УДК 004.032 + 004.412

СРАВНИТЕЛЬНЫЙ АНАЛИЗ ПРОИЗВОДИТЕЛЬНОСТИ МЕХАНИЗМОВ МЕТАПРОГРАММИРОВАНИЯ В ЯЗЫКЕ JAVA

А.Ф. Галиуллин[1], И.С. Шахова[2]

Высшая школа информационных технологий и интеллектуальных систем Казанского (Приволжского) федерального университета

[1]gali.azat@gmail.com, [2]is@it.kfu.ru

Аннотация

Использование определенных механизмов метапрограммирования при разработке программных библиотек на языке Java может негативно сказываться на времени сборки и работе конечного программного продукта, в котором они используются. Для того, чтобы нивелировать воздействие различных подходов, необходимо предложить комплексное решение, позволяющее регулировать их использование в зависимости от особенностей контекста, что, в свою очередь, требует проведения предварительного анализа. В данной статье рассмотрены существующие в языке Java механизмы метапрограммирования и представлены результаты сравнения влияния данных подходов на время сборки Android-приложений.

Ключевые слова: *annotation processing, Reflection, обработка аннотаций, рефлексия, кодогенерация, производительность, Android, Java*

ВВЕДЕНИЕ

Развитие сферы разработки программного обеспечения влечет за собой создание все более сложных программ. Для того чтобы сократить количество дублирования кода на этапе разработки приложения и тем самым снизить временные затраты, используются механизмы метапрограммирования. Подобные решения позволяют изменять или дополнять поведение приложения на основе анализа его исходного кода.

На текущий момент в языке программирования Java имеется два инструмента для решения данной задачи – annotation processing и reflection API.

В Java 5 представлены аннотации – специальные конструкции, которые позволяют добавлять метаданные исполняемого кода. У аннотации может быть множество областей применений, например, информация для сборки или развертывания приложения, указание свойств конфигурации, проверки качества. Одной из самых популярных является аннотация @Nullable. Ею помечают функции, которые могут возвращать null, или поля классов, которые могут иметь значение null. Использование @Nullable позволяет указать, что результат помеченной функции необходимо дополнительно проверить перед его использованием. Кроме того, аннотации часто используют как маркеры для генераторов кода. Например, библиотека ButterKnife [1] позволяет пометить аннотацией @BindView поля класса, которые содержат информацию об отображаемых элементах на экране. После этого библиотека сгенерирует код для инициализации помеченных полей.

Annotation processing – это возможность, добавленная в Java 5 и стандартизированная в Java 6 (JSR-269 [2]), которая позволяет генерировать код во время компиляции проекта. Чтобы лучше понимать, как он работает, необходимо рассмотреть, как происходит этап компиляции java-программы [3]. Этот этап производится в 3 стадии:

1. Parse and Enter. Компилятор составляет абстрактное синтаксическое дерево и создает таблицу символов.
2. Annotation Processing. Компилятор запускает зарегистрированные обработчики аннотаций, которые могут быть добавлены из сторонних библиотек.
3. Analyse and Generate. Компилятор анализирует синтаксическое дерево с целью построения класс-файлов.

Механизм обработки аннотаций в языке Java не позволяет модифицировать уже написанный код, а только дает возможность создавать новые файлы с кодом. Обработка аннотаций может происходить в несколько этапов, называемых раундами. В каждом раунде компилируется часть исходного кода, после чего на нем запускаются обработчики аннотаций. Если обработчики аннотаций сгенерировали еще файлы с кодом, то данные файлы будут обработаны в одном из следующих раундов. Обработка аннотаций заканчивается тогда, когда не осталось необработанных исходных файлов. Одним из преимуществ использования annotation processing является то, что весь сгенерированный код проходит этап

анализа (3-я стадия), и поэтому ошибки могут быть выявлены во время компиляции программы, а не во время ее выполнения. Благодаря этому улучшается качество выходного продукта, так как вероятность пользователя столкнуться с ошибками снижается. В связи с тем, что этап верификации проходит после обработки аннотаций, к сгенерированному коду можно напрямую обращаться в коде приложения. Из-за встраивания в процесс компиляции у инструмента annotation processing имеется существенный недостаток – увеличение времени сборки проекта. Данный минус сильно заметен при разработке средних и больших программных решений. Этот недостаток замедляет разработку таких систем и подталкивает разработчиков библиотек к использованию альтернативного способа динамического добавления логики в приложения.

