概括

简介

这是一本（非全面）指南适用于对Rust编程语言完全陌生的C#和.NET开发人员。一些概念和结构 在C#/.NET和Rust之间转换得相当好，会有少许表达方式不同，而另外一些完全不同，例如内存管理。 本指南通过简单示例对这些概念和结构做简明的比较和映射。

本指南的原作者本身就是C#/.NET开发人员对Rust完全陌生。本指南收集作者在编写Rust代码几个月 过程中获得的知识。这是作者在开始他们的Rust之旅时希望拥有的指南。作者鼓励你读Rust及术语的书 和网络上其他材料，而不是试图完全通过C#和.NET的视角来学习它。同时，本指南可以帮助快速回答一些问题， 例如：Rust是否支持继承、线程、异步编程等等?

假设：

- 读者是一位经验丰富的C#/.NET开发人员。
- 读者对Rust完全陌生。

目标：

- 提供C#/.NET与Rust各种主题的简明比较和映射对照。
- 提供Rust参考资料、书籍和文章的链接，以供进一步阅读相关主题。

非目标：

- 讨论设计模式和架构。
- Rust语言教程。
- 读者在阅读本指南后精通Rust。
- 虽然有一些主题的简短示例对比了C#和Rust代码，并不意味本指南是一本关于两种语言的编码经典书籍。

本指南的原作者（按字母顺序）： Atif Aziz, Bastian Burger, Daniele Antonio Maggio, Dariusz Parys 和 Patrick Schuler。

许可证

上述版权声明和本许可声明应包含在所有软件的副本或主要部分内容中。

该软件按"原样"提供，不提供任何形式的明示或暗示，包括但不限于适销性保证，适用于特定 目的且不侵权。在任何情况下都不得作者或版权所有者对任何索赔、损害或其他责任负责 责任，无论是合同诉讼、侵权诉讼还是其他诉讼，均由以下原因引起： 与本软件无关或与之相关，或者与本软件相关的使用或其他交易软件

贡献

我们邀请你贡献💔通过打开问题和提交拉取请求！

以下是一些想法💡关于如何以及在哪里可以提供最大贡献：

- 修正你在阅读时看到的任何拼写或语法错误。
- 修正技术不准确。
- 修复代码示例中的逻辑或编译错误。
- 提高英语水平，特别是如果它是你的母语或你精通该语言。
- 扩展解释以提供更多上下文或明确某些主题或概念。
- 保持C#、.NET和Rust的更改最新。例如，如果有一个C#或Rust变化使 这两种语言更接近，用可能需要修改示例代码。

如果你要进行小到适度的更正，例如修复示例代码中存在拼写错误或语法错误，请随意直接 提交拉取请求。对于可能需要你付出巨大努力的更改（以及审稿人的结果），强烈建议你在投入时间之前 提交问题和寻求维护者/编辑的批准。如果拉取请求因各种原因被拒绝，请避免心碎💔。

快速做出贡献变得非常简单。如果你看到一个页面错误并且恰好在线，你可以点击角落处的编辑图标📝，编辑Markdown源并提交更改。

贡献指南

- 坚持简介中列出的本指南的目标；用另一个方式，避免非目标！
- 更喜欢保持文本简短，并使用简短、简洁和现实的代码示例以说明一个观点。
- 尽可能多地用Rust和C＃示例来提供和比较。
- 请随意使用C#/Rust语言最新特性，使两种语言是示例简单、简洁、相似。
- 避免在C#示例中使用社区包。坚持尽可能用.NET 基础类库。由于Rust标准库API较小，某些功能调用crate更容易接受，为了说明的必要（例如 `rand` 用于随机数生成），但确保它们成熟、流行且值得信赖。
- 确保示例代码尽可能独立且可运行（除非想法是为了说明编译时或运行时错误）。
- 保持本指南的风格，如果读者正在被告知或指示，避免使用_你_；代替使用第三人称。例如, 代替说, "你提出用Rust可选数据 `Option<T>` 类型"，写为，"Rust有可选数据 `Option<T>` 类型"。

入门

Rust游乐场

无需任何本地安装的Rust游乐场是入门Rust最简单方法。它是一个最小化的前端开发工具， 运行在Web浏览器、允许写和运行Rust代码。 code.

开发容器

Rust游乐场执行环境有一些限制，比如总编译/执行时间、内存和网络，，所以另一个 不需要安装 Rust 的选项是使用 *开发容器*, 比如这个仓库 https://github.com/microsoft/vscode-remote-try-rust. 像Rust游乐场, 开发容器直接运行在浏览器通过GitHub Codespaces或者本地用 Visual Studio Code.

本地安装

对于Rust编译器及其开发工具的完整本地安装，请看Rust编程语言书的入门章节安装部分，或者安装页面在 rust-lang.org.

语言

本节比较C#和Rust语言特性。

标量类型

下面表格列出Rust中基本类型及其C#和.NET中的等效类型：

Rust	C#	.NET	注释
`bool`	`bool`	`Boolean`	
`char`	`char`	`Char`	看注释1
`i8`	`sbyte`	`SByte`	
`i16`	`short`	`Int16`	
`i32`	`int`	`Int32`	
`i64`	`long`	`Int64`	
`i128`		`Int128`	
`isize`	`nint`	`IntPtr`	
`u8`	`byte`	`Byte`	
`u16`	`ushort`	`UInt16`	

`u32`	`uint`	`UInt32`	
`u64`	`ulong`	`UInt64`	
`u128`		`UInt128`	
`usize`	`nuint`	`UIntPtr`	
`f32`	`float`	`Single`	
`f64`	`double`	`Double`	
	`decimal`	`Decimal`	
`()`	`void`	`Void` or `ValueTuple`	看注释2和3
	`object`	`Object`	看注释3

注释：

1. `char` 在Rust和 `Char` 在.NET有不同定义。Rust中一个 `char` 是4字节宽度 Unicode标量值，但.NET中，一个 `Char` 是2字节宽度存储UTF-16编码的字符。 更多信息，请看Rust `char` 文档。

2. Rust中单元 `()` (空元组) 是 *表达式值*，C#中最接近是 `void` 代表什么都不是。 `void` 不是 *表达式值* 不能用于指针和不安全代码。.NET中 `ValueTuple` 是一个空元组，但C#没有像 `()` 字面语法来表示它。 `ValueTuple` 可以在C#中使用，但这非常罕见。不像C#像Rust，F#有单元类型。

3. `void` 和 `object` 不是标量类型(标量类型如 `int` 在.NET类型结构中是 `object` 的子类型)，为方便理解它们包含在上面表格中。

另请参阅：

- 基本类型(Rust示例)

字符串

Rust中有两种字符串类型： `String` and `&str` 。前者分配在堆上和，后者是 `String` 或者 `&str` 切片。

这些.NET映射显示在下面表格中:

Rust	.NET	注释
`&mut str`	`Span<char>`	
`&str`	`ReadOnlySpan<char>`	
`Box<str>`	`String`	看注释1
`String`	`String`	
`String` (mutable)	`StringBuilder`	看注释1

Rust和.NET使用字符串有些不同，上面对应是一个很好的起点。一个不同点Rust字符串UTF-8是编码, 但.NET字符型是UTF-16编码。 此外.NET字符串是不可变的, 但Rust字符串可以是可变的当如此声明，例如 `let s = &mut String::from("hello");` 。

由于所有权的概念，在使用字符串方面也存在不同。有关字符串类型所有权的更多信息，请参阅Rust书.

注释：

1. Rust中 `Box<str>` 类型对应.NET中 `String` 类型。Rust中 `Box<str>` 和 `String` 类型不同点是前者 存储指针、大小而后者存储指针、大小、容量， `String` 允许大小增长。Rust中 `String` 声明可变，.NET类似是 `StringBuilder` 类型.

C#:

```
ReadOnlySpan<char> span = "Hello, World!";
string str = "Hello, World!";
StringBuilder sb = new StringBuilder("Hello, World!");
```

Rust:

```
let span: &str = "Hello, World!";
let str = Box::new("Hello World!");
let mut sb = String::from("Hello World!");
```

字符串字面值

.NET字符串字面值 `String` 类型是分配在堆上不可变的。Rust它是 `&'static str`, 不可变的具有全局生命周期、不能分配在堆上；嵌入在编译的二进制文件中。

C#

```
string str = "Hello, World!";
```

Rust

```
let str: &'static str = "Hello, World!";
```

C#原本字符串字面值对应Rust原始字符串字面值

C#

```
string str = @"Hello, \World/!";
```

Rust

```
let str = r#"Hello, \World/!"#;
```

C#UTF-8字符串字面值对应Rust字节字符串字面值。

C#

```
ReadOnlySpan<byte> str = "hello"u8;
```

Rust

```
let str = b"hello";
```

字符串插值

C#有内嵌的字符串插值特性，允许嵌入表达式在字符串字面值中。下面例子展示在C#怎么用字符串插值：

```
string name = "John";
int age = 42;
string str = $"Person {{ Name: {name}, Age: {age} }}";
```

Rust没有内嵌的字符串插值特性。替代用`format!`宏格式化字符串。面例子展示在Rust怎么用字符串插值：

```
let name = "John";
let age = 42;
let str = format!("Person {{ name: {name}, age: {age} }}");
```

在C#中自定义类和结构也可以用插值，因为每种类型可用`ToString()`方法，因为它继承自`object`。

```
class Person
{
    public string Name { get; set; }
    public int Age { get; set; }
```

```
    public override string ToString() =>
        $"Person {{ Name: {Name}, Age: {Age} }}";
}

var person = new Person { Name = "John", Age = 42 };
Console.Writeline(person);
```

在Rust中每种类型的实现/继承没有默认格式化。替代必须为每个类型实现`std::fmt::Display` trait，来转换字符串。

```
use std::fmt::*;

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

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

let person = Person {
    name: "John".to_owned(),
    age: 42,
};

println!("{person}");
```

另一个选项是使用`std::fmt::Debug` trait。所有标准类型实现`Debug` trait，可用于打印类型内部形式。 以下示例展示了如何使用`derive`属性打印自定义结构的内部形式，使用`Debug`宏。此声明为`Person`结构体 自动实现`Debug` trait：

```
#[derive(Debug)]
struct Person {
    name: String,
    age: i32,
}

let person = Person {
    name: "John".to_owned(),
    age: 42,
};

println!("{person:?}");
```

> 注释：使用:? 格式化说明符将使用`Debug` trait打印这结构体，省略它将使用`Display` trait。

另请参阅：

- Rust例子-调试

结构化类型

.NET常用对象和集合类型，它们在Rust映射

C#	Rust
`Array`	`Array`
`List`	`Vec`
`Tuple`	`Tuple`
`Dictionary`	`HashMap`

数组

Rust支持固定数组的方式和.NET相同

C#:

```
int[] someArray = new int[2] { 1, 2 };
```

Rust:

```
let someArray: [i32; 2] = [1,2];
```

列表

Rust对应`List<T>`是`Vec<T>`。Arrays可以转换到Vecs，反之亦然。

C#:

```
var something = new List<string>
{
    "a",
    "b"
};

something.Add("c");
```

Rust:

```
let mut something = vec![
    "a".to_owned(),
    "b".to_owned()
];

something.push("c".to_owned());
```

元组

C#:

```
var something = (1, 2)
Console.WriteLine($"a = {something.Item1} b = {something.Item2}");
```

Rust:

```
let something = (1, 2);
println!("a = {} b = {}", something.0, something.1);

// 支持解构
let (a, b) = something;
println!("a = {} b = {}", a, b);
```

> **注释**: Rust元组的元素不能命名，在C#中可以。唯一的方法访问元祖的原始是通过元素的索引或者解构元组。

字典

Rust对应`Dictionary<TKey, TValue>`是`Hashmap<K, V>`.

C#:

```
var something = new Dictionary<string, string>
{
    { "Foo", "Bar" },
    { "Baz", "Qux" }
};

something.Add("hi", "there");
```

Rust:

```
let mut something = HashMap::from([
    ("Foo".to_owned(), "Bar".to_owned()),
    ("Baz".to_owned(), "Qux".to_owned())
]);

something.insert("hi".to_owned(), "there".to_owned());
```

另请参阅：

- Rust标准库-集合

自定义类型

以下各节讨论各种主题和结构相关的开发自定义类型：

- 类
- 记录
- 结构体
- 接口
- 枚举类型
- Members

类

Rust没有类。它仅有structures or `struct`.

