


Этот сайт больше не обновляется и не поддерживается. Содержание предоставляется "как есть". Учитывая быстрое развитие технологий, некоторые материалы, код или иллюстрации могут к настоящему моменту устареть.

Изучайте > Технология Java

От Java-кода к Java-куче

Изучение и оптимизация использования памяти приложениями



Крис Бэйли

Опубликовано 29.02.2012

Тема оптимизации использования памяти кодом приложения не нова, тем не менее, нельзя сказать, что она исчерпана. Эта статья содержит краткое рассмотрение принципов использования памяти процессом Java и углубленный анализ использования памяти Java-кодом приложения. В конце статьи предлагаются способы повышения эффективности использования памяти приложением, особенно в части коллекций Java, таких как `HashMap` и `ArrayList`.

Введение: использование памяти процессом Java

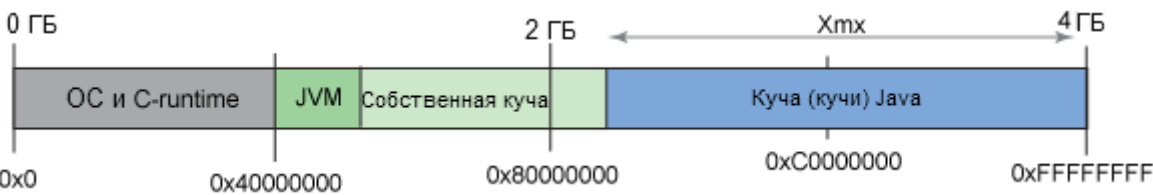
При запуске Java-приложения командой `java` из командной строки или с помощью того или иного промежуточного ПО Java среда исполнения Java создает процесс операционной системы — как если бы вы запустили программу на языке C. На самом деле большинство JVM написаны главным образом на C или C++. Будучи процессом операционной системы, среда исполнения Java сталкивается с теми же ограничениями на использование памяти, что и любой другой процесс: возможностями адресации, предоставляемыми архитектурой, и пространством пользователя, предоставляемым операционной системой.

Возможности адресации памяти, предоставляемые архитектурой, зависят от размерности процессора — например, 32 или 64 бита, или же 31 бит в случае мейнфрейма. Разрядность процессора определяет диапазон емкости памяти, которую он способен адресовать: 32 бита обеспечивает диапазон адресации 2^{32} , то есть 4 294 967 296 битов, или 4 ГБ. Диапазон адресации для 64-разрядного процессора значительно шире: 2^{64} — это 18 446 744 073 709 551 616, или 16 экзабайт.

Часть адресуемого диапазона, представленного архитектурой процессора, используется самой ОС для своего ядра и для среды исполнения C (в JVM, написанных на C или C++). Количество памяти, используемой ОС и средой исполнения C, зависит от операционной системы, но обычно оно значительно: Windows по умолчанию использует 2 ГБ. Остальное адресное пространство — называемое *пространством пользователя* — доступно фактически исполняемым процессам.

В случае Java-приложений пространство пользователя — это память, используемая процессом Java, по существу, состоящая из двух пулов: куч(и) Java и *собственной* (не-Java) кучи. Размером кучи Java управляют параметры кучи Java JVM: значения `-Xms` и `-Xmx` устанавливают соответственно минимальный и максимальный размер кучи Java. Собственная куча — это пространство памяти пользователя, оставшееся после выделения кучи Java максимального заданного размера. На рисунке 1 приведен пример того, как это может выглядеть для 32-разрядного процесса Java.

Рисунок 1. Пример распределения памяти для 32-разрядного процесса Java



как и ядро ОС и среда исполнения C — и что память, используемая JVM, это часть собственной кучи.

Анатомия объекта Java

Когда Java-код использует оператор `new` для создания экземпляра объекта Java, выделяется намного больше данных, чем следовало бы ожидать. Например, соотношение размеров между значением типа `int` и объектом `Integer` — самым малым объектом, который может содержать значение `int`, — обычно составляет 1:4. Накладные расходы приходятся на метаданные, которые JVM использует для описания объекта Java, в данном случае `Integer`.

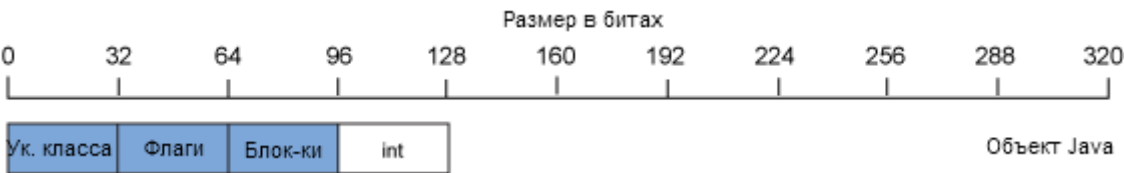
Количество метаданных объекта зависит от версии и поставщика JVM, но обычно включает:

- **Class:** указатель на сведения о классе, который описывает тип объекта. В случае объекта `java.lang.Integer`, например, это указатель на класс `java.lang.Integer`;
- **флаги:** набор флагов, которые описывают состояние объекта, включая хэш-код для объекта, если он есть, и *форму* объекта (то есть является ли объект массивом);
- **Lock:** сведения о синхронизации объекта — то есть синхронизирован ли объект в настоящее время.

За метаданными объекта следуют собственно данные объекта, состоящие из полей, хранящихся в экземпляре объекта. В случае объекта `java.lang.Integer` это одно значение типа `int`.

Таким образом, при создании экземпляра объекта `java.lang.Integer` во время запуска 32-разрядной JVM макет объекта может выглядеть, как показано на рисунке 2.

