26 ноября 2021

Java ♥ 1 🔥 1 💧 0 💩 0

☕ Разбираемся, почему в Java утекает память несмотря на сборщик мусора

Filipp Voronov

Telegram: @fivoronov

Сборщик мусора облегчает написание кода и справляется с основными проблемами, но не гарантирует полного отсутствия утечек памяти. Изучите базовые принципы его работы, чтобы понять, какими видами мусора он заниматься не будет.

3 1

Suggestions:

The Best Java course

Что такое утечка памяти?

Утечкой называют присутствие в памяти ненужных программе объектов и данных. На их содержание тратятся ресурсы и в конечном итоге это может привести к ошибке работы виртуальной машины `java.lang.OutOfMemoryError`, которая завершит работу программы.

Как сборщик мусора ищет мусор?

Как же сборщик мусора (англ. Garbage Collector, GC) определяет ненужность объекта? Базовый принцип работы GC – это поиск объектов, на которые программа потеряла ссылки.

Рассмотрим пример вызова метода, внутри которого создадим объект, используем его для подсчёта примитивного значения и без сохранения куда-либо созданного объекта возвращаем посчитанное значение:

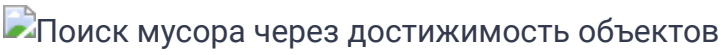
```
public static boolean isGreetingTooHard() {
    String msg = "Hello World!";
    return msg.length() > 10;
}
```

Suggestions:

OOP principles in practice

После завершения вызова метода `isGreetingTooHard` локальная переменная `msg` будет уничтожена и в нашей программе не останется ни одной ссылки на созданный объект строки. Значит, программа физически не сможет использовать объект нигде после вызова метода и GC может спокойно удалить его из памяти, не опасаясь что обращения нему в будущем.

В программах часто объекты путешествуют из одних методов в другие, сохраняются и удаляются из полей, массивов, коллекций, что усложняет определение факта потери на них ссылок. Но для GC это не является неразрешимой проблемой и он способен определить, можно ли до объекта дойти по ссылкам из существующей в этот момент ячейки программы (переменной, параметра,..), и, если нельзя, то этот объект точно является мусором, ведь программа его физически уже никогда не сможет использовать.


Поиск мусора через достижимость объектов

Поиск мусора через достижимость объектов

Когда мусор не найдётся?

Объект может быть достижим из живых локальных переменных / статических полей и других активных ячеек программы, но при этом всё равно быть уже ненужным. Этот объект будет мусором, но GC не будет очищать память от него. Рассмотрим некоторые примеры подобных и других утечек памяти в Java.

Статические поля

В отличие от нестатических полей, которые существуют пока не удалён содержащий их объект, статические поля обычно живут вплоть до завершения программы. Если в статическом поле сохранена ссылка на объект, то этот объект всегда будет считаться достижимым из программы и GC удалять его не будет. Если этот объект больше не нужен, то поле следует об-null-ить самому, чтобы GC смог его очистить.

Незакрытые ресурсы

Под открытые ресурсы (например, соединения или потоки ввода-вывода) выделяется

Например, в следующем примере сканнер закрыт не будет, если из входного потока будет считан 0, на который попытаются разделить:

```
public static double divide() {
    Scanner scanner = new Scanner(...);
    double result = 1;
    while (scanner.hasNextInt()) {
        result /= scanner.nextInt();
    }
    scanner.close();
    return result;
}
```

Чтобы избежать этого рода утечки памяти, используйте `try-with-resources` для автозакрытия ресурсов даже в случае возникновения исключений (до Java 7 используйте для этих целей `finally`):

```
public static double divide() {
    try (Scanner scanner = new Scanner(...)) {
        double result = 1;
        while (scanner.hasNextInt()) {
            result /= scanner.nextInt();
        }
        return result;
    }
}
```

Внутренние классы

Ряд механизмов в Java предполагают наличие неявной ссылки на объект. К числу таких относится и внутренний класс. Рассмотрим пример:

Country.java

```
public class Country {
```

Suggestions:

Good article about classes

How to use java.util.Scanner?

```
    public ForeignMinister newForeignMinister() {
        return new ForeignMinister();
    }

    public class ForeignMinister {
        protected int deals = 0; // количество успешных сделок

        public void acceptInvitation() {
            deals++;
```

Класс описывает объект с информацией о стране – в полях содержатся имя главы государства (поле `headOfState`) и имена всех жителей (большой массив `population`). Внутренний класс описывает объект министра иностранных дел – поле количества успешных договоров и метод приглашения главы государства на саммит.

