into_iter() for-in默认调用的方法，通过消耗所有权将集合转化为迭代器

```
fn main() {
    let vec = vec![1, 2, 3, 4, 5, 6, 7];
    for value in vec.iter().enumerate() {
        println!("{}: {}", value.0, value.1);
    }
}
```

尝试在遍历过程中修改不可变引用的值，会报错

```
fn main() {
    let mut vec = vec![1, 2, 3];

    for item in vec {
        item *= 2;
    }
    println!("{}: {}", vec);
}
```

输出

```
cannot assign twice to immutable variable
```

这说明了，默认情况下for-in返回的是不可变的类型T，我们无法对其进行修改，因此我们需要返回一个可变引用(&mut T)。

```
fn main() {
    let mut vec = vec![1, 2, 3];

    for item in vec.iter_mut() {
        *item *= 2;
    }
}
```

而默认的T类型具有移动语义，在for-in中，这个变量的所有权也会被移走。当我们使用for-in循环时，这个变量的所有权会被移动到循环内部，这就导致for-in循环结束后，这个变量不再可用。会引起借用检查器报错。

```
fn main() {
    let vec = vec![1, 2, 3];

    for value in vec {
        println!("{}: {}", value.0, value.1);
    }
    println!("{}: {}", vec[0].0, vec[0].1);
}
```

解决方法是使用正确的迭代器。

```
fn main() {
    let vec = vec![1, 2, 3];

    for value in vec.iter() {
        println!("{}" , value);
    }
    println!("{}" , vec[0]);
}
```

loop表达式

loop在设计上是一个无限循环，而它不仅仅是一个"while true"，它相较于while true有额外的特性，因为loop可以用作表达式。下面演示了找到第一个偶数的例子

```
fn main() {
    let vec = vec![1,5,6,4,5];
    let mut iter = vec.iter();

    let value = loop {
        let value = iter.next().unwrap();

        if value%2==0 {
            break value;
        }
    };

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

循环标签

在嵌套循环中，for/while/loop通常只能break或continue当前的循环，然而在某些情况下，我们需要跳出到外层循环，或者跳过当前循环。使用循环标签可以解决这个问题。定义标签

```
'label: loop
'label: while
'label: for
```

跳出标签

```
break 'label;
continue 'label;
```

示例

```
fn main() {
    'label: for i in 1..=10 {
        for j in 1..=10 {
            print!("{} ", j);
            if j == 5 {
                println!();
                continue 'label;
            }
        }
    }
}
```

Iterator trait

迭代器可以从头到尾遍历一个元素序列，可以很方便地和for/while/loop结合使用。

- 迭代器包含正向迭代器和反向迭代器。你也可以让迭代器返回普通的/可变引用/不可变引用的项
- 迭代器实现了Iterator trait。通常实现迭代器的类型都会维护一个游标，被称为项(Item)，也就是关联类型。
- 对于实现了Iterator trait的类型，next是唯一必须实现的方法。
- next方法会按顺序依次返回每个项，返回结果要么是 Some<T> 要么在项目遍历完后返回 None。迭代器的next方法会在for-in循环中被隐式调用，而使用while或loop时需要显式调用。

```
struct Triangular(i32);

impl Iterator for Triangular {
    type Item = i32;

    fn next(&mut self) -> Option<i32> {
        self.0 += 1;
        Some(self.0 * (self.0 + 1) / 2)
    }
}

fn main() {
    for t in Triangular(0).take(6) {
        println!("{} ", t);
    }
}
```

需要注意的是，这个三角序列是无穷的，next函数永远不会返回None，因此for-in会无限循环，相应的，使用take方法来仅返回前6项。

数据结构

本章节涵盖 Rust 中常用的数据结构，包括集合和哈希表等。

chapter_7

数组

- 数组是一种原生类型，可以在标准预制库中找到。
- 由固定数量的相同类型的元素组成。 数组的大小必须在编译时确定。 在 [] 内指定数组的元素类型和数量，如

```
array_name[type: length]
```

```
[value1, value2, value3, ..., valueN]
```

```
[value; repeat];
```

例如

```
fn main() {
    let array_1: [i32; 3] = [1, 2, 3];
    let array_2 = [4, 5, 6];
    let array_3 = [7; 3];
    println!("{:?}", array_1);
    println!("{:?}", array_2);
    println!("{:?}", array_3);
}
```

因为数组实现了Debug trait，因此我们可以通过{:?} 占位符打印数组。

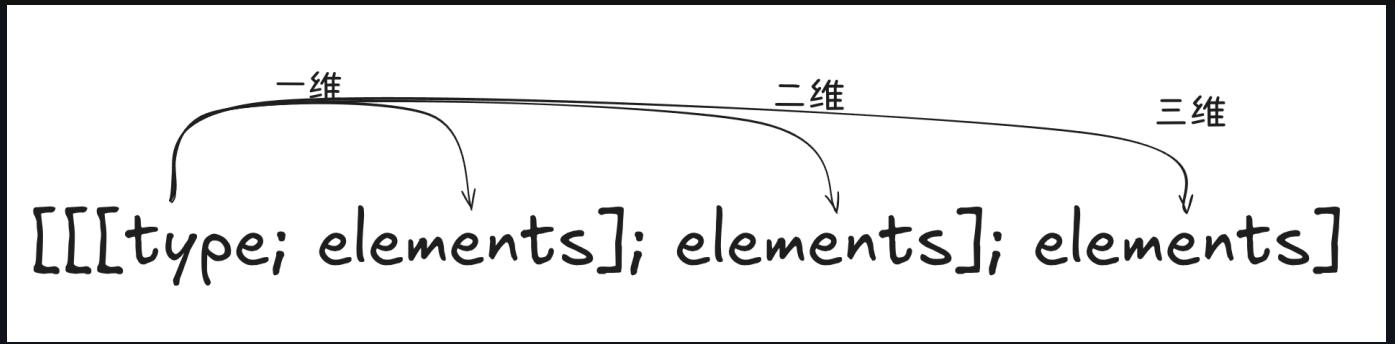
我们可以通过len方法输出数组的长度

```
fn main() {
    let array = [1, 2, 3, 4, 5, 6, 7];
    println!("{}", array.len());
}
```

多维数组

一维数组只由列组成，二维数组是由行和列组成的，而三维数组是由行、列和深度组成的。维度的数量是没有限制的。 多维数组的声明语法如下(以三维举例)：

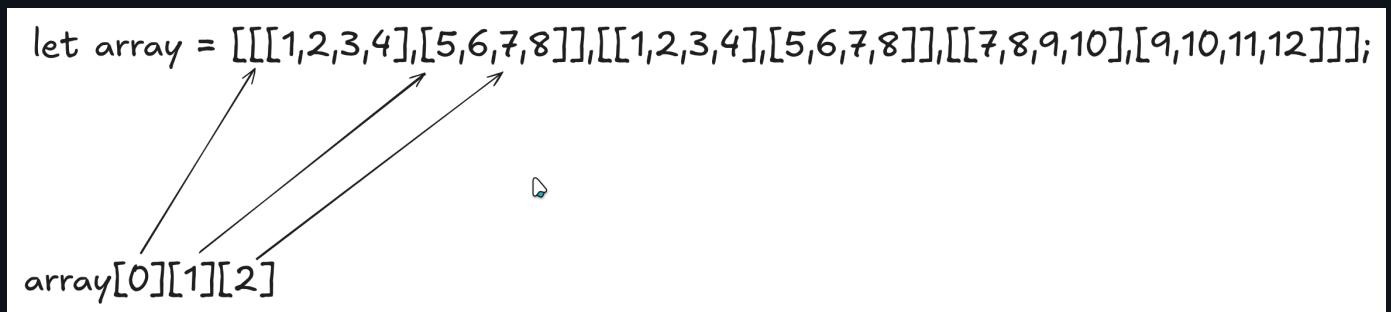
```
[[[type; elements]; elements]; elements]
```



访问数组元素

数组下标从0开始的

```
fn main() {
    let array = [[[1,2,3,4],[5,6,7,8]],[[1,2,3,4],[5,6,7,8]],[[7,8,9,10],[9,10,11,12]]];
    println!("{}",array[0][1][2])
}
```



当我们尝试在数组中移动一个元素，这会导致数组的一部分内容不再被数组拥有，是不可预算的行为。因此是不被允许的。

```
fn main() {
    let array = ["bob".to_string(), "alice".to_string()];
    let user = array[1];
}
```

修改方法就是使用借用

```
fn main() {
    let array = ["bob".to_string(), "alice".to_string()];
    let user = &array[1];
}
```

切片

一个切片是对数组一部分连续子元素构成的引用。语法如下

```
&arrayname[startIndex.. endIndex]
```

示例

```
fn main() {
    let array = [1,2,3,4,5,6];
    println!("{:?}", &array[..4]);
    println!("{:?}", &array[1..]);
    println!("{:?}", &array[1..4]);
    println!("{:?}", &array[1..=4]);
}
```

数组的比较

数组只实现了用于比较的PartialEq trait,只能比较==或!=。两个数组比较必须满足以下规则

- 必须是相同类型元素。
- 两个数组具有相同数量的元素
- 数组里的元素类型本身必须实现 PartialEq，否则没法比较。

```
fn main() {
    let array_1 = [1,2,3,4];
    let array_2 = [1,2,3];
    let array_3 = [1,2,3,5];
    let array_4 = [1,2,3,4];

    println!("{}" ,array_1==array_3);
    println!("{}" ,array_1==array_4);
}
```

迭代

数组实现了IntoIterator trait,可以使用for-in迭代，for-in会自动调用数组的into_iter方法转换成迭代器。

```
fn main() {
    let array = [1, 2, 3, 4, 5, 6, 7, 8, 9];

    // 不包含索引
    for item in array {
        println!("{}" ,item);
    }
    // 包含索引
    for (index, item) in array.iter().enumerate() {
        println!("{}:{}" ,index, item);
    }
}
```

隐式转换

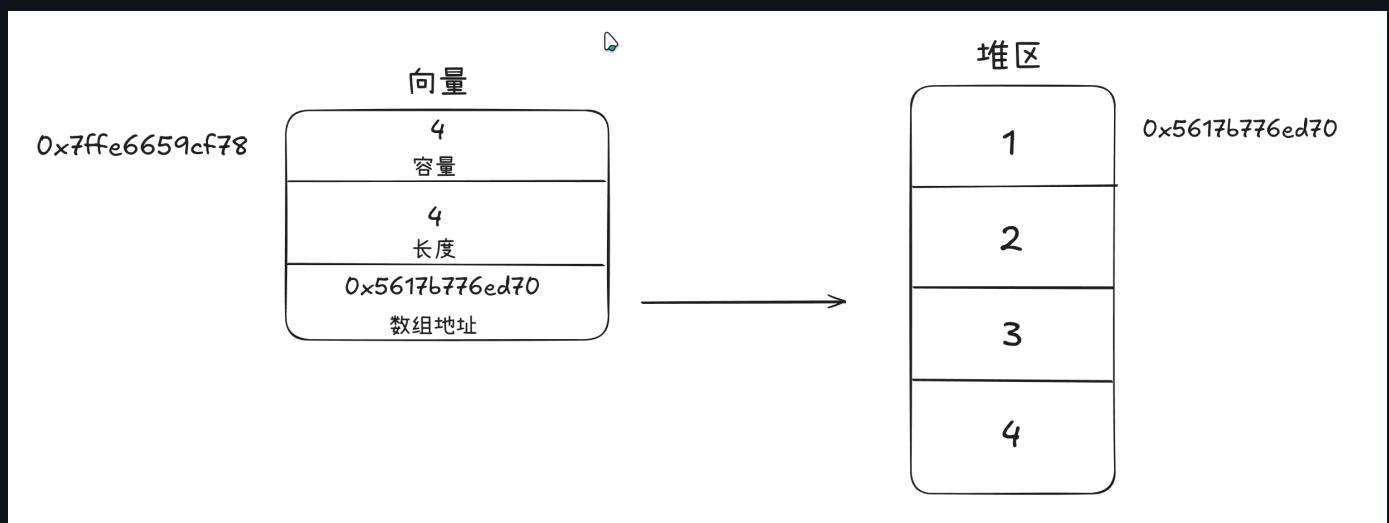
有时我们需要将数组转换为切片，或者将切片转换为数组。在Rust中，从数组到切片是隐式的，而从切片转换到数组不是隐式的。你可以在切片上调用try_into方法，这会返回一个 `Result<T, E>` 类型。如果成功，转换成的数组或包含在Ok()内，否则则返回一个 `Error<E>` 错误。数组类型需要手动声明，并且声明数组类型需要与原切片的长度相同。

```
fn main() {
    let slice = &[1,2,3,4][1..3];
    let array:[i32; 2] = slice.try_into().unwrap();
    println!("{}:?", array);
}
```

向量

向量是动态数组，与数组不同的区别是，向量可以动态地伸长和收缩。

- 由于向量的大小是动态的，它在编译时无法确定大小，因此向量本体不能驻留在栈上。
- 向量有一个底层数组，元素值存储在其中。这个底层数组是在堆上分配的。向量有三个字段
- 当前向量的大小
- 指向堆区底层数组的指针
- 底层数组的容量 这些字段不能直接访问，但可以通过调用方法进行访问。向量创建时会分配底层数组，当向量的绑定从内存中移除时，底层数组也会被释放。容量是向量当前的大小，长度是向量存储的实际元素的数量。当长度超出容量时，底层数组会被重新分配、复制，并增加容量。增加容量时，
 - 分配了一个更大的底层数组，
 - 所有值被复制到新的底层数组中
 - 原来的底层数组被释放
 - 更新向量的指针和容量。下图描述了一个常规向量的内存布局



创建向量

通过new方法创建一个空向量

```
fn main() {
    let empty_vec = Vec::<i32>::new();
    println!("容量:{} 长度:{}" , empty_vec.capacity(), empty_vec.len());
}
```

使用vec!初始化向量

```
fn main() {
    let vec = vec![1, 2, 3, 4];
    println!("{:?}", vec);
}
```

将数组转换为向量

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

将切片转换为向量

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

向量访问

对于向量的访问，使用get方法是更健全的方案。因为它返回的是一个Option枚举，如果成功，则返回 `Some<T>` (其中T是索引对应的值)，如果失败，则返回 `None`。

```
fn main() {
    let vec = vec![1, 2, 3, 4];
    if let Some(val) = vec.get(5) {
        println!("{}" , val);
    } else {
        println!("索引值为空");
    }
}
```

迭代元素

你可以想遍历数组那样遍历向量。

```
fn main() {
    let vec = vec![1, 2, 3, 4];
    for value in vec {
        println!("{}{}", value);
    }
}
```

添加或移除元素

- push() 将一个元素添加到vec末尾，没有返回值
- pop() 移除最后一个元素并返回 `Option<T>`，成功则将pop的值作为 `Some<T>` 返回，失败则返回None。
- insert(index, value) 向任何位置(指定位置前面)插入元素，没有返回值。

容量

对于一个向量，容量是其底层数组的大小。当向量的大小要超出当前容量时，底层数组会被重新分配。合理地管理容量能够提高向量的性能。常用管理容量的方法

- with_capacity 用于设置初始容量
- reserve 增加现有vec的容量
- shrink_to_fit减小vec的容量来节约未使用的内存。

```
fn main() {
    let mut vec = Vec::with_capacity(5);
    vec.push(1);
    vec.push(3);
    vec.push(5);
    vec.push(7);
    vec.push(9);
    println!("当前容量:{} 元素个数: {}", vec.capacity(), vec.len());

    vec.reserve(8);
    println!("当前容量:{} 元素个数: {}", vec.capacity(), vec.len());

    vec.shrink_to_fit();
    println!("当前容量:{} 元素个数: {}", vec.capacity(), vec.len());
}
```

chapter_7

哈希表是一种查找表，其中的条目有键(key)和值(value)组成。它是一个可变集合，可以在运行时插入和移除条目，类似于其他语言的字典和表。

- 键是唯一的，而值可以是重复的。
- 通过键，你可以快速查找到所需的值，就像访问数组一样方便。
- 键的类型不局限于usize,几乎所有类型都可以。比如整数、字符串、结构、数组甚至是其他哈希表。

HashMap<K, V>类型

其中K是键类型，V是值类型，

- 所有键K对应的同一类型，所有的值V也对应的同一类型
- 键的类型必须是实现了Eq和Hash这两个trait.

```
#[derive(PartialEq, Eq, Hash)]
```

- 对于HashMap类型,默认的散列函数实现采用了二次探测和SIMD查找。
- 此外，默认的哈希器内置了合理的防御机制，可以低于哈希DS攻击。
- 为了进一步增强安全性，哈希的计算过程还引入了基于系统熵的随机密钥。
- 开发者也可以自行实现BuildHasher trait来替换默认的哈希器。也可以在creates.io上寻找现成的第三方哈希器。

哈希是可变的集合，因此哈希表的条目会被放置在堆上。可以像向量一样设置哈希表的容量。

创建HashMap

HashMap类型不包含在标准预先导入中，其位于 `std::collections::HashMap` 中。

1. 使用new方法创建新的HashMap

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

fn main() {
    let map = HashMap::<i32, String>;new();
    println!("{} {}",map.capacity(),map.len());
}
```

2. 使用from方法从元组数组创建 其中 `tuple.0` 是K， `tuple.1` 是值

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

fn main() {
    let tuple_array = [("张三", 21), ("李四", 34), ("王五", 43)];
    let map = HashMap::from(tuple_array);
    println!("{:?}", map);
}
```

添加和删除元素

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

fn main() {
    let tuple_array = [("张三", 21), ("李四", 34), ("王五", 43)];
    let mut map = HashMap::from(tuple_array);
    println!("{:?}", map);
    map.insert("赵六", 49);
    println!("{:?}", map);
    map.remove("赵六");
    println!("{:?}", map);
}
```

访问HashMap

get方法可以用键(K)在HashMap中查找一个值。该方法返回一个Option枚举，如果键存在，其值作为Some(value)返回，如果键不存在则返回None。

更新条目

仅需对一个已经存在的键(K)插入一个新的元素即可

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

fn main() {
    let mut map = HashMap::from([('张三', 32), ('王五', 44), ('赵六', 66)]);

    map.insert('王五', 88);
    println!("{}: {}", "王五", map['王五']);
}
```

有时候知道insert是插入还是更新还是挺有用的，当插入新条目时，insert返回的是None;而当更新已存在的值时，insert返回的是 Some<T>，其中T是更新之前的值。

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

fn main() {
    let mut map = HashMap::from([('张三', 32), ('王五', 44), ('赵六', 66)]);

    let result = map.insert("李四", 88);

    match result {
        Some(old_value) => println!("{}被更新", old_value),
        None => println!("插入了新元素")
    }
}
```

另一种方式是通过entry方法，接受一个参数，即HashMap的键，返回一个 `Entry<K, V>` 枚举，表示该条目被占用还是空缺。声明如下：

```
pub enum Entry<'a, K: 'a, V: 'a> {
    Occupied(OccupiedEntry<'a, K, V>),
    Vacant(VacantEntry<'a, K, V>),
}
```

- Occupied(占用)，代表查找的条目被找到。
- Vacant(空缺)，代表此条目不存在。那么
- 如果条目是Occupied，则可以通过`or_insert`函数返回一个该值的可变引用。可以解引用修改这个值，
- 如果条目是Vacant，那么`or_insert`函数则设置一个默认值。

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

fn main() {
    let mut map = HashMap::new();
    map.insert("张三", 31);
    map.insert("王五", 42);
    map.insert("李四", 45);
    map.insert("赵六", 78);

    // 如果不存在，就创建新的Key，对于的Value为0
    let value = map.entry("张三").or_insert(0);
    // 已经存在，返回可变引用，可用于修改value值
    *value = 99;

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

迭代

可以使用`for-in`遍历

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

fn main() {
    let mut map = HashMap::new();
    map.insert("张三", 43);
    map.insert("李四", 56);
    map.insert("王五", 89);
    for (name, age) in map.iter() {
        println!("姓名:{}，年龄:{}", name, age);
    }
    map.insert("赵六", 99);
}
```

核心概念

本章节涵盖 Rust 语言的核心概念，包括所有权、生命周期和错误处理等。

chapter_8

要理解所有权，首先要能很好地区分栈和堆。

- 栈是分配给每个线程的私有内存空间，当一个函数被调用时，系统会在栈上创建一个栈帧，其中包含了函数的状态信息，例如局部变量和参数。栈帧的大小在编译阶段确定。当函数执行结束时，相应的栈帧就会从内存中移除，此时栈也会随之缩小，同时函数的状态信息，包括局部变量在内，都会从内存中清除掉。
- 堆内存为应用程序在运行时进行动态内存分配提供了存储空间。与栈不同，堆属于应用程序的共享内存区域，任何线程都可以访问。在分配好堆内存后，系统会返回一个指向所分配内存位置的指针。

有时候，栈上的变量会包含对堆内存的引用。当栈上的变量销毁(从内存中移除)时，与之相关的堆内存也必须被释放掉。

深拷贝和浅拷贝

- 浅复制是按位复制，这种复制既简单又高效，浅拷贝适合除了指针类型的绝大多数类型
-

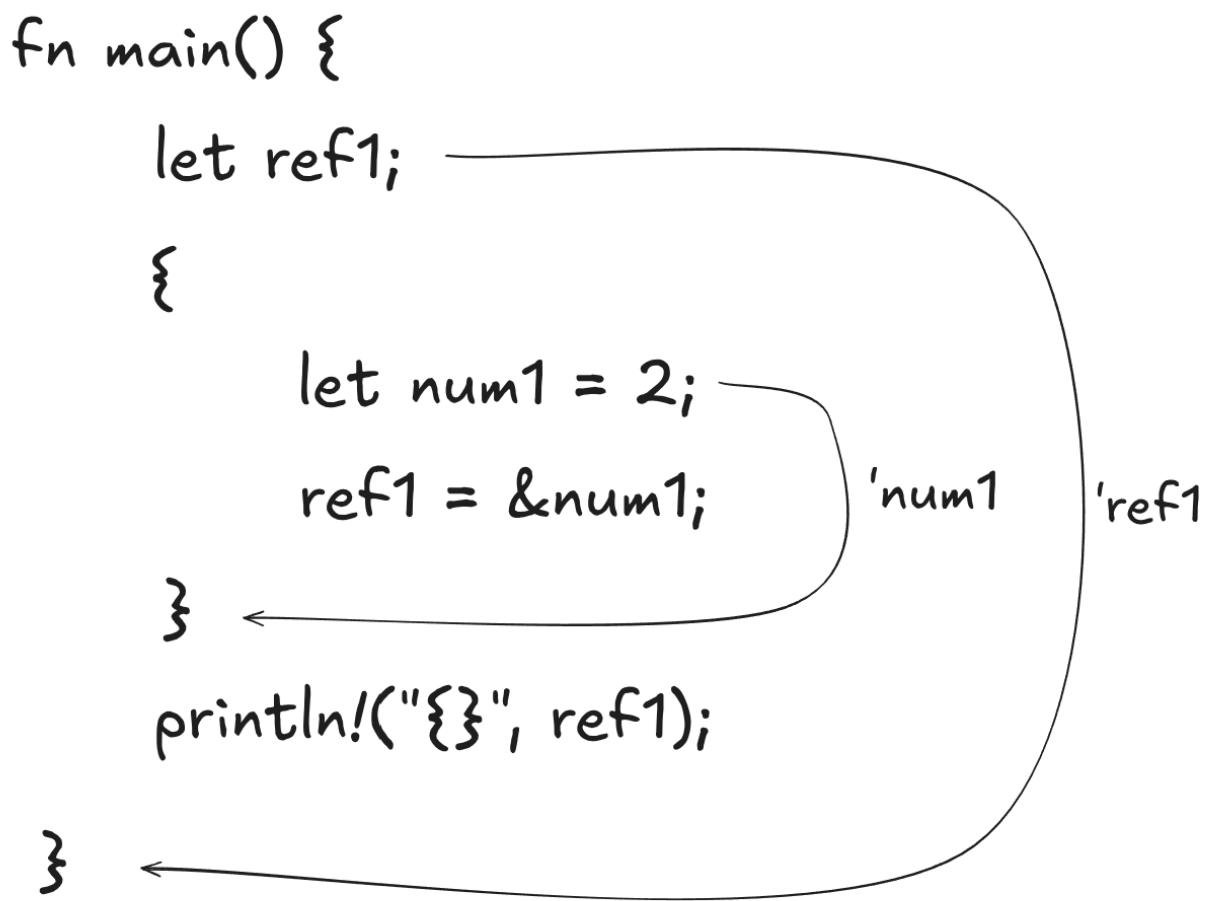
chapter_9

生命周期描述的是一个值在程序中的存活范围。不同生命周期的交叉关系，可能会导致悬垂引用的出现。

生命周期的命名方式是('生命周期名称)。不同通常以小写字母命名,如'a、'b、'c。