Рисунок 2. Пример макета объекта `java.lang.Integer` для 32-разрядного процесса Java



Как видно на [рисунке 2](#), для хранения 32 битов данных значения типа `int` используются 128 битов данных, потому что остальная часть этих 128 битов используется метаданными объекта.

Анатомия объекта Java типа массива

Форма и структура объекта типа массива, например массива значений `int`, аналогична стандартному Java-объекту. Основное различие заключается в том, что у объекта типа массива есть дополнительный фрагмент метаданных, указывающий размер массива. Таким образом, метаданные объекта типа массива содержат:

- **Class:** указатель на сведения о классе, который описывает тип объекта. В случае массива полей типа `int` это указатель на класс `int[]`;
- **флаги:** набор флагов, которые описывают состояние объекта, включая хэш-код для объекта, если он есть, и форму объекта (то есть, является ли объект массивом);
- **Lock:** сведения о синхронизации объекта — то есть, синхронизирован ли объект в настоящее время.
- **Size:** размер массива.

На рисунке 3 показан пример макета для объекта массива типа `int`.

Рисунок 3. Пример макета объекта массива значений типа `int` для 32-разрядного процесса Java



На [рисунке 3](#) 32 бита данных значения `int` хранятся в 160 битах данных, потому что остальная часть этих 160 битов используется для метаданных массива. Для примитивов, таких как `byte`, `int` и `long`, массив из одной записи дороже с

Анатомия более сложных структур данных

Для качественного объектно-ориентированного проектирования и программирования полезно использовать *инкапсуляцию* (обеспечивая связующие классы, которые управляют доступом к данным) и *делегирование* (использование вспомогательных объектов для решения задачи). Инкапсуляция и делегирование приводят к тому, что представление большинства структур данных содержит по несколько объектов. Простой пример — объект `java.lang.String`. Данные в объекте `java.lang.String` — это массив символов, инкапсулированный объектом `java.lang.String`, который управляет и контролирует доступ к массиву символов. Макет объекта `java.lang.String` для 32-разрядного процесса Java может выглядеть как на рисунке 4.

Рисунок 4. Пример макета объекта `java.lang.String` для 32-разрядного процесса Java



Как показано на [рисунке 4](#), объект `java.lang.String` — помимо стандартных метаданных объекта — содержит некоторые поля для управления данными типа строки. Как правило, эти поля представляют собой хэш-значение, размер строки, смещение в массиве строковых данных и ссылку на сам массив символов.

Это означает, что для строки из 8 символов (128 битов данных типа `char`) требуются 256 битов данных для массива символов и 224 бита данных для объекта `java.lang.String`, который им управляет, что в общей сложности дает 480 битов (60 байтов) для представления 128 битов (16 байтов) данных. Соотношение накладных расходов составляет 3,75:1.

В общем случае, чем сложнее структура данных, тем больше накладные расходы. Этот вопрос подробнее обсуждается в следующем разделе.

32-разрядные и 64-разрядные объекты Java

Размеры и накладные расходы для объектов из предыдущих примеров относятся к 32-разрядному процессу Java. Как говорилось в разделе [Введение: использование памяти процессом Java](#), 64-разрядный процессор адресует гораздо больше памяти, чем 32-разрядный. В случае 64-разрядного процесса размер некоторых полей данных в Java-объектах — в частности, метаданных объекта и любых полей, которые указывают на другой объект, — также необходимо увеличить до 64 битов. Размеры других типов полей данных — таких как `int`, `byte` или `long` — не меняются. На рисунке 5 показан макет 64-разрядного объекта `Integer` и массива значений `int`.

Рисунок 5. Пример макета объекта `java.lang.Integer` и массива значений `int` для 64-разрядного процесса Java



На [рисунке 5](#) видно, что при 64-разрядных объектах `Integer` для хранения 32 битов поля `int` теперь используются 224 бита данных — коэффициент накладных расходов составляет 7:1. Для хранения записи с 32-разрядным числом `int` в массиве из одного 64-разрядного элемента `int` используются 288 битов данных — накладные расходы составляют 9:1. Следствием этого для реальных приложений является резкое увеличение объема используемой памяти кучи Java приложением, которое ранее работало в 32-разрядной среде исполнения Java, при его переносе в 64-разрядную среду. Как правило, первоначальная куча увеличивается примерно на 70%. Так, Java-приложение, которое в 32-разрядной среде исполнения Java использовало 1-ГБ кучу Java, в 64-разрядной среде обычно использует кучу Java размером 1,7 ГБ.

В таблице 1 приведены размеры полей для объектов и массивов при выполнении приложения в 32- и 64-разрядных режимах.

Таблица 1. Размеры полей объектов для 32-и 64-разрядной среды исполнения Java

Тип поля	Размер поля (битов)			
	Объект		Массив	
	32-разрядный	64-разрядный	32-разрядный	64-разрядный
boolean	32	32	8	8
byte	32	32	8	8
char	32	32	16	16
short	32	32	16	16
int	32	32	32	32
float	32	32	32	32
long	64	64	64	64
double	64	64	64	64
Поля объектов	32	64 (32*)	32	64 (32*)
Метаданные объектов	32	64 (32*)	32	64 (32*)

*Размер полей объектов и данных, используемых для каждой из записей метаданных объекта, можно уменьшить до 32 бит с помощью технологий [сжатых ссылок](#) или [сжатых OOP](#).

