Руководство программиста FastScript .NET

Версия 2025.1

Общие сведения

FastScript .NET - это библиотека для выполнения скриптов на языке C#. FastScript .NET построен по классической схеме "лексер-парсер-интерпретатор", не использует компиляцию в машинный код и поэтому совместим с ограниченными средами, в которых запрещена кодогенерация (Native AOT, iOS, WASM).

Поддерживаемые особенности

FastScript .NET поддерживает следующие особенности языка C# (наиболее полно поддерживается спецификация C# 1.0, с добавлением многих особенностей более поздних версий языка):

C# 1.0:

- Классы (Classes)
- Структуры (Structs)
- Перечисления (Enums)
- Интерфейсы (Interfaces)
- События (Events)
- Перегрузка операторов (Operator overloading)
- Пользовательские операторы конверсии типов (User-defined conversion operators)
- Свойства (Properties)
- Индексные свойства (Indexers)
- Выходные параметры методов (Output parameters - out, ref)
- Массив параметров (params arrays)
- Делегаты (Delegates)
- Операторы и выражения (Operators and expressions)
- Буквальный идентификатор "@" (Verbatim identifier)

C# 2.0:

- Обобщенные типы (Generics)
- Частичные типы (Partial types)
- Nullable типы (Nullable value types)
- Разные модификаторы доступа у getter/setter (Getter/setter separate accessibility)
- Статические классы (Static classes)

C# 3.0:

- Авто-свойства (Auto-implemented properties)
- Методы расширения (Extension methods)
- Объявление переменных без указания типа (Implicitly typed local variables)

C# 4.0:

- Значения параметров по умолчанию (Optional arguments)

C# 6.0:

- Инициализаторы для авто-свойств (Auto-property initializers)
- Свойства и методы, возвращающие выражение "=>" (Expression bodied members)
- Оператор проверки на null "?." (Null propagator)

C# 7.0:

- Выходные переменные (Out variables)
- Локальные функции (Local functions)

C# 8.0:

- Модификатор "readonly" для полей (Readonly members)

- Статические локальные функции (Static local functions)
- Операторы объединения null "??" и "??=" (Null-coalescing operators)

C# 9.0:

- Инструкции верхнего уровня (Top-level statements)

Неподдерживаемые особенности

Следующие особенности C# 1.0 не поддерживаются:

- Директивы препроцессора (Preprocessor directives - #if, #region, etc)
- Атрибуты (Attributes)
- Неуправляемый код (Unmanaged code: pointers, unsafe keyword, P/Invoke)
- Инструкции checked, unchecked (checked, unchecked statements)
- Инструкция goto (goto statement)

Ограниченная поддержка

Термины, используемые ниже: "хост" означает ваше .net приложение; "скрипт" означает нечто, определенное в коде скрипта.

Наследование классов

Классы, определенные в скрипте, можно наследовать от других скриптовых классов либо от System.Object:

```
class MyScriptClass: OtherScriptClass // ok
class MyScriptClass: Object // ok
class MyScriptClass // ok, same as Object
class MyScriptClass: ArrayList // error
```

Структуры

Структура в FastScript представляет собой обычный класс. FastScript добавляет специальные методы для копирования структуры, когда ее значение передается в метод или в другую переменную. Объявление переменной, имеющей тип структуры, не создает экземпляр структуры автоматически:

```
MyStruct s; // s is null
s = new MyStruct(); // and must be initialized before use
```

Взаимодействие с хостом

Класс, определенный в скрипте, виден хосту как экземпляр `FastScript.Runtime.DataContext`.

Вы можете переопределять следующие методы в скриптовом классе:

- `ToString`
- `Equals`
- `GetHashCode`

Переопределенные методы будут также использоваться хостом.

Скриптовый класс может реализовать интерфейсы, определенные в хосте, но это будет иметь значение только в скрипте. Хост не сможет работать с экземпляром такого класса, как с реализующим интерфейс.

