# Взаимодействие с неуправляемыми кодом

17.06.2025

Платформа .NET Framework способствует взаимодействию с com-компонентами, службами COM+, библиотеками внешних типов и многими службами операционной системы. Типы данных, сигнатуры методов и механизмы обработки ошибок зависят от управляемых и неуправляемых объектных моделей. Чтобы упростить взаимодействие между компонентами .NET Framework и неуправляемым кодом и упростить путь миграции, среда CLR скрывает от клиентов и серверов различия в этих объектных моделях.

Код, который выполняется под контролем среды выполнения, называется управляемым кодом. И наоборот, код, выполняющийся вне среды выполнения, называется неуправляемый код. Примерами неуправляемого кода являются компоненты COM, интерфейсы ActiveX и функции Windows API.

## В этом разделе

Предоставление com-компонентов платформе .NET Framework
Описывает использование COM-компонентов из приложений .NET Framework.

Экспонирование компонентов платформы .NET Framework для COM
Описывает использование компонентов .NET Framework из COM-приложений.

Использование Неуправляемых Функций DLL
Описывает, как вызывать неуправляемые функции DLL с помощью вызова платформы.

Маршалинг межоперационных взаимодействий
Описывает маршалинг для COM-взаимодействия и вызова платформенных функций.

Практическое руководство. Сопоставление HRESULTs и исключений
Описывает сопоставление исключений и HRESULT.

Эквивалентность типов и внедренные типы взаимодействия
Описывает, как сведения о типах COM внедрены в сборки и как среда CLR определяет эквивалентность внедренных типов COM.

Практическое руководство. Создание основных сборок взаимодействия с помощью Tlbimp.exe

Описывается процесс создания первичных сборок взаимодействия с использованием *Tlbimp.exe* (Импортера библиотек типов).

Как: Регистрация основных сборок межоперационного взаимодействия
Описывает, как зарегистрировать основные сборки взаимодействия, прежде чем ссылаться на них в проектах.

Registration-Free COM-взаимодействие
Описывает, как COM-взаимодействие может активировать компоненты без использования реестра Windows.

Практическое руководство. Настройка com-компонентов .NET Framework-Based для активации Registration-Free
Описывает создание манифеста приложения и создание и внедрение манифеста компонента.

## Связанные разделы

COM-оболочки
Описывает оболочки, предоставляемые COM-взаимодействием.

# Предоставление доступа к COM-компонентам среде .NET Framework

17.06.2025

В этом разделе приводится сводка процесса, необходимого для предоставления существующего COM-компонента управляемому коду. Дополнительные сведения о написании COM-серверов, тесно интегрирующихся с .NET Framework, см. в разделе "Рекомендации по проектированию для взаимодействия".

Существующие компоненты COM — это ценные ресурсы в управляемом коде как бизнес-приложения среднего уровня или как изолированные функциональные возможности. Идеальный компонент имеет основную межоперационную сборку и соответствует стандартам программирования, используемым в COM.

## Для открытия доступа к COM-компонентам для платформы .NET Framework

1. Импорт библиотеки типов в виде сборки.

   Среда CLR требует метаданных для всех типов, включая COM-типы. Существует несколько способов получения сборки, содержащей типы COM, импортированные в виде метаданных.

2. Используйте типы COM в управляемом коде.

   Вы можете проверить типы COM, активировать экземпляры и вызвать методы в com-объекте так же, как и для любого управляемого типа.

3. Компиляция проекта интероперабельности.

   Пакет SDK для Windows предоставляет компиляторы для нескольких языков, совместимых со спецификацией CLS, включая Visual Basic, C#и C++.

4. Развертывание приложения взаимодействия.

   Приложения взаимодействия лучше всего развертываются как надежные и подписанные сборки в глобальном кэше сборок.

# См. также

- Взаимодействие с неуправляемым кодом
- Рекомендации по проектированию взаимодействия

- Пример взаимодействия COM: клиент .NET и COM-сервер
- независимость языка и компоненты Language-Independent
- Gacutil.exe (программа глобального кэша сборок)

# Импорт библиотеки типов в качестве сборки

17.06.2025

Определения типов COM обычно находятся в библиотеке типов. Напротив, компиляторы, соответствующие CLS, генерируют метаданные типов в сборке. Два источника сведений о типе совершенно разные. В этом разделе описываются методы создания метаданных из библиотеки типов. Результирующая сборка называется interop сборкой, а сведения о типах, содержащиеся в ней, позволяют приложениям .NET Framework использовать типы COM.

Существует два способа сделать эту информацию типа доступной для приложения:

- С использованием сборок взаимодействия исключительно во время проектирования: начиная с .NET Framework 4, можно инструктировать компилятор внедрять сведения о типе из сборки взаимодействия в ваш исполняемый файл. Компилятор внедряет только информацию о типах, которые использует ваше приложение. Вам не нужно развертывать сборку для взаимодействия с приложением. Это рекомендуемый метод.

- Развертывание сборок interop: Вы можете создать стандартную ссылку на interop-сборку. В этом случае сборка взаимодействия должна быть развернута непосредственно с вашим приложением. Если вы применяете этот метод и не используете приватный COM-компонент, всегда ссылайтесь на основную сборку взаимодействия (Primary Interop Assembly, PIA), опубликованную автором COM-компонента, который вы планируете включить в управляемый код. Дополнительные сведения о создании и использовании основных сборок взаимодействия см. в разделе "Основные сборки взаимодействия".

При использовании сборок взаимодействия во время разработки можно внедрить сведения о типе из основной сборки взаимодействия, опубликованной автором COM-компонента. Однако вам не нужно развертывать главную сборку взаимодействия вместе с приложением.

Использование сборок взаимодействия, предназначенных только для проектирования, сокращает размер приложения, потому что большинство приложений не используют все возможности COM-компонента. Компилятор очень эффективен при внедрении сведений о типе; Если приложение использует только некоторые методы в com-интерфейсе, компилятор не внедряет неиспользуемые методы. Если приложение, включающее сведения о внедренном типе, взаимодействует с другим таким приложением или с приложением, использующим основную сборку взаимодействия, то среда CLR использует

правила эквивалентности типов, чтобы определить, представляют ли два типа с одинаковым именем один и тот же тип COM. Эти правила не нужно знать, чтобы использовать COM-объекты. Однако если вы заинтересованы в правилах, см. раздел "Эквивалентность типов" и "Внедренные типы взаимодействия".

## Создание метаданных

Библиотеки типов COM могут быть автономными файлами с расширением TLB, например Loanlib.tlb. Некоторые библиотеки типов внедрены в раздел ресурсов файла .dll или .exe. Другими источниками сведений о библиотеке типов являются файлы .olb и .ocx.

После того как вы найдете библиотеку типов, содержащую реализацию целевого COM-типа, вы можете использовать следующие параметры для создания сборки взаимодействия для межоперабельности, включающей метаданные типа.

- Визуальная студия

  Visual Studio автоматически преобразует типы COM из библиотеки типов в метаданные в составе сборки. Инструкции см. в разделе "Практическое руководство. Добавление ссылок на библиотеки типов".

- Средство импорта библиотек типов (Tlbimp.exe)

  Средство импорта библиотек типов предоставляет параметры командной строки для настройки метаданных в результирующем файле взаимодействия, импортирует типы из существующей библиотеки типов и создает сборку взаимодействия и пространство имен. Инструкции см. в разделе "Практическое руководство. Создание сборок взаимодействия из библиотек типов".

- Класс System.Runtime.InteropServices.TypeLibConverter

  Этот класс предоставляет методы для преобразования соклассов и интерфейсов в библиотеку типов в метаданные в сборке. Он создает те же метаданные на выходе, что и Tlbimp.exe. Однако, в отличие от Tlbimp.exe, TypeLibConverter класс может преобразовать библиотеку типов в память в метаданные.

- Пользовательские оболочки

  Если библиотека типов недоступна или неправильная, один из вариантов — создать повторяющееся определение класса или интерфейса в управляемом исходном коде. Затем вы компилируете исходный код с компилятором, нацеленным на среду выполнения, чтобы создать метаданные в сборке.

Чтобы определить типы COM вручную, необходимо иметь доступ к следующим элементам:

- Точные описания определяемых coclasses и интерфейсов.
- Компилятор, например компилятор C#, который может создать соответствующие определения классов .NET Framework.
- Знание правил преобразования библиотеки типов в сборочное представление.

Написание пользовательской оболочки — это продвинутый метод. Дополнительные сведения о создании пользовательской оболочки см. в разделе "Настройка стандартных оболочков".

Дополнительные сведения о процессе импорта COM interop см. в сводке по преобразованию библиотеки типов в сборку.

## См. также

- TypeLibConverter
- Предоставление com-компонентов платформе .NET Framework
- Сводка о преобразовании библиотеки типов в сборку
- Tlbimp.exe (импорт библиотеки типов)
- Настройка стандартных оболочек
- Использование типов COM в управляемом коде
- Компиляция проекта Interop
- Развертывание приложения интероперабельности
- Практическое руководство. Добавление ссылок на библиотеки типов
- Практическое руководство. Создание сборок взаимодействия из библиотек типов

# Практическое руководство. Добавление ссылок на библиотеки типов

Статья • 02.03.2024

При добавлении ссылки на библиотеку типов Visual Studio генерирует сборку взаимодействия, в которой содержатся метаданные. Если первичная сборка взаимодействия доступна, Visual Studio обращается к существующей сборке, прежде чем генерировать новую.

## Добавление ссылки на библиотеку типов в Visual Studio

1. Если файл Windows Setup.exe не осуществит установку автоматически, установите DLL- или EXE-файл COM на компьютер.
2. Выберите **Проект**, **Добавить ссылку**.
3. В диспетчере ссылок выберите **COM**.
4. Выберите библиотеку типов из списка или найдите файл с расширением .TLB.
5. Выберите **OK**.
6. В обозревателе решений откройте контекстное меню добавленной ссылки и выберите **Свойства**.
7. Убедитесь, что в окне **Свойства** свойству **Внедрить типы взаимодействия** присвоено значение **True**. Visual Studio внедрит информацию о типах COM в исполняемые файлы, устранив тем самым необходимость развертывать основные сборки взаимодействия в приложении.

> ⓘ **Примечание**
>
> Пункты меню и параметры диалогового окна зависят от используемой версии Visual Studio.

## Добавление ссылки на библиотеку типов для компиляции командной строки

1. Создайте сборку взаимодействия, как описывается в разделе Практическое руководство. Создание сборок взаимодействия из библиотек типов.

2. Для внедрения информации о типах COM в исполняемые файлы используйте параметр компилятора -link (параметры компилятора C#) или -link (Visual Basic) с именем сборки взаимодействия.

## См. также

- Импорт библиотеки типов в виде сборки
- Предоставление клиентам .NET Framework доступа к COM-компонентам
- Пошаговое руководство. Внедрение типов из управляемых сборок в Visual Studio
- -link (параметры компилятора C#)
- -link (Visual Basic)

# Практическое руководство. Создание сборок взаимодействия их библиотек типов

Статья • 02.03.2024

Программа импорта библиотек типов (Tlbimp.exe) — это средство командной строки, которое преобразует коклассы и интерфейсы, содержащиеся в библиотеке типов COM, в метаданные. Это средство автоматически создает сборку взаимодействия и пространство имен для сведений о типе. После того как метаданные класса стали доступными, управляемые клиенты могут создавать экземпляры типа COM и вызывать его методы, как если бы это был экземпляр .NET. Средство Tlbimp.exe преобразует всю библиотеку типов в метаданные за один раз и не может создать сведения о типах для подмножества типов, определенных в библиотеке типов.

## Создание сборки взаимодействия из библиотеки типов

1. Используйте следующую команду:

   **tlbimp** <*файл_библиотеки_типов*>

   При указании параметра **/out:** создается сборка взаимодействия с измененным именем, например LOANLib.dll. Изменение имени сборки взаимодействия помогает отличить эту сборку от исходного файла DLL COM и избежать возможных проблем с повторяющимися именами.

# Пример

Следующая команда создает сборку Loanlib.dll в пространстве имен `Loanlib`.

Консоль

```
tlbimp Loanlib.tlb
```

Следующая команда создает сборку взаимодействия с измененным именем (LOANLib.dll).

Консоль

```
tlbimp LoanLib.tlb /out: LOANLib.dll
```

## См. также

- Импорт библиотеки типов в виде сборки
- Предоставление клиентам .NET Framework доступа к COM-компонентам

# Компиляция проекта интероперабельности

17.06.2025

Проекты взаимодействия COM, ссылающиеся на одну или несколько сборок, содержащих импортированные типы COM, компилируются как любой другой управляемый проект. Вы можете ссылаться на сборки взаимодействия в среде разработки, например Visual Studio, или ссылаться на них при использовании компилятора командной строки. В любом случае для правильной компиляции сборка взаимодействия должна находиться в том же каталоге, что и другие файлы проекта.

Существует два способа ссылки на интероперабельные сборки.

- Внедренные типы взаимодействия: начиная с .NET Framework 4 и Visual Studio 2010, можно указать компилятору внедрить сведения о типе из сборки взаимодействия в исполняемый файл. Это рекомендуемый метод.
- Развертывание сборок взаимодействия: Вы можете создать стандартную ссылку на сборку взаимодействия. В этом случае сборка взаимодействия должна быть развернута непосредственно с вашим приложением.

Различия между этими двумя методами подробно рассматриваются в использовании com-типов в управляемом коде.

Внедрение типов взаимодействия с Visual Studio демонстрируется в пошаговом руководстве. Внедрение типов из управляемых сборок в Visual Studio.

Чтобы ссылаться на сборку для взаимодействия с компилятором командной строки и внедрить информацию о типах в ваши исполняемые, используйте -link (параметры компилятора C#) или ключ компилятора -link (Visual Basic) и укажите имя сборки для взаимодействия.

> ⓘ **Примечание**
>
> Приложения Visual C++ не могут внедрять сведения о типах, но они могут взаимодействовать с приложениями или надстройками, которые могут это делать.

Чтобы скомпилировать приложение, включающее основную сборку взаимодействия при его развертывании, используйте переключатель компилятора **/reference** и укажите имя сборки взаимодействия.

# См. также

- Предоставление com-компонентов платформе .NET Framework
- независимость языка и компоненты Language-Independent
- Использование типов COM в управляемом коде
- Пошаговое руководство. Внедрение типов из управляемых сборок в Visual Studio
- Импорт библиотеки типов в качестве сборки

# Развертывание приложения интероперации

17.06.2025

Приложение взаимодействия обычно включает клиентную сборку .NET, одну или несколько сборок взаимодействия, представляющих отдельные библиотеки типов COM и один или несколько зарегистрированных COM-компонентов. Visual Studio и пакет SDK для Windows предоставляют средства для импорта и преобразования библиотеки типов в сборку взаимодействия, как описано в разделе импорта библиотеки типов в качестве сборки. Существует два способа развертывания приложения взаимодействия:

- Используя внедренные типы взаимодействия: начиная с .NET Framework 4, вы можете указать компилятору встроить информацию о типах из сборки взаимодействия в ваш исполняемый файл. Компилятор внедряет только информацию о типах, которые использует ваше приложение. Вам не нужно развертывать сборку для взаимодействия с приложением. Это рекомендуемый метод.

- При развертывании сборок взаимодействия, вы можете создать стандартную ссылку на сборку взаимодействия. В этом случае сборка взаимодействия должна быть развернута непосредственно с вашим приложением. Если вы применяете этот метод и не используете приватный COM-компонент, всегда ссылайтесь на основную сборку взаимодействия (Primary Interop Assembly, PIA), опубликованную автором COM-компонента, который вы планируете включить в управляемый код. Дополнительные сведения о создании и использовании основных сборок взаимодействия см. в разделе "Основные сборки взаимодействия".

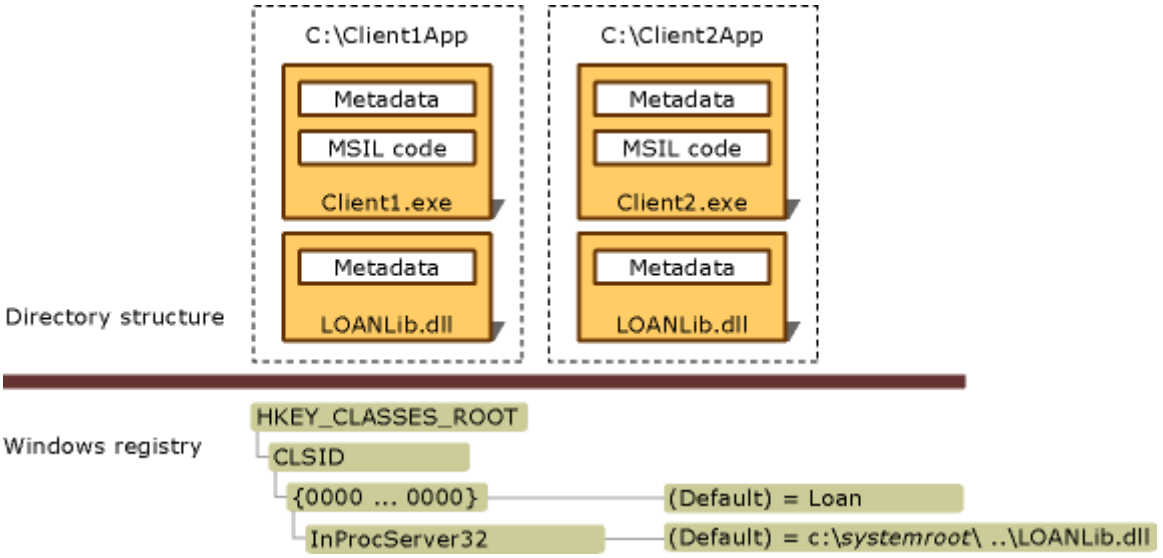
Если вы используете встроенные типы взаимодействия, развертывание является простым и незатруднительным. Нет ничего особенного, что вам нужно сделать. В остальной части этой статьи описываются сценарии развертывания и использования сборок для обеспечения взаимодействия вашего приложения.

## Развертывание сборок взаимодействия

Сборки могут иметь строгие имена. Сборка с строгим именем включает открытый ключ издателя, который предоставляет уникальную идентичность. Сборки, созданные импортом библиотеки типов (Tlbimp.exe), могут быть подписаны издателем с помощью параметра **/keyfile** . Подписанные сборки можно установить в глобальный кэш сборок. Неподписанные сборки должны быть установлены на компьютере пользователя в качестве частных сборок.

## Частные сборки

Чтобы установить сборку для частного использования, необходимо установить исполняемый файл приложения и сборку интероперабельности, содержащую импортированные типы COM, в одну структуру каталогов. На следующем рисунке показана не подписанная сборка для взаимодействия, предназначенная для частного использования в Client1.exe и Client2.exe, которые находятся в отдельных каталогах приложений. В этом примере сборка взаимодействия, которая называется LOANLib.dll, устанавливается дважды.




Все компоненты COM, связанные с приложением, должны быть установлены в реестре Windows. Если Client1.exe и Client2.exe на рисунке установлены на разных компьютерах, необходимо зарегистрировать com-компоненты на обоих компьютерах.

## Общие сборки

Сборки, совместно используемые несколькими приложениями, должны быть установлены в централизованном репозитории, называемом глобальным кэшем сборок. Клиенты .NET могут получить доступ к той же копии сборки взаимодействия, которая подписана и установлена в глобальном кэше сборок. Дополнительные сведения о создании и использовании основных сборок взаимодействия см. в разделе "Основные сборки взаимодействия".

# См. также

- Предоставление com-компонентов платформе .NET Framework
- Импорт библиотеки типов в качестве сборки
- Использование типов COM в управляемом коде
- Компиляция проекта Interop

# Пример взаимодействия COM: клиент .NET и COM-сервер

В этом примере показано, как клиент .NET, созданный для доступа к COM-серверу, создает экземпляр сокласса COM и вызывает членов класса для выполнения ипотечных вычислений.

В этом примере клиент создает и вызывает экземпляр сокласса `Loan` , передает четыре аргумента (одно из этих четырех равно нулю) экземпляру и отображает вычисления. Фрагменты кода из этого примера отображаются в этом разделе.

## Клиент .NET

C#

```
using System;
using LoanLib;

public class LoanApp {
   public static void Main(String[] Args) {

      Loan ln = new Loan();

      if (Args.Length < 4)
      {
         Console.WriteLine("Usage: ConLoan Balance Rate Term Payment");
         Console.WriteLine("    Either Balance, Rate, Term, or Payment
            must be 0");
         return;
      }

      ln.OpeningBalance = Convert.ToDouble(Args[0]);
      ln.Rate = Convert.ToDouble(Args[1])/100.0;
      ln.Term = Convert.ToInt16(Args[2]);
      ln.Payment = Convert.ToDouble(Args[3]);

      if (ln.OpeningBalance == 0.00) ln.ComputeOpeningBalance();
      if (ln.Rate == 0.00) ln.ComputeRate();
      if (ln.Term == 0) ln.ComputeTerm();
      if (ln.Payment == 0.00) ln.ComputePayment();

      Console.WriteLine("Balance = {0,10:0.00}", ln.OpeningBalance);
      Console.WriteLine("Rate    = {0,10:0.0%}", ln.Rate);
      Console.WriteLine("Term    = {0,10:0.00}", ln.Term);
      Console.WriteLine("Payment = {0,10:0.00}\n", ln.Payment);

      bool MorePmts;
      double Balance = 0.0;
      double Principal = 0.0;
```

```
      double Interest = 0.0;

      Console.WriteLine("{0,4}{1,10}{2,12}{3,10}{4,12}", "Nbr", "Payment",
        "Principal", "Interest", "Balance");
      Console.WriteLine("{0,4}{1,10}{2,12}{3,10}{4,12}", "---", "-------",
        "---------", "--------", "-------");

      MorePmts = ln.GetFirstPmtDistribution(ln.Payment, out Balance,
        out Principal, out Interest);

      for (short PmtNbr = 1; MorePmts; PmtNbr++) {
         Console.WriteLine("{0,4}{1,10:0.00}{2,12:0.00}{3,10:0.00}
           {4,12:0.00}", PmtNbr, ln.Payment, Principal, Interest,
            Balance);
         MorePmts = ln.GetNextPmtDistribution(ln.Payment, ref Balance,
           out Principal, out Interest);
      }
   }
}
```

# COM-сервер

C++

```
// Loan.cpp : Implementation of CLoan
#include "stdafx.h"
#include "math.h"
#include "LoanLib.h"
#include "Loan.h"

static double Round(double value, short digits);

STDMETHODIMP CLoan::get_OpeningBalance(double *pVal)
{
    *pVal = OpeningBalance;
    return S_OK;
}

STDMETHODIMP CLoan::put_OpeningBalance(double newVal)
{
    OpeningBalance = newVal;
    return S_OK;
}

STDMETHODIMP CLoan::get_Rate(double *pVal)
{
    *pVal = Rate;
    return S_OK;
}

STDMETHODIMP CLoan::put_Rate(double newVal)
{
```

```
    Rate = newVal;
    return S_OK;
}

STDMETHODIMP CLoan::get_Payment(double *pVal)
{
    *pVal = Payment;
    return S_OK;
}

STDMETHODIMP CLoan::put_Payment(double newVal)
{
    Payment = newVal;
    return S_OK;
}

STDMETHODIMP CLoan::get_Term(short *pVal)
{
    *pVal = Term;
    return S_OK;
}

STDMETHODIMP CLoan::put_Term(short newVal)
{
    Term = newVal;
    return S_OK;
}

STDMETHODIMP CLoan::ComputePayment(double *pVal)
{
    Payment = Round(OpeningBalance * (Rate /
        (1 - pow((1 + Rate), -Term))),2);
    *pVal = Payment;
    return S_OK;
}

STDMETHODIMP CLoan::ComputeOpeningBalance(double *pVal)
{
    OpeningBalance = Round(Payment / (Rate /
        (1 - pow((1 + Rate), -Term))),2);
    *pVal = OpeningBalance ;
    return S_OK;
}

STDMETHODIMP CLoan::ComputeRate(double *pVal)
{
    double DesiredPayment = Payment;

    for (Rate = 0.001; Rate < 28.0; Rate += 0.001)
    {
        Payment = Round(OpeningBalance * (Rate /
         (1 - pow((1 + Rate), -Term))),2);

        if (Payment >= DesiredPayment)
            break;
```

```
    }

    *pVal = Rate;
    return S_OK;
}

STDMETHODIMP CLoan::ComputeTerm(short *pVal)
{
    double DesiredPayment = Payment;
    for (Term = 1; Term < 480 ; Term ++)
    {
        Payment = Round(OpeningBalance * (Rate /
         (1 - pow((1 + Rate), -Term))),2);
        if (Payment <= DesiredPayment)
            break;
    }
    *pVal = Term;
    return S_OK;
}

STDMETHODIMP CLoan::GetFirstPmtDistribution(double PmtAmt, double *Balance, double
*PrinPortion, double *IntPortion, VARIANT_BOOL *pVal)
{
    *Balance = OpeningBalance;
    GetNextPmtDistribution(PmtAmt, Balance, PrinPortion, IntPortion,
      pVal);
    return S_OK;
}

STDMETHODIMP CLoan::GetNextPmtDistribution(double PmtAmt, double *Balance, double
*PrinPortion, double *IntPortion, VARIANT_BOOL *pVal)
{
    *IntPortion = Round(*Balance * Rate, 2);
    *PrinPortion = Round(PmtAmt - *IntPortion, 2);
    *Balance = Round(*Balance - *PrinPortion, 2);

    if (*Balance <= 0.0)
        *pVal = FALSE;
    else
        *pVal = TRUE;
    return S_OK;
}

STDMETHODIMP CLoan::get_RiskRating(BSTR *pVal)
{
    *pVal = (BSTR)RiskRating;
    return S_OK;
}

STDMETHODIMP CLoan::put_RiskRating(BSTR newVal)
{
    RiskRating = newVal;
    return S_OK;
}
static double Round(double value, short digits)
```

```
{
    double factor = pow(10, digits);
    return floor(value * factor + 0.5)/factor;
}
```

## См. также

- Предоставление com-компонентов платформе .NET Framework

---

Last updated on 02.12.2025

# Предоставление доступа .NET-компонентов для COM

17.06.2025

Написание типа .NET и потребление этого типа из неуправляемого кода — это отдельные задачи для разработчиков. В этом разделе описывается несколько советов по написанию управляемого кода, взаимодействующего с com-клиентами:

- Квалификация типов .NET для взаимодействия.

  Все управляемые типы, методы, свойства, поля и события, которые требуется предоставить COM, должны быть общедоступными. Типы должны иметь открытый конструктор без параметров, который является единственным конструктором, который можно вызвать через COM.

- Применение атрибутов взаимодействия.

  Пользовательские атрибуты в управляемом коде могут повысить совместимость компонента.

- Упаковка сборки для COM.

  Разработчики COM могут потребовать, чтобы вы предоставили обобщённое описание шагов, связанных с ссылкой и развертыванием ваших сборок.

Кроме того, этот раздел определяет задачи, связанные с потреблением управляемого типа из COM-клиента.

## Использование управляемого типа из COM

1. Регистрация сборок с помощью COM.

   Типы в сборке (и библиотеки типов) должны быть зарегистрированы во время разработки. Если установщик не регистрирует сборку, укажите разработчикам COM использовать Regasm.exe.

2. Ссылки на типы .NET из COM.

   Разработчики COM могут ссылаться на типы в сборке, используя те же средства и методы, которые они используют сегодня.

3. Вызовите объект .NET.

Разработчики COM могут вызывать методы для объекта .NET так же, как и методы для любого неуправляемого типа. Например, API COM **CoCreateInstance** активирует объекты .NET.

4. Развертывание приложения для com-доступа.

   Сборка с строгим именем может быть установлена в глобальном кэше сборок и требует подписи от своего издателя. Сборки, не имеющие строгого имени, должны быть установлены в каталоге приложений клиента.

## См. также

- Взаимодействие с неуправляемым кодом
- Пример взаимодействия COM: COM-клиент и сервер .NET

# Упаковка сборки .NET Framework для COM

17.06.2025

Разработчики COM могут воспользоваться следующими сведениями об управляемых типах, которые они планируют включить в свое приложение:

- Список типов, которые могут использовать COM-приложения

  Некоторые управляемые типы невидимы для COM; некоторые из них видны, но не могут быть создаваемыми; и некоторые из них являются видимыми и создаваемыми. Сборка может содержать любое сочетание невидимых, видимых, несоздаваемых и создаваемых типов. Для полноты определите типы в сборке, которую планируется предоставить COM, особенно если эти типы являются подмножеством типов, предоставляемых платформе .NET Framework.

  Дополнительные сведения см. в разделе "Квалификация типов .NET для взаимодействия".

- Инструкции по управлению версиями

  Управляемые классы, реализующие интерфейс класса (интерфейс, созданный для взаимодействия с COM), подвергаются ограничениям версионности.

  Рекомендации по использованию интерфейса класса см. в статье "Введение в интерфейс класса".

- Инструкции по развертыванию

  Сборки с строгими именами, подписанные издателем, можно установить в глобальное хранилище сборок. Неподписанные сборки должны быть установлены на компьютере пользователя в качестве частных сборок.

  Дополнительные сведения см. в разделе "Вопросы безопасности сборки".

- Добавление библиотеки типов

  Большинству типов требуется библиотека типов при использовании COM-приложения. Вы можете создать библиотеку типов или поручить разработчикам COM выполнить эту задачу. Пакет SDK для Windows предоставляет следующие параметры для создания библиотеки типов:

  - Экспортер библиотек типов

- Класс TypeLibConverter
- Средство регистрации сборок
- Средство установки служб .NET

  Независимо от выбранного механизма, в созданную библиотеку типов включены только общедоступные типы, определенные в заданной сборке.

См. инструкции в статье Практическое руководство: Внедрение библиотек типов в виде ресурсов Win32 в приложениях .NET-Based.

## Экспортер библиотек типов

Экспортер библиотек типов (Tlbexp.exe) — это средство командной строки, которое преобразует классы и интерфейсы, содержащиеся в сборке, в библиотеку типов COM. После получения сведений о типе класса COM-клиенты могут создать экземпляр класса .NET и вызвать методы экземпляра, как если бы он был COM-объектом. Tlbexp.exe преобразует всю сборку за один раз. Программу Tlbexp.exe нельзя использовать с целью генерации сведений о типах для подмножества типов, определенных в сборке.

## Класс TypeLibConverter

Класс TypeLibConverter , расположенный в пространстве имен **System.Runtime.Interop** , преобразует классы и интерфейсы, содержащиеся в сборке, в библиотеку типов COM. Этот API создает те же сведения о типе, что и экспортер библиотеки типов, описанные в предыдущем разделе.

**Класс TypeLibConverter** реализует ITypeLibConverter.

## Инструмент регистрации сборок

Средство регистрации сборок (Regasm.exe) может создавать и регистрировать библиотеку типов при применении параметра **/tlb**. Для клиентов COM требуется, чтобы библиотеки типов были установлены в реестре Windows. Без этого параметра Regasm.exe регистрирует только типы в сборке, а не библиотеку типов. Регистрация типов в сборке и регистрация библиотеки типов являются отдельными действиями.

## Средство установки служб .NET

Средство установки служб .NET (Regsvcs.exe) добавляет управляемые классы в службы компонентов Windows 2000 и объединяет несколько задач в одном средстве. Помимо загрузки и регистрации сборки, Regsvcs.exe может создавать, регистрировать и устанавливать библиотеку типов в существующее приложение COM+ 1.0.

## См. также

- TypeLibConverter
- ITypeLibConverter
- Экспонирование компонентов платформы .NET Framework для COM
- Определение типов .NET для взаимодействия
- Знакомство с интерфейсом класса
- Вопросы безопасности сборки
- Tlbexp.exe (экспортер библиотек типов)
- Регистрация сборок с помощью COM
- Практическое руководство. Внедрение библиотек типов в качестве ресурсов Win32 в приложениях

# Регистрация сборок с помощью COM

Вы можете запустить средство командной строки с именем средства регистрации сборок (Regasm.exe) для регистрации или отмены регистрации сборки для использования с COM. Regasm.exe добавляет сведения о классе в системный реестр, чтобы клиенты COM могли прозрачно использовать класс .NET Framework. Класс RegistrationServices предоставляет эквивалентные функциональные возможности.

Управляемый компонент должен быть зарегистрирован в реестре Windows, прежде чем его можно будет активировать из COM-клиента. В следующей таблице показаны ключи, которые Regasm.exe обычно добавляет в реестр Windows. (00000 указывает фактическое значение GUID.)

Развернуть таблицу

| ГУИД | Описание | Ключ реестра |
|---|---|---|
| CLSID | Идентификатор класса | HKEY_CLASSES_ROOT\CLSID\{000...000} |
| IID | Идентификатор интерфейса | HKEY_CLASSES_ROOT\Interface\{000...000} |
| LIBID | Идентификатор библиотеки | HKEY_CLASSES_ROOT\TypeLib\{000...000} |
| ProgID (идентификатор программы) | Программный идентификатор | HKEY_CLASSES_ROOT\000...000 |

В ключе HKCR\CLSID\{0000...0000} значение по умолчанию устанавливается в ProgID класса, а добавляются два новых именованных значения: Class и Assembly. Среда выполнения считывает значение сборки из реестра и передает его в сопоставитель сборок среды выполнения. Сопоставитель сборок пытается найти сборку на основе сведений о сборке, таких как имя и номер версии. Чтобы сопоставитель сборок мог найти сборку, сборка должна находиться в одном из следующих местоположений:

- Глобальный кэш сборок (должна иметь строгое имя).
- В каталоге приложения. Сборки, загруженные из каталога приложения, доступны только для этого приложения.
- По пути, указанному с помощью параметра **/codebase**, до Regasm.exe.

Regasm.exe также создает ключ InProcServer32 под HKCR\CLSID\{0000...0000}. Значение по умолчанию для ключа имеет имя библиотеки DLL, которая инициализирует среду CLR (Mscoree.dll).

# Изучение записей реестра

COM-взаимодействие предоставляет стандартную реализацию фабрики классов для создания экземпляра любого класса .NET Framework. Клиенты могут вызывать `DllGetClassObject` в управляемой библиотеке DLL, чтобы получить фабрику классов и создать объекты, так же как и в случае с любым другим COM-компонентом.

Для подпункта `InprocServer32` ссылка на Mscoree.dll отображается вместо традиционной библиотеки типов COM, чтобы указать, что общая среда выполнения создает управляемый объект.

# См. также

- Экспонирование компонентов платформы .NET Framework для COM
- Как ссылаться на типы .NET из COM
- Вызов объекта .NET
- Развертывание приложения для COM-доступа

---

Last updated on 02.12.2025

# Как: ссылаться на типы .NET из COM

С точки зрения кода клиента и сервера различия между COM и .NET Framework в значительной степени невидимы. Клиенты Microsoft Visual Basic могут просматривать объект .NET в браузере объектов, который предоставляет методы объекта и синтаксис, свойства и поля точно так же, как если бы он был любым другим COM-объектом.

Процесс импорта библиотеки типов немного сложнее для клиентов C++, хотя вы используете те же средства для экспорта метаданных в библиотеку типов COM. Чтобы обращаться к членам объектов .NET из неуправляемого клиента C++, используйте TLB-файл (созданный с помощью Tlbexp.exe) с директивой **#import**. При ссылке на библиотеку типов из C++необходимо указать `raw_interfaces_only` параметр или импортировать определения в библиотеке базовых классов Mscorlib.tlb.

## Импорт библиотеки

- `raw_interfaces_only` Укажите параметр в директиве **#import**. Рассмотрим пример.

  C++

  ```
  #import "..\LoanLib\LoanLib.tlb" raw_interfaces_only
  ```

  –или–

- Включите директиву #import для Mscorlib.tlb. Рассмотрим пример.

  C++

  ```
  #import "mscorlib.tlb"
  #import "..\LoanLib\LoanLib.tlb"
  ```

# См. также

- Предоставление доступа .NET Framework к COM
- Регистрация сборок с помощью COM
- Вызов объекта .NET
- Развертывание приложения для COM-доступа

---

Last updated on 02.12.2025

# Пример взаимодействия COM: COM-клиент и сервер .NET

17.06.2025

В этом примере демонстрируется взаимодействие COM-клиента и сервера .NET , выполняющего ипотечные вычисления. В этом примере клиент создает и вызывает экземпляр управляемого `Loan` класса, передает четыре аргумента (одно из этих четырех равно нулю) экземпляру и отображает вычисления. Примеры кода из этого примера отображаются в этом разделе.

