IT's TINKOFF

DOTNEXT

.NET: AOT in 2022

Евгений Пешков

16.06.2022

Ahead-of-time компиляция

- .NET работает в модели JIT компиляции:
  - Текст программы компилируется в IL-код
  - При выполнении программы IL-код компилируется в нативный код
- AOT – компиляция (части) кода программы сразу в нативный код, до запуска приложения

План

- Зачем нужна AOT компиляция
- Инструменты для AOT
- Сбор метрик по JIT и AOT
- Опыт использования AOT в JetBrains Rider
- Недостатки средств и подходов к AOT-компиляции

Зачем нужен AOT: недостатки JIT

- Занимает время на старте приложения (особенно, GUI)
  - Предкомпилировать часть кода в нативный
- Ограничен во времени и эффективности кода
  - Применить ресурсоёмкие оптимизации
- Поддерживается не на всех платформах (iOS)
  - Скомпилировать весь код в нативный, избавившись от JIT

AOT tools overview

Дополняющие JIT:

- Ngen
- ReadyToRun (crossgen2)
- Mono AOT

Нативные:

- Mono Full-AOT
- IL2CPP
- Burst (Unity)
- Native AOT (ex. CoreRT)
- LLVM backend
- UWP (.NET Native)

Другие:

- Blazor AOT

AOT tools overview

Дополняющие JIT:

- Ngen
- ReadyToRun (crossgen2)
- Mono AOT

Нативные:

- Mono Full-AOT
- IL2CPP
- Burst (Unity)
- Native AOT (ex. CoreRT)
- LLVM backend
- UWP (.NET Native)

Другие:

- Blazor AOT

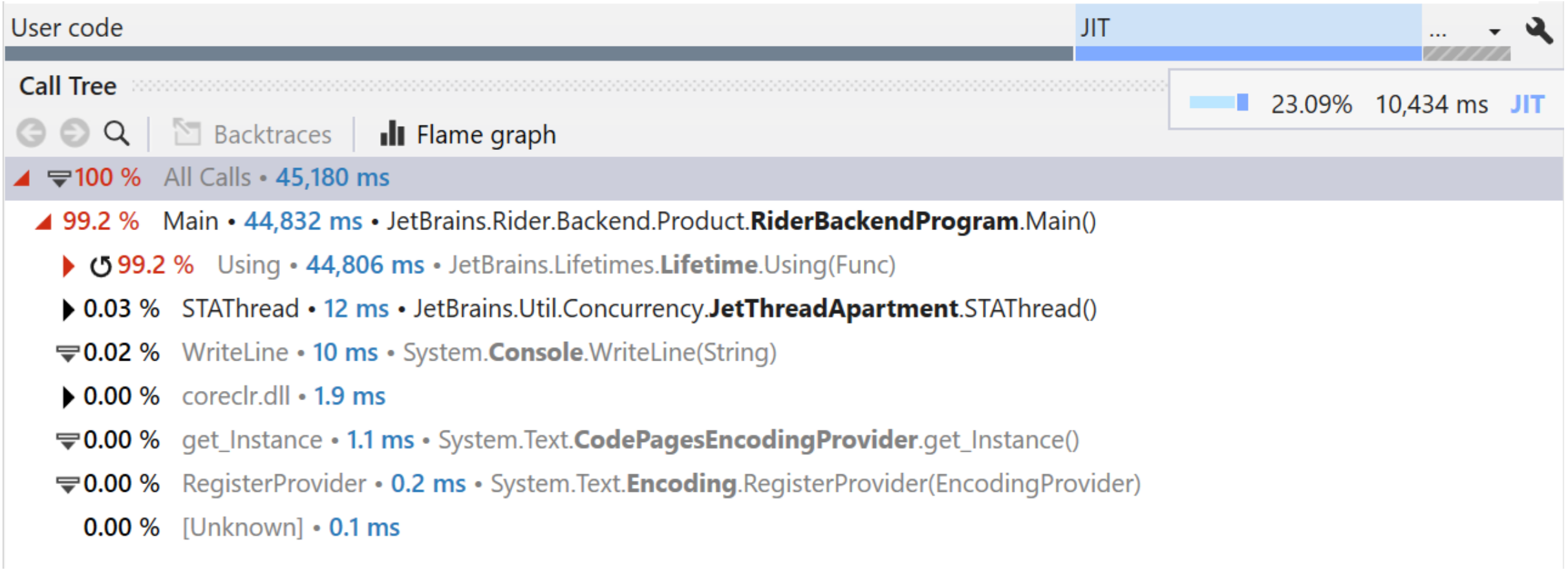
AOT для ускорения старта

Замеры времени JIT

- Основной инструмент: runtime events
  https://docs.microsoft.com/en-us/dotnet/fundamentals/diagnostics/runtime-method-events
- Собираются с помощью: ETW/EventListener/dotnet trace