```
fn main() {
    let ref1;
    {
        let num1 = 2;
        ref1 = &num1;
    }
    println!("{} {}", ref1);
}
```

在'num1的生命周期结束时，num1就被释放了。此时ref1就变成了一个悬垂引用，借用检查器检测到了这个问题，因此报错。



函数与生命周期

函数也可以产生不安全的内存访问，特别是引用作为参数并返回值的函数。对于一个包含引用的函数定义来说，我们需要考虑三种不同类型的生命周期：

- **输入生命周期** 即类型参数是引用类型的情况，一个函数可以有多个输入生命周期
- **输出生命周期** 指返回值的生命周期，输出生命周期通常会从函数的输入生命周期中选取(虽然可以返回静态值的引用，但这种情况不常见)

```
'a      'a
fn do_something(ref1: &i32)->&i32 {
    ref1 //返回'a
}

fn main() {
    let num1 = 1;           'a
    let result = do_something(&num1);
    println!("{}result");
}
```

'a是num1的生命周期

- **目标生命周期** 指的是绑定到函数返回值引用的生命周期

```
输入生命周期是'a
fn do_something(ref1: &i32)->&i32 {
    ref1
}

fn main() {
    let num1 = 1;
    let result;           'b
    result = do_something(&num1);
    println!("{}result");
}
```

输出生命周期是'a

目标生命周期是'b

'b是result的生命周期

'b是num1的生命周期

只要目标生命周期的范围超出输出生命周期，就有可能导致悬垂引用的问题。在前面这个例子中目标生命周期('b)完全包含在输出生命周期('a)中，因此result变量始终是有效的，编译器成功编译。

生命周期省略

在遵循一定原则的情况下，编译器可以自动推导生命周期参数。省略规则如下：

- 每个被省略的输入生命周期会被分配为一个独立的生命周期参数
- 当存在单一输入生命周期时，它会被省略，并且应用到输出引用
- 对于一个方法，如果self是一个引用，则self的生命周期会省略，并且会应用到输出引用

复杂的生命周期

函数接受多个引用类型参数，每个引用可能具有不同或相同的生命周期。这就意味着有多个候选的输出生命周期，这会给借用检查器带来歧义。即哪个输入生命周期应该分配给引用类型的返回值作为输出生命周期？为此，编译器不会猜测哪个是正确的输出生命周期，而是需要开发者显式标注生命周期来向编译器表达意图。

共享生命周期

你可以在多个参数之间共享输入生命周期。当你共享一个生命周期时，该生命周期的范围就是两个借用值生命周期的交集。

静态生命周期

'static表示的一个静态的生命周期，贯穿整个应用程序。在Rust中，字符串字面量具有静态生命周期。字符串字面量是&str类型的值，他们总是静态的。

结构体与生命周期

结构体中的生命周期和其他地方的生命周期具有相同的作用，即防止悬垂引用。因为结构体可以有引用的字段，如果字段引用的值没有存活足够长的时间，就会出现悬垂引用。因此结构体中的字段必须具有与结构体本身一样长的生命周期。

- 要先在结构体名称后声明生命周期参数，然后你可以将命名的生命周期分配给引用类型的字段。
- 结构体不支持生命周期省略，一旦有引用类型的字段，必须要显式添加生命周期标注

方法与生命周期

为结构体实现的方法也可以接受和返回引用类型，这也是潜在的能产生悬空引用的场景。当我们实现方法时，结构体的生命周期参数既包括在impl关键字后，也包括在结构体名称后。

```
struct Data<'a, 'b> {
    field1: &'a mut i32,
    field2: &'b mut i32
}

impl<'a, 'b> Data<'a, 'b> {
```

在实现方法时，不需要重新声明结构体的生命周期，只需要简单地将生命周期应用到方法声明的引用类型上，这些和应用到字段的生命周期是相同的。当相同的生命周期应用到多个值时，编译器会采取保守计算，得到的生命周期是各个生命周期的交集。

子类型化生命周期

```
fn foo<'a: 'b, 'b>(x: &'a i32, y: &'b i32, flag: bool) -> &'b i32
{
    if flag { x } else { y }
}

fn main() {
    let x = 1;
    {
        let y = 2;
        let z = foo(&x, &y, true);
        println!("z: {}", z);
    }
}
```

'a: 'b 表示 'a的生命周期必须至少和'b一样长或更长。换句话说，'a是'b的子类型。这意味着'a可以替代'b使用

匿名生命周期

生命周期省略可能并不完美，有可能会出现生命周期推断错误，不适用的情况。另外生命周期标注可能过于繁琐。对于这种情况，可以使用匿名生命周期。

如果你希望在impl中省略生命周期，可以使用'_

```
use std::fmt::Display;

struct Data <'a>{
    ref1: &'a i32,
}

impl Display for Data<'_> {
    fn fmt(&self, f: &mut std::fmt::Formatter<'_>) -> std::fmt::Result {
        write!(f, "{}", self.ref1)
    }
}
fn main() {
    let x = 5;
    let data = Data { ref1: &x };
    println!("{}", data);
}
```

泛型生命周期

泛型和生命周期声明的方式相同，而且它们可以结合使用。同时声明时，生命周期参数应该先于泛型类型参数。

chapter_10

错误处理是应用程序应对异常的能力。应用程序可以采取主动或被动的方式进行错误处理。

- 主动错误处理 Rust提供了Result和Option这两个标准类型
- 被动错误处理 Rust提供了panic。当执行过程中(运行时)发生了无法继续运行的异常事件时，就会引发panic。

Rust 彻底放弃了“异常”这种隐式传播的思路，而是走编译期强制处理错误的路线

- 可恢复错误 (Recoverable Errors) : 用 Result<T, E> 类型表示。调用者必须显式处理 (match、?运算符)。-> 这就是所谓的主动错误处理，错误传播路径在类型系统里清清楚楚。
- 不可恢复错误 (Unrecoverable Errors) : 用 panic! 表示。通常意味着程序逻辑出现严重问题，无法也不应该继续执行。-> 这就是 Rust 的被动错误处理，类似“终止程序”的紧急刹车。

panic 和异常的区别 虽然 panic 和异常都基于栈展开 (stack unwinding)，但理念不同：

- 异常：设计成可以恢复的一般错误机制。
- panic：设计成“程序出 bug 或进入不可能状态时”的终止信号。

```
fn divide_two_numbers_in_ratio(first: i32, second: i32) -> i32 {
    first / second
}

fn logic() {
    divide_two_numbers_in_ratio(1, 0);
}

fn main() {
    logic();
}
```

当panic发生时，栈会按顺序展开: divide_two_numbers_in_ratio和logic以及最后的main函数。在程序终止时会输出用于诊断的错误信息，其中包含了panic发生的具体位置

```
thread 'main' panicked at src/bin/13_error.rs:2:4:
attempt to divide by zero
```

错误信息中还包含了调用栈回溯信息。

```
stack backtrace:
  0: __rustc::rust_begin_unwind
    at /rustc/1159e78c4747b02ef996e55082b704c09b970588/library/std/src/panicking.rs:69
  7:5
  1: core::panicking::panic_fmt
    at /rustc/1159e78c4747b02ef996e55082b704c09b970588/library/core/src/panicking.rs:7
  5:14
  2: core::panicking::panic_const::panic_const_div_by_zero
    at /rustc/1159e78c4747b02ef996e55082b704c09b970588/library/core/src/panicking.rs:1
  75:17
  3: _13_error::divide_two_numbersing_ratio
    at ./src/bin/13_error.rs:2:4
  4: _13_error::logic
    at ./src/bin/13_error.rs:6:5
  5: _13_error::main
    at ./src/bin/13_error.rs:9:5
  6: core::ops::function::FnOnce::call_once
    at /home/cangli/.rustup/toolchains/stable-x86_64-unknown-linux-gnu/lib/rustlib/src/rust/library/core/src/ops/function.rs:253:5
note: Some details are omitted, run with `RUST_BACKTRACE=full` for a verbose
backtrace.
```

最后提示你可以运行 `RUST_BACKTRACE=full cargo run`，获取更详细的调用堆栈信息。

当发生panic时，栈展开的过程为应用程序提供了有序退出时机，这时最重要的是释放占用的资源和内存。

- 有些清理工作是自动完成的。比如删除局部变量
- 有些需要特殊处理。比如堆内存对外部资源的引用。

在栈展开的过程中，实现了Drop trait的值会自动调用drop函数。相当于其他语言的析构函数。

```

struct Person {
    name: String,
    age: i32,
}

impl Drop for Person {
    fn drop(&mut self) {
        println!("drop被调用");
    }
}

fn division(first: i32, second: i32) {
    let p = Person{name: "李四".to_string() , age: 18};
    first / second;
}

fn logic() {
    let p = Person{name: "张三".to_string() , age: 18};
    division(1, 0);
}
fn main() {
    logic();
}

```

如果你没有想好为panic事件准备好清理策略，这种情况下，栈展开的过程可能会失去作用。更重要的是，如果没有合适的清理措施，应用程序可能会陷入未知或不稳定的状态，这种情况下最佳方案就是在panic发生时直接终止应用程序。在Cargo.toml中添加配置来实现这一行为

```

[profile.dev]
panic = "abort"

```

需要注意的是，任何人都可以改变这个外部配置，也就意味着这种行为可能会在你的控制之外发生。除了栈回溯外，这也是panic不可预测的并且应该避免的另一种原因。

panic!宏

panic是由异常情况引发的。其实也可以使用panic!宏来主动强制引发panic. 如果可以的话还是要尽量避免panic。panic!的场景

- 传播现有的panic
- 无法找到可行的解决方案
- 向应用程序发出无法拒绝的通知

```

fn main() {
    panic!("hello world")
}

```

包含了自定义描述信息

```
thread 'main' panicked at src/bin/13_error.rs:2:5:
hello world
```

Rust还提供一种高级版本的panic!宏，支持格式化字符串。

```
fn main() {
    panic!("hello {}", "张三")
}
```

处理panic

你可以根据实际情况采取不同措施。避免栈展开进入外部代码，这可能导致无法预知的行为。比如展开到系统调用就可能引发各种问题。在Rust中最好做必要的错误处理，如果实在无法进行错误处理，那么可以对panic进行有限的处理，比如记录panic信息并重新触发panic

- catch_unwind方法用于处理panic,在std::panic模块中，

```
fn catch_unwind<F: FnOnce() -> R + UnwindSafe, R>
    (f: F) -> Result<R>
```

catch_unwind接受一个闭包作为参数，并返回一个Result

- 如果闭包没有触发panic,则返回Ok(value) (其中value是闭包调用的结果)
- 当发生panic时，该函数返回Err(error)，其中error是panic的错误值

```
use std::any::Any;
use std::panic::catch_unwind;

fn get_data(request: usize) -> Result<i8, Box<dyn Any+Send>> {
    let vec: Vec<i8> = (0..100).collect();
    let ret = catch_unwind(|| {
        vec[request]
    });
    ret
}

fn main() {
    match get_data(100) {
        Ok(value) => println!("{}" , value),
        Err(_) => println!("数组访问异常")
    }
}
```

可以看到，即使进行了处理，panic消息依然被输出。其实每个线程都有一个panic hook,它是一个在panic发生时会被调用并输出panic的函数。正是这个hook函数将上面的回溯信息输出出来，

当这个功能被启用的情况下，使用std::panic模块中的set_hook函数来替换这个hook,panic hook的调用在panic发生和处理之间。解决办法是使用一个空的闭包替换默认的hook来去除panic信息

```
use std::any::Any;
use std::panic;

fn get_data(request: usize) -> Result<i8, Box<dyn Any+Send>> {
    let vec: Vec<i8> = (0..100).collect();
    panic::set_hook(Box::new(|_info| {}));
    let ret = panic::catch_unwind(|| {
        vec[request]
    });
    ret
}

fn main() {
    match get_data(100) {
        Ok(value) => println!("{}" , value),
        Err(_) => println!("数组访问异常")
    }
}
```

要能够处理panic首先需要了解panic。与panic相关的信息会以任何类型提供，需要先将其转换为特定类型，然后才能访问有关panic的具体信息。在前面代码中，我们忽略了panic的具体信息，现在我们需要输出panic的信息。将其向下转型为String

```
use std::any::Any;
use std::panic;

fn get_data(request: usize) -> Result<i8, Box<dyn Any+Send>> {
    let vec: Vec<i8> = (0..100).collect();
    panic::set_hook(Box::new(|_info| {}));
    let ret = panic::catch_unwind(|| {
        vec[request]
    });
    ret
}

fn main() {
    match get_data(100) {
        Ok(value) => println!("{}" , value),
        Err(error) => println!("{}" , error.downcast::<String>()),
    }
}
```

unwrap

在应用程序开发和测试阶段，许多开发者会使用unwrap函数来简化错误处理。通过unwrap可以把Result或Option中的错误结果转换为panic,这种做法有两个原因。

- 在开发阶段，展示没想好如何处理这些特定的错误。
- 想确保错误不会被忽略。不过unwrap函数有一些变体适用于更多的场景，而不仅仅是开发阶段。

首先介绍unwrap函数

如果Option/Result的值为None/Err(E)，就会引发panic

```
fn main() {  
    let vec = [1,2,3,5,5];  
    let ret = vec.get(5).unwrap();  
    println!("{}",ret);  
}
```

expect

可以用expect替换unwrap函数，区别是expect可以在panic时指定错误信息，而unwrap只有默认信息。

```
fn main() {  
    let vec = [1,2,3,5,5];  
    let ret = vec.get(5).expect("索引越界");  
    println!("{}",ret);  
}
```

有些unwrap函数的变体出现错误时不会引发panic,这样的方法对于错误处理非常有用。甚至可以处于非开发阶段。

unwrap_or

当出现错误时，这个方法会返回一个预设的替代值而不是直接引发panic。

```
fn main() {
    let vec = [1,2,3,5,5];
    let ret = vec.get(5).unwrap_or(&100);
    println!("{}",ret);
}
```

另外一个变体是unwrap_or_else

unwrap_or_else

替代值是一个闭包，这个替代值需要通过计算或涉及复杂逻辑时很有用。当unwrap出现错误(None/Err)时，会自动调用这个闭包。

```
fn main() {
    let vec = [1,2,3,5,5];
    let ret = vec.get(5).unwrap_or_else(|| &100);
    println!("{}",ret);
}
```

unwrap_or_default

在遇到错误时会返回对应类型的默认值。返回的具体默认值由类型决定，比如整型的默认值是0。需要注意的是，并非所有类型都有默认值，只要实现了Default trait的类型才有相应的默认值，而引用没有实现这个trait，所以不能使用这个方法。

```
fn main() {
    let vec: Option<i8> = None;
    let ret = vec.unwrap_or_default();
    println!("{}",ret);
}
```

Result/Option模式匹配

主动实现错误处理的函数会返回Result/Option枚举。调用方需要解析返回值并做出正确的处理。

- 标准的做法是使用match表达式，分别处理成功和失败两种情况

```

fn faker()->Result<i8, String> {
    Ok(0)
}
fn transform()->Result<bool, String> {
    match faker() {
        Ok(val) => Ok(val>10),
        Err(err) => Err(err),
    }
}
fn main() {
    let ret = transform();
    println!("{}", ret.unwrap_or_default());
}

```

map

map函数是一个高阶函数,接收一个闭包来执行转换过程。Option和Result都实现了map函数，这种方式就不需要前面的match了，而是依赖于map来转换Result和Option的值。

```

fn Option<T>::map<U, F>(self, f: F)->Option<U>
where F: FnOnce(T)->U

```

map函数将 `Option<T>` 转换为 `Option<U>`。这本质上是将Some变体中的T转换为U,如果结果是None,则map函数只会简单地传递None。

```

pub fn map<U, F>(self, op: F)->Result<U, E>
where F: FnOnce(T)->U

```

对于Result类型，map的行为相似，不过不同的是 `Result<T, E>` 被转换为 `Result<U, E>`,相应地，就是把Ok变体T转换为U,如果结果是Err,则map函数将传递这个信息。

```

use std::collections::HashMap;

fn main() {
    let mut map = HashMap::<&str, i32>::new();
    map.insert("张三", 42);
    map.insert("李四", 53);
    let ret = map.get("张三").map(|temp| (*temp as f64)/2 as f64);
    println!("{}", ret.unwrap_or_default()); // 新类型的值
    println!("{}", map.get("张三").unwrap()); // 原来的值
}

```

HashMap的get方法将返回一个指定K对应的 `Option<i32>` ,然后map将原来的值转换为f64并除2.0。然后返回 `Option<f64>`

and_then

与map作用效果类似不过不同的是:and_then函数将 `Option<Option<f64>>` 扁平化为 `Option<f64>` , 并返回展开后的内层结果 而下面这段代码无法编译

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

fn f_to_c(f: f64) -> Option<f64> {
    let f = f as f64;
    Some((f-31.5)*0.566)
}

fn into_celsius(cities: &HashMap<&str, f64>, city: &str) -> Option<f64> {
    let ret = cities.get(&city).map(|temp| f_to_c(*temp));
    ret
}

fn main() {
    let mut map = HashMap::new();
    map.insert("a", 42.0);
    map.insert("b", 42.1);
    map.insert("c", 57.1);
    into_celsius(&map, "a");
}
```

这是因为这里map返回的是 `Option<Option<f64>>` , 然而into_celsius的返回值为 `Option<f64>` 对于这种情况 , and_then函数可以解决这个问题。

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

fn f_to_c(f: f64) -> Option<f64> {
    let f = f as f64;
    Some((f-31.5)*0.566)
}

fn into_celsius(cities: &HashMap<&str, f64>, city: &str) -> Option<f64> {
    let ret = cities.get(&city).and_then(|temp| f_to_c(*temp));
    ret
}

fn main() {
    let mut map = HashMap::new();
    map.insert("a", 42.0);
    map.insert("b", 42.1);
    map.insert("c", 57.1);
    into_celsius(&map, "a");
}
```

这里into_celsius函数返回的是一个Option枚举。不过也有一种合理的观点是它应该返回Result枚举。into_celsius之所以返回Option,是为了方便对齐HashMap::get返回的Option类型。如果将into_celsius改为Result类型 , 就无法直接使用map函数了。在这种情况下可以

- 使用Option::ok_or函数 , 它可以将Option转换为Result ,
- 反过来使用Result::ok函数将Result转换为Option

```
fn ok_or<E>(self, err: E) -> Result<T, E>
```

ok_or示例

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

fn f_to_c(f: f64) -> Option<f64> {
    let f = f as f64;
    Some((f-31.5)*0.566)
}

fn into_celsius(cities: &HashMap<&str, f64>, city: &str) -> Result<f64, String> {
    cities
        .get(city)
        .and_then(|temp| f_to_c(*temp))
        .ok_or("转换失败".to_string())
}

fn main() {
    let mut map = HashMap::new();
    map.insert("a", 42.0);
    map.insert("b", 42.1);
    map.insert("c", 57.1);
    let ret = into_celsius(&map, "a");
    println!("{}: {}", ret.unwrap().round(2), ret.unwrap());
}
```

富错误

错误的值都是各不相同的，有些错误值包含了丰富的信息，能提供额外的重要细节。带有丰富信息的要求是实现Error trait和Display trait。io::Error是一种包含丰富信息的错误类型代表。

- unwrap_err从Result中解包Err值 它和unwrap的作用相反，当解包出Ok时则会引发panic.

