Санкт-Петербургский государственный университет

Кафедра системного программирования

Группа 21.Б10-мм

Методы работы с утечками памяти в JVM

Слугин Александр Николаевич

Отчёт по учебной практике
в форме «Теоретическое исследование»

Научный руководитель:
старший преподаватель, С. Ю. Сартасов

Санкт-Петербург
2024

Оглавление

Введение

Главной характеристикой программного обеспечения является высокая надёжность и возможность непрерывного и стабильного функционирования. Однако во время разработки сложно предусмотреть все возможные сценарии сбоев и ошибок, даже при высоком уровне тестового покрытия кода. Поэтому программное обеспечение, в котором могут происходить ошибки, может попадать в производственную среду, что является критичным моментом, так как любые сбои в работе программы могут приводить к простоям в производственных процессах, вызывая, например, дополнительные финансовые затраты. В связи с этим необходимо разрабатывать средства и методы, направленные на борьбу со сбоями системы во время исполнения.

Одним из перспективных направлений в области обеспечения стабильной и бесперебойной работы являются самовосстанавливающиеся приложения — приложения, которые включают в себя методы, позволяющие автоматически обнаруживать, диагностировать и исправлять ошибки в реальном времени без какого-либо вмешательства человека. Такой подход позволяет минимизировать потери из-за простоя системы, увеличить надёжность и доступность программного обеспечения, а также уменьшить затраты на обслуживание [5]. Несмотря на данные преимущества, стоит отметить, что инструменты самовосстановления могут потреблять системные ресурсы, тем самым влияя на производительность всего приложения. Кроме того, при использовании методов обнаружения ошибок возникает вероятность ложных срабатываний или, наоборот, невозможности обнаружения ошибки, несмотря на её фактическое наличие.

Программы, написанные на языках программирования Java, Kotlin, Scala и некоторых других, исполняются при помощи виртуальной машины Java (JVM — Java Virtual Machine). Одной из самых распространённых ошибок, возникающих при выполнении программ на виртуальной машине Java, является ошибка OutOfMemoryError [10]. Основными причинами возникновения данной ошибки являются [13]:

- недостаток памяти для первоначальной загрузки классов среды выполнения Java;
- недостаточное пространство в куче для размещения нового объекта.

Первая проблема может быть решена увеличением памяти, выделяемой для JVM. Во втором случае ошибка OutOfMemoryError свидетельствует о том, что во время выполнения программы происходят утечки памяти. В этом случае сборщик мусора не может освободить достаточное количество памяти для создания нового объекта, и при этом куча не может быть расширена. Утечка памяти отдельного объекта может не приводить к возникновению ошибки, но такие утечки могут накапливаться со временем, что может привести к длительной работе программы без возникновения OutOfMemoryError. Это делает утечку памяти проблемой, трудной для выявления на стадии разработки и тестирования.

Для решения проблемы утечки памяти разрабатывалось множество инструментов, которые позволяют обнаруживать и анализировать утечки памяти, также были предложены способы борьбы с утечками памяти во время исполнения. Однако данные инструменты не были внедрены в версии JVM, которые используются в индустрии, так как все они требуют модификации различных частей JVM и при этом предоставляют только определённую вероятность определения утечки памяти, а также влекут за собой дополнительное потребление ресурсов. Поэтому самым популярным методом нахождения утечек памяти, применяемым в индустрии, является анализ снимка состояния кучи. Но данный способ не может предоставить всю необходимую информация о причине возникновения утечки памяти для её эффективного устранения.

В данной работе будут исследованы методы для обнаружения, диагностирования и предотвращения утечек памяти при выполнении программ на виртуальной машине Java, применение которых может повысить надёжность программного обеспечения, благодаря уменьшению вероятности возникновения ошибки OutOfMemoryError.

1. Постановка задачи

Целью работы является исследование целесообразности применения способов обнаружения и предотвращения утечек памяти в программах, исполняющихся при помощи виртуальной машины Java.

