

# Боремся за память



Клебанов С.Г.

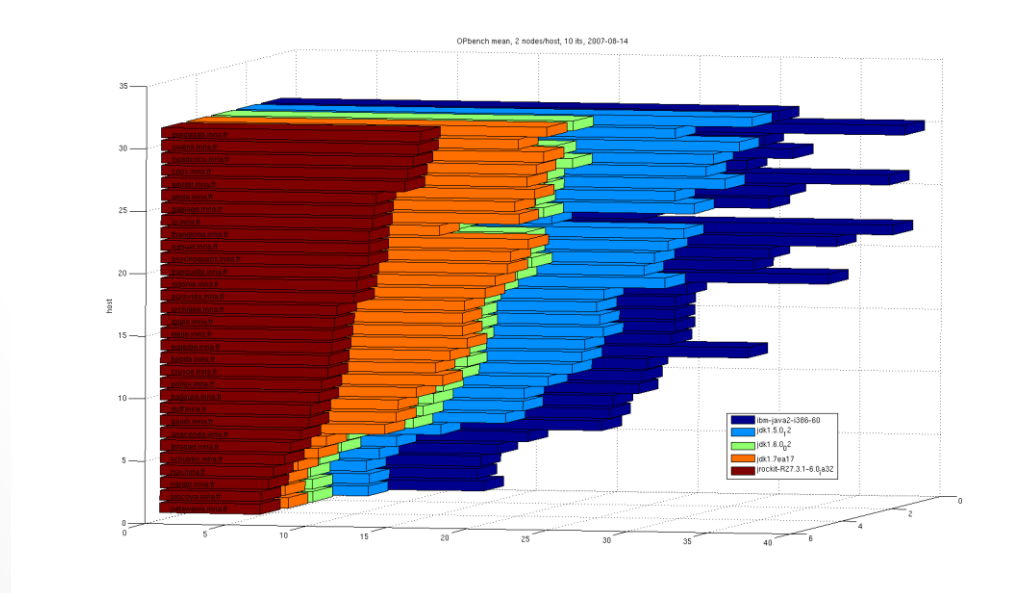
Матющенко Р.О.

# Содержание

- Объект в Java
- Сколько занимают объекты в памяти
- Коллекции
- Решения, позволяющие сократить расход памяти

# Цель

- Оптимизация использования ресурсов
- Полезно «знать объем двигателя машины, на которой Вы ездите»



Сколько нужно памяти для  
хранения десяти миллионов целых  
чисел в HashSet?



Сколько нужно памяти для хранения десяти миллионов целых чисел в HashSet?

подсказка:

$$4 \text{ байта} * 10.000.000 = 40 \text{ мб}$$

In java — everything is an object\*

\* Кроме, пожалуй, примитивов и ссылок на сами объекты

# Объект

Для каждого объекта JVM хранит:

- 1) Заголовок объекта
- 2) Память для примитивных типов
- 3) Память для ссылочных типов
- 4) Смещение/выравнивание

# Структура заголовка объекта

Каждый экземпляр класса содержит заголовок. Каждый заголовок для большинства JVM(Hotspot, openJVM) состоит из двух машинных слов.

	32-х разрядная система	64-х разрядная система
Размер заголовка	8 байт	16 байт

Структура заголовка:

- Mark Word
  - Hash Code
  - Garbage Collection Information
  - Lock
- Type Information Block Pointer
- Array Length



# Структура заголовка объекта

*Mark Word* — содержит *Hash Code*, *Garbage Collection Information*, *Lock*

*Hash Code* — каждый объект имеет хеш код. По умолчанию результат вызова метода `Object.hashCode()` вернет адрес объекта в памяти.

*Garbage Collection Information* — каждый java объект содержит информацию нужную для системы управления памятью.

*Lock* — каждый объект содержит информацию о состоянии блокировки. Это может быть указатель на объект блокировки или прямое представление блокировки.

# Структура заголовка объекта

*Type Information Block Pointer* — содержит информацию о типе объекта (таблица виртуальных методов, указатель на объект, указатели на некоторые дополнительные структуры).

*Array Length* — если объект — массив, то заголовок расширяется 4 байтами для хранения длины массива.

# Смещение/выравнивание

По сути, это несколько неиспользуемых байт, которые размещаются после данных самого объекта. Это сделано для того, чтобы адрес в памяти всегда был кратным машинному слову. Это необходимо для:

- ускорения чтения из памяти
- уменьшения количества бит для указателя на объект
- для уменьшения фрагментации памяти

Стоит также отметить, что в java размер любого объекта кратен 8 байтам.

