

**Богдан Стефанюк**

Нотатки про програмування, музику, подорожі та півку

[Про мене](#) • [Список нотаток](#) • [Плівка](#)

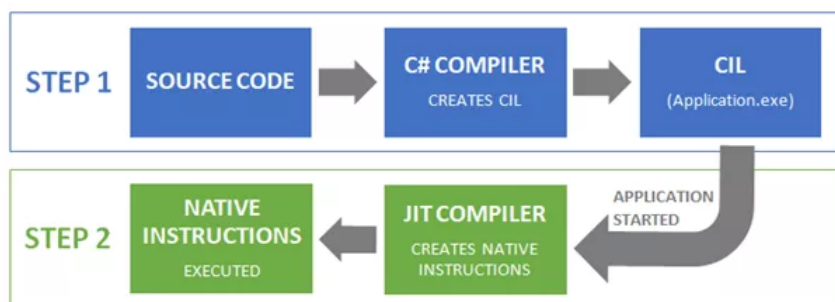
C# Debug vs Release. Сборки и дебаг

Оригинальная статья: [ТЫК](#).

Из коробки в C# нам доступны 2 способа сборки проекта release и debug.

О компиляции C# кода

Исходный код C# проходит через 2 этапа компиляции, чтобы стать инструкциями CPU, которые могут быть выполнены.



Обычно первый этап происходит на вашем CI сервере, а второй шаг происходит позже, во время работы самого приложения. Когда же мы работаем локально в Visual Studio, то она все эти шаги выполняет перед запуском приложения из меню Debug.

Шаг первый. Компиляция приложения

Ваш код превращается в Common Intermediate Language (CIL), который уже может быть выполнен в любом окружении, которое поддерживает CIL. Обратите внимание, что собранная сборка не является читаемым текстом IL, а фактически метаданными и байтовым кодом в виде двоичных данных.

На данном шаге будет выполнена некоторая оптимизация кода (будет описано дальше).

Шаг второй. JIT компилятор

JIT компилятор конвертирует IL код в инструкции процессора, которые можно выполнить на вашей машине. Однако не вся компиляция происходит заранее — в нормальном режиме, код компилируется, только тогда когда его вызывают в первый раз, после чего он кэшируется.

Компилятор JIT — это всего лишь один из целого ряда сервисов, которые составляют Common Language Runtime (CLR), позволяя ему выполнять код .NET.

Основная часть оптимизации кода будет проведена на этом шаге.

Что такое оптимизация кода в одном предложении?

Это процесс улучшения таких факторов, как скорость выполнения, размер кода, энергопотребление, а в случае .NET — время, которое требуется для компилятора JIT — без изменения функциональности.

Почему мы заинтересованы в оптимизации в этой статье?

На обоих этапах компиляции ваш код будет оптимизирован компиляторами. Одно из ключевых различий между конфигурациями сборки Debug и Release заключается в том, отключены ли оптимизации или нет, поэтому нам нужно понять последствия данных оптимизаций.

Оптимизация компилятора C#

C# компилятор на самом деле делает очень мало оптимизаций. На самом деле большинство оптимизаций производит JIT компилятор во время генерирования машинного кода. Тем не менее это все равно ухудшит работу по отладке.

Инструкция `nop` в IL

Команда `nop` имеет ряд применений при программировании на низком уровне, например, для включения небольших предсказуемых задержек или инструкции по перезаписи, которые вы хотите удалить. В IL коде данные конструкции помогают при использовании точек останова (breakpoints) для того чтобы они вели себя предсказуемо.

Если мы посмотрим на IL код, который сгенерирован с отключенными оптимизациями:

```
IL_001f: nop
```

Эта инструкция мапится с фигурной скобкой, для того чтобы мы могли поставить на нее точку остановки:



Данная инструкция была бы удалена если у нас включены оптимизации, что повлияло бы на отладку приложения.