Сжатые ссылки и сжатые указатели обычных объектов (OOP)

JVM от IBM и Oracle предоставляют возможность сжатия ссылок на объект с помощью сжатых ссылок (-Xcompressedrefs) и сжатых OOP(-XX:+UseCompressedOops) соответственно. Использование этих возможностей позволяет хранить поля объектов и значения метаданных объекта в 32 битах, вместо 64. Это исключает увеличение используемой памяти кучи Java на 70% при перемещении приложения из 32-разрядной среды исполнения Java в 64-разрядную. Отметим, что эти опции не влияют на использование памяти собственной кучи; для 64-разрядной среды исполнения Java оно остается увеличенным по сравнению с 32-разрядной средой.

Использование памяти коллекциями Java

В большинстве приложений большое количество данных хранится и управляется с помощью стандартных классов Java Collection, обеспечиваемых ядром API Java. Если для приложения важна оптимизация памяти, то полезно понимать, как работает каждая коллекция и какие накладные расходы в отношении памяти с ней связаны. В общем случае, чем выше уровень функциональных возможностей коллекции, тем больше служебной памяти она потребляет — так что использование коллекций тех типов, которые предоставляют больше возможностей, чем требуется, ведет к ненужному перерасходу памяти.

- [HashSet](#)
- [HashMap](#)
- [Hashtable](#)
- [LinkedList](#)
- [ArrayList](#)

За исключением HashSet, этот список упорядочен по убыванию как функциональных возможностей, так и накладных расходов в отношении памяти. (HashSet, будучи оболочкой вокруг объекта HashMap, на самом деле предоставляет меньше функций, чем HashMap, хотя и занимает чуть больше места.)

Коллекции Java: HashSet

HashSet представляет собой реализацию интерфейса Set. В документации API Java Platform SE 6 HashSet описывается следующим образом:

Коллекция, не содержащая повторяющихся элементов. А именно, такие множества не содержат таких пар элементов e1 и e2, для которых e1.equals(e2), и содержат максимум один пустой элемент. Как подразумевает его название, этот интерфейс моделирует математическое понятие множества.

HashSet предоставляет меньше возможностей, чем HashMap, так как не может содержать более одной пустой записи и повторяющиеся записи. Он реализован как оболочка вокруг HashMap, так что объект HashSet управляет тем, что позволено класть в объект HashMap. Дополнительная функция ограничения возможностей HashMap означает, что у множеств HashSet несколько более высокие накладные расходы в отношении памяти.

На рисунке 6 показано распределение и использование памяти HashSet в 32-разрядной среде исполнения Java.

Рисунок 6. Использование и распределение памяти HashSet в 32-разрядной среде Java

Class Name	Shallow Heap	Retained Heap
<Regex>	<Numeric>	<Numeric>
java.util.HashSet @ 0x10a6d908	16	144
java.util.HashMap @ 0x10a6d918	48	128

На [рисунке 6](#) показаны размер в байтах *мелкой кучи* (shallow heap - память, используемая отдельным объектом) и *рамзер удерживаемой кучи* (retained heap — память используемая отдельным объектом и его дочерними объектами) для объекта java.util.HashSet. Размер мелкой кучи составляет 16 байтов, а размер удерживаемой кучи — 144 байта. При создании коллекции HashSet ее *емкость по умолчанию*— количество записей, которые могут быть помещены в коллекцию, — составляет 16 записей. Когда HashSet создается с емкостью по умолчанию и без записей, она занимает 144 байта. Это на 16 байтов больше объема памяти, используемой коллекцией HashMap. В таблице 2 показаны свойства HashSet.

Таблица 2. Свойства HashSet

3

Емкость по умолчанию	16 записей
Размер пустой коллекции	144 байта
Накладные расходы	16 байтов плюс накладные расходы HashMap
Накладные расходы для коллекции в 10К записей	16 байтов плюс накладные расходы HashMap

4

Производительность операций поиска/вставки/удаления	O(1) — время выполнения постоянно, независимо от числа элементов (при отсутствии хэш-коллизий)
---	--

5

HashMap представляет собой реализацию интерфейса Map. В документации API Java Platform SE 6 HashMap описывается следующим образом:

Объект, который ставит значения в соответствие ключам. Отображение не может содержать повторяющиеся ключи; каждому ключу можно поставить в соответствие не более одного значения.

HashMap предоставляет способ хранения пар ключ/значение, используя функцию хеширования для преобразования ключа в индекс коллекции, в которой хранится пара ключ/значение. Это позволяет быстро обращаться к данным. Пустые и повторяющиеся записи разрешены; таким образом, HashMap является упрощением HashSet.

Реализацией HashMap является массив объектов HashMap\$Entry. На рисунке 7 показано использование и распределение памяти HashMap в 32-разрядной среде исполнения Java.

Рисунок 7. Использование и распределение памяти HashMap в 32-разрядной среде исполнения Java.

Class Name	Shallow Heap	Retained Heap
<Regex>	<Numeric>	<Numeric>
java.util.HashMap @ 0x10a6d918	48	128
java.util.HashMap\$Entry[16] @ 0x10a6d948	80	80

6

Как видно из рисунка 7, при создании HashMap результатом является объект HashMap и массив объектов HashMap\$Entry с емкостью по умолчанию в 16 записей. Это дает размер пустой коллекции HashMap в 128 байтов. Любая пара ключ/значение, которая вводится в HashMap, обортывается объектом HashMap\$Entry, который добавляет свои накладные расходы.

Большинство реализаций объектов HashMap\$Entry содержит следующие поля:

- int KeyHash
- Object next
- Object key
- Object value

Объект HashMap\$Entry длиной в 32 байта управляет парами данных ключ/значение, помещенными в коллекцию. Это означает, что общие накладные расходы HashMap для каждой записи состоят из объекта HashMap, записи массива HashMap\$Entry и объекта HashMap\$Entry. Это можно выразить формулой:

Объект HashMap + накладные расходы объекта Array + (количество записей * (запись массива HashMap\$Entry + объект HashMap\$Entry))
Для коллекции HashMap из 10; nbs000 записей накладные расходы только HashMap, массива HashMap\$Entry и объектов HashMap\$Entry составят примерно 360K. Это без учета размера хранящихся ключей и значений.