# Примитивы

Type	Java Lang Specification
byte	1 byte
short	2 byte
int	4 byte
long	8 byte
char	2 byte
float	4 byte
double	8 byte
boolean	1 bit

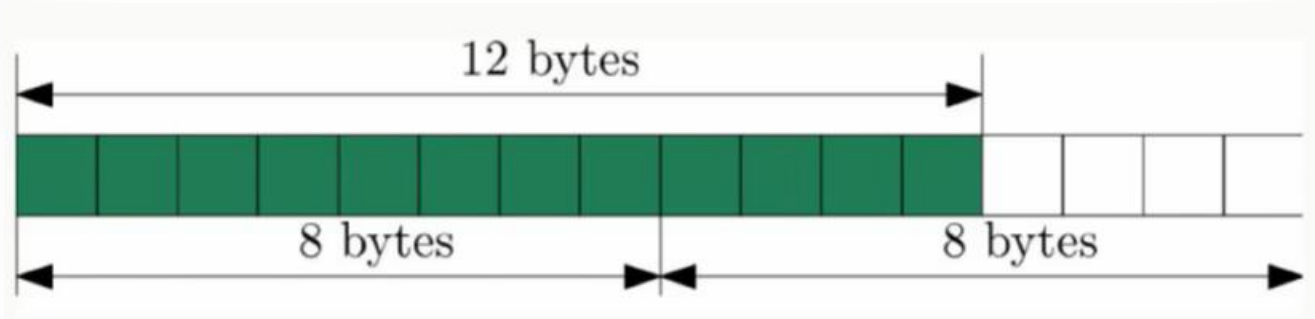
# Объекты

Что же говорит спецификация для объектов?

Ничего, кроме того, что у каждого объекта есть заголовок.

Иными словами, размеры экземпляров Ваших классов могут отличаться от одной JVM к другой.

# Объекты



8 bytes          @ 32 bit JVM

12 bytes        @ 64 bit JVM

# Объекты

```
//первый случай  
int a = 300;
```

4-х байтная переменная, которая содержит значение из стека.  
Размер - **sizeof(int)**

```
//второй случай  
Integer b = 301;
```

Ссылочная переменная и сам объект, на который эта переменная ссылается.  
Размер - **sizeof(reference) + sizeof(Integer)**

# Объекты

## Класс **Integer**

Заголовок	8 байт
Поле int	4 байта
Выравнивание для кратности 8	4 байта
Итого	16 байт

## Класс **String**:

```
private final char value[];  
private final int offset;  
private final int count;  
private int hash;
```

Заголовок	8 байт
Поля int	4 байта x 3 = 12 байт
Ссылочная переменная на объект массива	4 байта
Итого	24 байта



# Объекты

**new String(«a»)**

new String()

Заголовок	8 байт
Поля int	4 байта x 3 = 12 байт
Ссылочная переменная на объект массива	4 байта
Итого	24 байта

new char[1]

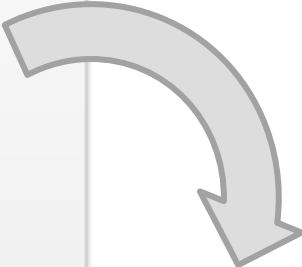
Заголовок	8 байт + 4 байта = 12 байт
Примитивы char	2 байта x 1 = 2 байта
Выравнивание для кратности 8	2 байта
Итого	16 байт

Итого, **new String("a") = 40 байт**


**new String("a")** и **new String("ab")** занимают одинаковое количество памяти

# Денормализация модели

```
class Cursor {  
    String icon;  
    Position pos;  
    Cursor(String icon, int x, int y) {  
        this.icon = icon;  
        this.pos = new Position(x, y);  
    }  
}
```



```
class Position {  
    int x;  
    int y;  
    Position(int x, int y) {  
        this.x = x;  
        this.y = y;  
    }  
}
```



```
class Cursor2 {  
    String icon;  
    int x;  
    int y;  
    Cursor2(String icon, int x, int y) {  
        this.icon = icon;  
        this.x = x;  
        this.y = y;  
    }  
}
```

# Денормализация модели

Cursor2 потребляет приблизительно на **30% меньше** памяти чем объект класса Cursor (по сути Cursor + Position).

*Это не призыв к созданию огромных классов по 100 полей. Ни в коем случае. Это может пригодиться исключительно в случае, когда Вы вплотную подошли к верхней границе Вашей оперативной памяти и в памяти у Вас много однотипных объектов.*

# Используем смещение в свою пользу

```
class A
{
    int a;
}

class B
{
    int a;
    int b;
}
```

Объекты класса A и B потребляют одинаковое количество памяти. Тут можно сделать сразу 3 вывода:

- Стоит ли добавить еще одно поле в класс или сэкономить и высчитать его позже на ходу? Иногда глупо жертвовать процессорным временем ради экономии памяти, учитывая что никакой экономии может и не быть вовсе.
- Хранить в поле дополнительные или промежуточные данные для вычислений или кеша (пример поле hash в классе String).
- Иногда нет никакого смысла использовать byte вместо int, так как за счет выравнивания разница все равно может нивелироваться.

# Порядок имеет значение

Допустим у нас есть два массива:

`Object[2][1000]`

`Object[1000][2]`

С точки зрения потребления памяти — разница колоссальна. В первом случае мы имеем 2 ссылки на массив из тысячи элементов. Во втором случае у нас есть **тысяча ссылок на двумерные массивы!**

Во втором случае количество потребляемой памяти больше на 998 размеров ссылок. А это около 7кб.