Для выполнения цели были поставлены следующие задачи:

1. Провести обзор методов, которые позволяют обнаруживать и предотвращать утечку памяти.
2. Определить методологию и сравнить рассмотренные методы.
3. Составить требования к методу для предотвращения утечек памяти.
4. Реализовать метод и проанализировать влияние на работу программы.

2. Обзор инструментов для определения утечек памяти

Большинство существующих инструментов для обнаружения утечек памяти используют следующие методы [11]:

1. Анализ состояния программы.
2. Визуализация состояния кучи.
3. Статический анализ исходного кода.
4. Обнаружение неиспользуемых объектов в куче.
5. Анализ роста используемой памяти.

Первые три способа не предназначены для борьбы с утечками памяти во время выполнения программы, а только лишь помогают проанализировать поведение программы во время выполнения. Также при использовании данных методов отсутствует доступ к информации времени исполнения, такой как трассировка размещения объектов в памяти. Это существенно осложняет определение причины возникновения утечки памяти. Преимуществом использования данных способов является отсутствие влияния на работу программы во время выполнения.

Инструменты, применяющие методы обнаружения неиспользуемых объектов в куче и анализ роста используемой памяти, взаимодействуют с запущенной виртуальной машиной Java, что позволяет им получить доступ к информации о распределении объектов в куче и их активности. Благодаря этому данные инструменты обладают более полной информацией о состоянии объектов в памяти во время выполнения, на основе которых можно предпринимать действия для предотвращения утечек памяти, и, следовательно, предотвращения ошибки OutOfMemoryError. Так как данные методы работают во время выполнения программы, они потребляют дополнительные ресурсы, что может влиять на производительность приложения.

2.1. Обзор инструментов, основанных на обнаружении неиспользуемых объектов в куче

Неиспользумые объекты — это объекты, которые не использовались долгое время, но при этом остаются в куче. Существование таких объектов может означать утечку памяти. На основе этого предположения были разработаны несколько инструментов для обнаружения утечек памяти, которые будут рассмотрены в этой главе. Многие из них были реализованы в исследовательской версии виртуальной машины Java — Jikes RVM [7].

2.1.1. Bell

Bell — Bit Encoding Leak Location [1] — вероятностный подход, который позволяет определить место выделения (allocation site) каждого объекта. Данный метод для каждого объекта кодирует в один бит информацию о месте его выделения в исходном коде. Несмотря на то, что при этом теряется большая часть информации, Bell может восстановить с высокой точностью место выделения объекта, имея достаточное количество объектов с закодированной информацией и известный ограниченный набор возможных мест выделения памяти для новых объектов. Закодированный бит хранится в одном из четырёх свободных бит заголовка объекта, поэтому данный метод не требует дополнительной памяти. Bell был реализован в Jikes RVM. Данный подход используется в инструментах для обнаружения утечек памяти, которые хранят место выделения для каждого объекта и другие метаданные об объекте.

2.1.2. Sleigh

Sleigh [1] — инструмент для обнаружения утечек памяти, который определяет неиспользуемые объекты и использует Bell для получения информации о месте выделения и последнего использования объекта. Данный метод использует четыре бита в заголовке объекта для хранения необходимой информации. Два из четырёх битов используются для

хранения информации о месте выделения объекта и месте последнего использования в исходном коде. Данная информация обрабатывается при помощи Bell. Оставшиеся два бита используются для отслеживания времени последнего использования объекта, для этого используется логарифмический счетчик, который увеличивается после выполнения определенного количества операций сборки мусора. Периодически вызывается функция определения утечки памяти, которая анализирует объекты с большим значением счетчика.

В реализации Jikes RVM в заголовке объекта имеется четыре свободных бита, поэтому Sleigh не требует дополнительной памяти. В других версиях JVM, например IBM J9 JVM [6], имеется только три свободных бита, но Sleigh может хранить необходимую информацию за пределами кучи, в таком случае дополнительное потребление памяти составляет 6,25%. При использовании Sleigh время выполнения программы увеличивается на 29%, но это значение может быть уменьшено до 11%, если использовать адаптивный статический профилировщик [4].