- MethodJittingStarted
- MethodLoad/MethodLoadVerbose

Замеры времени JIT: dotTrace Timeline



Замеры времени JIT: PerfView JITStats

Jitting Trigger	Num Compilations	% of total jitted compilations	Jit Time msec	Jit Time (% of total process CPU)
TOTAL	102166	100.0	35472.5	68.5
Foreground	83374	81.6	31898.2	61.6
Multicore JIT Background	8329	8.2	1590.1	3.1
Tiered Compilation Background	10463	10.2	1984.1	3.8
Main Thread Foreground	46559	45.6	17081.9	33.0

Виды JIT

- Foreground JIT – блокирующий, при первом обращении к типу/методу
- Main Thread Foreground – то же, но на основном потоке
- Multicore JIT Background – предиктивная компиляция на основе данных от предыдущих запусков (ProfileOptimization)
- Tiered Compilation Background – перекомпиляция часто используемых методов ради повышения эффективности кода (TieredCompilation)

Plot twist

- Все эти инструменты – бесполезны для приложений, которые имеет смысл ускорять с помощью AOT

Plot twist

- Все эти инструменты – бесполезны для приложений, которые имеет смысл ускорять с помощью AOT
- Реальность и профиляция расходятся в ~10 раз

Metric	PerfView	Real world
SolutionLoad[SolutionContainer]	11s	1.7s
SWEA.FirstCycle	0.16s	0.07s
ComponentContainer[SolutionInstanceContainer]	1.2s	0.2s
QuickFixesRegistration	1s	0.3s

Plot twist

- При замерах результативности AOT важны только метрики приложения
- ETW и профилирование – только вспомогательные инструменты

Crossgen2 и ReadyToRun

- AOT для современного .NET – общего назначения, т.е. работает в паре с JIT

- ReadyToRun (R2R) – формат данных, содержащих нативный код для .NET assemblies
  https://github.com/dotnet/runtime/blob/main/docs/design/coreclr/botr/readytorun-format.md
- crossgen/crossgen2 – утилиты для генерации R2R данных
  https://www.nuget.org/packages/Microsoft.NETCore.App.Crossgen2.win-x64
- ReadyToRun-образ – сборка, содержащая нативный код в R2R формате

Crossgen2

- Утилита для генерации R2R кода для .NET 6 и поздних
- Managed, использует JIT
- Поддерживает кросс-компиляцию под другие ОС

Ready2Run lifetime

Ready2Run сосуществует с JIT:

- AOT-компиляция до запуска
- Старт с использованием AOT
- JIT-компиляция недостающих методов (Tier0)
- Перекомпиляция горячего кода в Tier1

Использование crossgen

- Из коробки
  - Стандартная библиотека .NET уже предкомпилирована с помощью crossgen

Использование crossgen2.exe

- Через параметры dotnet publish
  - Наиболее стабильный вариант
  - Подходит только для стандартных .NET приложений, с несильно кастомизированным билдом

```
<PublishReadyToRun>true</...>
<PublishReadyToRunComposite>true</...>
<PublishReadyToRunExtraArgs>--arg1;--arg2</...>
```