Вот так на ровном месте можно потерять достаточно много памяти.

# Boolean и boolean

Размер логического типа **полностью зависит от Вашей JVM.**

Например, в Oracle HotSpot JVM под логический тип выделяется 4 байта, то есть столько же сколько и под int. За хранение 1 бита информации Вы платите 31 битом.

# Массив boolean

Класс **BitSet** ведёт себя подобно массиву `boolean`, но упаковывает данные так, что для одного бита выделяется всего один бит памяти (с небольшими издержками для всего массива).

**BitSet** хранит внутри себя массив типа `long`, а при запросе или установке значения определенного бита — высчитывает индекс нужного `long` и пользуясь побитовыми операциями и операциями сдвига производит вычисления над единственным битом.

```
BitSet bits = new BitSet();  
bits.set(0); // set the 0th bit  
bits.set(6); // set the 6th bit
```

# JMV cost

Type	JLS	JMV cost
byte	1 byte	1..8 bytes
short	2 byte	2..8 bytes
int	4 byte	4..8 bytes
long	8 byte	8..16 bytes
char	2 byte	2..8 bytes
float	4 byte	4..8 bytes
double	8 byte	8..16 bytes
boolean	1 bit	1..8 bytes



# JMV cost + wrap

Type	JLS	JMV cost	Wrapper*
byte	1 byte	1..8 bytes	16 bytes
short	2 byte	2..8 bytes	16 bytes
int	4 byte	4..8 bytes	16 bytes
long	8 byte	8..16 bytes	24 bytes
char	2 byte	2..8 bytes	16 bytes
float	4 byte	4..8 bytes	16 bytes
double	8 byte	8..16 bytes	24 bytes
boolean	1 bit	1..8 bytes	16 bytes

\* 64 bit JVM objects, thus adding 12 bytes

# Сколько памяти занимает класс?

```
public class
```

```
FooClass
```

```
{
```

```
    Object x = null;
```

```
    Object y = null;
```

```
}
```

- 16 bytes
- 24 bytes
- 36 bytes

# Сколько памяти занимает класс?

- **32 bit**

8 header + 4 ref + 4 ref = 16 bytes

- **64 bit + Compressed OOPs (Xmx < 32 gb)**

12 header + 4 ref + 4 ref = 20 (align) → 24 bytes

- **64 bit – Compressed OOPs (Xmx > 32 gb)**

12 header + 8 ref + 8 ref = 28 (align) → 32 bytes

# Compressed OOPs

В 32-х разрядных системах размер указателя на ячейку памяти занимает 32 бита. Следовательно максимально доступная память, которую могут использовать 32-х битные указатели —  $2^{32} = 4294967296$  байт или 4 ГБ.

В 64-х разрядных системах соответственно можно ссылаться на  $2^{64}$  объектов.

Такое огромное количество указателей излишне. Поэтому появилась опция сжатия ссылок - **XX:+UseCompressedOops**.

Это опция позволила уменьшить размер указателя в 64-х разрядных JVM до 32 бит.

# Compressed OOPs

## Profit

- Все объекты у которых есть ссылка, теперь занимают на 4 байта меньше на каждую ссылку.
- Сокращается заголовок каждого объекта на 4 байта.
- В некоторых ситуациях возможны уменьшенные выравнивания.
- Существенно уменьшается объем потребляемой памяти.

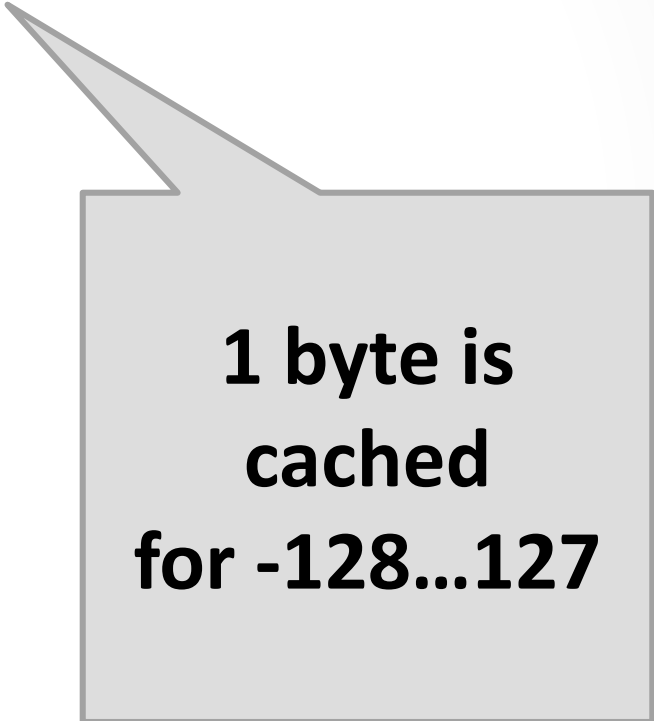
## Loss

- Количество возможных объектов упирается в  $2^{32}$ . Этот пункт сложно назвать минусом.
- Появляются доп. расходы на преобразование JVM ссылок в нативные и обратно.