2.1.3. LeakSurvivor

LeakSurvivor [12] — инструмент для предотвращения утечек памяти во время выполнения программы для языков со сборщиком мусора. Идея работы данного инструмента заключается в периодическом перемещении объектов, которые потенциально являются утечкой памяти, из виртуальной памяти на диск.

Для обнаружения объектов для перемещения на диск используется Sleigh. Так как Sleigh обнаруживает неиспользуемые объекты, то он может обнаружить объекты, которые не являются утечкой памяти, но при этом просто долго не используются. Для таких ложноположительных срабатываний Sleigh в LeakSurvivor реализован механизм для возвращения объектов в кучу, благодаря которому программа может продолжить выполняться даже при обращении к объекту, который был перенесён на диск.

После копирования объекта с виртуальной памяти на диск все ссылки, указывающие на объект, заменяются ссылками, которые указывают

на резервный адрес в памяти, обращение к которому означает, что объект был ошибочно помечен неиспользуемым и перенесён на диск. Поэтому при обращении к резервному участку памяти срабатывает механизм возвращения объекта в кучу. Также после копирования объекта все исходящие ссылки, которые содержит объект, сохраняются в специальную таблицу *Swap-Out Table (SOT)*, которая проверяется сборщиком мусора при каждой итерации сборки мусора. Это позволяет избежать некорректного удаления объектов, ссылки на которые существовали только в перенесённом объекте.

LeakSurvivor был реализован в Jikes RVM и протестирован на трёх приложениях, которые содержали утечки памяти. В одном приложении удалось увеличить производительность на 46% и время работы в два раза до возникновения OutOfMemoryError, а в двух других приложениях удалось полностью избавиться от утечек памяти. Несмотря на то, что работа LeakSurvivor потребляет системные ресурсы, удалось достичь увеличения производительности приложений на 27-46%, так как воздействие утечек памяти на производительность оказалось более значительным по сравнению с воздействием LeakSurvivor. Дополнительное потребление виртуальной памяти составило 8,5-9 Мб, 8 Мб из которых занимает буффер для временного хранения объектов перед перемещением на диск, оставшаяся часть используется для хранения таблицы SOT.

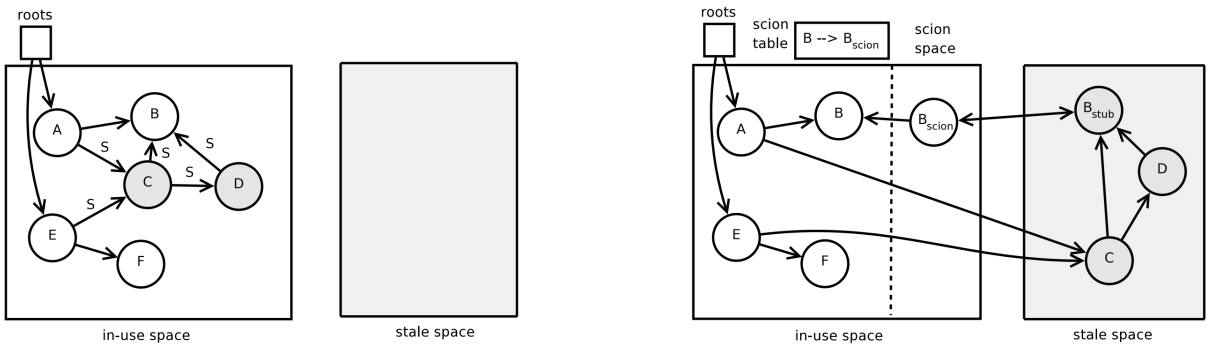
Тестирование на приложениях, которые не содержат утечек памяти, показало, что время работы программы увеличивается в среднем на 23,7%, а дополнительное потребление памяти отсутствует. Это обусловлено отсутствием объектов, которые надо переносить на диск, а также тем, что выделение памяти для хранения перемещённых объектов и таблицы SOT происходит только при необходимости. Стоить отметить, что значительную часть дополнительного времени работы занимает исполнение Sleigh, в то время как дополнительное время работы LeakSurvivor без использования Sleigh составляет всего 2,5%.