Более подробное обсуждение оптимизаций компилятора C# в статье Эрика Липперта: [Что делает переключатель оптимизации?](#). Существует также хороший комментарий о IL до и после оптимизации [здесь](#).

Оптимизация JIT компилятора

Несмотря на то, что он быстро выполняет свою работу во время выполнения, компилятор JIT также выполняет множество оптимизаций. О его внутренних деталях мало информации. Даже во время работы вашего приложения он производит профилирование и, возможно, перекомпилирование кода для повышения производительности.

Хорошие примеры оптимизации с помощью JIT компилятора можете посмотреть [здесь](#).

Я рассмотрю один пример, чтобы проиллюстрировать влияние оптимизации на отладку.

Встраивание методов

Для реальной оптимизации, сделанной компилятором JIT, я буду показывать инструкции по сборке. Это всего лишь макет на C #, чтобы дать вам общую идею.

Предположим, что у меня есть:

```
private long Add(int a, int b)
{
    return a + b;
}

public void MethodA()
{
    var r = Add(a, b);
}
```

Компилятор JIT, скорее всего, выполнит встроенное расширение. Он заменит вызов метода `Add()` телом данного метода:

```
public void MethodA()
{
    var r = a + b;
}
```

Конфигурации сборки по умолчанию

Итак, теперь, когда мы обновили понимание компиляции .NET и двух «слоев» оптимизации, давайте взглянем на 2 конфигурации сборки, доступные «из коробки»:

BUILD CONFIG	BUILD PROPERTIES	EQUIVALENT ON COMMAND LINE	C# COMPILED IL CODE OPTIMISED?	JIT COMPILED NATIVE CODE OPTIMISED?
Debug	<input type="checkbox"/> Optimize code Output Debugging information: Full	optimize- <u>debug:full</u>	FALSE	FALSE
Release	<input checked="" type="checkbox"/> Optimize code Output Debugging information: Pdb-only	optimize+ <u>debug:pdbonly</u>	TRUE	TRUE

Довольно просто — релиз полностью оптимизирован, отладка совсем отсутствует, что, как вы сейчас знаете, имеет фундаментальное значение для того, насколько легко отлаживать ваш код. Но это просто поверхностное представление о возможностях аргументов отладки и оптимизации.

Внутренности аргументов оптимизации и отладки

Я попытался продемонстрировать данные аргументы из кода Roslyn и mscorlib. Теперь мы имеем следующие классы:

1. [CSharpCommandLineParser](#)
2. [CodeGenerator](#)
3. [ILEmitStyle](#)
4. [debuggerattributes](#)
5. [Optimizer](#)
6. [OptimizationLevel](#)

Синие элементы обозначают аргументы командной строки, а зеленые их представление в коде.



Перечисление OptimizationLevel

OptimizationLevel.Debug отключает все оптимизации для C# и JIT компилятора с помощью `DebuggableAttribute.DebuggingModes`, который с помощью `ildasm`, мы можем видеть:

```
.DebuggableAttribute/*01000003*//DebuggingModes/*01000004*//) /* 0A000003 */ = ( 01 00 07 01 00 00 00 00 )
```

OptimizationLevel.Release включает все оптимизации (`DebuggableAttribute.DebuggingModes = (01 00 02 00 00 00 00 00)`) что в свою очередь соответствует `DebuggingModes.IgnoreSymbolStoreSequencePoints`

При этом уровне оптимизации точки останова могут быть оптимизированы. Что приведет к тому что мы не сможем их поставить и остановиться.

Типы IL

Типы IL кода описаны в классе [ILEmitStyle](#).

На диаграмме выше показано, что тип генерируемого IL кода C# компилятором зависит от OptimizationLevel.

Аргумент *debug* не меняет его, за исключением аргумента *debug+* когда OptimizationLevel установлен в Release.

ILEmitStyle.Debug — нету оптимизация IL в дополнение к добавлению пор инструкций для сопоставления точек останова с IL.