# Flyweights

~~int myVar = new Integer(5)~~

- <PrimitiveWrapper>.valueOf()
  - Byte
  - Short
  - Integer
  - Long
  - Character
- String.intern()



**1 byte is  
cached  
for -128...127**

# Collections

Collection (10 mln ints)	Overhead
Pure data	0 (40 mb)
int[]	almost 0 (40 mb)
Integer[]	5x (200 mb)
Integer[] (valueOf)	< 5x (200 mb)
ArrayList<Integer>()	5.15x (205 mb)
ArrayList<Integer>(10 mln)	< 5x (200 mb)
HashSet<Integer>()	13.7x (547 mb)
HashSet<Integer>(10 mln)	13.7x (547 mb)

**+16 bytes**

**+10 mln \* 4 bytes  
+10 mln \* 16 bytes**

**Preallocation**

**Preallocation**

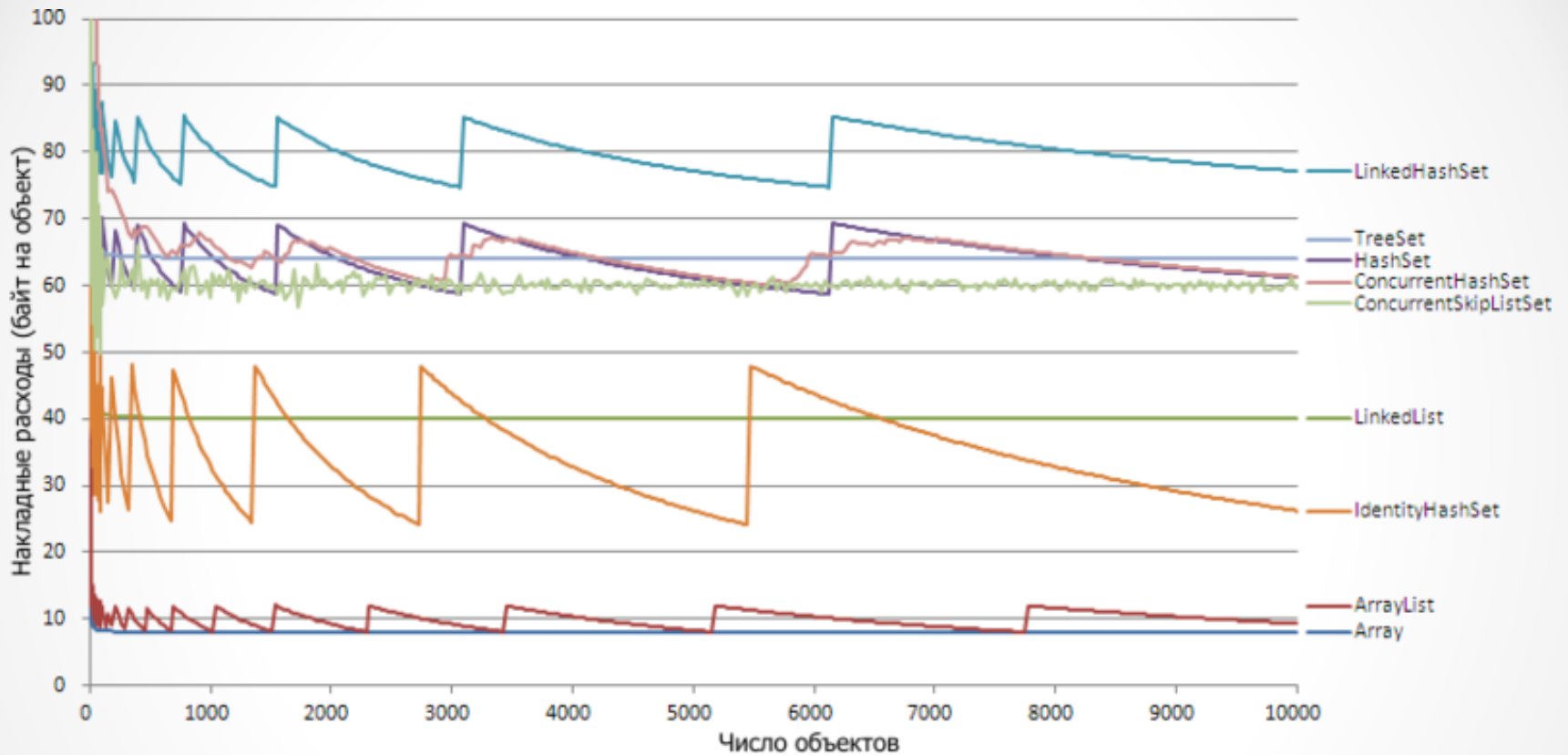
# Collections

- 500 коллекций каждого типа
- Количество элементов от 1 до 10000.
- В качестве элементов использовались строки случайной длины и значения.