自定义错误

你可以自定义包含丰富信息的错误。添加更多的信息有助于应用程序正确地响应错误。

高级特性

本章节涵盖 Rust 语言的高级特性，包括结构体、泛型、模式匹配、闭包、Trait 和线程等。

chapter_11

结构体

运算符重载

- 一元运算符 instance.operator()
- 二元运算符 左操作数lhs 右操作数rhs instance.operator(rhs)

一元运算符重载

```
use std::ops::Neg;

#[derive(Debug)]
struct RGB(u8,u8,u8);

impl Neg for RGB {
    type Output=RGB;

    fn neg(self) -> Self::Output {
        RGB(255-self.0, 255-self.1, 255-self.2)
    }
}

fn main() {
    let mut rgb = RGB(1,2,3);;
    rgb = -rgb;
    println!("{:?}",rgb)
}
```

二元运算符重载

Add trait是加法运算符的定义，是二元运算符的代表。

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

示例

```
use std::ops::{Add};

#[derive(Debug)]
struct RGB(u8,u8,u8);

impl Add for RGB {
    type Output=Self;

    fn add(self, rhs: Self) -> Self::Output {
        RGB {
            0: self.0+rhs.0,
            1: self.1+rhs.1,
            2: self.2+rhs.2,
        }
    }
}

fn main() {
    let rgb1 = RGB(1,2,3);
    let rgb2 = RGB(1,1,1);
    let rgb3 = rgb1+rgb2;

    println!("{:?}",rgb3);
}
```

chapter_12

泛型可以看作构建函数和类型定义的模板。标准库中有很多常见类型例如 `Result<T, E>`、`Option<T>`、`Vec<T>` 等都是泛型实现的。主要用于代码的特化。Rust是静态类型语言，因此泛型的类型参数(`T`)需要在编译期解析为具体类型，这种特性就是参数化多态性。泛型的实现还采用单态化技术，会根据实际传入的类型参数，将泛型实例化为唯一的专用类型。泛型的优点：

- 代码复用：只需编写一套代码，可以用于不同的类型。
- 重构：重构变得更简单，因为泛型只有一份源码，不需要维护多份类型不同但功能相同的代码
- 扩展性：可扩展性强，未来即使出现新的数据类型，泛型代码依然适用。
- 更不容易犯错：使用泛型减少了大量重复代码的实现，潜在的错误也变少。
- 独特功能：Rust中的泛型机制带来了独特的能力->函数重载

泛型函数

泛型函数是使用类型参数的函数模板，用于创建具体函数，

- 类型参数要在函数名后使用 `<>` 声明
- 命名惯例使用大驼峰命名，第一个类型参数一般用 `T`，接着是 `U`、`V` 等

```
fn function_name<T>(param: T) -> T {
    let variable: T;
}
```

示例

```
fn swap<T, U>(tuple: (T, U)) -> (U, T) {
    (tuple.1, tuple.0)
}
fn main() {
    let tuple = (1, "hello");
    let tuple1 = ("hello".to_string(), "world".to_string());
    let tuple2 = (3.14, 98);

    let t_swap = swap(tuple);
    let t1_swap = swap(tuple1);
    let t2_swap = swap(tuple2);

    println!("{}: {}", t_swap);
    println!("{}: {}", t1_swap);
    println!("{}: {}", t2_swap);
}
```

编译器自动推导出了具体的类型，编译器会根据函数定义，选择调用对应的函数版本。

编译器通常能从参数类型推导出对应的类型参数，但如果无法推导出具体类型，就需要显式指定类型

```
function_name::<type,...>(arg,...)
```

示例

```
fn do_something<T: Default>() -> T {
    let value: T = T::default();
    value
}

fn main() {
    let ret = do_something::<i8>();
    println!("{}", ret);
}
```

泛型约束

在编译期，编译器对这些参数类型的实际类型并不了解，因此，编译器需要对使用类型参数的代码添加合理的限制。可以把类型参数看作一个装着某种工具的黑盒子，你不知道里面具体是什么，但是你想安全地执行一些操作，这个时候会看有关于盒子内容的提示会非常有用。trait约束就是用来限制类型参数的行为，可以限定单个或多个trait，每个trait都会告诉编译器类型参数应该具备哪些能力。

使用():用于添加trait约束，如 <T: Trait>

```
use std::fmt::Display;

fn do_something<T: Display>(t: T) {
    println!("{}", t);
}

fn main() {
    do_something(20);
}
```

因为格式化字符串中的{}占位符需要实现Display trait,编译器不能假设类型参数T具有此特性。

可以给类型参数指定多个限制

```
<T: Trait1+Trait2+...>
```

示例

```

use std::cmp::Ordering, fmt::Display;

fn largest<T: Ord+Display>(arg1: T, arg2: T) {
    match arg1.cmp(&arg2) {
        Ordering::Less => println!("{} < {}", arg1, arg2),
        Ordering::Greater => println!("{} > {}", arg1, arg2),
        Ordering::Equal => println!("{} == {}", arg1, arg2),
    }
}

fn test() {
    largest(1, 2);
    largest("arg1", "arg2");
    largest('a', 'b');
    largest(100, 99);
}

fn main() {
    test();
}

```

largest是一个泛型函数，用于比较并输出结果。要支持比较，这个类型必须实现Ord trait; 使用占位符{},要求这个类型必须实现Display,这两个trait都被应用与T的约束。

where

where是约束的另一种表示方法，这种方法的表达力更强

```

use std::fmt::Debug;

fn do_something<T>(t: T)
where T: Debug {
    println!("{}?", t);
}
fn main() {
    let tuple = (1, 2, 3, 5);

    do_something(tuple);
}

```

泛型结构体

除了函数之外，结构体同样支持泛型，在定义结构体时，可以在结构体名称后面加上 `<T>` 之后该参数可以被用于结构体定义的任何位置(字段和方法)

```

struct Wrapper<T> {
    internal: T
}
fn main() {
    let obj = Wrapper {internal: 24};

    /* 会被单态化成i32类型
       struct wrapper {
           internal: i32
       }
    */
}

```

由于单态化机制，编译器会用i32类型实例化Wrapper结构体

泛型结构体不仅可以使用类型参数定义字段，还可以将类型参数用于方法定义。

```

use std::fmt::Debug;

#[derive(Debug)]
struct Wrapper<T> {
    internal: T
}

impl<T: Copy> Wrapper<T> {
    fn get(&self) -> T {
        self.internal
    }
}

fn main() {
    let obj = Wrapper {internal: 20};
    let obj1 = Wrapper {internal: "hello".to_string()};

    let ret = obj.get();
    println!("{}", ret);
}

```

只有字段类型实现了Copy的T，才能调用该impl块内的所有方法。由于set方法需要创建T类型的拷贝，所以需要T实现Copy trait。而String类型不属于实现了Copy trait的类型，因此实例化后不具备get方法。

除了从结构体那里继承的类型参数，方法也可以定义专属于自己的类型参数。

```
use std::fmt::{Debug, Display};

#[derive(Debug)]
struct Wrapper<T> {
    internal: T
}

impl<T: Copy+Display> Wrapper<T> {
    fn display<U: Display>(&self, prefix: U, suffix: U) {
        println!("{} {} {}", prefix, self.internal, suffix);
    }
}

fn main() {
    let obj = Wrapper { internal: 20 };
    obj.display("(,)");
}
```

关联函数

你可以将泛型与关联函数一起使用

```
use std::fmt::{Debug, Display};

#[derive(Debug)]
struct Wrapper<T> {
    internal: T
}

impl<T: Display> Wrapper<T> {
    fn hello(name: T) {
        println!("hello {}", name)
    }
}

fn main() {
    Wrapper::hello("张三");
}
```

由于hello方法传入了一个&str类型的参数，编译器可以推断出T的类型，因此就不需要手动指定。而下面这种情况需要手动指定类型：

```
use std::fmt::{Debug, Display};

#[derive(Debug)]
struct Wrapper<T> {
    internal: T
}
impl<T> Wrapper<T> {
    fn hello() {
        println!("helloworld");
    }
}

fn main() {
    Wrapper::hello();
}
```

枚举

枚举类型参数需要在枚举名称后的 `<>` 内声明，之后在定义每个枚举的变体(内部字段)时，都需要在 `()` 内指明具体应用了哪些类型参数

```
enum Option<T> {
    Some<T>
    None,
}
```

由于Option和Result这两个enum没有包含函数返回的全部情况，我们可以自定义一个新的枚举类型

```

enum Repeat<T, U> {
    Continue(U),
    Some(T),
    None,
}

fn do_something(number: i32) -> Repeat<i32, i32> {
    if number <= 0 {
        Repeat::None
    } else {
        Repeat::Continue(number - 1)
    }
}
fn main() {
    let mut number = 100;
    loop {
        if let Repeat::Continue(value) = do_something(number) {
            println!("{}", value);
            number = value
        } else {
            break;
        }
    }
}

```

也可以对泛型枚举中使用的类型参数进行约束

```

#[derive(Debug)]
enum Emp<T: Clone> {
    Id(T),
    Name(String),
}

fn get_emp() -> Emp<String> {
    Emp::Id("001".to_string())
}

fn main() {
    let emp = get_emp();
    println!("{:?}", emp)
}

```

这样，在创建实例时就必须满足。

泛型trait

泛型可以实现类型的抽象，而trait则用于代码的抽象。适当结合这两种技术，可以发挥泛型编程的最大潜力。

```
use std::fmt::Display;

struct XStruct<T, U> {
    field1: T,
    field2: U,
}

trait ATrait<T> {
    fn do_something(&self, arg: T);
}

impl<T: Display, U> ATrait<T> for XStruct<T, U> {
    fn do_something(&self, arg: T) {
        println!("{} {}", self.field1, self.field2);
    }
}

fn main() {
    let xstruct = XStruct{ field1: "张三", field2: "李四" };

    xstruct.do_something("王五");
}
```

通用扩展trait

它可以为所有类型提供一个统一的trait实现，需要组合泛型和trait才能获得的特殊功能。

chapter_13

模式在Rust中主要有两个用途

- 实现数据的模式匹配和解构，先看看这个数据长什么样子（是否符合某个结构），如果符合，就把它“拆开”，把里面的各个部分拿出来单独使用
- 程序流程控制，match、while、if等都可以根据模式是否与值匹配来决定程序的执行路径。模式匹配可以用于标量值，也可以用于符合数据类型(结构体、枚举、数组等)

模式由一系列规则和符号构成，由于描述一个值的结构。模式匹配可用于变量绑定、表达式、函数参数、返回值等场合。

let语句

let通过模式匹配机制将一个值绑定到新变量上，如果模式就是单个变量名的形式，那么它就属于不可反驳模式，即不存在匹配失败的情况。

```
fn main() {
    let a = 3;
    let b = (1,3);
    println!("{}a");
    println!("{}b:?")
}
```

以下是更复杂的例子

匹配元组

```
fn main() {
    let (a,b) = (1,2);
    println!("{} {}");
}
```

匹配数组

```
fn main() {
    let list = [1, 2, 3, 4];
    let [a, b, c, d] = list;
    println!("{} {} {} {}", a, b, c, d);
}
```

对现有变量进行模式匹配

```
fn main() {
    let mut a = 0;
    let mut b = 0;
    (a, b) = (1, 2);
    println!("{} {}", a, b)
}
```

通配符

下划线(_)用于忽略某个对应的值

```
fn main() {
    let list = [1, 2, 3, 4];
    let [a, _, _, d] = list;
    println!("{} {}", a, d)
}
```

双点号(..)用来忽略一部分连续值

```
fn main() {
    let list = [1, 2, 3, 4];
    let [a, .., d] = list;
    println!("{} {}", a, d)
}
```

复杂模式

可以使用更复杂的模式来对复合数据类型的值进行完全解构。

```
fn main() {
    let data = ((1, 2), (3, 4));
    let (a, b) = data;
    println!("{} {}", a, b)
}
```

我们还可以进一步对元组进行解构

```
fn main() {
    let data = ((1, 2), (3, 4));
    let ((a,b), (c,d)) = data;
    println!("{} {} {} {}", a, b, c, d)
}
```

使用模式移除引用语义

```
fn main() {
    let ref1 = &13;
    let (&data) = ref1;
    println!("{}", ref1);
}
```

所有权

在使用模式解构时，所有权可能会发生转移。

- 对于实现了Copy语义的类型，所有权不会转移
- 对于实现了Clone语义的类型，解构操作会导致所有权被转移给新的变量绑定。

```
fn main() {
    let tuple = ("Bob".to_string(), 13);
    let (a,b) = tuple;
    println!("{}", tuple.0); //会报错
    println!("{}", tuple.1);
    println!("{} {}");
}
```

由于字符串支持Clone语义，而整数实现了Copy语义，因此tuple.0的所有权被转移了，而tuple.1的所有权没被转移。解决办法是使用ref关键字，它会在模式匹配时创建对该值的引用

```
fn main() {
    let tuple = ("Bob".to_string(), 13);
    let (ref a,b) = tuple;
    println!("{}", tuple.0); //会报错
    println!("{}", tuple.1);
    println!("{} {}");
}
```

mut关键字可以出现在模式中，在解构一个值时，默认是不可变的，但可以用mut关键字声明可变的绑定。

```
fn main() {
    let (mut a, b) = ("bob".to_string(), 13);
    a = "joe".to_string();
    println!("{}");
}
```

可以在模式中组合ref和mut来声明一个可变引用。此时Rust的可变性规则(同一时间只能存在一个可变引用)依然适用

```
fn main() {
    let mut tuple = ("bob".to_string(), 13);
    let (ref mut a, b) = tuple;
    a.push('!');
    println!("{}" ,tuple.0);
    println!("{}" ,tuple.1);
}
```

当一个值没有实现Copy语义时，通过解构就会发生所有权转移。但是，如果在模式中使用`_`忽略该值，所有权就不会发生转移。

```
fn main() {
    let tuple = ("bob".to_string(), "def".to_string());
    let (_, b) = tuple;
    println!("{}" ,tuple.0);
    println!("{}" ,tuple.1); //报错了
}
```

但是，当我们忽略模式绑定到的变量`_variable_name`时，情况就不一样了。

```
fn main() {
    let tuple = ("bob".to_string(), "def".to_string());
    let (_ , _b) = tuple;
    println!("{}" ,tuple.0); // 被忽略了(未移动)
    println!("{}" ,tuple.1); // 被绑定了(被移动)
}
```

- 由于tuple.0被忽略了`(_)`，没有进行任何绑定，所以所有权没有转移；
- 对于tuple.1,虽然后面使用`_b`进行了绑定，而且`_b`并没有使用，但tuple.1的值仍然被绑定到`_b`上，所有权发生了转移。

不可反驳模式

在Rust中，模式分为可反驳和不可反驳两种。

- 不可反驳是全面的，可以匹配任何情况，因此始终会匹配成功
- 可反驳只能匹配一部分特定的表达式，也可能完全不匹配，因此你必须提供另一条控制分支以处理无法匹配的情况。`let`语句只接受不可反驳模式

```
fn main() {
    let a = 10;
    println!("{}" ,a);
}
```

试图用`let`语句处理可反驳模式是行不通的。

```
fn main() {
    let option = Some(12);
    let Some(value) = option;
    println!("{}", value);
}
```

let语句本身没有为匹配失败提供可替代方案。如果匹配失败，就必须提供后备计划，以指导编译器正确地编译。

范围模式

范围模式(模式中使用范围)属于可反驳模式。范围模式只适用于数值类型和字符类型。

- `begin..=end` -> 表示 `[begin, end]` 这个范围
- `begin..` -> 表示 `[begin, 最大值]` 这个范围
- `..=end` -> 表示 `[最小值, end]` 这个范围

示例

```
fn main() {
    let tuple = get_value();
    if let (1..=10, 1..=10) = tuple {
        println!("匹配成功")
    } else {
        println!("匹配失败")
    }
}

fn get_value() -> (i8, i8) {
    (10, 10)
}
```

前面在匹配成功时只输出一条信息，如果能知道匹配的具体值就好了，使用@语法可以将变量绑定到模式范围

`binding@range`

示例

```

fn main() {
    let tuple = get_value();
    if let (a@..=10, b@1..=10) = tuple {
        println!("匹配成功({a}, {b})")
    } else {
        println!("匹配失败")
    }
}

fn get_value()->(i8, i8) {
    (10, 10)
}

```

多个模式

你可以使用管道符(|)将多个模式组合起来

```

fn main() {
    let value = get_value();
    if let 10|12|30 = value {
        println!("匹配成功");
    } else {
        println!("匹配失败");
    }
}

fn get_value()->i8 {
    12
}

```

在复合模式中匹配一个值，想要知道匹配的具体是哪个值，我们同样可以使用@

```

fn main() {
    let value = get_value();
    if let a@(10|12|30) = value {
        println!("匹配成功 {a}");
    } else {
        println!("匹配失败");
    }
}

fn get_value()->i8 {
    12
}

```

控制流

if-let-else

前面我们使用了if-let-else这样的形式。这里的模式是可反驳的。

```
if let pattern=expression {
    // 匹配成功的代码块
} else {
    // 不匹配的代码块(可选的)
}
```

示例

```
fn main() {
    let value = get_value();
    if let Some(v) = value {
        println!("匹配的值为{}", v)
    }
}

fn get_value()->Option<i8> {
    Some(23)
}
```

while-let

也可以在while let表达式中使用模式。

- 只要模式匹配到值，while循环就会一直执行，直到模式不能匹配到值时终止。
- 通过模式匹配成功的值，只在while循环体内有效

```
fn main() {
    let vec = vec![1,2,3,4,5];
    let mut i = 0;
    while let Some(item) = vec.get(i) {
        println!("{}[{}]", item, i);
        i+=1;
    }
}
```

for-in

for表达式的实现基于迭代器和模式匹配机制，在每次迭代时，它会调用next方法。

- 只要next返回Some(item)，迭代就会继续执行

- 一旦返回None，迭代器就会终止。

```
fn main() {
    let data = [1,2,3,4,5];

    for (index, item) in data.iter().enumerate() {
        println!("index {index}; item {item}")
    }
}
```

结构体

模式匹配也用于结构体，对结构体进行解构赋值非常有用。对于结构体模式，至少包含结构体名称和正确的字段名(字段顺序无关紧要)，在默认情况下，结构体字段会被绑定到同名变量上。

```
struct Rectangle {
    p1: (u32, u32),
    p2: (u32, u32),
}
fn main() {
    let r = Rectangle {
        p1: (1, 2),
        p2: (3, 4),
    };
    let Rectangle {p1, p2} = r; //解构
    println!("{} {} {} {}", p1, p2);
}
```

我们还可以进一步解构

```
struct Rectangle {
    p1: (u32, u32),
    p2: (u32, u32),
}
fn main() {
    let r = Rectangle {
        p1: (1, 2),
        p2: (3, 4),
    };
    let Rectangle {p1: (a,b), p2:(c, d)} = r; //解构
    println!("{} {} {} {}", a, b, c, d);
}
```

在解构结构体时，还可以通过field_name:binding_name规则进行重命名

```

struct Rectangle {
    p1: (u32, u32),
    p2: (u32, u32),
}
fn main() {
    let r = Rectangle {
        p1: (1, 2),
        p2: (3, 4),
    };
    let Rectangle {p1: top_left, p2:bottom_right} = r; //解构
    println!("p1: {:_?}; p2: {:_?}", top_left, bottom_right);
}

```

字面量对模式进行了细化，只有满足字面量的值的模式才能匹配。通过这种方式为模式匹配添加额外条件。由于else分支的存在，该模式是可反驳的。

```

struct Rectangle {
    p1: (u32, u32),
    p2: (u32, u32),
}
fn main() {
    let r = Rectangle {
        p1: (1, 2),
        p2: (3, 4),
    };
    if let Rectangle {p1: top_left, p2:(3, 4)} = r {
        println!("p1: {:_?}", top_left);
    } //解构
}

```

你也可以在解构模式时使用(_)通配符来忽略一个字段值。使用(..)通配符来忽略任何剩余的值。

函数

模式匹配也可以用在函数参数上，不过只能使用不可反驳模式。

```

fn do_something((x, y): (u8,u8)) {
    println!("{x} {y}");
}

fn test() {
    do_something((1,2));
    do_something((2,3));
    do_something((3,4));
}
fn main() {
    test();
}

```

match表达式

match表达式与模式的结合被广泛用于控制流，match的每个分支左侧对应一个模式，当被测试的值与该模式匹配时，该分支就会被执行。匹配的模式必须穷尽所有可能的情况，如果没有，就需要添加一个默认模式(_)来匹配默认情况。因为默认模式是不可反驳的，所有它应该作为最后一个分支。

```
fn main() {
    let value = get_value();
    match value {
        1=>println!("One"),
        2=>println!("Two"),
        _=>println!("Unknown"),
    }
}
fn get_value()->i8 {
    12
}
```

匹配守卫

匹配守卫是模式匹配的过滤器，是一个布尔表达式，

- 如果匹配守卫为假，该模式会被过滤掉
- 如果匹配守卫为真，模式匹配会正常进行

```
struct Rectangle {
    p1: (u32, u32),
    p2: (u32, u32),
}

fn main() {
    let rect = Rectangle {
        p1: (1, 2),
        p2: (3, 4),
    };

    match rect {
        Rectangle {
            p1: (x1, _),
            p2: (x2, _),
        } if x1 > x2 => println!("1.{x1} {x2}"),
        Rectangle {
            p1: (x1, _),
            p2: (x2, _),
        } => println!("2.{x1} {x2}"),
        _ => println!("default"),
    }
}
```

我们将模式匹配实现为一个函数，该函数接受一种颜色参数，并判断颜色是不是灰色。

```

#[derive(Debug)]
struct RgbColor{
    red: i32,
    blue: i32,
    green: i32,
}

#[derive(Debug)]
struct CmykColor {
    cyan: i32,
    magenta: i32,
    yellow: i32,
    black: i32,
}

#[derive(Debug)]
enum Colors {
    RGB(RgbColor),
    CMYK(CmykColor),
}

impl Colors {
    fn is_gray(&self) -> bool {
        match self {
            Colors::RGB(color) => (color.red==color.green)==(color.green==color.blue),
            Colors::CMYK(color) => (color.cyan+color.magenta+color.yellow)==0,
        }
    }

    fn display_gray(&self) {
        match self {
            Colors::RGB(value) if self.is_gray() => println!("RGB {:?} is gray", value),
            Colors::CMYK(value) if self.is_gray() => println!("CMYK {:?} is gray", value),
            Colors::RGB(value) => println!("RGB {:?} is not gray", value),
            Colors::CMYK(value) => println!("CMYK {:?} is not gray", value),
        }
    }
}

fn main() {
    let rgb_color = RgbColor { red: 255, blue: 255, green: 255 };
    let rgb_color = Colors::RGB(rgb_color);

    rgb_color.display_gray();

    let cmyk_color = CmykColor { cyan: 0, magenta: 0, yellow: 0, black: 0 };
    let cmyk_color = Colors::CMYK(cmyk_color);
    cmyk_color.display_gray();
}

```

当一个模式由多个模式组合而成时，匹配守卫会应用到所有的模式。只有当模式被匹配且匹配守卫为true时，整个模式才会被匹配。

```
fn is_monday() -> bool {
    true
}

fn main() {
    match 1 {
        1 | 11 | 21 if is_monday() => println!("1 | 11 | 21"),
        _ => println!("Not Monday!"),
    }
}
```

chapter_14

闭包是可以捕获外部环境中的变量的匿名函数。它所引用的外部变量被称为自由变量(这些变量本不属于闭包函数的作用域范围)。与普通函数类似，闭包可以执行代码逻辑、接收参数输入，并返回值。相对于普通函数有如下优势:

- 如果只需要在一个地方使用某个函数，闭包是更简便的选择。
- 闭包是一等公民，可以将闭包作为函数参数、返回值，甚至赋值给变量。
- 闭包一般定义在使用它的附近，这使得代码的可维护性更好。

闭包可以实现Fn、FnMut、FnOnce三个不同的trait,分别对应不同的行为。需要注意的是，这里Fn和定义函数指针类型使用的fn是不同的。绝大多数情况下，编译器根据上下文自动推断闭包应该实现哪个trait. 需要注意的是

1. 闭包通常不需要声明返回值类型，编译器自动上下文推导出来。
2. 闭包不是嵌套函数，因为嵌套函数无法捕获定义在它们外层函数作用域中的变量。这也是闭包和嵌套函数的区别。
3. 嵌套函数有显式的函数名，而闭包则是匿名的

闭包语法

```
|parameter_1, .., parameter_n| -> return_type {
    代码块
}
```

let value = 10; 示例

```
fn main() {
    //计算立方体
    let cubed = |number: usize| -> usize {
        number * number * number
    };

    let value = 10;
    let ret = cubed(value);
    println!("{}" , ret);
}
```

以上代码可以简化为以下形式

```
fn main() {
    let cubed = |number| number*number*number;

    let value = 10;
    let ret = cubed(value);
    println!("{}", ret);
}
```

捕获变量

闭包可以捕获自由变量，被捕获的变量在闭包内部可用。最常见的情况，捕获的变量是对自由变量的借用。

```
fn main() {
    let value = 23;
    let cubed = || value*value*value;
    let ret = cubed();
    println!("{}", ret);
}
```

上述代码中，闭包捕获的变量是被借用的。

示例

```
fn main() {
    let mut values1 = (13, 20);

    // 借用开始
    let swap_values = || (values1.1, values1.0);

    let values2 = &mut values1;
    let ret = swap_values(); //借用结束

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

闭包对values1执行了借用，这个借用会一直持续，直到闭包被调用后。而values2变量在那个范围内声明为可变借用，而Rust不允许在同一个变量上同时存在可变和不可变的借用，这会导致编译错误。

```

fn main() {
    let mut values1 = (13, 20);

    // 借用开始
    let swap_values = || (values1.1, values1.0);
    let mut ret = swap_values(); //原借用结束位置
    let values2 = &mut values1;
    ret = swap_values(); //借用被延长后的结束位置

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

```

这段代码中，swap_values被调用两次，这将最初的借用范围扩展到第二次调用swap_values()之后。在这个范围内尝试对该自由变量执行可变借用，将无法编译。

修改如下

```

fn main() {
    let mut values1 = (13, 20);

    // 借用开始
    let swap_values = || (values1.1, values1.0);
    let mut ret = swap_values(); //原借用结束位置

    ret = swap_values(); //借用被延长后的结束位置

    println!("{:?}", ret);
    let values2 = &mut values1;
}

```

闭包作为函数参数

作为函数参数，需要使用impl关键字指定Fn trait,还必须提供函数定义。

闭包可能实现了Fn、FnMut、FnOnce trait，现在我们将专注于Fn trait。

```

fn do_closure(run: impl Fn()) {
    run();
}

fn main() {
    let display = || println!("Message");
    do_closure(display);
}

```

更复杂的例子

```

enum Calculation {
    Cubed,
    Quad,
}

fn get_retsult(run: impl Fn(i32) -> i32, value: i32) -> i32 {
    run(value)
}

fn main() {
    let cubed = |value: i32| value * value * value;
    let quad = |value: i32| value * value * value * value;
    let calculation = Calculation::Cubed;
    let ret = match calculation {
        Calculation::Cubed => get_retsult(cubed, 5),
        Calculation::Quad => get_retsult(quad, 5),
    };
    println!("{}", ret);
}

```

闭包作为返回值

与参数一样，`impl`关键字可用于指定一个闭包trait,比如Fn trait.