ILEmitStyle.Release — полная оптимизация.

ILEmitStyle.DebugFriendlyRelease — выполняет только те оптимизации, которые не мешают отладке приложения.

Следующий кусок кода продемонстрирует все это наглядно.

```
if(optimizations == OptimizationLevel.Debug)
{
    _ilEmitStyle = ILEmitStyle.Debug;
}
else
{
    _ilEmitStyle = IsDebugPlus() ?
        ILEmitStyle.DebugFriendlyRelease :
        ILEmitStyle.Release;
}
```

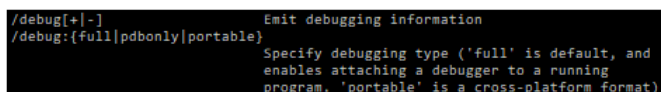
Комментарий в исходном файле Optimizer.cs гласит, что они не опускают никаких определенных пользователем локальных переменных (примеры на 28 строчке) и не переносят значения в стек между операторами.

Я рад, что прочитал это, так как я был немного разочарован своими собственными экспериментами в ildasm с debug +, поскольку все, что я видел, это сохранение локальных переменных.

Нет намеренной «деоптимизации», например, добавления команд пор.

Разница между debug, debug:full и debug:pdbonly.

На самом деле разницы никакой нету, что подтверждает официальная документация:



```
/debug[+|-] Emit debugging information
/debug:{full|pdbonly|portable} Specify debugging type ('full' is default, and
enables attaching a debugger to a running
program. 'portable' is a cross-platform format)
```

Результат остается одним и тем же — pdb файл создается.

Подглянув в [CSharpCommandLineParser](#) можем убедиться в этом. И для того чтобы проверить это, мне удалось отладить код с помощью WinDbg для обоих аргументов (pdbonly и full).

Они не влияют на оптимизацию кода.

Как плюсом могу отметить что [документация на Github](#) более актуальна, но она не открывает свет на специфическое поведение для *debug+*

А что же такое этот ваш pdb файл?

Все очень просто, данный файл содержит всю необходимую информацию для отладки DLL и EXE. Что помогает сопоставить отладчику IL код с инструкциями в оригинальном C# коде.

Что на счет debug+?

debug+ это особенный аргумент, который не может быть заменен с помощью *full* и *pdbonly*. Как многие заметили, данный аргумент соответствует *debug:full* — это на самом деле не совсем правда. Если мы используем аргумент *optimize-*, поведение будет таким же, но для *optimize+* будет свое собственное уникальное поведение (DebugFriendlyRelease).

debug- или без аргументов?

Значения по умолчанию, которые установлены в [CSharpCommandLineParser.cs](#).

```
bool debugPlus = false;
bool emitPdb = false;
```

а значения для *debug*:=:

```
case "debug-":
    if (value != null)
        break;

    bool emitPdb = false;
    bool debugPlus = false;
```

Таким образом, мы можем с уверенностью сказать, что *debug*- и отсутствие аргументов отладки, приведет к одному и тому же эффекту — *pdb* файл не будет создан.

Запрет оптимизации JIT при загрузке модуля (Suppress JIT optimizations)

Флажок в разделе «Options->Debugging->General» это опция для отладчика в Visual Studio и не повлияет на сборки, которые вы создаете.

Вы должны теперь оценить, что компилятор JIT делает большую часть значительных оптимизаций и является большим препятствием для сопоставления исходного исходного кода для отладки. Если этот параметр включен, тогда он запросит *DisableOptimizations*.

Обычно этот параметр включаю для того чтобы отладить внешние библиотеки или пакеты NuGet.

Если мы хотим подключиться отладчиком Visual Studio к продакшен сборке, которая собрана в релизной конфигурации, то при наличии у нас *pdb* файла, мы может еще одним способом указать JIT компилятору, о том чтобы он не оптимизировал код. Для этого нужно добавить *.ini* файл с таким же названием как выполняемая библиотека и указать в нем:

```
[.NET Framework Debugging Control]
AllowOptimize=0
```

Что такое Just My Code?