## COM-клиент

C++

```
// ConLoan.cpp : Defines the entry point for the console application.
#include "stdafx.h"
#import "..\LoanLib\LoanLib.tlb" raw_interfaces_only
using namespace LoanLib;

int main(int argc, char* argv[])
{
    HRESULT hr = CoInitialize(NULL);

    ILoanPtr pILoan(__uuidof(Loan));

    if (argc < 5)
    {
        printf("Usage: ConLoan Balance Rate Term Payment\n");
        printf("    Either Balance, Rate, Term, or Payment must be 0\n");
        return -1;
    }

    double openingBalance = atof(argv[1]);
    double rate = atof(argv[2])/100.0;
    short  term = atoi(argv[3]);
    double payment = atof(argv[4]);

    pILoan->put_OpeningBalance(openingBalance);
    pILoan->put_Rate(rate);
    pILoan->put_Term(term);
    pILoan->put_Payment(payment);

    if (openingBalance == 0.00)
         pILoan->ComputeOpeningBalance(&openingBalance);
    if (rate == 0.00) pILoan->ComputeRate(&rate);
    if (term == 0) pILoan->ComputeTerm(&term);
    if (payment == 0.00) pILoan->ComputePayment(&payment);
```

```
    printf("Balance = %.2f\n", openingBalance);
    printf("Rate    = %.1f%%\n", rate*100);
    printf("Term    = %.2i\n", term);
    printf("Payment = %.2f\n", payment);

    VARIANT_BOOL MorePmts;
    double Balance = 0.0;
    double Principal = 0.0;
    double Interest = 0.0;

    printf("%4s%10s%12s%10s%12s\n", "Nbr", "Payment", "Principal", "Interest",
"Balance");
    printf("%4s%10s%12s%10s%12s\n", "---", "-------", "---------",
"--------", "-------");

    pILoan->GetFirstPmtDistribution(payment, &Balance, &Principal, &Interest,
&MorePmts);

    for (short PmtNbr = 1; MorePmts; PmtNbr++)
    {
        printf("%4i%10.2f%12.2f%10.2f%12.2f\n",
        PmtNbr, payment, Principal, Interest, Balance);

        pILoan->GetNextPmtDistribution(payment, &Balance, &Principal, &Interest,
&MorePmts);
    }

    CoUninitialize();
    return 0;
}
```

# Сервер .NET

C#

```
using System;
using System.Reflection;

[assembly:AssemblyKeyFile("sample.snk")]
namespace LoanLib {

    public interface ILoan {
        double OpeningBalance{get; set;}
        double Rate{get; set;}
        double Payment{get; set;}
        short  Term{get; set;}
        String RiskRating{get; set;}

        double ComputePayment();
        double ComputeOpeningBalance();
        double ComputeRate();
        short ComputeTerm();
```

```
        bool GetFirstPmtDistribution(double PmtAmt, ref double Balance,
            out double PrinPortion, out double IntPortion);
        bool GetNextPmtDistribution(double PmtAmt, ref double Balance,
            out double PrinPortion, out double IntPortion);
    }

    public class Loan : ILoan {
        private double openingBalance;
        private double rate;
        private double payment;
        private short  term;
        private String riskRating;

        public double OpeningBalance {
            get { return openingBalance; }
            set { openingBalance = value; }
        }

        public double Rate {
            get { return rate; }
            set { rate = value; }
        }

        public double Payment {
            get { return payment; }
            set { payment = value; }
        }

        public short Term {
            get { return term; }
            set { term = value; }
        }

        public String RiskRating {
            get { return riskRating; }
            set { riskRating = value; }
        }

        public double ComputePayment() {
             Payment = Util.Round(OpeningBalance * (Rate / (1 -
                       Math.Pow((1 + Rate), -Term))), 2);
             return Payment;
        }

        public double ComputeOpeningBalance() {
            OpeningBalance = Util.Round(Payment / (Rate / (1 - Math.Pow((1
                             + Rate), -Term))), 2);
             return OpeningBalance;
        }

        public double ComputeRate() {
            double DesiredPayment = Payment;

            for (Rate = 0.001; Rate < 28.0; Rate += 0.001) {
                Payment = Util.Round(OpeningBalance * (Rate / (1 -
```

```
                        Math.Pow((1 + Rate), -Term))), 2);

                if (Payment >= DesiredPayment)
                    break;
            }
            return Rate;
        }

        public short ComputeTerm() {
            double DesiredPayment = Payment;

            for (Term = 1; Term < 480 ; Term ++) {
                Payment = Util.Round(OpeningBalance * (Rate / (1 -
                        Math.Pow((1 + Rate), -Term))),2);

                if (Payment <= DesiredPayment)
                    break;
            }

            return Term;
        }

        public bool GetFirstPmtDistribution(double PmtAmt, ref double
            Balance, out double PrinPortion, out double IntPortion) {
             Balance = OpeningBalance;
             return GetNextPmtDistribution(PmtAmt, ref Balance, out
             PrinPortion, out IntPortion);
        }

        public bool GetNextPmtDistribution(double PmtAmt, ref double
           Balance, out double PrinPortion, out double IntPortion) {
            IntPortion = Util.Round(Balance * Rate, 2);
            PrinPortion = Util.Round(PmtAmt - IntPortion,2);
            Balance = Util.Round(Balance - PrinPortion,2);

            if (Balance <= 0.0)
                return false;

            return true;
        }
     }

    internal class Util {
        public static double Round(double value, short digits) {
            double factor = Math.Pow(10, digits);
            return Math.Round(value * factor) / factor;
         }
    }
}
```

## См. также

- Экспонирование компонентов платформы .NET Framework для COM

# Использование неуправляемых функций DLL

17.06.2025

Вызов платформы — это служба, которая позволяет управляемому коду вызывать неуправляемые функции, реализованные в библиотеках динамических ссылок (DLL), например в API Windows. Он находит и вызывает экспортированную функцию и маршалирует свои аргументы (целые числа, строки, массивы, структуры и т. д.) через границу взаимодействия по мере необходимости.

В этом разделе приводятся задачи, связанные с использованием неуправляемых функций DLL и содержатся дополнительные сведения о вызове платформы. В дополнение к следующим задачам существуют общие рекомендации и ссылка, предоставляющая дополнительные сведения и примеры.

## Для использования экспортированных функций DLL

1. Определение функций в библиотеках DLL.

   Минимально необходимо указать имя функции и имя библиотеки DLL, содержащей ее.

2. Создайте класс для хранения функций DLL.

   Можно использовать существующий класс, создать отдельный класс для каждой неуправляемой функции или создать один класс, содержащий набор связанных неуправляемых функций.

3. Создание прототипов в управляемом коде.

   [Visual Basic] Используйте инструкцию **Declare** с ключевыми словами **Function** и **Lib** . В некоторых редких случаях можно использовать **DllImportAttribute** с ключевыми словами **Shared Function**. Эти случаи описаны далее в этом разделе.

   [C#] Используйте **dllImportAttribute** для идентификации библиотеки DLL и функции. Пометьте метод модификаторами **static** и **extern**.

   [C++] Используйте **dllImportAttribute** для идентификации библиотеки DLL и функции. Пометьте метод или функцию-оболочку с использованием **extern "C"**.

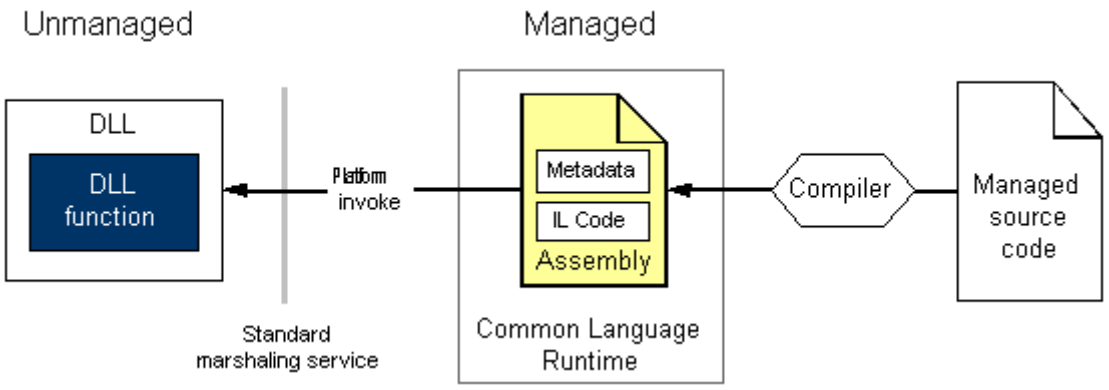
4. Вызов функции DLL.

Вызовите метод в управляемом классе, как и любой другой управляемый метод. Передача структур и реализация функций обратного вызова являются особыми случаями.

Примеры, демонстрирующие, как создавать объявления на основе .NET для использования с вызовом платформы, см. в разделе «Маршаллирование данных с помощью вызова платформы».

# Более подробный взгляд на вызов функций платформы

Вызов платформы (Platform Invoke) использует метаданные для поиска экспортированных функций и маршалирования их аргументов во время исполнения. На следующем рисунке показан этот процесс.




Когда механизм платформы вызывает неуправляемую функцию, он выполняет следующую последовательность действий:

1. Находит библиотеку DLL, содержащую функцию.
2. Загружает библиотеку DLL в память.
3. Находит адрес функции в памяти и отправляет его аргументы в стек, упаковывая данные по мере необходимости.

   > ⓘ **Примечание**
   >
   > Поиск и загрузка библиотеки DLL и поиск адреса функции в памяти происходят только при первом вызове функции.

4. Передает управление неуправляемой функции.

Вызов платформы создает исключения, созданные неуправляемой функцией для управляемого вызывающего объекта.

## См. также

- Взаимодействие с неуправляемым кодом
- Примеры вызова платформы
- Маршалинг межоперационных взаимодействий

# Идентификация функций в библиотеках DLL

17.06.2025

Идентичность функции библиотеки DLL состоит из следующих элементов:

- Имя функции или порядковый номер
- Имя DLL-файла, в котором можно найти реализацию

Например, указание функции **MessageBox** в User32.dll определяет функцию (**MessageBox**) и его расположение (User32.dll, User32 или user32). Интерфейс программирования приложений Microsoft Windows (API Windows) может содержать две версии каждой функции, обрабатывающей символы и строки: 1-байтовую версию ANSI и 2-байтовую версию Юникода. Если не указано, набор символов, представленный CharSet полем, по умолчанию использует ANSI. Некоторые функции могут иметь более двух версий.

**MessageBoxA** — это точка входа ANSI для функции **MessageBox** ; **MessageBoxW** — это версия Юникода. Вы можете перечислить имена функций для определенной библиотеки DLL, например user32.dll, выполнив различные средства командной строки. Например, можно использовать `dumpbin /exports user32.dll` или `link /dump /exports user32.dll` получить имена функций.

Вы можете переименовать неуправляемую функцию в любое, что вам нравится в коде, если вы сопоставляете новое имя с исходной точкой входа в библиотеке DLL. Инструкции по переименованию неуправляемой функции DLL в управляемом исходном коде см. в разделе "Указание точки входа".

Вызов функций платформы позволяет контролировать значительную часть функций операционной системы, вызывая функции из API Windows и других DLL. Помимо API Windows, существует множество других API и библиотек DLL, доступных для вас через вызов платформы.

В следующей таблице описано несколько часто используемых библиотек DLL в API Windows.

Развернуть таблицу

| DLL (динамическая библиотека) | Описание содержимого |
|---|---|
| GDI32.dll | Функции интерфейса графического устройства (GDI) для выходных данных устройства, например для управления рисунками и шрифтами. |

| DLL (динамическая библиотека) | Описание содержимого |
| --- | --- |
| Kernel32.dll | Низкоуровневые функции операционной системы для управления памятью и обработки ресурсов. |
| User32.dll | Функции управления Windows для обработки сообщений, таймеров, меню и обмена данными. |

Полную документацию по Windows API можно увидеть в SDK платформы. Примеры, демонстрирующие, как создавать объявления на основе .NET для использования с вызовом платформы, см. в разделе «Маршаллирование данных с помощью вызова платформы».

# См. также

- Использование Неуправляемых Функций DLL
- Указание точки входа
- Создание класса для хранения функций DLL
- Создание прототипов в управляемом коде
- Вызов функции DLL

# Создание класса для хранения функций DLL

17.06.2025

Упаковка часто используемой функции DLL в управляемом классе — эффективный подход к инкапсулировать функциональные возможности платформы. Хотя это не обязательно сделать в каждом случае, предоставление оболочки класса удобно, так как определение функций DLL может быть громоздким и подверженным ошибкам. Если вы программируете в Visual Basic или C#, необходимо объявить функции DLL в классе или модуле Visual Basic.

В классе определяется статический метод для каждой функции DLL, которую требуется вызвать. Определение может включать дополнительные сведения, такие как набор символов или соглашение о вызове, используемое при передаче аргументов метода; Пропуская эти сведения, вы выбираете параметры по умолчанию. Полный список параметров объявления и их значений по умолчанию см. в разделе "Создание прототипов в управляемом коде".

После обёртывания можно вызвать методы класса так же, как статические методы в любом другом классе. Вызов платформы автоматически обрабатывает базовую экспортированную функцию.

При разработке управляемого класса для вызова платформы следует учитывать связи между классами и функциями DLL. Например, доступны следующие возможности:

- Объявите функции DLL в существующем классе.
- Создайте отдельный класс для каждой функции DLL, сохраняя изолированные и простые для поиска функции.
- Создайте один класс для набора связанных функций DLL для формирования логических групп и снижения затрат.

Вы можете присвоить классу и его методам любое имя по вашему усмотрению. Примеры, демонстрирующие, как создавать объявления на основе .NET для использования с вызовом платформы, см. в разделе «Маршаллирование данных с помощью вызова платформы».

## См. также

- Использование Неуправляемых Функций DLL

- Определение функций в библиотеках DLL
- Создание прототипов в управляемом коде
- Вызов функции DLL

# Создание прототипов в управляемом коде

17.06.2025

В этом разделе описывается, как получить доступ к неуправляемым функциям и ввести несколько полей атрибутов, которые аннотируют определение метода в управляемом коде. Примеры, демонстрирующие, как создавать объявления на основе .NET для использования с вызовом платформы, см. в разделе «Маршаллирование данных с помощью вызова платформы».

Прежде чем получить доступ к неуправляемой функции DLL из управляемого кода, необходимо знать имя функции и имя библиотеки DLL, экспортируемой ею. С помощью этих сведений можно начать запись управляемого определения для неуправляемой функции, реализованной в библиотеке DLL. Кроме того, можно настроить способ, которым механизм вызова на платформе создает функцию и осуществляет передачу данных в функцию и обратно.

> ⓘ **Примечание**
>
> Функции API Windows, которые выделяют строку, позволяют освободить строку с помощью такого метода, как `LocalFree`. Вызов платформы обрабатывает такие параметры по-разному. Для вызовов платформы сделайте параметр типом `IntPtr` вместо `String` типа. Используйте методы, предоставляемые классом System.Runtime.InteropServices.Marshal , чтобы преобразовать тип в строку вручную и освободить его вручную.

## Основы декларации

Управляемые определения для неуправляемых функций зависят от языка, как показано в следующих примерах. Дополнительные примеры кода см. в разделе "Примеры вызова платформы".

VB

```
Friend Class NativeMethods
    Friend Declare Auto Function MessageBox Lib "user32.dll" (
        ByVal hWnd As IntPtr,
        ByVal lpText As String,
        ByVal lpCaption As String,
```

```
        ByVal uType As UInteger) As Integer
End Class
```

Чтобы применить поля DllImportAttribute.BestFitMapping, DllImportAttribute.CallingConvention, DllImportAttribute.ExactSpelling, DllImportAttribute.PreserveSig, DllImportAttribute.SetLastError или DllImportAttribute.ThrowOnUnmappableChar к объявлению Visual Basic, необходимо использовать атрибут DllImportAttribute вместо оператора `Declare`.

C#

```
using System;
using System.Runtime.InteropServices;

internal static class NativeMethods
{
    [DllImport("user32.dll")]
    internal static extern int MessageBox(
        IntPtr hWnd, string lpText, string lpCaption, uint uType);
}
```

# Настройка определения

Заданы они явным образом или нет, поля атрибутов активно определяют поведение управляемого кода. Вызов платформы работает в соответствии со значениями по умолчанию, заданными в различных полях, которые существуют как метаданные в сборке. Это поведение по умолчанию можно изменить, изменив значения одного или нескольких полей. Во многих случаях используется DllImportAttribute для задания значения.

В следующей таблице приведен полный набор полей атрибутов, относящихся к вызову платформы. Для каждого поля таблица содержит значение по умолчанию и ссылку на сведения об использовании этих полей для определения неуправляемых функций DLL.

Развернуть таблицу

| Поле | Описание |
| --- | --- |
| BestFitMapping | Включает или отключает оптимальное сопоставление. |
| CallingConvention | Указывает соглашение о вызове, используемое при передаче аргументов метода. По умолчанию устанавливается значение `WinAPI`, которое для 32-разрядных платформ на базе Intel соответствует `__stdcall`. |

| Поле | Описание |
| --- | --- |
| CharSet | Управляет изменением имени и способом маршаллинга строковых аргументов в функцию. Значение по умолчанию — `CharSet.Ansi`. |
| EntryPoint | Указывает вызываемую точку входа DLL. |
| ExactSpelling | Определяет, следует ли изменить точку входа, чтобы соответствовать набору символов. Значение по умолчанию зависит от языка программирования. |
| PreserveSig | Определяет, следует ли преобразовать сигнатуру управляемого метода в неуправляемую подпись, которая возвращает HRESULT и имеет дополнительный аргумент [out, retval] для возвращаемого значения.<br><br>Значением по умолчанию является `true` (подпись не должна быть преобразована). |
| SetLastError | Позволяет вызывающему объекту использовать `Marshal.GetLastWin32Error` функцию API, чтобы определить, произошла ли ошибка при выполнении метода. В Visual Basic используется `true` значение по умолчанию; в C# и C++, значение по умолчанию — `false`. |
| ThrowOnUnmappableChar | Управляет вызовом исключения для неуправляемого символа Юникода, преобразованного в символ ANSI "?" |

Подробные справочные сведения см. в разделе DllImportAttribute.

# Рекомендации по обеспечению безопасности при вызове процедур на платформе

Элементы `Assert`, `Deny`, и `PermitOnly` перечисления SecurityAction называются *модификаторами пошагового выполнения стека*. Эти члены игнорируются, если они используются в качестве декларативных атрибутов в объявлениях вызова платформы и инструкциях языка определения COM-интерфейса (IDL).

## Примеры вызова функций платформы

Примеры вызова платформы в этом разделе демонстрируют использование атрибута `RegistryPermission` с модификаторами прохождения стека.

В следующем примере модификаторы SecurityAction, `Assert`, `Deny` и `PermitOnly` игнорируются.

C#

```
[DllImport("MyClass.dll", EntryPoint = "CallRegistryPermission")]
[RegistryPermission(SecurityAction.Assert, Unrestricted = true)]
    private static extern bool CallRegistryPermissionAssert();

[DllImport("MyClass.dll", EntryPoint = "CallRegistryPermission")]
[RegistryPermission(SecurityAction.Deny, Unrestricted = true)]
    private static extern bool CallRegistryPermissionDeny();

[DllImport("MyClass.dll", EntryPoint = "CallRegistryPermission")]
[RegistryPermission(SecurityAction.PermitOnly, Unrestricted = true)]
    private static extern bool CallRegistryPermissionDeny();
```

Однако модификатор `Demand` в следующем примере допускается.

C#

```
[DllImport("MyClass.dll", EntryPoint = "CallRegistryPermission")]
[RegistryPermission(SecurityAction.Demand, Unrestricted = true)]
    private static extern bool CallRegistryPermissionDeny();
```

SecurityAction модификаторы работают правильно, если они помещаются в класс, содержащий (оболочки) вызов платформы.

C#

```
[RegistryPermission(SecurityAction.Demand, Unrestricted = true)]
class PInvokeWrapper
{
[DllImport("MyClass.dll", EntryPoint = "CallRegistryPermission")]
    private static extern bool CallRegistryPermissionDeny();
}
```

SecurityAction Модификаторы также работают правильно в вложенном сценарии, где они размещаются на вызывающей стороне вызова платформы.

C#

```
class PInvokeScenario
{
    [DllImport("MyClass.dll", EntryPoint = "CallRegistryPermission")]
    private static extern bool CallRegistryPermissionInternal();

    [RegistryPermission(SecurityAction.Assert, Unrestricted = true)]
    public static bool CallRegistryPermission()
    {
     return CallRegistryPermissionInternal();
```

```
    }
}
```

## Примеры взаимодействия COM

Примеры взаимодействия COM в этом разделе иллюстрируют использование атрибута `RegistryPermission` с модификаторами пошаговой проверки стека.

В следующих объявлениях интерфейсов взаимодействия COM игнорируются модификаторы `Assert`, `Deny` и `PermitOnly`, подобно примерам вызова платформы в предыдущем разделе.

C#

```
[ComImport, Guid("12345678-43E6-43c9-9A13-47F40B338DE0")]
interface IAssertStubsItf
{
[RegistryPermission(SecurityAction.Assert, Unrestricted = true)]
    bool CallRegistryPermission();
[FileIOPermission(SecurityAction.Assert, Unrestricted = true)]
    bool CallFileIoPermission();
}

[ComImport, Guid("12345678-43E6-43c9-9A13-47F40B338DE0")]
interface IDenyStubsItf
{
[RegistryPermission(SecurityAction.Deny, Unrestricted = true)]
    bool CallRegistryPermission();
[FileIOPermission(SecurityAction.Deny, Unrestricted = true)]
    bool CallFileIoPermission();
}

[ComImport, Guid("12345678-43E6-43c9-9A13-47F40B338DE0")]
interface IAssertStubsItf
{
[RegistryPermission(SecurityAction.PermitOnly, Unrestricted = true)]
    bool CallRegistryPermission();
[FileIOPermission(SecurityAction.PermitOnly, Unrestricted = true)]
    bool CallFileIoPermission();
}
```

Кроме того, `Demand` модификатор не принимается в сценариях объявления интерфейса взаимодействия COM, как показано в следующем примере.

C#

```
[ComImport, Guid("12345678-43E6-43c9-9A13-47F40B338DE0")]
interface IDemandStubsItf
```

```
{
[RegistryPermission(SecurityAction.Demand, Unrestricted = true)]
    bool CallRegistryPermission();
[FileIOPermission(SecurityAction.Demand, Unrestricted = true)]
    bool CallFileIoPermission();
}
```

## См. также

- Использование Неуправляемых Функций DLL
- Указание точки входа
- Указание набора символов
- Примеры вызова платформы
- Соображения безопасности Platform Invoke
- Определение функций в библиотеках DLL
- Создание класса для хранения функций DLL
- Вызов функции DLL

# Указание точки входа

Точка входа определяет расположение функции в библиотеке DLL. В управляемом проекте исходная точка входа или порядковый номер целевой функции определяет эту функцию через границу взаимодействия. Кроме того, можно сопоставить точку входа с другим именем, фактически переименовав функцию.

Ниже приведен список возможных причин переименования функции DLL:

- Чтобы избежать использования имен функций API с учетом регистра
- Соответствие существующим стандартам именования
- Для размещения функций, которые принимают различные типы данных (объявляя несколько версий одной и той же функции DLL)
- Упрощение использования API, содержащих версии ANSI и Юникода

В этом разделе показано, как переименовать функцию DLL в управляемом коде.

# Переименование функции в Visual Basic

Visual Basic использует ключевое `Function` слово в `Declare` инструкции для задания DllImportAttribute.EntryPoint поля. В следующем примере показано базовое объявление.

VB

```
Friend Class NativeMethods
    Friend Declare Auto Function MessageBox Lib "user32.dll" (
        ByVal hWnd As IntPtr,
        ByVal lpText As String,
        ByVal lpCaption As String,
        ByVal uType As UInteger) As Integer
End Class
```

Точку входа `MessageBox` можно заменить на `MsgBox`, включив ключевое слово `Alias` в определение, как это продемонстрировано в следующем примере. В обоих примерах `Auto` ключевое слово устраняет необходимость указать версию символьного набора точки входа. Дополнительные сведения о выборе набора символов см. в разделе "Указание набора символов".

VB

```
Friend Class NativeMethods
    Friend Declare Auto Function MsgBox _
        Lib "user32.dll" Alias "MessageBox" (
```

```
        ByVal hWnd As IntPtr,
        ByVal lpText As String,
        ByVal lpCaption As String,
        ByVal uType As UInteger) As Integer
End Class
```

## Переименование функции в C# и C++

Вы можете использовать поле DllImportAttribute.EntryPoint для указания функции DLL по имени или порядковому номеру. Если имя функции в определении вашего метода совпадает с точкой входа в библиотеке DLL, вам не нужно явно указывать функцию с полем `EntryPoint`. В противном случае используйте одну из следующих форм атрибутов, чтобы указать имя или порядковый номер:

C#

```
[DllImport("DllName", EntryPoint = "Functionname")]
[DllImport("DllName", EntryPoint = "#123")]
```

Обратите внимание, что необходимо префикс порядкового номера с знаком фунта (#).

В следующем примере показано, как заменить `MessageBoxA` на `MsgBox` в вашем коде, используя поле `EntryPoint`.

C#

```
using System;
using System.Runtime.InteropServices;

internal static class NativeMethods
{
    [DllImport("user32.dll", EntryPoint = "MessageBoxA")]
    internal static extern int MsgBox(
        IntPtr hWnd, string lpText, string lpCaption, uint uType);
}
```

C++

```
using namespace System;
using namespace System::Runtime::InteropServices;

typedef void* HWND;
[DllImport("user32", EntryPoint = "MessageBoxA")]
extern "C" int MsgBox(
    HWND hWnd, String* lpText, String* lpCaption, unsigned int uType);
```

# См. также

- DllImportAttribute
- Создание прототипов в управляемом коде
- Примеры вызова платформы
- Маршалинг данных с использованием Platform Invoke

---

Last updated on 02.12.2025

# Указание набора символов

17.06.2025

Поле DllImportAttribute.CharSet управляет маршаллингом строк данных и определяет способ, которым вызов платформы находит имена функций в библиотеке DLL. В этом разделе описывается оба поведения.

Некоторые API экспортируют две версии функций, которые принимают строковые аргументы: узкие (ANSI) и широкие (Юникод). Например, API Windows содержит следующие имена точек входа для функции **MessageBox** :

- **MessageBoxA**

  Предоставляет ANSI форматирование с 1 байтом, отличающееся добавлением "A" к названию точки входа. Вызовы **MessageBoxA** всегда маршалируют строки в формате ANSI.

- **MessageBoxW**

  Предоставляет форматирование 2-байтового символа Юникода, обозначенное добавлением "W" к имени входной точки. Вызовы **MessageBoxW** всегда маршалируют строки в формате Юникода.

# Сопоставление строк и сопоставления имен

Поле `CharSet` принимает следующие значения:

Ansi (значение по умолчанию)

- Маршалирование строк

  Платформа вызывает строки маршалов из управляемого формата (Юникод) в формат ANSI.

- Сопоставление имен

  Когда поле DllImportAttribute.ExactSpelling задано как `true`, что происходит по умолчанию в Visual Basic, платформа выполняет поиск только указанного вами имени. Например, если указать **MessageBox**, платформа пытается найти **MessageBox** и завершается сбоем, если она не может найти точное написание.

  `ExactSpelling` Когда поле имеет значение `false`, как это задано по умолчанию в C++ и C#, платформа сначала ищет немодифицированный псевдоним (**MessageBox**), а

затем модифицированное имя (**MessageBoxA**), если немодифицированный псевдоним не найден. Обратите внимание, что поведение сопоставления имен ANSI отличается от поведения сопоставления имен Юникода.

### Unicode

- Маршалирование строк

  Вызов платформы копирует строки из управляемого формата (Юникод) в формат Юникода.

- Сопоставление имен

  Когда поле `ExactSpelling` задано как `true`, что происходит по умолчанию в Visual Basic, платформа выполняет поиск только указанного вами имени. Например, если указать **MessageBox**, платформа вызывает функцию **MessageBox** и выдаёт ошибку, если не может найти точное написание.

  `ExactSpelling` Когда это поле установлено в `false` по умолчанию в C++ и C#, при платформенном вызове сначала ищется изменённое имя (**MessageBoxW**), а затем, если изменённое имя не найдено, неизменённый псевдоним (**MessageBox**). Обратите внимание, что поведение сопоставления имен Юникода отличается от поведения сопоставления имен ANSI.

### Auto

- Вызов платформы выбирает между форматами ANSI и Юникод во время выполнения в зависимости от целевой платформы.

## Указание набора символов в Visual Basic

Вы можете указать поведение набора символов в Visual Basic, добавив ключевое слово `Ansi`, `Unicode` или `Auto` в инструкцию объявления. Если опустить ключевое слово набора символов, DllImportAttribute.CharSet поле по умолчанию используется для набора символов ANSI.

В следующем примере функция **MessageBox** объявляется три раза, каждый раз с различным поведением набора символов. Первое утверждение пропускает ключевое слово набора символов, поэтому набор символов устанавливается по умолчанию как ANSI. Второй и третий операторы явно указывают набор символов с помощью ключевого слова.

```
VB
```

```
Friend Class NativeMethods
    Friend Declare Function MessageBoxA Lib "user32.dll" (
        ByVal hWnd As IntPtr,
        ByVal lpText As String,
        ByVal lpCaption As String,
        ByVal uType As UInteger) As Integer

    Friend Declare Unicode Function MessageBoxW Lib "user32.dll" (
        ByVal hWnd As IntPtr,
        ByVal lpText As String,
        ByVal lpCaption As String,
        ByVal uType As UInteger) As Integer

    Friend Declare Auto Function MessageBox Lib "user32.dll" (
        ByVal hWnd As IntPtr,
        ByVal lpText As String,
        ByVal lpCaption As String,
        ByVal uType As UInteger) As Integer
End Class
```

## Указание набора символов в C# и C++

Поле DllImportAttribute.CharSet определяет базовый набор символов как ANSI или Юникод. Набор символов определяет, как следует маршалировать строковые аргументы метода. Используйте одну из следующих форм, чтобы указать набор символов:

C#

```
[DllImport("DllName", CharSet = CharSet.Ansi)]
[DllImport("DllName", CharSet = CharSet.Unicode)]
[DllImport("DllName", CharSet = CharSet.Auto)]
```

В следующем примере показаны три управляемых определения функции **MessageBox**, атрибутирующихся для указания набора символов. В первом определении, в случае его отсутствия, поле по умолчанию использует набор символов ANSI.

C#

```
using System;
using System.Runtime.InteropServices;

internal static class NativeMethods
{
    [DllImport("user32.dll")]
    internal static extern int MessageBoxA(
        IntPtr hWnd, string lpText, string lpCaption, uint uType);

    [DllImport("user32.dll", CharSet = CharSet.Unicode)]
```

```
    internal static extern int MessageBoxW(
        IntPtr hWnd, string lpText, string lpCaption, uint uType);

    [DllImport("user32.dll", CharSet = CharSet.Auto)]
    internal static extern int MessageBox(
        IntPtr hWnd, string lpText, string lpCaption, uint uType);
}
```

## См. также

- DllImportAttribute
- Создание прототипов в управляемом коде
- Примеры вызова платформы
- Маршалинг данных с использованием Platform Invoke

# Примеры вызова функций платформы

В следующих примерах показано, как определить и вызвать `MessageBox` функцию в User32.dll, передав простую строку в качестве аргумента. В примерах поле DllImportAttribute.CharSet установлено на `Auto`, чтобы целевая платформа могла определить ширину символов и корректно обрабатывать строки.

C#

```
using System;
using System.Runtime.InteropServices;

public class Win32 {
     [DllImport("user32.dll", CharSet=CharSet.Auto)]
     public static extern IntPtr MessageBox(int hWnd, String text,
                     String caption, uint type);
}

public class HelloWorld {
    public static void Main() {
       Win32.MessageBox(0, "Hello World", "Platform Invoke Sample", 0);
    }
}
```

Дополнительные примеры см. в разделе "Маршаллирование данных с помощью вызова платформы".

## См. также

- DllImportAttribute
- Создание прототипов в управляемом коде
- Указание набора символов

---

Last updated on 02.12.2025

# Вызов функции DLL

17.06.2025

Хотя вызов неуправляемых функций DLL почти идентичен вызову другого управляемого кода, существуют различия, которые могут сделать функции DLL кажущимися запутанными вначале. В этом разделе приводятся разделы, описывающие некоторые необычные проблемы, связанные с вызовами.

Структуры, возвращаемые из вызовов платформы, должны быть типами данных, которые имеют одинаковое представление в управляемом и неуправляемом коде. Такие типы называются *блиттабл типами*, так как они не требуют преобразования (см. Блиттабл и не блиттабл типы). Чтобы вызвать функцию, которая имеет неподдерживаемую структуру в качестве возвращаемого типа, можно определить вспомогательный блиттовый тип того же размера, что и неблиттовый тип, и преобразовать данные после возврата функции.

## В этом разделе

Передача структур
Определяет вопросы передачи структур данных с предопределенной структурой.

функции обратного вызова
Предоставляет основные сведения о функциях обратного вызова.

Практическое руководство. Реализация функций обратного вызова
Описывает, как реализовать функции обратного вызова в управляемом коде.

## Связанные разделы

Использование Неуправляемых Функций DLL
Описывает, как вызывать неуправляемые функции DLL с помощью вызова платформы.

Маршалинг данных с использованием Platform Invoke
Описывает, как объявлять параметры метода и передавать аргументы функциям, экспортируемым неуправляемых библиотеками.

# Передача структур

Многие неуправляемые функции ожидают, что в качестве параметров будут переданы члены структур (определяемые пользователем типы в Visual Basic) или члены классов, которые определены в управляемом коде. При передаче структур или классов в неуправляемый код с использованием платформы вызова (platform invoke), необходимо предоставить дополнительную информацию для обеспечения сохранения исходного расположения и выравнивания. В этом разделе описывается атрибут StructLayoutAttribute, который используется для определения форматированных типов. Для управляемых структур и классов можно выбрать несколько прогнозируемых видов поведения расположения, предоставляемых перечислением `LayoutKind`.

Представленные в этом разделе понятия приводятся с учетом важного различия между структурами и типами классов. Структуры представляют собой типы значений, а классы — ссылочные типы. В классах всегда реализуется как минимум один уровень косвенного обращения к памяти (указатель на значение). Это отличие важно, поскольку неуправляемые функции часто используют непрямое обращение, как видно из сигнатур в первом столбце следующей таблицы. Управляемые структуры и объявления классов в оставшихся столбцах показывают степень, до которой можно настроить уровень косвенности в объявлении. Объявления предоставляются как для Visual Basic, так и для Visual C#.

Развернуть таблицу

| **Неуправляемая сигнатура** | **Управляемое объявление: без косвенности**<br>`Structure MyType`<br>`struct MyType;` | **Управляемое объявление: один уровень индирекции**<br>`Class MyType`<br>`class MyType;` |
|---|---|---|
| `DoWork(MyType x);`<br><br>Требуется ноль уровней косвенного обращения. | `DoWork(ByVal x As MyType)`<br>`DoWork(MyType x)`<br><br>Добавляет ноль уровней опосредованности. | Невозможно, поскольку один уровень косвенного обращения уже существует. |
| `DoWork(MyType* x);`<br><br>Требуется один уровень косвенного обращения. | `DoWork(ByRef x As MyType)`<br>`DoWork(ref MyType x)`<br><br>Добавляет один уровень косвенности. | `DoWork(ByVal x As MyType)`<br>`DoWork(MyType x)`<br><br>Добавляет ноль уровней опосредованности. |
| `DoWork(MyType** x);`<br><br>Требуются два уровня косвенного обращения. | Невозможно, так как `ByRef ByRef` или `ref ref` не может использоваться. | `DoWork(ByRef x As MyType)`<br>`DoWork(ref MyType x)` |

| Неуправляемая сигнатура | Управляемое объявление: без косвенности<br>`Structure MyType`<br>`struct MyType;` | Управляемое объявление: один уровень индирекции<br>`Class MyType`<br>`class MyType;` |
|---|---|---|
| | | Добавляет один уровень косвенности. |

Таблица описывает следующие рекомендации по объявлениям вызовов платформы:

- Если неуправляемая функция не требует косвенного обращения, используйте структуру, передаваемую по значению.
- Если неуправляемая функция требует один уровень косвенного обращения, используйте передаваемую по ссылке структуру или передаваемый по значению класс.
- Если неуправляемая функция требует два уровня косвенного обращения, используйте класс, передаваемый по ссылке.

# Объявление и передача структур

В следующем примере показано, как определить `Point` и `Rect` структуру в управляемом коде и передать типы в качестве параметра `PtInRect` функции в файле User32.dll. `PtInRect` имеет следующую неуправляемую подпись:

C++

```
BOOL PtInRect(const RECT *lprc, POINT pt);
```

Обратите внимание, что структуру Rect необходимо передавать по ссылке, поскольку функция принимает указатель на тип RECT.

C#

```
using System.Runtime.InteropServices;

[StructLayout(LayoutKind.Sequential)]
public struct Point {
    public int x;
    public int y;
}

[StructLayout(LayoutKind.Explicit)]
public struct Rect {
    [FieldOffset(0)] public int left;
    [FieldOffset(4)] public int top;
```

```
    [FieldOffset(8)] public int right;
    [FieldOffset(12)] public int bottom;
}

internal static class NativeMethods
{
    [DllImport("User32.dll")]
    internal static extern bool PtInRect(ref Rect r, Point p);
}
```

# Объявление и передача классов

Члены класса можно передавать в неуправляемую функцию DLL, если класс имеет фиксированное расположение членов. В следующем примере показано, как передать члены `MySystemTime` класса, определенные в последовательном порядке, `GetSystemTime` в файл User32.dll. `GetSystemTime` имеет следующую неуправляемую подпись:

C++

```
void GetSystemTime(SYSTEMTIME* SystemTime);
```

В отличие от типов значений, классы всегда имеют как минимум один уровень косвенного обращения.