Nullable типы

Nullable типы могут использовать только типы хоста.

Обобщенные типы и методы

В скрипте можно использовать только обобщенные типы и методы хоста. Вы не можете определять обобщенный тип или метод в скрипте.

Type forwarding

Если тип хоста обозначен как "forwarded", его нужно явно использовать в хосте, чтобы была возможность его использования в скрипте. Например:

```
var list = new System.ComponentModel.BindingList<int>(); // error, BindingList does not exist
```

Если добавить следующую строку кода в ваше приложение, скрипт будет скомпилирован без ошибок:

```
new System.ComponentModel.BindingList<int>();
```

Делегаты

Вы можете создавать делегаты на основе любых методов (определенных в скрипте или хосте). Передача делегатов хост не поддерживается. Также не поддерживается создание делегатов `Action<>` и `Func<>`.

Конверсии типов (implicit, explicit)

Конверсия типов, определенная пользователем (в коде скрипта или в хосте) ограничена теми типами, которые определены явно. Пример: если определена конверсия `T->int`, вы можете ее использовать. Но вы не сможете использовать конверсию `T->float`, если она явно не определена.

```
var m = new My();

int intValue = m; // ok
float floatValue = m; // error

int explicitIntValue = (int)m; // ok: explicit is not defined, but we have implicit one
float explicitFloatValue = (float)m; // error

floatValue = (int)m; // use this way

public class My
{
    private object _value;
    public static implicit operator int(My m) => (int)m._value;
}
```

Грамматика (C#)

Основные

```
program
    : using_directive* top_level_statements? namespace_member_declaration* EOF

using_directive
    : USING namespace_name ';'

top_level_statements
    : statement_list

namespace_member_declaration
    : namespace_declaration
    | type_declaration

namespace_declaration
    : NAMESPACE namespace_name namespace_body ';'?

namespace_name
    : identifier ('.' identifier)*

namespace_body
    : '{' using_directive* namespace_member_declaration* '}'

identifier
    : IDENTIFIER
```

Простые типы

```
namespace_or_type_name
    : name_part ('.' name_part)*

name_part
    : identifier type_argument_list?

type_
    : base_type '?'? rank_specifier*

type_no_rank
    : base_type '?'?

base_type
    : predefined_type
    | namespace_or_type_name

predefined_type
    : BOOL
    | BYTE
    | CHAR
    | DECIMAL
    | DOUBLE
    | FLOAT
    | INT
    | LONG
    | OBJECT
    | SBYTE
    | SHORT
    | STRING
    | UINT
    | ULONG
    | USHORT

type_argument_list
    : '<' type_ (',' type_)* '>'
```

Выражения

```
expression_list
    : expression (',' expression)*

expression
    : binary_expression (assignment | null_coalescing_expression | conditional_expression)?

assignment
    : assignment_operator expression

assignment_operator
    : '='
    | '+='
    | '-='
    | '*='
    | '/='
    | '%='
    | '&='
    | '|='
    | '^='
    | '??='

null_coalescing_expression
    : '??' expression

conditional_expression
    : '?' expression ':' expression
```

```
binary_expression
    : unary_expression (binary_operator unary_expression | is_operation | as_operation)*

binary_operator
    : '+'
    | '-'
    | '*'
    | '/'
    | '%'
    | '||'
    | '&&'
    | '=='
    | '!='
    | '<='
    | '>='
    | '|'
    | '&'
    | '^'
    | '<<'
    | '>>'
    | '<'
    | '>'

unary_expression
    : cast_expression
    | primary_expression
    | unary_operator unary_expression

unary_operator
    : '++'
    | '--'
    | '+'
    | '-'
    | '~'
    | '!'

cast_expression
    : '(' type_ ')' unary_expression

primary_expression
    : primary_expression_start bracket_expression* (
        (member_access | method_invocation | '++' | '- -') bracket_expression*
    )*

primary_expression_start
    : NUMBER | STRING
    | TRUE | FALSE | NULL
    | THIS | BASE
    | typeof_expression
    | name_part
    | '(' expression ')'
    | predefined_type
    | object_creation

typeof_expression
    : TYPEOF '(' type_ ')'

member_access
    : '?'? '.' name_part

bracket_expression
    : '?'? '[' expression_list ']'

is_operation
    : IS type_ identifier?

as_operation
    : AS type_
```

```
object_creation
    : NEW type_no_rank (method_invocation | array_creation)

method_invocation
    : '(' argument_list? ')'

argument_list
    : argument (',' argument)*

argument
    : out_argument
    | in_argument
    | ref_argument
    | expression

out_argument
    : OUT (embedded_variable_declaration | expression)

in_argument
    : IN expression

ref_argument
    : REF expression

array_creation
    : '[' expression_list? ']' rank_specifier* array_initializer?

array_initializer
    : '{' expression_list? ','? '}'
```