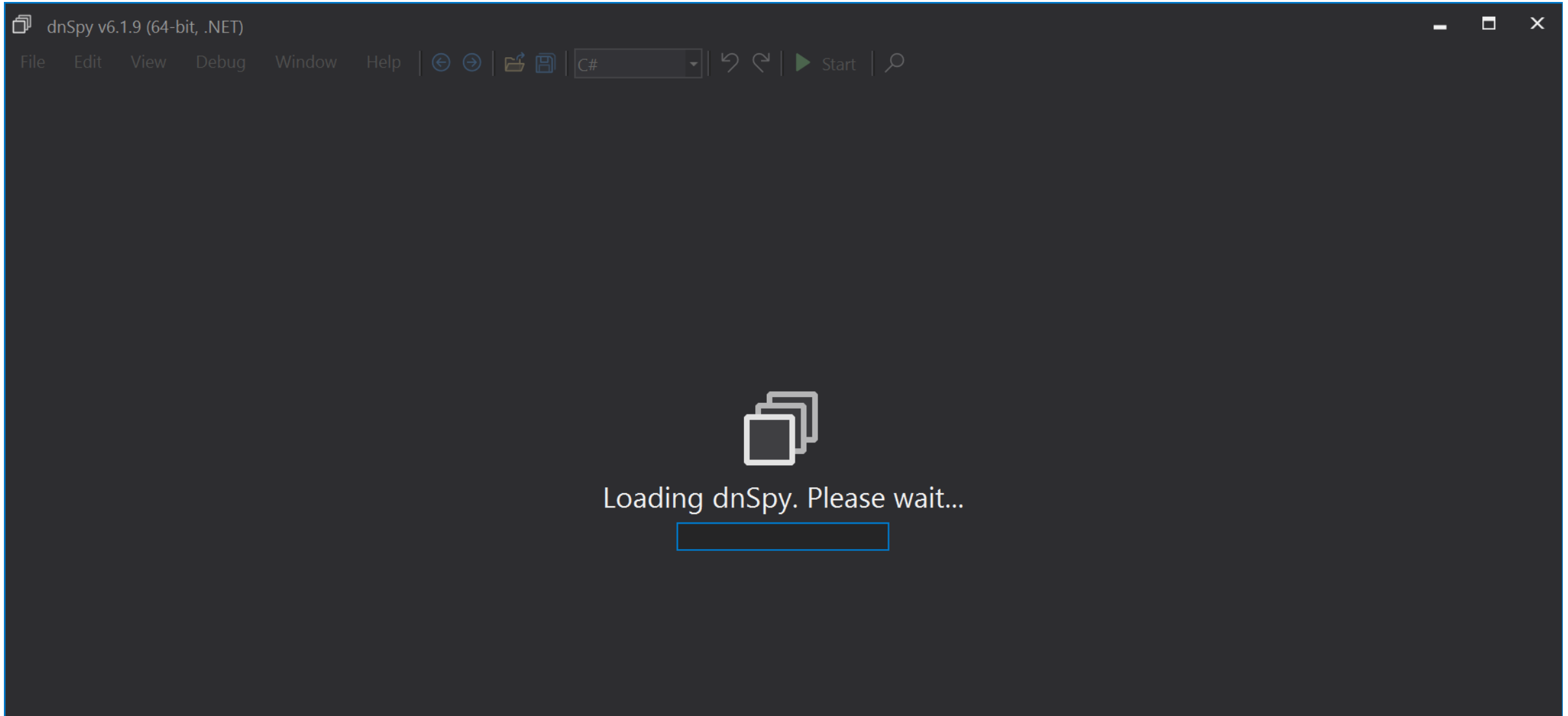
Использование crossgen2.exe

- Вызов утилиты напрямую
  - Сложный запуск
  - Подходит для CI и компиляции на компьютере пользователя
  - Позволяет внедрить AOT-компиляцию «сбоку»
  - Утилиту можно найти в NuGet или собрать dotnet/runtime
    - nuget: Microsoft.NETCore.App.Crossgen2.[rid]