C#

```
[StructLayout(LayoutKind.Sequential)]
public class MySystemTime {
    public ushort wYear;
    public ushort wMonth;
    public ushort wDayOfWeek;
    public ushort wDay;
    public ushort wHour;
    public ushort wMinute;
    public ushort wSecond;
    public ushort wMilliseconds;
}
internal static class NativeMethods
{
    [DllImport("Kernel32.dll")]
    internal static extern void GetSystemTime(MySystemTime st);

    [DllImport("user32.dll", CharSet = CharSet.Auto)]
    internal static extern int MessageBox(
        IntPtr hWnd, string lpText, string lpCaption, uint uType);
}

public class TestPlatformInvoke
{
```

```
    public static void Main()
    {
        MySystemTime sysTime = new MySystemTime();
        NativeMethods.GetSystemTime(sysTime);

        string dt;
        dt = "System time is: \n" +
              "Year: " + sysTime.wYear + "\n" +
              "Month: " + sysTime.wMonth + "\n" +
              "DayOfWeek: " + sysTime.wDayOfWeek + "\n" +
              "Day: " + sysTime.wDay;
        NativeMethods.MessageBox(IntPtr.Zero, dt, "Platform Invoke Sample", 0);
    }
}
```

## См. также

- Вызов функции DLL
- StructLayoutAttribute
- FieldOffsetAttribute

---

Last updated on 02.12.2025

# Функции обратного вызова

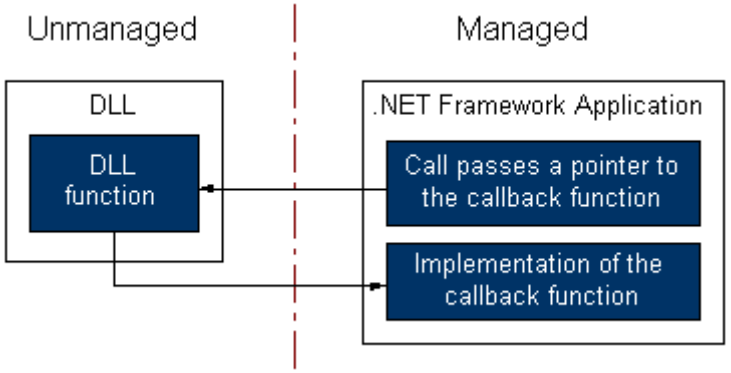
17.06.2025

Функция обратного вызова — это код в управляемом приложении, которое помогает неуправляемой функции DLL завершить задачу. Вызовы функции обратного вызова передаются косвенно из управляемого приложения, через функцию DLL и обратно в управляемую реализацию. Для правильного выполнения некоторых функций DLL, вызываемых с помощью платформы, требуется функция обратного вызова в управляемом коде.

Чтобы вызвать большинство функций DLL из управляемого кода, создайте управляемое определение функции, а затем вызовите ее. Процесс прост.

Использование функции DLL, требующей функции обратного вызова, имеет некоторые дополнительные шаги. Во-первых, необходимо определить, требуется ли функция обратного вызова, просмотрев документацию для функции. Затем необходимо создать функцию обратного вызова в управляемом приложении. Наконец, вы вызываете функцию DLL, передав указатель на функцию обратного вызова в качестве аргумента.

Следующая иллюстрация подытоживает функцию обратного вызова и шаги реализации.




Функции обратного вызова идеально подходят для использования в ситуациях, когда задача выполняется многократно. Другое распространенное использование заключается в функциях перечисления, таких как **EnumFontFamilies**, **EnumPrinters** и **EnumWindows** в API Windows. Функция **EnumWindows** перечисляет все существующие окна на компьютере, вызывая функцию обратного вызова для выполнения задачи в каждом окне. Инструкции и пример см. в статье "Практическое руководство. Реализация функций обратного вызова".

## См. также

- Практическое руководство. Реализация функций обратного вызова
- Вызов функции DLL

# Как реализовать функции обратного вызова

В следующей процедуре и примере показано, как управляемое приложение с помощью вызова платформы может распечатать значение дескриптора для каждого окна на локальном компьютере. В частности, процедура и пример используют функцию `EnumWindows` для перебора поэтапно списка окон и управляемую функцию обратного вызова (с именем CallBack) для печати значения дескриптора окна.

## Реализация функции обратного вызова

1. Ознакомьтесь с сигнатурой для `EnumWindows` функции, прежде чем продолжить реализацию. `EnumWindows` имеет следующую подпись:

   ```
   C++
   BOOL EnumWindows(WNDENUMPROC lpEnumFunc, LPARAM lParam)
   ```

   Одним из подсказок, что для этой функции требуется обратный вызов, является наличие аргумента `lpEnumFunc` . Часто можно увидеть префикс `lp` (длинный указатель) в сочетании с суффиксом `Func` в именах аргументов, которые принимают указатель на функцию обратного вызова. Документация по функциям Win32 см. в пакете SDK для Платформы Майкрософт.

2. Создайте управляемую функцию обратного вызова. В примере объявляется тип делегата `CallBack`, который принимает два аргумента (**hwnd** и **lparam**). Первый аргумент — это дескриптор окна; второй аргумент определяется самим приложением. В этом выпуске оба аргумента должны быть целыми числами.

   Функции обратного вызова обычно возвращают ненулевое значение, чтобы указать успешность и ноль, чтобы указать ошибку. Этот пример явно задает возвращаемое значение `true`, чтобы продолжить перечисление.

3. Создайте делегат и передайте его как аргумент в функцию `EnumWindows`. Вызов платформы преобразует делегат в знакомый формат обратного вызова автоматически.

4. Убедитесь, что сборщик мусора не удаляет делегат до того, как функция обратного вызова завершит свою работу. При передаче делегата в качестве параметра или передачи делегата, содержащегося в виде поля в структуре, он остается незаборным в течение длительности вызова. Таким образом, как показано в следующем примере

перечисления, функция обратного вызова завершает свою работу до возврата вызова и не требует дополнительных действий управляемого вызывающего объекта.

Однако если функция обратного вызова может вызываться после возврата вызова, управляемый вызывающий объект должен выполнить шаги, чтобы обеспечить, чтобы делегат не был завершен до завершения функции обратного вызова. Пример см. в примере GCHandle.

## Example

C#

```
using System;
using System.Runtime.InteropServices;

public delegate bool CallBack(int hwnd, int lParam);

public class EnumReportApp
{
    [DllImport("user32")]
    public static extern int EnumWindows(CallBack x, int y);

    public static void Main()
    {
        CallBack myCallBack = new CallBack(EnumReportApp.Report);
        EnumWindows(myCallBack, 0);
    }

    public static bool Report(int hwnd, int lParam)
    {
        Console.Write("Window handle is ");
        Console.WriteLine(hwnd);
        return true;
    }
}
```

## См. также

- Функции обратного вызова
- Вызов функции DLL

---

Last updated on 02.12.2025

# Маршалинг для обеспечения взаимодействия

17.06.2025

Маршалирование интероперабельности определяет, как данные передаются в аргументах методов и возвращаемых значениях между управляемой и неуправляемой памятью во время вызовов функций. Маршалирование взаимодействия — это выполняемое во время выполнения действие службы маршалирования общего языка выполнения.

Большинство типов данных имеют общие представления как в управляемой, так и неуправляемой памяти. Маршаллировщик взаимодействия обрабатывает эти типы. Другие типы могут быть неоднозначными или не представлены вообще в управляемой памяти.

Неоднозначный тип может иметь несколько неуправляемых представлений, которые сопоставляются с одним управляемым типом или отсутствуют сведения о типе, например размер массива. Для неоднозначных типов маршаллизатор предоставляет представление по умолчанию и альтернативные, когда существует несколько вариантов. Вы можете предоставить маршалеру явные инструкции о том, как маршализовать двусмысленный тип.

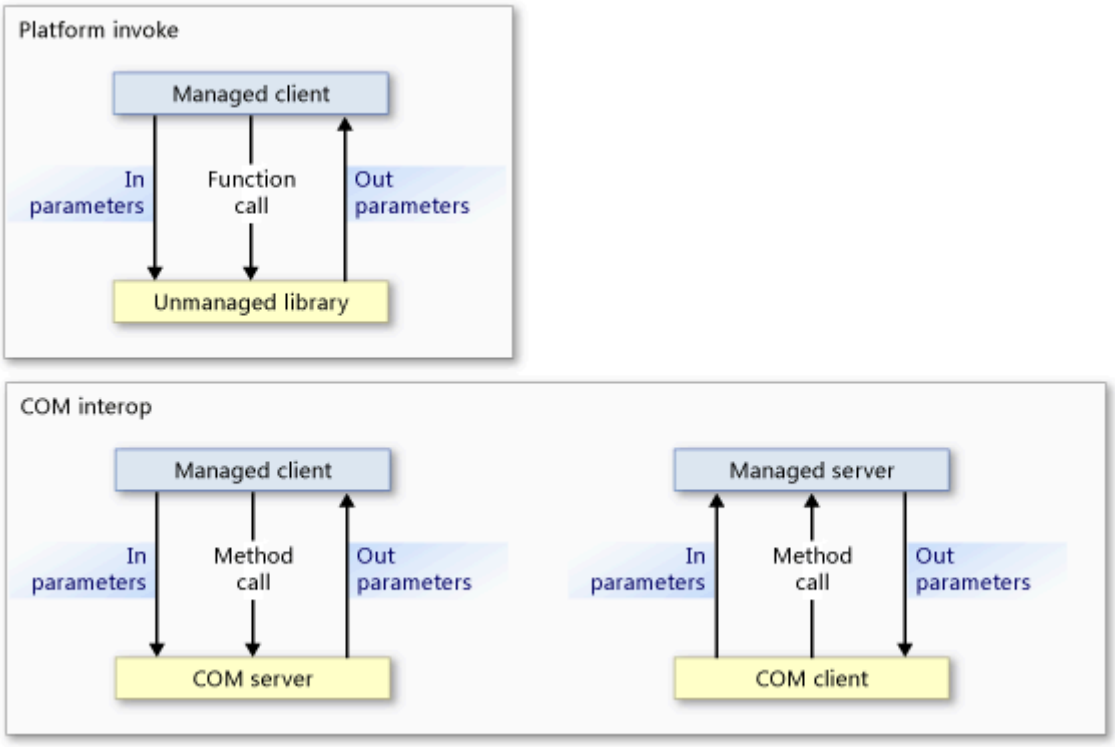
## Модели вызова платформы и COM-межоперабельности

Среда CLR предоставляет два механизма взаимодействия с неуправляемыми кодами:

- Вызов платформы, который позволяет управляемому коду вызывать функции, экспортированные из неуправляемой библиотеки.
- COM-взаимодействие, позволяющее управляемому коду взаимодействовать с объектами объектной модели компонентов (COM) через интерфейсы.

Вызовы платформы и COM-взаимодействия используют маршалинг для точного перемещения аргументов методов между вызывающим и вызываемым и обратно, если это необходимо. Как показано на следующем рисунке, вызов метода платформы выполняется из управляемого кода в неуправляемый и никогда наоборот, за исключением случаев, когда используются функции обратного вызова. Несмотря на то, что вызовы платформы могут передаваться только из управляемого в неуправляемый код, данные могут передаваться в обоих направлениях как входные или выходные

параметры. Вызовы методов взаимодействия COM могут выполняться в любом направлении.




На самом низком уровне оба механизма используют одну и ту же службу маршалинга для взаимодействия; однако некоторые типы данных поддерживаются исключительно межпроцессорным взаимодействием COM или вызовом платформы. Дополнительные сведения см. в разделе Поведение маршаллинга по умолчанию.

# Маршаллинг и com-квартиры

Маршализатор взаимодействия маршализирует данные между кучей среды CLR и неуправляемой кучей. Маршаллирование происходит в каждом случае, когда вызывающий и вызываемый не могут работать с тем же экземпляром данных. Маршализатор взаимодействия позволяет вызывающему и вызванному казаться, что они работают с одними и теми же данными, даже если у каждого есть своя копия данных.

COM также имеет компонент маршалирования, который передаёт данные между апартаментами COM или различными процессами COM. При вызове между управляемым и неуправляемым кодом в рамках одного компонентного объектного модельного апартамента (COM) маршализатор взаимодействия является единственным вовлечённым маршализатором. При вызове между управляемым и неуправляемым кодом в другой COM-квартире или другом процессе участвуют интероп-маршализатор и COM-маршализатор.

## COM-клиенты и управляемые серверы

Экспортируемый управляемый сервер с библиотекой типов, зарегистрированной с помощью Regasm.exe (средства регистрации сборок), имеет запись реестра `ThreadingModel` установленную в `Both`. Это значение указывает, что сервер можно активировать в однопоточной квартире (STA) или многопоточной квартире (MTA). Объект сервера создается в той же квартире, что и вызывающий объект, как показано в следующей таблице:

Развернуть таблицу

| COM-клиент | Сервер .NET | Требования маршаллинга |
|---|---|---|
| STA | `Both` становится STA. | Маршаллинг с одной квартирой. |
| MTA | `Both` становится MTA. | Маршаллинг с одной квартирой. |

Так как клиент и сервер находятся в одной среде, служба управления данными взаимодействия автоматически обрабатывает все данные. На следующем рисунке показана служба маршаллинга взаимодействия, работающая между управляемыми и неуправляемыми кучами в одной квартире в стиле COM.


Если вы планируете экспортировать управляемый сервер, помните, что COM-клиент определяет квартиру сервера. Управляемый сервер, вызываемый com-клиентом, инициализированным в MTA, должен обеспечить безопасность потоков.

## Управляемые клиенты и com-серверы

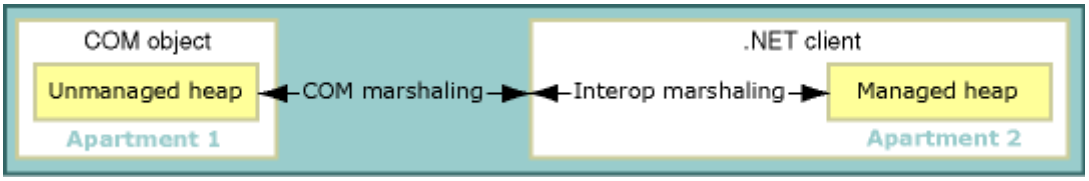
Значением по умолчанию для управляемых клиентских квартир является MTA; Однако тип приложения клиента .NET может изменить параметр по умолчанию. Например, параметром квартиры клиента Visual Basic является STA. Можно использовать свойство System.STAThreadAttribute, свойство System.MTAThreadAttribute, свойство Thread.ApartmentState или свойство Page.AspCompatMode для проверки и изменения параметров помещения управляемого клиента.

Автор компонента задает привязку потоков COM-сервера. В следующей таблице показаны сочетания параметров размещения для клиентов .NET и COM-серверов. В нем также показаны полученные требования к маршалингу для сочетаний.

Развернуть таблицу

| Клиент .NET | COM-сервер | Требования маршаллинга |
|---|---|---|
| MTA (по умолчанию) | MTA | Маршализация интероперабельности. |
| | STA | Взаимодействие и маршалинг COM. |
| STA | MTA | Взаимодействие и маршалинг COM. |
| | STA | Маршализация интероперабельности. |

Когда управляемый клиент и неуправляемый сервер находятся в одной квартире, служба маршаллинга взаимодействия обрабатывает все данные маршалинга. Однако при инициализации клиента и сервера в разных квартирах также требуется маршализация COM. На следующем рисунке показаны элементы межквартирного вызова:




Для маршаллинга между квартирами можно выполнить следующие действия:

- Примите накладные расходы на передачу данных между компонентами, что заметно только когда выполняется множество вызовов через границу. Необходимо зарегистрировать библиотеку типов компонента COM для успешного пересечения границы квартиры.
- Измените основной поток, установив клиентский поток на STA или MTA. Например, если клиент C# вызывает множество COM-компонентов STA, можно избежать межпотокового маршаллинга, установив основной поток в режим STA.

  > ⓘ **Примечание**
  >
  > После установки потока клиента C# вызовы к COM-компонентам MTA потребуют маршаллинга между контекстами.

Инструкции по явному выбору модели многопоточности см. подробнее в разделе "Управляемое и неуправляемое программирование потоков".

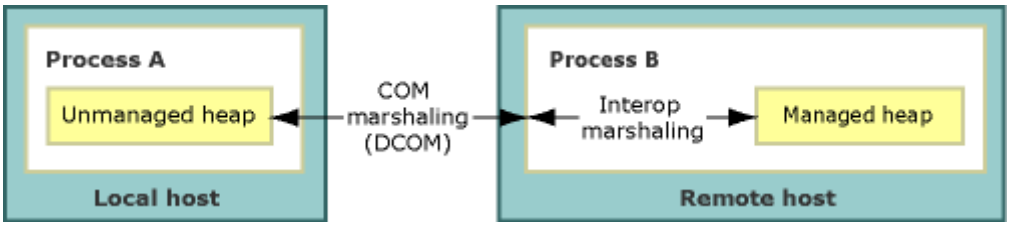
# Маршаллирование удаленных вызовов

Как и при маршаллинге между процессами, маршалирование COM вовлечено в каждый вызов между управляемым и неуправляемым кодом всякий раз, когда объекты находятся

в отдельных процессах. Рассмотрим пример.

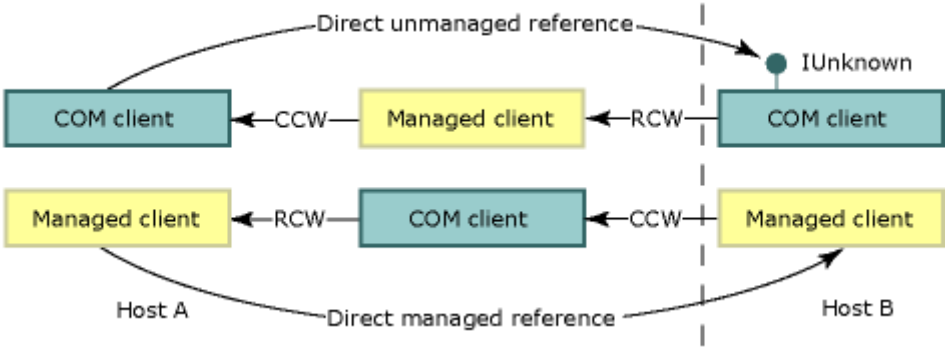
- COM-клиент, вызывающий управляемый сервер на удаленном узле, использует распределенный COM (DCOM).
- Управляемый клиент, вызывающий COM-сервер на удаленном узле, использует DCOM.

На следующем рисунке показано, как маршалирование интероп и маршалирование COM обеспечивают каналы связи между процессами и границами хостов.




## Сохранение удостоверения

Общая среда выполнения .NET сохраняет идентичность управляемых и неуправляемых ссылок. На следующем рисунке показан поток прямых неуправляемых ссылок (верхняя строка) и прямых управляемых ссылок (нижняя строка) между процессами и границами узла.




На этой иллюстрации:

- Неуправляемый клиент получает ссылку на COM-объект из управляемого объекта, который получает эту ссылку от удаленного узла. Механизм удаленного взаимодействия — DCOM.

- Управляемый клиент получает ссылку на управляемый объект из COM-объекта, который получает эту ссылку от удаленного узла. Механизм удаленного взаимодействия — DCOM.

> ⓘ **Примечание**

Экспортируемая библиотека типов управляемого сервера должна быть зарегистрирована.

Число границ процесса между вызывающим и вызываемым не имеет значения; то же самое прямое обращение происходит для внутрипроцессных и межпроцессных вызовов.

## Управляемое удаленное взаимодействие

Среда выполнения также предоставляет управляемое удаленное взаимодействие, которое можно использовать для установления канала связи между управляемыми объектами в пределах процесса и границ узлов. Управляемое удаленное взаимодействие может разместить брандмауэр между компонентами обмена данными, как показано на следующем рисунке:

Удаленные вызовы между брандмауэрами с использованием SOAP или класса TcpChannel

Некоторые неуправляемые вызовы можно направлять через SOAP, например вызовы между обслуживаемых компонентов и COM.

# Связанные разделы

Развернуть таблицу

| Название | Описание |
| --- | --- |
| Поведение маршаллинга по умолчанию | Описывает правила, которые служба межоперационного маршаллинга использует для обработки данных. |
| Маршалинг данных с использованием Platform Invoke | Описывает, как объявлять параметры метода и передавать аргументы функциям, экспортируемым неуправляемых библиотеками. |
| Маршаллирование данных с использованием COM-Interop | Описывает, как настроить COM-оболочки для изменения поведения маршаллинга. |
| Как это сделать: Перенос Managed-Code DCOM на WCF | Описание миграции из DCOM в WCF. |
| Практическое руководство. Сопоставление HRESULTs и исключений | Описывает, как сопоставлять пользовательские исключения с HRESULT и предоставляет полное сопоставление каждого |

| Название | Описание |
| --- | --- |
| | HRESULT с соответствующим классом исключений в .NET Framework. |
| Взаимодействие с использованием универсальных типов | Описывает, какие действия поддерживаются при использовании универсальных типов для взаимодействия COM. |
| Взаимодействие с неуправляемым кодом | Описывает службы взаимодействия, предоставляемые средой CLR. |
| Расширенное взаимодействие COM | Содержит ссылки на дополнительные сведения о включении COM-компонентов в приложение .NET Framework. |
| Рекомендации по проектированию взаимодействия | Предоставляет советы по написанию интегрированных com-компонентов. |

# Справка

System.Runtime.InteropServices

# Поведение маршаллинга по умолчанию

17.06.2025

Маршалирование взаимодействия работает с правилами, которые определяют поведение данных, связанных с параметрами метода, по мере того как они передаются между управляемой и неуправляемой памятью. Эти встроенные правила управляют такими действиями маршаллинга, как преобразования типа данных, независимо от того, может ли вызывающий объект изменять данные и возвращать эти изменения вызывающему объекту, и при каких обстоятельствах маршаллизатор обеспечивает оптимизацию производительности.

В этом разделе определяются характеристики поведения по умолчанию службы маршалинга взаимодействия. В нём представлены подробные сведения о массивах, маршалировании, типах Boolean, символьных типах, делегатах, классах, объектах, строках и структурах.

> ⓘ **Примечание**
>
> Маршаллирование универсальных типов не поддерживается. Дополнительные сведения см. в разделе "Взаимодействие с использованием универсальных типов".

# Управление памятью с маршализатором взаимодействия

Маршализатор взаимодействия всегда пытается освободить память, выделенную неуправляемым кодом. Это поведение соответствует правилам управления памятью COM, но отличается от правил, регулирующих собственный C++.

Путаница может возникнуть, если вы ожидаете типичное поведение C++ (без освобождения памяти), при использовании механизма Platform Invoke, который автоматически освобождает память для указателей. Например, вызов следующего неуправляемого метода из библиотеки DLL C++ не освобождает память автоматически.

## Неуправляемая сигнатура

C++

```
BSTR MethodOne (BSTR b) {
     return b;
```

```
}
```

Однако если вы определите метод как прототип платформенного вызова, замените каждый тип **BSTR** на тип String и вызовите `MethodOne`, то общая языкосвязанная среда выполнения (CLR) попытается освободить `b` дважды. Поведение маршаллинга можно изменить, используя IntPtr типы вместо типов **String**.

Среда выполнения всегда использует метод **CoTaskMemFree** в Windows и **бесплатный** метод на других платформах для освобождения памяти. Если память, которую вы используете, не была выделена с использованием метода **CoTaskMemAlloc** в Windows или метода **malloc** на других платформах, необходимо использовать **IntPtr** и освободить ее вручную с помощью соответствующего метода. Аналогичным образом можно избежать автоматического освобождения памяти в ситуациях, когда память никогда не должна быть освобождена, например при использовании функции **GetCommandLine** из Kernel32.dll, которая возвращает указатель на память ядра. Дополнительные сведения об освобождении памяти вручную см. в примере буферов.

# Маршалирование по умолчанию для классов

Классы можно передавать только через COM-взаимодействие и всегда передаются как интерфейсы. В некоторых случаях интерфейс, который используется для преобразования класса, называется интерфейсом класса. Сведения о переопределении интерфейса класса с помощью выбранного интерфейса см. в разделе "Введение в интерфейс класса".

## Передача классов в COM

Когда управляемый класс передается в COM, маршализатор взаимодействия автоматически упаковывает класс с помощью COM-прокси и передает интерфейс класса, созданный прокси-сервером, вызову метода COM. Затем прокси-сервер делегирует все вызовы интерфейса класса обратно в управляемый объект. Прокси-сервер также предоставляет другие интерфейсы, которые не реализуются явным образом классом. Прокси-сервер автоматически реализует интерфейсы, такие как **IUnknown** и **IDispatch** от имени класса.

## Передача классов в код .NET

Coclasses обычно не используются в качестве аргументов метода в COM. Вместо этого интерфейс по умолчанию обычно передается вместо coclass.

Когда интерфейс передается в управляемый код, маршаллизатор отвечает за обертывание интерфейса соответствующей оболочкой и передачу этой оболочки в

управляемый метод. Выбор оболочки может быть сложной задачей. Каждый экземпляр COM-объекта имеет одну уникальную оболочку, вне зависимости от того, сколько интерфейсов реализует объект. Например, один COM-объект, реализующий пять отдельных интерфейсов, имеет только одну оболочку. Одна и та же оболочка обеспечивает доступ ко всем пяти интерфейсам. Если создаются два экземпляра COM-объекта, создаются два экземпляра оболочки.

Чтобы оболочка поддерживала тот же тип в течение всего времени существования, маршализатор interop должен определить правильную оболочку при первом прохождении интерфейса, предоставляемого объектом, через маршализатор. Маршаллизатор определяет объект, просматривая один из интерфейсов, который реализует объект.

Например, маршаллизатор определяет, что оболочка класса должна использоваться для упаковки интерфейса, переданного в управляемый код. Когда интерфейс впервые передается через маршрутизатор, он проверяет, поступает ли интерфейс от известного объекта. Эта проверка выполняется в двух ситуациях:

- Интерфейс реализуется другим управляемым объектом, переданным в COM в другом месте. Маршализатор может легко идентифицировать интерфейсы, предоставляемые управляемыми объектами, и сопоставлять интерфейс с управляемым объектом, который обеспечивает её реализацию. Затем управляемый объект передается методу и не требуется оболочка.

- Объект, который уже был упакован, реализует интерфейс. Чтобы определить, является ли это так, маршаллировщик запрашивает у объекта его интерфейс **IUnknown** и сравнивает возвращенный интерфейс с интерфейсами других объектов, которые уже обернуты. Если интерфейс совпадает с другим интерфейсом, объекты имеют ту же идентичность, а существующая оболочка передаётся в метод.

Если интерфейс не является из известного источника, маршаллер выполняет следующее:

1. Маршаллизатор запрашивает объект для интерфейса **IProvideClassInfo2** . Если предоставлено, маршализатор использует CLSID, возвращенный из **IProvideClassInfo2.GetGUID** для идентификации кокласса, предоставляющего интерфейс. С помощью CLSID маршализатор может найти обертку в реестре, если сборка была зарегистрирована ранее.

2. Маршаллизатор запрашивает интерфейс для интерфейса **IProvideClassInfo** . Если предоставлено, маршаллизатор использует **ITypeInfo**, возвращенный из **IProvideClassInfo.GetClassInfo**, для определения CLSID класса, предоставляющего интерфейс. Маршаллер может использовать CLSID для поиска метаданных для оболочки.

3. Если маршаллер по-прежнему не может идентифицировать класс, он оборачивает интерфейс универсальным классом оболочки с именем **System.__ComObject**.

# Маршалирование по умолчанию для делегатов

Управляемый делегат управляется как COM-интерфейс или в виде указателя на функцию в зависимости от механизма вызова.

- Для вызова платформы делегат преобразуется в неуправляемый указатель на функцию по умолчанию.
- Для взаимодействия COM делегат маршалируется как COM-интерфейс типа **_Delegate** по умолчанию. Интерфейс **_Delegate** определен в библиотеке типов Mscorlib.tlb и содержит Delegate.DynamicInvoke метод, который позволяет вызывать метод, на который ссылается делегат.

В следующей таблице показаны параметры маршалинга для типа данных управляемого делегата. Атрибут MarshalAsAttribute предоставляет различные значения перечисления UnmanagedType для маршалирования делегатов.

Развернуть таблицу

| Тип перечисления | Описание неуправляемого формата |
|---|---|
| **UnmanagedType.FunctionPtr** | Указатель неуправляемой функции. |
| **UnmanagedType.Interface** | Интерфейс типа **_Delegate**, как определено в mscorlib.tlb. |

Рассмотрим следующий пример кода, в котором методы `DelegateTestInterface` экспортируются в библиотеку типов COM. Обратите внимание, что только делегаты, помеченные ключевым словом **ref** (или **ByRef**), передаются как входные/выходные параметры.

C#

```
using System;
using System.Runtime.InteropServices;

public interface DelegateTest {
void m1(Delegate d);
void m2([MarshalAs(UnmanagedType.Interface)] Delegate d);
void m3([MarshalAs(UnmanagedType.Interface)] ref Delegate d);
void m4([MarshalAs(UnmanagedType.FunctionPtr)] Delegate d);
void m5([MarshalAs(UnmanagedType.FunctionPtr)] ref Delegate d);
}
```

## Представление библиотеки типов

C++

```
importlib("mscorlib.tlb");
interface DelegateTest : IDispatch {
[id(…)] HRESULT m1([in] _Delegate* d);
[id(…)] HRESULT m2([in] _Delegate* d);
[id(…)] HRESULT m3([in, out] _Delegate** d);
[id()] HRESULT m4([in] int d);
[id()] HRESULT m5([in, out] int *d);
   };
```

Указатель функции можно разыменовыть так же, как и любой другой неуправляемый указатель функции.

В этом примере, когда два делегата маршалируются как UnmanagedType.FunctionPtr, результатом являются `int` и указатель на `int`. Поскольку типы делегатов маршалируются, `int` здесь представляет указатель на пустоту (`void*`), которая является адресом делегата в памяти. Другими словами, этот результат относится к 32-разрядным системам Windows, так как `int` здесь представляет размер указателя функции.

> ⓘ **Примечание**
>
> Ссылка на указатель функции на управляемый делегат, который удерживается неуправляемым кодом, не мешает среде выполнения общего языка (CLR) выполнять сборку мусора для управляемого объекта.

Например, следующий код является неверным, так как ссылка на `cb` объект, переданная `SetChangeHandler` методу, не поддерживает `cb` в активном состоянии за пределами времени выполнения метода `Test`. После того как объект `cb` будет собран сборщиком мусора, переданный в `SetChangeHandler` указатель функции больше не является допустимым.

C#

```
public class ExternalAPI {
   [DllImport("External.dll")]
   public static extern void SetChangeHandler(
      [MarshalAs(UnmanagedType.FunctionPtr)]ChangeDelegate d);
}
public delegate bool ChangeDelegate([MarshalAs(UnmanagedType.LPWStr) string S);
public class CallBackClass {
   public bool OnChange(string S){ return true;}
}
```

```
internal class DelegateTest {
   public static void Test() {
      CallBackClass cb = new CallBackClass();
      // Caution: The following reference on the cb object does not keep the
      // object from being garbage collected after the Main method
      // executes.
      ExternalAPI.SetChangeHandler(new ChangeDelegate(cb.OnChange));
   }
}
```

Чтобы компенсировать непредвиденную сборку мусора, вызывающий код должен убедиться, что `cb` объект остается доступным до тех пор, пока используется неуправляемый указатель функции. При необходимости, можно настроить неуправляемый код так, чтобы уведомлять управляемый код о том, когда указатель функции больше не требуется, как показано в следующем примере.

C#

```
internal class DelegateTest {
   CallBackClass cb;
   // Called before ever using the callback function.
   public static void SetChangeHandler() {
      cb = new CallBackClass();
      ExternalAPI.SetChangeHandler(new ChangeDelegate(cb.OnChange));
   }
   // Called after using the callback function for the last time.
   public static void RemoveChangeHandler() {
      // The cb object can be collected now. The unmanaged code is
      // finished with the callback function.
      cb = null;
   }
}
```

# Маршаллинг по умолчанию для типов значений

Большинство типов значений, таких как целые числа и числа с плавающей запятой, являются блитабельными и не требуют маршаллинга. Другие неблиттабельные типы имеют разнородные представления в управляемой и неуправляемой памяти и требуют маршаллинга. Тем не менее другие типы требуют явного форматирования по границе взаимодействия.

В этом разделе содержатся сведения о следующих типах форматированных значений:

- Типы значений, используемые в вызове платформы
- Типы значений, используемые в COM-взаимодействии

Помимо описания форматированных типов, в этом разделе определяются системные типы значений, обладающие необычным поведением маршалинга.

Форматированный тип — это сложный тип, содержащий информацию, которая явно контролирует расположение его членов в памяти. Сведения о макете элемента предоставляются с помощью атрибута StructLayoutAttribute . Макет может быть одним из следующих LayoutKind значений перечисления:

- **LayoutKind.Auto**

  Указывает, что среда CLR свободна для переупорядочения элементов типа для повышения эффективности. Однако, когда значимый тип передается в неуправляемый код, макет его элементов предсказуем. Попытка маршалировать такую структуру автоматически вызывает исключение.

- **LayoutKind.Sequential**

  Указывает, что элементы типа должны быть размещены в неуправляемой памяти в том же порядке, в котором они отображаются в определении управляемого типа.

- **LayoutKind.Explicit**

  Указывает, что элементы выкладываются в соответствии с предоставленным полем FieldOffsetAttribute .

## Типы значений, используемые в вызове платформы

В следующем примере типы `Point` и `Rect` предоставляют сведения о макете элементов с помощью **StructLayoutAttribute**.

C#

```
using System.Runtime.InteropServices;
[StructLayout(LayoutKind.Sequential)]
public struct Point {
   public int x;
   public int y;
}

[StructLayout(LayoutKind.Explicit)]
public struct Rect {
   [FieldOffset(0)] public int left;
   [FieldOffset(4)] public int top;
   [FieldOffset(8)] public int right;
   [FieldOffset(12)] public int bottom;
}
```

При передаче в неуправляемый код эти форматированные типы маршаллизируются как структуры стиля C. Это обеспечивает простой способ вызова неуправляемого API с аргументами структуры. Например, структуры `POINT` и `RECT` можно передать в функцию Microsoft Windows API **PtInRect** следующим образом:

```
C++

BOOL PtInRect(const RECT *lprc, POINT pt);
```

Вы можете передавать структуры, используя следующее определение вызова платформы:

```
C#

internal static class NativeMethods
{
   [DllImport("User32.dll")]
   internal static extern bool PtInRect(ref Rect r, Point p);
}
```

Тип `Rect` значения должен передаваться по ссылке, так как неуправляемый API ожидает, что указатель на `RECT` будет передан функции. `Point` Тип значения передается по значению, так как неуправляемый API ожидает, что `POINT` будет передан в стеке. Это тонкое различие очень важно. Ссылки передаются в неуправляемый код в виде указателей. Значения передаются в неуправляемый код в стеке.

> ⓘ **Примечание**
>
> Если форматированный тип маршалируется как структура, доступны только поля внутри типа. Если тип содержит методы, свойства или события, они недоступны из неуправляемого кода.

Классы также можно маршалировать в неуправляемый код в виде структур стиля C, если они имеют фиксированное расположение членов. Сведения о макете элемента для класса также предоставляются атрибутом StructLayoutAttribute . Основное различие между типами значений с фиксированным макетом и классами с фиксированным макетом заключается в том, как они маршалируются в неуправляемый код. Типы значений передаются по значению (в стеке) и, следовательно, любые изменения, внесенные в члены типа вызываемого объекта, не отображаются вызывающим элементом. Ссылочные типы передаются по ссылке (ссылка на тип передается в стеке); следовательно, все изменения, внесенные в элементы типа blittable типа вызываемого объекта, отображаются вызывающим элементом.

> ⓘ **Примечание**
>
> Если у ссылочного типа есть члены неуправляемых типов, преобразование требуется дважды: в первый раз при передаче аргумента на неуправляемую сторону и второй раз при возврате из вызова. Из-за этой добавленной нагрузки параметры In/Out должны быть явно применены к аргументу, если вызывающий объект хочет увидеть изменения, внесенные вызываемым объектом.

В следующем примере класс `SystemTime` имеет последовательное расположение членов и может передаваться в функцию Windows API **GetSystemTime**.

C#

```
[StructLayout(LayoutKind.Sequential)]
   public class SystemTime {
   public ushort wYear;
   public ushort wMonth;
   public ushort wDayOfWeek;
   public ushort wDay;
   public ushort wHour;
   public ushort wMinute;
   public ushort wSecond;
   public ushort wMilliseconds;
}
```

Функция **GetSystemTime** определена следующим образом:

C++

```
void GetSystemTime(SYSTEMTIME* SystemTime);
```

Эквивалентное определение вызова платформы для **GetSystemTime** выглядит следующим образом:

C#

```
internal static class NativeMethods
{
   [DllImport("Kernel32.dll", CharSet = CharSet.Auto)]
   internal static extern void GetSystemTime(SystemTime st);
}
```

Обратите внимание, что `SystemTime` аргумент не вводится в качестве ссылочного аргумента, так как `SystemTime` является классом, а не типом значения. В отличие от типов

значений, классы всегда передаются по ссылке.