Инструкции

```
block
    : '{' statement_list? '}'

statement_list
    : statement+

statement
    : embedded_statement
    | declaration_statement

declaration_statement
    : local_variable_declaration ';'
    | local_constant_declaration ';'
    | local_function_declaration

embedded_statement
    : block
    | simple_embedded_statement

simple_embedded_statement
    : ';'
    | toplevel_statement
    | expression ';'

toplevel_statement
    : return_statement
    | if_statement
    | switch_statement
    | while_statement
    | do_statement
    | for_statement
    | foreach_statement
    | break_statement
```

```
    | break_statement
    | continue_statement
    | throw_statement
    | try_statement
    | using_statement

if_statement
    : IF '(' expression ')' embedded_statement (ELSE embedded_statement)?

switch_statement
    : SWITCH '(' expression ')' '{' switch_section* '}'

while_statement
    : WHILE '(' expression ')' embedded_statement

do_statement
    : DO embedded_statement WHILE '(' expression ')' ';'

for_statement
    : FOR '(' for_initializer? ';' expression? ';' for_iterator? ')' embedded_statement

foreach_statement
    : FOREACH '(' embedded_variable_declaration IN expression ')' embedded_statement

break_statement
    : BREAK ';'

continue_statement
    : CONTINUE ';'

return_statement
    : RETURN expression? ';'

throw_statement
    : THROW expression? ';'

try_statement
    : TRY block (catch_clauses finally_clause? | finally_clause)

using_statement
    : USING '(' resource_acquisition ')' embedded_statement

switch_section
    : switch_label+ statement_list

switch_label
    : CASE expression ':'
    | DEFAULT ':'

for_initializer
    : local_variable_declaration
    | expression (',' expression)*

for_iterator
    : expression (',' expression)*

catch_clauses
    : specific_catch_clause+ general_catch_clause?
    | general_catch_clause

specific_catch_clause
    : CATCH '(' base_type identifier? ')' block

general_catch_clause
    : CATCH block

finally_clause
    : FINALLY block

resource_acquisition
```

```
    resource_acquisition
        : local_variable_declaration
        | expression

    local_variable_declaration
        : local_variable_type variable_declarators

    local_variable_type
        : VAR
        | type_

    embedded_variable_declaration
        : local_variable_type identifier

    local_constant_declaration
        : CONST type_ constant_declarators

    local_function_declaration
        : STATIC? type_ method_declaration
```

Типы

```
    type_declaration
        : all_member_modifier* (class_definition | struct_definition | interface_definition |
enum_definition | delegate_definition)

    class_definition
        : CLASS identifier base_classes? class_body ';'?

    base_classes
        : ':' base_type (',' base_type)*

    class_body
        : '{' class_member_declaration* '}'

    struct_definition
        : STRUCT identifier base_classes? class_body ';'?

    interface_definition
        : INTERFACE identifier base_classes? class_body ';'?

    enum_definition
        : ENUM identifier (':' type_)? enum_body

    enum_body
        : '{' enum_member_declarators '}'

    delegate_definition
        : DELEGATE type_ identifier '(' formal_parameter_list? ')' ';'
```