В таблице 3 приведены свойства коллекции HashMap.

Таблица 3. Свойства HashMap

7

Емкость по умолчанию	16 записей
Размер пустой коллекции	128 байт
Накладные расходы	64 байта плюс 36 байтов на каждую запись
Накладные расходы для коллекции в 10K записей	~360K
Производительность операций поиска/вставки/удаления	O(1) — время выполнения постоянно, независимо от числа элементов (при отсутствии хэш-коллизий)

Hashtable, как и HashMap, представляет собой реализацию интерфейса Map. В документации API Java Platform SE 6 Hashtable описывается следующим образом:

Этот класс реализует объект hashtable, который ставит значения в соответствие ключам. В качестве ключа или значения может использоваться любой непустой объект.

Hashtable очень похож на HashMap, но имеет два ограничения. Он не может принимать значения null ни в записи ключей, ни в записи значений, и это синхронизированная коллекция. Напротив, коллекция HashMap может принимать значения null и не синхронизирована, но ее можно сделать синхронизированной с помощью метода Collections.synchronizedMap().

Реализация Hashtable— также аналогично HashMap— представляет собой массив объектов записей, в данном случае объектов Hashtable\$Entry. На рисунке 8 показано использование и распределение памяти Hashtable в 32-разрядной среде исполнения Java.

Рисунок 8. Использование и распределение памяти Hashtable в 32-разрядной среде исполнения Java

Class Name	Shallow Heap	Retained Heap
<Regex>	<Numeric>	<Numeric>
java.util.Hashtable @ 0x1bae9290	40	104
java.util.Hashtable\$Entry[11] @ 0x1bae92b8	64	64

На рисунке 8 видно, что при создании Hashtable результатом является объект Hashtable, использующий 40 байтов памяти, наряду с массивом объектов Hashtable\$Entry емкостью по умолчанию в 11 записей, что в итоге дает размер пустой коллекции Hashtable 104 байта.

Hashtable\$Entry, по существу, хранит те же данные, что и HashMap:

- int KeyHash
- Object next
- Object key
- Object value

Это означает, что объект Hashtable\$Entry также имеет размер 32 байта на запись ключ/значение в Hashtable, и расчет накладных расходов и размера коллекции в 10K записей для Hashtable (около 360 K) подобен расчету для HashMap.

В таблице 4 показаны свойства коллекции Hashtable.

Таблица 4. Свойства коллекции Hashtable

Емкость по умолчанию	11 записей
Размер пустой коллекции	104 байта
Накладные расходы	56 байтов плюс 36 байтов на каждую запись
Накладные расходы для коллекции в 10K записей	~360K
Производительность операций поиска/вставки/удаления	O(1) — время выполнения постоянно, независимо от числа элементов (при отсутствии хэш-коллизий)

Как видите, Hashtable имеет немного меньшую емкость по умолчанию, чем HashMap (11 записей вместо 16). В основном основные различия — это неспособность Hashtable принимать пустые ключи и значения, а также синхронизация по умолчанию, которая может быть не нужна и уменьшает производительность коллекции.

LinkedList представляет собой реализацию интерфейса List со связным списком. В документации API Java Platform SE 6 LinkedList описывается следующим образом:

Упорядоченная коллекция (также называемая последовательностью). Пользователь данного интерфейса может точно управлять местом расположения каждого элемента в списке. Пользователь может обращаться к элементам по их целочисленному индексу (позиция в списке) и искать элементы в списке. В отличие от множеств, списки обычно допускают повторяющиеся элементы.

Реализация представляет собой связанный список объектов LinkedList\$Entry. На рисунке 9 показано использование и распределение памяти для LinkedList в 32-разрядной среде исполнения Java.

Рисунок 9. Использование и распределение памяти для LinkedList в 32-разрядной среде исполнения Java

Class Name	Shallow Heap	Retained Heap
<Regex>	<Numeric>	<Numeric>
java.util.LinkedList @ 0x11624d50 Thread	24	48
java.util.LinkedList\$Link @ 0x11624d68	24	24

На рисунке 9 видно, что при создании LinkedList результатом является объект LinkedList, использующий 24 байта памяти, а также один объект LinkedList\$Entry, что в сумме дает 48 байт памяти для пустого списка LinkedList.

Одним из преимуществ связанных списков является то, что они имеют точный размер и не меняют его. Емкость по умолчанию равна одной записи и динамически увеличивается или уменьшается по мере добавления или удаления записей. И все же существуют накладные расходы для каждого объекта LinkedList\$Entry со следующими полями данных:

- Object previous
- Object next
- Object value

Но они меньше, чем у объектов HashMap и Hashtable, потому что связанные списки хранят только одну запись, а не пару ключ/значение, и нет необходимости в хранении хэш-значения, поскольку поиск по массиву не используется. К недостаткам можно отнести то, что поиск в связанном списке может выполняться намного медленнее, потому что при поиске нужной записи нужно пройти по всему связанному списку. Для больших связанных списков это может привести к медлительному поиску.

В таблице 5 показаны свойства коллекции LinkedList.

