Санкт-Петербургский государственный университет

Влаев Никита Владиславович

Выпускная квалификационная работа

Оптимизация загрузки классов из кэша в виртуальной машине Java

Уровень образования: бакалавриат

Направление *09.03.04 «Программная инженерия»*

Основная образовательная программа *СВ.5080.2018 «Программная инженерия»*

Научный руководитель:
д. ф.-м. н., доцент Д. В. Луцив

Консультант:
ассистент кафедры системного программирования А. П. Козлов

Рецензент:
Инженер-программист ООО «Азул Системс» А. К. Петушков

Санкт-Петербург
2022

Saint Petersburg State University

Nikita Vlaev

Bachelor's Thesis

Java class loading optimization in a caching JVM

Education level: bachelor

Speciality *09.03.04 «Software Engineering»*

Programme *CB.5080.2018 «Software Engineering»*

Scientific supervisor:
C.Sc.D, docent D.V. Luciv

Consultant:
System programming department assistant A.P. Kozlov

Reviewer:
Software Engineer at "Azul" A.K. Petushkov

Saint Petersburg
2022

Оглавление

Введение

Язык Java [13] является строго типизированным объектно-ориентированным языком программирования общего назначения, разработанным в компании Sun Microsystems [17] в 1995 году. Java используется для разработки десктопных и мобильных приложений, встроенных систем и т.д. Согласно отчетам компании Oracle [15] (ныне развивающей Java), Java работает на 3 миллиардах устройств по всему миру, что делает его одним из самых популярных языков программирования. В индексе PYPL Java занимает второе место по популярности [16].

В начале своего развития язык Java страдал от недостатка производительности. В число самых эффективных усовершенствований с целью увеличения скорости выполнения программ на Java входит JIT-компиляция (Just-In-Time) [4] – технология трансляции байткода в машинный код непосредственно во время работы программы с возможностью сохранения версий класса в машинном коде. Другими слабыми местами Java являются скорость запуска Java-приложений, скорость достижения максимальной производительности за счет JIT-компиляций («прогрев»), а также сборка мусора и потребление памяти. В рамках данной работы будет обращено особое внимание на проблему скорости запуска Java-приложений.

Временем запуска Java-приложения считается временной отрезок от старта виртуальной машины (JVM) до окончания инициализации Java-приложения, например, до перехода в состояние готовности исполнения запросов от пользователя. Задержка запуска может быть вызвана, в частности, долгим процессом загрузки классов, в ходе которого необходимо выполнять операции загрузки с диска или сложную процедуру инициализации. Такая проблема особенно остро стоит для распределенных систем (Spark [19], Hadoop [1]), так как в этих системах каждый узел выполняет одну и ту же процедуру инициализации.

Для решения данной проблемы в компании Azul [2] был запущен проект C/RaM (Checkpoint/Restore at Main), в результате которого была разработана дополнительная функциональность JVM для Java 8.

Используя C/RaM, пользователь имеет возможность значительно ускорить запуск Java-приложений. C/RaM реализует ускорение загрузки путем старта Java процесса из специально подготовленного образа, который предназначен для исполнения данного приложения. Для доступа к этому образу внутри JVM используется кэш подготовленных Java-классов и JIT-компиляций. В текущей версии C/RaM для доступа к кэш-записи необходимо вычислить метаинформацию о байткоде: значение хеш-функции CRC32 и размер, а для этого необходимо его полностью загрузить с диска, тем самым существенно замедлив запуск приложения, особенно в условиях ограниченной пропускной способности носителя. Это открывает возможность оптимизации процесса загрузки классов, так как на деле для доступа к кэш-записи также можно использовать более эффективные с точки зрения доступа к диску подходы.

В рамках данной работы будут исследованы альтернативные способы получения метаинформации, необходимой для доступа к записи в кэше классов. Затем, необходимо будет внедрить использование данного способа в системные загрузчики классов. Кроме этого разработанный механизм может быть использован пользовательскими загрузчиками классов, поэтому будет описан соответствующий интерфейс.

1. Постановка задачи

Целью данной работы является увеличение производительности кэширующей Java-машины за счет уменьшения объема считываемых с диска данных во время загрузки Java-классов.

Задачи

1. Необходимо выполнить обзор предметной области: изучить процесс загрузки классов в Java, реализацию кэша в C/RaM, а также возможные способы получения метаинформации, необходимой для доступа к записи в кэше классов C/RaM.

2. Оптимизировать механизм загрузки классов JVM, реализовав способы обращения к кэш-записям без необходимости полной загрузки байткода класса.

3. Провести тестирование производительности новой схемы загрузки Java-классов.

2. Обзор

2.1. Процесс загрузки классов в Java

2.1.1. Загрузчики классов

Загрузчик классов – объект класса java.lang.ClassLoader, ответственная за загрузку Java классов в JVM динамически во время исполнения. JRE [10] содержит набор реализаций для загрузчиков и несколько инстанциированных загрузчиков по умолчанию для загрузки классов для первоначальной «раскрутки» приложения. Благодаря загрузчикам классов JVM не обязательно иметь информацию об используемой файловой системе чтобы исполнять Java программы.

Есть 3 встроенных типа загрузчиков классов: Bootstrap, Extension и Appication(или System) [14]. Application загрузчик загружает известные пользователю классы из `classpath`. Extension загрузчик загружает классы, которые являются расширением стандартных Java классов. Bootstrap загрузчик является родителем других загрузчиков и предназначен для загрузки внутренних классов JDK, обычно из `rt.jar` и других системных библиотек. Bootstrap загрузчик, в отличие от остальных, написан на C++, а не на Java.

JVM динамически загружает, связывает и инициализирует классы. Процесс состоит из 5 этапов [12]:

1. Загрузка: найти байткод класса по его имени;
2. Верификация: убедиться что байткод класса удовлетворяет требованиям JVM и безопасен;
3. Подготовка: выделение памяти и присвоение начальных значений для переменных класса;
4. Подстановка: замена символьных ссылок в Constant Pool [11] на прямые;
5. Инициализация: выполнение метода статической инициализации класса.