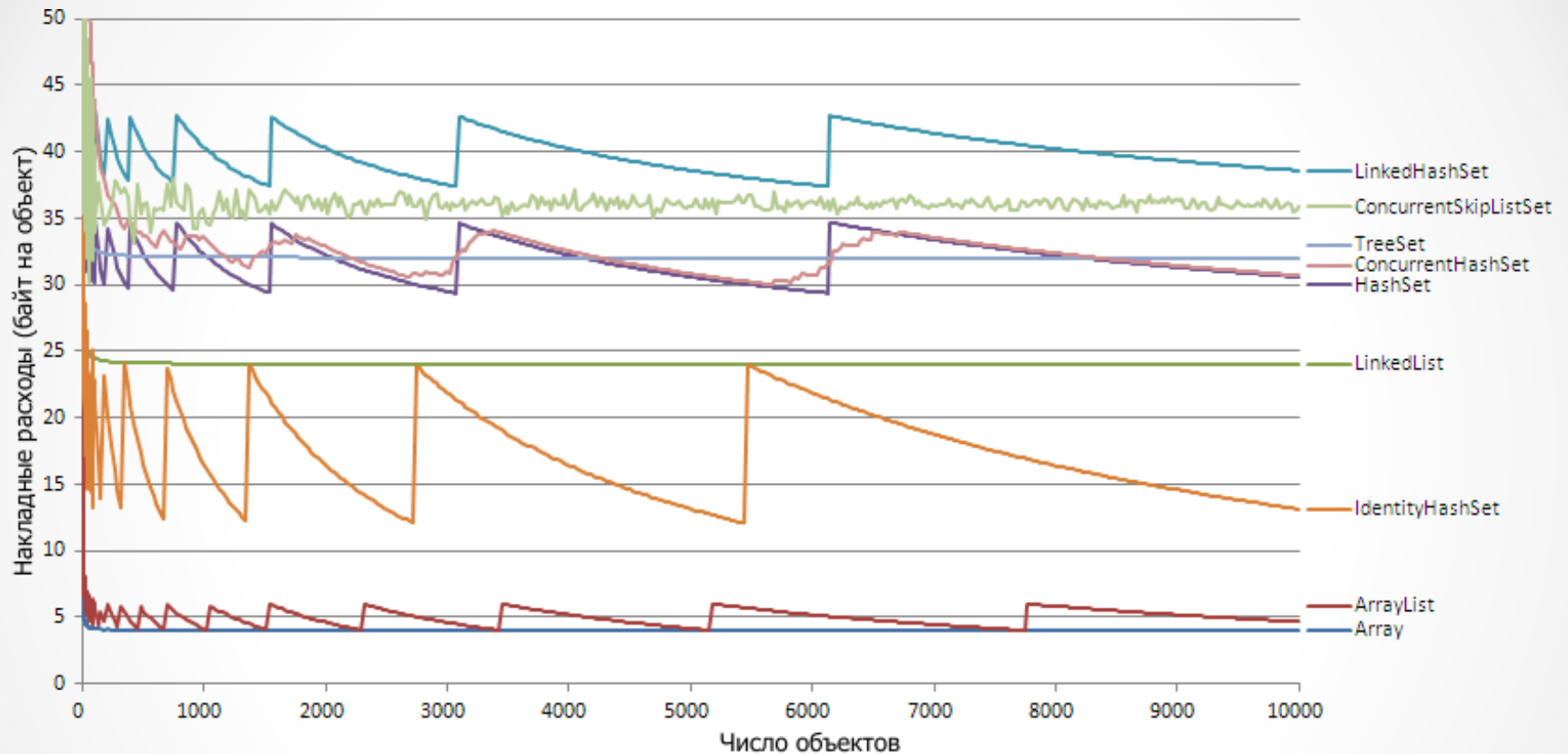
\*элементы (незначительно) влияют лишь на  
ConcurrentHashSet



# 64-bit Hotspot JVM (Java 1.6)



# 32-bit Hotspot JVM (Java 1.6)



# Array

Самый простой вариант — массив, для которого наперёд известно число элементов.

На каждый объект хранится ссылка	<b>4 (8) байт</b>
Длина массива (int)	<b>4 байта</b>
Header объекта	<b>8 (12) байт</b>
Пустой массив	<b>16 (24) байт</b>

# ArrayList

Т.к. заранее число элементов в массиве неизвестно, массив выделяется с запасом (по умолчанию на 10 элементов) и при необходимости расширяется чуть больше, чем в полтора раза:

**`int newCapacity = (oldCapacity * 3)/2 + 1`**

Поэтому график прыгает до 6 (12) байт.