Таблица 5. Свойства LinkedList

Емкость по умолчанию	1 запись
Размер пустой коллекции	48 байтов
Накладные расходы	24 байта плюс 24 байта на каждую запись
Накладные расходы для коллекции в 10K записей	~240K
Производительность операций поиска/вставки/удаления	O(n) — время нарастает линейно в зависимости от количества элементов

Коллекции Java: ArrayList

ArrayList представляет собой реализацию интерфейса List. В документации API Java Platform SE 6 ArrayList описывается следующим образом:

целочисленному индексу (позиция в списке) и искать элементы в списке. В отличие от множеств, списки обычно допускают повторяющиеся элементы.

14

В отличие от `LinkedList`, `ArrayList` реализуется с помощью массива элементов `Object`. На рисунке 10 показано использование и распределение памяти `ArrayList` в 32-разрядной среде исполнения Java.

Рисунок 10. Использование и распределение памяти для `ArrayList` в 32-разрядной среде Java

Class Name	Shallow Heap	Retained Heap
<Regex>	<Numeric>	<Numeric>
java.util.ArrayList @ 0x1fc279e0	32	88
java.lang.Object[10] @ 0x1fc27a00	56	56

Рисунок 10 показывает, что при создании `ArrayList` результатом является объект `ArrayList`, использующий 32 байта памяти, а также массив `Object` с размером по умолчанию 10 записей, что дает 88 байтов памяти для пустого `ArrayList`. Это означает, что `ArrayList` не имеет точного размера, и поэтому его емкость по умолчанию установлена в 10 записей.

В таблице 6 показаны свойства `ArrayList`.

Таблица 6. Свойства `ArrayList`

Емкость по умолчанию	10
Размер пустой коллекции	88 байтов
Накладные расходы	48 байтов плюс 4 байта на каждую запись
Накладные расходы для коллекции в 10K записей	~40K

15

Производительность операций поиска/вставки/удаления

$O(n)$ — время нарастает линейно в зависимости от количества элементов

Другие виды «коллекций»

В дополнение к стандартным коллекциям объект `StringBuffer` тоже можно считать коллекцией в том смысле, что он управляет символьными данными и по своей структуре и возможностям подобен любой другой коллекции. В документации API Java Platform SE 6 `StringBuffer` описывается следующим образом:

Потоково-безопасная, изменяемая последовательность символов... Каждый строковый буфер имеет свою емкость. До тех пор, пока длина последовательности символов, содержащихся в строковом буфере, не превышает его емкости, нет необходимости выделять новый массив внутреннего буфера. При переполнении внутреннего буфера он автоматически увеличивается. Реализация `StringBuffer` представляет собой массив элементов типа `char`. На рисунке 11 показано использование и распределение памяти для `StringBuffer` в 32-разрядной среде исполнения Java.

Рисунок 11. Использование и распределение памяти для `StringBuffer` в 32-разрядной среде Java

Class Name	Shallow Heap	Retained Heap
<Regex>	<Numeric>	<Numeric>
java.lang.StringBuffer @ 0x2898eb0 buffer text	24	72
char[16] @ 0x2898ec8 buffer text\u0000\u0000\u0000\u0000\u0000	48	48

Рисунок 11 показывает, что при создании `StringBuffer` результатом является объект `StringBuffer`, использующий 24 байта памяти, а также массив символов с размером по умолчанию 16, что в итоге дает 72 байта данных для пустого `StringBuffer`.

Как и коллекции, `StringBuffer` имеет емкость по умолчанию и механизм изменения размера. В таблице 7 показаны свойства `StringBuffer`.

Таблица 7. Свойства `StringBuffer`

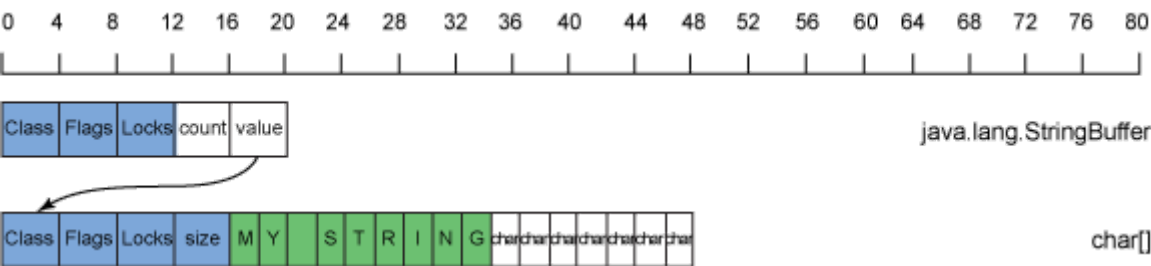
IBM Developer	Изучайте	Разрабатывайте	Подключайтесь
Емкость по умолчанию		16	
Размер пустой коллекции		72 байта	
Накладные расходы		24 байта	
Накладные расходы для коллекции в 10К записей		24 байта	
Производительность операций поиска/вставки/удаления		NA	

Пустое пространство в коллекциях

Накладные расходы различных коллекций с заданным количеством объектов — это еще не все. Измерения из предыдущих примеров предполагают, что коллекции имеют точный размер. Но для большинства коллекций это маловероятно. Большинство коллекций создаются с заданной начальной емкостью, и данные помещаются в эту коллекцию. Это означает, что коллекции обычно имеют емкость, превышающую объем хранящихся в них данных, что приводит к дополнительным накладным расходам.

Рассмотрим пример `StringBuffer`. Его емкость по умолчанию составляет 16 символов при размере 72 байта. Первоначально в этих 72 байтах не хранятся никакие данные. Если поместить что-то в массив символов — например, строку "MY STRING", — то в массиве на 16 символов будет храниться 9 символов. На рисунке 12 показано использование и распределение памяти для `StringBuffer` со строкой "MY STRING" в 32-разрядной среде исполнения Java.

