Санкт-Петербургский государственный университет

Математическое обеспечение и администрирование информационных систем
Кафедра системного программирования

Усов Илья Дмитриевич

Поддержка лямбда-выражений в отладчике IDE Rider

Выпускная квалификационная работа

Научный руководитель:
ст. преп. Я. А. Кириленко

Рецензент:
тех. лид. проекта Rider компании JetBrains Д. А. Иванов

Санкт-Петербург
2018

SAINT PETERSBURG STATE UNIVERSITY

Software and Administration of Information Systems
Software Engineering

Ilya Usov

Support for lambda-expressions in IDE Rider debugger

Bachelor's Thesis

Scientific supervisor:
Senior Lecturer Iakov Kirilenko

Reviewer:
Project Rider technical lead at JetBrains company Dmitry Ivanov

Saint Petersburg
2018

Содержание

Введение

На сегодняшний день существует огромное количество программ, многие из которых разрабатываются и поддерживаются на протяжении многих лет. К тому же очень часто в разработке участвует не один человек, а целая команда. Порой даже просто разобраться как работает чужой код бывает непросто, не говоря уже о том, чтобы исправить ошибки в этом коде. Для облегчения поиска ошибок существуют специальные программы — *отладчики*. Отладчик позволяет пошагово исполнять программу, устанавливать и убирать точки останова, просматривать и изменять значения переменных прямо в процессе отладки, а также многое другое [11]. Помимо этого большинство отладчиков, позволяют вызывать методы из пользовательского (*target*) процесса и получать их результат. Благодаря всему вышеперечисленному на основе данных, доступных отладчику, можно реализовать вычисление произвольных выражений, написанных на языке программирования. Большинство интегрированных сред разработки (Integrated Development Environment, *IDE*) [12] уже включают в себя отладчик и имеют специальные отладочные окна (*Watches*), с помощью которых и реализуется эта функциональность. Однако, чаще всего отладчики ничего не знают про языки программирования, поэтому вычисление выражений во время отладки реализует среда разработки поверх API отладчика.

Во многих современных языках программирования поддерживаются лямбда-выражения. Они бывают очень удобны, так как позволяют разработчику не тратить время на создание нового метода. А в случаях, когда лямбда-выражение захватывает внешний контекст, и вовсе избавляет разработчика от сложностей, связанных с передачей этого контекста в тело метода. За разработчика это делает компилятор. Однако для отладчика при вычислении значения выражения, которое содержит лямбда-выражение возникают проблемы, так как в пользовательском процессе не существует этого лямбда-выражения и, соответственно, отладчику просто нечего вызывать. Кроме того, специального API, которое по тексту лямбда-выражения создаст в пользовательском

процессе метод, чаще всего нет, так как отладчики обычно к языкам программирования не привязаны.

Наиболее частые случаи использования лямбда-выражений во время отладки: фильтрация коллекций, преобразование элементов коллекции, комбинирование предыдущих случаев.

JetBrains Rider IDE [3] — это кроссплатформенная интегрированная среда разработки для .Net приложений на базе JetBrains IntelliJ Platform [1] и JetBrains ReSharper [2].

На данный момент существует несколько платформ для .Net приложений.

- .Net Full Framework [6].
- .Net Core [5].
- Mono [13].

Разные платформы предоставляют разные отладочные интерфейсы.

- ICorDebug [4] — отладочный интерфейс для приложений работающий под .Net Full Framework и .Net Core.
- Soft-Debugger [8] — отладочный интерфейс для приложений работающий под Mono.

Оба этих интерфейса позволяют подписаться на получение событий, произошедших в отлаживаемом процессе, а также дают возможность повлиять на него.

Так как Rider поддерживает разработку приложений как под .Net/.Net Core, так и под Mono, он имеет два отладчика CorDebugger и SoftDebugger, работающих поверх отладочных интерфейсов ICorDebug и Soft-Debugger соответственно.

1. Постановка задачи

Целью данной работы является реализация решения для вычисления лямбда-выражений во время отладки в IDE Rider. При этом необходимо реализовать это для CorDebugger и SoftDebugger. Также решение должно быть кроссплатформенным.

Данную работу можно разбить на несколько подзадач.

- Сделать обзор существующих механизмов, позволяющих добавлять и изменять код в отлаживаемое приложение «налету». Сделать обзор существующих реализаций поддержки лямбда выражений в отладчиках.
- Разработать архитектуру решения для вычисления лямбда-выражений во время отладки.
- Реализовать решение.
- Провести апробацию.