Java Reflection API [4] – возможность из языка Java получать информацию об устройстве программы, а также взаимодействовать с ее элементами (классы, методы, поля и т. д.) во время исполнения. Например, при помощи рефлексии можно получить информацию об имеющихся конструкторах класса, его методах или же создать экземпляр класса, имя которого неизвестно во время компиляции программы. Данный инструмент не требует специальных дополнительных этапов во время компиляции проектов, поэтому и не влияет на время сборки приложений. Но, в свою очередь, модули программы, использующие рефлексию, никак не проверяются компилятором, поэтому ошибки будут обнаружены во время выполнения программы. Такие ошибки, в отличие от возникающих при компиляции, могут быть пропущены разработчиком и выявлены только на этапе тестирования или уже у конечных пользователей.

ИНСТРУМЕНТЫ И КРИТЕРИИ СРАВНЕНИЯ

Для сравнения инструментов была выбрана библиотека Dagger 2 [5] по следующим причинам:

1. Dagger 2 занимает вторую позицию по популярности среди Java Utilities библиотек [6].
2. Dagger 2 во время работы производит большое количество операций (строит граф зависимостей и создает фабрики для них), поэтому влияние изменения инструмента метапрограммирования, используемого библиотекой, будет заметнее.

3. Библиотека может генерировать код следующими способами:

- на этапе сборки проекта при помощи annotation processing;
- во время работы программы при помощи Java Reflection API (реализовано при помощи отдельной версии библиотеки dagger-reflect [7]);
- совмещая предыдущие способы – интерфейс для взаимодействия кода приложений с генерируемым кодом создается на этапе компиляции при помощи annotation processing, а реализация функционала интерфейса происходит во время работы приложения (включено в версию библиотеки dagger-reflect).

Dagger 2 – полностью статический фреймворк для внедрения зависимостей в Java-приложениях. Разработчики используют данную библиотеку, чтобы реализовать процесс Dependency Injection [8].

В качестве системы сборки был выбран Gradle [9], так как он имеет встроенный функционал для исследования и измерения времени процесса сборки проекта.

В качестве критериев были использованы следующие характеристики:

1. затраченное время на annotation processing;
2. полное затраченное время на сборку проекта.

Второй параметр важен, т. к. при использовании annotation processing добавляется время не только на сам механизм, но и на повторную компиляцию, а также выделяется дополнительная оперативная память, которая значительно влияет в случае использования swap.

РЕЗУЛЬТАТЫ СРАВНЕНИЯ

Эксперименты были проведены на 2 проектах, имеющих разный размер.

Тестовым Android-проектом небольшого размера был выбран проект GithubBrowserSample из репозитория Google [10] с примерами использования архитектурных компонентов. Данный проект был выбран, т. к. он используется в качестве примера того, как могут разрабатываться другие проекты с применением Dagger 2.

Эксперименты проводились на следующих компьютерах:

1. Lenovo Y510P (i7-4770HQ, 8GB RAM DDR3, 256 GB SSD), Windows 10 x64;

2. Apple Macbook Pro 15 2018 (i7-8750H, 16GB RAM DDR4, 256 GB SSD), Mac OS Mojave.

Выбор названных машин обоснован тем, что полученные результаты будут более полно отображать поведение методов кодогенерации с учетом возможных различий в операционных системах и аппаратным обеспечением. Кроме того, при сборке проектов на первом устройстве будет активно использоваться механизм swap операционной системы Windows, который часто применяется при сборке больших проектов даже на мощных компьютерах.

В таблицах 1–3 приведены результаты экспериментов при сборке проектов с нуля без использования закэшированных результатов предыдущих сборок.

Таблица 1. Результаты экспериментов на Lenovo Y510P в однопоточном режиме

	Время annotation processing	Время полной сборки проекта	Уменьшение времени сборки в сравнении с annotation processing
При использовании только annotation processing	42s	173s	0%
При использовании только reflection	8s	70s	60%
При совмещении reflection и annotation processing	35s	150s	13%

Таблица 2. Результаты экспериментов на Macbook Pro в однопоточном режиме

	Время annotation processing	Время полной сборки проекта	Уменьшение времени сборки в сравнении с annotation processing

При использовании только annotation processing	11s	36s	0%
При использовании только reflection	9s	34s	6%
При совмещении reflection и annotation processing	10s	35s	3%

Таблица 3. Результаты экспериментов на Macbook Pro с 6 потоками

	Время annotation processing (суммарно на всех потоках)	Время полной сборки проекта (суммарно на всех потоках)	Уменьшение времени сборки в сравнении с annotation processing
При использовании только annotation processing	11s	36s	0%
При использовании только reflection	10s	33s	8%
При совмещении reflection и annotation processing	10s	35s	6%

На Lenovo Y510P эксперименты проводились только в однопоточном режиме, так как узким местом было количество оперативной памяти. На Macbook Pro использовались 6 потоков.