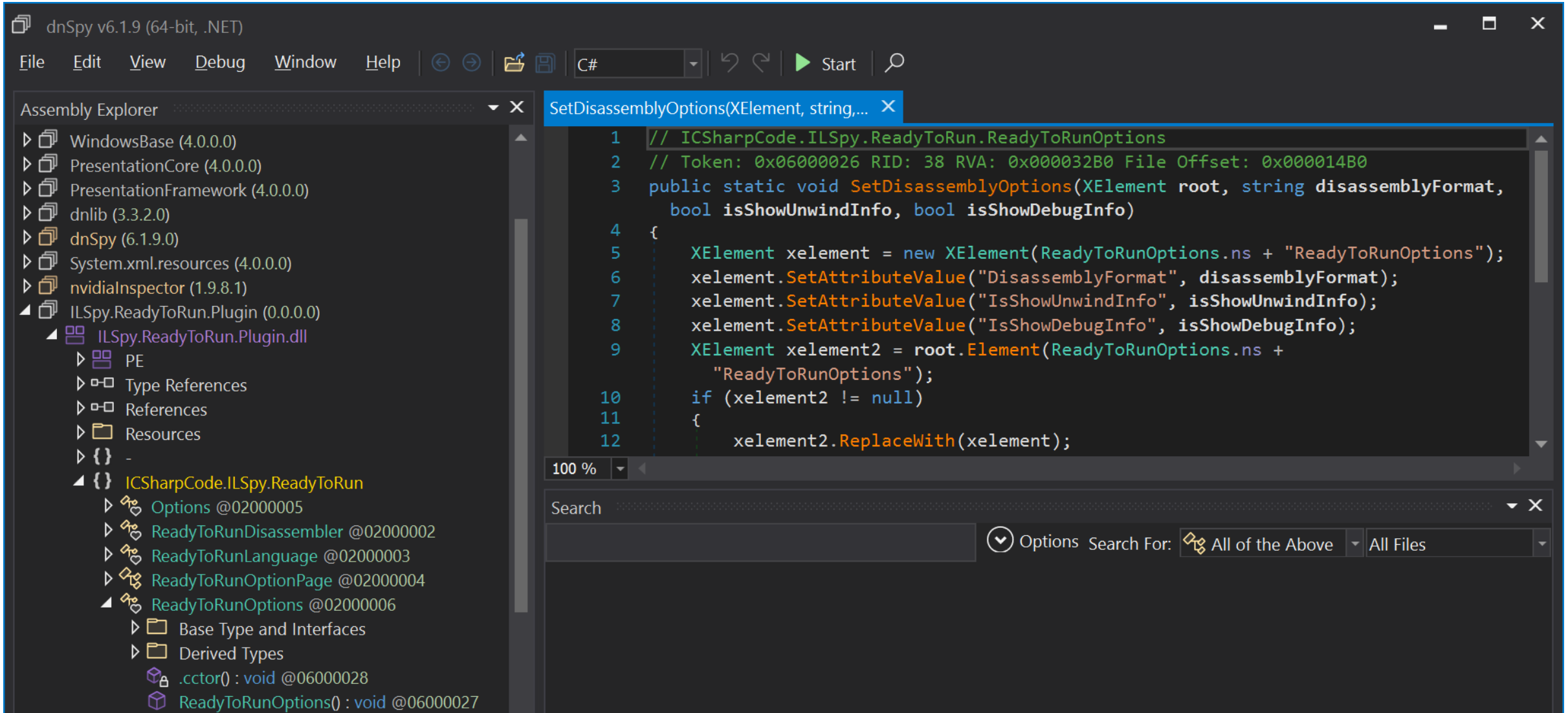
Режимы crossgen2

- Single-file-compilation:
  - Все сборки компилируются по-отдельности
  - Нативный код хранится внутри ReadyToRun header
  - Возможности инлайнинга сильно ограничены
- Composite:
  - Нативный код для набора сборок хранится в отдельном файле AppName.r2r.dll
  - Межсборочный инлайнинг (version bubble)
  - Сборки при билде должны быть такими же, как в рантайме

Тестовое приложение: dnSpy



Тестовое приложение: dnSpy



dnSpy: цена кроссгена

- Ускорение старта – незаметное:
  - 200 мс – обычный crossgen
  - 300 мс – composite
- Артефакт увеличился на:
  - 57 MB – обычный crossgen (x2, не считая ресурсов)
  - 190 MB – composite
- Весь IL-код и метаданные сохраняются
- При кроссгене в composite приложение вначале упало с ошибкой (из-за DirectWriteForwarder.dll)

Уменьшение размера

- Исключение части сборок из компиляции
- --optimize-space – не помогает
- Тримминг
- AOT-компиляция на компьютере пользователя, аналогично NGEN (требует ручного запуска crossgen2)
- Самый эффективный способ – сжатие артефакта
  - Оверхед уменьшился с 57 МБ до 20 МБ

AOT для ускорения старта: Rider IDE

Crossgen JetBrains Rider

- Задача:
  - Отказаться от .NET Framework на Windows
  - Для этого обойти по скорости .NET Framework + Ngen
  - Также ускорить время старта IDE