```

fn get_closure()->impl Fn(i32)->i32 {
    |number| number*number*number
}

fn main() {
    let cubed = get_closure();
    let ret = cubed(4);
    println!("{}", ret);
}

```

更复杂的例子

```

enum Calculation {
    Cubed,
    Quad,
}
fn get_closure(calculation: Calculation)->impl Fn(i32)->i32 {
    match calculation {
        Calculation::Cubed => |value: i32| value*value*value,
        Calculation::Quad => |value: i32| value*value*value*value,
    }
}

fn main() {
    let calculation = Calculation::Cubed;
    let fun = get_closure(calculation);

    let ret = fun(4);
    println!("{} {}", ret);
}

```

闭包的实现

Rust闭包在编译时会被转换为结构体，而捕获的变量会成为该结构体的字段。闭包函数实际上是闭包结构体的一个方法，像其他方法一样，第一个参数是self。闭包结构体内没有对闭包函数的引用，因为没有必要。闭包方法的属性(self参数的定义)，取决于编译期间编译器选择实现的是Fn、FnMut还是FnOnce。

```

fn main() {
    let (a, b) = (1, 2);
    let adder = |prefix: String| println!("{} {} {}", prefix, a+b);

    adder("Add: ".to_string());
}

```

被转换的结构体可能的形式

```

struct adder {
    a: i32,
    b: i32,
}

impl Fn<(String)> for adder {
    type Output = ();
    fn call(&self, args: Arg)->Self::Output {
        //这里省略细节
    }
}

```

这三个trait之间存在层次关系。实现闭包时，优先选择Fn,其次是FnMut,最后是FnOnce。

FnOnce



FnMut



Fn

例如，FnMut trait是Fn trait的父trait,这意味着在需要Fn trait的地方可以使用FnMut trait代替，而反过来却不行。

Fn trait

闭包如果是不可变的，则实现了 Fn trait，这意味着捕获的变量也必须是不可变的，对于 Fn trait，闭包方法的 self 是 &Self，捕获的变量则是借用语义。

```
fn do_closure(closure: impl Fn()) {
    closure()
}
fn main() {

    let hello = || println!("hello");
    do_closure(hello);
}
```

hello 闭包没有捕获任何变量，意味着这个闭包是不可变的，并实现了 Fn trait。

对上面代码稍作修改

```
fn do_closure(closure: impl Fn()) {
    closure()
}
fn main() {
    let hello_string = "hello".to_string();
    let hello = || println!("{}", hello_string);
    do_closure(hello);
}
```

现在 hello 闭包捕获了 hello_string，捕获的变量是不可变的，因此编译器自动为不可变上下文实现了 Fn trait

```
fn do_closure(closure: &mut impl FnMut()) {
    closure()
}

fn main() {
    let mut hello_string = "hello".to_string();
    let mut hello = || {hello_string.push_str("world"); println!("{}",
    hello_string)};

    do_closure(&mut hello);
}
```

上述代码中，hello_string 是可变的，变量仍然可以作为自由变量在 hello 闭包中被捕获。闭包现在有一个可变的状态，并且自动实现 FnMut。do_closure 函数参数更改为 FnMut 来确认该闭包实现了 FnMut。

一个没有捕获任何外部变量的闭包，实际上就等同于一个标准函数。因此，标准函数和无捕获上下文的闭包是可以相互转换的。这就是为什么 Fn trait 也适用于标准函数。

```
fn do_closure(closure: impl Fn()) {
    closure()
}

fn main() {
    fn hello() {
        println!("hello")
    }
    do_closure(hello);
}
```

FnMut trait

FnMut trait适用于那些捕获了可变上下文的闭包，对于实现了FnMut trait的闭包，其方法中的self参数类型是&mut Self。

```
fn do_closure(mut closure: impl FnMut()) {
    closure()
}

fn main() {
    let mut value = 0;
    let increment = || value=value+1;
    do_closure(increment);
    println!("{}", value);
}
```

FnOnce trait

FnOnce trait适用于那些只能被执行一次的闭包。实现了FnOnce trait的闭包就只能被执行一次。对于FnOnce trait,闭包方法的self参数类型是Self,即闭包获取了被捕获变量的所有权。这激素为什么FnOnce闭包只能被调用一次。

```
fn do_closure(closure: impl FnOnce()->String) {
    closure();
    closure(); //报错
}

fn main() {
    let hello_string = "hello".to_string();

    let hello = || hello_string;

    do_closure(hello);
}
```

在下面闭包中，捕获的字符串被丢弃，由于同一个字符串只能被丢弃一次，编译器可以识别这一点，并为闭包实现FnOnce trait。

```
fn do_closure(closure: impl FnOnce()) {
    closure();
}

fn main() {
    let hello_string = "hello".to_string();
    let hello = || drop(hello_string);
    do_closure(hello);
}
```

由于FnOnce是FnMut和Fn的subtrait,因此任何实现了Fn/FnMut的闭包都可以作为FnOnce实例使用。

```
fn do_closure(closure: impl FnOnce()) {
    closure();
}

fn main() {
    let hello_string = "hello".to_string();
    let hello = || println!("{}", hello_string);
    do_closure(hello);
}
```

尽管hello实现了Fn,由于绑定到closure这个FnOnce变量，因此在do_closure中只能被执行一次。

```
fn do_closure(closure: impl FnOnce()) {
    closure();
    closure(); //编译错误
}

fn main() {
    let hello_string = "hello".to_string();
    let hello = || println!("{}", hello_string);
    do_closure(hello);
}
```

move关键字

每个函数在运行时都会在栈上分配一个私有的内存区域(栈帧),用于存储函数的局部变量和寄存器状态等上下文信息。闭包可以从外部函数的栈帧中捕获变量值作为自由变量引用。由于这一机制，闭包并不完全拥有它们所使用的环境。这可能会导致一些问题。

```
fn get_closure() -> impl Fn() -> i32 {
    let a = 10;
    let b = 20;
    || a+b // 编译错误
}

fn main() {
    let fun = get_closure();
    fun();
}
```

get_closure函数返回一个依赖于捕获值a,b的闭包，这就产生了对外部函数环境的依赖。但是当闭包返回时，外部函数的栈帧会被销毁。

解决办法是使用move关键字，使用move关键字的闭包会获取去捕获环境的完整所有权，不再依赖于外部函数的栈帧。被捕获的变量值会被转移到闭包的环境中，根据值的类型决定采用移动或复制语义。

```
fn get_closure() -> impl Fn() -> i32 {
    let a = 10;
    let b = 20;
    move || a + b // 编译错误
}

fn main() {
    let fun = get_closure();
    let ret = fun();
    println!("{}", ret);
}
```

impl 关键字

trait本身是不确定大小的，因此我们无法直接创建trait的实例。要使用trait，可以通过静态分派和动态分派的方式管理具体实现了该trait的类型实例。impl关键字就是为闭包显式指定实现的trait。impl关键字并非适用于所有场景，如变量绑定的情况。例如

```
type closure_tuple = (impl Fn(), impl Fn(),)
```

或是

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

fn main() {
    let map = HashMap<i32, impl Fn()>::new();
    map.insert(0, || println!("hello"));
}
```

这种情况下，我们可以考虑使用静态分派和动态分派

静态分派

```
struct AStruct<T>
where T: Fn() {
    hello: T,
}
fn main() {
    let astruct = AStruct { hello: || println!("hello")};
    (astruct.hello)()
}
```

动态分派

```
fn main() {
    let hello: &dyn Fn() = &| | println!("hello");
    hello();
}
```

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

type Row = (char, i32, i32, i32);
type OperationType<'a> = &'a dyn Fn(Row)->i32;

fn main() {
    let mut matrix = vec![
        ('a', 4, 5, 0),
        ('m', 2, 6, 0),
        ('d', 9, 3, 0),
        ('s', 5, 6, 0),
    ];
    let mut operation: HashMap<char, OperationType>;
    operation = HashMap::new();
    operation.insert('a', &|row| row.1+row.2);
    operation.insert('m', &|row| row.1*row.2);
    operation.insert('d', &|row| row.1/row.2);
    operation.insert('s', &|row| row.1-row.2);

    for each_row in matrix.iter_mut() {
        each_row.3 = operation.get(&each_row.0).unwrap()(each_row);
    }
    println!("{:?}", matrix)
}
```

chapter_15

静态分发

相较于具体类型，采用trait作为函数和返回值类型可以使代码更具扩散性，也更加简洁。只要是实现了相同的trait的类型，在该trait的语境都是可以互换使用的，不局限于特定类型。因此可以将trait看作实现了相应行为的任何具体类型的占位符。

```
trait ATrait {  
}  
fn do_something(obj: ATrait) {  
}  
  
fn main() {  
}
```

这个例子无法编译，因为trait是UnSized的，不能用来创建实例。要解决这个问题，我们采用静态分发和动态分发。静态分发在编译时将trait解析成具体类型，然后编译器为这个特定类型创建一个函数的特化版本，这称为单态化，能够通过减少运行时消耗来提高性能，但是会导致代码膨胀。单态化的前提是能够在编译时能够识别具体类型。

使用impl关键字来定义静态分发

```
fn do_something(obj: impl ATrait) {  
}
```

impl可以与函数参数使用，但不能与变量绑定使用

示例

```
trait Human {
    fn get_name(&self);
}

#[derive(Debug)]
struct Adult(String); // 成人

impl Human for Adult {
    fn get_name(&self) {
        println!("{}: {:?}", self);
    }
}

#[derive(Debug)]
struct Child(String);

impl Human for Child {
    fn get_name(&self) {
        println!("{}: {:?}", self);
    }
}

// 外星人
trait Alien {
    fn get_name(&self);
}

// 火星人
#[derive(Debug)]
struct Martian(String);

impl Alien for Martian {
    fn get_name(&self) {
        println!("{}: {:?}", self);
    }
}

fn invite_to_paty(attendee: impl Human) {
    attendee.get_name();
}

fn main() {
    // 创建成年人实例
    let bob = Adult("Bob".to_string());

    // 创建儿童实例
    let janice = Child("janice".to_string());

    // 创建外星人实例
    let fred = Martian("Fred".to_string());

    invite_to_paty(bob);

    invite_to_paty(janice);

    invite_to_paty(fred); // 不允许外星人参加
}
```

动态分发

有时编译器无法推断出具体类型，这种情况下需要使用动态分发。对于动态分发，解决方案是使用引用，因为引用具有固定大小。不过普通引用所携带的信息还不够，为此，Rust将dyn关键字和引用组合起来提高了trait对象。它在运行时被初始化为两个指针，一个指向具体类型实例，另一个指向trait的实现。有几种方法声明trait对象

- 使用dyn关键字

```
&dyn trait
```

- 使用Box创建trait对象

```
Box<dyn trait>
```

示例

```
trait Human {
    fn display_name(&self);
}

#[derive(Debug)]
struct Adult(String); // 成人

impl Human for Adult {
    fn display_name(&self) {
        println!("{}: {:?}", self);
    }
}

#[derive(Debug)]
struct Child(String);

impl Human for Child {
    fn display_name(&self) {
        println!("{}: {:?}", self);
    }
}

// 外星人
trait Alien {
    fn display_name(&self);
}

// 火星人
#[derive(Debug)]
struct Martian(String);

impl Alien for Martian {
    fn display_name(&self) {
        println!("{}: {:?}", self);
    }
}

fn invite_to_patty(attendee: impl Human) {
    attendee.display_name();
}

fn create_person(adult: bool, name: String) -> Box {
    if adult {
        Box::new(Adult(name))
    } else {
        Box::new(Child(name))
    }
}

fn main() {
    let bob = create_person(true, "Bob".to_string());
    bob.display_name();

    let janice = create_person(false, "janice".to_string());
    janice.display_name();
}
```

动态分发还可以用于变量绑定

```
struct Rectangle;
struct Ellipse;

trait Shape {
    fn draw(&self) {
        println!("draw");
    }
}

impl Shape for Rectangle {}

impl Shape for Ellipse {}

fn main() {
    let shapes: Vec<&dyn Shape> = vec![&Rectangle{}, &Ellipse{}];
    for shape in shapes {
        shape.draw();
    }
}
```

枚举和trait

枚举类型同样可以实现trait,可以自由选择实现方式。不过有一种最佳实践: 在为枚举实现trait时 , 应当使用match表达式 , 针对每个枚举变体分别提供一种唯一的trait实现。

```
trait Schemes {
    fn get_rgb(&self)->(u8, u8, u8);
    fn get_cmyk(&self)->(u8, u8, u8, u8);
}

enum CoreColor {
    Red,
    Green,
    Blue,
}

impl Schemes for CoreColor {
    fn get_rgb(&self)->(u8, u8, u8) {
        match self {
            CoreColor::Red => (255, 0, 0),
            CoreColor::Green => (0, 255, 0),
            CoreColor::Blue => (0, 0, 255),
        }
    }

    fn get_cmyk(&self)->(u8, u8, u8, u8) {
        match self {
            CoreColor::Red => (0, 99, 100, 0),
            CoreColor::Green => (100, 0, 100, 0),
            CoreColor::Blue => (98, 59, 0, 1),
        }
    }
}

fn main() {
    let red = CoreColor::Red;
    let red_rgb = red.get_rgb();
    let red_cmyk = red.get_cmyk();
    println!("{:?}", red_rgb);
    println!("{:?}", red_cmyk);
}
```

chapter_16

在Rust中，每个进程最初只有一条主执行路径，被称为主线程。main函数执行的就是主线程。我们可以创建额外的线程，以实现并行执行任务或操作。Rust中的线程本质上就是操作系统线程或物理线程，Rust中的线程与操作系统线程存在一一对应的关系，这与一些语言的绿色线程(M: N线程模型)不同，绿色线程将多个逻辑线程(M)调度到少数个物理线程(N)上运行。并行和并发变成是引入多线程模型的两个重要原因。

- 并行编程旨在将一个进程拆分为多个并行操作，以提升整体性能。
- 并发致力于提供系统的响应能力。例如在执行排序等计算密集型操作时，仍然能保持用户界面的响应。

你可能会觉得既然2个线程可以提升性能，那10个线程效果岂不是更好？然而事实并非如此。决定是否带来性能改善的因素有很多，比如与操作系统相关的因素。一旦线程数量超过某个临界点，反而会导致性能下降。因为存在数据依赖和线程运行时开销(如上下文切换)，所以无法做到完全并行。

在Rust中，无畏并发的设计消除了并发变成的多种顾虑(数据竞争等)。

在一个进程这个空间中，每个线程也拥有私有资源。其中值得关注的是线程栈，用于维护该线程的局部变量、系统调用信息以及其他信息。在某些环境下，线程的默认栈大小可能是2MiB，对于拥有数十甚至上百个线程而言，栈空间的总占用是一笔相当大的内存开销。Rust为开发者提供了管理线程栈大小的机制，可以帮助你更好地管理内存占用。

相比于单线程，多线程的管理更加复杂

- 竞态条件，多个线程竞争贡献资源的情况。
- 死锁，一个线程无限期地等待另一个线程或资源变得可用。
- 不一致性，多线程应用程序如果实现不当会表现出不一致性。尽管增加了复杂性，多线程仍然是创建可扩展、响应快、高性能应用程序的一个重要工具。

同步函数调用

```
fn hello() {
    println!("hello, world");
}

fn main() {
    println!("In main");
    hello();
    println!("Back in main");
}
```

每个函数都有局部变量，这些变量被放置在保存线程状态的栈上。

```

fn display() {
    let b = 2;
    let c = 3.4;
    println!("{} {}", b, c);
}

fn main() {
    let a = 1;
    println!("{}", a);
    display();
}

```

每个函数都会获得一块专门的存储区域，被称为栈帧，用于保存自己的私有数据。随着同步函数调用的不断深入，新的栈帧被持续压入栈中，导致栈的空间持续增长。当函数执行完毕后，对应的栈帧会从栈中移除。

函数	局部变量	栈帧
main	val_a	0
display	val_b val_c	1

线程

Rust标准库中的thread模块提供了线程相关的功能。

- `thread::spawn` 该函数只接收一个参数(函数/闭包)作为新线程的入口点，用于创建并立即启动一个新线程。

```

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

```

需要注意的是

1. `spawn`返回一个`JoinHandle`, 用于线程同步和获取线程入口函数的返回值。

2. F是入口函数的类型参数，T是线程返回值的类型参数。
3. Send约束了该值可以安全地跨线程传递。
4. 'static生命周期是必须的，因为我们无法预知线程何时启动和结束，新线程的生命周期可能超过父线程，因此T和F都要求具有静态生命周期。
5. 父线程只是一个比喻性的说法，实际上这两个线程之间没有关系。

多线程示例

```
use std::thread;

fn main() {
    let thread = thread::spawn(|| println!("hello"));
    println!("In main");
}
```

main函数和闭包在独立的线程中同时运行。这个例子会出现不稳定行为：

- 如果main函数率先完成，则程序退出，包括终止其他正在运行的线程。闭包的问候信息来不及显示
- 这两个线程的执行顺序是不确定的，当你多次运行代码，结果可能会不同。问候信息可能显示也可能不显示。

spawn函数会返回一个JoinHandle,我们可以用其join方法让当前线程等待，直到由它管理的线程执行完毕后才继续执行。这种等待一直持续到关联的线程被分离,例如被丢弃。

```
use std::thread;

fn main() {
    let thread = thread::spawn(|| println!("hello"));

    println!("In main");
    let ret = thread.join(); // 等待线程结束
    println!("Break in main");
}
```

有时候可能需要获取线程的执行结果，也可以用JoinHandle,其join方法会阻塞当前线程，直到与其关联的线程执行完毕，join方法的返回值就是该线程的返回值(Ok(value))。

```
use std::thread;

fn main() {
    let t = thread::spawn(|| 1);
    let ret = t.join();
    println!("{}: {}", ret, ret.unwrap());
}
```

如果一个正在运行的线程没有成功完成执行，例如遇到了panic

- 对于主线程(通常是main函数)，那么整个程序将终止。
- 对于非主线程，线程将在栈展开后简单地终止，但其他线程将会继续执行。

如果该线程在join列表中，那么join函数会返回Err结果，作为对发生panic的通知。

```
use std::thread;

fn main() {
    let handle = thread::spawn(|| panic!("kaboom"));
    let ret = handle.join();

    match ret {
        Ok(value) => println!("{}" , value),
        Err(msg) => println!("{}" , msg),
    }
}
```

当线程由闭包创建时，可以通过捕获变量将数据传递给线程。这是线程常见的输入来源。然而线程异步方式运行，不受父线程作用域的限制。新线程的存在时间可能比父线程更长。因此为了避免数据所有权的问题，需要使用move关键字将捕获的数据所有权转移到闭包中。

```
use std::thread;

fn main() {
    let a = 10;
    let b = 20;
    let handle = thread::spawn(move || {
        let c = a + b;
        println!("result: {}", c)
    });

    let result = handle.join();
    println!("{}" , result.unwrap());
}
```

作用域线程消除了普通线程使用捕获变量的一些限制。更重要的，作用域线程的生命周期是确定的。它不会比创建它的代码块(作用域)存活更久，因此不需要move关键字。创建作用域线程需要使用thread::scope函数。该函数接收一个作用域对象作为参数，该对象定义了作用域线程的生存范围(作用域)。

```
fn scope<'env, F, T>(f: F)->T
where F: for<'scope> FnOnce(&'scope &Scope<'scope, 'env>)->T,
```

类型参数F将作用域对象描述为函数，T描述其返回值类型，'scope指作用域对象的生命周期。'env用于任何借用值。因此'scope的生命周期不长与'env。

```
use std::thread;

fn main() {
    let mut count = 0;
    thread::scope(|s|{
        s.spawn(|| count+=1);
    });
    println!("{}",count);
}
```

thread类型

Thread是线程的句柄类型，同时是一种不透明类型。我们无法直接创建Thread类型的实例，只能通过工厂函数创建，也就是thread::spawn或Builder::spawn函数间接创建。绝大多数情况下，线程本身不持有自己的句柄。拥有自身句柄将允许线程对自己进行管理。幸运的是，Rust提供了Thread::current()函数，可以获取代表当前线程的句柄。有了这个句柄，就可以调用各种方法操作或查询线程的状态了。

```
use std::thread;

fn main() {
    let thread_current = thread::current();
    let id = thread_current.id();
    let name = thread_current.name();
    println!("id: {} name: {}", id, name);
}
```

- thread::id()返回一个ThreadId类型，这是一直不透明的类型，代表当前线程所运行的进程的唯一标识符。当一个线程终止后，它的ThreadId不会重复使用。
- thread::name()返回Result枚举，类型为&str，默认的名称是该线程的入口函数名。如果线程是通过闭包创建的，则没有线程名(None)

```
use std::thread;

fn main() {
    let thread_current = thread::current();
    let id = thread_current.id();
    let name = thread_current.name();
    println!("id: {} name: {}", id, name);

    let result = thread::spawn(|| {
        println!("In thread2");
        let curr = thread::current();
        let id = curr.id();
        let name = curr.name();
        println!("{} {}", id);
        println!("{} {}", name);
    })
    .join();
}
```

CPU执行时间

- 并发运行的线程会共享CPU的执行时间。不同操作系统环境采用了不同的线程调度算法。大多数现代操作系统使用抢占式调度，线程也可以主动调用`thread::yield_new`函数主动让出剩余的时间片。这种方式更加友好。
- 我们还可以要求某个线程在指定时间段内睡眠，在睡眠期间，该线程不会活动CPU执行时间。这种做法通常是为了实现线程执行的同步和协调，也有可能是因为当前线程暂时没有可做的工作。`thread::sleep`函数会强制线程至少休眠指定的时间。`Duration`类型提供了一些函数允许以不同精度指定睡眠时间长度。

1. `duration::as_micros`: microseconds 读取`dur`的总微秒数
2. `duration::as_millis`: milliseconds 读取`dur`的总毫秒数
3. `duration::as_nanos`: nanoseconds 读取`dur`的总纳秒数
4. `duration::as_secs`: seconds 读取`dur`的总秒数

```
use std::{thread, time::Duration};

fn main() {
    for (name, dur) in [("T1", 30), ("T2", 40)] {
        thread::spawn(move || {
            let mut n = 1;
            while n < 5 {
                println!("{} {}", name, n);
                n += 1;
                thread::sleep(Duration::from_millis(dur));
            }
        });
    }

    thread::sleep(Duration::from_secs(3));
}
```

各种操作系统在底层为阻塞同步(即自旋锁)提供了支持。在自旋锁机制中，线程会次序"自旋"执行一段无谓的循环代码，耗费CPU时间，直到所需的同步资源变为可用状态。当资源竞争程度不高时，自旋锁往往比其他同步机制(如互斥和信号量)更高效。

- `thread::park`函数启动一个自旋锁，并有效阻塞当前线程。每个线程都关联一个令牌。`park`函数会让线程阻塞直到该令牌变为可用状态或超时。
- `unpark`方法可以解除线程的阻塞状态(释放自旋锁)，
- `thread::park_timeout`函数会让线程阻塞指定的时长，如果超时前线程没有被`unpark`,则该线程会自动唤醒。

```

use std::{thread::{self, Thread}, time::Duration};

fn main() {
    let open = store_open();