Члены

```
    all_member_modifier
        : PUBLIC
        | PROTECTED
        | INTERNAL
        | PRIVATE
        | READONLY
        | STATIC
        | PARTIAL
        | ABSTRACT
        | VIRTUAL
        | OVERRIDE
```

```
      | NEW

  class_member_declaration
      : all_member_modifier* (constant_declaration | typed_member_declaration | constructor_declaration |
operator_declaration | event_declaration)
      | type_declaration

  enum_member_declarators
      : enum_member_declarator (',' enum_member_declarator)* ','?

  enum_member_declarator
      : identifier ('=' expression)?

  constant_declaration
      : CONST type_ constant_declarators ';'

  constant_declarators
      : constant_declarator (',' constant_declarator)*

  constant_declarator
      : identifier '=' expression

  constructor_declaration
      : identifier '(' formal_parameter_list? ')' constructor_initializer? method_body

  constructor_initializer
      : ':' (THIS | BASE) method_invocation

  operator_declaration
      : implicit_operator
      | explicit_operator
      | overload_operator

  implicit_operator
      : IMPLICIT OPERATOR type_ op_method_declaration

  explicit_operator
      : EXPLICIT OPERATOR type_ op_method_declaration

  overload_operator
      : type_ OPERATOR (binary_operator | unary_operator) op_method_declaration

  op_method_declaration
      : '(' formal_parameter_list? ')' method_body

  event_declaration
      : EVENT base_type explicit_interface? identifier (';' | event_accessor)

  event_accessor
      : '{' (event_add | event_remove)+ '}'

  event_add
      : ADD method_body

  event_remove
      : REMOVE method_body

  typed_member_declaration
      : type_ (method_declaration | property_declaration | field_declaration |
extension_method_declaration)

  method_declaration
      : explicit_interface? identifier '(' formal_parameter_list? ')' method_body

  explicit_interface
      : identifier '.'

  formal_parameter_list
      : parameter_array
```

```
    | fixed_parameters (',' parameter_array)?

fixed_parameters
    : fixed_parameter (',' fixed_parameter)*

fixed_parameter
    : parameter_modifier? type_ identifier ('=' expression)?

parameter_modifier
    : REF
    | OUT
    | IN

parameter_array
    : PARAMS array_type identifier

array_type
    : base_type rank_specifier+

rank_specifier
    : '[' ','* ']'

method_body
    : block
    | ';'
    | lambda_expression

lambda_expression
    : '=>' expression ';'

property_declaration
    : index_property_declaration
    | regular_property_declaration

index_property_declaration
    : explicit_interface? THIS '[' formal_parameter_list ']' property_accessor

regular_property_declaration
    : explicit_interface? identifier property_accessor

property_accessor
    : property_get_set ('=' expression ';')?
    | lambda_expression

property_get_set
    : '{' (property_getter | property_setter)* '}'

property_getter
    : all_member_modifier* GET (';' | method_body)

property_setter
    : all_member_modifier* SET (';' | method_body)

field_declaration
    : variable_declarators ';'

variable_declarators
    : variable_declarator (',' variable_declarator)*

variable_declarator
    : identifier ('=' expression)?

extension_method_declaration
    : identifier '(' THIS formal_parameter_list ')' method_body
```

Использование FastScript .NET

Для использования FastScript .NET в своем проекте добавьте Nuget пакет `FastScript` (или `FastScript.Demo` в случае использования демо-версии).

Ограничения демо-версии: при компиляции скрипта в код некоторых методов добавляется генерация исключения с текстом "FastScript: Demo version". Это происходит в случайные (довольно редкие) моменты времени.