2.1.2. Механизм делегирования загрузки классов

Между загрузчиками существует отношение «делегирования» [14]: один загрузчик должен «попросить» другой загрузчик загрузить класс вместо себя. В таком случае, первый загрузчик «делегирует» запрос на загрузку второму. Например, пользовательские загрузчики делегируют все системные классы системному загрузчику. Это позволит иметь общие системные классы для всего приложения, таким образом избежав нарушения консистентности системы типов. При этом важно, что разделение пространств имен не нарушается, так как класс уникально определяется его именем и загрузчиком.

Загрузчик, получивший запрос на загрузку класса называется *инициирующим*. Загрузчик, выполнивший полную загрузку класса называется *определяющим*.

Делегирование загрузки класса происходит так: получив запрос на поиск класса, загрузчик классов будет сначала делегировать запрос загрузчику-родителю. Например, если есть запрос на загрузку пользовательского класса в JVM, то Appication загрузчик сначала делегирует этот запрос Extension, а тот в свою очередь загрузчику Bootstrap.

Только если Bootstrap, а затем Extension загрузчик не смогли загрузить класс, Application загрузчик попытается самостоятельно загрузить его (если он был инициирующим).

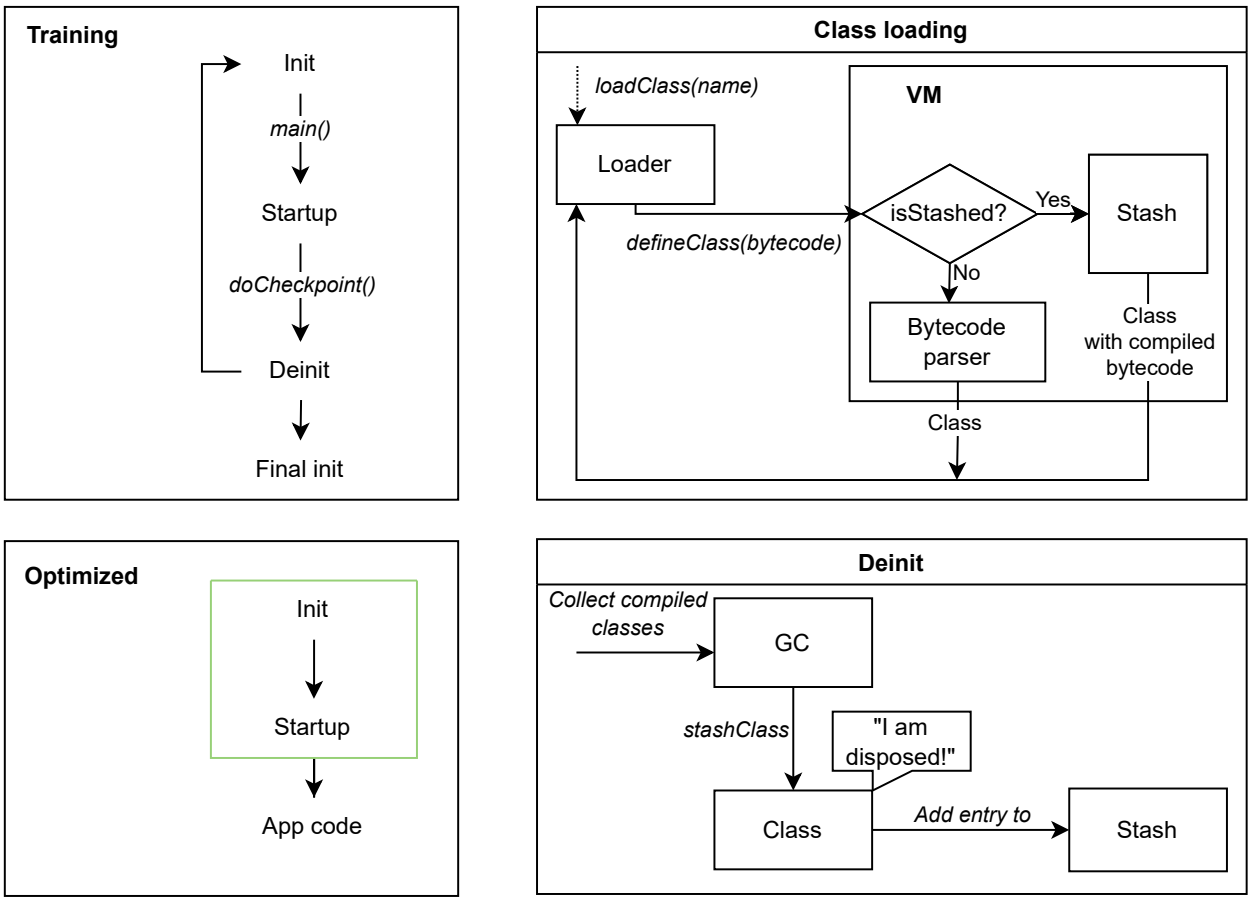
Рассмотрим загрузку классов подробнее на примере: Запрос на загрузку класса `K` поступает от другого класса `D`, в Constant Pool которого есть ссылка на `K`. Запрос содержит только имя класса. JVM вызывает функцию `loadClass(''K'')` загрузчика `D`, которая должна вернуть объект `Class`. Внутри этой функции проверяется, загружен ли этот класс: если да, то возвращается соответствующий объект. Если нет, то запрос на загрузку передается загрузчику-родителю. Процесс повторяется рекурсивно. Затем, если ни один из родителей не загрузил класс, инициирующий загрузчик вызывает свой метод `findClass(''K'')` для поиска байткода класса среди доступных ему ресурсов. Если байткод был загружен, вызывается метод `defineClass(''K'', bytes)`, который пре-

образует байты в объект `Class`. Если класс в итоге не был загружен, генерируется `ClassNotFoundException`.

2.2. Реализация кэша в C/RaM

Кэш используется для ускорения процесса загрузки в оптимизированных запусках приложения. Ускорение достигается за счет загрузки подготовленного образа класса, с методами которого проассоциированы готовые JIT-компиляции.

Рис. 1: Реализация кэша в C/RaM



Кэш классов в C/RaM можно представить как хеш-таблицу, где имени и метаинформации о байткоде класса сопоставляется его внутреннее представление класса в JVM. Каждое из таких представлений может быть связано с несколькими JIT-компиляциями.