Рисунок 12. Использование памяти массивом `StringBuffer`, содержащим строку "MY STRING" в 32-разрядной среде Java



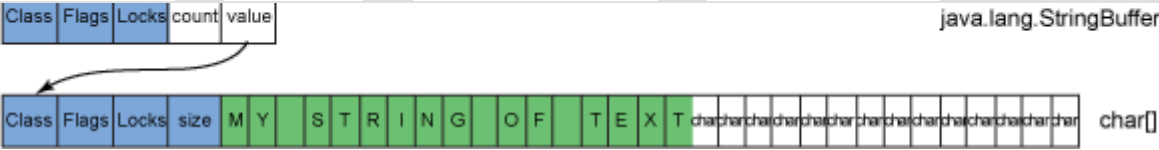
Как видно на [рисунке 12](#), 7 дополнительных символьных записей, имеющихся в массиве, не используются, но потребляют память — в этом случае дополнительные накладные расходы составят 112 байтов. В этой коллекции хранятся 9 записей при емкости 16, что дает *коэффициент заполнения* 0,56. Чем меньше коэффициент заполнения коллекции, тем больше накладные расходы, вызванные лишней емкостью.

Расширение и изменение размера коллекций

Когда коллекция достигает своей емкости и поступает запрос на добавление новой записи, размер коллекции изменяется, и она расширяется для размещения новых записей. Это увеличивает ее емкость, но зачастую уменьшает коэффициент заполнения и создает значительный перерасход памяти.

Алгоритм расширения может быть разным, но обычно емкость коллекции удваивается. Такой подход принят для `StringBuffer`. Если в предыдущем примере с `StringBuffer` добавить в буфер строку " OF TEXT", чтобы получить "MY STRING OF TEXT", вам понадобится расширить коллекцию, потому что в новой коллекции окажется 17 символов при текущей емкости 16. Результирующее использование памяти показано на рисунке 13.

Рисунок 13. Использование памяти массивом `StringBuffer`, содержащим строку "MY STRING OF TEXT", в 32-разрядной среде Java



Теперь как показано на [рисунке 13](#), массив в 32 символа содержит 17 элементов, что дает коэффициент заполнения 0,53. Коэффициент заполнения уменьшился не сильно, но накладные расходы теперь составляют 240 байт избыточной емкости.

В случае коротких строк и коллекций накладные расходы при низких коэффициентах заполнения и избыточной емкости могут показаться не слишком большой проблемой, но при больших размерах они становятся гораздо более очевидным и дорогостоящими. Например, при создании StringBuffer с 16 МБ данных он будет (по умолчанию) использовать массив символов размером до 32 МБ — создавая 16 МБ дополнительных накладных расходов в форме избыточной емкости.

Коллекции Java: резюме

В таблице 8 перечислены свойства коллекций.

Таблица 8. Итоговая информация по свойствам коллекций

Коллекция	Производительность	Емкость по умолчанию	Размер пустой коллекции	Накладные расходы при 10K записях
 HashSet	O(1)	16	144	360K
 HashMap	O(1)	16	128	360K
 Hashtable	O(1)	11	104	360K
 LinkedList	O(n)	1	48	240K
 ArrayList	O(n)	10	88	40K
 StringBuffer	O(1)	16	72	24

Производительность коллекций Hash намного лучше, чем у любой из коллекций List, но при гораздо большей стоимости на каждую запись. По причинам, связанным с временем доступа, при создании больших коллекций (например, для реализации кэша) лучше использовать Hash-коллекции, невзирая на дополнительные накладные расходы.

Для небольших коллекций, когда время доступа не так критично, лучше выбирать списки. Производительность коллекций ArrayList и LinkedList примерно одинакова, но по размерам занимаемой памяти они различаются: по размеру на запись ArrayList гораздо экономичнее, чем LinkedList, но не имеет точного размера. Выбор ArrayList или LinkedList зависит от того, насколько предсказуема длина списка. Если она неизвестна, лучше выбрать LinkedList, потому что в этой коллекции будет меньше пустого пространства. Если размер известен, то ArrayList обеспечит значительно меньшие накладные расходы.

Выбор правильного типа коллекции позволяет установить баланс между производительностью коллекции и размером занимаемой памяти. Кроме того, можно минимизировать объем памяти, правильно соразмерив коллекцию и добившись максимального коэффициента заполнения и минимального неиспользуемого пространства.

Коллекции в действии: PlantsByWebSphere и WebSphere Application Server версии 7

большим — конечно, до тех пор, пока не используется большое количество коллекций.

Таблица 9 иллюстрирует использование объекта-коллекции в составе кучи Java в 206 МБ, когда пример приложения PlantsByWebSphere, входящий в состав WebSphere® Application Server версии 7, выполняет тест с нагрузкой в пять пользователей.

Таблица 9. Использование коллекции приложением PlantsByWebSphere из WebSphere Application Server v7

Тип коллекции	Количество экземпляров	Общие накладные расходы коллекции (МБ)
Hashtable	262 234	26,5
WeakHashMap	19 562	12,6
HashMap	10 600	2,3
ArrayList	9 530	0,3
HashSet	1 551	1,0
Vector	1 271	0,04
LinkedList	1 148	0,1
TreeMap	299	0,03
Итого	306 195	42,9

