Кеширование в JVM

Волгина Анастасия, 5030102/30101

Виднов Максим, 5030102/30101

Память JVM

- **Куча (Heap)** - объекты и их данные

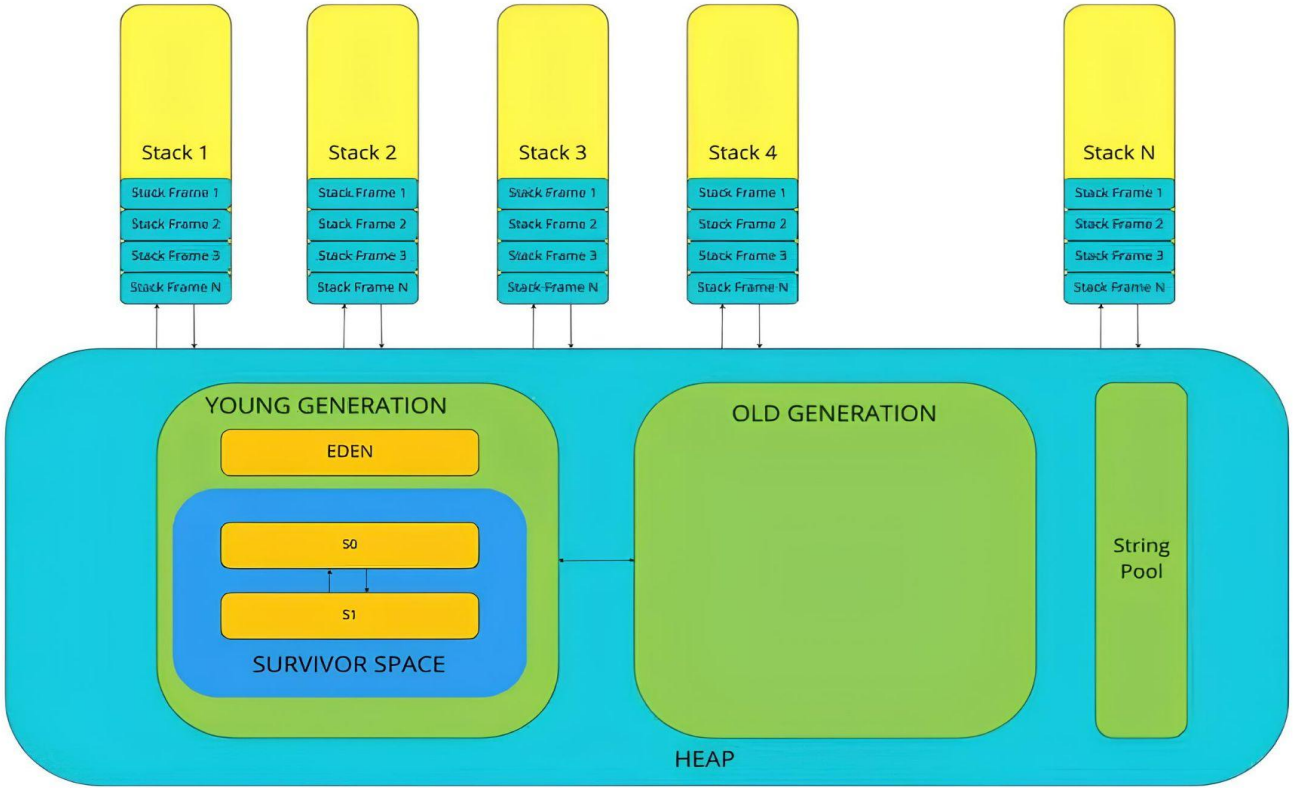
  Содержит пул строк (String Pool)

- **Стек (Stack)** - локальные переменные, ссылки на объекты, информация о вызовах методов

  Для каждого потока свой стек

  Стек - последовательность стековых фреймов (Stack Frame): 1 Stack Frame - 1 вызов метода

  LIFO


Stack 1
Stack Frame 1
Stack Frame 2
Stack Frame 3
Stack Frame N
Stack 2
Stack Frame 1
Stack Frame 2
Stack Frame 3
Stack Frame N
Stack 3
Stack Frame 1
Stack Frame 2
Stack Frame 3
Stack Frame N
Stack 4
Stack Frame 1
Stack Frame 2
Stack Frame 3
Stack Frame N
Stack N
Stack Frame 1
Stack Frame 2
Stack Frame 3
Stack Frame N
YOUNG GENERATION
EDEN
S0
S1
SURVIVOR SPACE
OLD GENERATION
String Pool
HEAP