Для получения ускорения при использовании C/RaM необходимо провести тренировочные запуски, во время которых производится несколь-

ко итераций прогона приложения от его старта до конца этапа инициализации. В это время формируются JIT-скомпилированнные версии методов, которые затем попадают в кэш. Кэш затем сохраняется в файловой системе для последующей загрузки в оптимизированном запуске.

На схеме 1 изображен в упрощенном виде процесс работы механизма, предназначенного для формирования и использования кэша компиляций. Фаза тренировки (training) состоит из четырех этапов: инициализация(init), загрузка(startup), деинициализация(deinit) и финальная инициализация(final init). Инициализацией считается все, что происходит до вызова метода `main` программы. Загрузкой считается все, что происходит от старта метода `main` до вызова метода `doCheckpoint`, сигнализирующего виртуальной машине, что этап загрузки завершен и необходимо начать деинициализацию. Деинициализация – это приведение приложения и используемых им ресурсов к состоянию, идентичному таковому в момент старта виртуальной машины.

2.2.1. Формирование кэша

На первой итерации фазы тренировки на этапах инициализации и загрузки загрузка классов происходит стандартным образом. Затем, на фазе деинициализации, происходит очистка состояния ВМ, вызывается сборщик мусора. В процессе, сборщик не освобождает внутренние структуры данных, а сохраняет в кэш.

2.2.2. Доступ к кэшу

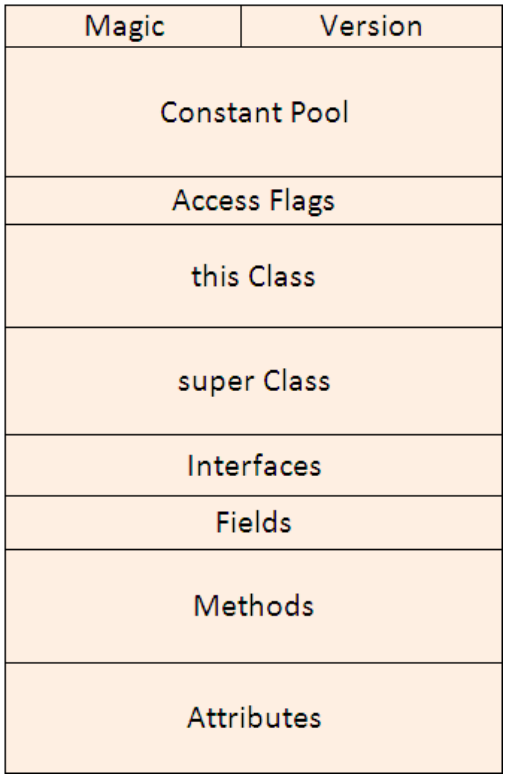
На последующих итерациях фазы тренировки и во время оптимизированного запуска происходит модифицированный процесс загрузки классов. Загрузчик классов, получив запрос на загрузку по имени, находит байткод и передает его в виртуальную машину с помощью метода `defineClass`, как и в обычном процессе загрузки. Затем, происходит проверка, есть ли соотвествующая запись в кэше классов для заданного байткода. Для этого вычисляется метаинформация, в том числе значение хеш-функции для байткода. Если такой класс есть, то он из-

влекается из кэша и используется как результат операции определения класса; иначе байткод передается в парсер байткода и процесс построения внутреннего представления класса не отличается от обычной загрузки.

2.3. Способы получения метаинформации для доступа к записи в кэше классов C/RaM

2.3.1. Формат .class

Рис. 2: Структура .class-файла[1]



[1]https://www.viralpatel.net/tutorial-java-class-file-format-revealed/

В результате компиляции класса Java создается файл формата `.class`, содержащий байткод. Структура `.class`-файла представлена на схеме 2. Важно отметить, что в поле `Magic` хранится специальное значение (`0xCAFEBABE`), отличающее байткод от других данных.

2.3.2. Вычисление метаинформации по байткоду

В C/RaM на момент создания работы для вычисления метаинформации с целью обращения к кэшу классов байткод загружается полностью с помощью стандартной процедуры с использованием метода `loadClass` класслоадеров. Это приводит к излишней нагрузке на файловую систему. При этом, в процессе работы были предложены следующие альтернативные способы найти в кэше внутреннее представление класса, позволяющие избежать такого эффекта.

2.3.3. Сохранение метаинформации во время тренировочного запуска

Одним из возможных подходов к получению метаинформации о байткоде без ущерба производительности программы в момент оптимизированного запуска является сохранение метаинформации, полученной во время тренировочного запуска. Это решение имеет существенный недостаток: если пользователь поменял класс и не перезапустил тренировку, то вместо новой версии класса загрузится старая скомпилированная версия. Для решения данной проблемы необходимо использовать механизм валидации таких сохраненных записей.

Для сохранения метаинформации таким образом необходимо поддерживать структуру для доступа к этому значению по имени класса во время загрузки класса. Это можно делать внутри оперативной памяти, выделенной для JVM, или в файловой системе. Первый способ позволяет загрузить структуру вместе с образом процесса во время контрольного запуска приложения. Такой подход позволяет сократить количество различных запросов к диску во время инициализации процесса, но лишает пользователя возможности самостоятельно инвалидировать записи в такой структуре. Кроме этого, что более важно, это вносит ограничения на файловую систему, внутри которой используется оптимизация, так как для валидации записи будет использоваться метаинформация именно из нее. Второй способ подразумевает фрагментацию доступа к диску для загрузки структуры, однако открывает