- Выглядит как повод использовать crossgen2

Проблема 1. Баги

- AOT – непопулярная фича
- Большинство тулов – экспериментальные

- Поэтому:
  - Будьте готовы контрибьютить в AOT-тулинг
  - Тестируйте свои приложения
  - Будьте готовы, что используемую вами тулу перестанут поддерживать

Проблема 2. Crossgen работает медленно

- N сборок → N запусков утилиты crossgen2
- Узкое место — загрузка сборок-зависимостей
- Простое распараллеливание не работает

- Решение: компилировать несколько сборок за один запуск crossgen2
- На .NET 7 такой способ работает хорошо, а на .NET 6 — работает только с DLL файлами

Проблема 3. Хранение R2R-образов

- Сборка, содержащая R2R-данные становится непригодна для использования в .NET Framework (BadImageFormatException)

- Казалось бы, можно хранить R2R в отдельном composite-файле
- Но даже в таком случае managed сборки содержат хедер-ссылку на composite файл

```
Type: OwnerCompositeExecutable (116)
Composite executable: "Rider.Backend.r2r.dll"
```

Проблема 3. Хранение R2R-образов

- Можно хранить R2R образы отдельно, в *.ni.dll/*.ni.exe или выносить в другие локации с помощью *.deps.json

- Но тогда приложение при использовании рефлексии все равно получит R2R-образы вместо оригинальных сборок
- R2R-образы никак не абстрагированы и являются неотъемлемой частью приложения

Проблема 3. Хранение R2R-образов

- Оглядываясь назад – именно эта проблема стала основной, т.к.:
  - требовала переработки билд системы
  - переработки инсталлера R#+Rider
  - увеличивала занимаемое место на диске
  - «заражала» сборки артефактами AOT
- В идеале хотелось бы подключать AOT-оптимизацию без модификации приложения

Проблема 4. Падение производительности

- Улучшения от кроссгена видны на компьютерах разработчиков, но на тестовой площадке метрики ухудшились
- Тестовая площадка – Mac Mini, начала 2010 годов
- **В чём причина?**

Проблема 4. Падение производительности

- Процессоры на тестовой площадке не поддерживали AVX2, а R2R-образы были собраны с требованием к поддержке этих инструкций
- Неудачная загрузка R2R-образа оказалась дорогой операцией
- Решение: собирать с настройками по-умолчанию, разницы в производительности замечено не было

Проблема 5. Падение производительности

- Crossgen образы могут включать код под разные наборы инструкций: под требуемый и оптимистичный
  - Требуемый по-умолчанию: SSE2 (т.е. любой X64)
  - Оптимистичный по-умолчанию: SSE4.2
- Код все равно будет ReJIT’иться при необходимости
- Не до конца ясно, что меняет заданный Instruction Set в нативном коде

Проблема 5. Нет эффекта на Linux/MacOS

- На Windows R2R даёт выигрыш по производительности
- На Linux и MacOS никакой разницы нет

- На .NET 7 preview 1 это было исправлено
- Тестируйте AOT на всех целевых платформах

Rider: итоги

- Rider перешел на .NET 6 теперь и на Windows
- AOT с помощью crossgen2 позволил сравняться по времени старта с .NET Framework (произошло уже без меня)
- Доступно в 2022.2 EAP4

- Запускается на компьютере пользователя
- Single-file compilation

Утилиты для ReadyToRun

PerfView/TraceEvent/EventListener

- R2RGetEntryPoint (EntryPoint != 0) event
- Позволяет понять, загружаются ли R2R-методы, сколько их