2.1.4. Melt

Melt [2] — аналогичный LeakSurvivor, инструмент для борьбы с утечками памяти во время выполнения программы. Работа данного метода так же заключается в определении неиспользуемых объектов и перемещении их на диск, тем самым освобождая место в виртуальной памяти. При попытке обращения к перемещённому объекту Melt возвращает его в виртуальную память.

Для механизма обнаружения неиспользуемых объектов требуется модификация сборщика мусора и динамического компилятора. При каждой сборке мусора объекты помечаются устаревшими (рис. 1a), если с предыдущей итерации сборки мусора приложение не загружало ссылку на объект. Данные о неиспользуемости объекта хранятся в одном из свободных бит заголовка объекта. Для большей эффективности ссылки на объекты также помечаются устаревшими с помощью незначимого бита в указателе. При обращении к объектам отметка о неиспользуемости снимается, для этого компилятор добавляет в приложении соответствующий инстументарий.

При работе сборщика мусора выполняется перемещение неиспользуемых объектов на диск. Ссылки, исходящие из перемещенных объектов, на активные объекты представляют собой проблему, так как при изменении адреса размещения активного объекта необходимо обновлять данные в перемещённых объектах, что может быть затратно, если на объект ссылается большое количество перемещённых объектов. Для решения этой проблемы используются пары объектов заглушка-потомок (stub-scion pair) (рис. 1b). Для всех активных объектов, на которые ссылается хоть один перемещённый объект, создается единственная пара объектов заглушка-потомок. Объект-заглушка создается на диске, и на него ссылаются перемещенные объекты. Объект-потомок размещается в виртуальной памяти и ссылается на активный объект и объект-потомок. Благодаря данному механизму всегда можно восстановить перемещенный на диск объект, гарантируя, что объект содержит актуальные ссылки.



(a) Устаревшие объекты и ссылки (b) Пары предок-потомок

Рис. 1: Структура памяти, используемая в Melt [2]

Для предотвращения дополнительного потребления ресурсов Melt находится в неактивном состоянии до тех пор, пока использование виртуальной памяти не превышает 50%. После того как свободной виртуальной памяти осталось менее 50%, Melt начинает определять и помечать неиспользуемые объекты, а после использования более 80% памяти — перемещать объекты на диск.

Данный подход был реализован в Jikes RVM и протестирован на десяти приложениях, для пяти из которых удалось полностью избавиться от утечек памяти. Исполнение данных приложений завершалось только из-за полного использования пространства диска. В трёх других приложениях удалось достичь увеличения времени работы в два раза. В оставшихся двух приложениях получилось незначительно увеличить время работы до возникновения OutOfMemoryError, но при этом из-за работы Melt была значительно уменьшена производительность приложений. Это было вызвано наличием большого количество объектов, доступ к которым производился редко, в связи с чем Melt переносил их на диск, но через какое-то время было необходимо восстанавливать данные объекты. Среднее увеличение времени работы, вызванное работой Melt, составило 6%.

В отличие от LeakSurvivor, Melt гарантирует, что дополнительное потребление виртуальной памяти и увеличение времени выполнения будут пропорциональными размеру используемой виртуальной памяти. Это вызвано тем, что LeakSurvivor сохраняет все ссылки, исходящие

из перемещенных на диск объектов, на активные объекты в таблице SOT, но при перемещении активного объекта на диск ссылка из таблицы не удаляется, что приводит к дополнительным затратам виртуальной памяти и времени при сборке мусора. В Melt при перемещении объекта на диск соответсвующий ему объект-потомок удаляется из кучи.

