JVM (Java Virtual Machine) - углубленный разбор

1. ClassLoader (Загрузчик классов)

1.1. Иерархия загрузчиков

JVM использует **делегирующую модель загрузки классов**:

1. **Bootstrap ClassLoader** (нативный, загружает `java.lang`, `java.util` и др. базовые классы)
2. **Extension ClassLoader** (загружает классы из `jre/lib/ext`)
3. **Application (System) ClassLoader** (загружает пользовательские классы из `classpath`)

```
ClassLoader loader = MyClass.class.getClassLoader();
System.out.println(loader); // sun.misc.Launcher$AppClassLoader
```

1.2. Этапы загрузки класса

1. **Загрузка (Loading)** – поиск `.class` файла и создание бинарного представления.
2. **Верификация (Verification)** – проверка корректности байт-кода.
3. **Подготовка (Preparation)** – выделение памяти для статических полей.
4. **Разрешение (Resolution)** – замена символических ссылок на прямые.
5. **Инициализация (Initialization)** – выполнение `<clinit>` (статический блок инициализации).

1.3. Пользовательские ClassLoader'ы

Используются для:

- **Горячего перезапуска** кода (например, в серверах приложений).
- **Изоляции классов** (как в OSGi, Tomcat).
- **Динамической загрузки** (например, плагины).

```
class CustomClassLoader extends ClassLoader {
    @Override
    protected Class<?> findClass(String name) throws ClassNotFoundException {
        byte[] bytecode = loadClassFromCustomSource(name);
        return defineClass(name, bytecode, 0, bytecode.length);
    }
}
```

---

2. Runtime Data Areas (Области памяти JVM)

2.1. Heap (Куча)

- **Хранит объекты** (экземпляры классов).
- **Делится на поколения** (Generational GC):

- **Young Generation** (`Eden`, `Survivor0`, `Survivor1`) – короткоживущие объекты.
- **Old Generation** – долгоживущие объекты.
- **Metaspace** (ранее PermGen) – метаданные классов (Java 8+).

2.2. Stack (Стек потока)

- **Хранит фреймы методов** (локальные переменные, операнды).
- **Каждый поток имеет свой стек**.
- **StackOverflowError** – при переполнении (обычно из-за бесконечной рекурсии).

2.3. Method Area (Область методов)

- **Хранит метаданные классов** (код методов, константы, статические переменные).
- В **Java 8** заменена на **Metaspace** (ранее была частью Heap как PermGen).

2.4. PC Register (Программный счетчик)

- Указывает на **текущую инструкцию** в потоке.
- У каждого потока свой PC.

2.5. Native Method Stack

- Для **нативных методов** (JNI, например, `System.loadLibrary`).

---

3. Garbage Collector (Сборщик мусора)

3.1. Типы сборщиков

Сборщик	Алгоритм	Применение
Serial GC	Mark-Sweep-Compact	Однопоточный, для малых приложений
Parallel GC	Многопоточный Mark-Sweep	По умолчанию в Java 8 (через `-XX:+UseParallelGC`)
CMS (Concurrent Mark-Sweep)	Параллельная маркировка, остановка мира только на финальной сборке	Устарел (deprecated в Java 14)
G1 (Garbage-First)	Разделение кучи на регионы, сборка наиболее заполненных	По умолчанию в Java 9+ (`-XX:+UseG1GC`)
ZGC (Z Garbage Collector)	Работает с огромными кучами (>8TB), паузы <1ms	Java 11+ (`-XX:+UseZGC`)
Shenandoah	Конкурентная компактизация	Альтернатива ZGC (`-XX:+UseShenandoahGC`)

3.2. Как работает GC

1. **Marking** – определение живых объектов (достижимых от `GC Roots`).
2. **Sweeping** – удаление мусора.
3. **Compacting** – дефрагментация памяти (не все GC делают это).

3.3. Типы сборок

- **Minor GC** – очистка **Young Generation**.
- **Major GC** – очистка **Old Generation**.
- **Full GC** – очистка всей кучи (обычно с остановкой приложения, `Stop-The-World`).

---

4. JIT-компиляция (Just-In-Time)

- **Горячий код** (выполняемый часто) компилируется в **нативный машинный код** для ускорения.
- **Уровни оптимизации** (C1, C2 в HotSpot):
  - **C1 (Client Compiler)** – быстрая компиляция, но менее агрессивная оптимизация.
  - **C2 (Server Compiler)** – сложные оптимизации (включая **встраивание методов**, **удаление мертвого кода**).

```
# Параметры JIT
-XX:+TieredCompilation  # Включение многоуровневой компиляции (Java 7+)
-XX:CompileThreshold=10000  # После скольких вызовов метод JIT-компилируется
```

---

5. Настройка памяти

5.1. Основные параметры

```
-Xms256m  # Начальный размер кучи (минимальный)
-Xmx1024m # Максимальный размер кучи
-Xmn128m  # Размер Young Generation
-XX:MetaspaceSize=64m  # Начальный размер Metaspace
-XX:MaxMetaspaceSize=256m  # Максимальный размер Metaspace
```

5.2. Оптимизация под разные сборщики

Для G1:

```
-XX:+UseG1GC
-XX:MaxGCPauseMillis=200  # Целевая пауза GC (мс)
-XX:G1HeapRegionSize=4m   # Размер региона
```

Для ZGC:

```
-XX:+UseZGC
-XX:ZAllocationSpikeTolerance=2.0  # Допустимый всплеск аллокаций
```

---

6. Профилирование JVM

6.1. Инструменты

Инструмент	Назначение
jconsole	Мониторинг кучи, потоков, GC
VisualVM	Расширенный профилировщик (CPU, память)
Java Flight Recorder (JFR)	Запись событий JVM (нагрузка <2%)
jstack	Дамп стека потоков (`jstack <PID>`)
jmap	Анализ памяти (`jmap -heap <PID>`)
jstat	Статистика GC (`jstat -gc <PID> 1s`)

6.2. Пример использования JFR

```
# Запуск записи
java -XX:+FlightRecorder -XX:StartFlightRecording=duration=60s,filename=recording.jfr
MyApp

# Анализ через JMC (Java Mission Control)
jcmd <PID> JFR.start duration=60s filename=recording.jfr
```

Вывод

- **ClassLoader** загружает классы, используя иерархию (Bootstrap → Extension → Application).
- **Heap** хранит объекты, **Stack** – фреймы методов, **Metaspace** – метаданные классов.
- **GC** бывает разных типов (Serial, Parallel, G1, ZGC), каждый под свои задачи.
- **JIT** компилирует горячий код в нативный для ускорения.
- **Настройка памяти** (`-Xms`, `-Xmx`, `-XX:MetaspaceSize`) критична для производительности.
- **Профилирование** (JFR, VisualVM) помогает находить узкие места.