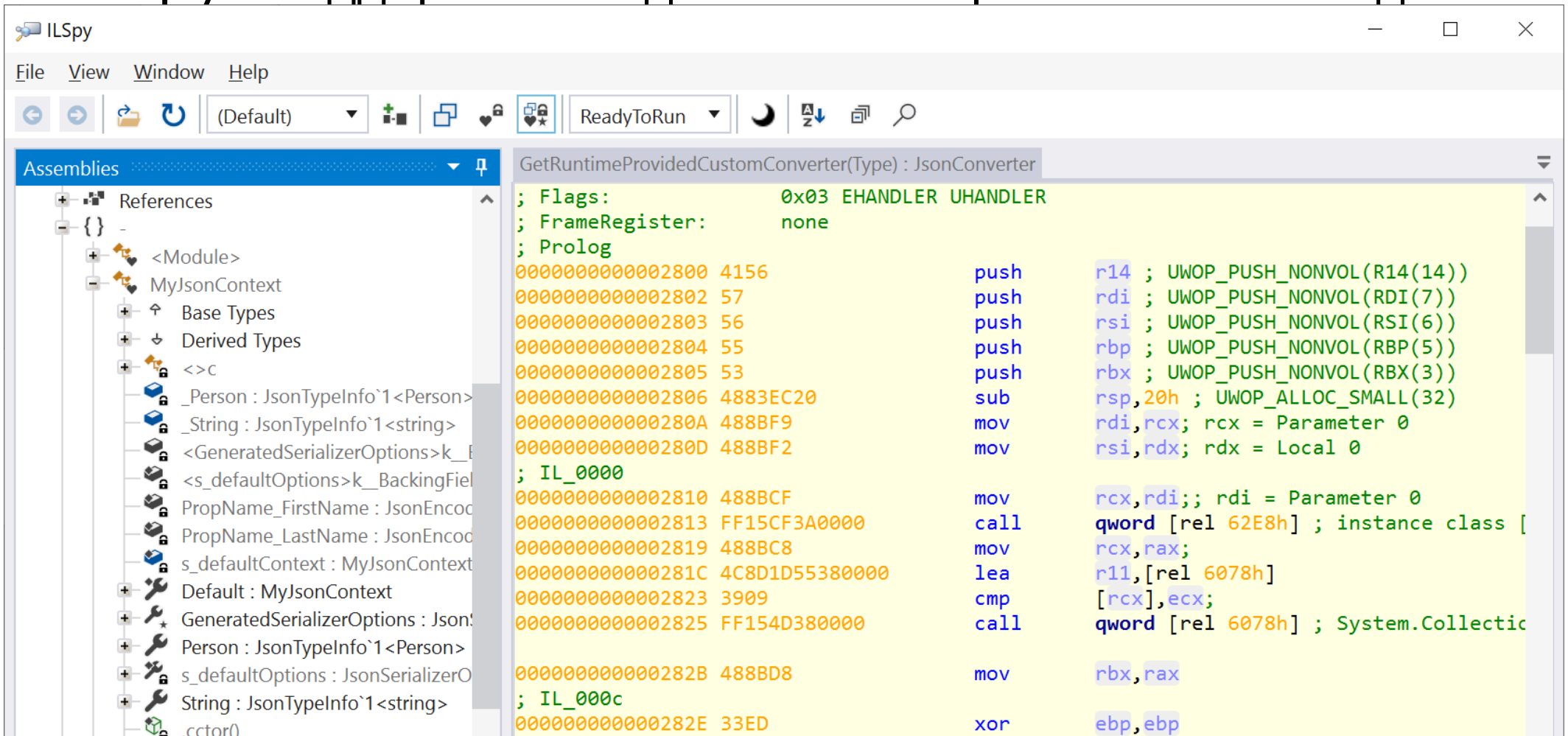
Утилиты для ReadyToRun

R2RDump

- Позволяет просматривать содержимое R2R-header
- Можно понять, что скомпилировал crossgen