В следующем примере кода показан другой класс `Point`, в котором есть метод с именем `SetXY`. Поскольку тип имеет последовательную компоновку, его можно передать в неуправляемый код и маршалировать как структуру. `SetXY` Однако элемент не вызывается из неуправляемого кода, даже если объект передается по ссылке.

C#

```
[StructLayout(LayoutKind.Sequential)]
public class Point {
   int x, y;
   public void SetXY(int x, int y){
      this.x = x;
      this.y = y;
   }
}
```

## Типы значений, используемые в COM-взаимодействии

Форматированные типы также можно передавать в вызовы метода взаимодействия COM. Фактически при экспорте в библиотеку типов типы значений автоматически преобразуются в структуры. Как показано в следующем примере, `Point` тип значения становится определением типа (typedef) с именем `Point`. Все ссылки на тип значения `Point` в других частях библиотеки типов заменяются на `Point` тип.

### Представление библиотеки типов

C++

```
typedef struct tagPoint {
   int x;
   int y;
} Point;
interface _Graphics {
   …
   HRESULT SetPoint ([in] Point p)
   HRESULT SetPointRef ([in,out] Point *p)
   HRESULT GetPoint ([out,retval] Point *p)
}
```

Те же правила, которые используются для маршалирования значений и ссылок при вызове функций платформы, применяются и при маршалировании через COM-интерфейсы. Например, когда экземпляр типа значения `Point` передается из .NET Framework в COM, `Point` передается по значению. Если тип значения `Point` передается по

ссылке, то указатель на `Point` передается через стек. Маршализатор взаимодействия не поддерживает более высокие уровни индирекции (**Точка** **) в обоих направлениях.

> ⓘ **Примечание**
>
> Структуры, в которых LayoutKind значение перечисления установлено на **Explicit**, не могут использоваться в COM-взаимодействии, так как экспортируемая библиотека типов не может выразить явный макет.

## Системные типы значений

Пространство System имен имеет несколько типов значений, представляющих прямоугольную форму примитивных типов среды выполнения. Например, структура типа System.Int32 значения представляет прямоугольную форму **ELEMENT_TYPE_I4**. Вместо того чтобы маршалировать эти типы как структуры, как это делается с другими форматированными типами, вы должны маршалировать их так же, как и примитивные типы, которые они обрамляют. Поэтому **System.Int32** маршалируется как **ELEMENT_TYPE_I4**, а не как структура с одним членом типа **long**. В следующей таблице содержится список типов значений в пространстве имен **System**, которые являются коробочными представлениями примитивных типов.

Развернуть таблицу

| Тип системного значения | Тип элемента |
|---|---|
| System.Boolean | ELEMENT_TYPE_BOOLEAN |
| System.SByte | ELEMENT_TYPE_I1 |
| System.Byte | ELEMENT_TYPE_UI1 |
| System.Char | ELEMENT_TYPE_CHAR |
| System.Int16 | ELEMENT_TYPE_I2 |
| System.UInt16 | ELEMENT_TYPE_U2 |
| System.Int32 | ELEMENT_TYPE_I4 |
| System.UInt32 | ELEMENT_TYPE_U4 |
| System.Int64 | ELEMENT_TYPE_I8 |
| System.UInt64 | ELEMENT_TYPE_U8 |

| Тип системного значения | Тип элемента |
|---|---|
| System.Single | ELEMENT_TYPE_R4 |
| System.Double | ELEMENT_TYPE_R8 |
| System.String | ELEMENT_TYPE_STRING |
| System.IntPtr | ELEMENT_TYPE_I |
| System.UIntPtr | ELEMENT_TYPE_U |

Некоторые другие типы значений в пространстве имен **системы** обрабатываются иначе. Поскольку в неуправляемом коде уже имеются прочно установленные форматы для этих типов, механизм маршаллинга имеет специальные правила для их обработки. В следующей таблице перечислены специальные типы значений в пространстве имен **системы** , а также неуправляемый тип, в который они маршаллируются.

Развернуть таблицу

| Тип системного значения | Тип IDL |
|---|---|
| System.DateTime | ДАТА |
| System.Decimal | ДЕСЯТИЧНАЯ СИСТЕМА |
| System.Guid | глобальный уникальный идентификатор |
| System.Drawing.Color | OLE_COLOR |

В следующем коде показано определение неуправляемых типов **DATE**, **GUID**, **DECIMAL** и **OLE_COLOR** в библиотеке типов Stdole2.

## Представление библиотеки типов

C++

```
typedef double DATE;
typedef DWORD OLE_COLOR;

typedef struct tagDEC {
    USHORT    wReserved;
    BYTE      scale;
    BYTE      sign;
    ULONG     Hi32;
    ULONGLONG Lo64;
} DECIMAL;

typedef struct tagGUID {
```

```
    DWORD Data1;
    WORD  Data2;
    WORD  Data3;
    BYTE  Data4[ 8 ];
} GUID;
```

В следующем коде показаны соответствующие определения в управляемом `IValueTypes` интерфейсе.

C#

```
public interface IValueTypes {
   void M1(System.DateTime d);
   void M2(System.Guid d);
   void M3(System.Decimal d);
   void M4(System.Drawing.Color d);
}
```

### Представление библиотеки типов

C++

```
[…]
interface IValueTypes : IDispatch {
   HRESULT M1([in] DATE d);
   HRESULT M2([in] GUID d);
   HRESULT M3([in] DECIMAL d);
   HRESULT M4([in] OLE_COLOR d);
};
```

## См. также

- Преобразуемые и непреобразуемые типы
- Копирование и закрепление
- Маршаллинг по умолчанию для массивов
- Маршаллирование по умолчанию для объектов
- Маршаллинг по умолчанию для строк

# Блиттабельные и неблиттабельные типы

Большинство типов данных имеют одинаковое представление как в управляемой, так и в неуправляемой памяти, и не требуют специальной обработки с помощью маршализатора для взаимодействия. Эти типы называются *blittable типами* , так как они не требуют преобразования при передаче между управляемым и неуправляемым кодом.

Структуры, возвращаемые из вызовов неуправляемого кода, должны иметь непреобразуемый тип. Вызов платформы не поддерживает нелиттируемые структуры в качестве возвращаемых типов.

Следующие типы из пространства имен System относятся к перемещаемым типам.

- System.Byte
- System.SByte
- System.Int16
- System.UInt16
- System.Int32
- System.UInt32
- System.Int64
- System.UInt64
- System.IntPtr
- System.UIntPtr
- System.Single
- System.Double

Также непреобразуемыми являются следующие сложные типы:

- Одномерные массивы непреобразуемых примитивных типов, например массивы целых чисел. Тем не менее тип, содержащий переменный массив blittable-типов, сам по себе не является blittable.
- Форматированные типы значений, которые содержат только типы blittable (и классы, если они маршалируются как форматированные типы). Дополнительные сведения о типах форматированных значений см. в разделе "Маршалирование по умолчанию" для типов значений.

Ссылки на объекты не являются перерезаемыми. Кроме того, массив ссылок на объекты, которые являются перерезаемыми сами по себе, не являются перерезаемыми. Например, можно определить структуру, которая является блиттабельной, но нельзя определить блиттабельный тип, содержащий массив ссылок на эти структуры.

В качестве оптимизации массивы примитивных типов и классов, содержащих только члены с возможностью вставки, закрепляются вместо копирования во время

маршаллинга. Эти типы могут быть маршалированы как входные/выходные параметры, когда вызывающий и вызываемый находятся в одном и том же апартаменте. Однако эти типы на самом деле маршалируются как параметры типа In, и необходимо применить атрибуты InAttribute и OutAttribute, если вы хотите маршалировать аргумент в качестве параметра In/Out.

Некоторые управляемые типы данных требуют отличающегося представления в неуправляемой среде. Эти неблиттабельные типы данных должны быть преобразованы в форму, которую можно передавать. Например, управляемые строки являются неблиттируемыми типами, поскольку они должны быть преобразованы в строковые объекты, прежде чем их можно будет передаваться.

В следующей таблице перечислены некопируемые типы в пространстве имен System. Делегаты, которые являются структурами данных, ссылающимися на статический метод или экземпляр класса, также являются неблитабельными.

Развернуть таблицу

| Неблитабельный тип | Описание |
|---|---|
| System.Array | Преобразует в массив в стиле C или `SAFEARRAY`. |
| System.Boolean | Преобразует в одно-, двух- или четырехбайтовое значение, где `true` выражается как 1 или -1. |
| System.Char | Преобразует в символ Юникода или ANSI. |
| System.Class | Преобразует в интерфейс класса. |
| System.Object | Преобразует в вариант или интерфейс. |
| System.String | Преобразует в строку, завершающуюся ссылкой NULL, или в BSTR. |
| System.ValueType | Преобразует в структуру с фиксированным расположением в памяти. |
| T[] | Преобразует в массив в стиле C или `SAFEARRAY`. |

Типы классов и объектов поддерживаются только COM-взаимодействием.

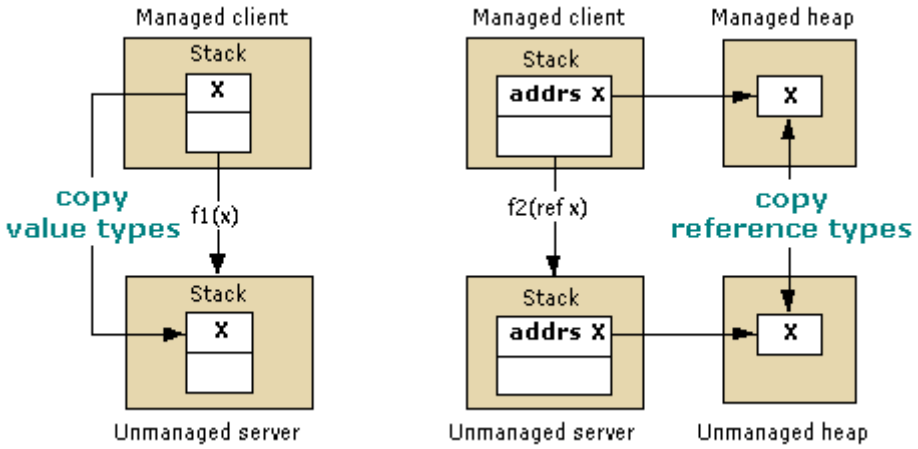
# См. также

- Поведение маршаллинга по умолчанию

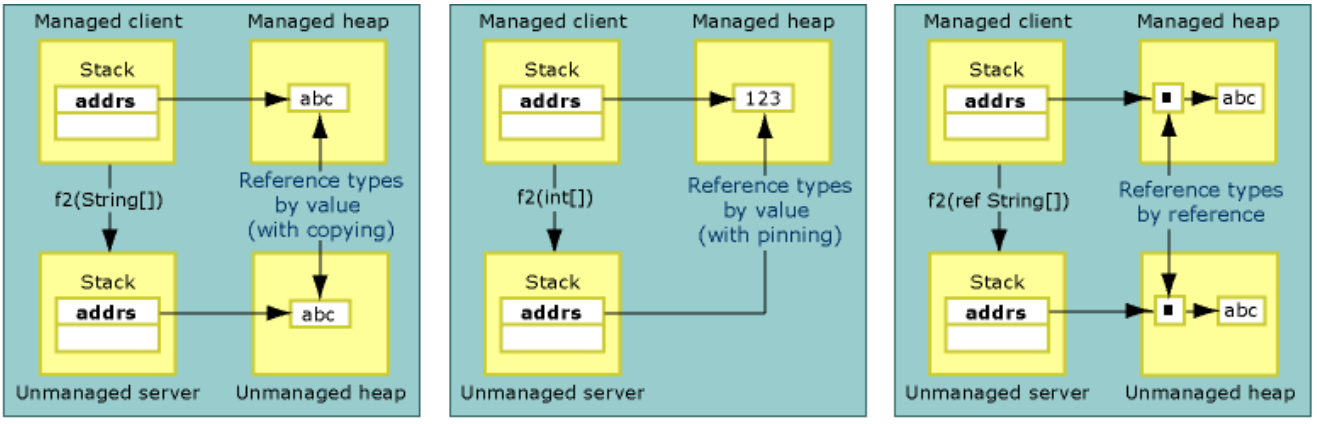
Last updated on 24.02.2026

# Копирование и закрепление

При маршалинге данных маршализатор взаимодействия может копировать или закреплять данные, которые маршалируются. Копирование данных помещает копию данных из одного расположения памяти в другое расположение памяти. На следующем рисунке показаны различия между копированием типа значения и копированием типа, передаваемого по ссылке из управляемой в неуправляемую память.




Аргументы метода, передаваемые по значению, маршалируются в неуправляемый код в виде значений в стеке. Процесс копирования прямой. Аргументы, передаваемые по ссылке, передаются как указатели в стеке. Ссылочные типы также передаются по значению и по ссылке. Как показано на следующем рисунке, ссылочные типы, передаваемые по значению, либо копируются, либо закрепляются.




Фиксация временно блокирует данные в их текущем расположении памяти, предотвращая их перемещение сборщиком мусора общеязыковой среды выполнения. Маршаллизатор закрепляет данные, чтобы сократить затраты на копирование и повысить производительность. Тип данных определяет, копируются ли данные или закрепляются в процессе маршаллинга. Закрепление выполняется автоматически в процессе маршалинга для таких объектов, как String. Однако можно также вручную закрепить память с помощью GCHandle класса. Примеры ручного закрепления с помощью `GCHandle` см. в

статье Сохранение управляемых объектов в лучших практиках взаимодействия с собственными библиотеками.

## Форматированные Blittable классы

Форматированные классы blittable имеют фиксированный макет (форматированный) и общее представление данных как в управляемой, так и неуправляемой памяти. Если эти типы требуют маршаллинга, указатель на объект в куче передается вызываемой функции напрямую. Вызываемый объект может изменить содержимое адреса памяти, на которое ссылается указатель.

> ⓘ **Примечание.**
>
> Вызывающий объект может изменить содержимое памяти, если параметр помечен как Out или In/Out. В отличие от этого, вызывающий объект должен избегать изменения содержимого, если параметру присвоено значение маршалировать как "B", что является значением по умолчанию для форматированных типов blittable. Изменение объекта In приводит к проблемам, когда этот же класс экспортируется в библиотеку типов и используется для межконтекстных вызовов.

## Форматированные классы, отличные от Blittable

Форматированные классы, отличные от таблицы , имеют фиксированный макет (форматированный), но представление данных отличается в управляемой и неуправляемой памяти. Для данных может потребоваться преобразование в следующих условиях:

- Если класс, не допускающий переключения, маршалируется по значению, вызывающий получает указатель на копию структуры данных.
- Если класс, не допускающий переключения, маршалируется по ссылке, вызывающий получает указатель на указатель на копию структуры данных.
- Если атрибут InAttribute задан, эта копия всегда инициализируется с состоянием экземпляра, при необходимости выполняя маршалинг.
- OutAttribute Если свойство задано, то состояние всегда копируется обратно в свой экземпляр при возврате, а маршалинг выполняется по мере необходимости.
- Если установлены и `InAttribute`, и `OutAttribute`, обе настройки обязательны. Если один из атрибутов опущен, маршаллизатор может оптимизировать, исключив одну

из копий.

## Ссылочные типы

Ссылочные типы могут передаваться по значению или по ссылке. При передаче по значению указатель на тип передается в стеке. При передаче по ссылке указатель на указатель на тип передается на стек.

Ссылочные типы имеют следующее условное поведение:

- Если ссылочный тип передается по значению и имеет члены типов, не допускающих переключения, эти типы преобразуются дважды:
  - Когда аргумент передается неуправляемой стороне.
  - При возврате из вызова.

  Чтобы избежать ненужного копирования и преобразования, эти типы обрабатываются как входные параметры. Чтобы вызывающий абонент мог увидеть изменения, внесенные вызываемым абонентом, необходимо явно применить атрибуты `InAttribute` и `OutAttribute` к аргументу.
- Если ссылочный тип передается по значению, и он содержит только члены ситуабельных типов, он может быть закреплен во время маршаллинга и любые изменения, внесенные в члены типа вызывающим пользователем, отображаются вызывающим элементом. Примените `InAttribute` и `OutAttribute`, если хотите добиться этого поведения явно. Без этих атрибутов направленности интероперационный маршалазер не экспортирует информацию о направлении в библиотеку типов (экспортируется как In, которая является стандартной), что может привести к проблемам с маршалингом в разных квартирах COM.
- Если ссылочный тип передается по ссылке, он будет маршалирован как in/Out по умолчанию.

## System.String и System.Text.StringBuilder

Когда данные маршалируются в неуправляемый код по значению или по ссылке, маршаллер обычно копирует данные в дополнительный буфер (возможно, преобразовывая наборы символов во время копирования) и передает ссылку на буфер вызываемой стороне. Если ссылка не выделена с помощью `BSTR` **SysAllocString**, она всегда выделяется с использованием **CoTaskMemAlloc**.

В качестве оптимизации, когда String или StringBuilder маршаллируются по значению (например, строка символов Unicode), маршаллизатор передает вызываемому прямой указатель на управляемые строки во внутреннем буфере Unicode, вместо копирования их в новый буфер.

> ⊗ **Внимание**
>
> Когда строка передается по значению, адресат никогда не должен изменять ссылку, передаваемую маршаллером. Это действие может повредить управляемую кучу.

При передаче System.String по ссылке маршаллизатор копирует содержимое строки в дополнительный буфер перед вызовом. Затем он копирует содержимое буфера в новую строку при возвращении из вызова. Этот метод гарантирует, что неизменяемая управляемая строка остается неизменной.

System.Text.StringBuilder Когда объект передается по значению, маршаллизатор передает вызывающему ссылку на временную копию внутреннего буфера `StringBuilder`. Вызывающая сторона и вызываемая сторона должны договориться о размере буфера. Вызывающий отвечает за создание `StringBuilder` достаточной длины. Вызываемый объект должен принять необходимые меры предосторожности, чтобы убедиться, что буфер не переполнен. `StringBuilder` — это исключение из правила, что ссылочные типы, которые передаются по значению, передаются как параметры `In` по умолчанию. `StringBuilder` всегда передается как `In`/`Out`.

# См. также

- Поведение маршаллинга по умолчанию
- Directional Attributes (Атрибуты направления)
- Маршалинг межоперационных взаимодействий
- Лучшие практики нативной интероперабельности
- fixed оператор (справочник по C#)

ⓘ **Примечание.** Автор создал эту статью с помощью ИИ. Подробнее

---

Last updated on 20.03.2026

# Маршалирование по умолчанию для массивов

Статья • 11.04.2023

Если приложение полностью состоит из управляемого кода, общеязыковая среда выполнения (CLR) передает типы массивов в качестве параметров ввода-вывода. В отличие от этого, маршализатор взаимодействия по умолчанию передает массив в качестве параметров In.

В рамках оптимизации закрепления непреобразуемый массив может выступать в качестве параметра ввода-вывода при взаимодействии с объектами в том же подразделении. Тем не менее при последующем экспорте кода в библиотеку типов, используемую для создания учетной записи посредника между компьютерами, если эта библиотека используется для маршалинга вызовов между подразделениями, вызовы могут получать фактические характеристики параметра ввода.

Массивы имеют сложную структуру, поэтому для проведения различий между управляемыми и неуправляемыми массивами требуется больше информации, чем для других преобразуемых типов.

## Управляемые массивы

Управляемые массивы могут иметь разные типы, однако базовым для всех этих типов является класс System.Array. Класс **System.Array** имеет свойства, определяющие ранг, длину, нижнюю и верхнюю границы массива, а также методы доступа, сортировки, поиска, копирования и создания массивов.

Эти типы массивов являются динамическими и не имеют соответствующих статических типов в библиотеке базовых классов. В качестве уникального типа массива удобно рассматривать сочетание типа элементов и ранга. Соответственно, тип одномерного массива целых чисел отличается от типа одномерного массива чисел типа double. Аналогичным образом, его тип будет отличаться от типа двухмерного массива целых чисел. При сравнении типов не учитываются границы массивов.

Как показано в следующей таблице, любой экземпляр управляемого массива должен иметь заданные тип элементов, ранг и нижнюю границу.

| Тип управляемого массива | Тип элемента | Ранг | Нижняя граница | Нотация сигнатуры |
|---|---|---|---|---|
| **ELEMENT_TYPE_ARRAY** | Задается по типу. | Задается по рангу. | При необходимости задается границами. | *type*[*n,m*] |
| **ELEMENT_TYPE_CLASS** | Неизвестно | Неизвестно | Неизвестно | **System.Array** |
| **ELEMENT_TYPE_SZARRAY** | Задается по типу. | 1 | 0 | *type*[*n*] |

# Неуправляемые массивы

Неуправляемыми могут быть безопасные массивы в стиле COM или массивы в стиле C фиксированной или переменной длины. Безопасный массив — это описывающий сам себя массив, передающий тип, ранг и границы соответствующего массива данных. Массивы в стиле C — это одномерные типизированные массивы с фиксированной нижней границей, равной 0. Служба маршалинга имеет ограниченную поддержку обоих типов массивов.

# Передача параметров массива в код .NET

Массивы в стиле языка C и безопасные массивы могут передаваться в код .NET из неуправляемого кода в виде безопасных массивов или массивов в стиле C. В следующей таблице показаны значения неуправляемого типа и соответствующих импортируемых типов.

| Неуправляемый тип | Импортируемый тип |
|---|---|
| **SafeArray**(*Type*) | < ELEMENT_TYPE_SZARRAY *ConvertedType*><br><br>Ранг = 1, нижняя граница = 0. Размер известен только в том случае, если он предоставлен в управляемой сигнатуре. Безопасные массивы, которые не имеют ранга = 1 или нижней границы = 0, не могут быть маршалированы как **SZARRAY**. |
| *Tun*[] | < ELEMENT_TYPE_SZARRAY *ConvertedType*><br><br>Ранг = 1, нижняя граница = 0. Размер известен только в том случае, если он предоставлен в управляемой сигнатуре. |

## Безопасные массивы

При импорте безопасного массива из библиотеки типов в сборку .NET он преобразуется в одномерный массив известного типа (например, **int**). К элементам массива применяются те же правила преобразования типов, что и к параметрам. Например, безопасный массив типов **BSTR** преобразуется в управляемый массив строк, а безопасный массив вариантов — в управляемый массив объектов. Тип элементов **SAFEARRAY** получается из библиотеки типов и сохраняется в значении **SAFEARRAY** перечисления UnmanagedType.

Поскольку ранг и границы безопасного массива нельзя определить из библиотеки типов, предполагается, что ранг равен 1, а нижняя граница — 0. Ранг и границы должны быть определены в управляемой сигнатуре, которая создается с помощью программы импорта библиотек типов (Tlbimp.exe). Если ранг, переданный в метод во время выполнения, отличается, возникает исключение SafeArrayRankMismatchException. Если тип массива, переданный во время выполнения, отличается, возникает исключение SafeArrayTypeMismatchException. В следующем примере показано применение безопасных массивов в управляемом и неуправляемом коде.

#### Неуправляемая сигнатура

```
C++

HRESULT New1([in] SAFEARRAY( int ) ar);
HRESULT New2([in] SAFEARRAY( DATE ) ar);
HRESULT New3([in, out] SAFEARRAY( BSTR ) *ar);
```

#### Управляемая сигнатура

```
C#

void New1([MarshalAs(UnmanagedType.SafeArray, SafeArraySubType=VT_I4)] int[]
ar) ;
void New2([MarshalAs(UnmanagedType.SafeArray, SafeArraySubType=VT_DATE)]
   DateTime[] ar);
void New3([MarshalAs(UnmanagedType.SafeArray, SafeArraySubType=VT_BSTR)]
   ref String[] ar);
```

Многомерные или ненулевые безопасные массивы можно маршалировать в управляемый код, если сигнатура метода, созданная Tlbimp.exe, изменена для указания типа элемента **ELEMENT_TYPE_ARRAY** вместо **ELEMENT_TYPE_SZARRAY**. Кроме того, можно использовать параметр **/sysarray** с программой Tlbimp.exe для импорта всех массивов в качестве объектов System.Array. Если передаваемый массив заведомо является многомерным, можно изменить код MSIL, создаваемый программой Tlbimp.exe, а затем повторно скомпилировать его. Дополнительные

сведения об изменении кода MSIL см. в разделе Настройка вызываемых оболочек времени выполнения.

## Массивы в стиле C

При импорте массива в стиле C из библиотеки типов в сборку .NET массив преобразуется в **ELEMENT_TYPE_SZARRAY**.

Тип элемента массива определяется из библиотеки типов и сохраняется во время импорта. К элементам массива применяются те же правила преобразования, что и к параметрам. Например, массив типов **LPStr** преобразуется в массив типов **String**. Программа Tlbimp.exe получает тип элементов массива и применяет к параметру атрибут MarshalAsAttribute.

Ранг массива принимается равным 1. Если ранг больше 1, массив маршалируется как одномерный массив в порядке столбцов. Нижняя граница всегда равна 0.

Библиотеки типов могут содержать массивы фиксированной или переменной длины. Поскольку в библиотеках типов содержится недостаточно информации для маршалинга массивов переменной длины, программа Tlbimp.exe может импортировать из них только массивы фиксированной длины. Для массивов фиксированной длины размер импортируется из библиотеки типов и сохраняется в атрибуте **MarshalAsAttribute**, который применяется к параметру.

Библиотеки типов, содержащие массивы переменной длины, необходимо определять вручную, как показано в следующем примере.

#### Неуправляемая сигнатура

```
C++

HRESULT New1(int ar[10]);
HRESULT New2(double ar[10][20]);
HRESULT New3(LPWStr ar[10]);
```

#### Управляемая сигнатура

```
C#

void New1([MarshalAs(UnmanagedType.LPArray, SizeConst=10)] int[] ar);
void New2([MarshalAs(UnmanagedType.LPArray, SizeConst=200)] double[] ar);
void New2([MarshalAs(UnmanagedType.LPArray,
   ArraySubType=UnmanagedType.LPWStr, SizeConst=10)] String[] ar);
```

Несмотря на то, что с помощью атрибутов **size_is** или **length_is** массива в исходном коде на языке IDL можно передать сведения о размере клиенту, компилятор MIDL не передает эту информацию в библиотеку типов. Не зная размер, служба маршалинга взаимодействия не может маршалал элементов массива. Таким образом, массивы переменной длины импортируются в виде ссылочных аргументов. Пример:

**Неуправляемая сигнатура**

```
C++

HRESULT New1(int ar[]);
HRESULT New2(int ArSize, [size_is(ArSize)] double ar[]);
HRESULT New3(int ElemCnt, [length_is(ElemCnt)] LPStr ar[]);
```

**Управляемая сигнатура**

```
C#

void New1(ref int ar);
void New2(ref double ar);
void New3(ref String ar);
```

Вы можете предоставить маршалу размер массива, изменив код MSIL, созданный Tlbimp.exe, а затем перекомпилируя его. Дополнительные сведения об изменении кода MSIL см. в разделе Настройка вызываемых оболочек времени выполнения. Чтобы указать число элементов в массиве, примените тип MarshalAsAttribute к параметру массива в определении управляемого метода одним из следующих способов:

- Определите еще один параметр, который содержит число элементов в массиве. Параметры определяются по позиции, начиная с первого, который получает номер 0.

  ```
  C#

  void New(
     int ElemCnt,
     [MarshalAs(UnmanagedType.LPArray, SizeParamIndex=0)] int[] ar );
  ```

- Определите размер массива в виде константы. Пример:

  ```
  C#
  ```

```
void New(
   [MarshalAs(UnmanagedType.LPArray, SizeConst=128)] int[] ar );
```

При маршалинге массивов из неуправляемого кода в управляемый код маршалировщик проверяет атрибут **MarshalAsAttribute** , связанный с параметром , чтобы определить размер массива. Если размер массива не указан, маршалируется только один элемент.

> ⓘ **Примечание**
>
> **Атрибут MarshalAsAttribute** не влияет на маршалирование управляемых массивов в неуправляемый код. В этом направлении размер массива определяется с помощью проверки. Маршалинг подмножества управляемого массива невозможен.

Маршализатор взаимодействия использует **методы CoTaskMemAlloc** и **CoTaskMemFree** для выделения и извлечения памяти. Эти методы также необходимо использовать при выделении памяти в неуправляемом коде.

# Передача массивов в COM

Все типы управляемых массивов можно передавать в неуправляемый код из управляемого кода. В зависимости от управляемого типа и примененных к нему атрибутов, доступ к массиву осуществляется как к безопасному массиву или как к массиву в стиле C, как показано в следующей таблице.

| **Тип управляемого массива** | **Экспортируется как** |
|---|---|
| < ELEMENT_TYPE_SZARRAY *Тип*> | UnmanagedType. **SafeArray**(*type*)<br><br>**UnmanagedType.LPArray**<br><br>Тип предоставляется в сигнатуре. Ранг всегда равен 1, а нижняя граница всегда — 0. Размер всегда известен во время выполнения. |
| < ELEMENT_TYPE_ARRAY *Тип*> <*Ранга*> [<*границы*>] | **UnmanagedType.SafeArray**(*type*)<br><br>**UnmanagedType.LPArray**<br><br>Тип, ранг и границы предоставляются в сигнатуре. Размер всегда известен во время выполнения. |

| Тип управляемого массива | Экспортируется как |
| --- | --- |
| **ELEMENT_TYPE_CLASS**<System.Array> | **UT_Interface**<br><br>**UnmanagedType.SafeArray**(*type*)<br><br>Тип, ранг, границы и размер всегда известны во время выполнения. |

В OLE-автоматизации существует ограничение в отношении массивов структур, которые содержат LPSTR или LPWSTR. Поэтому поля **String** должны быть маршалированы как **UnmanagedType.BSTR**. В противном случае будет создаваться исключение.

## ELEMENT_TYPE_SZARRAY

При экспорте метода, содержащего параметр **ELEMENT_TYPE_SZARRAY** (одномерный массив), из сборки .NET в библиотеку типов параметр массива преобразуется в массив **SAFEARRAY** заданного типа. Те же правила преобразования применяются к типам элементов массива. Содержимое управляемого массива автоматически копируется из управляемой памяти в **SAFEARRAY**. Пример:

### Управляемая сигнатура

```
C#

void New(long[] ar );
void New(String[] ar );
```

### Неуправляемая сигнатура

```
C++

HRESULT New([in] SAFEARRAY( long ) ar);
HRESULT New([in] SAFEARRAY( BSTR ) ar);
```

Ранг безопасных массивов всегда равен 1, а нижняя граница —0. Размер определяется во время выполнения на основе размера передаваемого управляемого массива.

Массив также можно маршалировать как массив в стиле C с помощью атрибута MarshalAsAttribute . Пример:

### Управляемая сигнатура

```
C#

void New([MarshalAs(UnmanagedType.LPArray, SizeParamIndex=1)]
   long [] ar, int size );
void New([MarshalAs(UnmanagedType.LPArray, SizeParamIndex=1)]
   String [] ar, int size );
void New([MarshalAs(UnmanagedType.LPArray, ArraySubType=
   UnmanagedType.LPStr, SizeParamIndex=1)]
   String [] ar, int size );
```

### Неуправляемая сигнатура

```
C++

HRESULT New(long ar[]);
HRESULT New(BSTR ar[]);
HRESULT New(LPStr ar[]);
```

Хотя маршал-маршал имеет сведения о длине, необходимые для маршалинга массива, длина массива обычно передается в виде отдельного аргумента для передачи длины вызывающему объекту.

## ELEMENT_TYPE_ARRAY

При экспорте метода, содержащего параметр **ELEMENT_TYPE_ARRAY**, из сборки .NET в библиотеку типов параметр массива преобразуется в массив **SAFEARRAY** заданного типа. Содержимое управляемого массива автоматически копируется из управляемой памяти в **SAFEARRAY**. Пример:

### Управляемая сигнатура

```
C#

void New( long [,] ar );
void New( String [,] ar );
```

### Неуправляемая сигнатура

```
C++

HRESULT New([in] SAFEARRAY( long ) ar);
HRESULT New([in] SAFEARRAY( BSTR ) ar);
```

Ранг, размер и границы безопасного массива определяются во время выполнения на основе характеристик управляемого массива.

Массив также можно маршалировать как массив B стиле C путем применения атрибута MarshalAsAttribute . Пример:

### Управляемая сигнатура

```
C#

void New([MarshalAs(UnmanagedType.LPARRAY, SizeParamIndex=1)]
   long [,] ar, int size );
void New([MarshalAs(UnmanagedType.LPARRAY,
   ArraySubType= UnmanagedType.LPStr, SizeParamIndex=1)]
   String [,] ar, int size );
```

### Неуправляемая сигнатура

```
C++

HRESULT New(long ar[]);
HRESULT New(LPStr ar[]);
```

Вложенные массивы не могут быть маршалированы. Например, при экспорте с использованием программы экспорта библиотеки типов (Tlbexp.exe) следующая сигнатура приводит к возникновению ошибки.

### Управляемая сигнатура

```
C#

void New(long [][][] ar );
```

## <ELEMENT_TYPE_CLASS System.Array>

При экспорте метода, содержащего параметр System.Array, из сборки .NET в библиотеку типов параметр массива преобразуется в интерфейс **_Array**. Содержимое этого управляемого массива доступно только через методы и свойства интерфейса **_Array**. **System.Array** также можно маршалировать как **SAFEARRAY** с помощью атрибута MarshalAsAttribute . При маршале как безопасного массива элементы массива маршалируются как варианты. Пример:

### Управляемая сигнатура

C#

```
void New1( System.Array ar );
void New2( [MarshalAs(UnmanagedType.Safe array)] System.Array ar );
```

### Неуправляемая сигнатура

C++

```
HRESULT New([in] _Array *ar);
HRESULT New([in] SAFEARRAY(VARIANT) ar);
```

## Массивы внутри структур

Неуправляемые структуры могут содержать вложенные массивы. По умолчанию эти поля внедренного массива маршалируются как SAFEARRAY. В следующем примере `s1` — это вложенный массив, который выделяется непосредственно в структуре.

### Неуправляемое представление

C++

```
struct MyStruct {
    short s1[128];
}
```

Массивы можно маршалировать как UnmanagedType, что требует задания MarshalAsAttribute поля. Размер может задаваться только в виде константы. В коде ниже показаны соответствующие управляемые определения `MyStruct`.

C#

```
[StructLayout(LayoutKind.Sequential)]
public struct MyStruct {
   [MarshalAs(UnmanagedType.ByValArray, SizeConst=128)] public short[] s1;
}
```

## См. также

- Поведение маршалинга по умолчанию
- Преобразуемые и непреобразуемые типы
- Directional Attributes (Атрибуты направления)
- Копирование и закрепление

# Маршалинг по умолчанию для объектов

Статья • 08.04.2023

Параметры и поля, типизированные как System.Object, могут предоставляться в неуправляемый код в виде одного из следующих типов:

- Вариант, если объект является параметром.
- Интерфейс, если объект является полем структуры.

Только COM-взаимодействие поддерживает маршалирование для типов объектов. По умолчанию выполняется маршалинг объектов в варианты COM. Эти правила применяются только к типу **Object** и не относятся к строго типизированным объектам, производным от класса **Object**.

## Параметры маршалинга

В следующей таблице показаны параметры маршалинга для типа данных **Object** . Атрибут MarshalAsAttribute предоставляет несколько значений перечисления UnmanagedType для маршалинга объектов.

| Тип перечисления | Описание неуправляемого формата |
|---|---|
| **UnmanagedType.Struct**<br><br>(по умолчанию для параметров) | Вариант в стиле COM. |
| **UnmanagedType.Interface** | Интерфейс **IDispatch**, если возможно. В противном случае интерфейс **IUnknown**. |
| **UnmanagedType.IUnknown**<br><br>(по умолчанию для полей) | Интерфейс **IUnknown**. |
| **UnmanagedType.IDispatch** | Интерфейс **IDispatch**. |

В следующем примере показано определение управляемого интерфейса для `MarshalObject`.

```
C#
```

```
interface MarshalObject {
   void SetVariant(Object o);
   void SetVariantRef(ref Object o);
   Object GetVariant();

   void SetIDispatch ([MarshalAs(UnmanagedType.IDispatch)]Object o);
   void SetIDispatchRef([MarshalAs(UnmanagedType.IDispatch)]ref Object o);
   [MarshalAs(UnmanagedType.IDispatch)] Object GetIDispatch();
   void SetIUnknown ([MarshalAs(UnmanagedType.IUnknown)]Object o);
   void SetIUnknownRef([MarshalAs(UnmanagedType.IUnknown)]ref Object o);
   [MarshalAs(UnmanagedType.IUnknown)] Object GetIUnknown();
}
```

Следующий пример кода экспортирует интерфейс `MarshalObject` в библиотеку типов.

C++

```
interface MarshalObject {
   HRESULT SetVariant([in] VARIANT o);
   HRESULT SetVariantRef([in,out] VARIANT *o);
   HRESULT GetVariant([out,retval] VARIANT *o)
   HRESULT SetIDispatch([in] IDispatch *o);
   HRESULT SetIDispatchRef([in,out] IDispatch **o);
   HRESULT GetIDispatch([out,retval] IDispatch **o)
   HRESULT SetIUnknown([in] IUnknown *o);
   HRESULT SetIUnknownRef([in,out] IUnknown **o);
   HRESULT GetIUnknown([out,retval] IUnknown **o)
}
```

**ⓘ Примечание**

Маршализатор взаимодействия автоматически освобождает все выделенные объекты внутри варианта после вызова.