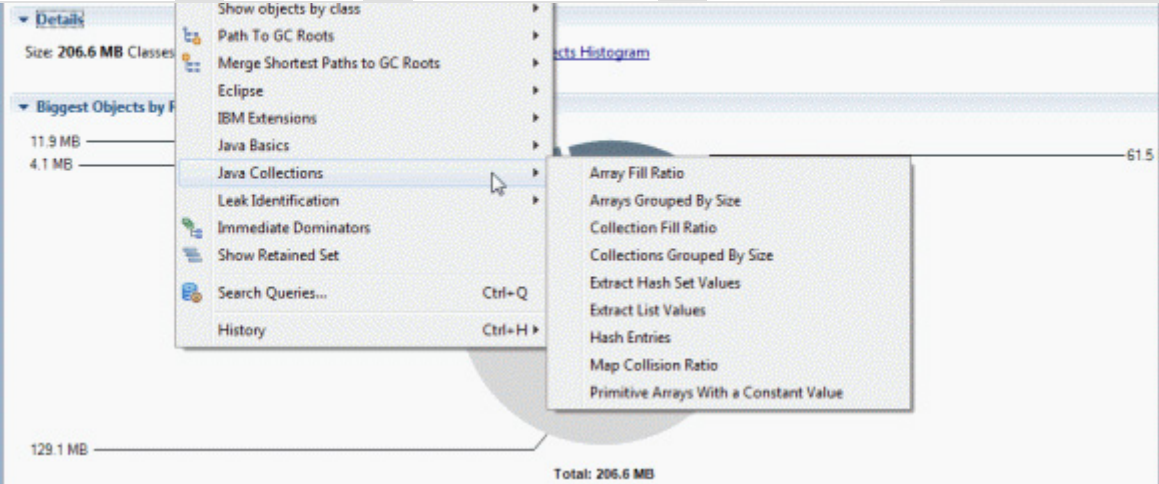
Из [таблицы 9](#) видно, что в приложении используется свыше 300 000 различных коллекций — и что сами эти коллекции, не считая содержащихся в них данных, занимают 42,9 МБ (21%) из 206 МБ используемой кучи Java. Это означает возможность существенной экономии памяти при изменении типов коллекций или более точного определения их размеров.

Поиск низких коэффициентов заполнения с помощью Memory Analyzer

Инструмент IBM Monitoring and Diagnostic Tools for Java - Memory Analyzer (Memory Analyzer), который входит в IBM Support Assistant, позволяет анализировать использование памяти коллекциями Java (см. раздел [Ресурсы](#)). В число его возможностей входит анализ коэффициентов заполнения и размеров коллекций. Это можно использовать для выявления любых коллекций, которые являются кандидатами на оптимизацию.

Возможности анализа коллекций в Memory Analyzer находятся в меню Open Query Browser -> Java Collections, как показано на рисунке 14.

Рисунок 14. Анализ коэффициента заполнения Java-коллекций в Memory Analyzer



Запрос Collection Fill Ratio, выбранный на [рисунке 14](#), — наиболее полезный инструмент выявления коллекций, которые намного больше, чем требуется на данный момент. Можно указать ряд параметров этого запроса, в том числе:

- **objects**: типы объектов (коллекций), которые вас интересуют;
- **segments**: диапазоны коэффициентов заполнения для группирования объектов.

Выполнение запроса, когда параметру objects присвоено значение java.util.Hashtable, а параметру segments — значение "10", дает результат, показанный на [рисунке 15](#).

Рисунок 15. Анализ коэффициента заполнения для объектов Hashtable в Memory Analyzer

Fill Ratio	# Objects	Shallow Heap	Retained Heap
<= 0.00	127,016	5,080,640	>= 9,903,168
<= 0.10	19,773	790,920	>= 4,342,728
<= 0.20	75,967	3,038,680	>= 9,869,153
<= 0.30	100	4,000	>= 87,648
<= 0.40	39,076	1,563,040	>= 10,935,440
<= 0.50	96	3,840	>= 316,986
<= 0.60	94	3,760	>= 562,104
<= 0.70	95	3,800	>= 565,888
<= 0.80	17	680	>= 209,816
Σ Total: 9 entries			
	262,234	10,489,360	

На [рисунке 15](#) видно, что из 262 234 экземпляров java.util.Hashtable 127 016 (48,4%) совершенно пустые и что почти все они содержат лишь небольшое количество записей.

Затем можно определить эти коллекции, выбрав строку таблицы результатов и щелкнув правой кнопкой мыши либо на **list objects -> with incoming reference**, чтобы узнать, какие объекты содержат эти коллекции, либо на **list objects -> with outgoing references**, чтобы увидеть их содержимое. На [рисунке 16](#) показаны результаты поиска пустых коллекций Hashtable с параметром incoming references и развернута пара записей.

Рисунок 16. Анализ на пустые коллекции Hashtable с параметром incoming references в Memory Analyzer

Class Name	Shallow Heap	Retained Heap
<Regex>	<Numeric>	<Numeric>
java.util.Hashtable @ 0x1004c420	40	104
packages javax.management.remote.rmi.NoCallStackClassLoader @ 0x1004c378	104	1,280
java.util.Hashtable @ 0x1004c500	40	104
methodCache javax.management.remote.rmi.NoCallStackClassLoader @ 0x1004c378	104	1,280
java.util.Hashtable @ 0x1004c568	40	104
fieldCache javax.management.remote.rmi.NoCallStackClassLoader @ 0x1004c378	104	1,280
java.util.Hashtable @ 0x1004c5d0	40	104
java.util.Hashtable @ 0x1004cd98	40	104

На [рисунке 16](#) видно, что некоторые пустые коллекции Hashtable относятся к коду javax.management.remote.rmi.NoCallStackClassLoader.

На представлении **Attributes** в левой панели Memory Analyzer видны конкретные сведения о самой коллекции Hashtable, как показано на [рисунке 17](#).