Память JVM

- **Область методов (Method Area)** - метаданные классов, пул констант, статические переменные, код методов

  Разделяется между всеми потоками

- **Регистры нативных методов (Native Methods Stacks)** - стековые фреймы для вызовов нативных методов

- **Code Cache** - машинный код, скомпилированный JIT компилятором, машинный код самой JVM

Введение

- **Кеширование** - сохранение данных или результатов вычислений для быстрого повторного использования

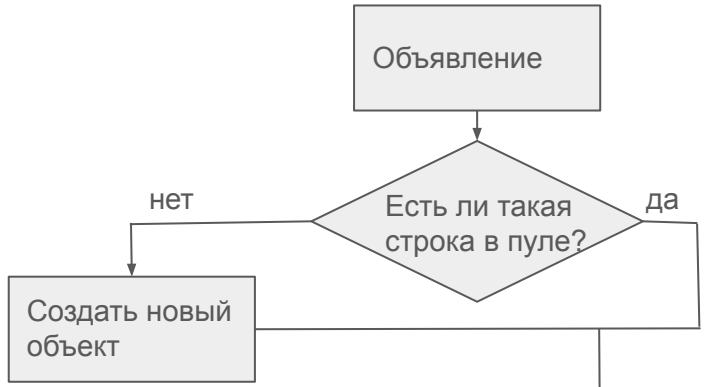
- Кеширование объектов: **пул строк, IntegerCache**

- Кеширование результатов операций: **JIT-компиляция и Code Cache**

Пул строк (String Pool)

- **Пул строк** – область памяти внутри кучи для хранения строковых литералов
- Создаем строки с повторяющимися значениями через **литералы** или метод **intern()** - переменные ссылаются на один и тот же объект

```java
public class StringPoolDemo {
    public static void main(String[] args) {
        String s1 = "hello";
        String s2 = "hello";

        System.out.println("s1 == s2? " + (s1 == s2));

        String s3 = new String("hello");
        System.out.println("s1 == s3? " + (s1 == s3));

        String s4 = s3.intern();
        System.out.println("s1 == s4? " + (s1 == s4));
    }
}
```




```
s1 == s2? true
s1 == s3? false
s1 == s4? true
```

Пул строк (String Pool)

- Позволяет сократить использование памяти при работе со строками, содержимое которых может повторяться

- Управление размером String Pool:

-XX:StringTableSize

-XX:StringTableSizePerBucket

Кеширование объектов классов-оболочек

- Классы-оболочки (**Integer**, **Byte**, **Short**, **Long**, **Character**) кешируют значения в ограниченном диапазоне
- Выполняется при использовании метода **valueOf** и при **автоупаковке**
- Кешируемый диапазон (по умолчанию): **от -128 до 127**
- Расширение верхней границы: **-XX:AutoBoxCacheMax=size**
- Для чего нужно - оптимизация памяти

Кеширование объектов классов-оболочек

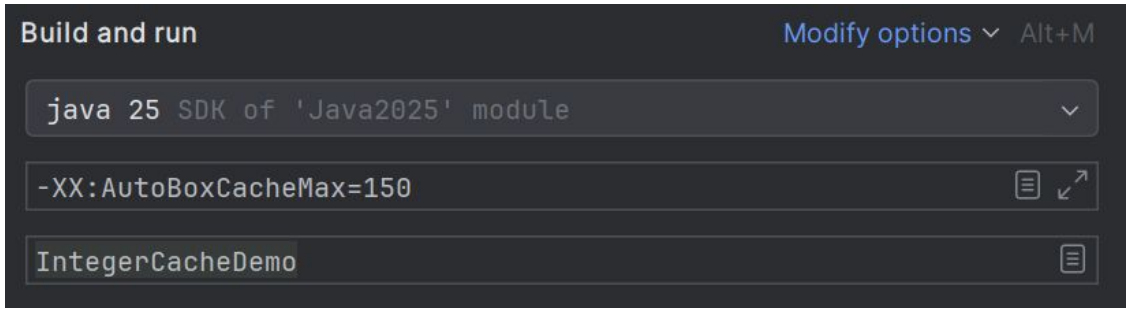
```java
public class IntegerCacheDemo {
    public static void main(String[] args) {
        Integer a = Integer.valueOf(10);
        Integer b = 10;
        System.out.println("a == b? " + (a == b));

        Integer c = 150;
        Integer d = 150;
        System.out.println("c == d? " + (c == d));
    }
}
```