Быстрый старт

Для запуска простых скриптов используйте код:

```
using FastScript.CSharp;

var text =
@"
using System;

Console.WriteLine(""Hello!"");
";

var script = new CSharpScript();

if (script.Compile(text))
{
    script.RunMain();
}
```

В приведенном примере используется скрипт с инструкциями верхнего уровня ("top level statements"). В случае успешной компиляции происходит запуск первой инструкции (Console.WriteLine) и в консоль выводится слово `Hello!`.

Компиляция скрипта

Метод `Script.Compile` выполняет полный разбор исходного текста (парсинг), а затем его компиляцию. В процессе парсинга создается абстрактное синтаксическое дерево (abstract syntax tree, AST), которое представляет исходный текст скрипта в виде древовидной структуры данных.

Если на этапе парсинга не возникло ошибок, происходит компиляция. На этапе компиляции выполняется парсинг AST и создаются структуры памяти, представляющие объявленные в скрипте типы. Эти типы доступны в свойстве `Script.Types`.

В процессе компиляции не используется генерация машинного кода. Это позволяет использовать FastScript .NET в средах с ограничениями на кодогенерацию, таких как Native AOT, iOS, WASM. В процессе работы FastScript .NET не создает сборок, которые остаются в памяти до конца работы приложения. Для работы скрипта выделяются структуры в памяти, которые удаляются сборщиком мусора (GC) после окончания работы с экземпляром скрипта.

Обработка ошибок

В процессе компиляции исходного кода скрипта могут возникать ошибки: ошибки парсинга (синтаксические ошибки) и ошибки компиляции (семантические ошибки).

Синтаксические ошибки возникают в случае, если скрипт не является корректным с точки зрения грамматики языка C# (например, в вызове метода отсутствует закрывающая скобка). В случае присутствия синтаксических ошибок дальнейшая обработка скрипта (компиляция) не производится. При нахождении ошибки происходит попытка продолжить разбор скрипта дальше; в результате может быть выдано несколько ошибок. В этом плане поведение FastScript похоже на то, что используется в CodeDOM/Roslyn.

Семантические ошибки могут возникнуть на этапе компиляции, например, при обращении к несуществующему методу.

Информация об ошибках содержится в свойстве `Script.Diagnostic.Errors`. Используйте следующий код для проверки наличия ошибок и вывода информации в консоль:

```
var script = new CSharpScript();

if (!script.Compile(script_text))
{
    foreach (var err in script.Diagnostic.Errors)
    {
        // basic error info
        Console.WriteLine($"{err.CompilationUnit.Name} ({err.Line},{err.Column}): error {err.Code}:
{err.Message}");
        // extended info
        Console.WriteLine(err.SourceCode());
    }

    return;
}
else
{
    script.RunMain();
}
```

В этом случае выдается расширенная информация об ошибке следующего вида:

```
Sample.cs (4,21): error cs69: The name 'My' does not exist in the current context
var list = new List<My>();
                    ^^
```

Запуск скрипта

Запуск скрипта на выполнение осуществляется методом `Script.RunMain`. Этот метод сканирует типы, объявленные в скрипте (список типов доступен в свойстве `Script.Types`): ищется тип, имеющий статический метод `Main`, которому и передается управление.

В примере ниже запуск скрипта методом `RunMain` найдет тип `TestClass`, имеющий статический метод `Main`, и выполнит его:

```
using FastScript.CSharp;

var text =
@"
namespace Test
{
    public class TestClass
    {
        public static void Main()
        {
            System.Console.WriteLine(""Hello!"");
        }
    }
}
";

var script = new CSharpScript();

if (script.Compile(text))
{
    script.RunMain();
}
```

Если запускаемый метод `Main` имеет параметры, их значения необходимо указать в методе `RunMain`, например:

```
    // script method:
    // public static void Main(int id, string text)

    script.RunMain(1, "abc");
```

Метод `RunMain` возвращает значение, которое возвращает метод `Main`, объявленный в скрипте. Если метод `Main` объявлен как `void`, возвращаемое значение неопределено.