    //准备工作
    disable_alarm();
    open_registers();

    open.unpark();

    thread::sleep(Duration::from_millis(10));
}

fn disable_alarm() {

}

fn open_registers() {

}

fn store_open() -> Thread {
    thread::spawn(|| {
        thread::park();
        loop {
            println!("开店营业中");
        }
    })
    .thread()
    .clone()
}

```

线程Builder

线程有两个可配置的属性: 线程名称和栈大小。线程名称用字符串表示，栈大小则需指定以字节为单位的值。可以通过Builder类型配置这些属性，配置完成后调用builder::spawn方法生成新线程，它会返回一个 `Result<JoinHandle<T>>` 类型的结果

- Builder::name函数用于设置线程的名称。为线程指定合理的名称方便调试。
- Builder::stack_size用于设置单个线程的栈空间大小。线程的初始栈大小由操作系统决定的，我们可以为除主线程外的任何线程设置栈大小，主线程的栈大小取决于运行环境。可以通过stack_size设置单个线程的栈大小，也可以通过RUST_MIN_STACK环境变量来统一指定所有新线程的默认栈大小。合理管理栈大小不仅可以提升性能，还可以减少进程的内存占用。

由于name()和stack_size()返回的都是Builder类型，因此可以链式调用。

```

use std::result, thread::{self, Builder}, time::Duration;

fn main() {
    let builder = Builder::new().name("Thread1".to_string()).stack_size(4096);
    let result = builder.spawn(|| {
        let thread = thread::current();
        println!("id:{} name:{}",&thread.id(), &thread.name());
    });

    let handle = result.unwrap();
    let result = handle.join();
}

```

通信顺序进程

通信顺序进程(CSP)理论，为线程编程定义了一种新颖的模型，在该模型中1,线程直接通过实现FIFO队列语义的异步消息传递对象来进行通信。CSP要求线程之间通过消息传递对象交换信息，而非共享内存。在Rust中，通信是线程之间的传输管道，它有两个部分: 发送者和接收者。发送者通过通道发送消息，接收者从管道接收消息。通道接收多生产这单消费者模型(每个通道有一个接收者，但可以有多个发送者)

注意，发送者和接收者是同一管道的两端，如果任何一方变得无效，则通过通道的通信将无效。

支持线程同步的工具在标准库std::sync模块中，包括互斥、锁和通道。其中通道有各种类型。

- Sender: 异步通道
- SyncSender: 同步通道

异步通道

异步通道没有大小限制，理论上可以无限存储数据。发送数据到通道时，发送者永远不会阻塞，但同时也无法确定接收者何时真正从通道获取数据，可能是立即获取，也可能永不获取。只有当接收者试图从空通道读取数据时，通道才会阻塞。使用mpsc::channel函数创建一个异步通道

```
fn channel<T>() -> (Sender<T>, Receiver<T>)
```

此函数返回一个包含通道双端的元组 (Sender<T>, Receiver<T>)。类型参数T指定了可以通过该通道传输的数据类型。

- Sender使用Sender::send函数将数据插入通道，如果需要多个发送端，则可以克隆Sender。
- Receiver使用Receiver::recv函数从通道读取数据。以下是异步通道的重要方法

```
fn Sender::send(&self, t: T)->Result<(), SendError<T>>
fn Receiver::recv(&self)->Result<T, RecvError>
```

示例

```
use std::{sync::mpsc, thread};

fn main() {
    let (sender, receiver) = mpsc::channel();

    thread::spawn(move || {
        sender.send("hello");
    });

    let data = receiver.recv().unwrap();
    println!("{}" , data);
}
```

该示例中，mpsc::channel函数返回一个包含Sender和Receiver的元组，代表通道的两端。在另一个线程中调用send方法向通道发送"hello"信息，在主线程中调用recv方法从通道中接收数据。recv方法会一直阻塞到有数据插入通道。

```

use std::sync::mpsc, thread;

fn main() {
    let (sender, receiver) = mpsc::channel();
    let sender1 = sender.clone();
    let sender2 = sender.clone();

    thread::spawn(move || {
        for i in 0..5 {
            sender.send(i);
        }
    });
    thread::spawn(move || {
        for i in 10..15 {
            sender1.send(i);
        }
    });
    thread::spawn(move || {
        for i in 20..25 {
            sender2.send(i);
        }
    });
}

let handle = thread::spawn(move || {
    while let Ok(data) = receiver.recv() {
        println!("data: {}", data);
    }
});

handle.join();
}

```

如果通道的任何一端断开连接，该通道将变得无法使用。这种情况发生在通道的Sender或Receiver被丢弃时，你将无法向该通道继续插入数据，但你仍然可以从通道中读取剩余的数据。

```

use std::sync::mpsc, thread, time::Duration;

fn main() {
    let (sender, receiver) = mpsc::channel();

    thread::spawn(move||{
        sender.send(1);
    });

    let data = receiver.recv();
    println!("{}:?", data);

    thread::sleep(Duration::from_secs(3));
    let data = receiver.recv();
    println!("{}:?", data.unwrap());
}

```

我们创建了一个异步通道，在向通道发送一个整数后，Sender端很快就被丢弃了，此时整个通道立即失效。你可以接收之前插入的那个整数(1)，但当再次尝试从通道中接收数据就会引发panic，因

为通道此时已经失效且为空。

同步通道

与异步通道不同的是，同步通道的大小是有界限的。在某些场景下，受限的通道大小反而更有益处，例如在实现消息队列时，我们可能希望限制队列中的消息数量以提高效率，这时同步通道更合适。使用mpsc::sync_channel函数创建同步通道：

```
fn sync_channel<T>(bound: usize) -> (SyncSender<T>, Receiver<T>)
```

- bound参数用于设置通道的最大容量，通道中的项目数量不能超过这个限制。
- 返回值是一个元组，包含同步通道的发送者(SyncSender)和接收者(Receiver)。
- 使用SyncSender::send函数向通道发送数据。如果同步通道已满，则send函数会阻塞到另一个线程接收数据，给通道腾出空间。
- 与异步通道的Recevier相同，使用Receiver::recv方法从通道接收数据，通道为空时会阻塞。

```
fn send(&self, t: T) -> Result<(), SenderError<T>>
```

示例

```
use std::{sync::mpsc, thread};

fn main() {
    let (sender, receiver) = mpsc::sync_channel(1);
    let handle = thread::spawn(move || {
        sender.send(1);
        println!("Sent 1");
        sender.send(2);
        println!("Sent 2");
    });

    let data = receiver.recv().unwrap();
    println!("data:{}", data);
    handle.join();
}
```

由于容量的限制(1)，只有第一个数据发送成功，直到使用recv取出一个数据项，这就为第二个数据项插入留下了空间。虽然第二个数据项被发送到通道，但是它从未被接收和消费。对于某些应用程序，可能会发生问题。

rendezvous通道

rendezvous通道提供了数据可靠传输的保证，从而解决了上述问题--如何确定通道中的数据何时被成功接收。rendezvous通道实际上是一个容量为0的同步通道。对于这种通道，

SyncSender::send函数是阻塞的，只有当发送的数据被接收者取走后，该函数才会解除阻塞，可以把它视为一种可靠传输的通信机制。

示例

```
use std::sync::mpsc, thread, time::Duration;

fn main() {
    let (sender, receiver) = mpsc::sync_channel(0);
    let handle = thread::spawn(move || {
        sender.send(1);
        println!("数据已接收");
    });

    thread::sleep(Duration::from_secs(10));

    let data = receiver.recv().unwrap();
    println!("{}: {}", data);
    handle.join();
}
```

try方法

试图向一个已经满了的通道继续发送数据时，发送者会被阻塞，既然这样，优势可能更倾向于先收到一个通知而不是直接阻塞，这就需要用到try_send方法。如果通道已满，则该方法返回TrySendError作为通知。以下是方法的定义

```
fn try_send(&self, t: T) -> Result<(), TrySendError<T>>
```

示例

```
use std::sync::mpsc;

fn main() {
    let (sender, receiver) = mpsc::sync_channel(2);

    sender.send(1);
    sender.send(2);

    let err = sender.try_send(3).unwrap_err();

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

接收者同样会被阻塞！当通道为空时调用recv方法时会进入阻塞状态，直到有发送者向通道插入数据，一旦有数据项，接收者就会解除阻塞状态并直接从通道取走那个数据项。try_recv提供了一种非阻塞的替代方案，当通道为空时try会返回一个Err结果，这样接收者线程将不会被阻塞而顺利执行。方法定义如下：

```
fn try_recv(&self) -> Result<T, TryRecvError>
```

try_recv方法的一个有效使用场景是执行空闲任务，线程原本是会被阻塞的，有了try_recv，就可以利用这段时间执行其他工作：

- 分阶段完成工作。资源清理就是一个很好的例子。通常资源清理工作较为耗时，一般需要在应用程序结束时执行。但我们利用空闲时间提前做部分清理工作可以减轻程序结束时的工作量。
- 非常适合处理低优先级或可选的任务。这些任务可以在没有其他更重要的事情需要处理时执行，例如执行用户界面处理程序。

```
use std::sync::mpsc;
use std::thread::Builder;

fn main() {
    let (sender, receiver) = mpsc::sync_channel(10);

    let builder = Builder::new().name("发送者".to_string()).stack_size(4096);
    let result = builder.spawn(move || {
        let messages = [
            "message 1".to_string(),
            "message 2".to_string(),
            "message 3".to_string(),
        ];
        for message in messages {
            sender.send(message);
        }
    });

    let builder = Builder::new().name("接收者".to_string()).stack_size(4096);
    let result = builder.spawn(move || {
        loop {
            match receiver.try_recv() {
                Ok(msg) => {
                    if msg.len() == 0 {
                        break 0;
                    }
                    println!("{}: {}", msg);
                },
                Err(_) => idle_work(),
            }
        }
    });

    let handle = result.unwrap();
    handle.join();
}

fn idle_work() {
    println!("idle_work");
}
```

recv_timeout函数是recv的另一个变体。recv_timeout函数会在管道为空时阻塞，但是当超过指定时间时，recv_timeout函数会被唤醒并返回RecvTimeoutError作为Err结果，以下是该函数定义

```
fn recv_timeout(&self, timeout: Duration) -> Result<T, RecvTimeoutError>
```

示例

```
use std::{sync::mpsc, thread, time::Duration};

fn main() {
    let (sender, receiver) = mpsc::sync_channel(10);

    thread::spawn(move || {
        thread::sleep(Duration::from_millis(200));
        sender.send(1);
    });
    let data = receiver.recv_timeout(Duration::from_millis(100));
    match data {
        Ok(value) => println!("接收到数据: {}", value),
        Err(_) => println!("Time out没有接收到数据"),
    }
}
```

也可以用迭代器的方式从一个通道接收数据项。迭代器可以扩大通道的用例，使其更具有扩展性。使用iter方法可以获取迭代器，然后可以使用next方法来访问通道的数据项。

```
use std::sync::mpsc;

fn main() {
    let (sender, receiver) = mpsc::sync_channel(10);

    sender.send(1);
    sender.send(2);
    sender.send(3);

    let mut iter = receiver.iter();
    println!("{}", iter.next().unwrap());
    println!("{}", iter.next().unwrap());
    println!("{}", iter.next().unwrap());
}
```

使用迭代器让通道更具有扩展性。你甚至可以使用for-in来迭代一个通道，这是因为Receiver实现了Iterator接口，当通道内没有数据项或通道变得无效时，for循环会停止迭代。

```
use std::{sync::mpsc, thread};

fn main() {
    let (sender, receiver) = mpsc::sync_channel(10);
    let sender1 = sender.clone();
    let handle = thread::spawn(move || {
        sender.send(1);
        sender.send(2);
    }); //发送者被丢弃
    for item in receiver {
        println!("{}", item);
    }
}
```

商店示例

```

use std::{
    sync::mpsc::{self, Receiver, Sender, SyncSender, channel},
    thread,
    time::Duration,
};

/// 使用channel函数创建一个异步通道来通知商店即将关闭
/// 当一个消息被发送到该通道时，商店应该被关闭，因此接收者被命名为(closing)
/// store_open函数负责管理商店的开启，closing通道作为其唯一参数，在函数内创建一个独立的线程
/// 来处理闭店操作。
/// 开店的准备工作(关闭报警系统和打开收银机)可以并行执行，因为这些任务被作为单独的线程启动，并
/// 使用join方法等待准备工作完成。
/// 准备工作完成后，商店就可以开门了！之后的循环代表处理顾客事件的过程。
/// 在循环内通过使用match表达式调用closing的try_recv方法来检查商店是否应该关闭，如果
/// closing通道接收到数据，那就是要关闭商店，进入ok分支执行关闭操作，否则进入默认分钟继续接待顾客
fn main() {
    let (sender, closing) = channel::<()>();

    store_open(closing);

    thread::sleep(Duration::from_secs(2));

    store_closing(sender);
}

fn store_open(closing: Receiver<()>) {
    thread::spawn(move || {
        //营业准备
        let alarms = thread::spawn(|| {
            // TODO 关闭报警系统
            println!("关闭报警系统");
        });
        let registers = thread::spawn(|| {
            // TODO 打开收银机
            println!("打开收银机");
        });
        alarms.join();
        registers.join();

        //开始营业
        loop {
            match closing.try_recv() {
                Ok(_) => {
                    break;
                }
                Err(_) => {
                    // TODO 处理用户事件
                    println!("正在营业");
                    thread::sleep(Duration::from_secs(2));
                }
            }
        }
    })
}

```

```
});  
}  
  
/// store_closing函数，接收一个Sender作为参数，第一步向closing通道发送一个值，这里选择空元组，通知store_open函数开始进行闭店操作(即停止接待客人)  
/// .sleep函数给了store_open中的线程一个机会清理然后退出。  
/// 创建单独的线程来重新启动报警器和关闭收银机。等它们完成，商店就关闭了  
fn store_closing(sender: Sender<()>) {  
    sender.send(());  
    thread::sleep(Duration::from_secs(2));  
  
    let alarms = thread::spawn(|| {  
        // TODO 打开报警系统  
        println!("打开报警系统");  
    });  
    let registers = thread::spawn(|| {  
        // TODO 关闭收银机  
        println!("关闭收银机");  
    });  
  
    alarms.join();  
    registers.join();  
    println!("闭店了");  
}
```

chapter_17

在很多方面，进程内的线程就像住在一个房子里的多个家庭成员。他们必须共享资源，并且常常会无意中竞争这些资源。如果没有适当的协调，那么这可能会导致冲突和不可预测的行为。同样有时也需要线程同步。join和通道都为线程之间的协调提供了帮助。

增加必要的同步有助于创建一个安全的环境，从而促使更多并行化。你可以自信地增加并行化层，而无需担心线程之间的冲突。然而，过度同步可能会增加应用程序的复杂性，降低性能。并行编程本质上比顺序变成更复杂。当问题出现时，人民更倾向于增加同步以获得更可预测的结果。当增加同步来解决问题的情况反复出现时，会产生大量的技术债务。最终你将得到一个在并行应用程序外壳下运行的、本质上是顺序执行的程序。虽然同步往往是合理的，有时甚至是必要的，但请记住要保持并行程序的并行性。在你第一个程序"Hello World"中，你可能体验到了线程同步。println!宏通常用于显示问候语，它通过内部互斥保证了安全。如果没有这种同步，则多个线程可能会同时使用println!宏，从而导致不可预测的结果。

互斥

mutex是(mutual exclusion (互相排斥))的缩写，互斥是最知名的同步原语。它提供了对共享数据的互斥访问。

- 共享数据访问可能会导致不可预测的结果，尤其是数据可变的情况下。通过互斥，你可以防止线程同时访问共享数据。互斥可以保证对数据的顺序访问，保护共享数据。
- 互斥体可以被锁定或解锁。当被锁定时，互斥体强制执行互斥的并发策略。拥有锁的线程可以独占访问数据。同时另一个线程(或多个线程)在尝试获取已经锁定的互斥体是会被阻塞。当互斥体解锁时，等待的线程可能会获得锁。如果成功，被阻塞的线程将被唤醒，并且可以访问共享数据。
- 互斥体具有线程亲和性。当它被锁定时，必须由同一个线程来解锁。这可以防止其他线程窃取对互斥体的访问。想象一下那种混乱，任何想要访问互斥体的线程，都可以简单地解锁它并访问被保护的值，后果将不堪设想。幸运的是，在Rust中这是被阻止的，因为这里没有解锁函数。在许多语言中，互斥的使用和源码中函数的正确放置有关。当访问被保护的数据时，你必须使用互斥体的lock和unlock方法将其包围起来。因此，同步的正确性完全基于程序员的自律性--将互斥体放在正确的位置。这在重构过程中可能会成为更大的问题，因为被保护的数据或任何互斥体可能会被无意间移动或删除。由于这些原因，Rust采取了不同的方法，并且将被保护的数据与互斥体关联。这种直接的关联防止了其他语言中发生的问题。

Mutex互斥体

- Mutex类型是互斥体的一种实现。它位于std::sync模块中，与其他同步组件一样，Mutex是泛型的，其中T代表被保护的数据。

- 你可以用Mutex::new构造函数创建一个新的互斥体，该构造函数也是泛型(T)的，唯一的参数是被保护的数据，函数定义如下：

```
fn new(t: T) -> Mutex<T>
```

Mutex::lock 函数用于锁定 Mutex 并独占受保护的数据。如果 Mutex 处于解锁状态，那么你将获取锁并继续执行。当 Mutex 已经锁定时，当前(尚未获取到锁)的线程将被阻塞直到可以获取锁。

```
fn lock(&self) -> LockResult<MutexGuard<'_, T>>
```

lock 函数返回一个 MutexGuard。它实现了 Deref(解引用) trait，从而提供内部值(即受保护的数据)的访问。MutexGuard 确保当前线程可以安全地访问数据。重要的是，当 MutexGuard 被释放时，Mutex 会自动解锁，这就是 Rust 不需要解锁函数的原因。

```
use std::sync::Mutex;

fn main() {
    let mutex = Mutex::new(0);
    let mut guard = mutex.lock().unwrap();

    *guard += 1;
    println!("{}", *guard);
    // 解锁互斥体
}
```

示例

```
use std::{sync::Mutex, thread};

fn main() {
    let mutex = Mutex::new(0);
    thread::scope(|s| {
        for count in 1..=2 {
            s.spawn(|| {
                let mut guard = mutex.lock().unwrap();
                *guard += 1;
                println!("当前:{}? Data:{}",
                    thread::current().id(), *guard);
            });
        }
    });
}
```

你可能会在无意中导致互斥体泄漏。因为互斥体在 MutexGuard 的生命周期内保持锁定状态，如果 MutexGuard 从未被释放，或只是延迟释放，那该互斥体将对其他线程不可用，这可能导致死锁。这是由多种原因引起的，包括对 MutexGuard 的管理不良。

```

use std::{sync::Mutex, thread, time::Duration};

fn main() {
    let mut hello = String::from("hello");
    let mutex = Mutex::new(&mut hello);
    {
        let mut guard = mutex.lock().unwrap();
        guard.push_str(", world");
        // 做一些耗时的事情，其他线程会暂停等待
        thread::sleep(Duration::from_secs(10));
    } // 解锁互斥体

    thread::scope(|s| {
        s.spawn(|| {
            println!("进入这里");
            let mut guard = mutex.lock().unwrap();
            guard.push_str("小王");
        });
    });
}

```

下面的示例几乎是相同的代码，区别是MutexGuard没有与变量进行绑定。这意味着MutexGuard是临时的，并在下一行代码中会被释放，此时互斥体会被解锁，我们无需等到代码可的结尾才解锁互斥体。

```

use std::{sync::Mutex, thread, time::Duration};

fn main() {
    let mut hello = String::from("hello");
    let mutex = Mutex::new(&mut hello);

    (*mutex.lock().unwrap()).push_str(", world");
    // 保护被丢弃，解锁互斥体
    // 做一些耗时的事情
}

```

你也可以显式地释放MutexGuard来解锁互斥体

```

use std::sync::Mutex, thread, time::Duration;

fn main() {
    let mut hello = String::from("hello");
    let mutex = Mutex::new(&mut hello);
    {
        let mut guard = mutex.lock().unwrap();
        guard.push_str(", world");
        drop(guard); //显式释放，解锁互斥体
        do_something();
    }

    thread::scope(|s| {
        s.spawn(|| {
            let mut guard = mutex.lock().unwrap();
            guard.push_str(" ! ");
        });
    });

    println!("{}", hello);
}

fn do_something() {
    thread::sleep(Duration::from_secs(3));
}

```

非作用域互斥体

你可以与非作用域线程共享一个互斥体，而Arc(原子类型计数)类型正是以这种方式共享互斥体的最佳解决方案。多线程应用程序往往需要共享所有权(多个线程共享数据的所有权)。Arc类型支持共享所有权，并通过引用计数来追踪所有者的数量。当最后一个共享所有者(即线程)退出后时，计数器降至0,此时共享数据被释放。引用计数是以原子的方式进行的，以防止竞争条件或引用计数被破坏。Arc位于std::sync模块中。可以通过Arc::new来创建一个新的Arc,它的唯一参数是共享数据。

```
fn new(data: T) -> Arc<T>
```

你可以克隆(clone)以与其他线程共享。每次克隆，引用计数都会增加。此外，Arc实现了Deref trait,以提供对内部值的访问。

```
use std::sync::Arc, thread;

fn main() {
    let arc_orig=Arc::new(0);
    let arc_clone = arc_orig.clone();
    let handle = thread::spawn(move ||{
        println!("Thread2 {}", arc_clone); //Deref
    });

    println!("Thread1 {}", arc_orig);
    handle.join();
}
```

上面代码中，主线程为一个整数值创建了一个Arc,然后克隆一个Arc并增加引用计数，克隆的Arc随后被移动到另一个线程。现在两个线程共享着数据，println!宏会自动解引用Arc以显示底层值。因为值是共享的，所以两个线程显示相同的结果。随着共享Arc数量的增加，命名可能会成为问题。导致Arc被命名为arc1、arc2、...。一个更好的解决方案是通过变量遮蔽在各个线程使用相同的名称。

```
use std::sync::Arc, thread;

fn main() {
    let arc = Arc::new(0);
    { // 新代码块
        let arc = arc.clone();
        let handle = thread::spawn(move || {
            println!("{}", arc); // Deref
        });
        handle.join();
    } // 代码块结束
    println!("{}",arc);
}
```

需要注意的是Arc只提供了共享所有权的引用计数，但它不是一个Mutex。Arc不会对数据访问进行同步，然而，Arc非常适合与非作用域共享Mutex。Arc共享Mutex,而Mutex保护数据。

示例 在本示例中，Mutex保护一个整数值(初始值为0),Mutex通过Arc类型的变量arc_mutex进行共享。创建了一个vec来存储for循环生成的handle

```

use std::{
    sync::{Arc, Mutex},
    thread,
};

fn main() {
    let arc_mutex = Arc::new(Mutex::new(0));
    let mut handles = vec![];
    for i in 0..=2 {
        let arc_mutex = Arc::clone(&arc_mutex);
        let handle = thread::spawn(move || {
            let mut guard = arc_mutex.lock().unwrap();
            *guard += 1;
            println!("{} {}", guard);
        });
        handles.push(handle);
    }

    for handle in handles {
        handle.join();
    }
}

```

for循环生成新线程，每个线程捕获arc_mutex的克隆版本。然后arc_mutex以同步对整数值的访问。如果获得了锁，则返回MutexGuard,它被解引用以访问内部值，然后该值递增。在随后的for循环中，对每个JoinHandle调用join等待

互斥体中毒

当一个线程在锁定互斥体时发生panic，并释放MutexGuard时，该互斥体就会中毒，底层的状态是不确定的，因此尝试锁定该互斥体会返回一个错误(PoisonError)。互斥体中毒会强制应用程序认识到潜在的问题。你也可以自行决定如何处理互斥体中毒的问题，当然你也可以选择忽略它，不过后果自负。