По умолчанию:

```
a == b? true
c == d? false
```

-XX:AutoBoxCacheMax=150

```
a == b? true
c == d? true
```



```java
private static final class IntegerCache {
    static final int low = -128;
    static final int high;

    @Stable
    static final Integer[] cache;
    static Integer[] archivedCache;

    static {
        // high value may be configured by property
        int h = 127;
        String integerCacheHighPropValue =
                VM.getSavedProperty( key: "java.lang.Integer.IntegerCache.high");
        if (integerCacheHighPropValue != null) {
            try {
                h = Math.max(parseInt(integerCacheHighPropValue), 127);
                // Maximum array size is Integer.MAX_VALUE
                h = Math.min(h, Integer.MAX_VALUE - (-low) - 1);
            } catch (NumberFormatException nfe) {
                // If the property cannot be parsed into an int, ignore it.
            }
        }
        high = h;
```

```
// Load IntegerCache.archivedCache from archive, if possible
CDS.initializeFromArchive(IntegerCache.class);
int size = (high - low) + 1;

// Use the archived cache if it exists and is large enough
if (archivedCache == null || size > archivedCache.length) {
    Integer[] c = new Integer[size];
    int j = low;
    // If archive has Integer cache, we must use all instances from it.
    // Otherwise, the identity checks between archived Integers and
    // runtime-cached Integers would fail.
    int archivedSize = (archivedCache == null) ? 0 : archivedCache.length;
    for (int i = 0; i < archivedSize; i++) {
        c[i] = archivedCache[i];
        assert j == archivedCache[i];
        j++;
    }
```

```
            // Fill the rest of the cache.
            for (int i = archivedSize; i < size; i++) {
                c[i] = new Integer(j++);
            }
            archivedCache = c;
        }
        cache = archivedCache;
        // range [-128, 127] must be interned (JLS7 5.1.7)
        assert IntegerCache.high >= 127;
    }

    private IntegerCache() {}
}
```

JIT и Code Cache

- **JIT (Just-In-Time)** компилятор компилирует часто выполняемый код - “горячие точки” (hot spots)
- Скомпилированный JIT компилятором машинный код хранится в **Code Cache**
- Повышение производительности - вместо многократной интерпретации одного и того же байт-кода выполняется быстрый машинный код из кеша
- При заполнении Code Cache JIT-компиляция прекращается

-XX:+UseCodeCacheFlushing - включить очистку Code Cache

JIT и Code Cache

- Управление размером Code Cache:

-XX:InitialCodeCacheSize - первоначальный размер

-XX:ReservedCodeCacheSize - максимальный размер

- Сегменты Code Cache:

- **JVM internal (non-method code)** - машинный код самой JVM (установить размер: -XX:NonNMethodCodeHeapSize)
- **Profiled code** - частично оптимизированный машинный код с коротким временем жизни (установить размер: -XX:ProfiledCodeHeapSize)
- **Non-profiled code** - полностью оптимизированный код с потенциально долгим временем жизни (установить размер: -XX: NonProfiledCodeHeapSize)

```java
public class JITCacheDemo {
    public static double process(int iterations) {  1 usage
        double result = 0;
        for (int i = 0; i < iterations; i++) {
            result += i * 0.001;
        }
        return result;
    }

    public static void main(String[] args) {
        long total = 0;
        for (int i = 0; i < 1_000_000; i++) {
            long start = System.nanoTime();
            double res = process(1000);
            total += (System.nanoTime() - start);

            if (i == 9_999) {
                System.out.println("Среднее время за первые 10 000 вызовов: " + (total / 10_000) + " нс");
            }
        }
        System.out.println("Среднее время за 1 000 000 вызовов: " + (total / 1_000_000) + " нс");
    }
}
```

```
Среднее время за первые 10 000 вызовов: 2641 нс
Среднее время за 1 000 000 вызовов: 253 нс
```

Заключение

- Пул строк, кеширование в классах-оболочках - позволяют избежать создания дублирующихся объектов

- JIT-компиляция и Code Cache - позволяют избежать повторной интерпретации одного и того же байт-кода за счет хранения скомпилированного машинного кода для “горячих точек”

Итог: оптимизация памяти и времени выполнения. Критично для приложений, работающих с большим количеством данных, высоконагруженных сервисов.

Источники

- https://proselyte.net/jvm-basics/
- https://java.msk.ru/integer-cache-%D0%B2-java/
- Исходный код JDK (java.lang.Integer)
- https://habr.com/ru/articles/536288/
- https://docs.oracle.com/javase/8/embedded/develop-apps-platforms/codecache.htm#A1100181

Спасибо за внимание!