Если скрипт имеет инструкции верхнего уровня (top level statements), для них создается специальный класс, имеющий статический метод `Main`. При запуске скрипта методом `RunMain`, этот метод получает управление.

Ниже представлен пример скрипта, аналогичный вышеописанному, который использует инструкции верхнего уровня:

```
using FastScript.CSharp;

var text =
@"
System.Console.WriteLine(""Hello!"");
";

var script = new CSharpScript();

if (script.Compile(text))
{
    script.RunMain();
}
```

Создание экземпляров классов

Метод `Script.RunMain` удобно использовать, если в скрипте есть явно или неявно объявленный статический метод `Main`. Другой сценарий использования скрипта заключается в следующем:

- создается экземпляр типа, объявленного в скрипте;
- производятся манипуляции с его свойствами и методами.

Рассмотрим следующий пример, в котором показано, как создать экземпляр типа `MyClass` и вызвать его метод `Print`:

```
using FastScript.CSharp;
using FastScript.Runtime.Types;

var text =
@"
using System;

namespace MyNamespace
{
    public class MyClass
    {
        public void Print(string message) => Console.WriteLine(message);
    }
}
";

var script = new CSharpScript();

if (script.Compile(text))
{
    // get MyClass type
    var myClass = script.Types["MyClass"] as ScriptTypeInfo;

    // make an instance of it (using default constructor)
    var myInstance = myClass.CreateInstance();

    // get Print method
    var printMethod = myClass.GetMethod("Print");

    // invoke it
    printMethod.Invoke(myInstance, new object[] { "Hello FastScript!" });
}
```

Пояснения к примеру:

- Типы, объявленные в скрипте, доступны в свойстве `Script.Types`.
- Скриптовый класс представлен классом `FastScript.Runtime.Types.ScriptTypeInfo`.
- Для создания экземпляра класса используется метод `ScriptTypeInfo.CreateInstance`. В параметрах метода можно указать агрументы для конструктора класса (или не указывать их, если вызывается конструктор без параметров).
- Для поиска метода `Print` используется метод `GetMethod`. Это API, аналогичное `System.Reflection`, включает в себя методы: `GetMember`, `GetMembers`, `GetMethod`, `GetMethods`, `GetConstructor`, `GetConstructors`, `GetField`, `GetFields`, `GetProperty`, `GetProperties`, `GetEvent`, `GetEvents`, `GetNestedType`, `GetNestedTypes`.
- Для запуска метода `Print` используется метод `Invoke`, аналогичный API из `System.Reflection`.

Использование модулей

Текст скрипта может быть разбит на несколько файлов ("модулей"). Для компиляции в таком случае надо использовать класс `CompilationUnit` и перегруженный вариант метода Script.Compile:

```
public bool Compile(params CompilationUnit[] units)
```

Вот пример использования двух модулей:

```
using FastScript;
using FastScript.CSharp;

var text1 =
@"
using Test;

// top-level statements: can be used in one unit only
TestClass.Testing();
TestClass.Tested();
";

var text2 =
@"
using System;

namespace Test
{
    public class TestClass
    {
        public static void Testing()
        {
            Console.WriteLine(""Testing..."");
        }

        public static void Tested()
        {
            Console.WriteLine(""Tested"");
        }
    }
}
";

var unit1 = new CompilationUnit("unit1.cs", text1);
var unit2 = new CompilationUnit("unit2.cs", text2);
var script = new CSharpScript();

if (script.Compile(unit1, unit2))
{
    script.RunMain();
}
```

Ограничение доступного API

В скрипте можно использовать любой API, доступный в вашем .NET приложении. Если скрипт может быть получен из ненадежного источника, это ставит проблему с безопасностью на первый план. FastScript .NET позволяет ограничить использование небезопасного API, такого, как операции с файловой системой или сетью. Вы можете ограничить использование целых сборок, пространств имен или отдельных типов.