记录

Rust没有任何用于创作记录的构造，没有C#中`record struct`也没有`record class`。

结构体

Rust和C#中的结构体有一些相似之处：

- 它们是用`struct`关键字定义的，但在Rust中，`struct`只是定义数据/字段。函数和方法的行为方面定义 *实现块*（`impl`）。
- Rust中它们能实现多个traits，C#实现多个interfaces。
- 它们不能被子类化。
- 默认情况下，它们在栈上分配，除非：

- .NET中装箱或强制转换为interface。
- Rust包裹在一个智能指针中，例如`Box`，`Rc`/`Arc`。

在C#中，`struct`是一种在.NET 中对 *值类型* 进行建模的方法，这通常是 一些特定领域的原语或具有值相等语义的复合。在Rust中，`struct`是任何数据结构的主要构造（另一个是`enum`）。

在C#中的`struct`（或`record struct`）具有按值复制和默认值相等语义，但在Rust中，这只需要使用`#derive`属性和列出要实现的traits：

```
#[derive(Clone, Copy, PartialEq, Eq, Hash)]
struct Point {
    x: i32,
    y: i32,
}
```

C#/.NET中的值类型通常由开发人员设计为不可变的。这被认为是从语义上来说的最佳实践，但这种语言并不 防止设计进行破坏性修改或就地修改的`struct`。在Rust中也是如此。一种类型必须有意识地发展为不变的。

由于Rust没有类，因此类型层次结构基于通过traits和泛型实现子类共享行为，以及使用虚拟调度实现多态性trait objects。

在C#中考虑`struct`以下表示矩形：

```
struct Rectangle
{
    public Rectangle(int x1, int y1, int x2, int y2) =>
        (X1, Y1, X2, Y2) = (x1, y1, x2, y2);

    public int X1 { get; }
    public int Y1 { get; }
    public int X2 { get; }
    public int Y2 { get; }

    public int Length => Y2 - Y1;
    public int Width => X2 - X1;

    public (int, int) TopLeft => (X1, Y1);
    public (int, int) BottomRight => (X2, Y2);

    public int Area => Length * Width;
    public bool IsSquare => Width == Length;

    public override string ToString() => $"({X1}, {Y1}), ({X2}, {Y2})";
}
```

在Rust中对应是:

```rust
#![allow(dead_code)]

struct Rectangle {
    x1: i32, y1: i32,
    x2: i32, y2: i32,
}

impl Rectangle {
    pub fn new(x1: i32, y1: i32, x2: i32, y2: i32) -> Self {
        Self { x1, y1, x2, y2 }
    }

    pub fn x1(&self) -> i32 { self.x1 }
    pub fn y1(&self) -> i32 { self.y1 }
    pub fn x2(&self) -> i32 { self.x2 }
    pub fn y2(&self) -> i32 { self.y2 }

    pub fn length(&self) -> i32 {
        self.y2 - self.y1
    }

    pub fn width(&self)  -> i32 {
        self.x2 - self.x1
    }

    pub fn top_left(&self) -> (i32, i32) {
        (self.x1, self.y1)
    }

    pub fn bottom_right(&self) -> (i32, i32) {
        (self.x2, self.y2)
    }

    pub fn area(&self)  -> i32 {
        self.length() * self.width()
    }

    pub fn is_square(&self)  -> bool {
        self.width() == self.length()
    }
}

use std::fmt::*;

impl Display for Rectangle {
    fn fmt(&self, f: &mut Formatter<'_>) -> Result {
        write!(f, "({}, {}), ({}, {})", self.x1, self.y2, self.x2, self.y2)
    }
}
```

在C#中`struct`从`object`继承了`ToString`方法，因此它 *覆盖* 实现提供自定义字符串表示。由于Rust中没有继承， 因此类型的方式支持 *formatted* 表示，是通过实现`Display` trait。这使得该结构的实例能够参与格式化，如下面展示调用`println!`：

```
fn main() {
    let rect = Rectangle::new(12, 34, 56, 78);
    println!("Rectangle = {rect}");
}
```

接口

Rust没有C#/.NET中的interfaces。代替它有 *traits*。与interface类似，trait代表一个抽象及其成员形成一个契约，实现类型必须履行该契约。

C#/.NET中interfaces可以包含所有类型的成员，包括属性、索引器、事件、方法，静态和基于实例。同样Rust中traits有（基于实例的）方法，有关联的函数（想想C#/.NET中静态方法）和常量。

除了类层次结构之外，接口是通过动态调度完成横切抽象多态性功能核心的手段。它们使用接口编写通用代码表示抽象，而不关心实现它们的具体类型。Rust同样可以通过 *trait对象* 以有限的方式实现。 一个trait对象本质上是一个用`dyn`关键字跟trait名称标识的 *v-table*（虚拟表），如`dyn Shape`（其中`Shape`是trait名称）。Trait对象始终位于指针后面，无论是引用（例如`&dyn Shape`）或堆分配的`Box`（例如`Box<dyn Shape>`）。 这有点像在.NET中，其中接口是引用类型，因此转换为接口的值类型会自动装箱到托管堆上。前面提到的trait对象的传递限制是无法恢复实现类型。换句话说，虽然它相当常见于向下转换或测试一个接口作为其他接口实例或子类型或具体类型，在Rust中是不可能的（没有额外的努力和支持）。

枚举类型

在C#中，`enum`是一种将符号名称映射到整数值的值类型：

```
enum DayOfWeek
{
    Sunday = 0,
    Monday = 1,
    Tuesday = 2,
    Wednesday = 3,
    Thursday = 4,
    Friday = 5,
    Saturday = 6,
}
```

Rust 具有几乎 *相同* 的语法来执行相同的操作：

```
enum DayOfWeek
{
    Sunday = 0,
    Monday = 1,
    Tuesday = 2,
    Wednesday = 3,
```

```
    Thursday = 4,
    Friday = 5,
    Saturday = 6,
}
```

与.NET不同，Rust中`enum`类型的实例没有任何继承的预定义行为。它甚至不能参与平等检查就像`dow == DayOfWeek::Friday`一样简单。 为了使其在某种程度上达到C#中`enum`的函数同等水平，使用`#derive`属性来自动让宏实现通常需要的功能：

```
#[derive(Debug,     // 启用格式"{:?}"
         Clone,     // 复制需要
         Copy,      // 启用按值复制语义
         Hash,      // 启用用于映射类型的哈希功能
         PartialEq  // 启用值相等（==）
)]
enum DayOfWeek
{
    Sunday = 0,
    Monday = 1,
    Tuesday = 2,
    Wednesday = 3,
    Thursday = 4,
    Friday = 5,
    Saturday = 6,
}

fn main() {
    let dow = DayOfWeek::Wednesday;
    println!("Day of week = {dow:?}");

    if dow == DayOfWeek::Friday {
        println!("Yay! It's the weekend!");
    }

    // 强制为整数
    let dow = dow as i32;
    println!("Day of week = {dow:?}");

    let dow = dow as DayOfWeek;
    println!("Day of week = {dow:?}");
}
```

如上面的示例所示，`enum`可以被强制为其分配整数值，但与C#一样，相反的情况是不可能的（有时C#/.NET缺点是`enum`实例可以保存未表示的值）。相反，由开发人员提供这样的辅助函数：

```
impl DayOfWeek {
    fn try_from_i32(n: i32) -> Result<DayOfWeek, i32> {
        use DayOfWeek::*;
        match n {
            0 => Ok(Sunday),
```

```
            1 => Ok(Monday),
            2 => Ok(Tuesday),
            3 => Ok(Wednesday),
            4 => Ok(Thursday),
            5 => Ok(Friday),
            6 => Ok(Saturday),
            _ => Err(n)
        }
    }
}
```

`try_from_i32` 函数在 `Result` 中返回一个 `DayOfWeek`，表示成功（`Ok`）。如果 `n` 有效将按原样返回 `Result` 中的 `n` 指示失败（`Err`）：

```
let dow = DayOfWeek::try_from_i32(5);
println!("{dow:?}"); // prints: Ok(Friday)

let dow = DayOfWeek::try_from_i32(50);
println!("{dow:?}"); // prints: Err(50)
```

Rust中存在一些板条箱可以帮助实现这样从整数类型的映射，而不必手动编码。

Rust中的 `enum` 类型也可以作为设计（有区别的）联合的一种方式类型，允许不同的 _变体 保存特定于每个变体的数据。 例如：

```
enum IpAddr {
    V4(u8, u8, u8, u8),
    V6(String),
}

let home = IpAddr::V4(127, 0, 0, 1);
let loopback = IpAddr::V6(String::from("::1"));
```

这种形式的 `enum` 声明在C#中不存在，但可以用（类）记录模拟：

```
var home = new IpAddr.V4(127, 0, 0, 1);
var loopback = new IpAddr.V6("::1");

abstract record IpAddr
{
    public sealed record V4(byte A, byte B, byte C, byte D): IpAddr;
    public sealed record V6(string Address): IpAddr;
}
```

两者之间的区别在于Rust定义会产生一个 *closed类型* 变体。换句话说，编译器知道除了 `IpAddr::V4` 和 `IpAddr::V6` 之外，没有其他 `IpAddr` 的变体，并且它可以使用 使用这些知识进行更严格的检查。例如，在 `match`

中这种表达式类似于C#的`switch`表达式，Rust编译器会除非涵盖所有变体，否则将返回失败代码。相比之下，使用C#进行模拟 实际上创建了一个类层次结构（虽然表达得很简洁），并且由于`IpAddr`是一个抽象基类，因此它可以包含的所有类型的集合表示对编译器来说是未知的。

成员

构造函数

Rust没有构造函数的任何概念。代替你只需要编写工厂方法返回该类型实例的函数。工厂函数可以是独立或类型*关联函数*。在C#术语中，关联函数就像在类型上具有静态方法。 通常，如果`struct`存在一个工厂函数，它的名字就是`new`：

```
struct Rectangle {
    x1: i32, y1: i32,
    x2: i32, y2: i32,
}

impl Rectangle {
    pub fn new(x1: i32, y1: i32, x2: i32, y2: i32) -> Self {
        Self { x1, y1, x2, y2 }
    }
}
```

由于Rust函数（关联或其他）不支持重载，这个工厂函数必须具有唯一名称。例如，下面是一些`String`上可用的所谓构造函数或工厂函数的示例：

- `String::new`：创建一个空字符串。
- `String::with_capacity`：创建具有初始缓冲区容量的字符串。
- `String::from_utf8`：从UTF-8编码文本的字节创建字符串。
- `String::from_utf16`：从UTF-16编码文本的字节创建字符串。

在Rust的`enum`类型中，变体充当构造函数。更多信息请参见枚举类型部分。

另请参阅：

- 构造函数是静态的、固有的方法（C-CTOR）

方法（静态和基于实例）

与C#一样，Rust类型（包括`enum`和`struct`）可以具有静态和基于实例的方法。 在Rust语言中，*方法* 总是基于实例的，并且通过其第一个命名为`self`参数这一事实来识别。 `self`参数没有类型注释，因为它始终是属于方法。一个静态方法被称为 *关联函数*。在下面的例子中，`new`是一个关联函数，其余的（`length`，`width`和`area`）是以下类型的方法：

```
struct Rectangle {
    x1: i32, y1: i32,
    x2: i32, y2: i32,
}

impl Rectangle {
    pub fn new(x1: i32, y1: i32, x2: i32, y2: i32) -> Self {
        Self { x1, y1, x2, y2 }
    }

    pub fn length(&self) -> i32 {
        self.y2 - self.y1
    }

    pub fn width(&self)  -> i32 {
        self.x2 - self.x1
    }

    pub fn area(&self)  -> i32 {
        self.length() * self.width()
    }
}
```

常量

与C#一样，Rust中的类型可以具有常量。然而，最有趣的方面是注意Rust也允许将类型实例定义为常量：

```
struct Point {
    x: i32,
    y: i32,
}

impl Point {
    const ZERO: Point = Point { x: 0, y: 0 };
}
```

在C#中，同样的需要一个静态只读字段：

```
readonly record struct Point(int X, int Y)
{
    public static readonly Point Zero = new(0, 0);
}
```

事件

Rust没有内置类型成员支持通告和触发事件，就像C#的`event`关键字一样。

属性

在C#中，某个类型的字段通常是私有的。然后他们就是由属性成员使用访问器方法（`get`和`set`）读取或写入这些字段。 访问器方法可以包含额外的逻辑，例如，在设置值时验证值或在读取时计算值。 Rust只有方法其中字段getter命名（在Rust方法中，名称可以与字段共享相同的标识符）和setter使用`set_`前缀。

下面的示例显示了，对于Rust中类型的类似属性访问器方法的典型外观：

```
struct Rectangle {
    x1: i32, y1: i32,
    x2: i32, y2: i32,
}

impl Rectangle {
    pub fn new(x1: i32, y1: i32, x2: i32, y2: i32) -> Self {
        Self { x1, y1, x2, y2 }
    }

    // like property getters (each shares the same name as the field)

    pub fn x1(&self) -> i32 { self.x1 }
    pub fn y1(&self) -> i32 { self.y1 }
    pub fn x2(&self) -> i32 { self.x2 }
    pub fn y2(&self) -> i32 { self.y2 }

    // like property setters

    pub fn set_x1(&mut self, val: i32) { self.x1 = val }
    pub fn set_y1(&mut self, val: i32) { self.y1 = val }
    pub fn set_x2(&mut self, val: i32) { self.x2 = val }
    pub fn set_y2(&mut self, val: i32) { self.y2 = val }

    // like computed properties

    pub fn length(&self) -> i32 {
        self.y2 - self.y1
    }

    pub fn width(&self)  -> i32 {
        self.x2 - self.x1
    }

    pub fn area(&self)  -> i32 {
        self.length() * self.width()
    }
}
```

扩展方法

C#中的扩展方法使开发人员能够附加给现有类型新的静态绑定方法，无需修改类型的原始定义。 在下面的C#示例中，一个新的`Wrap`方法被添加给`StringBuilder`类 *通过扩展*：

```
using System;
using System.Text;
using Extensions; // (1)

var sb = new StringBuilder("Hello, World!");
sb.Wrap(">>> ", " <<<"); // (2)
Console.WriteLine(sb.ToString()); // 打印:  >>> Hello, World! <<<

namespace Extensions
{
    static class StringBuilderExtensions
    {
        public static void Wrap(this StringBuilder sb,
                                string left, string right) =>
            sb.Insert(0, left).Append(right);
    }
}
```

请注意，要使(2)扩展方法可用，包含类型扩展方法的命名空间必须导入(1)。Rust通过traits提供了非常相似的设施，称为 *扩展traits*。 下列Rust中的示例相当于上面的C#示例；它扩展了`String`使用`wrap`方法：

```
#![allow(dead_code)]

mod exts {
    pub trait StrWrapExt {
        fn wrap(&mut self, left: &str, right: &str);
    }

    impl StrWrapExt for String {
        fn wrap(&mut self, left: &str, right: &str) {
            self.insert_str(0, left);
            self.push_str(right);
        }
    }
}

fn main() {
    use exts::StrWrapExt as _; // (1)

    let mut s = String::from("Hello, World!");
    s.wrap(">>> ", " <<<"); // (2)
    println!("{s}"); // 打印: >>> Hello, World! <<<
}
```

就像在C#中一样，扩展trait中的方法使得（2）可用，必须导入扩展trait（1）。还要注意，扩展特性标识符`StrWrapExt`本身可以在导入时通过`_`丢弃 而不影响`String`的`wrap`的可用性。

可见性/访问修改器

C#有许多可访问性或可见性修饰符：

- `private`
- `protected`
- `internal`
- `protected internal` (family)
- `public`