锁定一个中毒的互斥体会返回一个Result类型的Err,具体来说是PoisonError。

PoisonError::io_inner函数返回中毒的互斥体的MutexGuard，通过它，你可以像往常一样访问底层数据。

```

use std::{
    sync::{Arc, Mutex},
    thread,
};

fn main() {
    let arc_mutex = Arc::new(Mutex::new(0));

    let arc_mutex_clone = arc_mutex.clone();
    let handle = thread::spawn(move || {
        let mut guard = arc_mutex_clone.lock().unwrap();
        *guard += 1;
        println!("子线程：已将值修改为 {}, 但马上要崩溃！", *guard);
        panic!("子线程崩溃！");
    });

    // 捕获 join 结果，防止 panic 影响主线程
    if let Err(e) = handle.join() {
        println!("主线程：检测到子线程发生崩溃：{:?}", e);
    }

    match arc_mutex.lock() {
        Ok(guard) => println!("主线程：成功获取锁，当前值为 {}", *guard),
        Err(poisoned) => {
            let guard = poisoned.into_inner();
            println!(
                "主线程：检测到互斥体中毒，但仍成功恢复访问，值为 {}",
                *guard
            );
        }
    }
}

```

互斥体还有一个try_lock函数。与lock不同的是，try_lock在互斥体已经锁定时不会阻塞。代码将继续执行，函数返回一个Err作为Result。这允许你在互斥体被锁定时做一些其他的事情。

读写锁

读写锁类似于互斥，用于保护数据。它允许多个读者同时访问数据，

- 写者对数据拥有独占访问权，可以修改数据
- 读者只能读取数据 RwLock实现了读写锁。保护对读者和写者的实现。
- 读者调用read方法来获取读者锁。如果成功，则返回RwLockReadGuard,作为底层值，锁会一直有效，直到RwLockReadGuard被释放。而如果存在活跃的写锁，read函数将会阻塞。

```

fn read(&self) -> LockResult<RwLockReadGuard<'_, T>>

```

- 写者通过RwLock::write函数获取写入锁，如果成功，则返回Result包裹的RwLockWriteGuard。而锁会在RwLockWriteGuard释放时解锁。如果存在未完成的读取锁

(读取锁未解锁)或另一个活跃的写入锁，则write函数会阻塞

```
fn write(&self) -> LockResult<RwLockWriteGuard<'_, T>>
```

读写锁可能会中毒，但仅限于写线程。当RwLockWriteGuard在panic期间被释放时，读写锁会变成中毒状态。此时read和write都将返回一个错误。如果有多个等待的写入锁，则获取锁的顺序是不可预测的。

```
use std::{
    sync::{Arc, RwLock},
    thread,
    time::Duration,
};

fn main() {
    let rwlock = RwLock::new(0);
    let arc = Arc::new(rwlock);
    let mut handles = vec![];

    for i in 1..=3 {
        let arc = arc.clone();
        let handle = thread::spawn(move || {
            let guard = arc.read().unwrap();
            println!("Read Lock {} Data {}", i, *guard);
            thread::sleep(Duration::from_millis(400));
            println!("Reader UnLock");
        });
        handles.push(handle);
    }

    for i in 1..3 {
        let mut guard = arc.write().unwrap();
        println!("Write Lock");
        *guard += 1;
        thread::sleep(Duration::from_millis(600));
        println!("Write UnLock");
    }

    for handle in handles {
        handle.join();
    }
}
```

RwLock也有try_read和try_write函数。这些函数是非阻塞的，如果锁不可用，则返回Err，但执行会继续。

条件变量

条件变量提供基于自定义事件的线程同步。有些语言称条件变量为事件。条件变量的语义是由你决定的，使得每个变量都具有独特性。因此，条件变量也被认为是自定义同步机制。当其他同步类型

不适用时，条件变量通常是最佳解决方案，因为它可以被定制。将互斥体和条件变量配合使用，可以用来提供锁机制。它们通常在一个元组中结合使用。这样可以防止条件变量不经意地与其关联的互斥锁解耦，避免使用错误的互斥锁。此外，条件变量有一个关联的布尔值，用以确认事件的状态。**布尔值应该由互斥体保护**。条件变量是Condvar类型，你可以用Condvar::new函数创建一个Convar.它不需要任何参数。

```
fn new() -> Condvar
```

为了等待一个事件，Condvar::wait函数会阻塞当前线程，直到收到事件通知。wait函数的唯一参数来自关联互斥体的MutexGuard,返回一个新的MutexGuard。因此，在调用wait函数之前，必须锁定互斥体。

注意: wait函数会解锁互斥体

```
pub fn wait<'a, T>(
    &self,
    guard: MutexGuard<'a, T>
) -> LockResult<MutexGuard<'a, T>>
```

Condvar::notify_one 和Condvar::notify_all函数将通知等待的线程事件已经发生或已完成。

- notify_one函数唤醒一个等待的线程，即使有多个线程正在等待。
- notify_all函数通知所有等待的线程

```
fn notify_one(&self)
fn notify_all(&self)
```

示例

```

use std::sync::{Arc, Condvar, Mutex}, thread, time::Duration;

/// * 声明了setup_event的Arc, 它内部将Condvar和Mutex配对。Mutex保护一个布尔值, 该值指示设置是否已完成
/// * 创建一个专用线程来执行设置, 接收一个setup_event元组的克隆, 在执行设置之前, 我们锁定Mutex并接收相关的锁保护(setup状态)
/// * 当设置完成后, 使用锁保护将setup_status更新为true。然后使用notify_one函数通知其他线程 设置已经完成。
/// * 锁定互斥体, 得到包含设置状态的MutexGuard
/// * wait函数阻塞线程并从互斥体释放锁。
/// * 线程将保持阻塞状态, 直到有通知表示事件已经发生。在本例子中, 该事件是设置已经完成
/// !在while循环中调用wait函数, 当从等待中唤醒时, 线程需要重新检查条件以确保没有发生虚假唤醒。
/// !如果发生虚假唤醒, 则条件保持不变, 线程应该继续等待适当的事件。这种检查对于设置事件来说可能没有必要。
/// !事件状态不太可能从假变为真(设置已完成到设置未完成), 然后再变为假。总的来说, 这是Condvar的最佳使用模式。
fn main() {
    let setup_event = Arc::new((Mutex::new(false), Condvar::new()));
    {
        let setup_event = setup_event.clone();
        thread::spawn(move||{
            let mutex = &setup_event.0;
            let cond = &setup_event.1;

            let mut setup = mutex.lock().unwrap();

            println!("Doing setup");
            thread::sleep(Duration::from_secs(2));
            *setup=true;
            cond.notify_one();
        });
    }
    let mutex = &setup_event.0;
    let cond = &setup_event.1;

    let mut setup = mutex.lock().unwrap();
    while !(*setup) {
        println!("Wait for setup to complete");
        setup=cond.wait(setup).unwrap();
    }
    println!("Main program started");
}

```

原子操作

Rust的基础数据类型中包含了全套原子类型。具体的原子类型清单在不同操作系统上会有不同。尽管涉及多个汇编级别的步骤，但原子操作是单个不可中断的步骤执行的。以防在跨线程共享操作

时发生数据损坏，或其他问题。使用原子类型可以将某些操作(如读和写)，作为一个单元执行。最重要的是，原子类型的实现并不包括锁，从而提升了性能。我们已经间接地使用了原子操作。例如Arc类型会原子性地增加引用计数。这样是为了更安全地修改引用计数。原子类型位于std::sync::atomic模块中

- AtomicBool
- AtomicI8、AtomicI16等
- AtomicU8、AtomicU16等
- AtomicPtr
- AtomicIsize和AtomicUsiz

存储和加载

所有原子类型的接口是一致的，这些接口用于存储(store)和加载(load)数据。此外他们的值可以通过共享引用修改。原子操作具有一个排序参数，该参数提供了对操作排序的保证。最不受限制的排序是Relaxed,它保证了单个变量的原子性。然而，它对于多个变量的相对顺序没有任何保证，例如由内存屏障提供的那些保证。load和store函数是原子类型的核心功能。store函数用于更新值，load函数用于获取值。

```
fn load(&self, order: Ordering) -> u8
fn store(&self, val: u8, order: Ordering)
```

示例

```

use std::time::Duration;
use std::sync::atomic::AtomicU8, thread;
use std::sync::atomic::Ordering::Relaxed;

fn do_something() {
    thread::sleep(Duration::from_millis(2000));
}

fn main() {
    static LOAD: AtomicU8 = AtomicU8::new(0);

    let handle = thread::spawn(|| {
        for i in 0..100 {
            do_something();
            LOAD.store(i, Relaxed);
        }
    });

    thread::spawn(|| {
        loop {
            thread::sleep(Duration::from_millis(2000));
            let value = LOAD.load(Relaxed);
            println!("Pct done {}", value);
        }
    });
    handle.join();
}

```

获取和修改

获取(fetch)和修改(modify)比加载和存储更复杂。有一些函数支持fetch和modify操作，包括fetch_add、fetch_sub、fetch_or、fetch_and等。使用两个线程计算一个累加总和，使用AtomicU32::fetch_add函数为例

```
fn fetch_add(&self, val: u32, order: Ordering)->u32
```

```

use std::sync::atomic::Ordering::Relaxed;
use std::time::Duration;
use std::{sync::atomic::AtomicU32, thread};

fn main() {
    static TOTAL: AtomicU32 = AtomicU32::new(0);
    let handle = thread::spawn(|| {
        for i in 0..100 {
            TOTAL.fetch_add(i, std::sync::atomic::Ordering::Relaxed);
        }
    });
    {
        let handle = thread::spawn(|| {
            for i in 100..200 {
                TOTAL.fetch_add(i, Relaxed);
            }
        });
        handle.join();
    }

    handle.join();
    println!("{}", TOTAL.load(Relaxed));
}

```

每个线程都对累加和做出了贡献，最后使用load方法获取总数。如果操作不是原子的，那么并行线程中的加法操作可能会破坏数据。

比较和交换

设想对个线程争相修改同一个值，第一个到达的线程会原子性地修改这个值。稍后到达的线程应该注意到这个变化，并且不应该再修改这个值。这种情景在并行编程相对常见，也是比较并交换操作的基础。

在比较和交换操作中，你需要指明一个预期值。

- 如果找到预期值，则当前值更新为一个新值，这就是交换。
- 然而如果没有找到预期值，就假定另一个线程已经修改了该值，当这种情况发生时，不应该再进行另一次交换。两个线程尝试更新一个AtomicU32类型，方法定义如下

```

pub fn compare_exchange(&self, current: u32, new: u32,
    success: Ordering, failure: Ordering) -> Result<u32, u32>

```

- current是预期值，如果current与当前值相匹配，原子类型AtomicU32的值会被更新为new参数的值。最后两个参数是独立Ordering类型参数
- 第一个Ordering参数用于交换操作
- 第二个Ordering参数在操作未交换时使用。
- 如果交换没有发生，则返回一个Err

```
use std::sync::atomic::AtomicU8, thread;

fn main() {
    use std::sync::atomic::Ordering::Relaxed;
    static TOTAL: AtomicU8 = AtomicU8::new(0);
    let handle1 = thread::spawn(|| {
        TOTAL.compare_exchange(0, 1, Relaxed, Relaxed)
    });

    let handle2 = thread::spawn(|| {
        TOTAL.compare_exchange(0, 2, Relaxed, Relaxed)
    });

    handle1.join();

    let ret = handle2.join();

    println!("{}:?", ret);
    println!("Value is {}", TOTAL.load(Relaxed));
}
```

内存

大多数应用程序，不论服务器、区块链、人工智能、游戏还是其他领域。都需要数据。因此理解内存的复杂性相当重要。不同类型的内存往往依赖于几个因素：

- 数据大小
- 所有权
- 生命周期
- 可变性
- 持久性

这些因素综合起来，将帮助你做出明确的决定。其中三个主要的内存区域是栈、堆和静态。你可以将数据放置在这些位置中的任何一个。有时，Rust会提供一些指引，比如将向量的元素放置在堆上。**然而主要还是你自己决定数据的位置。**

应该注意的是，Rust没有正式的内存管理模型(一套定义良好的规则和机制，用于自动管理内存的分配，使用和释放从而确保内存高效、安全地使用)。但是Rust的特性(例如默认不可变性、智能指针、所有权和生命周期)形成了一种非正式的内存管理模型。

栈

每个线程都拥有一个栈，它是一种专用内存。

- 当线程调用一个函数时，栈会增长。
- 当线程从函数返回时，栈会缩小。每个函数都有一个栈帧，它为函数保留内存。栈帧中的内存用于局部变量、参数、返回值和系统数据。这些数据会被系统自动释放。栈的实现是一个先进后出(LIFO)队列，类似于一碟盘子，新的盘子总放置在顶部，并按顺序从顶部移除。这种方式意味着数据高效地存储在连续的内存中。栈具有可预测的行为，因此系统能有效地管理栈。对于Rust，除主线程(main线程)外，默认栈大小是2K字节，当生成一个线程时，
- 你可以使用Builder类型和stack_size函数明确设置最小栈的大小。
- 或者使用RUST_MIN_STACK环境变量更改默认栈大小。

然而上面两种方法都没有设置栈大小上限，因为栈是可增长的，会在需要时扩展，直到达到可用内存容量的限制。let语句可用于当前栈帧内的内存中创建一个本地变量。

```
fn main() {
    let a = 1;
    let b = 2;
    let c = do_something();

    println!("{:p} + {}(i32) = {:p} + {}(i32) {:p}", &a, &b-&a, &b, &c-&b, &c);
}

fn do_something()->i32 {
    3
}
```

可以看到，本地变量a、b和c在栈上占据连续的内存位置

即使在函数内部，数据也可以被添加和从栈中移除。

```
fn main() {
    let a = 1;
    let b = 2;
    {
        let c = 3;

        println!("{} + {} = {}", a, b, c); //报错，因为c已经被移除了
    }
}
```

UnSized(?Sized)类型不能放在栈上。

```
fn do_something(a: Copy) {
```

因为Copy trait是一个UnSized的。编译器阻止了不符合条件的参数被放置到栈上。解决方法是结合dyn(dyn Copy)或impl(impl Dopy)关键字，这些关键字用具体类型替换了trait,它们是定长的。

注意事项

- 栈可能会消耗大量内存，因此在栈上放置大量对象时要小心。
- 另一个问题是在递归函数，不经意的无限递归会迅速耗尽可用内存。
- 一些数据类型(比如向量和字符串)是智能指针，当使用let语句声明时，这些类型的值会在堆上分配。指向该值的指针存放在栈上

```
fn main() {
    let vp = vec![1,2,3,4];
    println!("{:?}", vp);
}
```

上述例子中，vp是一个变量，代表胖指针，被放在栈上，而值 [1,2,3,4] 被放置在堆上

静态值

静态值在应用程序的生命周期内是持久的。这是通过将静态值存储在二进制文件本身来实现的。这种方式使得这些值始终可用。这也意味着大量的静态值会导致二进制文件膨胀，这可能会影响性能。此外，为了保证静态安全，静态值很少是可变的。可以使用static关键字声明静态绑定。静态值的名称应该全大写。此外，静态值的类型不可被推断，必须显式声明类型。

```
fn main() {
    static PI: f64 = 3.14;
    let r = 4.0;
    println!("面积: {}", PI*r*r);
}
```

与栈变量相比，静态值的地址明确显示出它们位于内存中不同的区域。

```
fn main() {
    static A: i32 = 10;
    static B: i32 = 20;
    let a = 10;
    let b = 20;
    println!("Global: ptr_A {:p} ptr_B {:p}", &A, &B);
    println!("Stack: ptr_a {:p} ptr_b {:p}", &a, &b);
}
```

堆

堆是运行时可供应用程序使用的进程内存。这通常是应用程序最大的可用内存池，是放置大型对象的地方。在运行时，应用程序会根据需要在堆上分配内存。这通常被称为动态内存分配。当不需要时，堆内存可被释放，返回可用池中。堆内存取自应用程序的虚拟内存。一个进程与设备上其他正在运行的进程共享物理内存。因此一个应用程序并不拥有计算机上的所有内存。相反，应用程序被分配了一个虚拟地址空间(虚拟内存)，然后操作系统将其映射到物理内存。在申请对内存时，操作系统必须首先找到足够的连续内存以满足要求，然后在该位置分配内存，并返回一个指向该地址的指针。地位和分配内存的过程可能会比较耗时。

- 此外，堆可能会因为一系列不同大小的数据分配操作而变得碎片化。即使有足够的内可用存，但不是单一位置的整块内存，也可能导致内存分配失败。一些操作系统提供了系统API来对堆进行整理，以缓解该问题。

与栈不同，堆是进程内所有线程都可以访问的共享内存。因此堆上的数据可能不是内存安全的。但是我们可以用RwLock这样的类型来管理共享内存。

在Rust中，Box类型用于在堆上分配内存。当Box被释放时，通常在当前块的末尾，释放这个堆内存。然而，如果Box的值一直没得到释放，就会导致内存泄漏。或者，可以使用drop关键字显式释放Box及相关内存。

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

Box是泛型结构体，它的类型参数是T,T是动态分配的类型(?Sized)，类型参数A是对内存分配器的引用，Global是默认的分配器件，用于在堆上分配内存。如果需要，你可以使用自定义分配器。可以使用new构造函数创建一个Box

```
fn new(x: T)->Box<T, Global>
```

Box::new函数用于在堆上创建一个值，并返回一个Box值，而不是指向堆内存的原始指针。要访问堆上的Box值，需要对Box进行解引用。然而这种解引用也不是必须的，有时会发生自动解引用。例如println!宏。

```
fn main() {
    let boxa = Box::new(10);
    let stackb = *boxa + 1;
    println!("{} {}", boxa, stackb);
}
```

示例

```
fn main() {
    let boxa = Box::new(1);
    let boxb = Box::new(2);

    let c = 1;
    let d = 2;
    println!("boxa:{:p} boxb:{:p} &c{:p} &d{:p}", &boxa, &boxb, &c, &d);

    let rawa = Box::into_raw(boxa);
    let rawb = Box::into_raw(boxb);

    println!("rawa:{:p} rawb{:p} &c{:p} &d{:p}", rawa, rawb, &c, &d);

    let boxc;
    let boxd;
    unsafe {
        boxc = Box::from_raw(rawa);
        boxd = Box::from_raw(rawb);
    }

    println!("boxc value:{}", *boxc);
    println!("boxd value:{}", *boxd);
}
```

- Box本身位于栈上，即使引用了堆上的数据。
- into_raw函数用于获取Box值的原始指针(rawa和rawb)。原始指针直接指向堆内存，并且是unsafe的。当它被释放时，堆内存不会移除。
- Box值与局部变量c和d位于内存的不同区域。

- 你还可以使用from_raw函数将原始指针重新放回Box中，此后，Box将恢复对堆上数据项的责任。必须将from_raw标记为unsafe来调用。

你也可以将栈上的值移动到堆，其结果取决于值移动是复制语义还是移动语义。

例如，将String变为 `Box<String>` 变量时，所有权被转移到堆上，也就是说String智能指针本身被移动到堆上。

```
fn main() {
    let a = 10;
    let mut boxa = Box::new(a);
    *boxa += 1;
    println!("{} {}", a, *boxa);
}
```

内部可变性

内部可变性用一个场景来描述是。你管理你个大型连锁超市内的一家杂货店。在结账时，顾客购物车里的商品会被合计并记录到收据上。收据上的商店Id和交易Id是固定的，而总金额字段是可变的。

```
struct Transaction {
    storeid: i8,
    txid: i32,
    mut total: f64 //错误的，结构体某单个字段无法声明为可变的
}
```

结构体单个字段无法声明为可变，可变性是在结构体级别上声明的。但是这会导致不恰当的更改

```
#[derive(Debug)]
struct Transaction {
    storeid: i8,
    txid: i32,
    total: f64,
}

fn main() {
    let mut tx = Transaction {storeid: 0, txid: 0, total: 64.0};
    tx.storeid=101; // oops
    println!("{:?}", tx);
}
```

解决办法是内部可变性。支持内部可变性的类型是一种内部值的包装器，包装器呈现了一种不可变的外观，同时间接允许对其内部值的修改。

Cell

Cell是一种支持内部可变性的类型，它的类型参数是T，其中T描述了内部值，Cell位于std::cell模块中

Cell可以保证不变性，而内部值可以使用方法修改

- Cell::get 返回内部值的副本
- Cell::set 修改内部值

```
fn get(&self) -> T  
fn set(&self, val: T)
```

可以使用Cell::new函数创建一个Cell

```
fn new(value: T) -> Cell<T>
```

修改最开始的例子

```
use std::cell::Cell;  
  
fn main() {  
    let cell = Cell::new(0);  
    let data = cell.get();  
    cell.set(1);  
    println!("cell:{} data:{}", cell.get(), data);  
}
```

```

use std::cell::Cell;

#[derive(Debug)]
struct Transaction {
    storeid: i8,
    txid: i32,
    total: Cell<f64>,
}

fn main() {
    let item_prices = [11.21, 25.45, 30.5];
    let tx = Transaction {
        storeid: 100,
        txid: 203,
        total: Cell::new(0.0),
    };

    for prices in item_prices {
        let total = tx.total.get() + prices;
        tx.total.set(total);
    }

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

```

Cell还有一个好处，比如下面的代码，Rust不允许存在多个可变借用

```

fn main() {
    let mut a = 1;
    let ref1 = &a;
    let ref2 = &a;

    let mut ref3 = &mut a;
    let mut ref4 = &mut a;
    *ref3 = 2;
    println!("{}{ref3}");
}

```

如果用Cell

```

use std::cell::Cell;

fn main() {
    let a = 10;
    let cell = Cell::new(a);
    let cell1 = &cell;
    let cell2 = &cell;
    cell1.set(11);
    cell2.set(12);
    println!("{}{cell.get()}");
}

```

我们可以通过多个不同的引用修改内部值。还有一些Cell的函数

- replace: 用新值替换内部值，然后返回被替换的旧值
- swap: 交换两个Cell的内部值
- take: 获取内部值并将其内部替换为默认值

RefCell

RefCell也位于std::cell模块中，与Cell不同的是，RefCell只提供对内部值的引用，而不是副本。

- RefCell::borrow 获取不可变借用
- RefCell::borrow_mut 获取可变借用

```
fn borrow(&self) -> Ref<'_, T>
fn borrow_mut(&self) -> RefMut<'_, T>
```

可以用new函数创建一个ReCell

```
fn new(value: T) -> RefCell<T>
```

示例

```
use std::cell::RefCell;

fn main() {
    let ref_cell = RefCell::new(0);
    *ref_cell.borrow_mut() += 10;
    println!("*ref_cell: {}", ref_cell.borrow());
}
```

对于RefCell, 可变性规则(不允许存在多个可变借用)完全适用。然而这些规则是在运行时而不是编译时强制执行的。因此要格外小心不要违反这些规则，会出现panic

```
use std::cell::RefCell;

fn main() {
    let refcell = RefCell::new(0);
    let mut ref1 = refcell.borrow_mut();
    let mut ref2 = refcell.borrow_mut();

    *ref1 = 10;
    println!("{}", ref1);
}
```

示例二

```
```rust
use std::cell::RefCell;

fn main() {
 let refcell = RefCell::new(0);
 let ref1 = refcell.borrow();
 let mut ref2 = refcell.borrow_mut();

 println!("{}" ,ref1);
}
```

try\_borrow函数是borrow函数的一个替代方案。该函数返回一个Result类型，当已存在一个可变引用时，它不会引发panic.而是直接返回Result类型的Err。如果成功则返回Ok(reference)

```
use std::cell::RefCell;

fn main() {
 let refcell = RefCell::new(0);
 let ref1 = refcell.borrow();
 let ret = refcell.try_borrow_mut();
 match ret {
 Ok(value) => println!("Interior value: {}", value),
 Err(_) => println!("不可再声明可变借用"),
 }
}
```

还是修改最开始的例子

```
use std::cell::RefCell;

#[derive(Debug)]
struct Transaction {
 storeid: i8,
 txid: i32,
 total: RefCell<f64>,
}

fn main() {
 let item_prices = [11.5, 20.5, 30.0, 40.3];
 let tx = Transaction{storeid: 100, txid: 203, total: RefCell::new(0.0)};

 for prices in item_prices {
 *tx.total.borrow_mut() += prices
 }

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

RefCell还有其他有用的方法

- replace: 用另一个值替换内部值，并返回当前值

- swap: 交换两个RefCell的内部值

## OnceCell

与Cell和RefCell类似，区别是OnceCell只能修改一次内部值。如果再次修改会发生错误。可以用new函数创建一个OnceCell.