По умолчанию эта настройка уже включена (Options->Debugging→Enable Just My Code) и отладчик думает что оптимизированный код не является пользовательским. Поэтому отладчик никогда не зайдет в такой код.

Вы можете отключить данный флаг и это, теоретически, позволит вам поставить точку остановки. Но теперь вы отлаживаете код, оптимизированный как компиляторами C# так и JIT, который едва соответствуют исходному коду. В результате у вас будут такие артефакты отладки, как непредсказуемые переходы в коде и скорее все вам не получится получить значения в локальных переменных.

Этот параметр стоит отключать в том случае, когда у вас есть *pdb* файл.

Взглянем ближе на DebuggableAttribute

Выше я упомянул использование *ildasm* для изучения манифеста сборок для изучения *DebuggableAttribute*. Я также написал небольшой PowerShell скрипт для получения более дружелюбного результата ([ссылка на скачивание](#)).

Сборка Debug:

```
Assembly      : NoOptimiseDebugFull
JITTrackingEnabled : True
JITOptimizerDisabled : True
DebuggingFlags : Default, IgnoreSymbolStoreSequencePoints, EnableEditAndContinue,
                DisableOptimizations
```

Сборка Release:

```
Assembly      : OptimiseDebugPdbOnly
JITTrackingEnabled : False
JITOptimizerDisabled : False
DebuggingFlags : IgnoreSymbolStoreSequencePoints
```

Вы можете игнорировать `IsJITTrackingEnabled`, поскольку он был проигнорирован компилятором JIT с .NET 2.0.

Компилятор JIT всегда будет генерировать информацию для отслеживания во время отладки, чтобы сопоставлять IL с машинным кодом и отслеживать, где хранятся локальные переменные и аргументы функции.

IsJITOptimizerDisabled — просто проверяет `DebuggingFlags` на наявность `DebuggingModes.DisableOptimizations`. Он отвечает за включение оптимизация с помощью JIT.

DebuggingModes.IgnoreSymbolStoreSequencePoints — говорит отладчику выработать точки последовательности из IL кода вместо загрузки .pdb файла. Точки последовательности используются для сопоставления местоположений в коде IL с местоположениями в исходном коде C#. Если он включен, то JIT не будет загружать .pdb файл. Я не уверен, почему этот флаг добавляется в оптимизированные сборки компилятором C#.






Также об этом флаге можно почитать [здесь](#)

Ключевые понятия

1. *debug*- или отсутствие аргумента приводит к тому что не создается .pdb файл.
2. *debug*, *debug:full* и *debug:pdbonly* приводят к созданию .pdb файла. *debug+* делает тоже самое в случае когда установлен флаг *optimize-*.
3. *debug+* и *optimize+* создают такой IL код, который легче отлаживать.
4. Каждый слой оптимизации ухудшает отладку кода.
5. С .NET 2.0 компилятор JIT всегда будет генерировать информацию для отслеживания независимо от атрибута `IsJITTrackingEnabled`.
6. Будь то сборка через VS или `csc.exe`, атрибут `DebuggableAttribute` теперь всегда присутствует.
7. *optimised+* создает бинарники, которые отладчик воспринимает как сторонний код. Это поведение управляется с помощью опции Just My Code, но при ее отключении вы можете получить очень сомнительный опыт отладки.

Теперь у вас есть выбор:

1. Debug: `debug|debug:full|debug:pdbonly optimize+`
2. Release: `debug-|no debug argument optimize+`
3. DebugFriendlyRelease: `debug+ optimize+`

 Надіслати  Твітнути  Поділитись  Поділитись  Запінити

 6234 2018 dotnet [перевод](#)

Ctrl ←
Рубрикатор

Ctrl →
[Кратко о WebSocket](#)

Ваш коментар

Увійти через     

Надіслати

Ctrl + Enter