JEM: анализ исключений в JVM языках

Выполнил: Букреев Е.А.
Руководители: Ахин М.Х., Беляев М.А.

7 августа 2020 г.

Аннотация

Основная идея проекта JEM: сбор информации о том, какие явные исключения могут порождаться вызовами библиотеки, переданной для анализа в виде jar-архива.

1 Байткод для анализа программ

Байткод JVM содержит в себе всю информацию, необходимую для анализа исключений. В работе анализируются инструкции athrow и вызовы методов (invokestatic, invokevirtual и т.д.).

Стоит упомянуть, что прочие инструкции также могут порождать исключения, однако они не анализируются, т. к. при выполнении работы было принято решение сделать упор на точность, а не на полноту результата.

Для манипуляций с байткодом использовалась библиотека javassist, предоставляющая как низкоуровневые интерфейсы доступа к байткоду, так и довольно высокоуровневые (построение CFG, dominator tree и т.д.).

2 Анализ методов

Для анализа метода на предмет возможных исключений, необходимо построить CFG этого метода, и затем, проходясь по блокам, выбрать те исключения (которые могут возникнуть после вызова метода или быть выброшены инструкцией athrow), которые не ловятся затем инструкциями catch, или не перекрываются возможными исключениями в блоке finally.

Существует ряд особенностей обработки вызовов методов: это рекурсия и полиморфные методы, о них будет сказано далее (см. п.3 и п.4).

Алгоритм программы представлен ниже.

Algorithm 1 Get possible exceptions from method

```
1: polymprphMethods ← getExceptionsFromAllPolymorphMethods
2: previousMethods ← Map < Method, Exceptions >
3: function getPossibleExceptions(method)
4:     if previousMethods.notContainKey(method) then
5:         previousMethods[method] ← emptySet()
6:     end if
7:     exceptions ← Set
8:     for all block ∈ method.cfg.blocks do
9:         if block.isReachable then
10:            for all thrownException ∈ block.athrowInstructions do
11:                if thrownException.isNotCaught then
12:                    exceptions ← exceptions + thrownException
13:                end if
14:            end for
15:            for all method ∈ block.invokedMethods do
16:                methodExceptions
17:                if polymprphMethods.containKey(method) then
18:                    methodExceptions ← polymprphMethods[method]
19:                else if previousMethods.containKey(method) then
20:                    methodExceptions ← previousMethods[method]
21:                else
22:                    methodExceptions ← method.possibleExceptions
23:                end if
24:                for all exception ∈ methodExceptions do
25:                    if exception.isNotCaught then
26:                        exceptions ← exceptions + exception
27:                    end if
28:                end for
29:            end for
30:        end if
31:    end for
32:    while exceptions ≠ previousMethods[method] do
33:        previousMethods[method] ← exceptions
34:        exceptions ← getPossibleExceptions(method)
35:    end while
36:    return exceptions
37: end function
```

3 Рекурсия в методах

Не редко возникают ситуации, в которых либо метод рекурсивен, либо метод, который вызывается в теле текущего, внутри себя сам вызывает текущий метод.

Для решения этого вопроса в проекте используется следующий алгоритм: если метод f() в своём теле вызывает метод g(), а g() вызывает f(), то анализируя f(), и переходя к анализу g(), мы делаем предположение, что f() не выбрасывает исключений, и заканчиваем анализ g(), а затем, если предположение не сошлось с тем, что выбрасывает f(), мы заново анализируем g(), но уже с теми исключениями, которые нам показал анализ, выбрасывает f(), и так до тех пор, пока предположение не сойдётся с результатом анализа f().

Рассмотрим как этот алгоритм реализуется в **Algorithm 1**:

- В строке 2 объявляется ассоциативный массив, ключами которого являются методы, а значениями – предположения выбрасываемых исключений соответствующего метода.
- Затем, в строках 4-5, объявляется пустое предположение для метода, который не анализировался ранее.
- Впоследствии, в строках 19-20, если, метод был ранее принят для анализа, и, следовательно, существует соответствующая запись в ассоциативном массиве, за выбрасываемые этим методом исключения мы принимаем предположение, хранящееся в ассоциативном массиве.
- В конце алгоритма, в строках 32-35, пока найденные нами в результате анализа исключения не совпадут с предположением, мы меняем предположение на найденные исключения, и заново анализируем метод, но уже с другим предположением.

4 Полиморфные методы

Существуют полиморфные методы, выбор реализации которых определяется во время выполнения программы. Вызов таких методов (через инструкции invokevirtual или invokeinterface) может представлять определенные трудности из-за незнания конкретной реализации вызванного метода на этапе анализа байткода.

В проекте предложено следующее решение этого вопроса: в первую очередь, перед началом анализа любого метода, в программе инициализируется ассоциативный массив, ключами которого являются возможные в переданной библиотеке полиморфные методы, а значениями – возможные исключения, которые являются пересечением множеств всех возможных исключений найденных реализаций этого метода.

Соответствующий алгоритм выглядит следующим образом:

Algorithm 2 Get exceptions from polymorphic methods

```
1: function getExceptionsFromAllPolymorphMethods
2:     classesToHeirs ← getHeirsForClassesAndInterfaces
3:     methodsToOverriders ← Map < Method, Methods >
4:     for all (class, subclasses) ∈ classesToHeirs do
5:         for all method ∈ class.methods do
6:             overriders ← getOverridersForMethod(method, subclasses)
7:             methodsToOverriders.put(method, overriders)
8:         end for
9:     end for
10:    methodToExceptions ← Map < Method, Exceptions >
11:    for all (method, overriders) ∈ methodsToOverriders do
12:        exceptionsFromOverriders ← Set < Exceptions >
13:        for all overrider ∈ overriders do
14:            exceptionsFromOverriders.add(getPossibleExceptions(overrider))
15:        end for
16:        possibleExceptions ← exceptionsFromOverriders.first
17:        for all exceptions ∈ exceptionsFromOverriders do
18:            possibleExceptions ← possibleExceptions.intersect(exceptions)
19:        end for
20:        methodToExceptions.put(method, possibleExceptions)
21:    end for
22:    return methodToExceptions
23: end function
```

5 Особенности языка Kotlin

После просмотра байткода программ на языке Kotlin, выяснилось, что, перед тем как бросить исключение инструкцией athrow, оно приводится к типу java.lang.Throwable. Сделано это из-за того, что все исключения в Kotlin являются непроверяемыми. Обойти это приведение удалось следующим способом: при нахождении инструкции athrow смотрится не верхнее значение в стеке, а предпоследнее.