- set用于初始化内部值
- get返回内部值，可以根据需要多次获取内部值

```
fn new() -> OnceCell<T>
fn set(&self, value, T) -> Result<(), T>
fn get(&self) -> Option<&T>
```

### 示例

```
use std::cell::OnceCell;

fn main() {
 let once = OnceCell::new();
 let mut result = Ok(());
 for i in 1..=3 {
 result = once.set(i);
 match result {
 Ok(_) => println!("Updated"),
 Err(_) => println!("Not updated")
 }
 }
 println!("{}:{?}{", result);
}
```

### OnceCell其他有用的函数

- get\_mut 获取内部值的可变引用
- get\_or\_init 获取内部值，如果未初始化，则使用闭包初始化它
- take 获取内部值然后将内部设置为默认值

# chapter\_19

## 宏

Rust中的宏是一种强大的特性，允许你提供新功能，实现默认行为或者完成繁琐的操作。

- 最受欢迎的是`println!`宏，它提供了可变参数函数的功能。而Rust是不支持可变参数函数的。因此`println!`宏提供了语言标准没有的特性。
- 第二受欢迎的是`derive`属性宏。例如，用于`Clone trait`和`Copy trait`的`derive`属性宏，实际上实现了这些trait的默认行为。

本质上，宏是用于生成代码的代码---也就是所谓的元编程。宏在编译时被展开。因此，宏中的错误(特别是代码格式的错误)，通常在编译时被发现。

宏在Rust中无处不在，对于其他语言来说，宏只是一个附加工具，但Rust不是这样，宏是Rust语言的核心，例如`println!`、`format!`、`vec!`、`assert!`、`hash_map!`等都是宏。

Rust没有完善的反射，例如`any::type_name`。但宏提供了有限的反射能力。这也体现了宏在Rust中的重要性。

你可能认为宏只是一些高级函数。然而，宏具有与函数不同的特性。

- 宏支持可变参数
- 宏在编译时被展开
- 宏变量(元变量)是无类型的
- 宏具有不同的错误处理模型
- 宏的定义和使用的位置会有所不同

宏有两种类型：

- 声明宏 例如`println!`宏
- 过程宏 例如`derive`属性宏

Rust的宏非常多样化。这些灵活性也给开发者带来了额外的复杂性。此外，宏有时可能不够透明和易读。基于这些原因，如果一个任务可以通过函数来充分完成，那么你就应该使用函数！

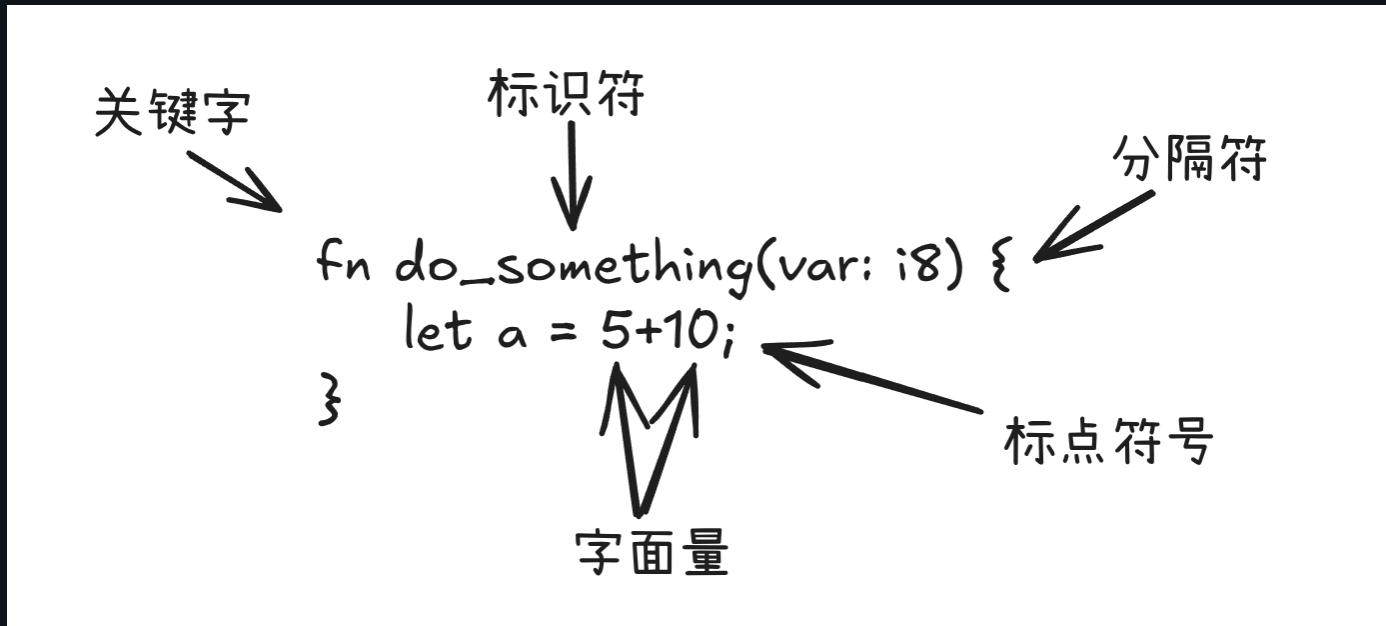
## 词条

- 在编译时，Rust程序首先被转换为一系列的词条(token)，这在编译阶段被称为词法分析(tokenization)。你可能对这阶段的一些词条比较熟悉，比如字面量或关键字
- 在词法分析后，编译的下一个阶段是将词条流转换为抽象语法树(AST)，这是一个由词条、树和叶子组成的层次结构，用来描述应用程序的行为。

- 宏在抽象语法树创建阶段之后被展开。因此，宏必须包含有效的语法，否则程序无法编译。在这个阶段，宏可以读取抽象语法树中的词条，以理解程序或代码片段并规划响应。响应可能是替换现有的词条或在词条流后插入额外的词条。

词条流使用以下不同类型的词条

- 关键字
- 标识符
- 字面量
- 生命周期
- 标点符号
- 分隔符



## 声明宏

声明宏类似于match表达式，但它针对代码而非值

- 对于声明宏，模式被称为宏匹配器，你将匹配一种代码片段类型，如表达式。
- 在匹配表达式中使用的匹配分支等价于宏中的宏转录器。
- 如果匹配成功，那么宏转录器就插入词条流中的替换代码。
  - 声明宏是自上而下展开的。在第一个匹配的模式处，转录就扩展到词条流中 使用 macro\_rules! 宏声明一个声明宏是一种自举，即宏创建宏。语法如下

```

macro_rules! identifier {
 (macro_matcher1) => {macro_transcriber};
 (macro_matcher2) => {macro_transcriber};
 (macro_matcher3) => {macro_transcriber}
}

```

以标识符(`identifier`)来命名宏，然后使用宏操作符(`!`)来调用宏。

```

macro_rules! hello {
 () => { //宏的匹配器
 println!("hello")
 };
}

fn main() {
 hello!() // 用println! ("hello")替换
}

```

在非宏的代码中，值被赋予类型(i8、f64、String等)，宏也有一套类型，然而宏操作的是代码而不是值，因此该类型也与代码有关。 "宏的类型"被称为片段说明宏。 以下是片段说明符的列表

- block: 代码块
- expr: 表达式
- ident: 标识符
- item: 代码中的语法项
- lifetime: 生命周期
- literal: 字面量或标签标识符
- meta: 元数据，属性的内容
- pat: 模式
- pat\_param: 一个允许or(|)词条的模式参数
- path: 路径类型
- stmt: 语句
- tt: 词条树(TokenTree)
- vis: 可见性 你可以声明绑定到代码的变量，这与声明一个绑定值的变量类似。绑定到代码片段的变量被称为元变量(metavariable)，并且前面带有美元符号(\$)

```

macro_rules! hello {
 ($name: expr) => {
 println!("hello, {}", $name)
 };
}

fn main() {
 hello!("Dog")
}

```

expr是片段说明符，用于表达式，并且是唯一可接受的模式。name元变量被绑定到宏输入。如果name是一个表达式，那么存在一个匹配示例二

```

macro_rules! talk {
 ($lit:literal) => {
 println!("Literal: {:?}", $lit)
 };
}
fn main() {
 talk!("Dog");

 let input = 12;
 talk!(input); //报错
}

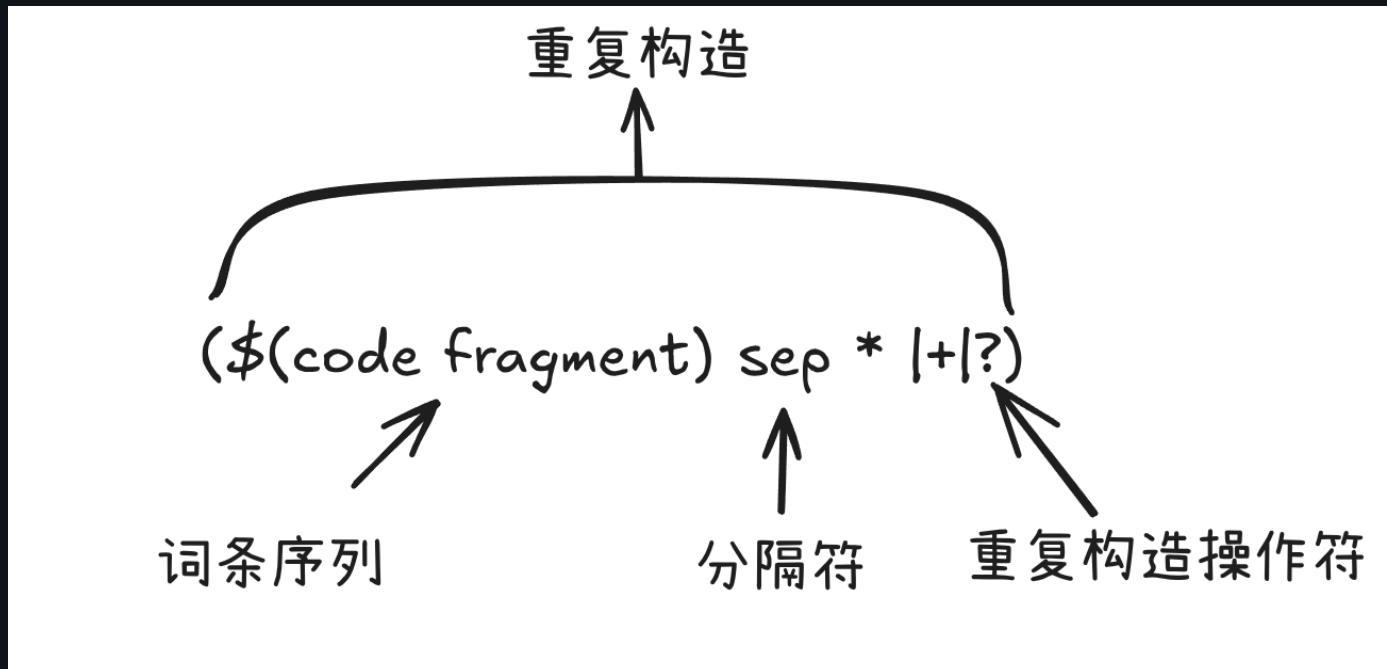
```

input是一个表达式(expr)，talk!宏只接受字面量(literal)，因此程序无法编译。

## 重复构造

Rust声明宏允许在代码片段上进行重复转换。它可以在宏匹配器或转录器内发挥作用。正是这种能力，为变参宏提供了支持。而核心语言不支持这一点。

`$(code fragment), * | + | ?)`



用于重复构造的操作符

- 星号(\*) 0或多个
- 加号(+) 1或多个
- 问号(?) 0或1个

```

macro_rules! vec_evens {
 ($($item: expr),*) => {
 {
 let mut result = Vec::new();
 $(
 result.push($item);
)*
 result
 }
 };
}

fn main() {
 let vec = vec_evens!(1,2,3,4,5);

 for ele in vec {
 println!("{}" ,ele);
 }
}

```

你甚至可以在宏内引用一个声明宏

```

macro_rules! hello_world {
 () => {
 println!("{} {}", hello!(), name!())
 };
}

macro_rules! hello {
 () => {
 "hello"
 };
}

macro_rules! name {
 () => {
 "小王"
 };
}

fn main() {
 hello_world!();
}

```

hello\_world! 宏无法通过编译，因为 hello! 和 name! 宏会在调用点展开，而它们在 println! 的展开作用域中并不可见。因此，hello\_world! 宏内部实际上引用了未在当前作用域内定义的宏。为避免这种问题，可以在宏定义中使用 \$create 元变量前缀，并通过 :: 作为分隔符来确保正确引用。

```

#[macro_export]
macro_rules! hello_world {
 () => {
 println!("{} {}", $crate::hello!(), $crate::name!());
 };
}

#[macro_export]
macro_rules! hello {
 () => {
 "hello"
 };
}

#[macro_export]
macro_rules! name {
 () => {
 "小王"
 };
}

fn main() {
 hello_world!();
}

```

## 多个宏匹配器

宏可以有多个宏匹配器

```

macro_rules! product {
 ($a:expr, $b:expr) => {
 $a * $b
 };
 ($a:expr, $b:expr, $c:expr) => {
 $a * $b * $c
 };
}

fn main() {
 let ret = product!(1, 2);
 println!("{}", ret);
 let ret = product!(1, 2, 3);
 println!("{}", ret);
}

```

# 过程宏

## 1. 过程宏概念

过程宏（Procedural Macro）是一种强大的 Rust 宏，它接收描述程序某个部分的词条流（`TokenStream`）作为输入，经过处理后生成新的 `TokenStream`，并将其插入到程序中相应的位置。

与声明宏不同，过程宏没有模式匹配器，而是直接解析输入的词条流并生成输出流。

注意：`TokenStream` 是所有宏的基本组成部分，表示一个抽象的词条序列，也就是代码片段。

## 2. 过程宏的三种类型

过程宏主要有三种风格：

1. 派生宏（Derive Macros）
2. 属性宏（Attribute Macros）
3. 类函数宏（Function-like Macros）

## 3. `TokenStream` 与工具

过程宏的输入和输出都是 `TokenStream`。为了方便操作源码，可以使用以下方法：

- `TokenStream::from_str` 将源码片段（字符串）转换为 `TokenStream`，如果源码有语法错误，会返回错误。

```
fn from_str(src: &str) -> Result<TokenStream, LexError>
```

- `syn crate` 提供将 `TokenStream` 转换为更易于操作的抽象语法树(AST)类型，如 `DeriveInput`。

```
use syn::{parse_macro_input, DeriveInput};

let input: DeriveInput = parse_macro_input!(input as DeriveInput);
```

- `quote crate` 提供 `quote!` 宏，将 Rust 代码以类似模板的方式生成词条，并可转换为 `TokenStream`：

```
use quote::quote;

let tokens = quote! {
 fn hello_world() {
 println!("Hello, world!");
 }
};

let token_stream: TokenStream = tokens.into();
```

syn和quote是额外工具，使用前需引入依赖：

```
[dependencies]
syn = "2.0"
quote = "1.0"
```

## 4. 创建过程宏 Crate

过程宏必须在独立的 crate 中实现，并在 `Cargo.toml` 的 `[lib]` 中声明：

```
[lib]
proc-macro = true
```

---

## 5. 派生宏 (Derive Macro)

派生宏是一种特殊的过程宏，通常用于为结构体、枚举或联合体自动实现 trait，例如 `Debug`、`Clone` 等。

- 使用方式：

```
#[derive(MacroName)]
struct MyStruct;
```

- 声明派生宏函数：

```
use proc_macro::TokenStream;

#[proc_macro_derive(Hello)]
pub fn hello(input: TokenStream) -> TokenStream {
 // 将代码片段转换为 TokenStream
 r##"
 fn hello_world() {
 println!("Hello, world!");
 }
 "#.parse().unwrap()
}
```

- 使用宏：

```
use packagename::Hello;

#[derive(Hello)]
struct Bob;

fn main() {
 let bob = Bob;

 // 派生宏生成的函数可以直接调用
 hello_world();
}
```

说明：在这个示例中，派生宏 Hello 会为 Bob 生成一个 hello\_world 函数，并插入到代码中。之后可以在 main 中直接调用该函数。

让我们改进代码

```

use proc_macro::TokenStream;
use syn::DeriveInput;

#[proc_macro_derive(Hello)]
pub fn hello(input: TokenStream) -> TokenStream {
 let derive_input = syn::parse_macro_input!(input as DeriveInput);
 let name = derive_input.ident; //这个目标的 标识符(名字), 类型是 syn::Ident
 let code = format!(r##"impl Hello for {name} {{
 fn hello_world() {{
 println!("hello {name}");
 }
 }}##);
 code.parse().unwrap()
}

// main.rs
use packagename::Hello;

trait Hello {
 fn hello_world();
}

#[derive(Hello)]
struct Bob;

fn main() {
 Bob::hello_world();
}

```

- derive\_input.ident 返回这个目标的 标识符(名字), 类型是 `syn::Ident`, 并将其绑定到name 变量。在上面代码中, 由于作用的目标是Bob结构体, 所以ident就是Bob
- TokenStream 实现了 FromStr, 也就是 TokenStream::from\_str()。所以可以通过 String::parse() 自动把字符串转换为 TokenStream。

下面展示Hello宏的最终版本 hello\_world函数在quote!宏中实现, 而不是使用字符串。quote ! hong1在编写kuoz或复杂宏时更加透明

## 注意事项

- quote!宏的结果可以通过TokenStream::from函数转换为proc\_macro::TokenStream。
- 为了在quote!宏中包含函数中的变量, 需要在变量前加#, 如#ident

```
use proc_macro::TokenStream;
use quote::quote;

#[proc_macro_derive(Hello)]
pub fn hello(input: TokenStream) -> TokenStream {
 let syn::DeriveInput { ident, .. } = syn::parse_macro_input!(input);

 let tokens = quote! {
 impl Hello for #ident {
 fn hello_world() {
 println!("Hello from {}", stringify!(#ident));
 }
 };
 tokens.into()
 }

 // main.rs
 use packagename::Hello;

 trait Hello {
 fn hello_world();
 }

 #[derive(Hello)]
 struct Bob;

 fn main() {
 Bob::hello_world();
 }
}
```

---

另一个实用的例子，Type trait提供了运行时类型信息(RTI)的接口。Type宏实现了该trait。在get函数中，调用any::type\_name可以获取目标类型的名称。

```

use proc_macro::TokenStream;
use quote::quote;

#[proc_macro_derive(Type)]
pub fn get_type(input: TokenStream) -> TokenStream {
 let syn::DeriveInput{ident,..} = syn::parse_macro_input!(input);
 let tokens = quote! {
 impl Type for #ident {
 fn get(&self) -> String {
 std::any::type_name::<#ident>().to_string()
 }
 };
 tokens.into()
 }
}

// main.rs
use packagename::Type;

pub trait Type {
 fn get(&self) -> String;
}

#[derive(Type)]
struct MyStruct;

fn main() {
 let my = MyStruct;
 let ret = my.get();
 println!("ret: {}", ret);
}

```

## 6. 属性宏

属性宏作为自定义属性被调用，这与派生宏不同，派生宏在derive属性中呈现。以下是不同点

- 属性宏以proc\_macro\_attribute修饰
- 属性宏定义新属性，而不是derive属性
- 属性宏也可以应用于结构体、枚举、联合体以及函数
- 属性宏用于替换目标对象
- 属性宏有两个TokenStream参数

```

#[proc_macro_attribute]
pub fn macro_name(parameter1: TokenStream,
 parameter2: TokenStream) -> TokenStream

```

属性宏有两个TokenStream参数

- 第一个是描述宏的参数，当没有参数时，TokenStream为空
- 第二个是描述属性宏的目标对象，例如结构体或函数。

属性宏返回一个TokenStream,与派生宏不同，TokenStream替换了目标，例如，如果你将属性宏应用于一个结构体，那么该宏会在词条流中完全替换该结构体。

```
use proc_macro::{TokenStream};
use quote::quote;

#[proc_macro_attribute]
pub fn info(parameters: TokenStream, target: TokenStream) -> TokenStream {
 let args = parameters.to_string();
 let current = target.to_string();
 let syn::DeriveInput { ident, .. } = syn::parse_macro_input!(target);
 quote! {
 struct #ident{}

 impl #ident {
 fn describe() {
 println!("Token1 {}", #args);
 println!("Token2 {}", #current);
 }
 }
 }
 .into()
}

use packagename::info;

#[info(a, b)]
struct Sample{};

fn main() {
 Sample::describe();
}
```

## 7. 类函数宏

类函数宏带有#[proc\_macro]属性声明，并用宏操作符(!)调用。 定义如下

```
#[proc_macro]
pub fn macro_name(parameter1: TokenStream)->TokenStream
```

示例

```
use packagename::create_hello;

create_hello!();

fn main() {
 hello_world();
}
```

# chapter\_20

有时Rust需要与其他编程语言编写的程序或库通信，包括调用系统API等。尽管crates.io和Rust生态系统提供了很多功能。这就意味着你可能需要在某时离开Rust舒适区。

Rust支持C **应用程序二进制接口(ABI)**。虽然有很多限制，但C ABI仍然是许多语言和操作系统的首选公共接口。C语言(或其他基于C的语言)已经存在超过50年了。因此用C/C++编写的应用程序涵盖了所有计算需求。互操作性为Rust开发者提供了对这一庞大库的访问能力。

在不同语言交换数据是巨大的挑战，

- 尤其是处理字符串。C语言的字符串是以空字符结尾的，而Rust则不是。这种不同语言的类型系统的差异常常会带来额外的兼容性问题。
- 另一个潜在问题是是指针的管理方式。这在不同语言之间可能存在差异。

解决这些问题的方案是使用外部函数接口(FFI)。它提供了Rust与其他语言之间传输数据的能力，以处理这些差异。

## 外部函数接口

外部函数接口是确保互操作性成功的粘合剂。它在Rust和C之间建立了一个翻译层。FFI的部分可以在std::ffi模块中找到。你可以看到大多数情况下将数据从Rust编组到C所需的标量、枚举和结构体。

字符串是最难正确编组的类型。这一点考虑到Rust和C/C++的差异。

- C字符串以空字符结尾，而Rust则不是
- C字符串不能包含控制码，而Rust允许
- C字符串可以通过原始指针直接访问。Rust需要通过胖指针访问，胖指针还包含了额外的元数据。
- Rust字符串主要使用Unicode字符集和Utf-8编码。对于C语言，Unicode的使用可能会有所不同。

甚至Rust中的char与C的字符也不同。

- 在Rust中的char是Unicode标量值，而C的char支持Unicode代码点

在FFI中，CString类型用于在Rust和C之间编组字符串。CStr类型用于将C字符串转换成&str。还有OsString和OsStr类型，用于读取操作系统字符串，例如命令行参数和环境变量。