Рисунок 17. Проверка пустой коллекции Hashtable в Memory Analyzer

int	elementCount	0
ref	elementData	java.util.Hashtable\$Entry[11] @ 0x1004c44
float	loadFactor	0.75
int	threshold	8
int	firstSlot	11
int	lastSlot	-1
int	modCount	0

На [рисунке 17](#) видно, что размер Hashtable составляет 11 (размер по умолчанию) и что она совершенно пуста.

В коде `javax.management.remote.rmi.NoCallStackClassLoader` можно оптимизировать использование коллекции следующим образом:

- экономно создавая коллекции Hashtable: если обычно коллекции Hashtable остаются пустыми, то имеет смысл создавать их только при наличии соответствующих данных;
- создавая коллекции Hashtable точного размера: поскольку используется размер по умолчанию, вполне возможно, что можно использовать более точный начальный размер.

Применимы ли эти методы оптимизации, зависит от типичного использования кода и данных, которые обычно хранятся в нем.

Пустые коллекции в примере PlantsByWebSphere

В таблице 10 показан результат анализа коллекций из примера PlantsByWebSphere для выявления пустых коллекций:




Таблица 10. Пустые коллекции в приложении PlantsByWebSphere из WebSphere Application Server v7

Тип коллекции	Количество экземпляров	Пустые экземпляры	% пустых
Hashtable	262 234	127 016	48,4
WeakHashMap	19 562	19 465	99,5
HashMap	10 600	7 599	71,7
ArrayList	9 530	4 588	48,1
HashSet	1 551	866	55,8
Vector	1 271	622	48,9
Итого	304 748	160 156	52,6

[Таблица 10](#) показывает, что в среднем более 50% коллекций пусты, так что оптимизация их использования может принести значительную экономию памяти. Она может применяться на разных уровнях приложения: в коде примера PlantsByWebSphere, в WebSphere Application Server, а также в самих классах коллекций Java.

Между версиями WebSphere Application Server 7 и 8 была проделана некоторая работа по повышению эффективности использования памяти в Java-коллекциях и промежуточном уровне. Например, большой процент накладных расходов экземпляров `java.util.WeakHashMap` вызван тем, что там содержится экземпляр `java.lang.ref.ReferenceQueue` для обработки слабых ссылок. На [рисунке 18](#) показано использование и распределение памяти для WeakHashMap в 32-разрядной среде исполнения Java.

Рисунок 18. Распределение памяти для WeakHashMap в 32-разрядное среде Java

▶  <class> class java.util.WeakHashMap @ 0x1007f748 System Class	8,863	8,863
▶  elementData java.util.WeakHashMap\$Entry[16] @ 0x1fcf6d60	80	80
▶  referenceQueue java.lang.ref.ReferenceQueue @ 0x1fcf6db0	32	560
Σ Total: 3 entries		

На [рисунке 18](#) видно, что объект `ReferenceQueue` отвечает за хранение 560 байтов данных, хотя `WeakHashMap` пуст, и потому `ReferenceQueue` не требуется. В случае примера `PlantsByWebSphere` с 19 465 пустых `WeakHashMap` объекты `ReferenceQueue` добавляют дополнительные 10,9 МБ данных, которые не требуются. В `WebSphere Application Server` версии 8 и выпуске среды исполнения `IBM Java 7` коллекция `WeakHashMap` претерпела некоторую оптимизацию: она содержит `ReferenceQueue`, которая, в свою очередь содержит массив объектов `Reference`. Этот массив был изменен таким образом, чтобы экономно выделять память — только тогда, когда в `ReferenceQueue` добавляются объекты.

Заключение

В любом — и особенно в сложном — приложении присутствует на удивление большое количество коллекций. Это часто создает возможности для экономии иногда значительного объема памяти путем выбора правильных типов коллекций, их размера и экономного создания. Такие решения лучше всего принимать во время проектирования и разработки, но можно также использовать инструмент `Memory Analyzer` для анализа существующих приложений на возможность оптимизации занимаемой памяти.

Ресурсы для скачивания

 [этот контент в PDF](#)

Похожие темы

- Оригинал статьи: [From Java code to Java heap](#).
- [Debugging from dumps](#) (Chris Bailey, Andrew Johnson, Kevin Grigorenko, developerWorks, март 2011 г.): как создавать дампы с помощью `Memory Analyzer` и использовать их для изучения состояния приложения.
- [The Support Authority: Why the Memory Analyzer \(with IBM Extensions\) isn't just for memory leaks anymore](#) (Chris Bailey, Kevin Grigorenko, Dr. Mahesh Rath, developerWorks, март 2011 г.): как использовать `Memory Analyzer` в сочетании с плагином `IBM Extensions for Memory Analyzer` для изучения состояния `WebSphere Application Server` и приложения.
- [5 things you didn't know about ... the Java Collections API, Part 1](#) (Ted Neward, developerWorks, апрель 2010 г.): пять советов по работе с коллекциями. Еще пять советов во [второй части](#).
- [IBM Extensions for Memory Analyzer](#): дополнительные возможности по отладке как универсальных Java-приложений, так и программных продуктов IBM.
- [Eclipse Memory Analyzer Tool \(MAT\)](#): MAT помогает находить утечки памяти и решать проблемы, связанные с повышенным потреблением памяти.

IBM Developer

Помощь

Сообщить о нарушениях

Уведомить сторонние ресурсы

Выбрать язык

English

中文

日本語

Русский

IBM Developer

Изучайте

Разрабатывайте

Подключайтесь

Бразильский

한글

Ленты

Библиотека документов

Рассылка (Английский)

Пробное ПО

Контактная информация

Конфиденциальность

Условия использования

Специальные возможности

Комментарии

Настройки файлов cookie

Russian Federation - Russian

▼