2. Обзор

2.1. Существующие механизмы изменения и добавления кода «на лету»

2.1.1. Edit And Continue

Edit And Continue [10] — это специальный механизм в .Net, предоставляемый отладочным интерфейсом ICorDebug, который позволяет прямо во время отладки изменить код программы. Для этого необходимо получить разницу (*Delta*) между MSIL-кодом до изменения и после. После этого надо передать эту разницу в общеязыковую среду исполнения (*CLR*).

2.1.2. Динамическая загрузка сборок в платформе .Net

В стандартной библиотеке классов .Net (*BCL*) существует класс *Assembly*, который позволяет динамически загружать сборки (*Assembly*) во время выполнения программы с помощью методов *Load* или *LoadFrom*. После загрузки сборки в программу можно использовать ее типы, методы и т.д. Так как отладчик позволяет вызывать любые методы, существующие в отлаживаемом процессе, можно со стороны отладчика вызвать любой из этих методов и загрузить любую сборку. В том числе, можно скомпилировать новую сборку прямо во время отладки и загрузить ее в процесс. Однако единственный способ выгрузить сборку — это выгрузить весь *AppDomain*.

2.1.3. Динамическая генерация сборок в платформе .Net

В стандартной библиотеке классов .Net (*BCL*) существует пространство имен *System.Reflection.Emit*, которое содержит классы, позволяющие прямо во время выполнения программы генерировать динамические методы и динамические сборки. Таким образом, можно вызывать методы этих классов напрямую из отладчика и сгенерировать динамический метод или сборку прямо в отлаживаемом процессе. Преимуще-

ство динамических сборок перед обычными сборками в том, что они могут быть собраны сборщиком мусора.

2.1.4. Загрузка классов в Java

В Java существует класс *java.lang.ClassLoader*, который обеспечивает загрузку классов. А точнее его наследники, так как сам *ClassLoader* абстрактный. Java — это язык с отложенной загрузкой кода. И каждый раз, когда загружается какой-либо класс, на самом деле это класс загружает какой-то *ClassLoader*. Существует стандартный загрузчик классов, который используется по умолчанию. Однако есть возможность реализовать свой собственный загрузчик классов и использовать его вместо стандартного [14].

2.2. Существующие реализации поддержки лямбда-выражений в отладчиках

2.2.1. Отладчик Microsoft Visual Studio для .Net приложений

В Microsoft Visual Studio, начиная с версии 2015, появилась возможность использовать лямбда-выражения во время отладки. Однако, внутри тела лямбда-выражения невозможен вызов методов, использующих нативный код. Ни исходного кода, ни каких-либо сведений о деталях реализации в открытом доступе нет [9].

2.2.2. Отладчик Intellij Idea

В Intellij Idea существует полноценная поддержка вычислений лямбда-выражений во время отладки. Ни открытого исходного кода, ни информации о деталях реализации в публичном доступе нет.

3. Архитектура решения

Для того, чтобы описать архитектуру решения, необходимо описать как работает модуль, отвечающий за вычисление выражений.

Сначала исходное выражение обрабатывается специальным распознавателем типов (*TypeResolver*), который подменяет имена всех типов, которые были указаны в выражении, на полные, если необходимая информация о соответствующих пространствах имен была указана в исходном коде программы в текущей точке отладки. Например, *List<int>*, преобразуется в *System.Collections.Generic.List<int>*.

Далее выражение преобразуется в абстрактное синтаксическое дерево (*AST*), с помощью парсера C#-кода. Полученное дерево интерпретируется. Для получения необходимой информации из отлаживаемого процесса, интерпретатор оперирует специальным абстрактным адаптером. С помощью адаптера можно получить следующую информацию: значения объектов, название переменных, методов и т.д. Конкретные реализации адаптеров предоставляют CorDebugger и SoftDebugger.

Так как в рамках данной работы использовать лямбда-выражения имеет смысл только как аргументы методов, рассмотрим более подробно, как именно происходит интерпретация вызовов методов.

- Вычисляются значения всех аргументов.
- Вычисляется объект, у которого вызывается метод или имя типа в случае вызова статического метода.
- Вычисляется метод, который надо вызвать.
- Вызывается метод.