На каждый объект хранится ссылка	<b>4 (8) байт</b>
Array	<b>16 (24) байт</b>
Header объекта	<b>8 (12) байт</b>
Фактический размер списка	<b>4 байта</b>
Количество модификаций	<b>4 байта</b>
Созданный по умолчанию пустой ArrayList	<b>80 (144) байт</b>

Это самый экономный способ хранить однотипные данные, если заранее неизвестно их количество.

# LinkedList

Для связанного списка картина похожа на массив.

Для каждого элемента списка создаётся по одному служебному объекту типа **java.util.LinkedList.Entry**. Каждый из этих объектов содержит по три ссылки:

- Сам элемент списка
- Previous Entry
- Next Entry

При этом из-за выравнивания в 32bit теряется по 4 байта, поэтому в итоге требуется **24 (40) байт на каждый Entry**.

# TreeSet

График тоже похож на LinkedList и массив.

Для каждого элемента создаётся ветвь дерева

java.util.TreeMap.Entry, которая содержит пять ссылок:

**ключ, значение, родитель, левый и правый ребёнок.**

Кроме них хранится **булева переменная**, указывающая цвет ветки, красный или чёрный. Entry занимает **32 (64) байта**.

Постоянные данных в TreeMap такие:

- ссылка на компаратор
- ссылка на корень дерева
- ссылки на entrySet, navigableKeySet, descendingMap
- размер и количество модификаций

В сумме выходит **64 (104) байта**.

# HashSet

HashSet основан на HashMap. Для каждого элемента заводится запись `java.util.HashMap.Entry`, содержащая ссылки на **ключ, значение, следующую Entry**, а также само значение хэша.

Всего Entry занимает **24 (48) байт**.

Помимо Entry есть ещё и **хэш-таблица, со ссылками на Entry**, которая содержит 16 элементов изначально и увеличивается вдвое, когда количество элементов превышает 75% (default) от её размера. То есть при конструировании по умолчанию увеличение таблицы происходит, когда количество элементов превышает 12, 24, 48, 96 и т. д. ( $2^n * 3$ , последний всплеск на графике — 6144 элемента).

По умолчанию пустой HashSet весит **136 (240) байт**.

# LinkedHashSet

Используется `java.util.LinkedHashMap.Entry`, которая наследует `java.util.HashMap.Entry`, добавляя две ссылки на **предыдущий** и **следующий** элементы, поэтому график на 8 (16) байт выше, чем для `HashSet`, достигая перед расширением таблицы 37.33 (74.67), а после — рекордных 42.67 (85.33).

Константа тоже увеличилась, так как наподобие `LinkedList` хранится головной `Entry`, который не ссылается на элемент множества.

Свежесозданный `LinkedHashSet` занимает **176 (320) байт**.



# IdentityHashMap

- IdentityHashMap сравнивает ключи по `==`, а не по `equals` и использует `System.identityHashCode`.
- Не создаёт объектов вроде `Entry`
- В случае коллизии не создаёт список, а записывает объект в первую свободную ячейку по ходу массива.

IdentityHashMap увеличивает размер массива вдвое каждый раз, когда он заполнен больше, чем на  $2/3$ . По умолчанию массив создаётся на 32 элемента. Расширение происходит при превышении 21, 42, 85, 170 и т. д.

Перед расширением массив содержит **в 3 раза** больше элементов, чем ключей в IdentityHashMap, а после расширения — **в 6 раз**.

Таким образом, накладные расходы составляют от **12 (24) до 24 (48) байт** на элемент.

Пустое множество по умолчанию занимает довольно много — **344 (656) байт**, но уже при девяти элементах становится экономичнее всех прочих множеств.

# ConcurrentHashMap

ConcurrentHashMap — первая коллекция, в которой график зависит от самих элементов (а точнее от их хэш-функций).

Это набор фиксированного числа сегментов (по умолчанию их 16), каждый из которых является синхронным аналогом HashMap.

Часть бит из модифицированного хэш-кода используется для выбора сегмента, обращение к разным сегментам может происходить параллельно. В пределе накладные расходы совпадают с накладными расходами самого HashMap.

Увеличение размера сегментов происходит независимо, потому график не поднимается одномоментно: сперва увеличиваются сегменты, в которые попало больше элементов.

Эта коллекция вышла на первое место по начальному размеру — **1304 (2328) байт**, потому что сразу же заводится **16 сегментов**, в каждом из которых таблица на **16 записей** и несколько вспомогательных полей.

Однако для 10000 элементов ConcurrentHashMap превышает размер HashSet всего на **0.3%**.

# Результат

Array	16(24) байт
ArrayList	80 (144) байт
LinkedList	24 (40) байт
TreeSet	64 (104) байт
HashSet	136 (240) байт
LinkedHashSet	176 (320) байт
IdentityHashMap	344 (656) байт
ConcurrentHashMap	1306 (2428) байт

# Результат

- Практически нигде расход памяти не зависит от самих объектов
- Степень сбалансированности дерева, количество коллизий в хэш-таблицах ни на что не влияют.
- Для неконкуррентных коллекций можно заранее определить размер накладных расходов с точностью до байта.