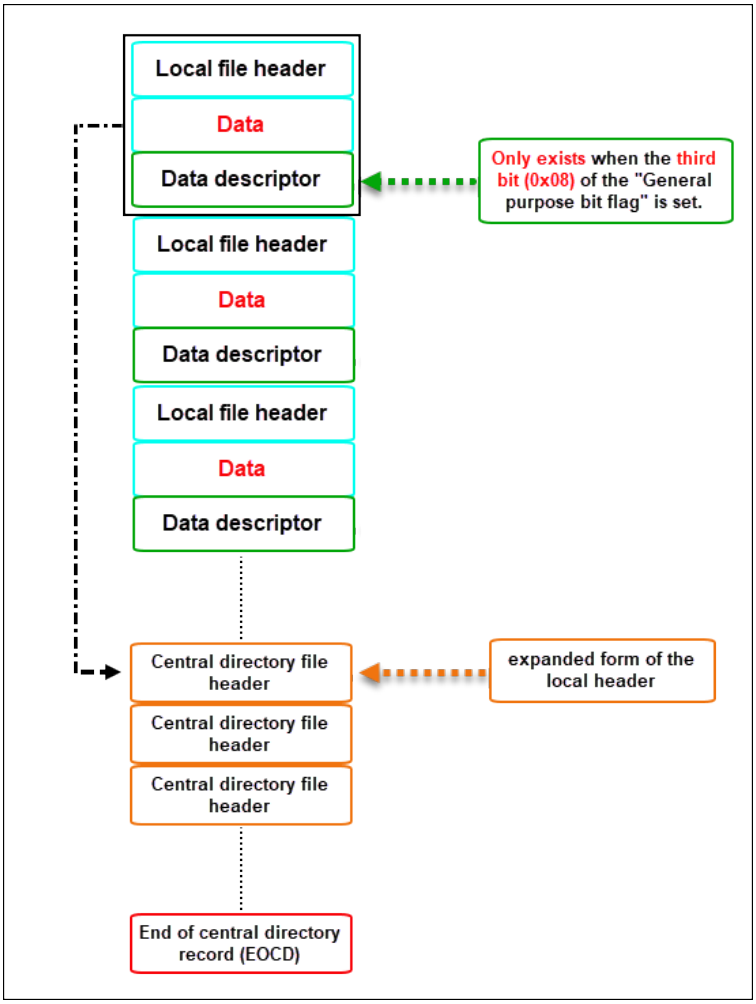
различные возможности для внешнего контроля состояния структуры, в том числе пользователем.

2.3.4. Получение метаинформации из Jar

Другим подходом к получению метаинформации о байткоде является обращение к Jar-файлу [9], в котором хранится байткод класса. Jar-файл является обычным ZIP-архивом [8], содержащим манифест Jar – специальный файл, содержащий информацию о запакованных файлах. Поэтому с Jar-файлом можно работать так же, как с ZIP.

Структура ZIP-архива представлена на схеме 3.

Рис. 3: Структура ZIP-файла[1]



[1]https://nightohl.tistory.com/entry/ZIP-Archive-file-format

ZIP-файлы представляют собой архивы, хранящие несколько фай-

лов. Составляющие архив файлы могут быть сжаты различными способами, в том числе, сохранены без сжатия. В конце ZIP-файла располагается специальная секция, называемая каталогом. В нем хранится список файлов, находящихся в ZIP архиве, и данные о местонахождении каждого сжатого файла внутри архива. С использованием данных каталога приложения могут быстро получить полный список файлов из архива, не читая весь ZIP-архив. Запись в заголовке центральной директории ZIP-архива содержит в себе дополнительные метаданные, содержащие, в том числе, значение CRC32 соответствующих несжатых данных. Это значение возможно использовать для доступа к записи кэша скомпилированных байткодов.

Проблема в том, что готового API для получения этой метаинформации внутри JDK нет, поэтому его необходимо реализовать и интегрировать в классслоадеры.

2.3.5. Реализации в аналогичных системах

CDS(Class-Data Sharing) [18] – JRE [10], которое позволяет преобразовать набор классов из системного jar-файла в общий архив, который может быть отображен в память в момент исполнения для сокращения времени загрузки. CDS также может помочь сократить количество используемой памяти, если несколько JVM разделяют один и тот же общий архив.

AppCDS [18] – расширение CDS на AppClassloader и пользовательские загрузчики классов.

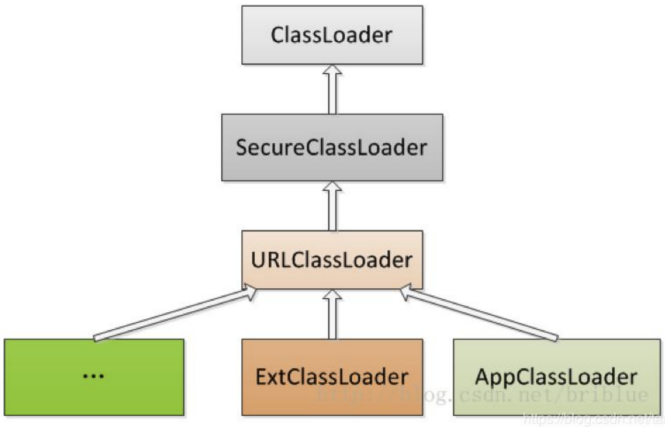
В AppCDS реализовано вычисление метаинфорации напрямую по загруженному байткоду [3].

3. Оптимизация механизма загрузки классов JVM

3.1. Исследование системных загрузчиков Java-классов

Boostrap загрузчик загружает системные классы из Jar-файла – `rt.jar`, а значит теоретически к нему применима оптимизация, основанная на метаданных, хранящихся в записях архива. Исследовав код загрузчика, было выявлено, что в связи с особенностями загрузки системных классов они не попадают в кэш классов. На этапе деинициализации такие классы просто «прячутся», а затем, на следующих итерациях, вновь становятся доступными сразу после старта виртуальной машины. Это имеет смысл, так как системные классы так или иначе загружаются вне зависимости от запущенного приложения. Эти выводы были подтверждены исследованием трейса процесса загрузки классов внутри Bootstrap загрузчика в различных фазах работы C/RaM.

Рис. 4: Иерархия системных загрузчиков классов[1]



[1]https://juejin.cn/post/6995180026668777509

Иерархия системных загрузчиков классов представлена на схеме 4. В результате анализа исходного кода загрузчиков, расположенных внизу иерархии, было выявлено, что они используют `java.net.URLClassloader` как базовую реализацию, отвечающую за до-

ступ к ресурсам с байткодом. Таким образом, имеет смысл глубокая оптимизация именно этого загрузчика.

3.2. Оптимизация java.net.URLClassLoader

Для того, чтобы реализовать оптимизации, основанные на получении метаинформации байткода без необходимости загрузки класса, необходимо предоставить интерфейс для передачи имени класса и его метаинформации из загрузчика в виртуальную машину.