在Rust中，编译是由模块树构建的，其中模块包含并定义 *项目*，如类型、特征、枚举、常量和函数。默认情况下，几乎所有内容都是私有的。 一个例外是，对于例如，公共trait中 *关联项目*，默认情况下是公共的。 这类似于C#接口的成员在没有任何公共声明方式，默认情况下源代码中的修饰符是公共的。Rust只有`pub`修改器来更改相对于模块树的可见性。 这里是`pub`的变体，可更改公众可见性的范围：

- `pub(self)`
- `pub(super)`
- `pub(crate)`
- `pub(in PATH)`

有关更多详细信息，请参阅Rust参考的可见性和隐私部分

下表是C#和Rust修饰符的映射近似值：

C#	Rust	注释
`private`	(default)	看注释 1.
`protected`	N/A	看注释 2.
`internal`	`pub(crate)`	
`protected internal` (family)	N/A	看注释 2.
`public`	`pub`	

1. 没有关键字表示私有可见性；这是Rust中的默认设置。

2. 由于 Rust 中没有基于类的类型层次结构，因此没有等同于`protected`。

可变性

在C#中设计类型时，开发人员有责任确定类型是可变的还是不可变的；是否支持破坏性或非破坏性可变。 C#确实支持不可变设计类型通过 *记录声明*（`record class`或`readonly record struct`）。 在Rust中可变性通过在类型方法上的`self`参数，如下例所示：

```
struct Point { x: i32, y: i32 }

impl Point {
    pub fn new(x: i32, y: i32) -> Self {
        Self { x, y }
```

```
    }

    // self是不可变的

    pub fn x(&self) -> i32 { self.x }
    pub fn y(&self) -> i32 { self.y }

    // self是可变的

    pub fn set_x(&mut self, val: i32) { self.x = val }
    pub fn set_y(&mut self, val: i32) { self.y = val }
}
```

在C#中，可以使用`with`进行非破坏性可变：

```
var pt = new Point(123, 456);
pt = pt with { X = 789 };
Console.WriteLine(pt.ToString()); // 打印： Point { X = 789, Y = 456 }

readonly record struct Point(int X, int Y);
```

Rust中没有`with`，但Rust模仿类似内容，它可以融入类型设计中：

```
struct Point { x: i32, y: i32 }

impl Point {
    pub fn new(x: i32, y: i32) -> Self {
        Self { x, y }
    }

    pub fn x(&self) -> i32 { self.x }
    pub fn y(&self) -> i32 { self.y }

    // 以下方法通过self并返回一个新实例

    pub fn set_x(self, val: i32) -> Self { Self::new(val, self.y) }
    pub fn set_y(self, val: i32) -> Self { Self::new(self.x, val) }
}
```

在C#中，`with`也可以与`struct`一起使用（而不是record），公开其读写字段：

```
struct Point
{
    public int X;
    public int Y;

    public override string ToString() => $"({X}, {Y})";
}
```

```
var pt = new Point { X = 123, Y = 456 };
Console.WriteLine(pt.ToString()); // 打印： (123, 456)
pt = pt with { X = 789 };
Console.WriteLine(pt.ToString()); // 打印： (789, 456)
```

Rust有一个 *struct更新语法*，可能看起来很相似：

```
mod points {
    #[derive(Debug)]
    pub struct Point { pub x: i32, pub y: i32 }
}

fn main() {
    use points::Point;
    let pt = Point { x: 123, y: 456 };
    println!("{pt:?}"); // 打印： Point { x: 123, y: 456 }
    let pt = Point { x: 789, ..pt };
    println!("{pt:?}"); // 打印： Point { x: 789, y: 456 }
}
```

但是，虽然C#中的`with`会进行非破坏性可变（复制然后更新），struct更新语法执行（部分）*移动* 并仅限适用于字段。 由于语法需要访问类型的字段，因此它通常更常见的是在Rust模块中使用它，该模块可以访问其类型的私有详细信息。

本地函数

C#和Rust提供本地函数，但Rust中的本地函数仅限于等价于C#中的静态局部函数。 换句话说，Rust中的本地函数不能使用其周围词法范围中的变量；但是闭包可以。

Lambda和闭包

C#和Rust允许将函数用作第一类值，从而实现编写 *高阶函数*。 高阶函数本质上是接受其他函数作为参数以允许调用方参与被调用函数的代码。 在C#中，*类型安全函数指针* 最常见的是`Func`和`Action`。 C#语言允许这些委托的临时实例是通过 *lambda表达式* 创建。

Rust也有函数指针，`fn`类型是最简单的：

```
fn do_twice(f: fn(i32) -> i32, arg: i32) -> i32 {
    f(arg) + f(arg)
}

fn main() {
    let answer = do_twice(|x| x + 1, 5);
```

```
    println!("The answer is: {}", answer); // 打印:  The answer is: 12
}
```

However, Rust makes a distinction between *function pointers* (where `fn` defines a type) and *closures*: a closure can reference variables from its surrounding lexical scope, but not a function pointer. While C# also has function pointers (`*delegate`), the managed and type-safe equivalent would be a static lambda expression.

但是，Rust区分 *函数指针*（其中`fn`定义类型）和 *闭包*：闭包作为变量引用在围绕词法范围，但不是函数指针。而C#也有函数指针（`*delegate`），托管和类型安全等效的是一个静态lambda表达式。

接受闭包的函数和方法是使用泛型类型编写的，这些泛型类型绑定到表示函数的特征之一：`Fn`、`FnMut`和`FnOnce`。 当需要值通过函数指针或闭包，Rust开发人员使用一个 *闭包表达式*（如上面的例子`|x| x + 1`），它转换与C#中的lambda表达式相同。 闭包表达式是创建函数指针还是闭包取决于关于闭包表达式是否引用其上下文。

当闭包从其环境中捕获变量时，所有权规则开始发挥作用，因为所有权最终以关闭而告终。 查看更多信息，请参阅"将捕获的值移出闭包和Fn Traits"部分。

变量

考虑以下有关C#中变量赋值的示例：

```
int x = 5;
```

在Rust中也是如此：

```
let x: i32 = 5;
```

到目前为止，这两种语言之间唯一明显的区别是类型声明的位置不同。 此外，C#和Rust都是类型安全：编译器保证存储在变量中的值始终是指定类型的。 使用编译器的自动推断变量类型的能力。在C#中：

```
var x = 5;
```

在Rust中：

```
let x = 5;
```

扩展第一个示例以更新变量的值时（重新赋值），C#和Rust的行为不同：

```
var x = 5;
x = 6;
Console.WriteLine(x); // 6
```

在Rust中，相同的语句不会编译：

```
let x = 5;
x = 6; // 错误：无法为不可变变量“x”赋值两次。
println!("{}", x);
```

在Rust中, 变量默认是 *不可变*。 将值绑定到名称后，无法更改变量的值。 变量可以通过以下方式 *可变* 在变量名称前面添加 `mut`：

```
let mut x = 5;
x = 6;
println!("{}", x); // 6
```

Rust提供了一种替代方法来修复上面的示例，它不需要通过可变性，而是用变量 *遮蔽*：

```
let x = 5;
let x = 6;
println!("{}", x); // 6
```

C#也支持遮蔽，例如，局部变量可以对字段进行遮蔽，类型成员可以遮蔽基类型中的成员。 在Rust中，上面的例子该遮蔽还允许在不更改名称情况下更改变量的类型，如果要将数据转换为不同的类型，而不必每次都想出一个不同的名称。

另请参阅：

- 数据争用和争用条件 以获取有关可变性影响的更多信息
- 范围和遮蔽
- 内存管理 有关 *移动* 和 *所有权* 的解释

命名空间

命名空间在.NET中用于组织类型，以及用于控制项目中的类型和方法的范围。

在Rust中，命名空间指的是一个不同的概念。 等效的命名空间在Rust中是一个模块。 对于C#和Rust，项目的可见性可以使用访问修饰符和可见性修饰符进行限制。 在Rust，默认可见性是 *private*（只有少数例外）。 C#的 `public` 在Rust中是 `pub`，而 `internal` 对应于 `pub(crate)`。 有关更细粒度的访问控制，请参阅 可见性修饰符。

相等

在C#中比较相等时，这在某些情况下是指测试_等价性_（也称为 *值相等性*），而在其他情况下是指测试 *引用平等性*，即测试两个变量是否引用内存中的同一个底层对象。 每个自定义类型都可以进行比较相等性，因为它继承自 `System.Object` （对于值类型，则为 `System.ValueType`，它继承自 `System.Object`），使用上述任一语义。

例如，在C#中比较等价性和引用相等性时：

```
var a = new Point(1, 2);
var b = new Point(1, 2);
var c = a;
Console.WriteLine(a == b); // (1) True
Console.WriteLine(a.Equals(b)); // (1) True
Console.WriteLine(a.Equals(new Point(2, 2))); // (1) False
Console.WriteLine(ReferenceEquals(a, b)); // (2) False
Console.WriteLine(ReferenceEquals(a, c)); // (2) True

record Point(int X, int Y);
```

1. 相等运算符 `==` 和 `record Point` 上的 `Equals` 方法比较值相等性，因为记录默认支持值类型相等性。
2. 比较引用相等性测试变量是否引用内存中的相同底层对象。

在Rust中相等：

```
#[derive(Copy, Clone)]
struct Point(i32, i32);

fn main() {
    let a = Point(1, 2);
    let b = Point(1, 2);
    let c = a;
    println!("{}", a == b); // Error: "an implementation of `PartialEq<_>` might be missing for
`Point`"
    println!("{}", a.eq(&b));
    println!("{}", a.eq(&Point(2, 2)));
}
```

上面的编译器错误说明，在Rust中，相等性比较 *总是* 与特征实现相关。要支持使用 `==` 的比较， 类型必须实现 `PartialEq`。

修复上述示例意味着为 `Point` 派生 `PartialEq`。默认情况下，派生 `PartialEq` 将比较所有字段的相等性， 因此必须自己实现 `PartialEq`。这相当于C#中记录的相等性。

```
#[derive(Copy, Clone, PartialEq)]
struct Point(i32, i32);
```

```
fn main() {
    let a = Point(1, 2);
    let b = Point(1, 2);
    let c = a;
    println!("{}", a == b); // true
    println!("{}", a.eq(&b)); // true
    println!("{}", a.eq(&Point(2, 2))); // false
    println!("{}", a.eq(&c)); // true
}
```

另请参阅：

- `Eq` 为更严格的 `PartialEq` 版本。

泛型

C#中的泛型提供了一种为类型和方法创建定义的方法可以在其他类型上进行参数化。 这提高了代码重用和类型安全性和性能（例如，避免运行时强制转换）。 考虑以下示例为任何值添加时间戳的泛型类型：

```
using System;

sealed record Timestamped<T>(DateTime Timestamp, T Value)
{
    public Timestamped(T value) : this(DateTime.UtcNow, value) { }
}
```

Rust也有如上面所示的等价泛型：

```
use std::time::*;

struct Timestamped<T> { value: T, timestamp: SystemTime }

impl<T> Timestamped<T> {
    fn new(value: T) -> Self {
        Self { value, timestamp: SystemTime::now() }
    }
}
```

另请参阅：

- 泛型数据类型

泛型类型约束

在C#中，通过使用 `where` 语句约束泛型类型。 以下示例显示了C#中的此类约束：

```
using System;

// 注意：记录自动实现`IEquatable`。以下实现明确地显示了这一点，以便与Rust进行比较。
sealed record Timestamped<T>(DateTime Timestamp, T Value) :
    IEquatable<Timestamped<T>>
    where T : IEquatable<T>
{
    public Timestamped(T value) : this(DateTime.UtcNow, value) { }

    public bool Equals(Timestamped<T>? other) =>
        other is { } someOther
        && Timestamp == someOther.Timestamp
        && Value.Equals(someOther.Value);

    public override int GetHashCode() => HashCode.Combine(Timestamp, Value);
}
```

在Rust中也可以实现同样的目标

```
use std::time::*;

struct Timestamped<T> { value: T, timestamp: SystemTime }

impl<T> Timestamped<T> {
    fn new(value: T) -> Self {
        Self { value, timestamp: SystemTime::now() }
    }
}

impl<T> PartialEq for Timestamped<T>
    where T: PartialEq {
    fn eq(&self, other: &Self) -> bool {
        self.value == other.value && self.timestamp == other.timestamp
    }
}
```

泛型类型约束在Rust中称为bounds。

在C#版本中，`Timestamped<T>`实例 *仅仅* 可以为`T`创建，该实例自己实现`IEquatable<T>`， 但请注意 Rust 版本更多灵活，因为它的`Timestamped<T>` *有条件实现* `PartialEq`。 这意味着不可等式的`T`仍然可以创建`Timestamped<T>`实例，但随后`Timestamped<T>`将不会 通过`PartialEq`表示这样的`T`方式实现平等。

另请参阅：

- 作为参数的特征
- 实现特征的返回类型

多态性

Rust不支持类和子类，因此多态性不能以与C#相同的方式实现。

另请参阅：

- 使用 *trait objects* 进行虚拟调度，如 结构 中所述
- 泛型
- 继承
- 运算符重载

继承

正如 结构体 一节中所解释的，Rust不提供如C#中继承（基于类）。 在结构之间提供共享行为的一种方法是通过利用特征。 但是与C#中的 *接口继承* 类似，Rust允许通过使用*超级特征*。

异常处理

在.NET中，异常是继承自`System.Exception`类。 如果代码段中出现问题，则引发异常。引发的异常将传递到堆栈中直到应用程序处理它或程序终止。

Rust没有异常，而是区分 *可恢复* 和 *不可恢复* 错误。 可恢复错误表示应该报告但程序仍继续运行的问题。 操作结果失败的可恢复错误属于`Result<T, E>`类型，其中`E`是错误变体的类型。 当程序遇到不可恢复的错误时，`panic!`宏会停止执行。不可恢复的错误始终是漏洞症状。

自定义错误类型

在.NET中，自定义异常源自`Exception`类。 有关如何创建用户定义异常 的文档提到了以下示例：

```
public class EmployeeListNotFoundException : Exception
{
    public EmployeeListNotFoundException() { }

    public EmployeeListNotFoundException(string message)
        : base(message) { }

    public EmployeeListNotFoundException(string message, Exception inner)
        : base(message, inner) { }
}
```