Теперь представим себе метод, который создаёт (и никуда не сохраняет) объект страны, спрашивает у него объект министра иностранных дел и возвращает только министра из метода:

```
public static ForeignMinister getAnyMinister() {
    Country proglibLand = new Country();
    return proglibLand.newForeignMinister();
}
```

Будет ошибкой заключить, что созданный объект страны будет подчищен GC после завершения метода. Класс министра это внутренний классом, его объекту доступны все поля объекта страны, от которой он создан. Для этого джава будет поддерживать неявную ссылку в объекте министра на объект страны, храня в памяти его вместе со всеми полями, включая огромный массив с именами жителей, который нашей программе логически не нужен, а значит является мусором.

Чтобы избежать подобной утечки памяти, достаточно помнить об особенностях работы внутренних классов. Если критично, можно всегда использовать вместо них статические вложенные классы, у которых такого эффекта нет.

Собственные структуры данных

набор данных с двумя операциями: `push(значение)` вставляет новое значение в конец набора; `pop()` вынимает значение с конца набора.

Рассмотрим упрощённую реализацию без красивой обработки ошибок:

LeakyStack.java

```
public class LeakyStack<T> {
    private T[] data;      // буфер
    private int nextIndex; // свободная ячейка буфера

    public LeakyStack(int size) {
        data = (T[]) new Object[size];
    }

    public void push(T value) {
        data[nextIndex++] = value;
    }

    public T pop() {
        return data[--nextIndex];
    }
```

При создании указывается максимальный размер стека и заводится массив, в котором будут храниться вставляемые элементы. Так как длину массива менять нельзя, сразу делаем массив размером с полный стек, чтобы у были ячейки про запас, а поле `nextIndex` будет указывать на первую свободную ячейку для хранения нового элемента. Схожие идеи используются во многих структурах данных, например, в `ArrayList` и `ArrayDeque`.

Пользоваться стеком можно так:

```
LeakyStack<String> stack = new LeakyStack<>(10);
stack.push("Petya");
stack.push("Katya");
System.out.println(stack.pop()); // Katya
```

удаление этого объекта из памяти через GC. Но этого не произойдёт, тк в нашем массиве ссылка на этот объект продолжит храниться, предотвращая удаление мусора из кучи.

Избавиться от этого поможет об-null-ение ячеек, значения которых больше не нужны:

```
public T pop() {
    T popped = data[--nextIndex];
    data[nextIndex] = null;
    return popped;
}
```

Итог

Наличие встроенных алгоритмов сборки мусора ещё не гарантирует что весь мусор будет вычищаться, а занятая им память освобождаться для переиспользования. Понимание принципов работы GC помогает избежать накопления такого мусора в программе, который не будет убран автоматически.

Источники

- https://www.baeldung.com/java-memory-leaks

Комментарии

Оставьте свой комментарий (можно использовать markdown)

Отправить

Популярные По порядку

Web Developer 30 ноября 2021 0 0

Пожалуйста, исправьте грамматические ошибки в тексте. Мешает воспринимать написанное всерьёз.

Ответить •••

space.engineerus 27 ноября 2021 0 0

Про ресурсы - мягкое с тёплым смешали. GC работает только с управляемыми объектами, стоило об этом сказать в самом начале. Поэтому не только эти ресурсы, но и многое другое требует контроллируемого освобождения.

Ответить •••

fivoronov 27 ноября 2021 space.engineerus 0 1

о том и речь, что GC это не про "не думай, оно всегда всё само очистит".

Ответить •••

ВАКАНСИИ

Добавить вакансию

Java-разработчик
Краснодар, от 50000 RUB до 75000 RUB

Middle Frontend разработчик
от 2500 USD до 3000 USD

Frontend developer
Тюмень, от 150000 RUB до 250000 RUB

+ Показать еще

Опубликовать вакансию

ЛУЧШИЕ СТАТЬИ ПО ТЕМЕ

Дорожная карта web-разработчика Java в 2019 году

Java – это огромная экосистема, в которой легко потеряться. Это подробный гайд по фреймворкам с подборкой книг и лайфхаков.

ТОП-10 лучших книг по Java для программистов

Не имеет значения, хотите вы улучшить скилл или только собираетесь начать изучение, здесь вы найдете лучшие книги по Java для программистов.

6 книг по Java для программистов любого уровня

Подборка материалов по Java. Если вы изучаете его, то обязательно найдете для себя что-то полезное и неважно на какой стадии изучения вы находитесь.

Политика конфиденциальности

Контакты

Push-уведомления

Темная тема