Как тогда уменьшить расходы  
памяти?



# Solutions. Trove

Collection (10 mln ints)	Size
Pure data	40 mb
TIntArrayList	almost 1.05x (42 mb)
TIntArrayList(10 mln)	almost 0 (40 mb)
TIntHashSet	almost 3.3x (131 mb)
TIntHashSet(10 mln)	almost 2.6x (105 mb)

# Solutions. More

Collection (10 mln ints)	Size
Pure data	40 mb
<b>fastutils</b> IntOpenHashSet	almost 2.1x (83 mb)
<b>org.a.c.c.p.</b> ArrayIntList	almost 1.4x (55 mb)
<b>hppc</b> IntIntOpenHashMap	almost 3.8x (150 mb)
<b>cern.colt.map</b> OpenIntIntHashMap	almost 6.5x (260 mb)

Нет Set,  
только списки

Есть Map, но  
больше  
overhead

# Solutions. MapDB

JVM 64bit Oracle Java 1.7.0\_09-b05 с HotSpot 23.5-b02

Включены сжатые указатели (-XX: + UseCompressedOops)

Сколько записей можно вставить в map с 16 Гб  
оперативной памяти?





# Solutions. MapDB

Мы всегда можем уйти от динамической памяти, где сборщик мусора не увидит наши данные.

**MapDB** предоставляет TreeMap и HashMap при поддержке базы данных.

Поддерживает различные режимы хранения, включая вариант который не в динамической памяти.

# Solutions. MapDB

```
Map m = new TreeMap();  
for(long counter=0;;counter++)  
{  
    m.put(counter, "");  
    if(counter%1000000==0)  
        System.out.println(""+counter);  
}
```

172 mln

276 mln

Аналогичный цикл с LongHashMap

980 mln

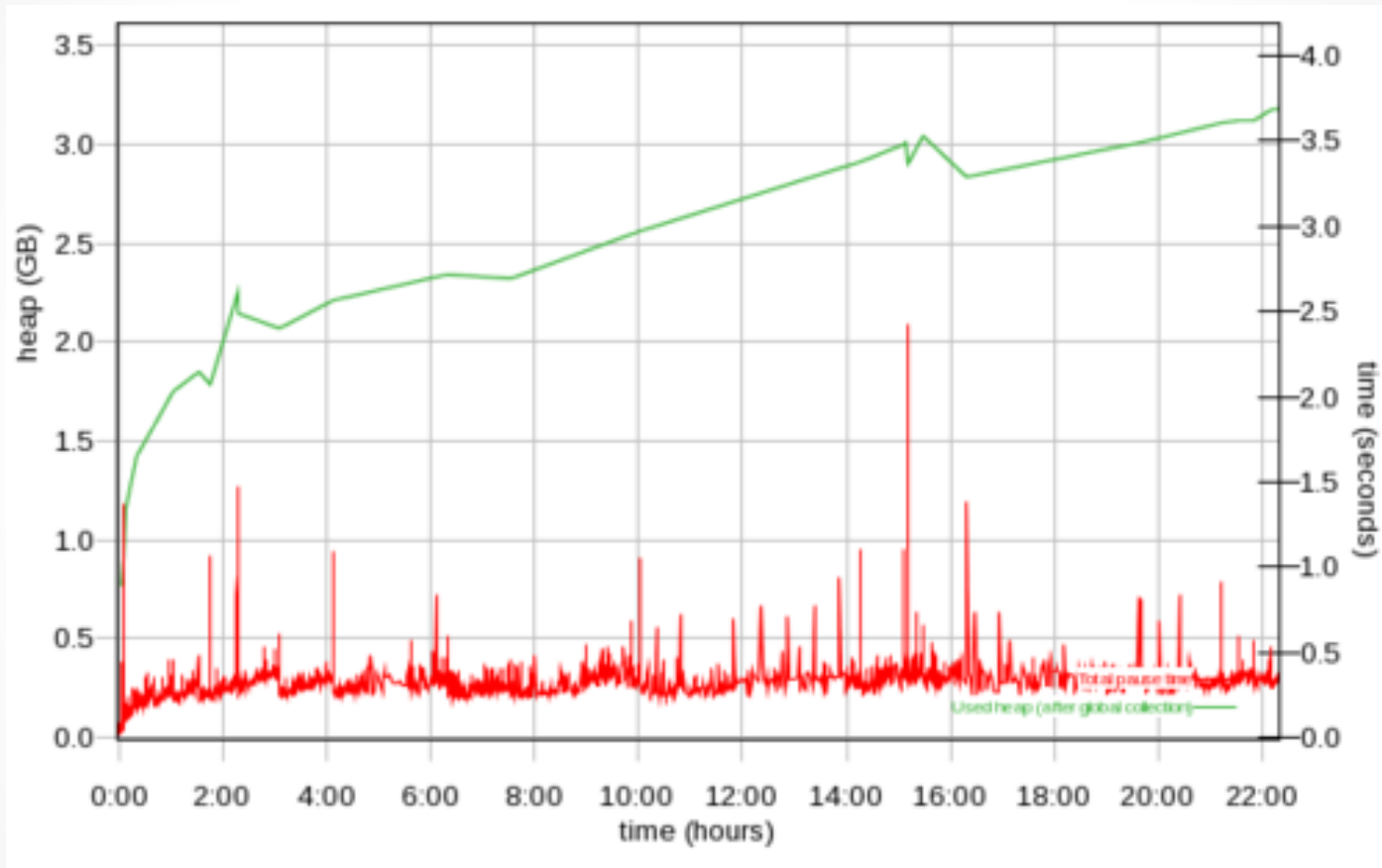
```
DB db = DBMaker.newDirectMemoryDB().transactionDisable().make();  
Map m = db.getTreeMap(«test»);  
for(long counter=0;;counter++)  
{  
    m.put(counter, "");  
    if(counter%1000000==0) System.out.println(""+counter);  
}
```