在Rust中，可以通过实现`Error`特征来实现对错误值的基本期望。 Rust中最小的用户定义错误实现是：

```
#[derive(Debug)]
pub struct EmployeeListNotFound;

impl std::fmt::Display for EmployeeListNotFound {
    fn fmt(&self, f: &mut std::fmt::Formatter<'_>) -> std::fmt::Result {
```

```
            f.write_str("Could not find employee list.")
        }
    }

    impl std::error::Error for EmployeeListNotFound {}
```

Rust中的`Error::source()`方法相当于.NET的`Exception.InnerException`属性。 但是不需要为`Error::source()`提供实现，统一（默认）实现将返回`None`。

引发异常

要在C#中引发异常，请抛出异常的一个实例：

```
    void ThrowIfNegative(int value)
    {
        if (value < 0)
        {
            throw new ArgumentOutOfRangeException(nameof(value));
        }
    }
```

对于Rust中的可恢复错误，从方法返回`Ok`或`Err`变体：

```
    fn error_if_negative(value: i32) -> Result<(), &'static str> {
        if value < 0 {
            Err("Specified argument was out of the range of valid values. (Parameter 'value')")
        } else {
            Ok(())
        }
    }
```

`panic!`宏会产生无法恢复的错误：

```
    fn panic_if_negative(value: i32) {
        if value < 0 {
            panic!("Specified argument was out of the range of valid values. (Parameter 'value')")
        }
    }
```

错误传播

在.NET中，异常会沿堆栈向上传递，直到被处理或程序终止。 在Rust中，不可恢复的错误表现类似，但处理它们并不常见。

但是，可恢复错误需要明确传播和处理。 它们的存在始终由Rust函数或方法签名指示。 捕获异常允许您根据C#中错误的存在与否采取行动：

```
void Write()
{
    try
    {
        File.WriteAllText("file.txt", "content");
    }
    catch (IOException)
    {
        Console.WriteLine("Writing to file failed.");
    }
}
```

在Rust中，这大致相相当于：

```
fn write() {
    match std::fs::File::create("temp.txt")
        .and_then(|mut file| std::io::Write::write_all(&mut file, b"content"))
    {
        Ok(_) => {}
        Err(_) => println!("Writing to file failed."),
    };
}
```

通常，可恢复错误只需传播，而不需要处理。 为此，方法签名需要与传播错误的类型兼容。 `?` 运算符以符合人体工程学的方式传播错误：

```
fn write() -> Result<(), std::io::Error> {
    let mut file = std::fs::File::create("file.txt")?;
    std::io::Write::write_all(&mut file, b"content")?;
    Ok(())
}
```

注意：要使用问号运算符传播错误，错误实现需要 *兼容*，如*传播错误的快捷方式*中所述。 最通用的"兼容"错误类型是错误[特征对象] `Box<dyn Error>`。

堆栈跟踪

在.NET中抛出未处理的异常将导致运行时打印堆栈跟踪，从而允许使用其他上下文调试问题。

对于Rust中不可恢复的错误， `panic!` Backtraces 提供了类似的行为。

稳定版Rust中的可恢复错误尚不支持Backtraces，但目前在实验性Rust中使用 [provide方法]支持它。

可空性和可选性

在C#中，`null`通常用于表示缺失、不存在或逻辑上未初始化的值。例如：

```
int? some = 1;
int? none = null;
```

Rust没有`null`，因此没有可空上下文可启用。可选或缺失值则由`Option<T>`表示。 上述C#代码在Rust中的等效代码如下：

```
let some: Option<i32> = Some(1);
let none: Option<i32> = None;
```

Rust中的`Option<T>`实际上与F中的`'T option`相同。

可选性的控制流程

在C#中，当使用可空值时，您可能一直使用`if` / `else`语句来控制流程。

```
uint? max = 10;
if (max is { } someMax)
{
    Console.WriteLine($"The maximum is {someMax}."); // The maximum is 10.
}
```

您可以使用模式匹配在 Rust 中实现相同的行为： 使用`if let`甚至会更简洁：

```
let max = Some(10u32);
if let Some(max) = max {
    println!("The maximum is {}.", max); // The maximum is 10.
}
```

空条件运算符

The null-conditional operators (`?.` and `?[]`) make dealing with `null` in C# more ergonomic. In Rust, they are best replaced by using the `map` method. The following snippets show the correspondence:

空条件运算符（`?.`和`?[]`）使C#中处理`null`更加符合人体工程学。 在Rust中，最好使用`map`方法来替换它们。 以下代码片段显示了对应关系：

```
string? some = "Hello, World!";
string? none = null;
Console.WriteLine(some?.Length); // 13
Console.WriteLine(none?.Length); // (blank)
```

```
let some: Option<String> = Some(String::from("Hello, World!"));
let none: Option<String> = None;
println!("{:?}", some.map(|s| s.len())); // Some(13)
println!("{:?}", none.map(|s| s.len())); // None
```

空合并运算符

当可空值为`null`时，通常使用空合并运算符（`??`）默认为另一个值：

```
int? some = 1;
int? none = null;
Console.WriteLine(some ?? 0); // 1
Console.WriteLine(none ?? 0); // 0
```

在Rust中，你可以使用`unwrap_or`来获得相同的行为：

```
let some: Option<i32> = Some(1);
let none: Option<i32> = None;
println!("{:?}", some.unwrap_or(0)); // 1
println!("{:?}", none.unwrap_or(0)); // 0
```

注意：如果默认值的计算成本很高，您可以改用`unwrap_or_else`。 它以闭包作为参数，允许您延迟初始化默认值。

空值包容运算符

空值包容运算符(`!`)在Rust中没有对应等效构造，因为它仅影响C#中编译器的静态流分析。、 在Rust中，无需使用它的替代品。

丢弃

在C#中，丢弃 表示编译器和其他程序忽略表达式的结果（或部分）。

有多种情况可以应用此方法，例如，作为一个基本示例，忽略表达式的结果。在C#中，它看起来像：

```
_ = city.GetCityInformation(cityName);
```

在Rust中，忽略表达式的结果看起来相同的：

```
_ = city.get_city_information(city_name);
```

在C#中，丢弃也适用于解构元组：

```
var (_, second) = ("first", "second");
```

并且，在Rust中同样如此：

```
let (_, second) = ("first", "second");
```

除了解构元组之外，Rust还提供了使用`..`解构结构体和枚举的功能，其中`..`代表类型的剩余部分：

```
struct Point {
    x: i32,
    y: i32,
    z: i32,
}

let origin = Point { x: 0, y: 0, z: 0 };

match origin {
    Point { x, .. } => println!("x is {}", x), // x is 0
}
```

进行模式匹配时，丢弃或忽略匹配表达式的一部分通常很有用，例如在C#中：

```
_ = ("first", "second") switch
{
    ("first", _) => "first element matched",
    (_, _) => "first element did not match"
};
```

同样，在Rust中这看起来几乎完全相同：

```
_ = match ("first", "second")
{
    ("first", _) => "first element matched",
    (_, _) => "first element did not match"
};
```

转换

C#和Rust在编译时都是静态类型的。 因此，在声明变量后，禁止将不同类型的值（除非它可以隐式转换为目标类型）赋给变量。 在C#中有几种方法可以转换，Rust中具有等效类型的类型。

隐式转换

C#隐式转换，也存在Rust中（称为类型强制）。 请看以下示例：

```
int intNumber = 1;
long longNumber = intNumber;
```

Rust对于允许的类型强制有更严格的限制：

```
let int_number: i32 = 1;
let long_number: i64 = int_number; // error: expected `i64`, found `i32`
```

使用 子类型 进行有效隐式转换的示例为：

```
fn bar<'a>() {
    let s: &'static str = "hi";
    let t: &'a str = s;
}
```

另请参考：

- 解除强制
- 子类型和变异

显式转换

如果转换可能导致信息丢失，则C#需要使用强制转换表达式进行显式转换：

```
double a = 1.2;
int b = (int)a;
```

显式转换可能会在运行时失败，并在 *向下转换* 时出现诸如 `OverflowException` 或 `InvalidCastException` 之类的异常。

Rust不提供基元类型之间的强制，而是使用显示转换通过 `as` 关键字（转换）。 在Rust中转换不会引起恐慌。

```
let int_number: i32 = 1;
let long_number: i64 = int_number as _;
```

自定义转换

通常，.NET类型提供用户定义的转换运算符来将一种类型转换为另一种类型。 此外， `System.IConvertible` 可用于将一种类型转换为另一种类型。

在Rust中，标准库包含一个抽象，用于将值转换为不同类型的值，形式为 `From` 特征及其倒数 `Into`。 在为类型实现 `From` 时，会自动提供 `Into` 的默认实现（在Rust中称为 *全面实现*）。 以下示例说明了两种此类类型转换：

```
fn main() {
    let my_id = MyId("id".into()); // 由于`String`的`From<&str>`特征实现，`into()`是自动实现的。
    println!("{}", String::from(my_id)); // 这使用`From<MyId>`实现`String`。
}

struct MyId(String);

impl From<MyId> for String {
    fn from(MyId(value): MyId) -> Self {
        value
    }
}
```

另请参阅：

- `TryFrom` 和 `TryInto` 针对可能失败的 `From` 和 `Into` 版本。

运算符重载

自定义类型可以重载C#中的 *重载运算符*。考虑一下以下C#中的示例：

```
Console.WriteLine(new Fraction(5, 4) + new Fraction(1, 2));  // 14/8

public readonly record struct Fraction(int Numerator, int Denominator)
```

```
{
    public static Fraction operator +(Fraction a, Fraction b) =>
        new(a.Numerator * b.Denominator + b.Numerator * a.Denominator, a.Denominator *
b.Denominator);

    public override string ToString() => $"{Numerator}/{Denominator}";
}
```

在Rust中，许多运算符可以通过重载tracts。 这是可能的因为运算符是方法调用的语法糖。例如`a + b`中的`+`运算符调用`add`方法（请参见运算符重载）：

```
use std::{fmt::{Display, Formatter, Result}, ops::Add};

struct Fraction {
    numerator: i32,
    denominator: i32,
}

impl Display for Fraction {
    fn fmt(&self, f: &mut Formatter<'_>) -> Result {
        f.write_fmt(format_args!("{}/{}", self.numerator, self.denominator))
    }
}

impl Add<Fraction> for Fraction {
    type Output = Fraction;

    fn add(self, rhs: Fraction) -> Fraction {
        Fraction {
            numerator: self.numerator * rhs.denominator + rhs.numerator * self.denominator,
            denominator: self.denominator * rhs.denominator,
        }
    }
}

fn main() {
    println!(
        "{}",
        Fraction { numerator: 5, denominator: 4 } + Fraction { numerator: 1, denominator: 2 }
    ); // 14/8
}
```

文档注释

C#提供了一种使用注释语法为类型编写API文档的机制其中包含XML文本。 C#编译器生成一个XML文件，该文件包含表示评论和API签名的结构化数据。 其他工具可以处理该输出，以不同的方式提供人类可读的文档形式。 C#中的一个简单示例：

```
/// <summary>
/// This is a document comment for <c>MyClass</c>.
/// </summary>
public class MyClass {}
```

在Rust文档注释中提供了与C#文档注释等效的内容。 Rust中的文档注释使用Markdown语法。 `rustdoc`是Rust代码的文档编译器，通常通过调用`cargo doc`，它将注释编译成文档。 例如：

```
/// This is a doc comment for `MyStruct`.
struct MyStruct;
```

在中.NET SDK中没有与`cargo doc`等效，例如`dotnet doc`。

另请参阅:

- 如何编写文档
- 文档测试

内存管理

像C#和.NET 一样，Rust 也 *内存安全* 来避免与内存访问有关的类错误，最终成为许多安全的来源软件中的漏洞。 但是，Rust可以保证内存安全编译时；没有运行时（如CLR）进行检查。 一个例外情况是数组绑定检查，这些检查是由编译后的代码完成的在运行时，无论是Rust编译器还是.NET中的JIT 编译器。 像C#一样，它也是可以在Rust中编写不安全的代码，事实上，两种语言甚至共享相同的关键字，*字面意思* `unsafe`， 以标记不再保证内存安全的函数和代码块。

Rust没有垃圾收集器（GC）。 所有内存管理完全是开发者的责任。 也就是说，*Rust安全* 有相关所有权规则确保内存在不再使用时立即释放（例如当离开块或函数的范围时）。 编译器执行通过（编译时）静态分析，帮助管理这一点，这是一项艰巨的工作通过[所有权]规则。 如果违反，编译器将拒绝代码出现编译错误。

在.NET里面，除了GC根（静态字段、线程堆栈上的局部变量、CPU寄存器、句柄等）之外，没有内存所有权的概念。 GC在收集过程中从根开始确定通过以下引用并清除其余引用，所有正在使用的内存。 设计类型和编写代码时，.NET开发人员可以不在意所有权、内存管理、甚至垃圾收集器在大多数情况下是如何工作的， 除非对性能敏感的代码需要注意数量以及在堆上分配对象的速率。 相比之下，Rust的所有权规则要求开发者明确思考和表达所有权在任何时候都会影响到从功能设计、类型、数据结构以及代码的编写方式。 除此之外，Rust关于如何使用数据的严格规则，使得它可以在编译时识别， 数据竞态条件以及损坏问题（需要线程安全）这可能在运行时发生。 本节仅关注内存管理和所有权。

某些内存只能有一个所有者，无论是在堆栈上还是在堆上，在Rust中的任何给定时间支持一个结构。 编译器分配生命周期并跟踪所有权。可以传递或让出所有权，在Rust中称为 *转移*。 这些想法简要地在下面的Rust代码示例中进行了说明：

```
#![allow(dead_code, unused_variables)]
```

```
struct Point {
    x: i32,
    y: i32,
}

fn main() {
    let a = Point { x: 12, y: 34 }; // a有point所有权
    let b = a;                      // b现在有point所有权
    println!("{}, {}", a.x, a.y);   // 编译错误！
}
```