В следующем примере показан форматированный тип значения.

C#

```
public struct ObjectHolder {
   Object o1;
   [MarshalAs(UnmanagedType.IDispatch)]public Object o2;
}
```

Следующий пример кода экспортирует форматированный тип в библиотеку типов.

C++

```
struct ObjectHolder {
   VARIANT o1;
   IDispatch *o2;
}
```

# Маршалинг объекта в интерфейс

Если объект предоставляется модели COM в виде интерфейса, это будет интерфейс класса для управляемого типа Object (интерфейс **_Object**). В полученной библиотеке типов этот интерфейс типизируется как **IDispatch** (UnmanagedType) или **IUnknown** (**UnmanagedType.IUnknown**). Клиенты COM могут динамически вызывать члены управляемого класса или любые члены, реализуемые его производными классами, используя интерфейс **_Object**. Клиент также может вызывать **QueryInterface** для получения любого другого интерфейса, явно реализуемого управляемым типом.

# Маршалинг объекта в Variant

При маршале объекта в вариант внутренний тип варианта определяется во время выполнения на основе следующих правил:

- Если ссылка на объект имеет значение NULL (**Nothing** в Visual Basic), объект маршалируется в вариант типа **VT_EMPTY**.
- Если объект является экземпляром любого типа, указанного в следующей таблице, результирующий тип варианта определяется правилами, встроенными в маршаллировщик и показанными в таблице.
- Другие объекты, которым необходимо явно управлять поведением маршалинга, могут реализовать IConvertible интерфейс . В этом случае тип варианта определяется в соответствии с кодом типа, возвращаемым из метода IConvertible.GetTypeCode. В противном случае объект маршалируется как вариант типа **VT_UNKNOWN**.

## Маршалинг системных типов в variant

В следующей таблице показаны управляемые типы объектов и соответствующие им варианты модели COM. Эти типы преобразуются только в том случае, если сигнатура вызываемого метода относится к типу System.Object.

| Тип объекта | Тип варианта COM |
| --- | --- |
| Ссылка на объект NULL (**Nothing** в Visual Basic). | VT_EMPTY |
| System.DBNull | VT_NULL |
| System.Runtime.InteropServices.ErrorWrapper | VT_ERROR |
| System.Reflection.Missing | VT_ERROR с E_PARAMNOTFOUND |
| System.Runtime.InteropServices.DispatchWrapper | VT_DISPATCH |
| System.Runtime.InteropServices.UnknownWrapper | VT_UNKNOWN |
| System.Runtime.InteropServices.CurrencyWrapper | VT_CY |
| System.Boolean | VT_BOOL |
| System.SByte | VT_I1 |
| System.Byte | VT_UI1 |
| System.Int16 | VT_I2 |
| System.UInt16 | VT_UI2 |
| System.Int32 | VT_I4 |
| System.UInt32 | VT_UI4 |
| System.Int64 | VT_I8 |
| System.UInt64 | VT_UI8 |
| System.Single | VT_R4 |
| System.Double | VT_R8 |
| System.Decimal | VT_DECIMAL |
| System.DateTime | VT_DATE |
| System.String | VT_BSTR |
| System.IntPtr | VT_INT |
| System.UIntPtr | VT_UINT |
| System.Array | VT_ARRAY |

В следующем примере кода с помощью интерфейса `MarshalObject` демонстрируется передача различных типов вариантов на COM-сервер.

C#

```
MarshalObject mo = new MarshalObject();
mo.SetVariant(null);            // Marshal as variant of type VT_EMPTY.
mo.SetVariant(System.DBNull.Value); // Marshal as variant of type VT_NULL.
mo.SetVariant((int)27);          // Marshal as variant of type VT_I2.
mo.SetVariant((long)27);          // Marshal as variant of type VT_I4.
mo.SetVariant((single)27.0);   // Marshal as variant of type VT_R4.
mo.SetVariant((double)27.0);   // Marshal as variant of type VT_R8.
```

Типы COM, не имеющие соответствующих управляемых типов, можно маршалировать с помощью классов-оболочек, таких как ErrorWrapper, DispatchWrapper, UnknownWrapperи CurrencyWrapper. В следующем примере кода демонстрируется использование этих классов-оболочек для передачи различных типов вариантов на COM-сервер.

C#

```
using System.Runtime.InteropServices;
// Pass inew as a variant of type VT_UNKNOWN interface.
mo.SetVariant(new UnknownWrapper(inew));
// Pass inew as a variant of type VT_DISPATCH interface.
mo.SetVariant(new DispatchWrapper(inew));
// Pass a value as a variant of type VT_ERROR interface.
mo.SetVariant(new ErrorWrapper(0x80054002));
// Pass a value as a variant of type VT_CURRENCY interface.
mo.SetVariant(new CurrencyWrapper(new Decimal(5.25)));
```

Классы-оболочки определяются в пространстве имен System.Runtime.InteropServices.

## Маршалинг интерфейса IConvertible в Variant

Типы, отличные от перечисленных в предыдущем разделе, могут управлять тем, как они маршалируются, реализуя IConvertible интерфейс . Если объект реализует интерфейс **IConvertible**, тип варианта COM определяется во время выполнения на основе значения перечисления TypeCode, которое возвращается методом IConvertible.GetTypeCode.

В следующей таблице показаны возможные значения перечисления **TypeCode** и соответствующие им типы вариантов COM.

| Код типа | Тип варианта COM |
|---|---|
| TypeCode.Empty | VT_EMPTY |

| Код типа | Тип варианта COM |
|---|---|
| **TypeCode.Object** | **VT_UNKNOWN** |
| **TypeCode.DBNull** | **VT_NULL** |
| **TypeCode.Boolean** | **VT_BOOL** |
| **TypeCode.Char** | **VT_UI2** |
| **TypeCode.Sbyte** | **VT_I1** |
| **TypeCode.Byte** | **VT_UI1** |
| **TypeCode.Int16** | **VT_I2** |
| **TypeCode.UInt16** | **VT_UI2** |
| **TypeCode.Int32** | **VT_I4** |
| **TypeCode.UInt32** | **VT_UI4** |
| **TypeCode.Int64** | **VT_I8** |
| **TypeCode.UInt64** | **VT_UI8** |
| **TypeCode.Single** | **VT_R4** |
| **TypeCode.Double** | **VT_R8** |
| **TypeCode.Decimal** | **VT_DECIMAL** |
| **TypeCode.DateTime** | **VT_DATE** |
| **TypeCode.String** | **VT_BSTR** |
| Не поддерживается. | **VT_INT** |
| Не поддерживается. | **VT_UINT** |
| Не поддерживается. | **VT_ARRAY** |
| Не поддерживается. | **VT_RECORD** |
| Не поддерживается. | **VT_CY** |
| Не поддерживается. | **VT_VARIANT** |

Значение com-варианта определяется путем вызова интерфейса **IConvertible.To***Type* , где **To***Type* — это подпрограмма преобразования, соответствующая типу, возвращенного из **IConvertible.GetTypeCode**. Например, объект, возвращающий **TypeCode.Double** из **IConvertible.GetTypeCode,**

маршалируется как com-вариант типа **VT_R8**. Чтобы получить значение варианта (хранится в поле **dblVal** варианта COM), можно выполнить приведение к интерфейсу **IConvertible** и вызвать метод ToDouble.

## Маршалинг варианта в объект

При маршалинге варианта в объект тип, а иногда и значение маршалированного варианта определяет тип создаваемого объекта. В следующей таблице указаны каждый тип варианта и соответствующий тип объекта, который m маршалирует при передаче варианта из COM в платформа .NET Framework.

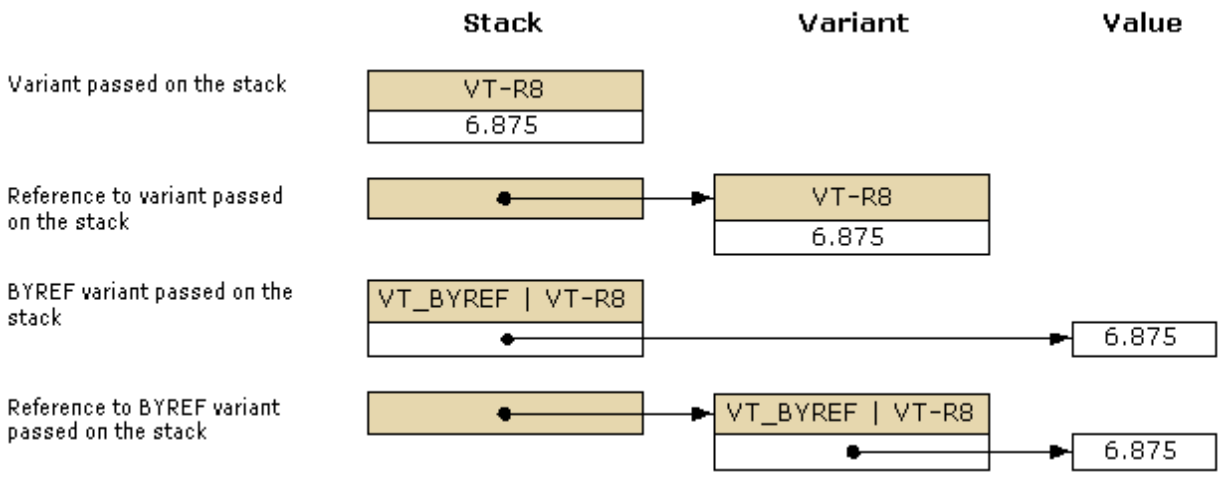
| Тип варианта COM | Тип объекта |
|---|---|
| VT_EMPTY | Ссылка на объект NULL (**Nothing** в Visual Basic). |
| VT_NULL | System.DBNull |
| VT_DISPATCH | **System.__ComObject** или значение NULL если (pdispVal == null) |
| VT_UNKNOWN | **System.__ComObject** или значение NULL если (punkVal == null) |
| VT_ERROR | System.UInt32 |
| VT_BOOL | System.Boolean |
| VT_I1 | System.SByte |
| VT_UI1 | System.Byte |
| VT_I2 | System.Int16 |
| VT_UI2 | System.UInt16 |
| VT_I4 | System.Int32 |
| VT_UI4 | System.UInt32 |
| VT_I8 | System.Int64 |
| VT_UI8 | System.UInt64 |
| VT_R4 | System.Single |
| VT_R8 | System.Double |
| VT_DECIMAL | System.Decimal |
| VT_DATE | System.DateTime |
| VT_BSTR | System.String |

| Тип варианта COM | Тип объекта |
|---|---|
| **VT_INT** | System.Int32 |
| **VT_UINT** | System.UInt32 |
| \| VT_ARRAY **VT_*** | System.Array |
| **VT_CY** | System.Decimal |
| **VT_RECORD** | Соответствующий тип упакованного значения. |
| **VT_VARIANT** | Не поддерживается. |

Типы вариантов, передаваемые из модели COM в управляемый код и обратно в COM, могут не сохранять тип варианта во время вызова. Рассмотрим, что произойдет при передаче типа варианта **VT_DISPATCH** из модели COM в .NET Framework. Во время маршалинга вариант преобразуется в System.Object. Если **объект** передается обратно в COM, он маршалируется обратно в вариант типа **VT_UNKNOWN**. Нет никакой гарантии, что вариант, создаваемый при маршале объекта из управляемого кода в COM, будет иметь тот же тип, что и вариант, изначально используемый для создания объекта.

# Маршалирование вариантов ByRef

Сами по себе варианты могут передаваться по значению или по ссылке. Несмотря на это, также можно использовать флаг **VT_BYREF** с любым типом варианта, чтобы указать, что содержимое варианта передается по ссылке, а не по значению. Разница между маршалингом вариантов по ссылке и маршалингом варианта с **VT_BYREF** набором флагов может быть запутанной. На следующем рисунке показаны различия между этими способами:




Варианты, передаваемые по значению и по ссылке

**Поведение по умолчанию для маршалинга объектов и вариантов по значению**

- При передаче объектов из управляемого кода в COM содержимое объекта копируется в новый вариант, созданный маршалером, с помощью правил, определенных в разделе Маршалинг объекта в Variant. Изменения, внесенные в вариант в неуправляемом коде, не применяются к исходному объекту после возврата из вызова.

- При передаче вариантов из COM в управляемый код содержимое варианта копируется в только что созданный объект с помощью правил, определенных в разделе Маршалирование вариантов в объект. Изменения, внесенные в объект в управляемом коде, не применяются к исходному варианту после возврата из вызова.

**Поведение по умолчанию для маршалинга объектов и вариантов по ссылке**

Для распространения изменений в вызывающем объекте параметры необходимо передавать по ссылке. Например, можно использовать ключевое слово **ref** в C# (или **ByRef** в управляемом коде Visual Basic) для передачи параметров по ссылке. В COM ссылочные параметры передаются с помощью указателя, например **variant ***.

- При передаче объекта в COM по ссылке маршаллировщик создает новый вариант и копирует содержимое ссылки на объект в вариант перед вызовом. Вариант передается в неуправляемую функцию, в которой пользователь может изменять его содержимое. После возврата из вызова изменения, внесенные в вариант в неуправляемом коде, применяются к исходному объекту. Если тип варианта отличается от типа варианта, переданного в вызов, изменения применяются к объекту другого типа. Таким образом, тип переданного в вызов объекта может отличаться от типа объекта, возвращаемого из вызова.

- При передаче варианта в управляемый код по ссылке маршализатор создает новый объект и копирует содержимое варианта в объект перед вызовом. Ссылка на объект передается в управляемую функцию, в которой пользователь может изменять сам объект. После возврата из вызова изменения, внесенные в указываемый по ссылке объект, применяются к исходному варианту. Если тип объекта отличается от типа объекта, переданного в вызов, тип исходного варианта изменяется и значение передается обратно в вариант. Аналогичным образом, тип переданного в вызов варианта может отличаться от типа варианта, возвращаемого из вызова.

**Поведение по умолчанию для маршалинга варианта с установленным флагом VT_BYREF**

- Для варианта, передаваемого в управляемый код по значению, может быть установлен флаг **VT_BYREF**, который указывает, что вариант содержит ссылку вместо значения. В этом случае вариант по-прежнему маршалируется в объект , так как вариант передается по значению. Маршаллировщик автоматически разыменовывает содержимое варианта и копирует его во вновь созданный объект перед выполнением вызова. После этого объект передается в управляемую функцию. Тем не менее при возврате из вызова объект не применяется к исходному варианту. Изменения, внесенные в управляемый объект, утрачиваются.

  > ⊗ **Внимание!**
  >
  > Изменить значение варианта, переданного по значению, нельзя, даже если для варианта установлен флаг **VT_BYREF**.

- Для варианта, передаваемого в управляемый код по ссылке, также может быть установлен флаг **VT_BYREF**, который указывает, что вариант содержит еще одну ссылку. Если это так, вариант маршалируется в объект **ссылки** , так как вариант передается по ссылке. Маршаллировщик автоматически разыменовывает содержимое варианта и копирует его во вновь созданный объект перед выполнением вызова. При возврате из вызова значение объекта возвращается по ссылке с исходным вариантом только в том случае, если тип объекта совпадает с типом переданного объекта. Таким образом, при передаче не изменяется тип варианта с установленным флагом **VT_BYREF**. Если во время вызова тип объекта изменяется, при возврате из него возникает исключение InvalidCastException.

В следующей таблице описываются общие правила распространения для вариантов и объектов.

| **Исходный тип** | **Кому** | **Возвращаемые изменения** |
|---|---|---|
| **Вариант***v* | **Объект***o* | Никогда |
| **Объект***o* | **Вариант***v* | Никогда |
| **Вариант***Pv* | **Объект ссылки***o* | Всегда |
| **Объект ссылки***o* | **Вариант***Pv* | Всегда |

| Исходный тип | Кому | Возвращаемые изменения |
|---|---|---|
| **Variant*v*(VT_BYREF\|VT_*)** | **Объект*o*** | Никогда |
| **Variant*v*(VT_BYREF\|VT_)** | **Объект ссылки*o*** | Только если тип не был изменен. |

# См. также

- Поведение маршалинга по умолчанию
- Преобразуемые и непреобразуемые типы
- Directional Attributes (Атрибуты направления)
- Копирование и закрепление

# Маршалирование по умолчанию для строк

Как классы System.String , так и System.Text.StringBuilder имеют аналогичное поведение маршаллинга.

Строки маршалируются как тип COM-стиля `BSTR` или как строка, завершающаяся значением NULL (массив символов, заканчивающийся нулевым символом). Символы внутри строки могут быть маршалированы как Юникод (по умолчанию для систем Windows) или ANSI.

## Строки, используемые в интерфейсах

В следующей таблице показаны параметры маршалинга для строкового типа данных при передаче в качестве аргумента метода в неуправляемый код. Атрибут MarshalAsAttribute предоставляет несколько значений перечисления UnmanagedType для маршалирования строк в интерфейсы COM.

Развернуть таблицу

| Тип перечисления | Описание неуправляемого формата |
|---|---|
| `UnmanagedType.BStr` (по умолчанию) | COM-стиль `BSTR` с префиксной длиной и символами Юникод. |
| `UnmanagedType.LPStr` | Указатель на массив символов ANSI, завершающийся значением NULL. |
| `UnmanagedType.LPWStr` | Указатель на массив символов Юникода, завершающийся значением NULL. |

Эта таблица относится к String. Для StringBuilder разрешены только варианты `UnmanagedType.LPStr` и `UnmanagedType.LPWStr`.

В следующем примере показаны строки, объявленные в интерфейсе `IStringWorker`.

C#

```
public interface IStringWorker
{
    void PassString1(string s);
```

```
    void PassString2([MarshalAs(UnmanagedType.BStr)] string s);
    void PassString3([MarshalAs(UnmanagedType.LPStr)] string s);
    void PassString4([MarshalAs(UnmanagedType.LPWStr)] string s);
    void PassStringRef1(ref string s);
    void PassStringRef2([MarshalAs(UnmanagedType.BStr)] ref string s);
    void PassStringRef3([MarshalAs(UnmanagedType.LPStr)] ref string s);
    void PassStringRef4([MarshalAs(UnmanagedType.LPWStr)] ref string s);
}
```

В следующем примере показан соответствующий интерфейс, описанный в библиотеке типов.

C++

```
interface IStringWorker : IDispatch
{
    HRESULT PassString1([in] BSTR s);
    HRESULT PassString2([in] BSTR s);
    HRESULT PassString3([in] LPStr s);
    HRESULT PassString4([in] LPWStr s);
    HRESULT PassStringRef1([in, out] BSTR *s);
    HRESULT PassStringRef2([in, out] BSTR *s);
    HRESULT PassStringRef3([in, out] LPStr *s);
    HRESULT PassStringRef4([in, out] LPWStr *s);
};
```

# Строки, используемые в Platform Invoke (вызове платформы)

Если CharSet имеет значение Unicode или строковый аргумент явно помечен как [MarshalAs(UnmanagedType.LPWSTR)] и строка передается по значению (не как `ref` или `out`), строка закрепляется и используется непосредственно в нативном коде. В противном случае вызов платформы копирует строковые аргументы, преобразуя формат .NET Framework (Юникод) в неуправляемый формат платформы. Строки неизменяемы и не копируются из неуправляемой памяти в управляемую память при возврате вызова.

Нативный код отвечает только за освобождение памяти, если строка передается по ссылке и ей назначается новое значение. В противном случае среда выполнения .NET владеет памятью и отпустит ее после вызова.

В следующей таблице перечислены варианты маршалинга строк в качестве аргументов метода платформенного вызова. Атрибут MarshalAsAttribute предоставляет несколько UnmanagedType значений перечисления для маршалирования строк.

Развернуть таблицу

| Тип перечисления | Описание неуправляемого формата |
|---|---|
| `UnmanagedType.AnsiBStr` | Стиль COM `BSTR` с префиксной длиной и символами ANSI. |
| `UnmanagedType.BStr` | COM-стиль `BSTR` с префиксной длиной и символами Юникод. |
| `UnmanagedType.LPStr` (по умолчанию) | Указатель на массив символов ANSI, завершающийся значением NULL. |
| `UnmanagedType.LPTStr` | Указатель на массив символов, зависимых от платформы, завершаемых значением NULL. |
| `UnmanagedType.LPUTF8Str` | Указатель на массив символов, завершаемых значением NULL, в кодировке UTF-8. |
| `UnmanagedType.LPWStr` | Указатель на массив символов Юникода, завершающийся значением NULL. |
| `UnmanagedType.TBStr` | Стиль COM `BSTR` с префиксной длиной и зависимыми от платформы символами. |
| `VBByRefStr` | Значение, которое позволяет Visual Basic изменять строку в неуправляемом коде и отражать результаты в управляемом коде. Это значение поддерживается только для вызова платформы. Это значение по умолчанию в Visual Basic для строк `ByVal`. |

Эта таблица относится к String. Для StringBuilder, допустимы только варианты `LPStr`, `LPTStr` и `LPWStr`.

В следующем определении типа показано правильное использование `MarshalAsAttribute` для вызовов платформы platform invoke.

**C#**

```
class StringLibAPI
{
    [DllImport("StringLib.dll")]
    public static extern void PassLPStr([MarshalAs(UnmanagedType.LPStr)] string s);
    [DllImport("StringLib.dll")]
    public static extern void PassLPWStr([MarshalAs(UnmanagedType.LPWStr)] string
s);
    [DllImport("StringLib.dll")]
    public static extern void PassLPTStr([MarshalAs(UnmanagedType.LPTStr)] string
s);
    [DllImport("StringLib.dll")]
    public static extern void PassLPUTF8Str([MarshalAs(UnmanagedType.LPUTF8Str)]
string s);
    [DllImport("StringLib.dll")]
```

```
    public static extern void PassBStr([MarshalAs(UnmanagedType.BStr)] string s);
    [DllImport("StringLib.dll")]
    public static extern void PassAnsiBStr([MarshalAs(UnmanagedType.AnsiBStr)]
string s);
    [DllImport("StringLib.dll")]
    public static extern void PassTBStr([MarshalAs(UnmanagedType.TBStr)] string s);
}
```

# Строки, используемые в структурах

Строки являются допустимыми элементами структур; StringBuilder однако буферы недопустимы в структурах. В следующей таблице показаны варианты передачи для типа данных String, когда тип передаётся как поле. Атрибут MarshalAsAttribute предоставляет несколько значений перечисления UnmanagedType для маршалирования строк в поле.

Развернуть таблицу

| Тип перечисления | Описание неуправляемого формата |
|---|---|
| `UnmanagedType.BStr` | COM-стиль `BSTR` с префиксной длиной и символами Юникод. |
| `UnmanagedType.LPStr` (по умолчанию) | Указатель на массив символов ANSI, завершающийся значением NULL. |
| `UnmanagedType.LPTStr` | Указатель на массив символов, зависимых от платформы, завершаемых значением NULL. |
| `UnmanagedType.LPUTF8Str` | Указатель на массив символов, завершаемых значением NULL, в кодировке UTF-8. |
| `UnmanagedType.LPWStr` | Указатель на массив символов Юникода, завершающийся значением NULL. |
| `UnmanagedType.ByValTStr` | Массив символов фиксированной длины; Тип массива определяется набором символов содержащей структуры. |

Тип `ByValTStr` используется для встроенных массивов символов фиксированной длины, которые отображаются в структуре. Другие типы применяются к ссылкам на строки, содержащихся в структурах, которые содержат указатели на строки.

Аргумент `CharSet` , применяемый StructLayoutAttribute к содержащей структуре, определяет формат символов строк в структурах. В следующих примерах структур содержатся строковые ссылки и встроенные строки, а также ANSI, Юникод и зависимые от платформы символы. Представление этих структур в библиотеке типов отображается в следующем коде C++:

C++

```
struct StringInfoA
{
    char *  f1;
    char    f2[256];
};

struct StringInfoW
{
    WCHAR * f1;
    WCHAR   f2[256];
    BSTR    f3;
};

struct StringInfoT
{
    TCHAR * f1;
    TCHAR   f2[256];
};
```

В следующем примере показано, как использовать MarshalAsAttribute для определения одной и той же структуры в разных форматах.

C#

```
[StructLayout(LayoutKind.Sequential, CharSet = CharSet.Ansi)]
struct StringInfoA
{
    [MarshalAs(UnmanagedType.LPStr)] public string f1;
    [MarshalAs(UnmanagedType.ByValTStr, SizeConst = 256)] public string f2;
}

[StructLayout(LayoutKind.Sequential, CharSet = CharSet.Unicode)]
struct StringInfoW
{
    [MarshalAs(UnmanagedType.LPWStr)] public string f1;
    [MarshalAs(UnmanagedType.ByValTStr, SizeConst = 256)] public string f2;
    [MarshalAs(UnmanagedType.BStr)] public string f3;
}

[StructLayout(LayoutKind.Sequential, CharSet = CharSet.Auto)]
struct StringInfoT
{
    [MarshalAs(UnmanagedType.LPTStr)] public string f1;
    [MarshalAs(UnmanagedType.ByValTStr, SizeConst = 256)] public string f2;
}
```

# Буферы строк фиксированной длины

В некоторых случаях буфер символов фиксированной длины должен быть передан в неуправляемый код для управления. Простая передача строки не работает в данном случае, так как вызываемый объект не может изменить содержимое переданного буфера. Даже если строка передается по ссылке, невозможно инициализировать буфер в заданный размер.

Решение заключается в передаче `byte[]` или `char[]` (в зависимости от ожидаемой кодировки) в качестве аргумента вместо аргумента String. Массив, помеченный `[Out]`, может быть разыменован и изменен вызывающей стороной, если он не превышает объём выделенного массива.

Например, функция API Windows GetWindowText (определенная в *winuser.h*) требует, чтобы вызывающий объект передал буфер символов фиксированной длины, в который функция записывает текст окна. Аргумент `lpString` указывает на выделенный вызывающим буфер размером `nMaxCount`. Ожидается, что вызывающая сторона выделяет буфер и указывает аргумент `nMaxCount`, равный размеру выделенного буфера. В следующем примере показано `GetWindowText` объявление функции, определённое в *winuser.h*.

C++

```
int GetWindowText(
    HWND hWnd,         // Handle to window or control.
    LPTStr lpString,   // Text buffer.
    int nMaxCount      // Maximum number of characters to copy.
);
```

`char[]` может быть разыменован и изменён вызываемой функцией. Рекомендуемый подход — использовать ArrayPool<T> для аренды `char[]`, что позволяет избежать повторяющихся выделений кучи. В следующем примере кода показан этот шаблон.

C#

```
using System;
using System.Buffers;
using System.Runtime.InteropServices;

internal static class NativeMethods
{
    [DllImport("User32.dll", CharSet = CharSet.Unicode)]
    public static extern int GetWindowText(IntPtr hWnd, [Out] char[] lpString, int
nMaxCount);
}
```

```
public class Window
{
    internal IntPtr h;        // Internal handle to Window.
    public string GetText()
    {
        char[] buffer = ArrayPool<char>.Shared.Rent(256 + 1);
        try
        {
            int length = NativeMethods.GetWindowText(h, buffer, buffer.Length);
            return new string(buffer, 0, length);
        }
        finally
        {
            ArrayPool<char>.Shared.Return(buffer);
        }
    }
}
```

Вы также можете рассмотреть возможность передачи StringBuilder вместо String. Буфер, созданный при маршаллинге `StringBuilder`, может быть разыменован и изменен вызываемой стороной, если только он не превышает объем `StringBuilder`. Его также можно инициализировать до фиксированной длины. Например, если инициализировать буфер `StringBuilder` вместимостью `N`, маршаллизатор предоставляет буфер размером (`N` +1) символов. +1 учитывает тот факт, что неуправляемая строка имеет нулевой терминатор, а в `StringBuilder` он отсутствует.

**⊗ Внимание**

Избегайте `StringBuilder` параметров при решении проблем с производительностью. Маршаллирование `StringBuilder` *всегда* создает копию нативного буфера. Типичный вызов для извлечения строки из нативного кода может привести к четырём выделениям:

1. Управляемый `StringBuilder` буфер.
2. Собственный буфер, выделенный во время маршаллинга.
3. Если `[Out]`, содержимое собственного буфера копируется в недавно выделенный управляемый массив.
4. Выделено `string` `ToString()` по .

Повторное использование одного и того же `StringBuilder` для вызовов сохраняет только одно выделение памяти. Использование буфера символов, арендованного из `ArrayPool<char>`, гораздо более эффективно — это уменьшает последующие вызовы только до выделения для `ToString()`.

Кроме того, `StringBuilder` емкость **не** включает скрытый конечный элемент NULL, для которого всегда учитывается взаимодействие. Это распространенная ошибка, так как большинство API хотят, чтобы в размер буфера входил нулевой символ. Это может привести к потере ресурсов или ненужным выделениям, и это предотвращает оптимизацию `StringBuilder` маршаллинга для сокращения количества копий.

Дополнительные сведения см. в разделе **"Строковые параметры** " и **CA1838: избегайте StringBuilder параметров для P/Invokes**.

## См. также

- Поведение маршаллинга по умолчанию
- Упорядочение строк
- Преобразуемые и непреобразуемые типы
- атрибуты направления
- Копирование и закрепление

ⓘ **Примечание.** Автор создал эту статью с помощью ИИ. Подробнее

---

Last updated on 08.04.2026

# Маршаллирование данных с помощью платформенного вызова

Для вызова функций, экспортированных из неуправляемой библиотеки, приложению .NET Framework требуется прототип функции в управляемом коде, представляющего неуправляемую функцию. Чтобы создать прототип, который позволяет платформе правильно выполнять маршалирование данных, необходимо выполнить следующее:

- Примените атрибут к статической функции или методу DllImportAttribute в управляемом коде.
- Замените неуправляемые типы данных на управляемые типы данных.

Документацию, предоставленную неуправляемой функцией, можно использовать для создания эквивалентного управляемого прототипа, применяя атрибут с необязательными полями и заменяя управляемые типы данных для неуправляемых типов. Инструкции о том, как применять DllImportAttribute, см. в разделе «Использование неуправляемых функций DLL».

В этом разделе приведены примеры, демонстрирующие создание прототипов управляемых функций для передачи аргументов и получения возвращаемых значений из функций, экспортированных неуправляемых библиотеками. Примеры также демонстрируют, когда атрибут MarshalAsAttribute и класс Marshal следует использовать для явного маршалирования данных.

## Вызов типов данных платформы

В следующей таблице перечислены типы данных, используемые в api-интерфейсах Windows и функциях в стиле C. Многие неуправляемые библиотеки содержат функции, которые передают эти типы данных в качестве параметров и возвращаемых значений. Третий столбец содержит соответствующий встроенный тип или класс .NET Framework, используемый в управляемом коде. В некоторых случаях можно заменить один тип другим того же размера, указанным в таблице.

Развернуть таблицу

| Неуправляемый тип в API Windows | Неуправляемый тип языка C | Управляемый тип | Описание |
|---|---|---|---|
| `VOID` | `void` | System.Void | Применяется к функции, которая не возвращает значение. |

| Неуправляемый тип в API Windows | Неуправляемый тип языка C | Управляемый тип | Описание |
|---|---|---|---|
| `HANDLE` | `void *` | System.IntPtr или System.UIntPtr | 32 бита в 32-разрядных операционных системах Windows, 64 бита в 64-разрядных операционных системах Windows. |
| `BYTE` | `unsigned char` | System.Byte | 8 бит |
| `SHORT` | `short` | System.Int16 | 16 бит |
| `WORD` | `unsigned short` | System.UInt16 | 16 бит |
| `INT` | `int` | System.Int32 | 32 бита |
| `UINT` | `unsigned int` | System.UInt32 | 32 бита |
| `LONG` | `long` | System.Int32 | 32 бита |
| `BOOL` | `long` | System.Boolean или System.Int32 | 32 бита |
| `DWORD` | `unsigned long` | System.UInt32 | 32 бита |
| `ULONG` | `unsigned long` | System.UInt32 | 32 бита |
| `CHAR` | `char` | System.Char | Украшайте с ANSI. |
| `WCHAR` | `wchar_t` | System.Char | Украсьте с помощью Юникода. |
| `LPSTR` | `char *` | System.String или System.Text.StringBuilder | Украшайте с ANSI. |
| `LPCSTR` | `const char *` | System.String или System.Text.StringBuilder | Украшайте с ANSI. |
| `LPWSTR` | `wchar_t *` | System.String или System.Text.StringBuilder | Украсьте с помощью Юникода. |
| `LPCWSTR` | `const wchar_t *` | System.String или System.Text.StringBuilder | Украсьте с помощью Юникода. |
| `FLOAT` | `float` | System.Single | 32 бита |
| `DOUBLE` | `double` | System.Double | 64 бита |

Сведения о соответствующих типах в Visual Basic, C#и C++см. в статье "Введение в библиотеку классов .NET Framework".

# PinvokeLib.dll

Следующий код определяет функции библиотеки, предоставляемые PInvokeLib.dll. Многие примеры, описанные в этом разделе, используют эту библиотеку.