+ кеширование узлов дерева, прежде чем они  
вставлены

1738 mln

# Memory leak

Типичная ситуация утечки памяти



Сборщик мусора периодически собирает неиспользуемые объекты, но мы видим, что график использования кучи уверенно и верно ползёт вверх.

# Memory leak. Substring

## Строковые операции

Вызов **substring()** у строки, возвращается экземпляр String с лишь изменёнными значениями переменных length и offset — длины и смещения char-последовательности.

При этом, если мы получаем строку длиной 5000 символов и хотим получить её префикс, используя метод **substring()**, то 5000 символов будут продолжать храниться в памяти.

Для систем, которые получают и обрабатывают множество сообщений, это может быть серьёзной проблемой.

Для того, чтобы избежать данную проблему, можно использовать два варианта:

**String prefix = new String(longString.substring(0,5));** //первый вариант

**String prefix = longString.substring(0,5).intern();** //второй вариант

# Memory leak. Object I/O Streams

## **ObjectInputStream** и **ObjectOutputStream**

Классы **ObjectInputStream** и **ObjectOutputStream** хранят ссылки на все объекты, с которыми они работали, чтобы передавать их вместо копий.

Это вызывает утечку памяти при непрерывной использовании (к примеру, при сетевом взаимодействии).

Для решения этой проблемы необходимо периодически вызывать метод **reset()**

# Memory leak. Non-static nested classes

Каждый экземпляр **нестатического** внутреннего класса, который вы используете, хранит ссылку на внешний класс. Это приводит к хранению большого графа объектов, что негативно сказывается на использовании памяти.

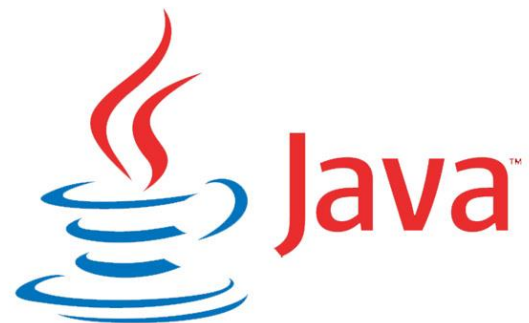
В ситуациях, где явно не нужно использовать ссылку на внешний класс, реализовывайте внутренние классы **статическими**.

# Memory leak. Static

Также частым случаем утечки памяти в Java-приложениях служит неправильное использование `static`. Статическая переменная хранится своим классом, а как следствие, его **загрузчиком** (classloader).

По причине внешнего использования увеличивается шанс, что сборщик мусора **не соберёт данный экземпляр**. Также зачастую в `static`-переменных кэшируется информация или же хранятся состояния, используемые несколькими потоками. Отдельным примером являются статические коллекции.

**Хорошим же тоном при архитектурном проектировании служит полное избегание изменяемых статических объектов — зачастую существует лучшая альтернатива.**



Спасибо за внимание



# Links

<http://habrahabr.ru/post/76481/>  
<http://habrahabr.ru/post/158451/>  
<http://habrahabr.ru/post/159557/>  
<http://habrahabr.ru/post/134102/>  
<http://habrahabr.ru/post/117274/>  
<http://habrahabr.ru/post/136883/>  
<http://habrahabr.ru/post/84165/>  
<http://habrahabr.ru/post/134910/>  
<http://habrahabr.ru/post/136136/>  
<http://habrahabr.ru/post/132500/>  
<http://habrahabr.ru/post/51107/>  
<http://habrahabr.ru/post/142409/>  
<http://habrahabr.ru/post/71704/>  
<http://habrahabr.ru/post/112676/>  
<http://juravskiy.ru/?p=1369>  
<http://www.devclub.eu/2012/12/03/video-nikita-salnikov-java-objects/>  
<http://docs.oracle.com/javase/specs/#22909>  
[http://ru.wikipedia.org/wiki/Слабая\\_ссылка](http://ru.wikipedia.org/wiki/Слабая_ссылка)