2.2. Обзор инструментов, основанных на анализе роста используемой памяти

В этой главе будут рассмотрены инструменты, основанные на анализе роста различных параметров во время работы приложения. В качестве параметров могут выступать совокупный размер объектов определённого типа, количество объектов или размер используемой виртуальной памяти.

2.2.1. Panacea

Panacea [3] — фреймворк для разработки программного обеспечения с возможностью самовосстановления. Подход работы данного инструмента основан на добавлении во время разработки аннотаций к классам, доступ к которым должен иметь фреймворк для их мониторинга и управления во время выполнения программы.

Одной из возможностей данного фреймворка является предотвращение утечек памяти. *ObjectDumpHealer* — компонент, который отслеживает использование памяти и может перемещать объекты из виртуальной памяти на диск. Для того чтобы ObjectDumpHealer имел возможность управлять объектами в процессе выполнения, их классы необходимо пометить аннотацией *@Dumpable*. С помощью данной аннотации можно указать через какое время с момента последнего использования объект должен быть перемещён на диск. Для каждого объекта, помеченного аннотацией, создаётся прокси-объект, который предоставляет такой же интерфейс, как у объекта, и содержит единственную ссылку на объект. Все существующие ссылки на объект заменяются ссылками на соответствующий ему прокси-объект. При вызове методов прокси-объекта, вызов

перенаправляется к объекту, если он активен и находится в виртуальной памяти, или ObjectDumpHealer возвращает объект в виртуальную память, и после чего происходит вызов метода. Также прокси-объект необходим для корректного удаления сборщиком мусора объекта при его перемещении на диск.

В каждом прокси-объекте содержится бит, который служит для определения неиспользуемых объектов. Периодически ObjectDumpHealer в каждом прокси-объекте данному биту присваивает 0, а при обращении к объекту биту присваивается 1. При использовании памяти более чем на 80%, происходит поиск прокси-объектов, у которых бит равен 0. Соответствующие им объекты копируются на диск, а ссылка в прокси-объекте на объект зануляется, что позволяет сборщику мусора удалить перемещенный на диск объект.

Тестирование показало, что увеличение времени работы программы в связи с работой Panacea составляет 2,5%. При этом удалось увеличить количество выделенных в виртуальной памяти объектов в четыре раза в наихудшем случае.

Недостатком инструмента является тот факт, что прокси-объекты никогда не удаляются из кучи, тем самым занимая место в виртуальной памяти, что при большом количестве перенесённых на диск объектов может приводить к переполнению памяти. Стоить отметить, что Panacea не определяет автоматически объекты, которые потенциально являются утечкой памяти, а обрабатывает те объекты, которые программист пометил аннотацией. Также данный подход может быть реализован изменением байт-кода перед выполнением программы вместо внесения изменений в исходный код программы.

2.2.2. Cork

Cork [8] — инструмент для выявления утечек памяти во время выполнения программы для языков со сборщиком мусора, основанный на анализе роста используемой памяти. Данный инструмент находит структуры данных (классы) и места выделения объектов, которые приводят к постоянному увеличению используемой памяти, что в конечном итоге

может привести к переполнению памяти и завершению программы.

Для анализа используемой памяти Cork создаёт объект, который представляет собой граф (*class point-from graph — CPFG*). Узлы графа содержат информацию о размере активных объектов определённого класса. Каждый узел отвечает за один класс объектов. Рёбра графа представляют собой ссылки между объектами разных классов и имеют вес пропорциональный размеру активных объектов, на класс которых указывает ребро. При каждой итерации сборки мусора происходит обновление данных в CPFG, после чего определяются узлы графа с классами, размер которых увеличился. Определение таких узлов происходит путём сравнения текущего объекта CPFG с объектами, которые были получены при предыдущих итерациях сборки мусора. Для всех выявленных увеличившихся узлов, Cork обходит граф по ссылкам, размер которых также увеличился, и находит классы структур данных, которые задействованы в росте используемой памяти. Результатом работы Cork является список классов объектов, вызывающих утечки памяти, и список динамических структур данных, которые содержат данные объекты. Также Cork возвращает список мест в исходном байт-коде, в которых происходит выделение найденных объектов. Стоить отметить, что возвращается список всех возможных мест выделения объектов, а не конкретные места выделения вызвавших утечку памяти объектов, так как это повлекло бы дополнительное потребление ресурсов.