Используйте свойство `Script.TypeProvider` для управления загрузкой типов .NET. Оно имеет следующие свойства:

- `IncludeAssemblies` : свойство типа `string[]` содержит список имен сборок. Типы в этих сборках будут доступны в скрипте. Если список пустой, будут доступны типы во всех сборках, которые загружены в вашем приложении .NET, кроме тех, что указаны в свойстве `ExcludeAssemblies` .
- `ExcludeAssemblies` : свойство типа `string[]` содержит список имен сборок. Типы в этих сборках будут недоступны в скрипте.
- `ExcludeNamespaces` : свойство типа `string[]` содержит список пространств имен. Типы в этих пространствах имен (а также в их дочерних пространствах) будут недоступны в скрипте.

Рассмотрим пример, ограничивающий типы в сборках и пространстве имен `System.IO` :

```
using FastScript.CSharp;

var text =
@"
using System;

Console.WriteLine(""OK"");

// this will fail with API restriction
var dir = System.IO.Directory.GetCurrentDirectory();
Console.WriteLine(""Current dir: "" + dir);
";

var script = new CSharpScript();

// use types defined in these two assemblies only
script.TypeProvider.IncludeAssemblies = ["System.Private.Corelib", "System.Console"];

// restrict usage of System.IO namespace and its types
script.TypeProvider.ExcludeNamespaces = ["System.IO"];

if (!script.Compile(text))
{
    foreach (var err in script.Diagnostic.Errors)
    {
        Console.WriteLine($"({err.Line},{err.Column}): error {err.Code}: {err.Message}");
        Console.WriteLine(err.SourceCode());
    }
    return;
}

script.RunMain();
```

При запуске кода вы получите ошибку:

```
(7,17): error cs69: The name 'IO' does not exist in the current context
var dir = System.IO.Directory.GetCurrentDirectory();
                ^^^
```

Для ограничения использования отдельных классов вы можете применить следующий прием.

Допустим, нужно запретить использование класса `System.IO.File`, но оставить возможность использования других классов в пространстве имен `System.IO`. В этом случае ограничение с помощью свойства `Script.TypeProvider.ExcludeNamespaces` не поможет. Добавьте в ваш скрипт следующий код (это можно сделать, дописав код в конец скрипта, либо поместив его в отдельный модуль - compilation unit):

```
namespace System.IO
{
    public class File { }
}
```

Этот скрипт заменит оригинальный класс `System.IO.File` новым, пустым классом. При попытке использовать методы или свойства оригинального класса будет выдана ошибка времени компиляции.

Использование FastScript в Native AOT

В скрипте можно использовать только типы, доступные в хосте. Приложение, скомпилированное в режиме Native AOT, может не содержать некоторые типы (или члены типов), которые вы бы хотели использовать в скрипте, потому что тип/член был исключен из сборки (trimmed).

Другая проблема - использование обобщенных типов/методов. В Native AOT, вы можете использовать те обобщенные типы, которые доступны в хосте. Например, хост использует класс `List<int>`, но не пользуется классом `List<double>`. Первый можно будет использовать в скрипте, но при попытке создать тип `List<double>` будет выдана ошибка.

Таким образом, ваша задача будет состоять в том, чтобы сделать типы (и члены типов) статически доступными в вашем приложении, чтобы их можно было использовать в скрипте. Это можно сделать различными способами (создание экземпляров типов, использование атрибутов):

```
[DynamicDependency(DynamicallyAccessedMemberTypes.All, typeof(List<>))]
public void EnsureAOTVisible()
{
    var list = new List<int>();
}
```

Если в хосте статически доступен обобщенный тип, то поддерживается его использование с параметрами, представляющими reference type. Например, если доступен тип `List<>`, то можно создавать типы `List<object>`, `List<string>`, `List<ArrayList>` и т.п.