此外，一些Rust类型可以"按原样"完全兼容，包括浮点数，整数和基本枚举。相反，动态大小的类型不受支持，比如trait和切片

对于有限数量的条目，创建适当的接口以进行编组是可行的。然而当有数百个条目需要编组时，这种方式难以维持。例如编组C标准库的部分内容，你不会想要将整个stdlib.h头文件进行编组。幸运的是，libc crate提供了绑定来对C标准库中的部分内容进行编组。

## 基础示例

我们从"Hello World"开始。

目录结构如下



1. 创建一个C源文件

```
// hello.c
#include<stdio.h>

void hello() {
 printf("HelloWorld");
}
```

2. 使用clang编译器和llvm工具编译c源代码，并创建一个静态库

```
clang hello.c -c
ar rcs libhello.a hello.o # linux
或者
llvm-lic hello.o # windows
```

3. 将从C导出的函数定义在extern "C"块中，该块可以看出Rust风格的头文件。调用这个函数时必须使用unsafe包裹。因为Rust无法保证外部函数的安全性。

```

extern "C" {
 fn hello();
}

fn main() {
 unsafe {
 hello()
 }
}

```

4. 是时候构建应用程序了。使用rustc编译器，你可以从Rust源文件创建一个可执行的crate, 同时链接到一个外部库

```
rustc main.rs -L ./ -l static=hello
```

- `-L ./` -> 告诉 rustc 在这个目录找库
- `-l static=hello` -> 链接 libhello.a

你可以通过build.rs脚本来自动化完成构建过程，包括链接库。build.rs文件在构建过程中由cargo自动检测和执行。

```

extern crate cc;

fn main() {
 cc::Build::new().file("c_src/hello.c").compile("hello");
}

```

- 第一步是使用 `Builder::new` 构造函数创建一个Builder类型。
- `Builder::file` 函数识别输入文件(这里是一个c文件)以进行编译。
- `file::compile` 函数执行实际的编译并生成静态库 libhello.a (Linux/macOS) 或 hello.lib (Windows)

生成的静态库会放在 Cargo 自动管理的 target 目录中，并且 Rust 可以在后续的 FFI 中链接使用。

这些函数都在cc crate中被定义，你必须在使用前在cargo.toml中的build-dependencies部分添加cc(crate.io中的cc)

```
[build-dependencies]
cc = "1.2.41"
```

## libc crate

libc crate包含了与C标准库的一部分进行编组的ffi绑定，包括stdlib.h的ffi绑定。在extern块中，你只要列出将应用程序中使用的C标准库的数据项，而不需要其他东西，这就是libc crate的好处。

```
use std::ffi::{c_double, c_longlong, CString};

unsafe extern "C" {
 //将字符串浮点数转换为浮点数
 fn atof(p: *const i8) -> c_double;
 //将字符串整数转换为长整型
 fn atoi(p: *const i8) -> c_longlong;
}

fn main() {
 let f_str = "256.78".to_string();
 let f_cstr = CString::new(f_str).unwrap();
 let i_str = "345".to_string();
 let i_cstr = CString::new(i_str).unwrap();

 let mut f_ret: c_double;
 let mut i_ret: c_long;
 unsafe {
 f_ret = atof(f_cstr.as_ptr()); // 转换为整数
 i_ret = atoi(i_cstr.as_ptr()); // 转换为整数
 }

 println!("{}", f_ret);
 println!("{}", i_ret);
}
```

atof和atoi函数接收字符串作为参数，并分别返回一个浮点数和整数。CString::new函数将String类型转换为CString类型。as\_ptr函数将CStrings转换为指针，这等同于char\*。调用函数的结果被保存在适当的类型c\_double和c\_longlong。由于使用了libc crate。构建该程序无需考虑特殊因素，正常编译。

## 结构体

我们需要经常对复杂类型(如结构体)进行编组。例如，系统API通常需要结构体作为参数或返回值。

编组复杂类型需要额外考虑。内存对齐可能有所不同。此外，C结构体的内存布局可能会受到用户定义的打包和内存边界的影响。Rust并不保证其结构体的内存布局。这对编组可能是一场噩梦。解决方案是采用C语言模型来编组结构体。你可以应用#[repr(C)]属性到结构体上，这将消除C与Rust结构体之间的内存布局差异。

```
#[repr(C)]
struct AStruct {}
```

结构体将按照其组成部分进行编组。只有这样，你才能确定正确的编组方法。

```
struct AStruct {
 int field1;
 int field2;
 int field3;
}
```

```
struct AStruct {
 field1: c_int,
 field2: c_int,
 field3: c_int,
}
```

示例

```

use std::ffi::{c_int, CStr, CString};

#[repr(C)]
pub struct Person {
 first: *const i8,
 last: *const i8,
 age: c_int,
}

unsafe extern "C" {
 fn get_person() -> *mut Person;
 fn set_person(new_person: Person);
}

fn main() {
 let person;

 unsafe {
 person = get_person();
 println!("{}:{:?}", (*person).age);
 println!("{}:{:?}", CStr::from_ptr((*person).first));
 println!("{}:{:?}", CStr::from_ptr((*person).last));
 }

 let first = CString::new("Sally").to_string().unwrap();
 let pfirst = first.as_ptr();
 let last = CString::new("Johnson").to_string().unwrap();
 let plast = last.as_ptr();
 let new_person = Person {
 first: pfirst,
 last: plast,
 age: 43,
 };
 unsafe {
 set_person(new_person);
 println!("{}:{:?}", (*person).age);
 println!("{}:{:?}", CStr::from_ptr((*person).first));
 println!("{}:{:?}", CStr::from_ptr((*person).last));
 }
}
}

```

以下是程序各部分的描述

- `extern "C"` 块中导入`get_person`和`set_person`函数
- 在第一个`unsafe`代码块中，获取并显示`gPerson`的默认值。
  - 调用`get_person`函数返回`*Person`类型的默认值，对指针解引用来访问字段。
  - 调用`CStr::from_ptr`函数将`first`和`last`字段转换成字符串字面量。
  - 在`println!`宏中显示结构体中的三个字段
- 第二个`unsafe`代码块中，修改并显示`gPerson`的值。
  - 创建一个新的结构体，为结构体的每个字段添加新值。
  - 调用`set_person`函数将`gPerson`设置为新值。

# bindgen

你可以花大量时间创建正确的FFI绑定，以便在Rust和C之间传输数据。可是，当包含数十甚至数百个需要编组的文件时，这个过程相当繁琐。此外，如果处理不当，你可能会花费更多时间俩调试。幸运的是bindgen(绑定生成)工具会自动化帮你完成这个过程。

bindgen为C定义创建正确的FFI绑定，以免你手动完成创建映射这种繁琐工作。 bindgen可以读取C头文件并为其自动生成包含所有对应Rust绑定的源文件。这对于libc未包含的C标准库非常有用。 Bindgen源于crates.io，你可以通过cargo安装bindgen.

```
cargo add bindgen
```

按照bindgen-cli可以使用bindgen命令

```
cargo install bindgen-cli
```

读取头文件并生成适当的FFI绑定

```
bindgen time.h > time.rs
```

## C调用Rust函数

为了互操作性，我们在要导出的Rust函数前加上extern关键字。 Rust为其函数名进行混淆，以赋予它独一无二的身份。混淆后的名称结合了crate名、哈希值、函数本身的名称以及其他因素。这意味着其他语言(不知道这个方案)将无法识别Rust函数的内部名称。因此，为了使函数保持透明，需要使用no\_mangle属性禁用函数的名称混淆。

```
#[no_mangle]
extern fn display_rust() {
 println!("Greetings from Rust");
}
```

为了与其他语言互操作，你必须为Rust应用程序构建一个静态或动态的库。其他语言可以通过这个库来访问其导出函数。请将lib部分添加到cargo.toml文件中，为crate创建一个库。

- crate-type字段设置为staticlib，则代表创建一个静态库
- crate-type字段设置为cdylib，则代表创建一个动态库

默认以包名为库名，如果需要，可以使用name字段显示设置名字

```
[lib]
name = "greeting"
crate-type = ["staticlib", "cdylib"]
```

上面的cargo.toml片段要求Rust程序同时创建静态库和动态库

### 1. lib.rs

```
#[unsafe(no_mangle)]
pub extern fn display_rust() {
 println!("Greetings from Rust");
}
```

编译生成的库在target/下

### 2. sample.h

```
void display_rust();
```

### 3. sample.c

```
#include "hello.h"
int main(void) {
 display_rust();
}
```

在构建C程序时，你必须包含C源文件并链接到Rust创建的库。

```
clang sample.c libgreeting.a -o sample
或者
clang sample.c libgreeting.so -o sample
```

这个命令会在当前目录下产生c的可执行文件，`sample`

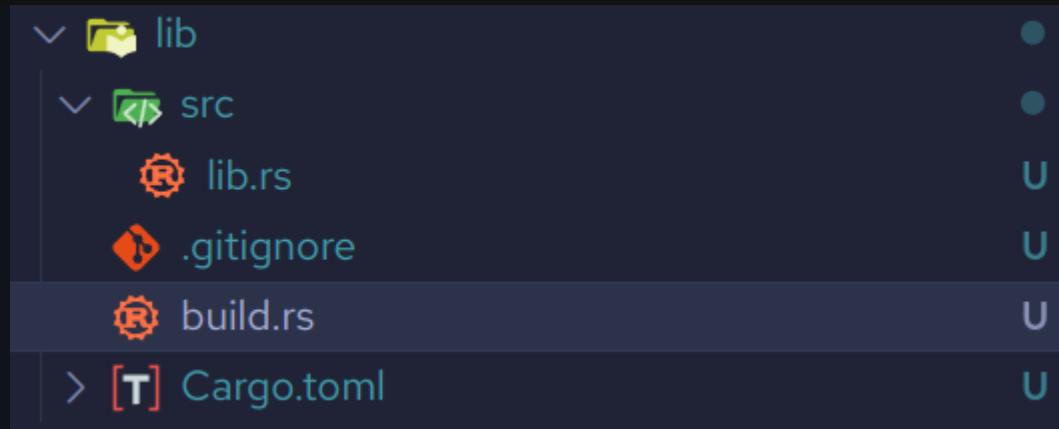
## cbindgen

- `bindgen` 工具用于创建C文件生成Rust的FFI绑定，让Rust能调用C代码。
- `cbindgen` 工具与 `bindgen` 工具正好相反。`cbindgen` 工具从Rust代码中生成C头文件，让C能够调用Rust的代码。

`cbindgen` 可以在crates.io中找到

你可以结合 `cbindgen` 和 `build.rs` 来自动化该过程。

项目结构如下



### 1. lib.rs

```
#[unsafe(no_mangle)]
pub extern fn max3(first: i64, second: i64, third: i64)->i64 {
 let value = if first>second {
 first
 }else {
 second
 };

 if value>third {
 value
 } else {
 third
 }
}
```

2. cargo.toml 在cargo.toml文件中，将cbindgen添加为构建依赖项。它将会在被build.rs构建的过程中使用。我们还要求生成静态库和动态库。

```
[build-dependencies]
cbindgen = "0.29.0"

[lib]
name = "example"
crate-type = ["staticlib", "cdylib"]
```

### 3. build.rs

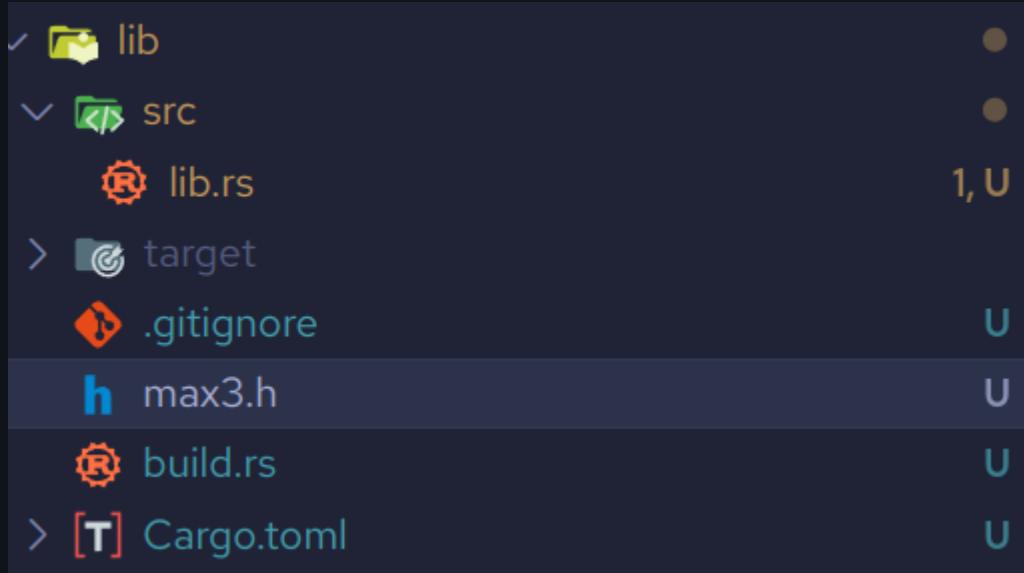
```
extern crate cbindgen;

fn main() {
 cbindgen::Builder::new()
 .with_crate(".")
 .generate()
 .expect("Unable to generate bindings")
 .write_to_file("max3.h");
}
```

编译

```
cargo build
```

这会在crate根目录生产c头文件



生成的max3.h文件

```
#include <cstdarg>
#include <cstdint>
#include <cstdlib>
#include <iostream>
#include <new>

extern "C" {

int64_t max3(int64_t first, int64_t second, int64_t third);

} // extern "C"
```

编写一个调用max3函数的c++程序,它包含了cbindgen生成的头文件

```
#include<stdio.h>
#include "max3.h" // 加上max3.h头文件的路径

int main() {
 long answer = max3(100, 200, 300);
 printf("Max value is %ld\n", answer);
 return 0;
}
```

以下命令将编译的C++程序链接到Rust库

```
clang myapp.cpp libexample.a -o myapp
```

这会在调用命令所在的目录生成可执行文件myapp

需要注意的是，我们与C++应用程序进行交互操作没有想象中的轻松，因为C++没有稳定的ABI,通常情况下，Rust与C++安全交互需要通过C ABI来完成。

# 模块化编程

模块在软件开发中扮演着至关重要的角色，它使开发者能够更有效地组织和管理代码。通过将程序结构按层级或功能进行划分，可以避免“单体式开发”带来的弊端。

在单体程序中，所有代码都集中在一个源文件中，这对于包含成百上千行代码的大型项目来说，无疑会使代码查找、理解和后续维护变得异常困难。将程序拆分为多个模块是解决这些问题的有效途径。

## 模块的本质与命名

从本质上讲，模块用于汇集相关的程序元素，这些元素可以包括结构体、枚举、函数以及全局变量等。值得注意的是，模块也可以是空的，即不包含任何元素。

模块的命名应清晰地反映其语义上下文。

## Rust 中的模块

Rust 语言本身就大量依赖模块来对功能进行分类和组织，从而极大地提高了代码的可理解性。常见的 Rust 标准库 (std crate) 模块示例如下：

- std::string：集合了与字符串操作相关的各种项，包括常用的字符串类型。
- std::fmt：提供了对 format! 等格式化宏的支持。
- std::io：将所有输入/输出相关的项进行分组，例如 Stdout (标准输出)、Stdin (标准输入) 和 StdErr (标准错误) 等类型。

这些模块的名称清晰地反映了它们的内容上下文，例如 std::io 明确指代输入/输出功能。

## 模块的层级结构

Rust 允许通过模块构建逻辑层级结构，形成一个模块树。模块路径使用 :: 作为分隔符。例如，路径 std::os::linux::net 揭示了一个层级关系：

- std 是包含 os 模块的顶级 Crate。
- os、linux 和 net 都是子模块。

net 模块的上下文是与其祖先模块的上下文（操作系统、Linux、网络）相结合的。因此，net 模块包含了诸如 TcpStreamExt 和 SocketAddrExt 类型的 Linux 特定网络功能实现。

## 与其他语言的对比

尽管模块与其它编程语言（如 Java、Python、C++）中的命名空间有相似之处，它们都用于防止命名冲突和组织代码，但 Rust 的模块还提供了额外的功能和更严格的可见性控制。

### Rust中的模块分为两类

- **模块项** 定义在单个源文件内部的模块
- **模块文件** 以独立文件的形式存在的模块

### 模块有两种模式

- **新模式** 在Rust1.30中引入的
- **遗留模式** Rust1.30之前的模式

两种模式至今都被支持且使用广泛

## 模块项

模块使用mod关键字声明。你可以在源文件中声明一个模块项

```
mod name {
 /* 模块内容 */
}
```

在花括号内，可以添加结构体和其他项。在默认情况下，模块项是私有项(仅在模块内可见)。为了让其在模块外可见，在项前加pub关键字。

```
mod greetings {
 pub fn hello() {
 println!("Hello, world!");
 }
}
fn hello() {
 println!("Hello");
}

fn main() {
 hello();
 greetings::hello(); // 可以调用, 因为 hello 是 pub
}
```

通过上述例子，我们可以发现，使用模块可以避免命名冲突。

模块与结构体、枚举以及其他类型共享相同的命名空间。因此，你不能在同一作用域内让类型和结构体使用相同名称。

```
mod example {
}

struct example; //报错

fn main() {
}
```

可以在模块中声明子模块。通过这种方式可以创建逻辑层次结构。子模块的深度没有限制。在访问某项时，你必须以完整的模块路径作为前缀。

```
mod solar_system {
 pub mod earth {
 pub fn get_name() -> &'static str {
 "Earth" //地球
 }
 pub mod constants {
 pub static DISTANCE: i64 = 93_000_000; // 离太阳的距离
 pub static CIRCUMFERENCE: i32 = 24_901; // 地球周长
 }
 }
 /* 其他八个行星 */
}

fn main() {
 println!("{}", solar_system::earth::get_name());
 println!("Distance from Sun {}", solar_system::earth::constants::DISTANCE)
}
```

封装是模块的另一个特性。在一个模块中，你可以为用户定义一个公共接口(使用pub关键字)，同时隐藏其他细节。使用pub关键字定义公共项和公共接口。其他所有内容都是默认隐藏的。这允许开发者在应用程序中更好地建模现实世界的实体。

```

// 定义一个名为 money_vault 的模块
mod money_vault {
 // 1. 定义一个公共结构体。注意：结构体本身是公共的，但它的字段默认是私有的。
 pub struct Vault {
 // 2. balance 字段是私有的（默认），外部代码无法直接访问或修改它。
 balance: f64,
 }

 // 3. 为 Vault 实现一个公共接口
 impl Vault {
 // 公共构造函数（关联函数）：允许外部代码创建 Vault 实例。
 pub fn new(initial_balance: f64) -> Vault {
 Vault {
 balance: initial_balance,
 }
 }

 // 公共方法：允许外部代码安全地存入金额。
 pub fn deposit(&mut self, amount: f64) {
 if amount > 0.0 {
 self.balance += amount;
 self.log_transaction("Deposit", amount); // 调用私有方法
 }
 }

 // 公共方法：允许外部代码安全地取出金额。
 pub fn withdraw(&mut self, amount: f64) -> bool {
 if amount > 0.0 && self.balance >= amount {
 self.balance -= amount;
 self.log_transaction("Withdrawal", amount); // 调用私有方法
 true
 } else {
 false
 }
 }

 // 公共方法：允许外部代码查看余额。
 pub fn get_balance(&self) -> f64 {
 self.balance // 允许读取私有字段
 }

 // 4. 私有方法（默认私有，没有 pub 关键字）：外部代码无法调用它。
 fn log_transaction(&self, kind: &str, amount: f64) {
 println!("Transaction: {} of ${:.2}. New balance: ${:.2}", kind, amount, self.balance);
 }
 }

 fn main() {
 // 1. 成功：通过公共构造函数创建实例。
 let mut safe = money_vault::Vault::new(100.0);
 println!("Initial balance: ${:.2}", safe.get_balance()); // 输出：100.00

 // 2. 成功：通过公共方法进行操作。
 safe.deposit(50.0);
 }
}

```

```

if safe.withdraw(20.0) {
 println!("Withdrawal successful.");
} else {
 println!("Withdrawal failed.");
}

println!("Current balance: ${:.2}", safe.get_balance()); // 输出: 130.00

// 3. 编译错误 (Encapsulation enforced): 尝试直接访问私有字段。
// safe.balance = 9999.0;
// ^ 错误: 字段 `balance` 是私有的 (field `balance` is private)

// 4. 编译错误 (Encapsulation enforced): 尝试调用私有方法。
// safe.log_transaction("Hacking attempt", 1000.0);
// ^ 错误: 方法 `log_transaction` 是私有的 (method `log_transaction` is
private)
}

```

## 模块文件

模块文件包含整个文件。这与模块不同，模块项使用大括号来定义模块的范围。而对于模块文件来说，文件本身就是模块的范围，因此不需要使用大括号。用mod关键字和名称定义一个模块文件。

```
mod mymod; // mymod.rs
```

无论是模块文件还是模块项，模块的逻辑路径不会改变。你仍然需要使用::来分隔模块路径中的模块。

```

// hello.rs
pub fn hello() {
 println!("hello world");
}

// main.rs
mod hello;
fn main() {
 hello::hello();
}

```

main.rs和hello.rs在同一目录  alt text

然而，你可能希望模块文件创建子目录，以保持物理结构。使用mod关键字，你可以在模块文件内声明子模块文件。子模块文件根据模块名称(如module.rs)放在子目录中。

可以通过一个示例来展示，比如创建一个以不同语言展示HelloWorld的程序。

 alt text

```

mod greeting {
 pub mod chinese;
 pub mod english;
}

fn main() {
 greeting::chinese::hello();
}

// chinese.rs
pub fn hello() {
 println!("你好世界");
}

// english.rs
pub fn hello() {
 println!("Hello World");
}

```

path 属性 使用path属性可以显式设置模块的物理位置来覆盖默认设置。可以直接将path属性引用于模块。

```

// main.rs #[path="../cool/cooler.rs"]
mod cooler;

fn main() {
 cooler::funca();
}

// cooler.rs #[path="../cool/cooler.rs"]
pub funca() {
 println!("Doing something");
}

```

## 函数与模块

也可以在函数内声明模块，其原理是相同的。模块可用于在函数内对项分组、创建层次结构、消除歧义等。

- 在函数内声明的模块(包括其项)无法在函数外访问。
- 在模块内声明的变量必须是静态值或常量。
- 函数内声明的模块的项目必须是pub的才能在函数中使用。

```

fn funca(input: bool) {
 if input {
 mod1::do_something();
 } else {
 mod2::do_something();
 }

 mod mod1 {
 use std::println;

 pub fn do_something() {
 println!("in mod1");
 }
 }

 mod mod2 {
 use std::println;

 pub fn do_something() {
 println!("in mod2");
 }
 }
}

fn main() {
 funca(true);
}

```

## crate、super、self关键字

可以在模块路径中使用crate、super和self关键字。

- crate 用于从crate根模块进行导航。在任何模块内部都是绝对路径
- super 从父模块开始导航。这是一个相对路径，通常比crate(绝对路径)更可靠
- self 指的是当前模块

## 遗留模式

虽然在Rust1.30引入了模块的新模式。但遗留模式仍然被使用。使用遗留模式和新模式创建的模块没有区别。对于遗留模式的模块创建步骤如下:

1. 在模块中声明一个子模块，如mod name(模块名称).
2. 使用该模块名称创建一个子目录
3. 在子目录中放置mod.rs文件
4. 在模块子目录中，为之前命名的美国子模块创建同名模块文件。



```
// main.rs
mod hello;

fn main() {
 hello::chinese::hello();
}

//mod.rs
pub mod chinese;
pub mod english;

//chinese.rs
pub fn hello() {
 println!("你好世界");
}

// english.rs
pub fn hello() {
 println!("Hello World");
}
```