С этой целью была расширена семантика метода `java.lang.Classloader.defineClass`. Его сигнатура:

```
Class<?> defineClass(String name, java.nio.ByteBuffer b, CodeSource cs)
```

Подразумевается, что в качестве параметра `b` типа `java.nio.ByteBuffer` будет передан байткод. Однако байткод от других данных отличает специальный заголовок. Поэтому вместо байткода внутрь виртуальной машины можно передать другие данные с другим заголовком, отличающим их от байткода. Соответствующий буфер для оптимизации формируется так:

```
b = *header* + *CRC32* + *verification_metadata*
```

Таким образом, необходимо лишь получить метаинформацию байткода внутри загрузчика классов, не загружая сам байткод.

3.2.1. Получение метаинформации из Jar

Описанная ранее схема реализована для механизма загрузки классов из Jar внутри URLClassloader. Если для необходимого ресурса невозможно найти метаинформацию в Jar-файле без загрузки, происходит откат на стандартную загрузку класса.

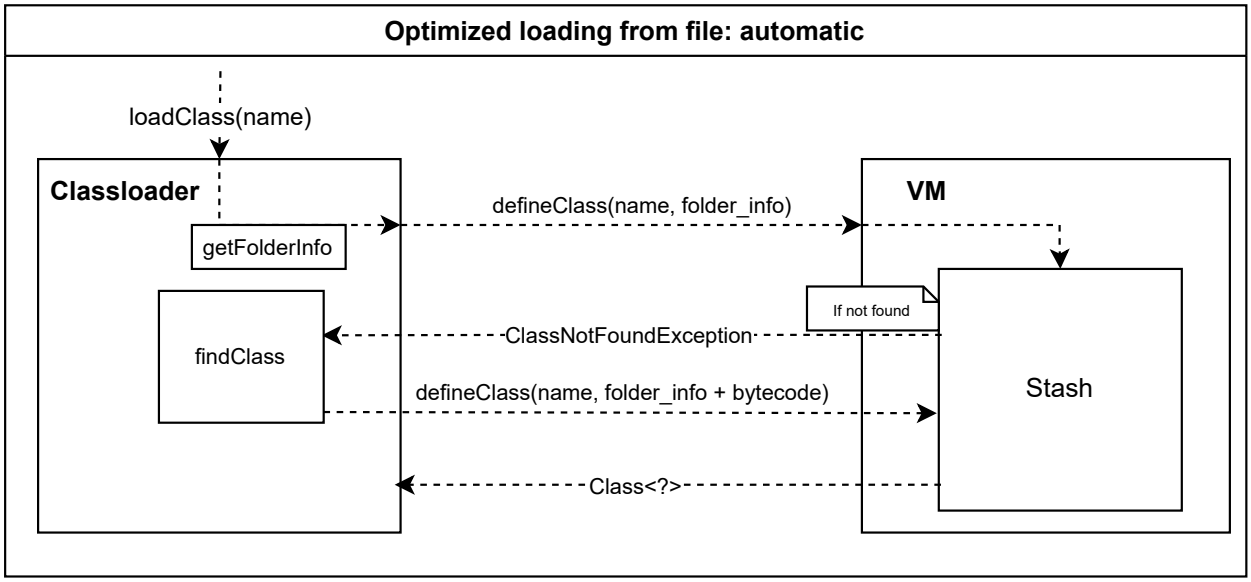
3.2.2. Сохранение метаинформации байткода во время тренировочного запуска

Кроме Jar-архивов, классы зачастую загружаются из обычных .class файлов внутри директорий. В таких случаях используется оптимизация загрузки классов путем сохранения метаинформации о папке(полный

путь, время модификации), откуда загружается класс, внутри кэша. Во время загрузки пара `(имя, метаинформация_о_папке)` используется как ключ для доступа к кэшу, если гарантировано, что метаинформация о папке для данного загрузчика уникально идентифицирует класс. Для этого достаточно, например, чтобы после изменения исходного файла с байткодом обновлялось время модификации папки, его содержащей. Оно используется в качестве верхней оценки реального времени модификации файла. Таким образом, сравнив сохраненное время модификации и верхнюю оценку для его реального значения, можно выяснить, является ли информация в кэше классов актуальной на момент запуска.

Описанный процесс представлен на схеме 5.

Рис. 5: Сохранение метаинформации байткода во время тренировочного запуска



Однако, не все файловые системы поддерживают обновление времени модификации директории согласно содержащимся в ней файлам. Поэтому представлено альтернативное решение: доверить процесс валидации пользователю.

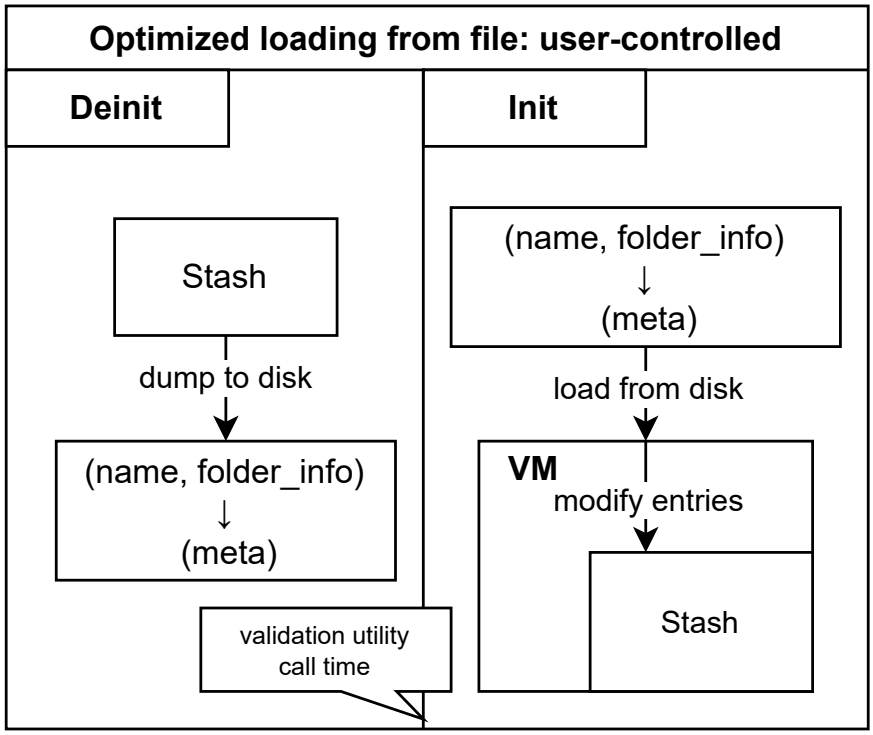
Во время деинициализации из кэша выгружается на диск структура, содержащая отображение имени и метаинформации о папке с файлом класса в метаинформацию класса. Перед следующим оптимизиро-

ванным запуском, пользователь имеет возможность валидировать, насколько данная структура соответствует обновленному состоянию классов в соотвествующих директориях. Для этого была разработана утилита, оптимизирующая этот шаг. Для каждой записи, содержащейся в структуре, она загружает класс по имени и проверяет актуальность метаинформации директории, содержащей класс. Если информация устарела – запись инвалидируется.