`main` 中的第一个语句将分配 `Point`，并且该内存将归 `a` 所有。 在第二个语句中，所有权从 `a` 转移到 `b`，并且 `a` 不再可用，因为它不再拥有任何东西或代表有效内存。 最后一个尝试通过 `a` 打印 point 字段的语句将编译失败。假设 `main` 已修复为如下所示：

```
fn main() {
    let a = Point { x: 12, y: 34 }; // a有point所有权
    let b = a;                      // b现在有point所有权
    println!("{}, {}", b.x, b.y);   // 用b，没问题
}   // b之后point被丢弃
```

请注意，当`main`退出时，`a`和`b`将超出范围。 由于堆栈返回到调用`main`之前的状态，`b`后面的内存将被释放。 在Rust中，人们说`b`后面的point被_丢弃_。 但是，请注意，由于`a`将其对该point的所有权让给了`b`，因此当`a`超出范围时，没有什么可丢弃的。

Rust中的 `struct` 可以通过实现 `Drop` 特征来定义在丢弃实例时执行的代码。

C#中 *dropping* 的大致对应物是类 finalizer，但虽然GC会在未来的某个时间点 *自动* 调用终结器， 但Rust中的dropping始终是即时且确定的；也就是说，它发生在编译器根据范围和生命周期确定实例没有所有者时。 在.NET中， `Drop` 的对应物是 `IDisposable`，并由类型实现以释放它们持有的任何非托管资源或内存。 *确定性处置* 不是强制执行或保证的，但C#中的`using`语句通常用于确定一次性类型的实例，以便它在`using`语句块的末尾被确定地处置。

Rust具有全局生命周期的概念，用`'static`表示，这是一个保留的生命周期说明符。 C#中一个非常粗略的近似值是静态的 *只读* 类型的字段。

在C#和.NET 中，引用可以自由共享而无需过多考虑， 因此单一所有者和让出/移动所有权的想法在Rust中似乎非常有限，但可以使用智能指针类型 `Rc` 在Rust中拥有 *共享所有权*； 它添加了引用计数。每次克隆智能指针时，引用计数都会增加。 当克隆删除时，引用计数会减少。当引用计数达到零时，智能指针背后的实际实例将被删除。以下基于前一个示例的示例说明了这些要点：

```
#![allow(dead_code, unused_variables)]

use std::rc::Rc;

struct Point {
    x: i32,
    y: i32,
```

```
}

impl Drop for Point {
    fn drop(&mut self) {
        println!("Point dropped!");
    }
}

fn main() {
    let a = Rc::new(Point { x: 12, y: 34 });
    let b = Rc::clone(&a); // share with b
    println!("a = {}, {}", a.x, a.y); // okay to use a
    println!("b = {}, {}", b.x, b.y);
}

// prints:
// a = 12, 34
// b = 12, 34
// Point dropped!
```

请注意：

- `Point` 实现了 `Drop` 特征的 `drop` 方法，并在丢弃 `Point` 实例时打印一条消息。
- 在 `main` 中创建的point被包装在智能指针 `Rc` 后面，因此智能指针拥有该point而不是 `a` 。
- `b` 获取智能指针的克隆，从而有效地将引用计数增加到 2。与前面的示例不同，其中 `a` 将其对point的所有权转让给 `b` ， `a` 和 `b` 都拥有智能指针的各自不同克隆，因此可以继续使用 `a` 和 `b` 。
- 编译器将确定 `a` 和 `b` 在 `main` 结束时超出范围时注入调用以丢弃每个。 `Rc` 的 `Drop` 实现将减少引用计数，并且如果引用计数已达到零，还将丢弃其拥有的内容。 当发生这种情况时， `Point` 的 `Drop` 实现将打印消息“Point 已丢弃！”。 该消息只打印一次，表明只创建、共享和丢弃了一个point。

`Rc` 不是线程安全的。对于多线程程序中的共享所有权，Rust标准库提供了 `Arc` 。 Rust语言将阻止跨线程使用 `Rc` 。

在.NET中，值类型（如 C# 中的 `enum` 和 `struct` ）位于栈中，而引用类型（C# 中的 `interface` 、 `record class` 和 `class` ）则分配在堆中。 在Rust中，类型的种类（Rust中基本上是 `enum` 或 `struct` ）并不决定后备内存最终将位于何处。 默认情况下，它始终位于栈中，但.NET和C#具有装箱值类型的概念，即将它们复制到堆中，在堆上分配类型的方法是使用 `Box` 对其进行装箱：

```
let stack_point = Point { x: 12, y: 34 };
let heap_point = Box::new(Point { x: 12, y: 34 });
```

与 `Rc` 和 `Arc` 一样， `Box` 是一个智能指针，但与 `Rc` 和 `Arc` 不同，它独占拥有其背后的实例。 所有这些智能指针都分配堆上其类型参数“T”的实例。

C#中的 `new` 关键字创建一个类型的实例，而成员因为您在示例中看到的 `Box::new` 和 `Rc::new` 似乎具有类似的目的， `new` 在Rust中没有特殊名称。 它只是一个 *约定名称*，这意味着一个工厂。 事实上，他们被称为类型的我 *相关函数*，这是Rust静态方法的方式。

资源管理

上一节内存管理介绍了.NET与Rust之间的差异在GC、所有权和终结器。 强烈推荐阅读它。

本节仅限于提供虚构的示例 _数据库连接_涉及SQL连接的closed/disposed/dropped

```
{
    using var db1 = new DatabaseConnection("Server=A;Database=DB1");
    using var db2 = new DatabaseConnection("Server=A;Database=DB2");

    // ...code using "db1" and "db2"...
}   // "Dispose" of "db1" and "db2" called here; when their scope ends

public class DatabaseConnection : IDisposable
{
    readonly string connectionString;
    SqlConnection connection; //this implements IDisposable

    public DatabaseConnection(string connectionString) =>
        this.connectionString = connectionString;

    public void Dispose()
    {
        //Making sure to dispose the SqlConnection
        this.connection.Dispose();
        Console.WriteLine("Closing connection: {this.connectionString}");
    }
}
```

```
struct DatabaseConnection(&'static str);

impl DatabaseConnection {
    // ...functions for using the database connection...
}

impl Drop for DatabaseConnection {
    fn drop(&mut self) {
        // ...closing connection...
        self.close_connection();
        // ...printing a message...
        println!("Closing connection: {}", self.0)
    }
}

fn main() {
    let _db1 = DatabaseConnection("Server=A;Database=DB1");
    let _db2 = DatabaseConnection("Server=A;Database=DB2");
    // ...code for making use of the database connection...
} // "Dispose" of "db1" and "db2" called here; when their scope ends
```

在.NET中，尝试在对象上调用`Dispose`后使用对象通常会导致在运行时引发`ObjectDisposedException`。 在Rust中，编译器确保在编译时不会发生这种情况。

线程

Rust标准库支持线程、同步和并发。 此外，语言本身和标准库确实对这些概念提供了基本支持，许多附加功能由crate提供，本文将不予介绍。

以下列出了.NET中线程类型和方法到Rust的近似映射：

.NET	Rust
`Thread`	`std::thread::thread`
`Thread.Start`	`std::thread::spawn`
`Thread.Join`	`std::thread::JoinHandle`
`Thread.Sleep`	`std::thread::sleep`
`ThreadPool`	-
`Mutex`	`std::sync::Mutex`
`Semaphore`	-
`Monitor`	`std::sync::Mutex`
`ReaderWriterLock`	`std::sync::RwLock`
`AutoResetEvent`	`std::sync::Condvar`
`ManualResetEvent`	`std::sync::Condvar`
`Barrier`	`std::sync::Barrier`
`CountdownEvent`	`std::sync::Barrier`
`Interlocked`	`std::sync::atomic`
`Volatile`	`std::sync::atomic`
`ThreadLocal`	`std::thread_local`

在C#/.NET和Rust中，启动线程并等待其完成的方式相同。 下面是一个简单的C#程序，它创建一个线程（线程将一些文本打印到标准输出），然后等待它结束：

```
using System;
using System.Threading;

var thread = new Thread(() => Console.WriteLine("Hello from a thread!"));
```

```
thread.Start();
thread.Join(); // wait for thread to finish
```

Rust中的相同代码如下：

```
use std::thread;

fn main() {
    let thread = thread::spawn(|| println!("Hello from a thread!"));
    thread.join().unwrap(); // wait for thread to finish
}
```

在.NET中，创建和初始化线程对象以及启动线程是两个不同的操作， 而在Rust中，这两个操作通过`thread::spawn`同时发生。

在.NET中，可以将数据作为参数发送给线程：

```
#nullable enable

using System;
using System.Text;
using System.Threading;

var t = new Thread(obj =>
{
    var data = (StringBuilder)obj!;
    data.Append(" World!");
});

var data = new StringBuilder("Hello");
t.Start(data);
t.Join();

Console.WriteLine($"Phrase: {data}");
```

然而，更现代或更简洁的版本将使用闭包：

```
using System;
using System.Text;
using System.Threading;

var data = new StringBuilder("Hello");

var t = new Thread(obj => data.Append(" World!"));

t.Start();
t.Join();
```

```
Console.WriteLine($"Phrase: {data}");
```

在Rust中，没有 `thread::spawn` 的变体可以实现相同的功能。 相反，数据通过闭包传递给线程：

```
use std::thread;

fn main() {
    let data = String::from("Hello");
    let handle = thread::spawn(move || {
        let mut data = data;
        data.push_str(" World!");
        data
    });
    println!("Phrase: {}", handle.join().unwrap());
}
```

需要注意以下几点：

- 需要使用 `move` 关键字来将 `data` 的所有权移动或传递给线程的闭包。 完成此操作后，在 `main` 中继续使用 `main` 的 `data` 变量就不再合法。 如果需要，必须复制或克隆 `data` （取决于值支持的类型）。
- Rust线程可以返回值，就像C#中的任务一样，它将成为 `join` 方法的返回值。
- 也可以通过闭包将数据传递给C#线程，就像Rust示例一样，但C#版本不需要担心所有权，因为一旦没有人再引用数据，GC就会回收数据背后的内存。

同步

When data is shared between threads, one needs to synchronize read-write access to the data in order to avoid corruption. The C# offers the `lock` keyword as a synchronization primitive (which desugars to exception-safe use of `Monitor` from .NET):

```
using System;
using System.Threading;

var dataLock = new object();
var data = 0;
var threads = new List<Thread>();

for (var i = 0; i < 10; i++)
{
    var thread = new Thread(() =>
    {
        for (var j = 0; j < 1000; j++)
        {
            lock (dataLock)
                data++;
```

```
        }
    });
    threads.Add(thread);
    thread.Start();
}

foreach (var thread in threads)
    thread.Join();

Console.WriteLine(data);
```

In Rust, one must make explicit use of concurrency structures like `Mutex`:

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

fn main() {
    let data = Arc::new(Mutex::new(0)); // (1)

    let mut threads = vec![];
    for _ in 0..10 {
        let data = Arc::clone(&data); // (2)
        let thread = thread::spawn(move || { // (3)
            for _ in 0..1000 {
                let mut data = data.lock().unwrap();
                *data += 1; // (4)
            }
        });
        threads.push(thread);
    }

    for thread in threads {
        thread.join().unwrap();
    }

    println!("{}", data.lock().unwrap());
}
```

需要注意以下几点：

- 由于`Mutex`实例的所有权以及它所保护的数据将由多个线程共享，因此它被包装在`Arc`(1)中。 `Arc`提供原子引用计数，每次克隆时都会递增(2)，每次删除时都会递减。 当计数达到零时，互斥锁以及它所保护的数据将被删除。 这在内存管理中更详细地讨论。
- 每个线程的闭包实例都获得 *克隆的引用* (2)的所有权(3)。
- 类似指针的代码`*data += 1`(4)，即使看起来像，也不是某种不安全的指针访问。 它正在更新 *包装* 在互斥锁保护中的数据。

与C#版本不同，在C#版本中，可以通过注释掉`lock`语句使其线程不安全， 而Rust版本如果以任何方式（例如注释掉部分）更改，使其线程不安全，则会拒绝编译。 这表明，在C#和.NET 中，通过谨慎使用同步结构编写线程

安全代码是开发人员的责任，而在Rust中，可以依赖编译器。

编译器能够提供帮助，因为Rust中的数据结构由特殊特征标记（参见接口）：`Sync` 和 `Send`。 `Sync` 表示对类型实例的引用可以在线程之间安全地共享。 `Send` 表示跨线程边界对类型的实例是安全的。 有关更多信息，请参阅Rust书中的"无畏并发"章节。

生产者-消费者

生产者-消费者模式非常常见，用于在线程之间分配工作，其中数据从生产线程传递到消费线程，而无需共享或锁定。 .NET对此有非常丰富的支持，但在最基本的层面上，`System.Collections.Concurrent` 提供了 `BlockingCollection`， 如下一个C#示例中所示：

```
using System;
using System.Threading;
using System.Collections.Concurrent;

var messages = new BlockingCollection<string>();
var producer = new Thread(() =>
{
    for (var n = 1; i < 10; i++)
        messages.Add($"Message #{n}");
    messages.CompleteAdding();
});

producer.Start();

// main thread is the consumer here
foreach (var message in messages.GetConsumingEnumerable())
    Console.WriteLine(message);

producer.Join();
```

在Rust中，也可以使用 *channels* 来实现同样的效果。 标准库主要提供 `mpsc::channel`，这是一个支持多个生产者和单个消费者的通道。 上述C#示例在Rust中的粗略翻译如下：

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

fn main() {
    let (tx, rx) = mpsc::channel();

    let producer = thread::spawn(move || {
        for n in 1..10 {
            tx.send(format!("Message #{}", n)).unwrap();
        }
    });

    // main thread is the consumer here
```

```
    for received in rx {
        println!("{}", received);
    }

    producer.join().unwrap();
}
```

与Rust中的通道一样，.NET也在`System.Threading.Channels`命名空间中提供通道，但它主要设计用于使用`async`和`await`的任务和异步编程。 Rust空间中的异步友好通道的等效项是Tokio运行时提供。

测试

测试组织

.NET解决方案使用单独的项目来托管测试代码，无论使用的测试框架（xUnit、NUnit、MSTest 等）和编写的测试类型（单元或集成）如何。 因此，测试代码位于与被测试的应用程序或库代码不同的程序集中。 在Rust中，更常见的是 *单元测试* 位于单独的测试子模块中（传统上）名为`tests`，但该子模块与测试主题的应用程序或库模块代码位于同一 *源文件* 中。 这有两个好处：