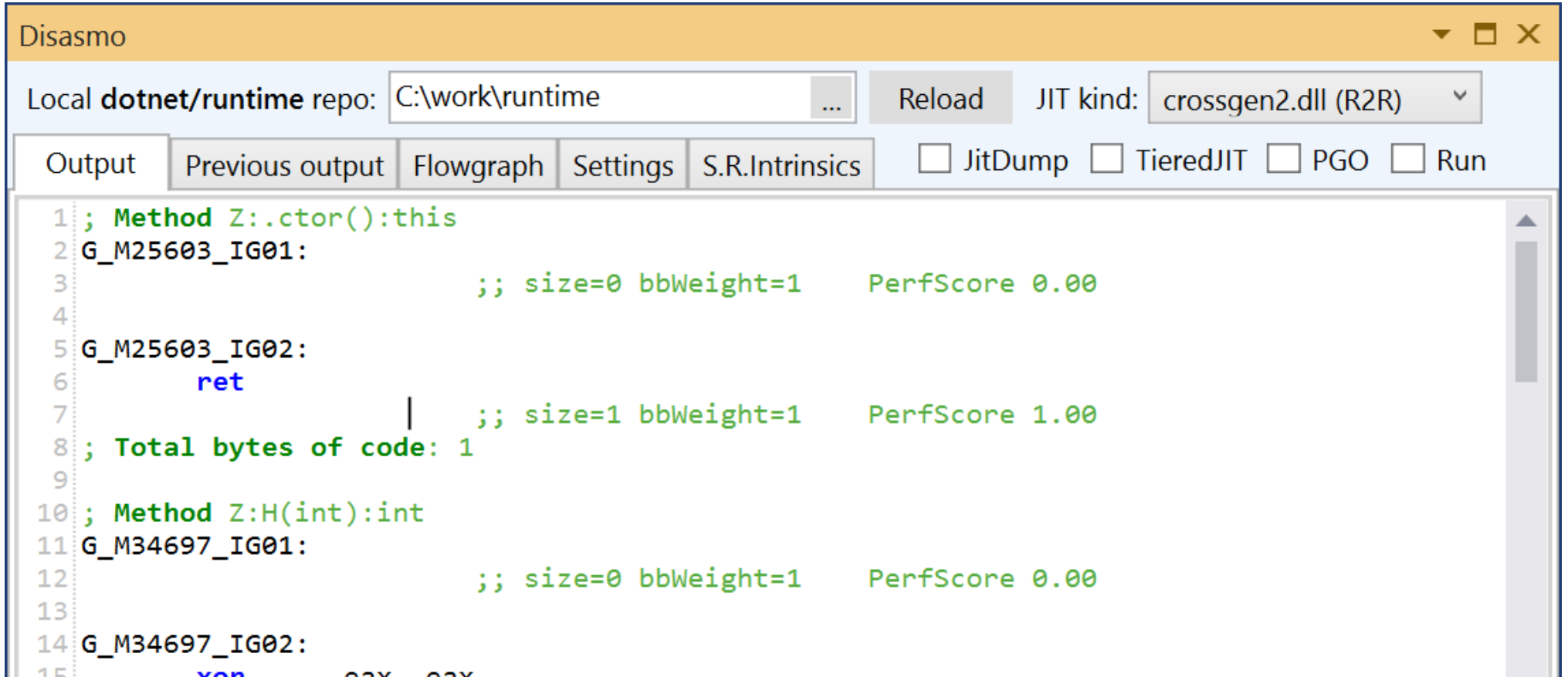
Утилиты для ReadyToRun

ILSpy: Поддерживает дизассемблирование R2R-кода



Утилиты для ReadyToRun

Disasmo (@EgorBo): Плагин для Visual Studio, позволяющий дизассемблировать в том числе R2R код



Нативный AOT

NativeAOT

- Отдельный рантайм
- На выходе один *.exe с нативным кодом
- IL не используется
- Trimming сборок приложения

- https://github.com/dotnet/runtime/issues/61231

NativeAOT: что работает

- Console apps
- Shared libraries
- Web Apps (ASP.NET Core)

NativeAOT

```
<ItemGroup>
    <PackageReference
     Include="Microsoft.DotNet.ILCompiler"
     Version="7.0.0-*" />
  </ItemGroup>
```

+ нужен компилятор

Отличия NativeAOT и ReadyToRun

	ReadyToRun	NativeAOT
Задача	Ускорение старта	Нативное приложение
Размер артефакта	Больше	Меньше
Целевое приложение	Любое	Подготовленное
Рантайм	CoreCLR	Собственный

Обычный .NET и Native AOT

С NativeAOT приходят преимущества нативного кода, но исчезают привычные фичи managed рантайма

- .NET – runtime driven
  - Динамическая загрузка сборок
  - Зависимость от рефлексии и кодогенерации
- Поэтому:
  - Приложение должно быть подготовлено
  - (Пока) не все приложения могут быть скомпилированы в натив

Подготовка приложения к NativeAOT

Рефлексия и метаданные:

- Рефлексия – с ограничениями
- Expressions – только интерпретация

Using member 'System.Text.Json.JsonSerializer.Serialize<TValue>(TValue, JsonSerializerOptions)' which has 'RequiresUnreferencedCodeAttribute' can break functionality when trimming application code. JSON serialization and deserialization might require types that cannot be statically analyzed. Use the overload that takes a JsonTypeInfo or JsonSerializerContext, or make sure all of the required types are preserved.

```
public static class JsonSerializer
in namespace System.Text.Json in assembly System.Text.Json (System.Text.Json.dll)
```

Provides functionality to serialize objects or value types to JSON and to deserialize JSON into objects or value types.