Таким образом, происходит вызов методов из отлаживаемого процесса. Однако отладчик не может вызвать из отлаживаемого процесса метод, которого в нем не существует. Соответственно, введенное лямбда-выражение отладчик вызвать не сможет, так как его не существует в отлаживаемом процессе. Как вариант решения проблемы,

можно поддержать интерпретацию лямбда-выражений. Однако это реализовать сильно сложнее, чем интерпретацию обычных выражений. Поэтому было предложено альтернативное решение — добавить в отлаживаемый процесс код, соответствующий лямбда-выражению.

Рассмотрим подробнее предложенный алгоритм получения лямбда-выражения в отлаживаемом процессе.

1. Скомпилировать лямбда-выражение, то есть получить его MSIL-код.
2. Загрузить полученный MSIL-код в отлаживаемый процесс.
3. Получить в отладчике ссылку на объект, соответствующий скомпилированному лямбда-выражению.

Таким образом, во время вычисления значений аргументов метода можно скомпилировать и загрузить лямбда-выражение в отлаживаемый процесс и вызвать его, так как после получения ссылки на объект, который соответствует лямбда-выражению, отладчик может продолжить интерпретировать выражение как и раньше.

Помимо этого, необходимо реализовать захват контекста и передачу его в скомпилированное лямбда-выражение, если оно содержит замыкания. Для этого необходимо скомпилировать лямбда-выражение таким образом, чтобы можно было изменять значения переменных, входящих в замыкание со стороны отладчика.

4. Реализация решения

4.1. Компиляция лямбда-выражения

Для компиляции лямбда-выражений было решено использовать компилятор Roslyn [7], который является кроссплатформенным, с открытым исходным кодом, разрабатываемый компанией Microsoft. Для того чтобы скомпилировать лямбда-выражение, необходимо сгенерировать специальный код, получить сборки, необходимые для компиляции этого кода и передать его в Roslyn. Стоит отметить, что тип лямбда-выражения зависит от контекста и задача вывода этого типа достаточно сложная. Поэтому было решено сгенерировать код таким образом, чтобы Roslyn сам вывел тип лямбда-выражения.

Генерируется код следующего вида.

```
using /* namespaces */

class LambdaClass
{
    public static void LambdaMethod()
    {
        default(/*Type*/).Method(/*lambda exression*/)
    }
}
```

Пространства имен (namespaces) генерируются те же, что указаны в исходном коде в текущей момент отладки. Для этого вычитывается необходимая информация о пространствах имен для текущего метода из специально файла с отладочными символами (PDB-файла), который генерируется компилятором. Однако этого недостаточно, потому что довольно часто может произойти ситуация, когда сборка, содержащая какое-то из этих пространств имен, еще не загружена в отлаживаемый процесс. Соответственно, Roslyn не сможет скомпилировать такой код. Для решения этой проблемы необходимо вычитать из всех загруженных

в процесс сборок все пространства имен и пересечь их с вычитанными из PDB.

Для того чтобы сгенерировать тело метода, необходимо сначала вычислить тип объекта, у которого вызывается метод и типы аргументов, не являющиеся лямбда-выражениями, для того, чтобы Roslyn мог корректно определить правильный метод и тип лямбда-выражения.

Чтобы скомпилировать полученный код, необходимо передать в Roslyn все сборки доступные в процессе.

4.2. Загрузка скомпилированного лямбда-выражения в отлаживаемый процесс

Результатом компиляции является массив байт, который мы можем загрузить в отлаживаемый процесс с помощью метода Load из класса System.Reflection.Assembly. Однако, для того чтобы можно было вызвать этот метод со стороны отладчика, необходимо в отлаживаемом процессе получить этот массив байт. Для этого был реализован метод, который получает в отлаживаемом процессе полную копию массива байт, существующего в процессе отладчика. Сначала создается массив байт в отлаживаемом процессе с помощью вызова метода System.Array.CreateInstance со стороны отладчика, а затем полученные массивы заполняются данными. Рассмотрим более подробно, как происходит заполнения массива данными в конкретных отладчиках.

- В SoftDebugger уже существовал метод, который позволяет заполнить массив данными.

- В CorDebugger вычисляется адрес первого элемента в массиве и данные записываются напрямую в память отлаживаемого процесса по этому адресу.

4.3. Вызов загруженного лямбда-выражения со стороны отладчика

Для того чтобы вызвать скомпилированное лямбда-выражение, необходимо понять, во что его компилирует Roslyn. Для этого можно изучить исходный код Roslyn или же изучить, что получится после декомпиляции кода, содержащего лямбда-выражения.