## Пример

C++

```
// PInvokeLib.cpp : Defines the entry point for the DLL application.
//

#define PINVOKELIB_EXPORTS
#include "PInvokeLib.h"

#include <strsafe.h>
#include <objbase.h>
#include <stdio.h>

#pragma comment(lib,"ole32.lib")

BOOL APIENTRY DllMain( HANDLE hModule,
                       DWORD  ul_reason_for_call,
                       LPVOID lpReserved )
{
    switch (ul_reason_for_call)
    {
        case DLL_PROCESS_ATTACH:
        case DLL_THREAD_ATTACH:
        case DLL_THREAD_DETACH:
        case DLL_PROCESS_DETACH:
            break;
    }

    return TRUE;
}

//*****************************************************************
// This is the constructor of a class that has been exported.
CTestClass::CTestClass()
{
    m_member = 1;
}

int CTestClass::DoSomething( int i )
{
    return i*i + m_member;
}

PINVOKELIB_API CTestClass* CreateTestClass()
{
    return new CTestClass();
```

```
}

PINVOKELIB_API void DeleteTestClass( CTestClass* instance )
{
    delete instance;
}

//*****************************************************************
PINVOKELIB_API int TestArrayOfInts( int* pArray, int size )
{
    int result = 0;

    for ( int i = 0; i < size; i++ )
    {
        result += pArray[ i ];
        pArray[i] += 100;
    }
    return result;
}

//*****************************************************************
PINVOKELIB_API int TestRefArrayOfInts( int** ppArray, int* pSize )
{
    int result = 0;

    // CoTaskMemAlloc must be used instead of the new operator
    // because code on the managed side will call Marshal.FreeCoTaskMem
    // to free this memory.

    int* newArray = (int*)CoTaskMemAlloc( sizeof(int) * 5 );

    for ( int i = 0; i < *pSize; i++ )
    {
        result += (*ppArray)[i];
    }

    for ( int j = 0; j < 5; j++ )
    {
        newArray[j] = (*ppArray)[j] + 100;
    }

    CoTaskMemFree( *ppArray );
    *ppArray = newArray;
    *pSize = 5;

    return result;
}

//*****************************************************************
PINVOKELIB_API int TestMatrixOfInts( int pMatrix[][COL_DIM], int row )
{
    int result = 0;

    for ( int i = 0; i < row; i++ )
    {
```

```
        for ( int j = 0; j < COL_DIM; j++ )
        {
            result += pMatrix[i][j];
            pMatrix[i][j] += 100;
        }
    }
    return result;
}

//*****************************************************************
PINVOKELIB_API int TestArrayOfStrings( char* ppStrArray[], int count )
{
    int result = 0;
    STRSAFE_LPSTR temp;
    size_t len;
    const size_t alloc_size = sizeof(char) * 10;

    for ( int i = 0; i < count; i++ )
    {
        len = 0;
        StringCchLengthA( ppStrArray[i], STRSAFE_MAX_CCH, &len );
        result += len;

        temp = (STRSAFE_LPSTR)CoTaskMemAlloc( alloc_size );
        StringCchCopyA( temp, alloc_size, (STRSAFE_LPCSTR)"123456789" );

       // CoTaskMemFree must be used instead of delete to free memory.

       CoTaskMemFree( ppStrArray[i] );
       ppStrArray[i] = (char *) temp;
   }

   return result;
}

//*****************************************************************
PINVOKELIB_API int TestArrayOfStructs( MYPOINT* pPointArray, int size )
{
    int result = 0;
    MYPOINT* pCur = pPointArray;

    for ( int i = 0; i < size; i++ )
    {
        result += pCur->x + pCur->y;
        pCur->y = 0;
        pCur++;
    }

    return result;
}

//*****************************************************************
PINVOKELIB_API int TestStructInStruct( MYPERSON2* pPerson2 )
{
    size_t len = 0;
```

```
    StringCchLengthA( pPerson2->person->last, STRSAFE_MAX_CCH, &len );
    len = sizeof(char) * ( len + 2 ) + 1;

    STRSAFE_LPSTR temp = (STRSAFE_LPSTR)CoTaskMemAlloc( len );
    StringCchCopyA( temp, len, (STRSAFE_LPSTR)"Mc" );
    StringCbCatA( temp, len, (STRSAFE_LPSTR)pPerson2->person->last );

    CoTaskMemFree( pPerson2->person->last );
    pPerson2->person->last = (char *)temp;

    return pPerson2->age;
}

//*****************************************************************
PINVOKELIB_API int TestArrayOfStructs2( MYPERSON* pPersonArray, int size )
{
    int result = 0;
    MYPERSON* pCur = pPersonArray;
    STRSAFE_LPSTR temp;
    size_t len;

    for ( int i = 0; i < size; i++ )
    {
        len = 0;
        StringCchLengthA( pCur->first, STRSAFE_MAX_CCH, &len );
        len++;
        result += len;
        len = 0;
        StringCchLengthA( pCur->last, STRSAFE_MAX_CCH, &len );
        len++;
        result += len;

        len = sizeof(char) * ( len + 2 );
        temp = (STRSAFE_LPSTR)CoTaskMemAlloc( len );
        StringCchCopyA( temp, len, (STRSAFE_LPCSTR)"Mc" );
        StringCbCatA( temp, len, (STRSAFE_LPCSTR)pCur->last );
        result += 2;

        // CoTaskMemFree must be used instead of delete to free memory.
        CoTaskMemFree( pCur->last );
        pCur->last = (char *)temp;
        pCur++;
   }

   return result;
}

//*****************************************************************
PINVOKELIB_API void TestStructInStruct3( MYPERSON3 person3 )
{
    printf( "\n\nperson passed by value:\n" );
    printf( "first = %s last = %s age = %i\n\n",
            person3.person.first,
            person3.person.last,
```

```
            person3.age );
}

//************************************************************************
PINVOKELIB_API void TestUnion( MYUNION u, int type )
{
    if ( ( type != 1 ) && ( type != 2 ) )
    {
        return;
    }
    if ( type == 1 )
    {
        printf( "\n\ninteger passed: %i", u.i );
    }
    else if ( type == 2 )
    {
        printf( "\n\ndouble passed: %f", u.d );
    }
}

//************************************************************************
PINVOKELIB_API void TestUnion2( MYUNION2 u, int type )
{
    if ( ( type != 1 ) && ( type != 2 ) )
    {
        return;
    }
    if ( type == 1 )
    {
        printf( "\n\ninteger passed: %i", u.i );
    }
    else if ( type == 2 )
    {
        printf( "\n\nstring passed: %s", u.str );
    }
}

//************************************************************************
PINVOKELIB_API void TestCallBack( FPTR pf, int value )
{
    printf( "\nReceived value: %i", value );
    printf( "\nPassing to callback..." );
    bool res = (*pf)(value);

    if ( res )
    {
        printf( "Callback returned true.\n" );
    }
    else
    {
        printf( "Callback returned false.\n" );
    }
}

//************************************************************************
```

```
PINVOKELIB_API void TestCallBack2( FPTR2 pf2, char* value )
{
    printf( "\nReceived value: %s", value );
    printf( "\nPassing to callback..." );
    bool res = (*pf2)(value);

    if ( res )
    {
        printf( "Callback2 returned true.\n" );
    }
    else
    {
        printf( "Callback2 returned false.\n" );
    }
}

//*****************************************************************
PINVOKELIB_API void TestStringInStruct( MYSTRSTRUCT* pStruct )
{
    wprintf( L"\nUnicode buffer content: %s\n", pStruct->buffer );

    // Assuming that the buffer is big enough.
    StringCbCatW( pStruct->buffer, pStruct->size, (STRSAFE_LPWSTR)L"++" );
}

//*****************************************************************
PINVOKELIB_API void TestStringInStructAnsi( MYSTRSTRUCT2* pStruct )
{
    printf( "\nAnsi buffer content: %s\n", pStruct->buffer );

    // Assuming that the buffer is big enough.
    StringCbCatA( (STRSAFE_LPSTR) pStruct->buffer, pStruct->size,
(STRSAFE_LPSTR)"++" );
}

//*****************************************************************
PINVOKELIB_API void TestOutArrayOfStructs( int* pSize, MYSTRSTRUCT2** ppStruct )
{
    const int cArraySize = 5;
    *pSize = 0;
    *ppStruct = (MYSTRSTRUCT2*)CoTaskMemAlloc( cArraySize * sizeof( MYSTRSTRUCT2
));

    if ( ppStruct != NULL )
    {
        MYSTRSTRUCT2* pCurStruct = *ppStruct;
        LPSTR buffer;
        *pSize = cArraySize;

        STRSAFE_LPCSTR teststr = "***";
        size_t len = 0;
        StringCchLengthA(teststr, STRSAFE_MAX_CCH, &len);
        len++;

        for ( int i = 0; i < cArraySize; i++, pCurStruct++ )
```

```
        {
            pCurStruct->size = len;
            buffer = (LPSTR)CoTaskMemAlloc( len );
            StringCchCopyA( buffer, len, teststr );
            pCurStruct->buffer = (char *)buffer;
        }
    }
}

//*****************************************************************************
PINVOKELIB_API char * TestStringAsResult()
{

    const size_t alloc_size = 64;
    STRSAFE_LPSTR result = (STRSAFE_LPSTR)CoTaskMemAlloc( alloc_size );
    STRSAFE_LPCSTR teststr = "This is return value";
    StringCchCopyA( result, alloc_size, teststr );

    return (char *) result;
}

//*****************************************************************************
PINVOKELIB_API void SetData( DataType typ, void* object )
{
    switch ( typ )
    {
        case DT_I2: printf( "Short %i\n", *((short*)object) ); break;
        case DT_I4: printf( "Long %i\n", *((long*)object) ); break;
        case DT_R4: printf( "Float %f\n", *((float*)object) ); break;
        case DT_R8: printf( "Double %f\n", *((double*)object) ); break;
        case DT_STR: printf( "String %s\n", (char*)object ); break;
        default: printf( "Unknown type" ); break;
    }
}

//*****************************************************************************
PINVOKELIB_API void TestArrayInStruct( MYARRAYSTRUCT* pStruct )
{
    pStruct->flag = true;
    pStruct->vals[0] += 100;
    pStruct->vals[1] += 100;
    pStruct->vals[2] += 100;
}
```

Чтобы вызвать функции библиотеки из управляемого кода, реализуйте управляемые прототипы для каждой функции, которую требуется вызвать. Если неуправляемый код использует любые пользовательские типы, необходимо также объявить эти типы в управляемом коде. Украшайте каждый прототип атрибутом DllImportAttribute и используйте StructLayoutAttribute для управления макетом управляемых структур, чтобы они соответствовали неуправляемыми эквивалентами.

## Объявления управляемых типов

В следующем коде показаны управляемые эквиваленты неуправляемых типов, определенных в `PInvokeLib.h`. Каждая структура использует `StructLayout` для соответствия макету поля с неуправляемым макетом.

C#

```
// Managed type declarations that correspond to the unmanaged types in
PinvokeLib.dll.

[StructLayout(LayoutKind.Sequential)]
internal struct MyPoint
{
    public int x;
    public int y;
}

[StructLayout(LayoutKind.Sequential, CharSet = CharSet.Ansi)]
internal struct MyPerson
{
    public string first;
    public string last;
}

[StructLayout(LayoutKind.Sequential)]
internal struct MyPerson2
{
    public IntPtr person; // Pointer to a MyPerson structure
    public int age;
}

[StructLayout(LayoutKind.Sequential)]
internal struct MyPerson3
{
    public MyPerson person; // Embedded MyPerson structure
    public int age;
}

[StructLayout(LayoutKind.Explicit)]
internal struct MyUnion
{
    [FieldOffset(0)] public int i;
    [FieldOffset(0)] public double d;
}

[StructLayout(LayoutKind.Sequential)]
internal struct MyArrayStruct
{
    public bool flag;

    [MarshalAs(UnmanagedType.ByValArray, SizeConst = 3)]
```

```
    public int[] vals;
}
```

## Прототипы управляемых функций

В следующем коде показаны `DllImport` объявления, которые предоставляют неуправляемые функции из `PinvokeLib.dll` для управляемого кода.

C#

```
internal static class NativeMethods
{
    [DllImport("PinvokeLib.dll", CallingConvention = CallingConvention.Cdecl)]
    internal static extern int TestArrayOfInts(
        [In, Out] int[] array, int size);

    [DllImport("PinvokeLib.dll", CallingConvention = CallingConvention.Cdecl)]
    internal static extern int TestRefArrayOfInts(
        ref IntPtr array, ref int size);

    [DllImport("PinvokeLib.dll", CallingConvention = CallingConvention.Cdecl)]
    internal static extern int TestMatrixOfInts(
        [In, Out] int[,] matrix, int row);

    [DllImport("PinvokeLib.dll", CallingConvention = CallingConvention.Cdecl)]
    internal static extern int TestArrayOfStrings(
        [In, Out] string[] array, int size);

    [DllImport("PinvokeLib.dll", CallingConvention = CallingConvention.Cdecl)]
    internal static extern int TestArrayOfStructs(
        [In, Out] MyPoint[] pointArray, int size);

    [DllImport("PinvokeLib.dll", CallingConvention = CallingConvention.Cdecl)]
    internal static extern int TestArrayOfStructs2(
        [In, Out] MyPerson[] personArray, int size);

    [DllImport("PinvokeLib.dll", CallingConvention = CallingConvention.Cdecl)]
    internal static extern int TestStructInStruct(ref MyPerson2 person2);

    [DllImport("PinvokeLib.dll", CallingConvention = CallingConvention.Cdecl)]
    internal static extern void TestStructInStruct3(MyPerson3 person3);

    [DllImport("PinvokeLib.dll", CallingConvention = CallingConvention.Cdecl)]
    internal static extern void TestUnion(MyUnion u, int type);

    [DllImport("PinvokeLib.dll", CallingConvention = CallingConvention.Cdecl)]
    internal static extern void TestArrayInStruct(ref MyArrayStruct myStruct);
}
```

Дополнительные сведения и примеры см. в следующих разделах:

- Маршаллинг классов, структур и профсоюзов
- Упорядочение строк
- Маршалинг различных типов массивов

ⓘ **Примечание.** Автор создал эту статью с помощью ИИ. Подробнее

---

Last updated on 12.03.2026

# Маршаллинг строк

17.06.2025

При необходимости вызов платформы копирует строковые параметры, преобразуя их из формата .NET Framework (Юникод) в неуправляемый формат (ANSI). Так как управляемые строки неизменяемы, вызов платформы не копирует их из неуправляемой памяти в управляемую память при возврате функции.

В следующей таблице перечислены параметры маршаллинга строк, описание их использования и ссылка на соответствующий пример .NET Framework.

Развернуть таблицу

| Струна | Описание | Образец |
|---|---|---|
| По значению. | Передает строки в качестве входных параметров. | MsgBox |
| В результате. | Возвращает строки из неуправляемого кода. | строки |
| По ссылке. | Передает строки в качестве параметров in/Out с помощью StringBuilder. | Буферы |
| В структуре по значению. | Передает строки в структуре, являющейся параметром In. | структуры |
| В структуре через ссылку **(char*)**. | Передает строки в структуре, которая используется как входной и выходной параметр. Неуправляемая функция ожидает указатель на буфер символов, а размер буфера является членом структуры. | строки |
| В структуре, передаваемой по ссылке **(char[])**. | Передает строки в структуре, которая используется как входной и выходной параметр. Неуправляемая функция ожидает встроенный буфер символов. | OSInfo |
| В классе по значению **(char*)**. | Передает строки в классе (класс является параметром In/Out). Неуправляемая функция ожидает указатель на буфер символов. | OpenFileDlg |
| В классе по значению **(char[])**. | Передает строки в классе (класс является параметром In/Out). Неуправляемая функция ожидает встроенный буфер символов. | OSInfo |
| Как массив строк по значению. | Создает массив строк, передаваемых по значению. | Массивы |
| Как массив структур, в которых строки | Создает массив структур, содержащих строки, и массив передается по значению. | Массивы |

| Струна | Описание | Образец |
| --- | --- | --- |
| представлены в виде значений. | | |

## См. также

- Маршаллинг по умолчанию для строк
- Маршалинг данных с использованием Platform Invoke
- Маршаллинг классов, структур и профсоюзов
- Маршалинг различных типов массивов
- Различные примеры маршаллинга

# Пример MsgBox

17.06.2025

В этом примере показано, как передавать строковые типы по значению в качестве входных параметров и когда использовать поля EntryPoint, CharSet и ExactSpelling.

В примере MsgBox используется следующая неконтролируемая функция, показанная с ее исходным объявлением функции.

- **MessageBox** экспортируется из User32.dll.

```
C++

int MessageBox(HWND hWnd, LPCTSTR lpText, LPCTSTR lpCaption,
   UINT uType);
```

В этом примере `NativeMethods` класс содержит управляемый прототип для каждой неуправляемой функции, вызываемой классом `MsgBoxSample` . У методов `MsgBox`, `MsgBox2` управляемого прототипа и `MsgBox3` разные объявления для одной неуправляемой функции.

Объявление для `MsgBox2` приводит к неправильному выводу в окне сообщения, так как тип символа, указанный как ANSI, не соответствует точке входа `MessageBoxW`, которая является именем функции Unicode. Объявление для `MsgBox3` создаёт рассогласование между полями **EntryPoint**, **CharSet** и **ExactSpelling**. При вызове `MsgBox3` вызывается исключение. Подробные сведения об именовании строк и управлении именами см. раздел Указание набора символов.

## Объявление прототипов

```
C#

internal static class NativeMethods
{
    // Declares managed prototypes for unmanaged functions.
    [DllImport("User32.dll", EntryPoint = "MessageBox",
        CharSet = CharSet.Auto)]
    internal static extern int MsgBox(
        IntPtr hWnd, string lpText, string lpCaption, uint uType);

    // Causes incorrect output in the message window.
    [DllImport("User32.dll", EntryPoint = "MessageBoxW",
        CharSet = CharSet.Ansi)]
    internal static extern int MsgBox2(
```

```
        IntPtr hWnd, string lpText, string lpCaption, uint uType);

    // Causes an exception to be thrown. EntryPoint, CharSet, and
    // ExactSpelling fields are mismatched.
    [DllImport("User32.dll", EntryPoint = "MessageBox",
        CharSet = CharSet.Ansi, ExactSpelling = true)]
    internal static extern int MsgBox3(
        IntPtr hWnd, string lpText, string lpCaption, uint uType);
}
```

## Вызов функций

C#

```
public class MsgBoxSample
{
    public static void Main()
    {
        NativeMethods.MsgBox(0, "Correct text", "MsgBox Sample", 0);
        NativeMethods.MsgBox2(0, "Incorrect text", "MsgBox Sample", 0);

        try
        {
            NativeMethods.MsgBox3(0, "No such function", "MsgBox Sample", 0);
        }
        catch (EntryPointNotFoundException)
        {
            Console.WriteLine($"{nameof(EntryPointNotFoundException)} thrown as
expected!");
        }
    }
}
```

## См. также

- Упорядочение строк
- Маршаллинг по умолчанию для строк
- Создание прототипов в управляемом коде
- Указание набора символов

# Маршалинг классов, структур и объединений

Статья • 08.04.2023

Классы и структуры в .NET Framework похожи. И те и другие могут иметь поля, свойства и события. Они также могут иметь статические и нестатические методы. Примечательным отличием является то, что структуры являются типами значений, а классы — ссылочными типами.

В следующей таблице перечислены параметры маршалинга для классов, структур и объединений; описывает их использование; и предоставляет ссылку на соответствующий пример вызова платформы.

| Тип | Описание | Пример |
|---|---|---|
| Класс по значению. | Передает класс с целочисленными членами в качестве параметра In или Out (как и в случае управляемого класса). | Пример SysTime |
| Структура по значению. | Передает структуры в качестве параметров In. | Пример структур |
| Структура по ссылке. | Передает структуры в качестве параметров In и Out. | Пример OSInfo |
| Структура с вложенными структурами (выровненная). | Передает класс, представляющий структуру с вложенными структурами, в неуправляемую функцию. Структура выравнивается в одну большую структуру в управляемом прототипе. | Пример FindFile |
| Структура с указателем на другую структуру. | Передает структуру, содержащую указатель на другую структуру в качестве члена. | Пример структур |
| Массив структур с целочисленными значениями по значению. | Передает массив структур, содержащих только целые числа, в виде параметра In или Out. Элементы массива можно изменять. | Пример массивов |
| Массив структур с целочисленными значениями и строками по ссылке. | Передает массив структур, содержащих целые числа и строки, в качестве параметра Out. Вызываемая функция выделяет память под массив. | Пример OutArrayOfStructs |
| Объединения с типами значений. | Передает объединения с типами значений (целочисленными и двойной точности). | Пример объединений |

| Тип | Описание | Пример |
| --- | --- | --- |
| Объединения со смешанными типами. | Передает объединения со смешанными типами (целое число и строка). | Пример объединений |
| Структура с макетом, зависящим от платформы. | Передает тип с определениями собственной упаковки. | Пример платформы |
| Значения Null в структуре. | Передает пустую ссылку (**Nothing** в Visual Basic) вместо ссылки на тип значения. | Пример HandleRef |

# Пример структур

В этом примере показан способ передачи структуры, указывающей на вторую структуру, передачи структуры с внедренной структурой и передачи структуры с внедренным массивом.

В примере используются следующие неуправляемые функции, показанные с исходными объявлениями:

- функция **TestStructInStruct**, экспортированная из PinvokeLib.dll;

```
C++

int TestStructInStruct(MYPERSON2* pPerson2);
```

- функция **TestStructInStruct3**, экспортированная из PinvokeLib.dll;

```
C++

void TestStructInStruct3(MYPERSON3 person3);
```

- функция **TestArrayInStruct**, экспортированная из PinvokeLib.dll.

```
C++

void TestArrayInStruct(MYARRAYSTRUCT* pStruct);
```

PinvokeLib.dll — это пользовательская неуправляемая библиотека, содержащая реализацию вышеуказанных функций и четырех структур: **MYPERSON**, **MYPERSON2**, **MYPERSON3** и **MYARRAYSTRUCT**. Эти структуры содержат следующие элементы:

C++

```
typedef struct _MYPERSON
{
   char* first;
   char* last;
} MYPERSON, *LP_MYPERSON;

typedef struct _MYPERSON2
{
   MYPERSON* person;
   int age;
} MYPERSON2, *LP_MYPERSON2;

typedef struct _MYPERSON3
{
   MYPERSON person;
   int age;
} MYPERSON3;

typedef struct _MYARRAYSTRUCT
{
   bool flag;
   int vals[ 3 ];
} MYARRAYSTRUCT;
```

Управляемые структуры `MyPerson`, `MyPerson2`, `MyPerson3` и `MyArrayStruct` обладают следующими характеристиками:

- Структура `MyPerson` содержит только члены-строки. Поле CharSet задает формат строк ANSI при передаче строки в неуправляемую функцию.
- `MyPerson2` содержит указатель **IntPtr** на структуру `MyPerson`. Тип **IntPtr** заменяет исходный указатель на неуправляемую структуру, так как приложения .NET Framework не используют указатели, если код не помечен как **небезопасный**.
- Структура `MyPerson3` содержит структуру `MyPerson` в качестве внедренной. Внедренную структуру можно выровнять путем помещения ее элементов прямо в основную структуру, или ее можно оставить внедренной, как показано в примере.
- Структура `MyArrayStruct` содержит массив целочисленных значений. Атрибут MarshalAsAttribute задает для перечисления UnmanagedType значение **ByValArray**, которое используется для указания количества элементов в массиве.

Для всех структур в этом примере применяется атрибут StructLayoutAttribute, гарантирующий последовательное размещение элементов в памяти в порядке их появления.

Класс `NativeMethods` содержит управляемые прототипы методов `TestStructInStruct`, `TestStructInStruct3` и `TestArrayInStruct`, вызываемые классом `App`. Каждый прототип объявляет один параметр указанным ниже способом.

- `TestStructInStruct` объявляет в качестве своего параметра ссылку на тип `MyPerson2`.
- `TestStructInStruct3` в качестве своего параметра объявляет тип `MyPerson3` и передает параметр по значению.
- `TestArrayInStruct` объявляет в качестве своего параметра ссылку на тип `MyArrayStruct`.

Структуры, выступающие в роли аргументов методов, передаются по значению, если параметр не содержит ключевого слова **ref** (**ByRef** в Visual Basic). Например, метод `TestStructInStruct` передает в неуправляемый код ссылку (значение адреса) на объект типа `MyPerson2`. Для работы со структурой, на которую указывает `MyPerson2`, в примере с помощью методов Marshal.AllocCoTaskMem и Marshal.SizeOf создается буфер заданного размера и возвращается его адрес. Далее в неуправляемый буфер копируется содержимое управляемой структуры. Наконец, используются метод Marshal.PtrToStructure для маршалинга данных из неуправляемого буфера в управляемый объект и метод Marshal.FreeCoTaskMem для освобождения неуправляемого блока памяти.

## Объявление прототипов

C#

```
// Declares a managed structure for each unmanaged structure.
[StructLayout(LayoutKind.Sequential, CharSet = CharSet.Ansi)]
public struct MyPerson
{
    public string first;
    public string last;
}

[StructLayout(LayoutKind.Sequential)]
public struct MyPerson2
{
    public IntPtr person;
    public int age;
```

```
}

[StructLayout(LayoutKind.Sequential)]
public struct MyPerson3
{
    public MyPerson person;
    public int age;
}

[StructLayout(LayoutKind.Sequential)]
public struct MyArrayStruct
{
    public bool flag;

    [MarshalAs(UnmanagedType.ByValArray, SizeConst = 3)]
    public int[] vals;
}

internal static class NativeMethods
{
    // Declares a managed prototype for unmanaged function.
    [DllImport("..\\LIB\\PinvokeLib.dll", CallingConvention =
CallingConvention.Cdecl)]
    internal static extern int TestStructInStruct(ref MyPerson2 person2);

    [DllImport("..\\LIB\\PinvokeLib.dll", CallingConvention =
CallingConvention.Cdecl)]
    internal static extern int TestStructInStruct3(MyPerson3 person3);

    [DllImport("..\\LIB\\PinvokeLib.dll", CallingConvention =
CallingConvention.Cdecl)]
    internal static extern int TestArrayInStruct(ref MyArrayStruct
myStruct);
}
```

## Вызов функций

C#

```
public class App
{
    public static void Main()
    {
        // Structure with a pointer to another structure.
        MyPerson personName;
        personName.first = "Mark";
        personName.last = "Lee";

        MyPerson2 personAll;
        personAll.age = 30;

        IntPtr buffer = Marshal.AllocCoTaskMem(Marshal.SizeOf(personName));
```

```
        Marshal.StructureToPtr(personName, buffer, false);

        personAll.person = buffer;

        Console.WriteLine("\nPerson before call:");
        Console.WriteLine("first = {0}, last = {1}, age = {2}",
            personName.first, personName.last, personAll.age);

        int res = NativeMethods.TestStructInStruct(ref personAll);

        MyPerson personRes =
            (MyPerson)Marshal.PtrToStructure(personAll.person,
            typeof(MyPerson));

        Marshal.FreeCoTaskMem(buffer);

        Console.WriteLine("Person after call:");
        Console.WriteLine("first = {0}, last = {1}, age = {2}",
            personRes.first, personRes.last, personAll.age);

        // Structure with an embedded structure.
        MyPerson3 person3 = new MyPerson3();
        person3.person.first = "John";
        person3.person.last = "Evans";
        person3.age = 27;
        NativeMethods.TestStructInStruct3(person3);

        // Structure with an embedded array.
        MyArrayStruct myStruct = new MyArrayStruct();

        myStruct.flag = false;
        myStruct.vals = new int[3];
        myStruct.vals[0] = 1;
        myStruct.vals[1] = 4;
        myStruct.vals[2] = 9;

        Console.WriteLine("\nStructure with array before call:");
        Console.WriteLine(myStruct.flag);
        Console.WriteLine("{0} {1} {2}", myStruct.vals[0],
            myStruct.vals[1], myStruct.vals[2]);

        NativeMethods.TestArrayInStruct(ref myStruct);
        Console.WriteLine("\nStructure with array after call:");
        Console.WriteLine(myStruct.flag);
        Console.WriteLine("{0} {1} {2}", myStruct.vals[0],
            myStruct.vals[1], myStruct.vals[2]);
    }
}
```

# пример FindFile

В этом примере показан способ передачи структуры, содержащей другую, внедренную структуру, в неуправляемую функцию. Также показано, как с помощью атрибута MarshalAsAttribute объявить массив фиксированной длины внутри структуры. В этом примере элементы внедренной структуры добавляются в родительскую структуру. Пример внедренной структуры (не преобразованной в плоскую структуру) см. в разделе Пример структур.

В примере FindFile используются следующие неуправляемые функции, показанные с исходными объявлениями:

- функция **FindFirstFile**, экспортированная из Kernel32.dll.

```
C++

HANDLE FindFirstFile(LPCTSTR lpFileName, LPWIN32_FIND_DATA
lpFindFileData);
```

Исходная структура, переданная в функцию, содержит следующие элементы:

```
C++

typedef struct _WIN32_FIND_DATA
{
  DWORD    dwFileAttributes;
  FILETIME ftCreationTime;
  FILETIME ftLastAccessTime;
  FILETIME ftLastWriteTime;
  DWORD    nFileSizeHigh;
  DWORD    nFileSizeLow;
  DWORD    dwReserved0;
  DWORD    dwReserved1;
  TCHAR    cFileName[ MAX_PATH ];
  TCHAR    cAlternateFileName[ 14 ];
} WIN32_FIND_DATA, *PWIN32_FIND_DATA;
```

В этом примере класс `FindData` содержит члены данных, соответствующие каждому элементу исходной и внедренной структур. Вместо двух исходных символьных буферов класс подставляет строки. Атрибут **MarshalAsAttribute** задает для перечисления UnmanagedType значение **ByValTStr**, которое используется для идентификации встроенных символьных массивов фиксированной длины в неуправляемых структурах.

Класс `NativeMethods` содержит управляемый прототип метода `FindFirstFile`, передающий класс `FindData` в качестве параметра. Этот параметр должен быть объявлен с атрибутами InAttribute и OutAttribute, так как классы, являющиеся ссылочными типами, по умолчанию передаются как параметры In.

## Объявление прототипов

C#

```
// Declares a class member for each structure element.
[StructLayout(LayoutKind.Sequential, CharSet = CharSet.Auto)]
public class FindData
{
    public int fileAttributes = 0;
    // creationTime was an embedded FILETIME structure.
    public int creationTime_lowDateTime = 0;
    public int creationTime_highDateTime = 0;
    // lastAccessTime was an embedded FILETIME structure.
    public int lastAccessTime_lowDateTime = 0;
    public int lastAccessTime_highDateTime = 0;
    // lastWriteTime was an embedded FILETIME structure.
    public int lastWriteTime_lowDateTime = 0;
    public int lastWriteTime_highDateTime = 0;
    public int nFileSizeHigh = 0;
    public int nFileSizeLow = 0;
    public int dwReserved0 = 0;
    public int dwReserved1 = 0;
    [MarshalAs(UnmanagedType.ByValTStr, SizeConst = 260)]
    public string fileName = null;
    [MarshalAs(UnmanagedType.ByValTStr, SizeConst = 14)]
    public string alternateFileName = null;
}

internal static class NativeMethods
{
    // Declares a managed prototype for the unmanaged function.
    [DllImport("Kernel32.dll", CharSet = CharSet.Auto)]
    internal static extern IntPtr FindFirstFile(
        string fileName, [In, Out] FindData findFileData);
}
```

## Вызов функций

C#

```
public class App
{
    public static void Main()
    {
        FindData fd = new FindData();
        IntPtr handle = NativeMethods.FindFirstFile("C:\\*.*", fd);
        Console.WriteLine($"The first file: {fd.fileName}");
    }
}
```

# пример Unions

В этом примере показан способ передачи структур, содержащих только типы значений, а также структур, содержащих тип значения и строку, в качестве параметров в неуправляемую функцию, ожидающую объединения. Объединение представляет собой ячейку памяти, которая может совместно использоваться двумя и более переменными.

В примере используются следующие неуправляемые функции, показанные с исходными объявлениями:

- функция **TestUnion**, экспортированная из PinvokeLib.dll.

```
C++

void TestUnion(MYUNION u, int type);
```

PinvokeLib.dll — это пользовательская неуправляемая библиотека, содержащая реализацию указанной выше функции и два объединения: **MYUNION** и **MYUNION2**. Объединение содержит следующие элементы:

```
C++

union MYUNION
{
    int number;
    double d;
}

union MYUNION2
{
    int i;
    char str[128];
};
```

В управляемом коде объединения определяются как структуры. Структура `MyUnion` в качестве своих членов содержит два типа значений: целочисленный и число двойной точности. Для управления точным положением каждого члена данных задается атрибут StructLayoutAttribute. Атрибут FieldOffsetAttribute предоставляет физическое положение полей внутри неуправляемого представления объединения. Обратите внимание, что значения смещения у обоих членов одинаковы, что позволяет членам определять один и тот же участок памяти.

Объединения `MyUnion2_1` и `MyUnion2_2` содержат тип значения (целое число) и строку соответственно. В управляемом коде типы значений и ссылочные типы не

могут перекрываться. Чтобы при вызове одной и той же неуправляемой функции вызывающий объект мог использовать оба типа, в этом примере применяется перегрузка метода. Структура `MyUnion2_1` задана явным образом и имеет точное значение смещения. Напротив, `MyUnion2_2` имеет последовательную структуру, так как структуры, заданные явным образом, нельзя использовать со ссылочными типами. Атрибут MarshalAsAttribute задает для перечисления UnmanagedType значение **ByValTStr**, которое используется для идентификации встроенных символьных массивов фиксированной длины в неуправляемом представлении объединения.

Класс `NativeMethods` содержит прототипы для методов `TestUnion` и `TestUnion2`. `TestUnion2` перегружается для объявления `MyUnion2_1` или `MyUnion2_2` в качестве параметров.

## Объявление прототипов

C#

```
// Declares managed structures instead of unions.
[StructLayout(LayoutKind.Explicit)]
public struct MyUnion
{
    [FieldOffset(0)]
    public int i;
    [FieldOffset(0)]
    public double d;
}

[StructLayout(LayoutKind.Explicit, Size = 128)]
public struct MyUnion2_1
{
    [FieldOffset(0)]
    public int i;
}

[StructLayout(LayoutKind.Sequential)]
public struct MyUnion2_2
{
    [MarshalAs(UnmanagedType.ByValTStr, SizeConst = 128)]
    public string str;
}

internal static class NativeMethods
{
    // Declares managed prototypes for unmanaged function.
    [DllImport("..\\LIB\\PInvokeLib.dll")]
    internal static extern void TestUnion(MyUnion u, int type);

    [DllImport("..\\LIB\\PInvokeLib.dll")]
```

```
    internal static extern void TestUnion2(MyUnion2_1 u, int type);

    [DllImport("..\\LIB\\PInvokeLib.dll")]
    internal static extern void TestUnion2(MyUnion2_2 u, int type);
}
```

## Вызов функций

C#

```
public class App
{
    public static void Main()
    {
        MyUnion mu = new MyUnion();
        mu.i = 99;
        NativeMethods.TestUnion(mu, 1);

        mu.d = 99.99;
        NativeMethods.TestUnion(mu, 2);

        MyUnion2_1 mu2_1 = new MyUnion2_1();
        mu2_1.i = 99;
        NativeMethods.TestUnion2(mu2_1, 1);

        MyUnion2_2 mu2_2 = new MyUnion2_2();
        mu2_2.str = "*** string ***";
        NativeMethods.TestUnion2(mu2_2, 2);
    }
}
```

# Пример платформы

В некоторых сценариях макеты `struct` и `union` могут различаться в зависимости от целевой платформы. Например, рассмотрим тип STRRET при определении в сценарии COM:

c++

```
#include <pshpack8.h> /* Defines the packing of the struct */
typedef struct _STRRET
    {
    UINT uType;
    /* [switch_is][switch_type] */ union
        {
        /* [case()][string] */ LPWSTR pOleStr;
        /* [case()] */ UINT uOffset;
        /* [case()] */ char cStr[ 260 ];
```

```
        }  DUMMYUNIONNAME;
    }  STRRET;
#include <poppack.h>
```

Приведенный выше `struct` объявляется с заголовками Windows, которые влияют на макет памяти типа. При определении в управляемой среде эти сведения о макете необходимы для правильного взаимодействия с машинным кодом.

Правильное управляемое определение этого типа в 32-разрядном процессе имеет следующий вид:

CSharp

```
[StructLayout(LayoutKind.Explicit, Size = 264)]
public struct STRRET_32
{
    [FieldOffset(0)]
    public uint uType;

    [FieldOffset(4)]
    public IntPtr pOleStr;

    [FieldOffset(4)]
    public uint uOffset;

    [FieldOffset(4)]
    public IntPtr cStr;
}
```

В 64-разрядном процессе размеры смещений полей *и* различаются. Правильный макет:

CSharp

```
[StructLayout(LayoutKind.Explicit, Size = 272)]
public struct STRRET_64
{
    [FieldOffset(0)]
    public uint uType;

    [FieldOffset(8)]
    public IntPtr pOleStr;

    [FieldOffset(8)]
    public uint uOffset;

    [FieldOffset(8)]
    public IntPtr cStr;
}
```

Неправильный выбор собственного макета в сценарии взаимодействия может привести к возникновению случайных сбоев или, что еще хуже, к неправильным вычислениям.

По умолчанию сборки .NET могут работать как в 32-разрядной, так и в 64-разрядной версии среды выполнения .NET. Приложение должно ожидать от среды выполнения решения о том, какое из предыдущих определений использовать.

В следующем фрагменте кода показан пример выбора между 32-разрядным и 64-разрядным определением в среде выполнения.

```
C#

if (IntPtr.Size == 8)
{
    // Use the STRRET_64 definition
}
else
{
    Debug.Assert(IntPtr.Size == 4);
    // Use the STRRET_32 definition
}
```

# Пример SysTime

В этом примере показано, как передать указатель на класс в неуправляемую функцию, ожидающую указатель на структуру.

В примере используется следующая неуправляемая функция, показанная с исходным объявлением:

- функция **GetSystemTime**, экспортированная из Kernel32.dll.

  ```
  C++

  VOID GetSystemTime(LPSYSTEMTIME lpSystemTime);
  ```

Исходная структура, переданная в функцию, содержит следующие элементы:

```
C++

typedef struct _SYSTEMTIME {
    WORD wYear;
    WORD wMonth;
    WORD wDayOfWeek;
    WORD wDay;
```

```
    WORD wHour;
    WORD wMinute;
    WORD wSecond;
    WORD wMilliseconds;
} SYSTEMTIME, *PSYSTEMTIME;
```

В этом примере класс `SystemTime` содержит элементы исходной структуры, представленные в виде членов класса. Чтобы гарантировать последовательное размещение членов в памяти в порядке их появления, применяется атрибут StructLayoutAttribute .

Класс `NativeMethods` содержит управляемый прототип метода `GetSystemTime`, который по умолчанию передает класс `SystemTime` в качестве параметра In или Out. Этот параметр должен быть объявлен с атрибутами InAttribute и OutAttribute, так как классы, являющиеся ссылочными типами, по умолчанию передаются как параметры In. Чтобы вызывающий объект получал результаты, необходимо явным образом применить атрибуты направления. Класс `App` создает экземпляр класса `SystemTime` и осуществляет доступ к его полям данных.

## Примеры кода

C#

```
using System;
using System.Runtime.InteropServices;

[StructLayout(LayoutKind.Sequential)]
public class SystemTime
{
    public ushort year;
    public ushort month;
    public ushort weekday;
    public ushort day;
    public ushort hour;
    public ushort minute;
    public ushort second;
    public ushort millisecond;
}

internal static class NativeMethods
{
    // Declares a managed prototype for the unmanaged function using
Platform Invoke.
    [DllImport("Kernel32.dll")]
    internal static extern void GetSystemTime([In, Out] SystemTime st);
}

public class App
```

```
{
    public static void Main()
    {
        Console.WriteLine("C# SysTime Sample using Platform Invoke");
        SystemTime st = new SystemTime();
        NativeMethods.GetSystemTime(st);
        Console.Write("The Date is: ");
        Console.Write($"{st.month} {st.day} {st.year}");
    }
}

// The program produces output similar to the following:
//
// C# SysTime Sample using Platform Invoke
// The Date is: 3 21 2010
```

# Пример OutArrayOfStructs

В этом примере показано, как передать в неуправляемую функцию массив структур, содержащий целочисленные значения и строки, в виде параметров Out.

В этом примере демонстрируется, как вызывать собственную функцию с помощью класса Marshal и небезопасного кода.

В примере используются функции-оболочки и вызовы неуправляемого кода, определенные в библиотеке PinvokeLib.dll и содержащиеся в исходных файлах. В нем используется функция `TestOutArrayOfStructs` и структура `MYSTRSTRUCT2`. Структура содержит следующие элементы:

C++

```
typedef struct _MYSTRSTRUCT2
{
   char* buffer;
   UINT size;
} MYSTRSTRUCT2;
```

Класс `MyStruct` содержит строковый объект из символов ANSI. Поле CharSet определяет формат ANSI. `MyUnsafeStruct` — это структура, содержащая тип IntPtr вместо строки.