- 代码/模块及其单元测试并存。
- 不需要像.NET中存在的`[InternalsVisibleTo]`这样的解决方法，因为测试可以通过虚拟子模块访问内部内容。

测试子模块使用`#[cfg(test)]`属性进行注释，其效果是，只有发出`cargo test`命令时，才会（有条件地）编译和运行整个模块。

在测试子模块中，测试函数使用`#[test]`属性进行注释。

集成测试通常位于名为`tests`的目录中，该目录与包含单元测试和源代码的`src`目录相邻。 `cargo test`将该目录中的每个文件编译为单独的包，并运行使用`#[test]`属性注释的所有方法。 由于可以理解集成测试在`tests`目录中，因此无需使用`#[cfg(test)]`属性标记其中的模块。

另请参阅：

- 测试组织

运行测试

简单来说，Rust 中`dotnet test`的对应物是`cargo test`。

`cargo test`的默认行为是并行运行所有测试，但可以将其配置为仅使用单个线程连续运行：

```
cargo test -- --test-threads=1
```

有关详细信息，请参阅"并行或连续运行测试"。

测试输出

对于非常复杂的集成或端到端测试，.NET 开发人员有时会记录测试期间发生的情况。 他们实际执行此操作的方式因每个测试框架而异。 例如，在NUnit中，这就像使用`Console.WriteLine`一样简单，但在 XUnit 中，人们使用`ITestOutputHelper`。 在Rust中，它类似于NUnit；也就是说，人们只需使用`println!`写入标准输出。 除非使用`--show-output`选项运行`cargo test`，否则默认情况下不会显示测试运行期间捕获的输出：

```
cargo test --show-output
```

有关详细信息，请参阅“显示函数输出”。

断言

.NET用户有多种断言方式，具体取决于所使用的框架。例如，xUnit.net 的断言可能如下所示：

```
[Fact]
public void Something_Is_The_Right_Length()
{
    var value = "something";
    Assert.Equal(9, value.Length);
}
```

Rust不需要单独的框架或板条箱。标准库附带内置的 *宏*，足以满足测试中的大多数断言：

- `assert!`
- `assert_eq!`
- `assert_ne!`

下面是`assert_eq`实际运行的一个示例：

```
#[test]
fn something_is_the_right_length() {
    let value = "something";
    assert_eq!(9, value.len());
}
```

标准库没有提供任何数据驱动测试方向的内容，例如xUnit.net中的`[Theory]`。

模拟

When writing tests for a .NET application or library, there exist several frameworks, like Moq and NSubstitute, to mock out the dependencies of types. There are similar crates for Rust too, like `mockall`, that can help with mocking. However, it is also possible to use conditional compilation by making use of the `cfg` attribute as a

simple means to mocking without needing to rely on external crates or frameworks. The `cfg` attribute conditionally includes the code it annotates based on a configuration symbol, such as `test` for testing. This is not very different to using `DEBUG` to conditionally compile code specifically for debug builds. One downside of this approach is that you can only have one implementation for all tests of the module.

When specified, the `#[cfg(test)]` attribute tells Rust to compile and run the code only when executing the `cargo test` command, which behind-the-scenes executes the compiler with `rustc --test`. The opposite is true for the `#[cfg(not(test))]` attribute; it includes the annotated only when testing with `cargo test`.

The example below shows mocking of a stand-alone function `var_os` from the standard that reads and returns the value of an environment variable. It conditionally imports a mocked version of the `var_os` function used by `get_env`. When built with `cargo build` or run with `cargo run`, the compiled binary will make use of `std::env::var_os`, but `cargo test` will instead import `tests::var_os_mock` as `var_os`, thus causing `get_env` to use the mocked version during testing:

在为.NET应用程序或库编写测试时，存在多个框架（如 Moq 和 NSubstitute）来模拟类型的依赖关系。 Rust也有类似的包，如 `mockall`，可以帮助模拟。 但是，也可以通过使用 `cfg` 属性 作为简单的模拟方法，而无需依赖外部包或框架，从而使用条件编译。 `cfg` 属性根据配置符号（例如用于测试的 `test`）有条件地包含它注释的代码。 这与使用 `DEBUG` 有条件地编译专门用于调试版本的代码没有太大区别。 这种方法的一个缺点是，您只能为模块的所有测试提供一个实现。

指定后，`#[cfg(test)]` 属性会告诉Rust仅在执行 `cargo test` 命令时编译并运行代码，该命令在后台使用 `rustc --test` 执行编译器。 `#[cfg(not(test))]` 属性则相反；它仅在使用 `cargo test` 进行测试时才包含注释。

以下示例显示了标准中独立函数 `var_os` 的模拟，该函数读取并返回环境变量的值。 它有条件地导入 `get_env` 使用的 `var_os` 函数的模拟版本。当使用 `cargo build` 构建或使用 `cargo run` 运行时，编译后的二进制文件将使用 `std::env::var_os`， 但 `cargo test` 将改为导入 `tests::var_os_mock` 作为 `var_os`，从而导致 `get_env` 在测试期间使用模拟版本：

```
// Copyright (c) Microsoft Corporation. All rights reserved.
// Licensed under the MIT license.

/// Utility function to read an environmentvariable and return its value If
/// defined. It fails/panics if the valus is not valid Unicode.
pub fn get_env(key: &str) -> Option<String> {
    #[cfg(not(test))]                // for regular builds...
    use std::env::var_os;            // ...import from the standard library
    #[cfg(test)]                     // for test builds...
    use tests::var_os_mock as var_os; // ...import mock from test sub-module

    let val = var_os(key);
    val.map(|s| s.to_str()     // get string slice
                 .unwrap()     // panic if not valid Unicode
                 .to_owned())  // convert to "String"
}

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

```
    pub(crate) fn var_os_mock(key: &str) -> Option<OsString> {
        match key {
            "FOO" => Some("BAR".into()),
            _ => None
        }
    }

    #[test]
    fn get_env_when_var_undefined_returns_none() {
        assert_eq!(None, get_env("???"));
    }

    #[test]
    fn get_env_when_var_defined_returns_some_value() {
        assert_eq!(Some("BAR".to_owned()), get_env("FOO"));
    }
}
```

代码覆盖率

在分析测试代码覆盖率时，.NET有复杂的工具。 在Visual Studio中，工具是内置和集成的。 在Visual Studio Code中，存在插件。.NET 开发人员可能也熟悉coverlet。

Rust提供内置代码覆盖率实现来收集测试代码覆盖率。

Rust也有插件可帮助进行代码覆盖率分析。 它不是无缝集成的，但通过一些手动步骤，开发人员可以以可视化的方式分析他们的代码。

Visual Studio Code的Coverage Gutters插件和Tarpaulin的组合允许在Visual Studio Code中可视化分析代码覆盖率。 Coverage Gutters需要LCOV文件。除了Tarpaulin 之外，其他工具也可用于生成该文件。

设置后，运行以下命令：

```
cargo tarpaulin --ignore-tests --out Lcov
```

这将生成一个 LCOV 代码覆盖率文件。 一旦启用 `Coverage Gutters: Watch`，它将被Coverage Gutters插件拾取，该插件将在源代码编辑器中显示有关行覆盖率的内联视觉指示器。

> 注意：LCOV 文件的位置至关重要。 如果存在包含多个包的工作区（请参阅项目结构），并且使用 `--workspace` 在根目录中生成了 LCOV 文件，则该文件就是正在使用的文件-即使包的根目录中直接存在一个文件。 与在根目录中生成 LCOV 文件相比，将文件隔离到正在测试的特定包会更快。

基准测试

在Rust中运行基准测试是通过 `cargo bench` 完成的，这是 `cargo` 的一个特定命令，它执行所有用 `#[bench]` 属性注释的方法。 此属性目前不稳定，仅适用于夜间频道。

.NET 用户可以使用`BenchmarkDotNet`库来对方法进行基准测试并跟踪其性能。 `BenchmarkDotNet`的等效项是名为`Criterion`的包。

根据其文档，`Criterion`会收集和存储每次运行的统计信息，并可以自动检测性能回归以及测量优化。

使用`Criterion`可以使用`#[bench]`属性，而无需转到夜间频道。

与`BenchmarkDotNet`一样，也可以将基准测试结果与用于持续基准测试的GitHub Action 集成。 事实上，`Criterion`支持多种输出格式，其中还有`bencher`格式，模仿夜间的`libtest`基准测试并与上述操作兼容。

日志记录和跟踪

.NET支持多种日志记录 API。在大多数情况下，`ILogger`是一个很好的默认选择，因为它可以与各种内置和第三方日志记录提供程序配合使用。 在C#中，结构化日志记录的一个最小示例可能如下所示：

```
using Microsoft.Extensions.Logging;

using var loggerFactory = LoggerFactory.Create(builder => builder.AddConsole());
var logger = loggerFactory.CreateLogger<Program>();
logger.LogInformation("Hello {Day}.", "Thursday"); // Hello Thursday.
```

在Rust中，log提供了一个轻量级的日志外观。它的功能比`ILogger`少，例如，它尚未提供（稳定的）结构化日志记录或日志范围。

对于具有与.NET功能相同的功能，Tokio提供了`tracing`。 `tracing`是一个用于检测Rust应用程序以收集结构化、基于事件的诊断信息的框架。 `tracing_subscriber`可用于实现和编写`tracing`订阅者。 上面带有`tracing`和`tracing_subscriber`的相同结构化日志记录示例如下所示：

```
fn main() {
    // install global default ("console") collector.
    tracing_subscriber::fmt().init();
    tracing::info!("Hello {Day}.", Day = "Thursday"); // Hello Thursday.
}
```

OpenTelemetry提供了一系列工具、API和SDK，用于根据OpenTelemetry规范检测、生成、收集和导出遥测数据。在撰写本文时，OpenTelemetry Logging API尚不稳定，Rust实现尚不支持日志记录，但支持跟踪API。

条件编译

.NET和Rust都提供了根据外部条件编译特定代码的可能性。

在.NET中，可以使用一些预处理器指令来控制条件编译

```
#if debug
    Console.WriteLine("Debug");
#else
    Console.WriteLine("Not debug");
#endif
```

除了预定义符号外，还可以使用编译器选项 *定义常量* 来定义可与`#if`、`#else`、`#elif`和`#endif`一起使用的符号，以有条件地编译源文件。

在Rust中，可以使用`cfg属性`、`cfg_attr属性`或`cfg宏`来控制条件编译。

根据.NET，除了预定义符号外，还可以使用编译器标志`--cfg`任意设置配置选项

`cfg属性`需要并评估`配置谓词`

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

struct MyStruct;

// This implementation of Display is only included when the OS is unix but foo is not equal to bar
// You can compile an executable for this version, on linux, with 'rustc main.rs --cfg
foo=\"baz\"'
#[cfg(all(unix, not(foo = "bar")))]
impl Display for MyStruct {
    fn fmt(&self, f: &mut Formatter<'_>) -> std::fmt::Result {
        f.write_str("Running without foo=bar configuration")
    }
}

// This function is only included when both unix and foo=bar are defined
// You can compile an executable for this version, on linux, with 'rustc main.rs --cfg
foo=\"bar\"'
#[cfg(all(unix, foo = "bar"))]
impl Display for MyStruct {
    fn fmt(&self, f: &mut Formatter<'_>) -> std::fmt::Result {
        f.write_str("Running with foo=bar configuration")
    }
}

// This function is panicking when not compiled for unix
// You can compile an executable for this version, on windows, with 'rustc main.rs'
#[cfg(not(unix))]
impl Display for MyStruct {
    fn fmt(&self, _f: &mut Formatter<'_>) -> std::fmt::Result {
        panic!()
    }
}

fn main() {
```

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

`cfg_attr属性`根据配置谓词有条件地包含属性。

```
#[cfg_attr(feature = "serialization_support", derive(Serialize, Deserialize))]
pub struct MaybeSerializableStruct;

// When the `serialization_support` feature flag is enabled, the above will expand to:
// #[derive(Serialize, Deserialize)]
// pub struct MaybeSerializableStruct;
```

内置的 `cfg宏` 接受单个配置谓词，并在谓词为真时计算为真文字，为假时计算为假文字。

```
if cfg!(unix) {
  println!("I'm running on a unix machine!");
}
```

另请参阅：

- 条件编译

特征

当需要提供可选依赖项时，条件编译也很有用。 使用Cargo“功能”，包在Cargo.toml的`[features]`表中定义一组命名的功能，每个功能都可以启用或禁用。 可以使用`--features`等标志在命令行上启用正在构建的包的功能。可以在Cargo.toml中的依赖项声明中启用依赖项的功能。

另请参阅：

- 功能

环境和配置

访问环境变量

.NET通过`System.Environment.GetEnvironmentVariable`方法提供对环境变量的访问。 此方法在运行时检索环境变量的值。

```
using System;

const string name = "EXAMPLE_VARIABLE";
```

```
var value = Environment.GetEnvironmentVariable(name);
if (string.IsNullOrEmpty(value))
    Console.WriteLine($"Variable '{name}' not set.");
else
    Console.WriteLine($"Variable '{name}' set to '{value}'.");
```

Rust通过`std::env`模块中的`var`和`var_os`函数提供在运行时访问环境变量的相同功能。

`var`函数返回`Result<String, VarError>`，如果设置则返回变量，如果未设置变量或变量不是有效的Unicode则返回错误。

`var_os`具有不同的签名，返回`Option<OsString>`，如果设置了变量则返回某个值，如果未设置变量则返回None。 `OsString`不需要是有效的Unicode。

```
use std::env;


fn main() {
    let key = "ExampleVariable";
    match env::var(key) {
        Ok(val) => println!("{key}: {val:?}"),
        Err(e) => println!("couldn't interpret {key}: {e}"),
    }
}
```

```
use std::env;

fn main() {
    let key = "ExampleVariable";
    match env::var_os(key) {
        Some(val) => println!("{key}: {val:?}"),
        None => println!("{key} not defined in the enviroment"),
    }
}
```