Также необходимо принять во внимание, что во время проведения экспериментов на Lenovo Y510P активно использовался механизм swap, который значительно повлиял на результаты, а именно, увеличил время сборки.

Кроме Dagger 2, в выбранном для экспериментов проекте annotation processing используется еще в библиотеке Room [11] и androidx.lifecycle [12]. Поэтому, даже в случае использования только reflection в Dagger, во время сборки проекта расходуется время на этап обработки аннотаций другими библиотеками.

Для полной картины были проведены эксперименты на проекте, приближенном к проектам больших организаций. Данный проект содержит код не только на языке Java, но и на Kotlin, который компилируется в тот же byte-код, что и программы на языке Java. Kotlin позволяет запускать обработчики аннотаций, написанные для языка Java, что и используется в данном тестовом проекте. В проекте 131 модуль, примерно 400 тысяч строк кода и активно используется Dagger 2. Эксперименты были проведены только на Apple Macbook Pro 15 2018 (характеристики описаны ранее), так как на первом компьютере сборка данного проекта занимает очень большое количество времени, и на практике подобные проекты собирают на компьютерах, обладающих большой мощностью.

Таблица 4. Результаты экспериментов на Macbook Pro в однопоточном режиме

	Время annotation processing	Время полной сборки проекта	Уменьшение времени сборки в сравнении с annotation processing
При использовании только annotation processing	204s	563s	0%
При использовании только reflection	0s	354s	37%
При совмещении reflection и annotation processing	168s	515s	9%

Таблица 5. Результаты экспериментов на Macbook Pro с 6 потоками

	Время annotation processing	Время полной сборки проекта	Уменьшение времени сборки в сравнении с annotation processing
При использовании только annotation processing	400s	1602s	0%
При использовании только reflection	0s	1218s	24%
При совмещении reflection и annotation processing	366s	1550s	3%

В данном проекте кроме dagger 2 не подключены другие библиотеки, которые применяют annotation processing. Именно поэтому в случаях, когда используется версия dagger, основанная только reflection, время сборки уменьшается вплоть до 37%. Это происходит благодаря тому, что процесс обработки аннотаций не запускается во время сборки. В первом же проекте оставались библиотеки, использующие annotation processing, из-за чего компилятор не мог пропустить этап обработки аннотаций.

ЗАКЛЮЧЕНИЕ

Полученные результаты экспериментов наглядно демонстрируют уменьшение времени сборки проектов при переводе библиотеки на использование Java Reflection API. При этом наиболее заметно ускорение сборки на слабых компьютерах, так как, кроме нагрузки на процессор, annotation processing активно использует запись на жесткий диск, из-за чего время может заметно увеличиться на компьютерах с медленной скоростью записи в хранилище. На большом проекте использование реализации, основанной на рефлексии, может уменьшить время

сборки на 37% (или 2 минуты 30 секунд). Для больших компаний это является существенным преимуществом, так как увеличивает производительность разработчиков. Однако у рефлексии имеется существенный недостаток – компилятор не может проверить ошибки. Данные ошибки могут быть пропущены разработчиком и найдены только на этапе тестирования или у конечного пользователя. Исправление подобных ошибок влечет больше временных и финансовых затрат, чем исправление на этапе разработки. Именно поэтому разработчики библиотек изначально выбирали инструмент annotation processing. Но на больших проектах использование этих библиотек замедляет компиляцию. Чтобы справиться с этим, некоторые библиотеки повторно реализуются при помощи Java Reflection API (например, butterknife-reflect [13] или dagger-reflect [7]). Фактически разработчики дублируют логику исходной программной библиотеки, что требует времени и усложняет развитие и поддержку библиотек. Например, dagger-reflect до сих не поддерживает весь функционал исходной библиотеки, из-за чего область применения сильно уменьшается.

СПИСОК ЛИТЕРАТУРЫ

1. GitHub – JakeWharton/butterknife: Bind Android views and callbacks to fields and methods. URL: https://github.com/JakeWharton/butterknife.
2. The Java Community Process(SM) Program – JSRs: Java Specification Requests - detail JSR# 269. URL: https://jcp.org/en/jsr/detail?id=269.
3. Compilation Overview. URL: http://openjdk.java.net/groups/compiler/doc/compilation-overview/index.html.
4. Trail: The Reflection API (The Java™ Tutorials). URL: https://docs.oracle.com/javase/tutorial/reflect/index.html
5. GitHub – google/dagger: A fast dependency injector for Android and Java. URL: https://github.com/google/dagger.
6. Dagger – Android SDK statistics | AppBrain. URL: https://www.appbrain.com/stats/libraries/details/dagger/dagger
7. GitHub – JakeWharton/dagger-reflect: A reflection-based implementation of the Dagger dependency injection library for fast IDE builds. URL: https://github.com/JakeWharton/dagger-reflect.