Класс `NativeMethods` содержит перегруженный метод прототипа `TestOutArrayOfStructs`. Если в качестве параметра метод объявляет указатель, класс должен быть помечен зарезервированным словом `unsafe`. Так как Visual Basic не

может использовать небезопасный код, перегруженный метод, модификатор unsafe и структуры `MyUnsafeStruct` не нужны.

Класс `App` реализует метод `UsingMarshaling`, который выполняет все задачи, необходимые для передачи массива. Чтобы указать, что данные передаются от вызываемого объекта к вызывающему, массив помечается зарезервированным словом `out` (`ByRef` в Visual Basic). Реализация использует следующие методы класса Marshal:

- метод PtrToStructure для маршалинга данных из неуправляемого буфера в управляемый объект;
- метод DestroyStructure для освобождения памяти, зарезервированной для строк структуры;
- метод FreeCoTaskMem для освобождения памяти, зарезервированной для массива.

Как упоминалось ранее, C# допускает небезопасный код, который Visual Basic не поддерживает. В примере кода C# `UsingUnsafePointer` является альтернативной реализацией метода, в которой для обратной передачи массива, содержащего структуру `MyUnsafeStruct`, вместо класса Marshal используются указатели.

## Объявление прототипов

C#

```
// Declares a class member for each structure element.
[StructLayout(LayoutKind.Sequential, CharSet = CharSet.Ansi)]
public class MyStruct
{
    public string buffer;
    public int size;
}

// Declares a structure with a pointer.
[StructLayout(LayoutKind.Sequential)]
public struct MyUnsafeStruct
{
    public IntPtr buffer;
    public int size;
}

internal static unsafe class NativeMethods
{
    // Declares managed prototypes for the unmanaged function.
    [DllImport("..\\LIB\\PInvokeLib.dll")]
```

```
    internal static extern void TestOutArrayOfStructs(
        out int size, out IntPtr outArray);

    [DllImport("..\\LIB\\PInvokeLib.dll")]
    internal static extern void TestOutArrayOfStructs(
        out int size, MyUnsafeStruct** outArray);
}
```

## Вызов функций

C#

```
public class App
{
    public static void Main()
    {
        Console.WriteLine("\nUsing marshal class\n");
        UsingMarshaling();
        Console.WriteLine("\nUsing unsafe code\n");
        UsingUnsafePointer();
    }

    public static void UsingMarshaling()
    {
        int size;
        IntPtr outArray;

        NativeMethods.TestOutArrayOfStructs(out size, out outArray);
        MyStruct[] manArray = new MyStruct[size];
        IntPtr current = outArray;
        for (int i = 0; i < size; i++)
        {
            manArray[i] = new MyStruct();
            Marshal.PtrToStructure(current, manArray[i]);

            //Marshal.FreeCoTaskMem((IntPtr)Marshal.ReadInt32(current));
            Marshal.DestroyStructure(current, typeof(MyStruct));
            current = (IntPtr)((long)current + Marshal.SizeOf(manArray[i]));

            Console.WriteLine("Element {0}: {1} {2}", i, manArray[i].buffer,
                manArray[i].size);
        }

        Marshal.FreeCoTaskMem(outArray);
    }

    public static unsafe void UsingUnsafePointer()
    {
        int size;
        MyUnsafeStruct* pResult;

        NativeMethods.TestOutArrayOfStructs(out size, &pResult);
```

```
        MyUnsafeStruct* pCurrent = pResult;
        for (int i = 0; i < size; i++, pCurrent++)
        {
            Console.WriteLine("Element {0}: {1} {2}", i,
                Marshal.PtrToStringAnsi(pCurrent->buffer), pCurrent->size);
            Marshal.FreeCoTaskMem(pCurrent->buffer);
        }

        Marshal.FreeCoTaskMem((IntPtr)pResult);
    }
}
```

## См. также

- Маршалирование данных с помощью вызова платформы
- Маршалинг строк
- Маршалирование различных типов массивов

# Маршаллирование различных типов массивов

17.06.2025

Массив — это ссылочный тип в управляемом коде, который содержит один или несколько элементов одного типа. Хотя массивы являются ссылочными типами, они передаются в качестве параметров в неуправляемые функции. Это поведение не соответствует тому, как управляемые массивы передаются в управляемые объекты в качестве входных/выходных параметров. Дополнительные сведения см. в разделе "Копирование и закрепление".

В следующей таблице перечислены параметры маршаллинга для массивов и описывается их использование.

Развернуть таблицу

| Массив | Описание |
|---|---|
| Целочисленные числа по значению. | Передает массив целых чисел в качестве параметра In. |
| Целочисленные числа по ссылке. | Передает массив целых чисел в качестве параметра In/Out. |
| Целые числа по значению (двумерные). | Передает матрицу целых чисел в качестве параметра In. |
| Строки по значению. | Передает массив строк в качестве параметра In. |
| Структуры с целыми числами. | Передает массив структур, содержащих целые числа в качестве параметра In. |
| Структуры со строками. | Передает массив структур, содержащих только строки в качестве параметра In/Out. Элементы массива можно изменять. |

# Пример

В этом примере показано, как передать следующие типы массивов:

- Массив целых чисел по значению.
- Массив целых чисел по ссылке, размер которого можно изменить.
- Многомерный массив (матрица) целых чисел по значению.

- Массив строк по значению.
- Массив структур с целыми числами.
- Массив структур со строками.

Если массив не маршалируется явно по ссылке, поведение по умолчанию маршалирует массив в качестве параметра In. Вы можете изменить это поведение, явно применяя атрибуты InAttribute и OutAttribute.

В примере массивов используются следующие неуправляемые функции, показанные с их оригинальными объявлениями функций:

- **TestArrayOfInts** экспортируется из PinvokeLib.dll.

```
C++

int TestArrayOfInts(int* pArray, int pSize);
```

- **TestRefArrayOfInts** экспортируется из PinvokeLib.dll.

```
C++

int TestRefArrayOfInts(int** ppArray, int* pSize);
```

- **TestMatrixOfInts** экспортируется из PinvokeLib.dll.

```
C++

int TestMatrixOfInts(int pMatrix[][COL_DIM], int row);
```

- **TestArrayOfStrings** экспортируется из PinvokeLib.dll.

```
C++

int TestArrayOfStrings(char** ppStrArray, int size);
```

- **TestArrayOfStructs** экспортируется из PinvokeLib.dll.

```
C++

int TestArrayOfStructs(MYPOINT* pPointArray, int size);
```

- **TestArrayOfStructs2** экспортируется из PinvokeLib.dll.

```
C++

int TestArrayOfStructs2 (MYPERSON* pPersonArray, int size);
```

PinvokeLib.dll — это пользовательская неуправляемая библиотека, содержащая реализации для ранее перечисленных функций и двух переменных структуры , **MYPOINT** и**MYPERSON**. Структуры содержат следующие элементы:

```
C++

typedef struct _MYPOINT
{
   int x;
   int y;
} MYPOINT;

typedef struct _MYPERSON
{
   char* first;
   char* last;
} MYPERSON;
```

В этом примере структуры `MyPoint` и `MyPerson` содержат внедренные типы. Атрибут StructLayoutAttribute задан, чтобы гарантировать последовательное размещение членов в памяти в порядке их появления.

Класс `NativeMethods` содержит набор методов, вызываемых классом `App` . Более подробные сведения о передаче массивов см. в комментариях к следующему примеру. Массив, являющийся ссылочным типом, передается в качестве параметра In по умолчанию. Чтобы вызывающий объект получил результаты, **InAttribute** и **OutAttribute** должны применяться явным образом к аргументу, содержащего массив.

## Объявление прототипов

```
C#

// Declares a managed structure for each unmanaged structure.
[StructLayout(LayoutKind.Sequential)]
public struct MyPoint
{
    public int X;
    public int Y;

    public MyPoint(int x, int y)
    {
        this.X = x;
        this.Y = y;
```

```
    }
}

[StructLayout(LayoutKind.Sequential, CharSet = CharSet.Ansi)]
public struct MyPerson
{
    public string First;
    public string Last;

    public MyPerson(string first, string last)
    {
        this.First = first;
        this.Last = last;
    }
}

internal static class NativeMethods
{
    // Declares a managed prototype for an array of integers by value.
    // The array size cannot be changed, but the array is copied back.
    [DllImport("..\\LIB\\PinvokeLib.dll", CallingConvention =
CallingConvention.Cdecl)]
    internal static extern int TestArrayOfInts(
        [In, Out] int[] array, int size);

    // Declares a managed prototype for an array of integers by reference.
    // The array size can change, but the array is not copied back
    // automatically because the marshaler does not know the resulting size.
    // The copy must be performed manually.
    [DllImport("..\\LIB\\PinvokeLib.dll", CallingConvention =
CallingConvention.Cdecl)]
    internal static extern int TestRefArrayOfInts(
        ref IntPtr array, ref int size);

    // Declares a managed prototype for a matrix of integers by value.
    [DllImport("..\\LIB\\PinvokeLib.dll", CallingConvention =
CallingConvention.Cdecl)]
    internal static extern int TestMatrixOfInts(
        [In, Out] int[,] pMatrix, int row);

    // Declares a managed prototype for an array of strings by value.
    [DllImport("..\\LIB\\PinvokeLib.dll", CallingConvention =
CallingConvention.Cdecl)]
    internal static extern int TestArrayOfstrings(
        [In, Out] string[] stringArray, int size);

    // Declares a managed prototype for an array of structures with integers.
    [DllImport("..\\LIB\\PinvokeLib.dll", CallingConvention =
CallingConvention.Cdecl)]
    internal static extern int TestArrayOfStructs(
        [In, Out] MyPoint[] pointArray, int size);

    // Declares a managed prototype for an array of structures with strings.
    [DllImport("..\\LIB\\PinvokeLib.dll", CallingConvention =
CallingConvention.Cdecl)]
```

```
    internal static extern int TestArrayOfStructs2(
        [In, Out] MyPerson[] personArray, int size);
}
```

## Вызов функций

C#

```
public class App
{
    public static void Main()
    {
        // array ByVal
        int[] array1 = new int[10];
        Console.WriteLine("Integer array passed ByVal before call:");
        for (int i = 0; i < array1.Length; i++)
        {
            array1[i] = i;
            Console.Write(" " + array1[i]);
        }

        int sum1 = NativeMethods.TestArrayOfInts(array1, array1.Length);
        Console.WriteLine("\nSum of elements:" + sum1);
        Console.WriteLine("\nInteger array passed ByVal after call:");

        foreach (int i in array1)
        {
            Console.Write(" " + i);
        }

        // array ByRef
        int[] array2 = new int[10];
        int size = array2.Length;
        Console.WriteLine("\n\nInteger array passed ByRef before call:");
        for (int i = 0; i < array2.Length; i++)
        {
            array2[i] = i;
            Console.Write(" " + array2[i]);
        }

        IntPtr buffer = Marshal.AllocCoTaskMem(Marshal.SizeOf(size)
           * array2.Length);
        Marshal.Copy(array2, 0, buffer, array2.Length);

        int sum2 = NativeMethods.TestRefArrayOfInts(ref buffer, ref size);
        Console.WriteLine("\nSum of elements:" + sum2);
        if (size > 0)
        {
            int[] arrayRes = new int[size];
            Marshal.Copy(buffer, arrayRes, 0, size);
            Marshal.FreeCoTaskMem(buffer);
            Console.WriteLine("\nInteger array passed ByRef after call:");
```

```
        foreach (int i in arrayRes)
        {
            Console.Write(" " + i);
        }
    }
    else
    {
        Console.WriteLine("\nArray after call is empty");
    }

    // matrix ByVal
    const int DIM = 5;
    int[,] matrix = new int[DIM, DIM];

    Console.WriteLine("\n\nMatrix before call:");
    for (int i = 0; i < DIM; i++)
    {
        for (int j = 0; j < DIM; j++)
        {
            matrix[i, j] = j;
            Console.Write(" " + matrix[i, j]);
        }

        Console.WriteLine("");
    }

    int sum3 = NativeMethods.TestMatrixOfInts(matrix, DIM);
    Console.WriteLine("\nSum of elements:" + sum3);
    Console.WriteLine("\nMatrix after call:");
    for (int i = 0; i < DIM; i++)
    {
        for (int j = 0; j < DIM; j++)
        {
            Console.Write(" " + matrix[i, j]);
        }

        Console.WriteLine("");
    }

    // string array ByVal
    string[] strArray = { "one", "two", "three", "four", "five" };
    Console.WriteLine("\n\nstring array before call:");
    foreach (string s in strArray)
    {
        Console.Write(" " + s);
    }

    int lenSum = NativeMethods.TestArrayOfstrings(strArray, strArray.Length);
    Console.WriteLine("\nSum of string lengths:" + lenSum);
    Console.WriteLine("\nstring array after call:");
    foreach (string s in strArray)
    {
        Console.Write(" " + s);
    }
```

```
        // struct array ByVal
        MyPoint[] points = { new MyPoint(1, 1), new MyPoint(2, 2), new MyPoint(3,
3) };
        Console.WriteLine("\n\nPoints array before call:");
        foreach (MyPoint p in points)
        {
            Console.WriteLine($"X = {p.X}, Y = {p.Y}");
        }

        int allSum = NativeMethods.TestArrayOfStructs(points, points.Length);
        Console.WriteLine("\nSum of points:" + allSum);
        Console.WriteLine("\nPoints array after call:");
        foreach (MyPoint p in points)
        {
            Console.WriteLine($"X = {p.X}, Y = {p.Y}");
        }

        // struct with strings array ByVal
        MyPerson[] persons =
        {
            new MyPerson("Kim", "Akers"),
            new MyPerson("Adam", "Barr"),
            new MyPerson("Jo", "Brown")
        };

        Console.WriteLine("\n\nPersons array before call:");
        foreach (MyPerson pe in persons)
        {
            Console.WriteLine($"First = {pe.First}, Last = {pe.Last}");
        }

        int namesSum = NativeMethods.TestArrayOfStructs2(persons, persons.Length);
        Console.WriteLine("\nSum of name lengths:" + namesSum);
        Console.WriteLine("\n\nPersons array after call:");
        foreach (MyPerson pe in persons)
        {
            Console.WriteLine($"First = {pe.First}, Last = {pe.Last}");
        }
    }
}
```

## См. также

- Типы данных для вызова функций платформы
- Создание прототипов в управляемом коде

# Маршалинг делегата как метода обратного вызова

17.06.2025

В этом примере показано, как передать делегаты в неуправляемую функцию, ожидающую указателей функции. Делегат — это класс, который может содержать ссылку на метод и эквивалентен типа-безопасному указателю функции или функции обратного вызова.

> ⓘ **Примечание**
>
> При использовании делегата внутри вызова среда CLR защищает делегата от сборки мусора в течение этого вызова. Однако, если неуправляемая функция сохраняет делегат для использования после завершения вызова, то необходимо вручную предотвратить сборку мусора до тех пор, пока неуправляемая функция полностью не завершит свою работу с делегатом. Дополнительные сведения см. в примере HandleRef и GCHandle.

В примере обратного вызова используются следующие неуправляемые функции, показанные с их исходным объявлением функции:

- `TestCallBack` экспортируется из PinvokeLib.dll.

```
C++

void TestCallBack(FPTR pf, int value);
```

- `TestCallBack2` экспортируется из PinvokeLib.dll.

```
C++

void TestCallBack2(FPTR2 pf2, char* value);
```

PinvokeLib.dll — это пользовательская неуправляемая библиотека, содержащая реализацию для ранее перечисленных функций.

В этом примере `NativeMethods` класс содержит управляемые прототипы для `TestCallBack` методов и `TestCallBack2` методов. Оба метода передают делегат в функцию обратного вызова в качестве параметра. Подпись делегата должна соответствовать сигнатуре

метода, на который он ссылается. Например, у делегатов `FPtr` и `FPtr2` сигнатуры идентичны методам `DoSomething` и `DoSomething2`.

## Объявление прототипов

C#

```
public delegate bool FPtr(int value);
public delegate bool FPtr2(string value);

internal static class NativeMethods
{
    // Declares managed prototypes for unmanaged functions.
    [DllImport("..\\LIB\\PinvokeLib.dll", CallingConvention =
CallingConvention.Cdecl)]
    internal static extern void TestCallBack(FPtr cb, int value);

    [DllImport("..\\LIB\\PinvokeLib.dll", CallingConvention =
CallingConvention.Cdecl)]
    internal static extern void TestCallBack2(FPtr2 cb2, string value);
}
```

## Вызов функций

C#

```
public class App
{
    public static void Main()
    {
        FPtr cb = new FPtr(App.DoSomething);
        NativeMethods.TestCallBack(cb, 99);
        FPtr2 cb2 = new FPtr2(App.DoSomething2);
        NativeMethods.TestCallBack2(cb2, "abc");
    }

    public static bool DoSomething(int value)
    {
        Console.WriteLine($"\nCallback called with param: {value}");
        // ...
        return true;
    }

    public static bool DoSomething2(string value)
    {
        Console.WriteLine($"\nCallback called with param: {value}");
        // ...
        return true;
```

```
    }
}
```

## См. также

- Различные примеры маршаллинга
- Типы данных вызова платформы
- Создание прототипов в управляемом коде

# Маршаллирование данных с помощью компонентного взаимодействия COM

17.06.2025

COM-взаимодействие поддерживает как использование COM-объектов из управляемого кода, так и предоставление управляемых объектов в COM. Поддержка маршаллинга данных между COM и другими компонентами является высокой и почти всегда обеспечивает правильную обработку данных.

Пакет SDK для Windows включает следующие средства взаимодействия COM:

- Средство импорта библиотеки типов (Tlbimp.exe), которое преобразует библиотеку типов COM в сборку для взаимодействия. Из этой сборки служба маршаллинга для взаимодействия создает оболочки, которые выполняют маршалирование данных между управляемой и неуправляемой памятью.
- Экспортер библиотеки типов (Tlbexp.exe), который создает библиотеку типов COM из сборки и создает оболочку, которая выполняет маршаллирование во время вызовов методов.

В следующих разделах приведены ссылки на темы, которые описывают процессы настройки оболочек для интероперабельности, когда вы можете (или обязаны) предоставить маршализатору дополнительные сведения о типах.

## В этом разделе

Практическое руководство. Создание оболочки вручную Описывает создание COM-оболочки вручную в управляемом исходном коде.

Как это сделать: Перенос Managed-Code DCOM на WCF
Описывает перенос управляемого кода DCOM в WCF для наиболее безопасного решения.

## Связанные разделы

Типы данных COM
Предоставляет соответствующие управляемые и неуправляемые типы данных.

Настройка вызываемых оболочек COM
Описывает, как явно обрабатывать типы данных с помощью атрибута MarshalAsAttribute в процессе разработки.

### Настройка вызываемых оболочек среды выполнения

Описывается, как настроить поведение маршаллинга типов в сборке взаимодействия, а также как вручную определить типы COM.

### Расширенное взаимодействие COM

Содержит ссылки на дополнительные сведения о включении COM-компонентов в приложение .NET Framework.

### Общие сведения о преобразовании сборки в библиотеку типов

Описывает процесс преобразования для экспорта сборки в библиотеку типов.

### Сводка о преобразовании библиотеки типов в сборку

Описывает библиотеку типов для процесса преобразования импорта сборки.

### Взаимодействие с использованием универсальных типов

Описывает, какие действия поддерживаются при использовании универсальных типов для взаимодействия COM.

# Практическое руководство. Создание оболочек вручную

Статья • 07.04.2023

Если вы решили объявлять типы COM в управляемом исходном коде вручную, лучше всего начать с существующего файла языка IDL или библиотеки типов. Если у вас нет файла IDL или вы не можете создать файл библиотеки типов, можно имитировать типы COM, создав управляемые объявления и экспортировав получившуюся сборку в библиотеку типов.

## Имитация типов COM из управляемого источника

1. Объявите типы на языке, совместимом со спецификацией CLS, и скомпилируйте файл.

2. Экспортируйте сборку, содержащую типы, с помощью программы экспорта библиотек типов (Tlbexp.exe).

3. Экспортированная библиотека типов COM используется в качестве основы для объявления управляемых типов, ориентированных на COM.

## Создание вызываемой оболочки времени выполнения

1. Если у вас есть IDL-файл или файл библиотеки типов, решите, какие классы и интерфейсы нужно включить в пользовательскую вызываемую оболочку времени выполнения. Вы можете исключить любые типы, которые не будут использоваться в приложении прямо или косвенно.

2. Создайте файл исходного кода на языке, совместимом с CLS, и объявите типы. Полное описание процедуры преобразования при импорте см. в разделе Обзор преобразования библиотеки типов в сборку. Фактически при создании пользовательской вызываемой оболочки времени выполнения вы вручную выполняете все операции по преобразованию типов, предоставляемые программой импорта библиотек типов (Tlbimp.exe). В примере в следующем разделе показаны типы в файле IDL или файле библиотеки типов и соответствующие типы в коде C#.

3. После завершения объявлений следует выполнить компиляцию этого файла так же, как и компиляцию любого другого управляемого исходного кода.

4. Как и в случае с типами, импортированными с помощью программы Tlbimp.exe, для некоторых типов требуются дополнительные сведения, которые можно добавить непосредственно в код. Подробную информацию см. в разделе Практическое руководство. Изменение сборок взаимодействия.

# Пример

Ниже приведен пример кода интерфейса `ISATest` и класса `SATest` в IDL вместе с соответствующими типами в исходном коде C#.

### Файл IDL или файл библиотеки типов

C++

```
 [
object,
uuid(40A8C65D-2448-447A-B786-64682CBEF133),
dual,
helpstring("ISATest Interface"),
pointer_default(unique)
 ]
interface ISATest : IDispatch
 {
[id(1), helpstring("method InSArray")]
HRESULT InSArray([in] SAFEARRAY(int) *ppsa, [out,retval] int *pSum);
 };
 [
uuid(116CCA1E-7E39-4515-9849-90790DA6431E),
helpstring("SATest Class")
 ]
coclass SATest
 {
  [default] interface ISATest;
 };
```

### Оболочка в управляемом исходном коде

C#

```
using System;
using System.Runtime.InteropServices;
using System.Runtime.CompilerServices;

[assembly:Guid("E4A992B8-6F5C-442C-96E7-C4778924C753")]
[assembly:ImportedFromTypeLib("SAServerLib")]
namespace SAServer
{
 [ComImport]
 [Guid("40A8C65D-2448-447A-B786-64682CBEF133")]
```

```
 [TypeLibType(TypeLibTypeFlags.FLicensed)]
 public interface ISATest
 {
  [DispId(1)]
  //[MethodImpl(MethodImplOptions.InternalCall,
  // MethodCodeType=MethodCodeType.Runtime)]
  int InSArray( [MarshalAs(UnmanagedType.SafeArray,
      SafeArraySubType=VarEnum.VT_I4)] ref int[] param );
 }
 [ComImport]
 [Guid("116CCA1E-7E39-4515-9849-90790DA6431E")]
 [ClassInterface(ClassInterfaceType.None)]
 [TypeLibType(TypeLibTypeFlags.FCanCreate)]
 public class SATest : ISATest
 {
  [DispId(1)]
  [MethodImpl(MethodImplOptions.InternalCall,
  MethodCodeType=MethodCodeType.Runtime)]
  extern int ISATest.InSArray( [MarshalAs(UnmanagedType.SafeArray,
  SafeArraySubType=VarEnum.VT_I4)] ref int[] param );
 }
}
```

# См. также

- Настройка вызываемых оболочек времени выполнения
- Типы данных COM
- Практическое руководство. Изменение сборок взаимодействия
- Общие сведения о преобразовании библиотеки типов в сборку
- Tlbimp.exe (программа экспорта библиотек типов)
- Tlbexp.exe (программа экспорта библиотек типов)

# Как перенести DCOM с управляемым кодом на WCF

Windows Communication Foundation (WCF) — это рекомендуемый и безопасный выбор распределенной объектной модели компонентов (DCOM) для вызовов управляемого кода между серверами и клиентами в распределенной среде. В этой статье показано, как перенести код из DCOM в WCF для следующих сценариев.

- Удаленная служба возвращает клиенту объект по значению
- Клиент отправляет объект по значению в удаленный сервис.
- Удаленная служба возвращает объект по ссылке клиенту.

По соображениям безопасности отправка объекта по ссылке от клиента в службу не допускается в WCF. Сценарий, требующий беседы между клиентом и сервером, можно достичь в WCF с помощью дуплексной службы. Дополнительные сведения о дуплексных службах см. в разделе "Дуплексные службы".

Дополнительные сведения о создании служб и клиентов WCF для этих служб см. в статье "Базовый программирование WCF", "Проектирование и реализация служб" и "Создание клиентов".

## Пример кода DCOM

Для этих сценариев интерфейсы DCOM, которые иллюстрируются с помощью WCF, имеют следующую структуру:

C#

```
[ComVisible(true)]
[Guid("AA9C4CDB-55EA-4413-90D2-843F1A49E6E6")]
public interface IRemoteService
{
   Customer GetObjectByValue();
   IRemoteObject GetObjectByReference();
   void SendObjectByValue(Customer customer);
}

[ComVisible(true)]
[Guid("A12C98DE-B6A1-463D-8C24-81E4BBC4351B")]
public interface IRemoteObject
{
}

public class Customer
```

```
{
}
```

# Служба возвращает объект по значению

В этом сценарии вы вызываете службу, метод которой возвращает объект, который передается по значению с сервера клиенту. Этот сценарий представляет следующий вызов COM:

C#

```
public interface IRemoteService
{
    Customer GetObjectByValue();
}
```

В этом сценарии клиент получает десериализированную копию объекта из удаленной службы. Клиент может взаимодействовать с этой локальной копией без вызова службы. Другими словами, клиенту гарантируется, что служба не будет вовлечена каким-либо образом при вызове методов на локальной копии. WCF всегда возвращает объекты из службы по значению, поэтому следующие шаги описывают создание обычной службы WCF.

## Шаг 1. Определение интерфейса службы WCF

Определите общедоступный интерфейс для службы WCF и пометьте его атрибутом [ServiceContractAttribute] . Пометьте методы, которые необходимо предоставить клиентам с помощью атрибута [OperationContractAttribute]. В следующем примере показано использование этих атрибутов для идентификации серверного интерфейса и методов интерфейса, которые клиент может вызывать. Метод, используемый для этого сценария, показан полужирным шрифтом.

C#

```
using System.Runtime.Serialization;
using System.ServiceModel;
using System.ServiceModel.Web;
. . .
[ServiceContract]
public interface ICustomerManager
{
    [OperationContract]
    void StoreCustomer(Customer customer);

    [OperationContract]     Customer GetCustomer(string firstName, string
```

```
lastName);

}
```

## Шаг 2. Определение контракта данных

Затем необходимо создать контракт данных для службы, который будет описывать, как данные будут обмениваться между службой и ее клиентами. Классы, описанные в контракте данных, должны быть помечены атрибутом [DataContractAttribute] . Отдельные свойства или поля, которые должны отображаться для клиента и сервера, должны быть помечены атрибутом [DataMemberAttribute] . Если вы хотите, чтобы типы, производные от класса в контракте данных, были разрешены, необходимо определить их с атрибутом [KnownTypeAttribute] . WCF будет сериализовать или десериализовать типы, включенные в интерфейс службы, и типы, определенные как известные. Если вы пытаетесь использовать тип, который не является известным типом, возникнет исключение.

Дополнительные сведения о контрактах данных см. в разделе "Контракты данных".

C#

```
[DataContract]
[KnownType(typeof(PremiumCustomer))]
public class Customer
{
    [DataMember]
    public string Firstname;
    [DataMember]
    public string Lastname;
    [DataMember]
    public Address DefaultDeliveryAddress;
    [DataMember]
    public Address DefaultBillingAddress;
}
 [DataContract]
public class PremiumCustomer : Customer
{
    [DataMember]
    public int AccountID;
}

 [DataContract]
public class Address
{
    [DataMember]
    public string Street;
    [DataMember]
    public string Zipcode;
    [DataMember]
    public string City;
```

```
    [DataMember]
    public string State;
    [DataMember]
    public string Country;
}
```

## Шаг 3. Реализация службы WCF

Затем необходимо реализовать класс службы WCF, который реализует интерфейс, определенный на предыдущем шаге.

C#

```
public class CustomerService: ICustomerManager
{
    public void StoreCustomer(Customer customer)
    {
        // write to a database
    }
    public Customer GetCustomer(string firstName, string lastName)
    {
        // read from a database
    }
}
```

## Шаг 4. Настройка службы и клиента

Чтобы запустить службу WCF, необходимо объявить конечную точку, которая предоставляет этот интерфейс службы по определенному URL-адресу с помощью определенной привязки WCF. Привязка задает сведения о транспорте, кодировке и протоколе для клиентов и сервера для обмена данными. Обычно в файл конфигурации проекта службы добавляются привязки (web.config). Ниже представлена запись привязки для службы-примера:

XML

```
<configuration>
  <system.serviceModel>
    <services>
      <service name="Server.CustomerService">
        <endpoint address="http://localhost:8083/CustomerManager"
                  binding="basicHttpBinding"
                  contract="Shared.ICustomerManager" />
      </service>
    </services>
```

```
  </system.serviceModel>
</configuration>
```

Затем необходимо настроить клиент для сопоставления сведений о привязке, указанных службой. Для этого добавьте следующий код в файл конфигурации приложения клиента (app.config).

XML

```
<configuration>
  <system.serviceModel>
    <client>
      <endpoint name="customermanager"
                address="http://localhost:8083/CustomerManager"
                binding="basicHttpBinding"
                contract="Shared.ICustomerManager"/>
    </client>
  </system.serviceModel>
</configuration>
```

## Шаг 5. Запуск службы

В завершение, вы можете самостоятельно разместить этот сервис в консольном приложении, добавив следующие строки в приложение-службу, а затем запустив его. Для получения более подробной информации о других способах размещения приложений служб WCF см. Размещение служб.

C#

```
ServiceHost customerServiceHost = new ServiceHost(typeof(CustomerService));
customerServiceHost.Open();
```

## Шаг 6. Вызов службы от клиента

Чтобы вызвать службу с клиента, необходимо создать фабрику каналов для службы и запросить канал, который позволит напрямую вызывать метод `GetCustomer` из клиента. Канал реализует интерфейс службы и обрабатывает базовую логику запроса и ответа. Возвращаемое значение из этого вызова метода — десериализированная копия ответа службы.

C#

```
ChannelFactory<ICustomerManager> factory =
     new ChannelFactory<ICustomerManager>("customermanager");
```

```
ICustomerManager service = factory.CreateChannel();
Customer customer = service.GetCustomer("Mary", "Smith");
```

# Клиент отправляет объект по значению на сервер.

В этом сценарии клиент отправляет объект серверу по значению. Это означает, что сервер получит десериализированную копию объекта. Сервер может вызывать методы для этой копии и гарантировать отсутствие обратного вызова в клиентский код. Как упоминалось ранее, обычные обмены данными WCF осуществляются по значению. Это гарантирует, что вызов методов для одного из этих объектов выполняется только локально. Он не будет вызывать код на клиенте.

Этот сценарий представляет следующий вызов метода COM:

**C#**

```
public interface IRemoteService
{
    void SendObjectByValue(Customer customer);
}
```

В этом сценарии используется тот же интерфейс службы и контракт данных, что и в первом примере. Кроме того, клиент и служба будут настроены таким же образом. В этом примере канал создается для отправки объекта и выполнения таким же образом. Однако в этом примере вы создадите клиент, который вызывает сервис, передав объект по значению. Метод службы, который клиент будет вызывать в контракте службы, отображается полужирным шрифтом:

**C#**

```
[ServiceContract]
public interface ICustomerManager
{
    [OperationContract]     void StoreCustomer(Customer customer);

    [OperationContract]
    Customer GetCustomer(string firstName, string lastName);
}
```

## Добавьте код клиенту, который отправляет объект по значению

В следующем коде показано, как клиент создает новый объект клиента по значению, создает канал для взаимодействия со `ICustomerManager` службой и отправляет в него объект клиента.

Объект клиента будет сериализован и отправлен в службу, где она десериализирована службой в новую копию этого объекта. Все методы, вызывающие службу этого объекта, будут выполняться только локально на сервере. Важно отметить, что этот код иллюстрирует отправку производного типа (`PremiumCustomer`). Контракт обслуживания ожидает объект `Customer`, но контракт на данные обслуживания использует атрибут [KnownTypeAttribute] для указания, что `PremiumCustomer` также допускается. WCF не сможет выполнить попытки сериализации или десериализации любого другого типа через этот интерфейс службы.

**C#**

```
PremiumCustomer customer = new PremiumCustomer();
customer.Firstname = "John";
customer.Lastname = "Doe";
customer.DefaultBillingAddress = new Address();
customer.DefaultBillingAddress.Street = "One Microsoft Way";
customer.DefaultDeliveryAddress = customer.DefaultBillingAddress;
customer.AccountID = 42;

ChannelFactory<ICustomerManager> factory =
   new ChannelFactory<ICustomerManager>("customermanager");
ICustomerManager customerManager = factory.CreateChannel();
customerManager.StoreCustomer(customer);
```

# Сервис возвращает объект по ссылке

В этом сценарии клиентское приложение вызывает удаленную службу, а метод возвращает объект, который передается по ссылке из службы клиенту.

Как упоминалось ранее, службы WCF всегда возвращают объект по значению. Однако вы можете добиться аналогичного результата, используя класс EndpointAddress10. Сериализуемый по значению объект EndpointAddress10 может использоваться клиентом для получения сеансового объекта по ссылке на сервере.

Поведение объекта по ссылке в WCF, показанное в этом сценарии, отличается от поведения в DCOM. В DCOM сервер может напрямую возвращать объект по ссылке клиенту, а клиент может вызывать методы этого объекта, которые выполняются на сервере. Однако в WCF возвращаемый объект всегда передаётся по значению. Клиент должен принять этот объект по значению, представленный EndpointAddress10 и использовать его для создания собственного сеансового объекта по ссылке. Клиентский

метод вызывает сеансовый объект, выполняемый на сервере. Другими словами, этот ссылочный объект в WCF — это обычная служба WCF, настроенная на сеансовое взаимодействие.

В WCF сеанс — это способ сопоставления нескольких сообщений, отправленных между двумя конечными точками. Это означает, что после подключения к этой службе между клиентом и сервером будет установлена сессия. Клиент будет использовать один уникальный экземпляр объекта на стороне сервера для всех его взаимодействий в рамках этого одного сеанса. Контракты сеансового WCF похожи на шаблоны сетевых запросов и ответов, ориентированных на подключение.

Этот сценарий представлен следующим методом DCOM.

C#

```
public interface IRemoteService
{
    IRemoteObject GetObjectByReference();
}
```

## Шаг 1. Определите интерфейс и реализацию службы WCF с поддержкой сеансов

Сначала определите интерфейс службы WCF, содержащий объект с поддержкой сессий.

В этом коде сеансовый объект помечается атрибутом `ServiceContract`, который идентифицирует его как обычный интерфейс WCF. Кроме того, свойство SessionMode установлено на указание, что это будет служба сессий.

C#

```
[ServiceContract(SessionMode = SessionMode.Allowed)]
public interface ISessionBoundObject
{
    [OperationContract]
    string GetCurrentValue();

    [OperationContract]
    void SetCurrentValue(string value);
}
```

В следующем коде показана реализация службы.

Служба помечена атрибутом [ServiceBehavior], а свойство InstanceContextMode имеет значение InstanceContextMode.PerSessions, чтобы указать, что для каждого сеанса

необходимо создать уникальный экземпляр этого типа.

C#

```
[ServiceBehavior(InstanceContextMode = InstanceContextMode.PerSession)]
    public class MySessionBoundObject : ISessionBoundObject
    {
        private string _value;

        public string GetCurrentValue()
        {
            return _value;
        }

        public void SetCurrentValue(string val)
        {
            _value = val;
        }

    }
```

## Шаг 2. Определение службы фабричного WCF для сеансового объекта

Служба, создающая сессионный объект, должна быть определена и реализована. В следующем коде показано, как это сделать. Этот код создает другую EndpointAddress10 службу WCF, которая возвращает объект. Это сериализованная форма конечной точки, которая может быть использована для создания сессионного объекта.

C#

```
[ServiceContract]
    public interface ISessionBoundFactory
    {
        [OperationContract]
        EndpointAddress10 GetInstanceAddress();
    }
```

Ниже приведена реализация этой службы. Эта реализация поддерживает фабрику однотонных каналов для создания сеансовых объектов. При `GetInstanceAddress` вызове он создает канал и создает объект, указывающий EndpointAddress10 на удаленный адрес, связанный с этим каналом. EndpointAddress10 — это тип данных, который можно вернуть клиенту в виде значения.

C#

```
public class SessionBoundFactory : ISessionBoundFactory
    {
        public static ChannelFactory<ISessionBoundObject> _factory =
            new ChannelFactory<ISessionBoundObject>("sessionbound");

        public SessionBoundFactory()
        {
        }

        public EndpointAddress10 GetInstanceAddress()
        {
            IClientChannel channel = (IClientChannel)_factory.CreateChannel();
            return EndpointAddress10.FromEndpointAddress(channel.RemoteAddress);
        }
    }
```

## Шаг 3. Настройка и запуск служб WCF

Чтобы разместить эти службы, необходимо внести следующие дополнения в файл конфигурации сервера (web.config).