На этапе инициализации вышеупомянутая структура загружается с диска в виртуальную машину и используется, в свою очередь, для инвалидации записей в самом кэше. Таким образом, запрос к кэшу, содержащий только имя класса и корректную метаинформацию о директории, где он лежит, достаточны для определения класса, который необходимо вернуть.

Описанный процесс представлен на схеме 6.

Рис. 6: Сохранение метаинформации байткода во время тренировочного запуска



3.3. Предоставление интерфейса для использования оптимизации в пользовательских загрузчиках классов

Обновленная семантика метода `java.lang.Classloader.defineClass` была описана в javadoc(автоматическая документация Java). С помощью этого пользователи могут, разработав собственный способ получения метаинформации байткода не проводя полной загрузки для своего загрузчика, ускорить загрузку классов, использую C/RaM.

Пример такого загрузчика представлен в листинге 1.

Листинг 1: Пример загрузчика

```
import java.io.*;
import java.nio.charset.StandardCharsets;
import java.nio.file.Files;
import java.nio.file.Path;
import java.nio.file.Paths;

import com.google.common.primitives.Bytes;

public class CustomClassLoader extends ClassLoader {
    private byte[] customMetaHeader = "1234".getBytes(StandardCharsets.UTF_8);

    @Override
    public Class findClass(String name) throws ClassFormatError {
        try {
            byte[] b = Bytes.concat(customMetaHeader, getCRC32FromFile(name));
            return defineClass(name, b, 0, b.length);
        } catch (ClassFormatError | IOException e) {
            byte[] b = loadClassFromFile(name);
            return defineClass(name, b, 0, b.length);
        }
    }

    private byte[] getCRC32FromFile(String fileName) throws IOException {
        Path path = Paths.get(fileName.replace('.', File.separatorChar) + ".CRC32");
        return Files.readAllBytes(path);
    }

    private byte[] loadClassFromFile(String fileName)  {
        InputStream inputStream = getClass().getClassLoader().getResourceAsStream(
                fileName.replace('.', File.separatorChar) + ".class");
        byte[] buffer;
        ByteArrayOutputStream byteStream = new ByteArrayOutputStream();
        int nextValue = 0;
        try {
            while ( (nextValue = inputStream.read()) != -1 ) {
                byteStream.write(nextValue);
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
        buffer = byteStream.toByteArray();
        return buffer;
    }
}
```

4. Тестирование производительности

В данной главе представлены результаты апробации оптимизации. Для этого был проведен сравнительный анализ производительности C/RaM и C/RaM с оптимизацией.

Сравнение было проведено на оборудовании: Ubuntu 20.04, Intel core i7-6700 CPU, 3.4GHz, DDR4 16Gb RAM, SSD WDC PC SN730 SDBQNTY-512G-1001.

Для ограничения доступа на диск были использованы опции Docker [6] `--device-read-bps --device-write-bps`. Приложения запускались поочередно на чистой версии и на версии с патчем: кэши файловой системы очищались перед каждым запуском. Важно отметить, что оптимизированная версия C/RaM успешно прошла стандартный набор внутренних тестов в Azul, что показывает ее корректность.

Для тестирования производительности были использованы бенчмарки.

4.1. Реальные бенчмарки

В первую очередь были рассмотрены результаты производительности на бенчмарках, приближенных к реальным приложениям. Такие бенчмарки редко имеют заранее известные характерности в обращении к накопителю, подсистеме памяти (GC) и JIT компилятору. Вместо этого, такие особенности как взаимодействия с диском приходится изучать эмпирическим путём. Так как C/RaM направлен на ускорение процесса запуска Java-приложений, запуском в данном случае имеет смысл считать время от старта до завершения процесса Java-машины, запущенной с этим приложением.

4.1.1. DaCapo

DaCapo [5] – набор тестов, используемый в качестве инструмента для сравнительного анализа Java сообществами по языкам программирования, управлению памятью и компьютерной архитектуре. Он состо-

	1000mb/s	1000mb/s	100mb/s	100mb/s	50mb/s	50mb/s	20mb/s	20mb/s	10mb/s	10mb/s
0	3.320	3.388	3.836	3.914	4.535	4.628	7.554	7.751	12.731	12.991
1	3.390	3.459	3.930	3.962	4.610	4.704	7.544	7.852	12.580	12.837
2	3.404	3.441	3.884	3.967	4.631	4.726	7.463	7.808	12.758	13.018
3	3.378	3.382	3.845	3.923	4.557	4.650	7.554	7.704	12.718	12.978
4	3.396	3.417	3.793	3.870	4.585	4.679	7.507	7.833	12.643	12.901
5	3.323	3.391	3.968	3.999	4.612	4.706	7.535	7.953	12.832	13.094
6	3.303	3.370	3.843	3.921	4.558	4.651	7.641	7.918	12.766	13.027
7	3.386	3.462	3.914	3.944	4.588	4.682	7.493	7.842	12.646	12.904
8	3.389	3.410	3.957	3.981	4.571	4.664	7.515	7.921	12.830	13.092
mean	3.365	3.413	3.885	3.942	4.583	4.677	7.534	7.842	12.723	12.982

Таблица 1: Время запуска DaCapo avrora (в секундах)

	1000mb/s	1000mb/s	100mb/s	100mb/s	50mb/s	50mb/s	20mb/s	20mb/s	10mb/s	10mb/s
avrora	3.365	3.413	3.885	3.942	4.583	4.677	7.534	7.842	12.723	12.982
h2	16.738	17.041	18.440	18.798	21.304	21.587	22.465	29.060	31.016	31.448
lusearch	3.055	3.112	3.557	3.683	4.385	4.492	7.026	7.170	11.764	12.089
lusearch-fix	3.323	3.411	3.411	3.481	4.209	4.547	7.342	7.800	12.501	12.717
pmd	6.467	6.647	7.868	8.074	9.103	9.441	14.430	15.218	23.447	23.925
sunflow	3.004	3.061	3.409	3.485	4.283	4.359	6.912	7.163	12.045	12.305
xalan	2.966	3.378	4.265	4.359	5.016	5.119	9.882	10.084	16.081	16.320

Таблица 2: Среднее время запуска приложений DaCapo (в секундах)

ит из набора реальных приложений с открытым исходным кодом.