Cork был реализован в Jikes RVM и протестирован на 15 приложениях, для всех из которых точно определил наличие или отсутствие утечки памяти. При этом данный инструмент увеличивает время выполнения программы на 2,3%, а использование виртуальной памяти менее чем на 0,5%. Необходимо отметить, что данный инструмент предназначен только для поиска информации о возможных утечках памяти и не предоставляет средств для их исправления во время выполнения программы.

3. Сравнение инструментов

В таблице 1 проведено сравнение вышеперечисленных инструментов для обнаружения и предотвращения утечек памяти.

Только три рассмотренных инструмента (Sleigh, LeakSurvivor, Melt) могут определять объекты, которые потенциально являются утечкой памяти. При этом их реализация требует внесения изменений в сборщик мусора и динамический компилятор. Sleigh и LeakSurvivor значительно увеличиют время выполнения программы (на 29% и 23,7% соответсвенно), но не требуют модификации динамического компилятора. Melt требует модификации как сборщика мусора, так и компилятора, но при этом увеличивает время выполнения программы всего на 6%.

Единственным инструментом из рассмотренных, который не требует изменений ни сборщика мусора, ни компилятора, является Panacea. Данный фреймворк незначительно увеличивает время выполнения программы (на 2,5%), но он не включает в себя механизм обнаружения утёкших объектов, а работает только с классами объектов, помеченными аннотацией во время разработки. Поэтому эффективность работы Panacea зависит от программиста, который сталкивается с нетривиальной задачей выбора классов объектов, которые необходимо пометить специальной аннотацией.

Cork и Sleigh предназначены только для обнаружения объектов, которые являются утечкой памяти. Оба инструмента требуют модификации сборщика мусора. Sleigh увеличивает время выполнения программы на 29%, в то время как Cork на 2,3%. Также Cork более точно определяет утёкшие объекты, так как Sleigh основан на вероятностом подходе Bell, а Cork анализирует граф классов объектов. Преимуществом Sleigh перед Cork является то, что Sleigh определяет точное место выделение объекта в исходном коде, тогда как Cork определяет только класс объекта и возможные места выделения объектов данного класса.

У всех инструментов дополнительное потребление виртуальной памяти является незначительным или вовсе отсутствует. Отсутствие дополнительного расхода памяти не свидетельствует о её нулевом использовании,

а указывает на то, что эти инструменты эффективно уменьшают общий объем используемой памяти, справляясь с утечками объектов.

Метод	**Определяет утёкшие объекты**	**Предотвращает сбой программы**	**Требует модификации сборщика мусора**	**Требует модификации компилятора**	**Требует модификации исходного кода**	**Издержки по времени**	**Издержки по памяти**
Sleigh	Да	Нет	Да	Нет	Нет	29% (м.б. уменьшено до 11%)	0
LeakSurvivor	Да	Да	Да	Нет	Нет	23,7% (2,5% добавляет к Sleigh)	0-9 Мб
Melt	Да	Да	Да	Да	Нет	6%	-
Panacea	Нет	Да	Нет	Нет	Да	2,5%	-
Cork	Да (Только класс объекта)	Нет	Да	Нет	Нет	2,3%	1%

Таблица 1: Сравнение инструментов для борьбы с утечками памяти

4. План реализации

В дальнейшем планируется реализовать метод для предотвращения утечек памяти в OpenJDK[1]. Метод будет выполнять следующие задачи:

1. обнаружение объектов, которые являются утечкой памяти;
2. информирование о возможной утечке;
3. перемещение обнаруженных объектов на диск с возможностью их восстановления;
4. удаление объектов с диска при переполнении диска.