1. Добавьте раздел `<client>`, который описывает конечную точку для объекта с поддержкой сеансов. В этом сценарии сервер также выступает в качестве клиента и должен быть настроен для включения этого.

2. В разделе `<services>` объявите конечные точки службы для фабрики и объект, поддерживающий сеансы. Это позволяет клиенту взаимодействовать с конечными точками службы, получать EndpointAddress10 и создавать сеансовый канал.

Ниже приведен пример файла конфигурации с этими параметрами:

XML

```
<configuration>
  <system.serviceModel>
    <client>
      <endpoint name="sessionbound"
                address="net.tcp://localhost:8081/SessionBoundObject"
                binding="netTcpBinding"
                contract="Shared.ISessionBoundObject"/>
    </client>

    <services>
      <service name="Server.MySessionBoundObject">
        <endpoint address="net.tcp://localhost:8081/SessionBoundObject"
                  binding="netTcpBinding"
                  contract="Shared.ISessionBoundObject" />
      </service>
      <service name="Server.SessionBoundFactory">
```

```
        <endpoint address="net.tcp://localhost:8081/SessionBoundFactory"
                  binding="netTcpBinding"
                  contract="Shared.ISessionBoundFactory" />
      </service>
    </services>
  </system.serviceModel>
</configuration>
```

Добавьте следующие строки в консольное приложение, чтобы самостоятельно разместить службу и запустить приложение.

C#

```
ServiceHost factoryHost = new ServiceHost(typeof(SessionBoundFactory));
factoryHost.Open();

ServiceHost sessionBoundServiceHost = new ServiceHost(
typeof(MySessionBoundObject));
sessionBoundServiceHost.Open();
```

## Шаг 4. Настройка клиента и вызов службы

Настройте клиент для взаимодействия со службами WCF, выполнив следующие записи в файле конфигурации приложения проекта (app.config).

XML

```
<configuration>
  <system.serviceModel>
    <client>
      <endpoint name="sessionbound"
                address="net.tcp://localhost:8081/SessionBoundObject"
                binding="netTcpBinding"
                contract="Shared.ISessionBoundObject"/>
      <endpoint name="factory"
                address="net.tcp://localhost:8081/SessionBoundFactory"
                binding="netTcpBinding"
                contract="Shared.ISessionBoundFactory"/>
    </client>
  </system.serviceModel>
</configuration>
```

Чтобы вызвать службу, добавьте код клиенту, чтобы выполнить следующие действия:

1. Создайте канал к `ISessionBoundFactory` службе.
2. Используйте канал для вызова `ISessionBoundFactory` службы и получения EndpointAddress10 объекта.

3. Используйте EndpointAddress10 для создания канала, чтобы получить сеансовый объект.

4. Вызовите методы `SetCurrentValue` и `GetCurrentValue`, чтобы продемонстрировать, что в нескольких вызовах используется один и тот же экземпляр объекта.

C#

```
ChannelFactory<ISessionBoundFactory> factory =
        new ChannelFactory<ISessionBoundFactory>("factory");

ISessionBoundFactory sessionBoundFactory = factory.CreateChannel();

EndpointAddress10 address = sessionBoundFactory.GetInstanceAddress();

ChannelFactory<ISessionBoundObject> sessionBoundObjectFactory =
    new ChannelFactory<ISessionBoundObject>(
        new NetTcpBinding(),
        address.ToEndpointAddress());

ISessionBoundObject sessionBoundObject =
        sessionBoundObjectFactory.CreateChannel();

sessionBoundObject.SetCurrentValue("Hello");
if (sessionBoundObject.GetCurrentValue() == "Hello")
{
    Console.WriteLine("Session-full instance management works as expected");
}
```

# См. также

- базовое программирование WCF
- Проектирование и реализация служб
- Создание клиентов
- Услуги дуплексной печати

---

Last updated on 31.03.2026

# Практическое руководство. Сопоставление значений HRESULT и исключений

Статья • 05.03.2024

Методы COM сообщают об ошибках, возвращая значения HRESULT. Методы .NET в этом случае вызывают исключения. Среда выполнения обеспечивает сопоставление этих двух элементов. Каждый класс платформы .NET Framework сопоставляется со значением HRESULT.

Соответствующие значения HRESULT задаются определяемыми пользователем классами исключений. Эти классы исключений могут динамически изменять объект HRESULT, возвращаемый при создании исключения, задав `HResult` поле для объекта исключения. Дополнительные сведения об исключении предоставляются клиенту через `IErrorInfo` интерфейс, который реализуется в объекте .NET в неуправляемом процессе.

Если вы создаете класс, расширяющийся `System.Exception`, необходимо задать поле HRESULT во время строительства. В противном случае значение HRESULT будет присваиваться базовым классом. Чтобы сопоставить новые классы исключений с существующим значением HRESULT, укажите значение в конструкторе исключения.

Обратите внимание, что среда выполнения в некоторых случаях будет игнорировать значение `HRESULT`, если в потоке присутствует `IErrorInfo`. Это возможно в тех случаях, когда `HRESULT` и `IErrorInfo` представляют разные ошибки.

## Создание нового класса исключения и его сопоставление со значением HRESULT

1. В следующем коде создается новый класс исключения `NoAccessException`, который сопоставляется со значением HRESULT `E_ACCESSDENIED`.

```
C++

Class NoAccessException : public ApplicationException
{
    NoAccessException () {
    HResult = E_ACCESSDENIED;
}
}
CMyClass::MethodThatThrows
```

```
{
throw new NoAccessException();
}
```

На любом языке программирования могут встречаться программы, в которых одновременно используется управляемый и неуправляемый код. Например, пользовательский маршаллировщик в следующем примере кода использует `Marshal.ThrowExceptionForHR(int HResult)` метод для создания исключения с определенным значением HRESULT. Этот метод выполняет поиск значения HRESULT и создает исключение соответствующего типа. Например, HRESULT в следующем фрагменте кода создает `ArgumentException`.

C++

```
CMyClass::MethodThatThrows
{
    Marshal.ThrowExceptionForHR(COR_E_ARGUMENT);
}
```

В следующей таблице приводится общий перечень сопоставлений значений HRESULT с соответствующими классами исключений в .NET. Значения HRESULT без явных сопоставлений сопоставляются с `COMException`. Актуальные сведения о сопоставлении представлены в репозитории среды выполнения/dotnet.

Развернуть таблицу

| HRESULT | Исключение .NET |
|---|---|
| `COR_E_APPLICATION` | `ApplicationException` |
| `COR_E_ARGUMENT` или `E_INVALIDARG` | `ArgumentException` |
| `COR_E_ARGUMENTOUTOFRANGE` | `ArgumentOutOfRangeException` |
| `COR_E_ARITHMETIC or ERROR_ARITHMETIC_OVERFLOW` | `ArithmeticException` |
| `COR_E_ARRAYTYPEMISMATCH` | `ArrayTypeMismatchException` |
| `COR_E_BADIMAGEFORMAT or ERROR_BAD_FORMAT` | `BadImageFormatException` |
| `COR_E_DIRECTORYNOTFOUND or ERROR_PATH_NOT_FOUND` | `DirectoryNotFoundException` |
| `COR_E_DIVIDEBYZERO` | `DivideByZeroException` |
| `COR_E_DUPLICATEWAITOBJECT` | `DuplicateWaitObjectException` |
| `COR_E_ENDOFSTREAM` | `EndOfStreamException` |

| HRESULT | Исключение .NET |
| --- | --- |
| `COR_E_ENTRYPOINTNOTFOUND` | `EntryPointNotFoundException` |
| `COR_E_EXCEPTION` | `Exception` |
| `COR_E_EXECUTIONENGINE` | `ExecutionEngineException` |
| `COR_E_FIELDACCESS` | `FieldAccessException` |
| `COR_E_FILENOTFOUND or ERROR_FILE_NOT_FOUND` | `FileNotFoundException` |
| `COR_E_FORMAT` | `FormatException` |
| `COR_E_INDEXOUTOFRANGE` | `IndexOutOfRangeException` |
| `COR_E_INVALIDCAST or E_NOINTERFACE` | `InvalidCastException` |
| `COR_E_INVALIDFILTERCRITERIA` | `InvalidFilterCriteriaException` |
| `COR_E_INVALIDOPERATION` | `InvalidOperationException` |
| `COR_E_IO` | `IOException` |
| `COR_E_MEMBERACCESS` | `AccessException` |
| `COR_E_METHODACCESS` | `MethodAccessException` |
| `COR_E_MISSINGFIELD` | `MissingFieldException` |
| `COR_E_MISSINGMANIFESTRESOURCE` | `MissingManifestResourceException` |
| `COR_E_MISSINGMEMBER` | `MissingMemberException` |
| `COR_E_MISSINGMETHOD` | `MissingMethodException` |
| `COR_E_NOTFINITENUMBER` | `NotFiniteNumberException` |
| `E_NOTIMPL` | `NotImplementedException` |
| `COR_E_NOTSUPPORTED` | `NotSupportedException` |
| `COR_E_NULLREFERENCE orE_POINTER` | `NullReferenceException` |
| `COR_E_OUTOFMEMORY or`<br><br>`E_OUTOFMEMORY` | `OutOfMemoryException` |
| `COR_E_OVERFLOW` | `OverflowException` |
| `COR_E_PATHTOOLONG or ERROR_FILENAME_EXCED_RANGE` | `PathTooLongException` |

| HRESULT | Исключение .NET |
| --- | --- |
| `COR_E_RANK` | `RankException` |
| `COR_E_REFLECTIONTYPELOAD` | `ReflectionTypeLoadException` |
| `COR_E_SECURITY` | `SecurityException` |
| `COR_E_SERIALIZATION` | `SerializationException` |
| `COR_E_STACKOVERFLOW orERROR_STACK_OVERFLOW` | `StackOverflowException` |
| `COR_E_SYNCHRONIZATIONLOCK` | `SynchronizationLockException` |
| `COR_E_SYSTEM` | `SystemException` |
| `COR_E_TARGET` | `TargetException` |
| `COR_E_TARGETINVOCATION` | `TargetInvocationException` |
| `COR_E_TARGETPARAMCOUNT` | `TargetParameterCountException` |
| `COR_E_THREADINTERRUPTED` | `ThreadInterruptedException` |
| `COR_E_THREADSTATE` | `ThreadStateException` |
| `COR_E_TYPELOAD` | `TypeLoadException` |
| `COR_E_TYPEINITIALIZATION` | `TypeInitializationException` |
| `COR_E_VERIFICATION` | `VerificationException` |

Чтобы получить дополнительные сведения об ошибке, управляемый клиент должен проверить поля созданного объекта исключения. Чтобы объект исключения предоставлял полезную информацию об ошибке, com-объект должен реализовать `IErrorInfo` интерфейс. Среда выполнения использует сведения, предоставленные `IErrorInfo` для инициализации объекта исключения.

Если COM-объект не поддерживается `IErrorInfo`, среда выполнения инициализирует объект исключения со значениями по умолчанию. В следующей таблице перечислены все поля, связанные с объектом исключения, идентифицируют источник сведений по умолчанию при поддержке `IErrorInfo`COM-объекта.

Обратите внимание, что среда выполнения в некоторых случаях будет игнорировать значение `HRESULT`, если в потоке присутствует `IErrorInfo`. Это возможно в тех случаях, когда `HRESULT` и `IErrorInfo` представляют разные ошибки.

Развернуть таблицу

| Поле Exception | Источник сведений из модели COM |
|---|---|
| `ErrorCode` | Значение HRESULT, возвращенное из вызова. |
| `HelpLink` | Если `IErrorInfo->HelpContext` значение ненулевое, строка формируется путем объединения `IErrorInfo->GetHelpFile` и "#" и `IErrorInfo->GetHelpContext`. В противном случае строка возвращается из `IErrorInfo->GetHelpFile`. |
| `InnerException` | Всегда является пустой ссылкой (`Nothing` в Visual Basic). |
| `Message` | Строка, возвращаемая `IErrorInfo->GetDescription`. |
| `Source` | Строка, возвращаемая `IErrorInfo->GetSource`. |
| `StackTrace` | Трассировка стека. |
| `TargetSite` | Имя метода, который вернул значение HRESULT со сбоем. |

Поля исключений, например `Message` `Source`, и `StackTrace` недоступны для `StackOverflowException` них.

# См. также

- Расширенное COM-взаимодействие
- Исключения

# Практическое руководство. Создание оболочек COM

Статья • 10.05.2023

Программы-оболочки модели COM можно создавать с использованием функций Visual Studio или средств платформы .NET Framework (Tlbimp.exe и Regasm.exe). Оба метода позволяют создать два типа программ-оболочек COM:

- Вызываемая оболочка времени выполнения из библиотеки типов для выполнения COM-объектов в управляемом коде.
- Вызываемая оболочка COM с соответствующими параметрами реестра для выполнения управляемого объекта в собственном приложении.

В Visual Studio оболочку COM можно добавить в проект в виде ссылки.

## Создание оболочек для COM-объектов в управляемом приложении

### Создание вызываемой оболочки времени выполнения с использованием Visual Studio

1. Откройте проект управляемого приложения.
2. В меню **Проект** выберите пункт **Показать все файлы**.
3. В меню **Проект** щелкните команду **Добавить ссылку**.
4. В диалоговом окне "Добавление ссылки" перейдите на вкладку **COM**, выберите нужный компонент и нажмите кнопку **OK**.

   Обратите внимание, что в **обозревателе решений** в папку ссылок проекта добавляется COM-компонент.

Теперь можно написать код для доступа к COM-объекту. Сначала можно объявить объект, например с помощью оператора `Imports` для Visual Basic или оператора `Using` для C#.

> ⓘ **Примечание**

При программировании компонентов Microsoft Office сначала необходимо установить **распространяемые основные сборки взаимодействия Microsoft Office**.

### Создание вызываемой оболочки времени выполнения с использованием средств платформы .NET Framework

- Запустите средство Tlbimp.exe (программа экспорта библиотек типов).

Это средство создает сборку, которая содержит метаданные времени выполнения для типов, определенных в исходной библиотеке типов.

## Создание оболочек для управляемых объектов в собственном приложении

### Создание вызываемой оболочки COM с использованием Visual Studio

1. Создайте проект библиотеки классов для управляемого класса, который требуется выполнять в машинном коде. Класс должен содержать конструктор без параметров.

   Убедитесь, что в файле AssemblyInfo присутствует полный номер версии сборки, состоящий из четырех частей. Этот номер необходим для управления версиями в реестре Windows. Дополнительные сведения о номерах версий см. в разделе Управление версиями сборки.

2. В меню **Проект** выберите пункт **Свойства**.

3. Откройте вкладку **Компиляция**.

4. Установите флажок **Регистрация для COM-взаимодействия**.

При построении проекта сборка автоматически регистрируется для COM-взаимодействия. При создании собственного приложения в Visual Studio можно использовать сборку, выбрав **Добавить ссылку** в меню **Проект**.

### Создание вызываемой оболочки COM с использованием средств платформы .NET Framework

Запустите программу Regasm.exe (средство регистрации сборок).

Это средство считывает метаданные сборки и добавляет в реестр необходимые записи. В результате этого клиенты COM получают возможность прозрачно создавать классы .NET Framework. Сборку можно использовать так, как если бы она была собственным COM-классом.

Программу Regasm.exe можно запускать для сборки, расположенной в любом каталоге. После этого необходимо запустить Gacutil.exe (программу глобального кэша сборок), чтобы перенести ее в глобальный кэш сборок. При переносе сборки записи расположения в реестре сохраняют силу, поскольку во всех случаях, когда сборка не найдена, проверяется глобальный кэш сборок.

## См. также

- Вызываемая оболочка времени выполнения
- Вызываемая оболочка COM

# Эквивалентность типов и встраиваемые типы интероперабельности

17.06.2025

Начиная с .NET Framework 4, общеязыковая среда исполнения поддерживает внедрение сведений о типах COM непосредственно в управляемые сборки, а не требует получения сведений о типах COM из сборок взаимодействия. Так как сведения о внедренном типе включают только типы и элементы, которые фактически используются управляемой сборкой, две управляемые сборки могут иметь очень разные представления одного и того же типа COM. Каждая управляемая сборка имеет свой собственный объект Type для представления ее представления типа COM. Общая среда выполнения поддерживает эквивалентность типов между этими различными представлениями интерфейсов, структур, перечислений и делегатов.

Эквивалентность типов означает, что com-объект, передаваемый из одной управляемой сборки в другую, можно привести к соответствующему управляемому типу в принимающей сборке.

> **Примечание**
>
> Эквивалентность типов и внедренные типы взаимодействия упрощают развертывание приложений и надстроек, использующих com-компоненты, так как не требуется развертывать сборки взаимодействия с приложениями. Разработчики общих COM-компонентов по-прежнему должны создавать первичные сборки взаимодействия (PIAs), если они хотят, чтобы их компоненты использовались более ранними версиями .NET Framework.

## Эквивалентность типов

Эквивалентность типов COM поддерживается для интерфейсов, структур, перечислений и делегатов. Типы COM считаются эквивалентными, если все следующие условия истинны:

- Эти типы представляют собой либо оба интерфейса, либо обе структуры, либо оба перечисления, либо оба делегата.
- Типы обладают тем же тождеством, как описано в следующем разделе.
- Оба типа могут быть эквивалентны, как описано в разделе Маркировка типов COM для эквивалентности типов.

## Идентификация типа

Два типа имеют одинаковую идентичность, если их области и идентичности совпадают, то есть если они имеют атрибут TypeIdentifierAttribute, а оба атрибута содержат совпадающие свойства Scope и Identifier. Сравнение для Scope выполняется без учета регистра.

Если у типа нет TypeIdentifierAttribute атрибута или имеется TypeIdentifierAttribute атрибут, который не указывает область и идентификатор, тип по-прежнему можно рассматривать для эквивалентности следующим образом:

- Для интерфейсов вместо свойства GuidAttribute используется значение TypeIdentifierAttribute.Scope, и вместо свойства Type.FullName используется свойство TypeIdentifierAttribute.Identifier (т. е. имя типа, включая пространство имен).
- Для структур, перечислений и делегатов используется сборка, содержащая GuidAttribute, вместо свойства Scope, а свойство Type.FullName используется вместо свойства Identifier.

## Маркировка типов COM для эквивалентности типов

Тип можно пометить как подходящий для эквивалентности типов двумя способами:

- Примените TypeIdentifierAttribute атрибут к типу.
- Введите тип импорта COM. Интерфейс является типом импорта COM, если имеет атрибут, обозначенный как ComImportAttribute. Интерфейс, структура, перечисление или делегат — это тип импорта COM, если сборка, в которой она определена, имеет ImportedFromTypeLibAttribute атрибут.

# См. также

- IsEquivalentTo
- Использование типов COM в управляемом коде
- Импорт библиотеки типов в качестве сборки

# Практическое руководство. Создание основной сборки взаимодействия с помощью программы Tlbimp.exe

Статья • 04.07.2024

Существует два способа создания основной сборки взаимодействия.

- С помощью программы импорта библиотек типов (Tlbimp.exe), предоставляемой Windows SDK.

  Использование программы Tlbimp.exe — это самый простой способ создания основных сборок взаимодействия. Эта программа предоставляет следующие меры безопасности:

  - перед созданием сборок взаимодействия для ссылок на любые вложенные библиотеки типов проверяет наличие других зарегистрированных основных сборок взаимодействия;
  - отказывается создавать основную сборку взаимодействия, если не указан контейнер или имя файла, позволяющие присвоить строгое имя основной сборке взаимодействия;
  - отказывается создавать основную сборку взаимодействия, если пропущены ссылки на зависимые сборки;
  - отказывается создавать основную сборку взаимодействия, если вы добавили ссылки на зависимые сборки, не являющиеся основными сборками взаимодействия.

- Создание основных сборок взаимодействия в исходном коде вручную с помощью языка, совместимого со спецификацией CLS, например C#. Этот подход полезен, когда библиотека типов недоступна.

Для подписи сборки строгим именем необходимо иметь пару криптографических ключей. Подробнее см. в разделе Создание пары ключей.

## Создание основной сборки взаимодействия с помощью программы Tlbimp.exe

1. В командной строке введите:

**tlbimp** *файл_tlb* **/primary /keyfile:** *имя_файла* **/out:** *имя_сборки*.

В этой команде *файл_tlb* — это файл, содержащий библиотеку типов COM, *имя_файла* — это имя контейнера или файла, содержащего пару ключей, а *имя_сборки* — это имя сборки, которую необходимо подписать строгим именем.

Основные сборки взаимодействия могут ссылаться только на другие основные сборки взаимодействия. Если сборка ссылается на типы из библиотеки типов COM стороннего разработчика, то перед созданием собственной основной сборки взаимодействия необходимо получить основную сборку взаимодействия от издателя. Если издателем являетесь вы, перед созданием ссылающейся основной сборки взаимодействия нужно создать основную сборку взаимодействия для зависимой библиотеки типов.

Зависимую основную сборку взаимодействия с номером версии, отличающимся от номера версии исходной библиотеки типов, невозможно обнаружить при установке в текущий каталог. Необходимо или зарегистрировать зависимую основную сборку взаимодействия в реестре Windows, или воспользоваться параметром **/reference**, чтобы программа Tlbimp.exe обнаружила зависимую библиотеку DLL.

Также можно включить несколько версий библиотеки типов. Инструкции см. в разделе Практическое руководство. Включение нескольких версий библиотек типов.

# Пример

В приведенном ниже примере выполняется импорт библиотеки типов COM `LibUtil.tlb` и подписание сборки `LibUtil.dll` строгим именем с помощью файла ключа `CompanyA.snk`. Так как имя пространства имен не указано, в этом примере создается пространство имен по умолчанию (`LibUtil`).

Консоль

```
tlbimp LibUtil.tlb /primary /keyfile:CompanyA.snk /out:LibUtil.dll
```

Чтобы имя было более понятным (соответствовало правилу именования *ИмяПоставщика.ИмяБиблиотеки*), в приведенном ниже примере переопределяются имена файла сборки и пространства имен, используемые по умолчанию.

Консоль

```
tlbimp LibUtil.tlb /primary /keyfile:CompanyA.snk
/namespace:CompanyA.LibUtil /out:CompanyA.LibUtil.dll
```

В приведенном ниже примере выполняется импорт библиотеки `MyLib.tlb`, которая ссылается на `CompanyA.LibUtil.dll`, и подписание сборки `CompanyB.MyLib.dll` строгим именем с помощью файла ключа `CompanyB.snk`. Пространство имен `CompanyB.MyLib` переопределяет пространство имен по умолчанию.

Консоль

```
tlbimp MyLib.tlb /primary /keyfile:CompanyB.snk /namespace:CompanyB.MyLib
/reference:CompanyA.LibUtil.dll /out:CompanyB.MyLib.dll
```

# См. также

- Практическое руководство. Регистрация основных сборок взаимодействия

# Практическое руководство. Регистрация основных сборок взаимодействия

Статья • 11.04.2023

Классы могут маршалироваться только с помощью COM-взаимодействия и всегда маршалируются как интерфейсы. Иногда интерфейс, используемый для маршалинга класса, называют интерфейсом класса. Сведения о переопределении интерфейса класса выбранным интерфейсом см. в разделе Вызываемая оболочка COM.

Разработчик, желающий использовать типы COM из приложения .NET Framework, может создать сборку взаимодействия, однако это связано с проблемой. Каждый раз, когда разработчик импортирует и подписывает библиотеку типов COM, он создает набор уникальных типов, которые несовместимы с типами, импортированными и подписанными другим разработчиком. Чтобы решить эту проблему несовместимости, каждый разработчик должен получить предоставляемую поставщиком и подписанную основную сборку взаимодействия.

Если вы планируете предоставлять доступ к сторонним типам COM другим приложениям, всегда используйте основную сборку взаимодействия, предоставленную тем же издателем, что и определяемая ею библиотека типов. Помимо обеспечения гарантированной совместимости типов, основные сборки взаимодействия часто настраиваются поставщиками так, чтобы оптимизировать взаимодействие.

Даже если вы не планируете предоставлять доступ к сторонним типам COM, использование основной сборки взаимодействия может упростить задачу взаимодействия с COM-компонентами. Однако такая стратегия не обеспечивает изоляции от изменений, которые поставщик может вносить в типы, определенные в основной сборке взаимодействия. Если приложению требуется такая изоляция, создайте собственную сборку взаимодействия вместо использования основной сборки взаимодействия.

Вам необходимо зарегистрировать все полученные основные сборки взаимодействия на своем компьютере, прежде чем вы сможете ссылаться на них в Visual Studio. Visual Studio ищет основную сборку взаимодействия и использует ее при первой ссылке на тип из библиотеки типов COM. Если программа Visual Studio не может найти основную сборку взаимодействия, связанную с библиотекой типов, она предлагает приобрести ее или создать сборку взаимодействия. Средство

импорта библиотек типов (Tlbimp.exe) также использует реестр для обнаружения основных сборок взаимодействия.

Если вы не планируете использовать Visual Studio, регистрировать основные сборки взаимодействия необязательно, однако регистрация предоставляет два преимущества.

- Зарегистрированная основная сборка взаимодействия явным образом помечается в разделе реестра исходной библиотеки типов. Регистрация — наилучший способ обнаружения основной сборки взаимодействия на компьютере.
- Использование Visual Studio для ссылки на тип, для которого существует незарегистрированная основная сборка взаимодействия, позволяет избежать случайного создания и использования новой сборки взаимодействия.

Для регистрации основной сборки взаимодействия используется средство регистрации сборок (Regasm.exe).

# Регистрация основной сборки взаимодействия

1. В командной строке введите следующее:

   **regasm***assemblyname*

   В этой команде *имя_сборки* — это имя файла регистрируемой сборки. Программа Regasm.exe добавляет запись об основной сборке взаимодействия в тот же раздел реестра, что и для исходной библиотеки типов.

# Пример

В примере ниже регистрируется основная сборка взаимодействия `CompanyA.UtilLib.dll`.

Консоль

```
regasm CompanyA.UtilLib.dll
```

# См. также

- Программирование с использованием основных сборок взаимодействия
- Locating Primary Interop Assemblies (Обнаружение основных сборок взаимодействия)
- Распространение основных сборок взаимодействия

# Registration-Free COM-взаимодействие

17.06.2025

Взаимодействие COM без регистрации активирует компонент без использования реестра Windows для хранения сведений о сборке. Вместо регистрации компонента на компьютере во время развертывания вы создаете файлы манифестов в стиле Win32 во время разработки, содержащие сведения о привязке и активации. Эти файлы манифестов, а не разделы реестра, управляют активацией объекта.

Использование активации без регистрации для сборок вместо регистрации во время развертывания предлагает два преимущества:

- Вы можете контролировать, какая версия DLL активируется при установке нескольких версий на компьютере.
- Конечные пользователи могут использовать XCOPY или FTP для копирования приложения в соответствующий каталог на своем компьютере. Затем приложение можно запустить из этого каталога.

В этом разделе описаны два типа манифестов, необходимых для взаимодействия COM без регистрации: манифесты приложений и компонентов. Эти манифесты являются XML-файлами. Манифест приложения, созданный разработчиком приложения, содержит метаданные, описывающие сборки и зависимости сборок. Манифест компонента, созданный разработчиком компонента, содержит сведения, которые обычно находятся в реестре Windows.

## Требования к безрегистрационному COM-взаимодействию

1. Поддержка взаимодействия COM без регистрации слегка меняется в зависимости от типа сборки библиотеки; в частности, является ли сборка неуправляемой (COM side-by-side) или управляемой (.NET-based). В следующей таблице показаны требования к версии операционной системы и .NET Framework для каждого типа сборки.

Развернуть таблицу

| Тип сборки | Операционная система | Версия платформы .NET Framework |
|---|---|---|
| COM бок о бок | Microsoft Windows XP | Необязательно. |
| На основе | Windows XP с пакетом обновления 2 | NET Framework версии 1.1 или более |

| Тип сборки | Операционная система | Версия платформы .NET Framework |
| --- | --- | --- |
| .NET | (SP2) | поздней. |

Семейство Windows Server 2003 также поддерживает взаимодействие COM без регистрации для сборок на основе .NET.

Для обеспечения совместимости класса на базе .NET с активацией без реестра из COM, класс должен иметь конструктор без параметров и быть публичным.

## Настройка COM-компонентов для активации без регистрации

1. Чтобы компонент COM участвовал в активации без регистрации, его необходимо развернуть в виде параллельной сборки. Параллельные сборки — это неуправляемые сборки. Дополнительные сведения см. в разделе "Использование параллельных сборок".

   Для использования сборок COM side-by-side разработчик приложений на основе .NET должен предоставить манифест приложения, содержащий сведения о привязке и активации. Поддержка неуправляемых параллельных сборок встроена в операционную систему Windows XP. Среда выполнения COM, поддерживаемая операционной системой, проверяет манифест приложения для получения сведений о активации, когда компонент, активируемый, не находится в реестре.

   Активация без регистрации необязательна для компонентов COM, установленных в Windows XP. Подробные инструкции по добавлению параллельной сборки в приложение см. в разделе "Использование параллельных сборок".

   > **Примечание**
   >
   > Параллельное выполнение — это функция .NET Framework, которая позволяет одновременно запускать на одном компьютере несколько версий среды выполнения, а также несколько версий приложений и компонентов, использующих версию среды выполнения. Параллельное выполнение и параллельные сборки являются различными механизмами обеспечения параллельной функциональности.

# См. также

- Практическое руководство. Настройка com-компонентов .NET Framework-Based для активации Registration-Free

# Как настроить COM-компоненты на базе .NET Framework для активации без регистрации

Активация без регистрации для компонентов на основе .NET Framework немного сложнее, чем для com-компонентов. Для установки требуется два манифеста:

- COM-приложения должны иметь манифест приложения в стиле Win32 для идентификации управляемого компонента.
- Компоненты на основе .NET Framework должны иметь манифест компонента для сведений о активации, необходимых во время выполнения.

В этом разделе описывается связывание манифеста приложения с приложением; связывание манифеста компонента с компонентом; и внедрение манифеста компонента в сборку.

## Создание манифеста приложения

1. Используя редактор XML, создайте (или измените) манифест приложения, принадлежащий com-приложению, которое взаимодействует с одним или несколькими управляемыми компонентами.
2. Вставьте следующий стандартный заголовок в начале файла:

   XML

   ```xml
   <?xml version="1.0" encoding="UTF-8" standalone="yes"?>
   <assembly xmlns="urn:schemas-microsoft-com:asm.v1" manifestVersion="1.0">
   </assembly>
   ```

   Сведения о элементах манифеста и их атрибутах см. в разделе "Манифесты приложения".
3. Определите владельца манифеста. В следующем примере `myComApp` версия 1 владеет файлом манифеста.

   XML

   ```xml
   <?xml version="1.0" encoding="UTF-8" standalone="yes"?>
   <assembly xmlns="urn:schemas-microsoft-com:asm.v1" manifestVersion="1.0">
     <assemblyIdentity type="win32"
                       name="myOrganization.myDivision.myComApp"
   ```

```
                    version="1.0.0.0"
                    processorArchitecture="msil"
  />
</assembly>
```

4. Определите зависимые сборки. В следующем примере `myComApp` зависит от `myManagedComp`.

```
XML
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<assembly xmlns="urn:schemas-microsoft-com:asm.v1" manifestVersion="1.0">
  <assemblyIdentity type="win32"
                    name="myOrganization.myDivision.myComApp"
                    version="1.0.0.0"
                    processorArchitecture="x86"
                    publicKeyToken="8275b28176rcbbef"
  />
  <dependency>
    <dependentAssembly>
      <assemblyIdentity type="win32"
                    name="myOrganization.myDivision.myManagedComp"
                    version="6.0.0.0"
                    processorArchitecture="X86"
                    publicKeyToken="8275b28176rcbbef"
      />
    </dependentAssembly>
  </dependency>
</assembly>
```

5. Сохраните и присвойте файлу манифеста имя. Имя манифеста приложения — это имя исполняемого файла сборки, за которым следует расширение .manifest. Например, имя файла манифеста приложения для myComApp.exe myComApp.exe.manifest.

Манифест приложения можно установить в том же каталоге, что и приложение COM. Кроме того, его можно добавить как ресурс в файл .exe приложения. Дополнительные сведения см. в разделе "О параллельных сборках".

# Создание манифеста компонента

1. С помощью редактора XML создайте манифест компонента для описания управляемой сборки.

2. Вставьте следующий стандартный заголовок в начале файла:

```
XML
```

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<assembly xmlns="urn:schemas-microsoft-com:asm.v1" manifestVersion="1.0">
</assembly>
```

3. Определите владельца файла. Элемент `<assemblyIdentity>` элемента в файле манифеста `<dependentAssembly>` приложения должен соответствовать элементу в манифесте компонента. В следующем примере `myManagedComp` версия 1.2.3.4 владеет файлом манифеста.

XML

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<assembly xmlns="urn:schemas-microsoft-com:asm.v1" manifestVersion="1.0">
       <assemblyIdentity
                     name="myOrganization.myDivision.myManagedComp"
                     version="1.2.3.4"
                     publicKeyToken="8275b28176rcbbef"
                     processorArchitecture="msil"
       />
</assembly>
```

4. Определите каждый класс в сборке. `<clrClass>` Используйте элемент для уникальной идентификации каждого класса в управляемой сборке. Элемент, являющийся подэлементом `<assembly>` элемента, имеет атрибуты, описанные в следующей таблице.

Развернуть таблицу

| Свойство | Description | Обязательно |
|---|---|---|
| `clsid` | Идентификатор, указывающий активируемый класс. | Да |
| `description` | Строка, которая сообщает пользователю о компоненте. Пустая строка по умолчанию. | нет |
| `name` | Строка, представляющая управляемый класс. | Да |
| `progid` | Идентификатор, используемый для активации с поздней привязкой. | нет |
| `threadingModel` | Модель потоков COM. "Оба" — это значение по умолчанию. | нет |
| `runtimeVersion` | Указывает версию общеязыковой среды выполнения (CLR). Если этот атрибут не указан, а среда CLR еще не загружена, компонент загружается с последней установленной средой CLR до среды CLR версии 4. Если указать версию 1.0.3705, 1.1.4322 или 2.0.50727, она автоматически обновляется до последней установленной версии CLR перед версией 4 CLR | нет |

| Свойство | Description | Обязательно |
| --- | --- | --- |
| | (обычно 2.0.50727). Если другая версия среды CLR уже загружена, и указанная версия может быть загружена параллельно в процессе, указанная версия загружается; в противном случае используется загруженная среда CLR. Это может привести к сбою загрузки. | |
| `tlbid` | Идентификатор библиотеки типов, содержащей сведения о типе класса. | нет |

Все теги атрибутов чувствительны к регистру. Вы можете получить CLSID, ProgID, модели потоков и версию среды выполнения, просмотрев экспортированную библиотеку типов для сборки с помощью OLE/COM ObjectViewer (Oleview.exe).

Следующий манифест компонента определяет два класса `testClass1` и `testClass2`.

XML

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<assembly xmlns="urn:schemas-microsoft-com:asm.v1" manifestVersion="1.0">
       <assemblyIdentity
                    name="myOrganization.myDivision.myManagedComp"
                    version="1.2.3.4"
                    publicKeyToken="8275b28176rcbbef"
       />
       <clrClass
                    clsid="{65722BE6-3449-4628-ABD3-74B6864F9739}"
                    progid="myManagedComp.testClass1"
                    threadingModel="Both"
                    name="myManagedComp.testClass1"
                    runtimeVersion="v1.0.3705">
       </clrClass>
       <clrClass
                    clsid="{367221D6-3559-3328-ABD3-45B6825F9732}"
                    progid="myManagedComp.testClass2"
                    threadingModel="Both"
                    name="myManagedComp.testClass2"
                    runtimeVersion="v1.0.3705">
       </clrClass>
       <file name="MyManagedComp.dll">
       </file>
</assembly>
```

5. Сохраните и присвойте файлу манифеста имя. Имя манифеста компонента — это имя библиотеки сборок, за которой следует расширение .manifest. Например, myManagedComp.dll соответствует myManagedComp.manifest.

Необходимо внедрить манифест компонента в качестве ресурса в сборку.

### Внедрение манифеста компонента в управляемую сборку

1. Создайте скрипт ресурса, содержащий следующую инструкцию:

   ```
   1 RT_MANIFEST myManagedComp.manifest
   ```

   В этом заявлении `myManagedComp.manifest` является именем внедряемого манифеста компонента. В этом примере имя файла скрипта `myresource.rc`.

2. Скомпилируйте скрипт с помощью компилятора ресурсов Microsoft Windows (Rc.exe). В командной строке введите следующую команду:

   ```
   rc myresource.rc
   ```

   Rc.exe создает `myresource.res` файл ресурса.

3. Скомпилируйте исходный файл сборки еще раз и укажите файл ресурсов с помощью параметра **/win32res** :

   ```
   /win32res:myresource.res
   ```

   Опять же, `myresource.res` это имя файла ресурса, содержащего внедренные ресурсы.

## См. также

- Взаимодействие COM без регистрации
- Требования к COM-взаимодействию без регистрации
- Настройка COM-компонентов для активации без регистрации
- Активация компонентов на платформе .NET без регистрации: пошаговое руководство

---

Last updated on 03.12.2025