Rust还提供了在编译时访问环境变量的功能。`std::env`中的`env!`宏在编译时扩展变量的值，返回`&'static str`。如果未设置变量，则会发出错误。

```
use std::env;

fn main() {
    let example = env!("ExampleVariable");
    println!("{example}");
}
```

在.NET中，可以通过源生成器以不太直接的方式实现对环境变量的编译时访问。

配置

.NET中的配置可通过配置提供程序实现。该框架通过`Microsoft.Extensions.Configuration`命名空间和NuGet包提供多种提供程序实现。

配置提供程序使用不同的来源从键值对中读取配置数据，并通过`IConfiguration`类型提供配置的统一视图。

```
using Microsoft.Extensions.Configuration;

class Example {
    static void Main()
    {
        IConfiguration configuration = new ConfigurationBuilder()
            .AddEnvironmentVariables()
            .Build();

        var example = configuration.GetValue<string>("ExampleVar");

        Console.WriteLine(example);
    }
}
```

其他提供程序示例可在官方文档.NET中的配置提供程序中找到。

通过使用第三方包（例如figment或config），Rust中也提供类似的配置体验。

请参阅以下使用config包的示例：

```
use config::{Config, Environment};

fn main() {
    let builder = Config::builder().add_source(Environment::default());

    match builder.build() {
        Ok(config) => {
            match config.get_string("examplevar") {
                Ok(v) => println!("{v}"),
                Err(e) => println!("{e}")
            }
        },
        Err(_) => {
            // something went wrong
        }
    }
}
```

LINQ

本节讨论上下文中的LINQ，目的是查询或转换序列（`IEnumerable` / `IEnumerable<T>`）以及通常的集合（如lists, sets and dictionaries）。

`IEnumerable<T>`

Rust中`IEnumerable<T>`的等价物是`IntoIterator`。 正如`IEnumerable<T>.GetEnumerator()`的实现在.NET中返回一个`IEnumerator<T>`， `IntoIterator::into_iter`的实现返回一个`Iterator`。 然而，当需要迭代容器中的项目时，通过上述类型来宣传迭代支持，这两种语言都以循环构造的形式为可迭代对象提供语法糖。 在C#中，有`foreach`：

```
using System;
using System.Text;

var values = new[] { 1, 2, 3, 4, 5 };
var output = new StringBuilder();

foreach (var value in values)
{
    if (output.Length > 0)
        output.Append(", ");
    output.Append(value);
}

Console.Write(output); // Prints: 1, 2, 3, 4, 5
```

在Rust中，等效的是`for`：

```
use std::fmt::Write;

fn main() {
    let values = [1, 2, 3, 4, 5];
    let mut output = String::new();

    for value in values {
        if output.len() > 0 {
            output.push_str(", ");
        }
        // ! discard/ignore any write error
        _ = write!(output, "{value}");
    }

    println!("{output}");  // Prints: 1, 2, 3, 4, 5
}
```

可迭代对象的`for`循环本质上简化为以下内容：

```
use std::fmt::Write;

fn main() {
    let values = [1, 2, 3, 4, 5];
    let mut output = String::new();

    let mut iter = values.into_iter();      // get iterator
    while let Some(value) = iter.next() {   // loop as long as there are more items
        if output.len() > 0 {
            output.push_str(", ");
        }
        _ = write!(output, "{value}");
    }

    println!("{output}");
}
```

Rust的所有权和数据竞争条件规则适用于所有实例和数据，迭代也不例外。 因此，虽然循环遍历数组可能看起来很简单，并且与C#非常相似，但当需要多次迭代同一个集合/可迭代对象时，必须注意所有权。 以下示例迭代整数列表两次，一次打印它们的和，另一次确定并打印最大整数：

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

    // sum all values

    let mut sum = 0;
    for value in values {
        sum += value;
    }
    println!("sum = {sum}");

    // determine maximum value

    let mut max = None;
    for value in values {
        if let Some(some_max) = max { // if max is defined
            if value > some_max {     // and value is greater
                max = Some(value)     // then note that new max
            }
        } else {                      // max is undefined when iteration starts
            max = Some(value)         // so set it to the first value
        }
    }
    println!("max = {max:?}");
}
```

但是，上面的代码由于一个细微的差别而被编译器拒绝：`values` 已从数组更改为 `Vec<int>`，即 *vector*，这是Rust的可增长数组类型（如.NET中的 `List<T>`）。 `values` 的第一次迭代最终在整数相加时 *消耗* 每个值。换句话说，vector中 *每个项目* 的所有权传递给循环的迭代变量：`value`。 由于 `value` 在循环的每次迭代结束时超出范

围，因此它拥有的实例将被删除。如果 `values` 是堆分配数据的vector，则当循环移动到下一个项目时，支持每个项目的堆内存将被释放。 要解决该问题，必须在 `for` 循环中通过 `&values` 请求对 *共享* 引用进行迭代。因此，`value` 最终成为对某个项目的共享引用，而不是取得其所有权。

下面是编译的上一个示例的更新版本。修复方法是简单地在每个 `for` 循环中将 `values` 替换为 `&values`。

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

    // sum all values

    let mut sum = 0;
    for value in &values {
        sum += value;
    }
    println!("sum = {sum}");

    // determine maximum value

    let mut max = None;
    for value in &values {
        if let Some(some_max) = max { // if max is defined
            if value > some_max {     // and value is greater
                max = Some(value)     // then note that new max
            }
        } else {                      // max is undefined when iteration starts
            max = Some(value)         // so set it to the first value
        }
    }
    println!("max = {max:?}");
}
```

The ownership and dropping can be seen in action even with `values` being an array instead of a vector. Consider just the summing loop from the above example over an array of a structure that wraps an integer:

```
struct Int(i32);

impl Drop for Int {
    fn drop(&mut self) {
        println!("{} dropped", self.0)
    }
}

fn main() {
    let values = [Int(1), Int(2), Int(3), Int(4), Int(5)];
    let mut sum = 0;

    for value in values {
        sum += value.0;
    }
```

```
    println!("sum = {sum}");
}
```

`Int`实现了`Drop`，这样当实例被丢弃时就会打印一条消息。运行上述代码将打印：

```
value = Int(1)
Int(1) dropped
value = Int(2)
Int(2) dropped
value = Int(3)
Int(3) dropped
value = Int(4)
Int(4) dropped
value = Int(5)
Int(5) dropped
sum = 15
```

很明显，每个值都是在循环运行时获取和丢弃的。循环完成后，将打印总和。 如果将`for`循环中的`values`改为`&values`，如下所示：

```
for value in &values {
    // ...
}
```

那么程序的输出将发生根本性的变化：

```
value = Int(1)
value = Int(2)
value = Int(3)
value = Int(4)
value = Int(5)
sum = 15
Int(1) dropped
Int(2) dropped
Int(3) dropped
Int(4) dropped
Int(5) dropped
```

这次，在循环过程中会获取值但不会丢弃，因为每个项都不会被迭代循环的变量所拥有。 循环完成后会打印总和。最后，当仍然拥有所有`Int`实例的`values`数组在`main`末尾超出范围时，它的丢弃会依次丢弃所有`Int`实例。

这些示例表明，虽然迭代集合类型似乎在Rust和C#之间有很多相似之处，从循环构造到迭代抽象，但在所有权方面仍然存在细微的差异，这可能导致编译器在某些情况下拒绝代码。

另请参阅：

- Iterator
- Iterating by reference

运算符

LINQ中的运算符以C#扩展方法的形式实现，这些扩展方法可以链接在一起形成一组操作，最常见的操作是对某种数据源进行查询。 C#还提供了受SQL启发的 *查询语法*，其中包含 `from`、`where`、`select`、`join` 等子句，可以作为方法链接的替代或补充。 许多命令式循环可以在LINQ中重写为更具表现力和可组合性的查询。

Rust不提供任何类似C#的查询语法。它具有在Rust术语中称为[适配器]的可迭代类型方法，因此可直接与C#中的方法链接相媲美。 但是，虽然在C#中将命令式循环重写为 LINQ 代码通常有利于提高表现力、稳健性和可组合性，但性能会有所降低。 计算绑定的命令式循环通常运行得更快，因为它们可以通过JIT编译器进行优化，并且产生的虚拟调度或间接函数调用更少。 Rust中令人惊讶的部分是，选择在抽象（如迭代器）上使用方法链与手动编写命令式循环之间没有性能权衡。 因此，在代码中看到前者更为常见。

下表列出了 Rust 中最常见的LINQ方法及其近似对应方法。

.NET	Rust	Note
`Aggregate`	`reduce`	See note 1.
`Aggregate`	`fold`	See note 1.
`All`	`all`	
`Any`	`any`	
`Concat`	`chain`	
`Count`	`count`	
`ElementAt`	`nth`	
`GroupBy`	-	
`Last`	`last`	
`Max`	`max`	
`Max`	`max_by`	
`MaxBy`	`max_by_key`	
`Min`	`min`	
`Min`	`min_by`	
`MinBy`	`min_by_key`	
`Reverse`	`rev`	
`Select`	`map`	
`Select`	`enumerate`	

`SelectMany`	`flat_map`	
`SelectMany`	`flatten`	
`SequenceEqual`	`eq`	
`Single`	`find`	
`SingleOrDefault`	`try_find`	
`Skip`	`skip`	
`SkipWhile`	`skip_while`	
`Sum`	`sum`	
`Take`	`take`	
`TakeWhile`	`take_while`	
`ToArray`	`collect`	See note 2.
`ToDictionary`	`collect`	See note 2.
`ToList`	`collect`	See note 2.
`Where`	`filter`	
`Zip`	`zip`	

1. 不接受种子值的 `Aggregate` 重载相当于 `reduce`，而接受种子值的 `Aggregate` 重载相当于 `fold`。

2. Rust中的 `collect` 通常适用于任何可收集类型，该类型被定义为一种可以从迭代器初始化自身的类型（参见 `FromIterator`）。 `collect` 需要一个目标类型，编译器有时很难推断，因此 *turbofish*（`::<>`）经常与其一起使用，如 `collect::<Vec<_>>()`。 这就是为什么 `collect` 出现在许多 LINQ 扩展方法旁边，这些方法将可枚举/可迭代源转换为某些集合类型实例。

以下示例显示了C#中的转换序列与Rust中的转换序列有多么相似。首先在 C# 中：

```
var result =
    Enumerable.Range(0, 10)
              .Where(x => x % 2 == 0)
              .SelectMany(x => Enumerable.Range(0, x))
              .Aggregate(0, (acc, x) => acc + x);

Console.WriteLine(result); // 50
```

在Rust中：

```
let result = (0..10)
    .filter(|x| x % 2 == 0)
    .flat_map(|x| (0..x))
```

```
    .fold(0, |acc, x| acc + x);

println!("{result}"); // 50
```

延迟执行（惰性）

LINQ中的许多运算符被设计为惰性的，因此它们只在绝对需要时才工作。 这样就可以组合或链接多个操作/方法而不会产生任何副作用。 例如，LINQ运算符可以返回已初始化的`IEnumerable<T>`，但在迭代之前不会生成、计算或实现 `T` 的任何项。 该运算符被认为具有 *延迟执行* 语义。如果每个`T`在迭代到达时计算（而不是在迭代开始时计算），则该运算符被称为 *流式传输* 结果。

Rust迭代器具有相同的*惰性*和流式传输概念。

在这两种情况下，这都允许表示 *无限序列*，其中底层序列是无限的，但开发人员决定如何终止序列。 以下示例在 C# 中显示了这一点：

```
foreach (var x in InfiniteRange().Take(5))
    Console.Write($"{x} "); // Prints "0 1 2 3 4"

IEnumerable<int> InfiniteRange()
{
    for (var i = 0; ; ++i)
        yield return i;
}
```

Rust通过无限范围支持相同的概念：

```
// Generators and yield in Rust are unstable at the moment, so
// instead, this sample uses Range:
// https://doc.rust-lang.org/std/ops/struct.Range.html

for value in (0..).take(5) {
    print!("{value} "); // Prints "0 1 2 3 4"
}
```

迭代器方法（`yield`）

C#具有`yield`关键字，使开发人员能够快速编写 *迭代器方法*。 迭代器方法的返回类型可以是`IEnumerable<T>`或`IEnumerator<T>`。 然后，编译器将方法的主体转换为返回类型的具体实现，而开发人员不必每次都编写一个完整的类。 在撰写本文时，*Coroutines*（Rust中的名称）仍被视为不稳定的功能。

元编程

元编程可以看作是一种编写代码来编写/生成其他代码的方式。

Roslyn为C#中的元编程提供了一项功能，该功能自.NET 5开始提供，称为`Source Generators`。 源生成器可以在构建时创建新的C#源文件，并将其添加到用户的编译中。 在引入`Source Generators`之前，Visual Studio一直通过`T4 Text Templates`提供代码生成工具。 T4工作原理的一个示例是以下[模板]或其[具体化]。

Rust还提供了元编程功能：[宏]。有`声明性宏`和`过程宏`。

声明性宏允许您编写控制结构，该控制结构采用表达式，将表达式的结果值与模式进行比较，然后运行与匹配模式相关的代码。

以下示例是`println!`宏的定义，可以调用该宏来打印一些文本`println!("Some text")`

```
macro_rules! println {
    () => {
        $crate::print!("\n")
    };
    ($($arg:tt)*) => {{
        $crate::io::_print($crate::format_args_nl!($($arg)*));
    }};
}
```

要了解有关编写声明性宏的更多信息，请参阅Rust参考章节宏示例或Rust宏小册子。

[过程宏]与声明性宏不同。它们接受一些代码作为输入，对该代码进行操作，并生成一些代码作为输出。

C#中用于元编程的另一种技术是反射。Rust不支持反射。

类似函数的宏

函数式宏的形式如下：`function!(...)`

以下代码片段定义了一个名为`print_something`的函数式宏，它生成一个`print_it`方法来打印“Something”字符串。

在lib.rs中：

```
extern crate proc_macro;
use proc_macro::TokenStream;

#[proc_macro]
pub fn print_something(_item: TokenStream) -> TokenStream {
    "fn print_it() { println!(\"Something\") }".parse().unwrap()
}
```

在main.rs中：

```
use replace_crate_name_here::print_something;
print_something!();
```