Рассмотренные инструменты, которые позволяют избежать прекращения работы программы, не содержат механизма для информарирования об обнаруженных утечках памяти. Поэтому для более эффективного исправления ошибок в исходном коде в реализации метода планируется добавить возможность информирования об утечках памяти, например с помощью логирования.

При использовании методов, которые перемещают объекты на диск, выполнение программы прекращалось из-за переполнения диска. Для предотвращения этого планируется реализовать механизм удаления с диска объектов, которые раньше всех были перемещены на диск.

[1]OpenJDK [9] — версия среды разработки Java (Java Development Kit) с открытым исходным кодом, которая включает в себя виртуальную машину Java (JVM) и среду выполенения Java (JRE).

Заключение

В результате работы были выполнены следующие задачи:

- Проведен обзор инструментов для обнаружения и предотвращения утечек памяти.
- Выполнено сравнение рассмотренных инструментов.
- Составлены требования к методу для предотвращения утечек памяти.

В дальнейшем планируется реализовать метод для предотвращения утечек памяти и проанализировать влияние на выполнение программы.

Список литературы

[1] Bond McKinley. Bell: Bit-Encoding Online Memory Leak Detection // ACM SIGARCH Computer Architecture News. — 2006. — Vol. 34, no. 5. — P. 61–72.

[2] Bond McKinley. Tolerating Memory Leaks // OOPSLA'08: Proceedings of the 23rd ACM SIGPLAN conference on Object-oriented programming systems languages and applications. — 2008. — P. 109–126.

[3] Breitgand Goldstein. PANACEA — Towards a Self-healing Development Framework // 10th IFIP/IEEE International Symposium on Integrated Network Management. — 2007.

[4] Chilimbi Hauswirth. Low-Overhead Memory Leak Detection Using Adaptive Statistical Profiling // ASPLOS XI: Proceedings of the 11th international conference on Architectural support for programming languages and operating systems. — 2004. — P. 156–164.

[5] Handa Nikhil. Self-Healing Code: Revolutionizing the World of Programming. — 2023. — URL: https://www.linkedin.com/pulse/self-healing-code-revolutionizing-world-programming (дата обращения: 1 октября 2023 г.).

[6] IBM J9 JVM. — URL: https://www.ibm.com/docs/en/order-management-sw/10.0?topic=machines-j9-jvm (дата обращения: 28 ноября 2023 г.).

[7] Jikes RVM. — URL: https://www.jikesrvm.org/ (дата обращения: 28 ноября 2023 г.).

[8] Jump McKinley. Detecting memory leaks in managed languages with Cork // Software – Practice and Experience. — 2009. — Vol. 40, no. 1. — P. 1–22.

[9] OpenJDK. — URL: https://openjdk.org/ (дата обращения: 27 декабря 2023 г.).

[10] Oumous Mouad. The Most Common Java Runtime Errors. — 2023. — URL: https://medium.com/thefreshwrites/the-most-common-java-runtime-errors-b6bc49a7716e#18a6 (дата обращения: 18 ноября 2023 г.).

[11] Sor Srirama. Memory leak detection in Java: Taxonomy and classification of approaches // The Journal of Systems and Software. — 2014. — Vol. 96. — P. 139–151.

[12] Tang Gao Qin. LeakSurvivor: towards safely tolerating memory leaks for garbage-collected languages // USENIX 2008 Annual Technical Conference. — 2008. — P. 307–320. — URL: https://dl.acm.org/doi/10.5555/1404014.1404040 (дата обращения: 17 декабря 2023 г.).

[13] Understand the OutOfMemoryError Exception. — URL: https://docs.oracle.com/javase/8/docs/technotes/guides/troubleshoot/memleaks002.html (дата обращения: 18 ноября 2023 г.).