`JsonSerializer` on docs.microsoft.com ↗

Подготовка приложения к NativeAOT

Рефлексия и метаданные:

- Добавить всё в артефакт
  - `<IlcGenerateCompleteTypeMetadata>false</...>`
  - `<IlcDisableReflection>true</...>`
- Указать динамически используемые типы в rd.xml
  - https://github.com/dotnet/runtimelab/blob/feature/NativeAOT/docs/using-nativeaot/rd-xml-format.md
- Использовать Source generators

Source generators

- Source generators for real life:
  - Json
  - Regex
  - DI

youtube.com/watch?v=6QWZds35rGs: Андрей Дятлов — Source Generators в действии
youtube.com/watch?v=v8ED9g-rTig: Source generators v2.0 — инкрементальные генераторы

Output size

Json serializing app:

Scenario	*.exe size
Newtonsoft.Json	26 MB
System.Text.Json	5 MB
System.Text.Json source generator	1.9 MB

WebApp:

Scenario	*.exe size
Asp.NET Core server	28 MB

NativeAOT: benchmarking

```
var toolchain = NativeAotToolchainBuilder
   .Create()
   .IlcGenerateCompleteTypeMetadata(true)
   .IlcOptimizationPreference("Speed")
   .UseNuGet(
      microsoftDotNetILCompilerVersion: "7.0.0-preview.5.22301.12")
   .IlcGenerateStackTraceData(true)
   .IlcInstructionSet("avx2")
   .ToToolchain();
```

NativeAOT: benchmarking

```
private Json CreateJson(int deep = 8)
{
  var j = new Json(
    "abcd", "qazwsx",
    random.Next(), random.NextInt64(), random.NextDouble(),
    null, null);

  if (deep > 0)
  {
      j.Inner1 = CreateJson(deep - 1);
      j.Inner2 = CreateJson(deep - 1);
  }
  return j;
}
```

NativeAOT: benchmarking

Toolchain=.NET 7.0

Method	Mean	Error	StdDev
Serialize	147.4 us	0.75 us	0.70 us
Deserialize	386.8 us	1.65 us	1.54 us

Toolchain=ILCompiler 7.0.0-preview.5.22301.12

Method	Mean	Error	StdDev
Serialize	153.9 us	0.55 us	0.52 us
Deserialize	420.3 us	1.94 us	1.81 us

https://github.com/dotnet/runtime/issues/64856

NativeAOT without CoreCLR

NativeAOT – независимый от основного CoreCLR рантайм

C# и платформа .NET – разделяемы

- Можно скомпилировать приложение:
  - Без стандартной библиотеки
  - Без рантайма
  - В отличном от обычного формате

- https://github.com/MichalStrehovsky
  - Self-contained C# game in 8 kB
  - 11 kB UEFI version

  https://twitter.com/KooKiz/status/1533133858312445956
  - Proof-of-concept C# GC, profiler with NativeAOT

Выводы

- AOT действительно работает, пусть и не всегда ожидаемо
- AOT – нишевая технология, пользователей мало
- Не каждое приложение получит пользу от AOT
- Многие утилиты и подходы – экспериментальные

- NativeAOT с source generators – новое развитие платформы .NET

Ссылки

- Dotnext 2018 Piter: Используем AOT-компиляцию правильно
  - https://www.youtube.com/watch?v=uokOGsIw5Ns
- Pro.net chat
  - https://t.me/pro_net
- NativeAOT docs
  - https://github.com/dotnet/runtime/tree/main/src/coreclr/nativeaot/docs
  - https://github.com/dotnet/runtime/issues/61231
  - https://github.com/dotnet/designs/blob/main/accepted/2020/form-factors.md
- Michal Strehovský github
  - https://github.com/MichalStrehovsky
- Other links:
  - https://codevision.medium.com/arm-and-leg-for-nativeaot-363df373e1a2
  - https://github.com/dotnet/runtime/issues/67805

IT's TINKOFF

Евгений Пешков

@ epeshk

https://t.me/epeshkblog


IT's
TINKOFF