В таблице 1 в качестве примера представлены результаты 9 запусков приложения `avrora` из DaCapo. В ячейках таблицы лежит время работы приложения для соответствующего запуска. В колонках таблицы, выделенных серым, приводятся результаты для версии с оптимизацией, в белых – для оригинальной версии. В названии колонки приведено ограничение пропускной способности диска, соответствующее эксперименту. Пронумерованным строкам таблицы соотвествуют единичные эксперименты, в строке с названием «mean» представлены усредненные значения времени для каждой колонки.

В таблице 2 приведено среднее значение времени запуска при различных ограничениях доступа на диск для приложений из DaCapo. Колонки в этой таблице организованы аналогично таблице 1, каждой строке соотвествует бенчмарк.

На графиках 7, 8, 9 проиллюстровано соотношение производительности версий C/RaM при различных ограничениях доступа на диск.

4.1.2. Выводы

Графики показывают улучшение производительности на бенчмарках от 1.5% до 5% при ограничении пропускной способности диска со средним значением 2.5%. Такие значения могут быть обусловлены тем, что фазой инициализации приложения в данном эксперименте считается всё время работы виртуальной машины, а значит загрузка классов занимает пропорционально значительно меньше времени. Кроме этого видно, что при большей пропускной способности ускорение от оптимизации проявляется в меньшей степени, что является ожидаемым результатом, так как оптимизация основана на уменьшении взаимодействия с диском во время загрузки классов.

Рис. 7: DaCapo avrora

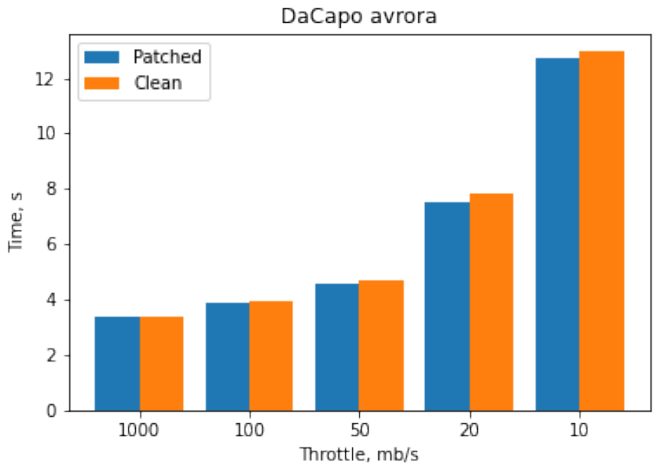


Рис. 8: DaCapo h2

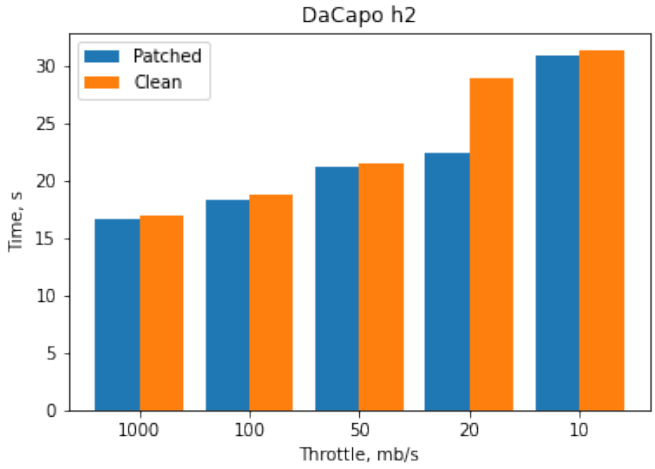
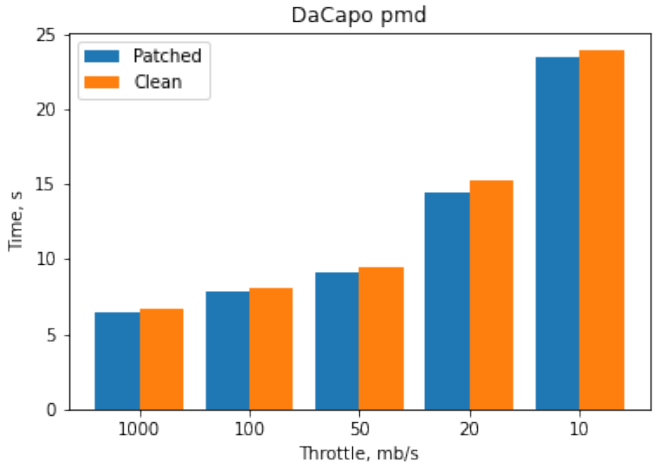


Рис. 9: DaCapo pmd



4.2. Синтетические бенчмарки

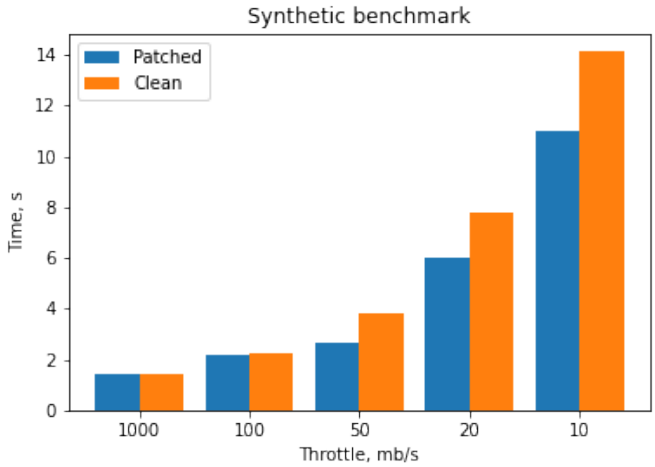
Кроме бенчмарков, принятых для сравнения прозводительности JVM в сообществе разработчиков виртуальных машин, было принято решение использовать синтетический бенчмарк, а именно приложение, которое поочередно загружает классы как из Jar-файлов, так и из директорий файловой системы. Такой бенчмарк является приближением к выделенной и изолированной фазе загрузки Java-классов произвольного приложения. Таким образом, нивелируются эффекты от нагрузки на GC и JIT и становится возможно анализировать только доступ к накопителю. При этом известно [14], что в процессе инициализации и загрузки именно загрузка классов занимает немалую часть времени.