Рассмотрим, что получится после декомпиляции такого кода.

```
public static void LambdaMethod()
{
    int[] a = {1, 2, 3, 4};
    a.Select(x => -x);
}
```

Сгенерируется вложенный класс, который будет содержать в себе метод, соответствующий лямбда-выражению.

```
[CompilerGenerated]
private sealed class <>c
{
    internal int <LambdaMethod>b__0_0(int x)
    {
        return -x;
    }
}
```

После ряда экспериментов было установлено, что все лямбда-выражения, не имеющие замыканий, скомпилируются в подобные методы внутри класса <>c. Таким образом, зная во что скомпилируется лямбда-выражение, можно получить его из загруженной в процесс сборки с помощью рефлексии. Однако, со стороны отладчика вызывать много методов из отлаживаемого процесса крайне неудобно, поэтому был написан специальный класс-помощник (Helper), который загружается отладчиком в пользовательский процесс, обрабатывает переданную ему

сборку и по запросу отладчика выдает ему делегат, соответствующий запрошенному лямбда-выражению.

4.4. Захват контекста

Для того чтобы вычислить лямбда-выражения, которые содержат внешний контекст, необходимо особым образом сгенерировать код, который передается в Roslyn. В отличии от случая с обычными лямбда-выражениями необходимо для каждой переменной, входящей в замыкание, сгенерировать статическое поле, тип которого будет совпадать с типом самой переменной. Важно отметить, что поля являются именно статическими, потому что в этом случае сгенерированный компилятором код не будет отличаться от кода, который будет сгенерирован для обычных лямбда-выражений.

После компиляции сгенерированного кода, процесс дальнейшей работы не отличается от аналогичного для обычных лямбда-выражений вплоть до вызова лямбда-выражения. Разница заключается в том, что перед вызовом лямбда-выражения необходимо задать значения статическим полям так, чтобы они соответствовали текущим значениям переменных, входящих в захваченный контекст.

5. Апробация

Для тестирования реализованной функциональности были написаны интеграционные тесты, которые проверяют корректность вычислений выражений во время отладки для следующий сценариев: вызов обычных методов, методов расширений доступных из контекста и методов расширений из System.Linq, даже если они не доступны, с лямбда-выражениями в качестве аргументов.

Перед тем как реализованная функциональность попала в публичную версию, она была протестирована QA и остальными разработчиками.

После того как реализованная функциональность была протестирована, она попала в публичную версию.

Заключение

В ходе данной работы были достигнуты следующие результаты.

- Сделан обзор существующих механизмов добавления и изменения кода на лету. Сделан обзор существующих реализаций поддержки лямбда-выражений в отладчиках
- Разработана архитектура решения для вычисления лямбда-выражений во время отладки.
- Решение реализовано и внедрено в коммерческий продукт Rider.

Список литературы

[1] JetBrains. Idea. — URL: https://www.jetbrains.com/idea/.

[2] JetBrains. ReSharper. — URL: https://www.jetbrains.com/resharper/.

[3] JetBrains. Rider. — URL: https://www.jetbrains.com/rider/.

[4] Microsoft. ICorDebug. — URL: https://docs.microsoft.com/en-us/dotnet/framework/unmanaged-api/debugging/icordebug-interface/.

[5] Microsoft. .Net Core. — URL: https://docs.microsoft.com/ru-ru/dotnet/core/.

[6] Microsoft. .Net Full framework. — URL: https://en.wikipedia.org/wiki/.NET_Framework.

[7] Microsoft. Roslyn. — URL: https://github.com/dotnet/roslyn/.

[8] Mono. SoftDebugger. — URL: http://www.mono-project.com/docs/advanced/runtime/docs/soft-debugger/.

[9] Nelson Patrick. Support for debugging lambda expressions with Visual Studio. — URL: https://blogs.msdn.microsoft.com/devops/2014/11/12/support-for-debugging-lambda-expressions-with-visual-studio-2015

[10] Varty Josh. Edit And Continue. — URL: https://joshvarty.com/2016/04/18/edit-and-continue-part-1-introduction/.

[11] Wikipedia. Debugger. — URL: https://en.wikipedia.org/wiki/Debugger.

[12] Wikipedia. IDE. — URL: https://en.wikipedia.org/wiki/Integrated_development_environment.

[13] Xamarin. Mono. — URL: https://https://www.mono-project.com/.

[14] Алиевский Даниил. ClassLoader. — URL: http://samag.ru/archive/article/68.