表8.1 默认接口重构特性

|  |  |
| --- | --- |
| **C# 8.0 引入的接口构造** | **示例代码** |
| **静态成员**  可以在接口上定义静态成员，包括字段、构造函数和方法（甚至可以定义一个静态 Main 方法，即程序的入口点）。接口上的静态成员的默认访问可访问性是public。 | public interface ISampleInterface  {  private static string? \_Field;  public static string? Field  {  get => \_Field;  private set => \_Field = value;  }  static ISampleInterface() =>  Field = "Nelson Mandela";  public static string? GetField() =>  Field;  } |
| **已实现的实例属性和方法**  可以在接口上定义已实现的属性和成员。由于不支持实例字段，所以属性不能依赖于支持字段。另外，也不支持自动实现的属性。  注意，为了访问默认实现的属性，必须把它转型为包含该成员的接口。在右边的例子中，除非Person类实现IPerson类规定的属性（名字、姓氏等），否则不可以使用默认接口成员Name。 | public interface IPerson  {  // 标准的抽象属性定义  string FirstName { get; set; }  string LastName { get; set; }  // 已实现的实例属性和方法  public string Name => GetName();  public string GetName() =>  $"{FirstName} {LastName}";  }  public class Person : IPerson  {  // ...  }  public class Program  {  public static void Main()  {  Person inigo = new  Person("Inigo", "Montoya");  Console.Write(((IPerson)inigo).Name);  }  } |
| **public 访问修饰符**  这是所有实例接口成员的默认可访问性。人为添加此关键字，有助于澄清代码的可访问性。但要注意，无论是否有public访问修饰符，编译器生成的 CIL 代码都是一样的。 | public interface IPerson  {  // 所有成员默认public  string FirstName { get; set; }  public string LastName { get; set; }  string Initials  => $"{FirstName[0]}{LastName[0]}";  public string Name => GetName();  public string GetName()  => $"{FirstName} {LastName}";  } |
| **protected 访问修饰符**  只能从当前接口或派生接口中访问。 | public interface IPerson  {  // ...  protected void Initialize() =>  { /\* ... \*/ };  } |
| **private 访问修饰符**  private 访问修饰符限制成员仅在声明它的接口中可用。它设计用于支持默认接口成员的重构。所有私有成员必须包括一个实现。 | public interface IPerson  {  string FirstName { get; set; }  string LastName { get; set; }  string Name => GetName();  private string GetName() =>  $"{FirstName} {LastName}";  } |
| **internal 访问修饰符**  internal 成员只在声明它们的同一程序集内可见。 | public interface IPerson  {  string FirstName { get; set; }  string LastName { get; set; }  string Name => GetName();  internal string GetName() =>  $"{FirstName} {LastName}";  } |
| **protected internal访问修饰符**  protected internal 成员是 protected和 internal的超集，其可见性限于同一程序集，以及从包容接口派生的其他接口。类似于protected成员，程序集外部的类看不到protected internal成员。 | public interface IPerson  {  string FirstName { get; set; }  string LastName { get; set; }  string Name => GetName();  protected internal string GetName() =>  $"{FirstName} {LastName}";  } |
| **private protected 访问修饰符**  只有在当前接口或派生接口中才能访问 private protected 成员。即使实现了接口的类也无法访问 private protected 成员。在右侧的Person类中，PersonTitle属性就是一个错误示范。 | class Program  {  static void Main()  {  IPerson? person = null;  // 实现接口的类不能调用  // private protected接口成员  \_ = person?.GetName();  Console.WriteLine(person);  }  }  public interface IPerson  {  string FirstName { get; }  string LastName { get; }  string Name => GetName();  private protected string GetName() =>  $"{FirstName} {LastName}";  }  public interface IEmployee : IPerson  {  int EmployeeId  => GetName().GetHashCode();  }  public class Person : IPerson  {  public Person(string firstName,  string lastName)  {  FirstName = firstName ??  throw new  ArgumentNullException(nameof(firstName));  LastName = lastName ??  throw new  ArgumentNullException(nameof(lastName));  }  public string FirstName { get; }  public string LastName { get; }  // private protected接口成员不能在实现  // 该接口的类中访问。  public string PersonTitle =>  GetName().ToUpper();  } |
| **virtual 修饰符**  接口中已实现的成员默认virtual，这意味着当调用接口成员时，会调用该方法具有相同签名的派生实现（如果有的话）。但是，和public访问修饰符一样，可以显式地将一个成员标记为virtual，从而增强可读性。未实现的接口成员不允许使用virtual修饰符。和其他时候一样，virtual修饰符与private，static和sealed修饰符不兼容。 | public interface IPerson  {  // 未提供实现的接口成员不允许virtual  string FirstName { get; set; }  string LastName { get; set; }  virtual string Name => GetName();  private string GetName() =>  $"{FirstName} {LastName}";  } |
| **sealed 修饰符**  为了防止实现接口的类重写方法，可以将其标记为 sealed，确保实现类无法修改方法实现。 | public interface IWorkflowActivity  {  // 私有，所以非虚  private static void Start() =>  Console.WriteLine(  "IWorkflowActivity.Start()...");  // 密封以防止重写  sealed void Run()  {  try  {  Start();  InternalRun();  }  finally  {  Stop();  }  }  protected void InternalRun();  // 私有，所以非虚  private static void Stop() =>  Console.WriteLine(  "IWorkflowActivity.Stop()..");  } |
| **abstract 修饰符**  只有未提供实现的成员才可以使用abstract 修饰符，但写等于没写，因为这种成员默认抽象。所有抽象成员都自动virtual；将抽象成员显式声明为virtual会引发编译错误。 | public interface IPerson  {  // 没有实现的成员可以abstract  /\* virtual \*/  abstract string FirstName { get; set; }  string LastName { get; set; }  // 有实现的成员不允许abstract  /\* abstract \*/  string Name => GetName();  private string GetName() =>  $"{FirstName} {LastName}";  } |
| **分部接口和分部方法**  现在可以提供方法的分部（partial）实现，这些方法不能有输出数据（return或ref/out），并可选择在同一接口的第二个声明中提供完全实现的方法。分部方法总是private——它们不支持访问修饰符。 | public partial interface IThing  {  string Value { get; protected set; }  void SetValue(string value)  {  AssertValueIsValid(value);  Value = value;  }  static partial void  AssertValueIsValid(string value);  }  public partial interface IThing  {  static partial void  AssertValueIsValid(string value)  {  // 值无效就抛出异常  switch (value)  {  case null:  throw new  ArgumentNullException(  nameof(value));  case "":  throw new ArgumentException(  "空串无效", nameof(value));  case string \_ when  string.IsNullOrWhiteSpace(value):  throw new ArgumentException(  "不能为空白字符",  nameof(value));  };  }  } |