В качестве источника классов для синтетического теста была выбрана библиотека Google Guava [7]. В соответствующем Jar-файле находится 2060 классов суммарным размером 2.9 мб. В таблице 3 и на графике 10 представлены результаты сравнения. Таблица 3 построена аналогично 2.

	1000mb/s	1000mb/s	100mb/s	100mb/s	50mb/s	50mb/s	20mb/s	20mb/s	10mb/s	10mb/s
0	0.976	1.069	2.119	2.162	2.451	3.948	6.380	7.709	10.819	14.097
1	0.983	1.059	2.110	2.153	2.522	3.797	6.524	7.967	10.914	14.202
2	0.976	1.087	2.043	2.085	2.690	3.825	5.624	7.569	11.155	14.182
3	1.296	1.345	2.149	2.193	2.542	3.678	6.038	8.557	10.972	14.159
4	1.085	2.728	2.328	2.376	2.892	4.107	5.892	7.619	10.943	14.159
5	1.265	1.625	2.034	2.075	2.715	3.735	5.905	7.680	10.939	14.156
6	1.074	1.492	2.061	2.103	2.602	4.046	5.895	7.621	11.009	14.165
7	1.279	1.305	2.374	2.408	2.587	3.725	5.880	7.633	11.091	14.122
8	1.101	1.189	2.366	2.988	2.968	3.757	6.002	7.602	11.027	14.156
mean	1.405	1.433	2.176	2.283	2.663	3.846	6.016	7.773	10.985	14.155

Таблица 3: Время запуска синтетического теста (в секундах)

Рис. 10: Синтетические тесты: Guava



4.2.1. Выводы

При уменьшении ограничения доступа на диск улучшение производительности синтетического бенчмарка росло от 0.5% без ограничений до 20% при ограничении в 10 Мб/с. Такие значения показывают, что предложенная оптимизация загрузки классов на фазе инициализации эффективна при ограниченной пропускной способности носителя.

Заключение

1. Выполнен обзор предметной области.

    - Исследован процесс загрузки классов в Java и реализация кэша классов в C/RaM. Выявлены области для потенциальной оптимизации: загрузка классов из Jar-файлов, загрузка классов из директорий.
    - Исследованы способы получения метаинформации байткода Java-класса: загрузка и вычисление, сохранение в процессе подготовки, использование метаданных из Jar-файла.

2. Оптимизирован механизм загрузки классов в C/RaM.

    - Проведено исследование системных загрузчиков: необходимо оптимизировать java.net.URLClassLoader как базовую реализацию для остальных загрузчиков классов в коде стандартной библиотеки Java.
    - Оптимизирован java.net.URLClassLoader: для доступа к кэшу классов использованы подсчитанные метаданные в Jar-файлах и предпосчитанные на этапе подготовки. Разработаны автоматический и ориентированный на пользователя механизмы валидации кэш-записей.
    - Описан интерфейс использования оптимизации для пользовательских загрузчиков.

3. Проведено тестирование производительности новой схемы загрузки: процесс загрузки и инициализации ускорен на 20% в условиях ограниченной пропускной способности диска на синтетических тестах и на 2.5%(от 1.5% до 5%) на DaCapo benchmarks.

Список литературы

[1] Apache Hadoop Architecture, Applications, and Hadoop Distributed File System // Semiconductor Science and Information Devices. — 2022. — 05. — Vol. 4. — P. 14.

[2] Azul company. — https://www.azul.com/.

[3] CDS specification. — https://openjdk.java.net/jeps/310.

[4] Compiling Java just in time / Timothy Cramer, Richard Friedman, Terrence Miller et al. // Ieee micro. — 1997. — Vol. 17, no. 3. — P. 36–43.

[5] The DaCapo Benchmarks: Java Benchmarking Development and Analysis / Stephen M. Blackburn, Robin Garner, Chris Hoffmann et al. — 2006. — P. 169–190. — URL: https://doi.org/10.1145/1167473.1167488.

[6] Docker. — https://www.docker.com/.

[7] Formal language theory: refining the Chomsky hierarchy. — https://github.com/google/guava.

[8] Hepworth Brian, Simpson Dan. The ZIP project // Z User Workshop, Oxford 1990 / Springer. — 1991. — P. 129–133.

[9] JAR format specification. — https://docs.oracle.com/javase/7/docs/technotes/guides/jar/jar.html.

[10] JRE: Java Runtime Environment. — https://docs.oracle.com/javase/specs/.

[11] JVM Constant pool. — https://docs.oracle.com/javase/specs/jvms/se7/html/jvms-5.html#jvms-5.1.

[12] Java class loading stages. — https://docs.oracle.com/javase/specs/jvms/se7/html/jvms-5.html.

[13] The Java language specification / James Gosling, Bill Joy, Guy Steele, Gilad Bracha. — Addison-Wesley Professional, 2000.

[14] Liang Sheng, Bracha Gilad. Dynamic Class Loading in the Java Virtual Machine // SIGPLAN Not. — 1998. — oct. — Vol. 33, no. 10. — P. 36–44. — URL: https://doi.org/10.1145/286942.286945.

[15] Oracle Corporation. — https://www.oracle.com/id/index.html.

[16] PopularitY of Programming Language. — https://pypl.github.io/PYPL.html.

[17] Sun Microsystems. — https://en.wikipedia.org/wiki/Sun_Microsystems.

[18] T saravanan. Java CDS/ISIS: A Rigorous Tool For Effective And Efficient Data Sorting. — 2019. — 08.

[19] Veith Alexandre, Assuncao Marcos. Apache Spark. — 2018. — 01. — P. 1–5.