8. Dependency injection. URL: https://habr.com/ru/post/350068.
9. Gradle Build Tool. URL: https://gradle.org.
10. GitHub – android/architecture-components-samples: Samples for Android Architecture Components. URL: https://github.com/android/ architecture-components-samples.
11. Room Persistence Library. URL: https://developer.android.com/topic/ libraries/architecture/room.
12. Lifecycle | Android Developers. URL: https://developer.android.com/ jetpack/androidx/releases/lifecycle.
13. GitHub – butterknife/butterknife-reflect at master JakeWharton/butterknife. URL: https://github.com/JakeWharton/butterknife/ tree/master/ butterknife-reflect.

COMPARATIVE ANALYSIS OF THE PERFORMANCE OF METAPROGRAMMING MECHANISMS IN THE JAVA LANGUAGE

Azat Galiullin[1], Irina Shakhova[2]

Higher School of Information Technology and Intelligent Systems, Kazan Federal University

[1]gali.azat@gmail.com, [2]is@it.kfu.ru

Abstract

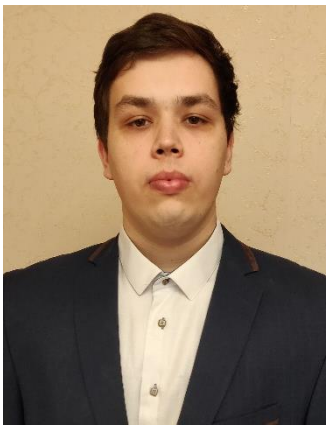
The use of different metaprogramming mechanisms for program libraries development in Java may have some negative effect on build time and end products. The article is aimed to describe metaprogramming mechanisms in the Java language and present the results of experiments that compare the impact of approaches on the build time of small and large projects.

Keywords: *annotation processing, reflection, code generation, performance, Android, Java.*

REFERENCES

1. GitHub – JakeWharton/butterknife: Bind Android views and callbacks to fields and methods. URL: https://github.com/JakeWharton/butterknife.
2. The Java Community Process(SM) Program – JSRs: Java Specification Requests - detail JSR# 269. URL: https://jcp.org/en/jsr/detail?id=269.
3. Compilation Overview. URL: http://openjdk.java.net/groups/compiler/doc/ compilation-overview/index.html.
4. Trail: The Reflection API (The Java™ Tutorials). URL: https://docs.oracle.com/javase/tutorial/reflect/index.html
5. GitHub – google/dagger: A fast dependency injector for Android and Java. URL: https://github.com/google/dagger.
6. Dagger – Android SDK statistics | AppBrain. URL: https://www.appbrain.com/stats/libraries/details/dagger/dagger
7. GitHub – JakeWharton/dagger-reflect: A reflection-based implementation of the Dagger dependency injection library for fast IDE builds. URL: https://github.com/JakeWharton/dagger-reflect.
8. Dependency injection. URL: https://habr.com/ru/post/350068.
9. Gradle Build Tool. URL: https://gradle.org.
10. GitHub – android/architecture-components-samples: Samples for Android Architecture Components. URL: https://github.com/android/ architecture-components-samples.
11. Room Persistence Library. URL: https://developer.android.com/topic/ libraries/architecture/room.
12. Lifecycle | Android Developers. URL: https://developer.android.com/ jetpack/androidx/releases/lifecycle.
13. GitHub – butterknife/butterknife-reflect at master JakeWharton/butterknife. URL: https://github.com/JakeWharton/butterknife/ tree/master/ butterknife-reflect.

СВЕДЕНИЯ ОБ АВТОРАХ



ГАЛИУЛЛИН Азат Фердинандович – магистрант высшей школы информационных технологий и информационных систем КФУ, г. Казань.

Azat GALIULLIN – postgraduate student of the Higher School of Information Technologies and Intelligent Systems, Kazan (Volga region) Federal University, Kazan.

e-mail: gali.azat@gmail.com

ШАХОВА Ирина Сергеевна – старший преподаватель кафедры программной инженерии Высшей школы информационных технологий и интеллектуальных систем Казанского федерального университета. Сфера научных интересов – цифровые образовательные системы, индивидуализация образования, мобильное обучение.

Irina Sergeevna SHAKHOVA – senior teacher of the Higher School of Information Technologies and Intelligent Systems at Kazan Federal University. Research interests include digital educational systems, individualization of education, mobile learning.

email: is@it.kfu.ru

Материал поступил в редакцию 7 апреля 2020 года