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

派生宏

派生宏可以根据结构、枚举或联合的标记流创建新项。 派生宏的一个示例是`#[derive(Clone)]`，它生成使输入结构/枚举/联合实现`Clone`特征所需的代码。

为了了解如何定义自定义派生宏，可以阅读派生宏的rust参考

属性宏

属性宏定义新属性可以附加到rust项目。 在使用异步代码时，如果使用Tokio，第一步将是用属性宏装饰新的异步主程序，如以下示例所示：

```
#[tokio::main]
async fn main() {
    println!("Hello world");
}
```

为了理解如何定义自定义属性宏，可以阅读rust参考中的属性宏

异步编程

.NET和Rust都支持异步编程模型，它们的用法看起来彼此相似。以下示例从非常高的层次展示了C#中的异步代码：

```
async Task<string> PrintDelayed(string message, CancellationToken cancellationToken)
{
    await Task.Delay(TimeSpan.FromSeconds(1), cancellationToken);
    return $"Message: {message}";
}
```

Rust代码的结构类似。以下示例依赖async-std来实现`sleep`：

```
use std::time::Duration;
use async_std::task::sleep;

async fn format_delayed(message: &str) -> String {
    sleep(Duration::from_secs(1)).await;
```

```
    format!("Message: {}", message)
}
```

1. Rust `async` 关键字将代码块转换为状态机，该状态机实现了名为 `Future` 的特征， 类似于C#编译器将 `async` 代码转换为状态机的方式。在这两种语言中，这都允许按顺序编写异步代码。

2. 请注意，对于Rust和C#，异步方法/函数都以async关键字为前缀，但返回类型不同。 C#中的异步方法指示完整和实际的返回类型，因为它可能会有所不同。 例如，通常会看到一些方法返回 `Task<T>` ，而其他方法返回 `ValueTask<T>` 。 在Rust中，指定 *内部类型* `String` 就足够了，因为它 *总是一些future*；也就是说，实现 `Future` 特征的类型。

3. C#和Rust中的 `await` 关键字位置不同。在C#中，通过在表达式前添加 `await` 来等待 `Task` 。 在Rust中，在表达式后添加 `.await` 关键字可以实现 *方法链*，即使 `await` 不是方法。

另请参阅：

- Rust中的异步编程

执行任务

从以下示例中可以看出，尽管没有等待 `PrintDelayed` 方法，但它仍会执行：

```
var cancellationToken = CancellationToken.None;
PrintDelayed("message", cancellationToken); // Prints "message" after a second.
await Task.Delay(TimeSpan.FromSeconds(2), cancellationToken);

async Task PrintDelayed(string message, CancellationToken cancellationToken)
{
    await Task.Delay(TimeSpan.FromSeconds(1), cancellationToken);
    Console.WriteLine(message);
}
```

在Rust中，相同的函数调用不会打印任何内容。

```
use async_std::task::sleep;
use std::time::Duration;

#[tokio::main] // used to support an asynchronous main method
async fn main() {
    print_delayed("message"); // Prints nothing.
    sleep(Duration::from_secs(2)).await;
}

async fn print_delayed(message: &str) {
    sleep(Duration::from_secs(1)).await;
    println!("{}", message);
}
```

这是因为Future是惰性的：它们在运行之前什么都不做。 运行`Future`的最常见方式是`.await`。当在`Future`上调用`.await`时，它将尝试运行它直至完成。 如果`Future`被阻止，它将放弃对当前线程的控制。 当可以取得更多进展时，执行器将拾取`Future`并恢复运行，从而允许`.await`解析（参见`async/.await`）。

虽然等待函数可以从其他`async`函数中工作，但`main`不允许是`async`。 这是因为Rust本身不提供用于执行异步代码的运行时。因此，有用于执行异步代码的库，称为[异步运行时]。 Tokio就是这样一个异步运行时，它被频繁使用。 上面例子中的`tokio::main`将`async main`函数标记为运行时要执行的入口点，该入口点在使用宏时自动设置。

Task cancellation

前面的C#示例包括将`CancellationToken`传递给异步方法，这在.NET 中被认为是最佳实践。 `CancellationToken`可用于中止异步操作。

由于futures在Rust中是惰性的（它们只有在轮询时才会取得进展），因此在Rust中取消的工作方式不同。 当删除`Future`时，`Future`将不会再取得进展。它还将删除所有实例化值，直到由于某些未完成的异步操作而暂停future为止。 这就是为什么Rust中的大多数异步函数不接受参数来发出取消信号的原因，也是为什么删除future有时被称为 *取消* 的原因。

`tokio_util::sync::CancellationToken`提供与.NET`CancellationToken`等效的功能， 用于在无法在`Future`上实现`Drop`特征的情况下，对取消发出信号并做出反应。

执行多个任务

在.NET中，`Task.WhenAny`和`Task.WhenAll`经常用于处理多个任务的执行。

`Task.WhenAny`在任何任务完成后立即完成。 例如，Tokio提供了`tokio::select!`宏作为`Task.WhenAny`的替代方案，这意味着等待多个并发分支。

```
var cancellationToken = CancellationToken.None;

var result =
    await Task.WhenAny(Delay(TimeSpan.FromSeconds(2), cancellationToken),
                       Delay(TimeSpan.FromSeconds(1), cancellationToken));

Console.WriteLine(result.Result); // Waited 1 second(s).

async Task<string> Delay(TimeSpan delay, CancellationToken cancellationToken)
{
    await Task.Delay(delay, cancellationToken);
    return $"Waited {delay.TotalSeconds} second(s).";
}
```

对于Rust来说也是同样的例子：

```
use std::time::Duration;
use tokio::{select, time::sleep};
```

```rust
#[tokio::main]
async fn main() {
    let result = select! {
        result = delay(Duration::from_secs(2)) => result,
        result = delay(Duration::from_secs(1)) => result,
    };

    println!("{}", result); // Waited 1 second(s).
}

async fn delay(delay: Duration) -> String {
    sleep(delay).await;
    format!("Waited {} second(s).", delay.as_secs())
}
```

同样，这两个示例在语义上存在重大差异。 最重要的是，`tokio::select!` 将取消所有剩余分支，而 `Task.WhenAny` 则让用户自行决定是否取消任何正在进行的任务。

类似地，`Task.WhenAll` 可以被 `tokio::join!` 替换。

多个消费者

在.NET中，`Task` 可以跨多个消费者使用。 所有消费者都可以等待任务，并在任务完成或失败时收到通知。 在Rust中，`Future` 无法克隆或复制，并且 `await` 会转移所有权。 `futures::FutureExt::shared` 扩展为 `Future` 创建可克隆的句柄，然后可以将其分发给多个消费者。

```rust
use futures::FutureExt;
use std::time::Duration;
use tokio::{select, time::sleep, signal};
use tokio_util::sync::CancellationToken;

#[tokio::main]
async fn main() {
    let token = CancellationToken::new();
    let child_token = token.child_token();

    let bg_operation = background_operation(child_token);

    let bg_operation_done = bg_operation.shared();
    let bg_operation_final = bg_operation_done.clone();

    select! {
        _ = bg_operation_done => {},
        _ = signal::ctrl_c() => {
            token.cancel();
        },
    }

    bg_operation_final.await;
}
```

```rust
async fn background_operation(cancellation_token: CancellationToken) {
    select! {
        _ = sleep(Duration::from_secs(2)) => println!("Background operation completed."),
        _ = cancellation_token.cancelled() => println!("Background operation cancelled."),
    }
}
```

异步迭代

虽然.NET中有`IAsyncEnumerable<T>`和`IAsyncEnumerator<T>`， 但Rust在标准库中还没有异步迭代的API。为了支持异步迭代， `futures`中的`Stream`特性提供了一组类似的功能。

在C#中，编写异步迭代器的语法与编写同步迭代器的语法类似：

```csharp
await foreach (int item in RangeAsync(10, 3).WithCancellation(CancellationToken.None))
    Console.Write(item + " "); // Prints "10 11 12".

async IAsyncEnumerable<int> RangeAsync(int start, int count)
{
    for (int i = 0; i < count; i++)
    {
        await Task.Delay(TimeSpan.FromSeconds(i));
        yield return start + i;
    }
}
```

在Rust中，有几种类型实现了`Stream`特性，因此可用于创建流，例如`futures::channel::mpsc`。 对于更接近C#的语法， `async-stream`提供了一组宏，可用于简洁地生成流。

```rust
use async_stream::stream;
use futures_core::stream::Stream;
use futures_util::{pin_mut, stream::StreamExt};
use std::{
    io::{stdout, Write},
    time::Duration,
};
use tokio::time::sleep;

#[tokio::main]
async fn main() {
    let stream = range(10, 3);
    pin_mut!(stream); // needed for iteration
    while let Some(result) = stream.next().await {
        print!("{} ", result); // Prints "10 11 12".
        stdout().flush().unwrap();
    }
}

fn range(start: i32, count: i32) -> impl Stream<Item = i32> {
```

```
    stream! {
        for i in 0..count {
            sleep(Duration::from_secs(i as _)).await;
            yield start + i;
        }
    }
}
```

项目结构

虽然在.NET中有一些关于构建项目的惯例，但与Rust项目结构惯例相比，它们不那么严格。 当使用Visual Studio 2022（一个类库和一个xUnit测试项目）创建双项目解决方案时，它将创建以下结构：

```
.
|   SampleClassLibrary.sln
+---SampleClassLibrary
|       Class1.cs
|       SampleClassLibrary.csproj
+---SampleTestProject
        SampleTestProject.csproj
        UnitTest1.cs
        Usings.cs
```

- 每个项目都位于单独的目录中，并有自己的`.csproj`文件。
- 存储库的根目录下有一个`.sln`文件。

Cargo使用以下约定来定义包布局，以便轻松深入了解新的Cargo包：

```
.
+-- Cargo.lock
+-- Cargo.toml
+-- src/
|   +-- lib.rs
|   +-- main.rs
+-- benches/
|   +-- some-bench.rs
+-- examples/
|   +-- some-example.rs
+-- tests/
    +-- some-integration-test.rs
```

- `Cargo.toml`和`Cargo.lock`存储在包的根目录中。
- `src/lib.rs`是默认库文件，`src/main.rs`是默认可执行文件（参见目标自动发现）。
- 基准测试放在`benches`目录中，集成测试放在`tests`目录中（参见测试、基准测试）。
- 示例放在`examples`目录中。

- 没有单独的单元测试包，单元测试与代码位于同一文件中（参见测试）。

管理大型项目

对于Rust中的大型项目，Cargo提供工作区来组织项目。 工作区可以帮助管理同时开发的多个相关包。有些项目使用*虚拟清单*，尤其是在没有主包的情况下。

管理依赖项版本

在.NET中管理较大的项目时，使用诸如中央包管理之类的策略来集中管理依赖项的版本可能是合适的。 Cargo引入了工作区继承来集中管理依赖项。

编译和构建

.NET CLI

Rust中.NET CLI（`dotnet`）的对应工具是Cargo（`cargo`）。 这两种工具都是入口点包装器，可简化其他低级工具的使用。 例如，尽管您可以直接调用C#编译器（`csc`）或通过`dotnet msbuild`调用 MSBuild，但开发人员倾向于使用`dotnet build`来构建他们的解决方案。 同样，在Rust中，虽然您可以直接使用Rust编译器（`rustc`），但使用`cargo build`通常要简单得多。

构建

使用`dotnet build`在.NET中构建可执行文件会恢复包，将项目源编译为[程序集]。 程序集包含中间语言(IL)中的代码，通常可以在.NET支持的任何平台上运行，前提是主机上安装了.NET运行时。 来自依赖包的程序集通常与项目的输出程序集位于同一位置。 Rust中的`cargo build`执行相同的操作，只是Rust编译器将所有代码静态链接（尽管存在其他链接选项）为单个平台相关的二进制文件。

开发人员使用`dotnet publish`准备.NET可执行文件以供分发，无论是作为 *框架相关部署*(FDD)还是 *自包含部署*(SCD)。 在Rust中，没有与`dotnet publish`等同的功能，因为构建输出已经包含每个目标的单个、依赖于平台的二进制文件。

使用`dotnet build`在.NET中构建库时，它仍将生成包含IL的程序集。 在Rust中，构建输出同样是每个库目标的依赖于平台的已编译库。

另请参阅：

- Crate

依赖

在.NET中，项目文件的内容定义构建选项和依赖项。 在Rust中，当使用Cargo时，`Cargo.toml`会声明包的依赖项。 典型的项目文件如下所示：

```
<Project Sdk="Microsoft.NET.Sdk">

  <PropertyGroup>
    <OutputType>Exe</OutputType>
    <TargetFramework>net6.0</TargetFramework>
  </PropertyGroup>

  <ItemGroup>
    <PackageReference Include="morelinq" Version="3.3.2" />
  </ItemGroup>

</Project>
```

Rust中等效的`Cargo.toml`定义为：

```
[package]
name = "hello_world"
version = "0.1.0"

[dependencies]
tokio = "1.0.0"
```

Cargo遵循一个约定，即`src/main.rs`是与包同名的二进制包的包根。 同样，Cargo知道，如果包目录包含`src/lib.rs`，则包包含与包同名的库包。

包

NuGet最常用于安装软件包，各种工具都支持它。 例如，使用.NET CLI添加NuGet软件包引用将向项目文件添加依赖项：

dotnet add package morelinq

在Rust中，如果使用Cargo添加软件包，其工作原理几乎相同。

cargo add tokio

.NET最常见的软件包注册是nuget.org，Rust软件包通常通过crates.io共享。

静态代码分析

从.NET 5开始，Roslyn分析器与.NET SDK捆绑在一起，提供代码质量和代码样式分析。Rust中等效的linting工具是Clippy。

与.NET类似，如果通过将`TreatWarningsAsErrors`设置为`true`出现警告，构建就会失败，如果编译器或Clippy发出警告（`cargo clippy -- -D warnings`），Clippy也会失败。

还有进一步的静态检查可以考虑添加到Rust CI管道中：

- 运行`cargo doc`以确保文档正确。
- 运行`cargo check --locked`以强制`Cargo.lock`文件是最新的。