Коллекции Java

Коллекции или контейнеры – это классы позволяющий хранить и производить операции над множеством объектов. Коллекции используются для сохранения, получения, манипулирования данными и обеспечивают агрегацию одних объектов другими.

Во многих языках программирования (Java, C, C++, Pascal) единственным встроенным в язык средством хранения объектов являются массивы. Однако, массивы обладают значительными недостатками, одним из них является конечный размер массива, как следствие необходимость следить за размером массива. Другим - индексная адресация, что не всегда удобно, т.к. ограничивает возможности добавления и удаления объектов. Чтобы избавиться от этих недостатков уже несколько десятилетий программисты используют рекурсивные типы данных, такие как списки и деревья. Стандартный набор коллекций Java служит для избавления программиста от необходимости самостоятельно реализовывать эти типы данных и снабжает его дополнительными возможностями.

До выхода Java 2 v1.5 Tiger коллекции обладали значительным недостатком по сравнению с массивами. Дело в том что до версии 1.5 Java не поддерживал настраиваемые типы данных, что не позволяло создавать типизированные коллекции. С введением в Java 2 v1.5 настраиваемых (generics) типов Collections Framework был переписан и сейчас поддерживает строгую типизацию. Т.е. можно объявить коллекцию, которая сможет хранить объекты только определенного класса и потомков этого класса.

В данной статье будут рассмотрены основные классы и интерфейсы Collections Framework в однозадачной среде. Т.е. в ней не будут рассмотрены вопросы синхронизации коллекций и коллекции специально созданные для работы в многозадачной среде, которые находятся в пакете `java.util.concurrent.*`.

Интерфейсы

Интерфейсы в Collections Framework играют ключевую роль. Все классы коллекций унаследованы от различных интерфейсов, которые определяют поведение коллекции. Иными словами интерфейс определяет «что делает коллекция», а конкретная реализация «как коллекция делает то что определяет интерфейс».

При разработке приложений рекомендуется, там где это возможного, использовать интерфейсы. Такая организация позволяет легко заменять реализацию интерфейса, с целью повышения производительности, например. А так же позволяет разработчику сконцентрироваться на задаче, а не на особенностях реализации.

Как не трудно догадаться, первая задача которая встает перед разработчиком при использовании Collections Framework – это выбор интерфейса. К выбору интерфейса следует подходить исходя из предметной области. Например, если в реальном мире Вы имеете дело с очередью на обслуживание, вполне вероятно, что Вам необходимо использовать интерфейс `Queue<E>`.

Рассмотрим интерфейсы подробнее:

`Iterable<T>`

Интерфейс `Iterable<T>` означает то что данная коллекция может формировать объект-итератор[1], а значит может быть использована в конструкции `for` (в виде `for-each`)[2] .

`Collection<E>`

1 Итераторы используются для последовательного обхода всех элементов коллекции.

2 Использовать конструкцию `for` в виде `for-each` стало возможным начиная с Java 2 v1.5 Tiger

Это базовый интерфейс Collections Framework. В интерфейсе `Collection<E>` определены основные методы для манипуляции с данными, такие как вставка (`add`, `addAll`), удаление (`remove`, `removeAll`, `clear`), поиск (`contains`). Однако, в конкретной реализации часть методов может быть не определена, а их использование, в этом случае, вызовет исключение `UnsupportedOperationException`. Поэтому я бы рекомендовал использовать интерфейс более адекватно описывающий предметную область.

Интерфейсы наследующие интерфейс `Collection<E>`:

`List<E>`

Интерфейс `List<E>` служит для описания списков. Данный интерфейс определяет поведение коллекций, которые служат для хранения упорядоченного набора объектов. Порядок в котором хранятся элементы определяется порядком их добавления в список. Коллекции, реализующие интерфейс `List<E>` могут хранить 1,2 и более копий одного и того же объекта (ссылки на объект). Определена операция получения части списка. А в классе `Collections` определен метод для сортировки списков.

Реализации: `ArrayList<E>`, `LinkedList<E>`.

`Queue<E>`

Реализует FIFO – буфер. Позволяет добавлять и получать объекты. При этом объекты могут быть получены в том порядке, в котором они были добавлены.

Реализации: `ArrayDeque<E>`, `LinkedList<E>`.

`Deque<E>`

Наследует `Queue<E>`. Двунаправленная очередь. Позволяет добавлять и удалять объекты с двух концов. Так же может быть использован в качестве стека.

Реализации: `ArrayDeque<E>`, `LinkedList<E>`.

`Set<E>`

Коллекции, наследующие интерфейс `Set<E>` обеспечивают уникальность хранимых объектов. Иными словами, один и тот же объект не может быть добавлен более одного раза. При использовании данного интерфейса крайне желательно переопределить метод `equals` хранимых объектов, т.к. он используется для определения уникальности объектов[3].

Реализации: `HashSet<E>`, `LinkedHashSet<E>`, `TreeSet<E>`.

`SortedSet<E>`

Наследует `Set<E>`. Реализации этого интерфейса, помимо того что следят за уникальностью хранимых объектов, поддерживают их в порядке возрастания. Отношение порядка между объектами может быть определено, как с помощью метода `compareTo` интерфейса `Comparable<T>`, так и при помощи специального класса наследующего интерфейс `Comparator<T>`.

Реализации: `TreeSet<E>`.

Отображения:

`Map<K,V>`

Класс `Map<K,V>` используется для отображения каждого элемента из одного множества объектов (ключей) на другое (значений). При этом каждому элементу из

3 Не всегда. Например в коллекции `TreeSet` для определения уникальности используется метод `compareTo`. Большое спасибо Евгению Матюшкину aka Skipy (http://www.skipy.ru/).

множества ключей ставится в соответствие 1 элемент из множества значений. В то же время одному элементу из множества значений может соответствовать 1, 2 и более элементов из множества ключей. Интерфейс `Map<K,V>` описывает функциональность ассоциативных массивов.

Реализации: `HashMap<K,V>`, `LinkedHashMap<K,V>`, `TreeMap<K,V>`, `WeakHashMap<K,V>`.

`SortedMap<K,V>`

Наследует `Map<K,V>`. Реализации этого интерфейса обеспечивают хранение элементов множества ключей в порядке возрастания (см. `SortedSet<E>`).

Реализации: `TreeMap<K,V>`.

Реализации

Как видно Collections API обеспечивает богатый выбор интерфейсов, практически на любой случай. Однако эти интерфейсы всего лишь описание того что класс коллекции должен делать. Интерфейсы не говорят о том как тот или иной класс коллекции реализует свой интерфейс. Более того, различные реализации могут накладывать дополнительные ограничения.

Общие вопросы хранения данных

Для начала разберемся каким образом классы Collections API хранят данные. Если вы посмотрите на список реализаций интерфейса `List<E>`, например, то увидите десять его реализаций[4]. Возникает резонный вопрос: неужели существует 10 способов хранения списков? Вовсе нет. Все способы хранения данных в Collections Framework можно свести трем основным: массивы, связанные списки, бинарные деревья. Рассмотрим их подробнее.

Массивы – один из старейших способов хранения данных. Суть в том что данные помещаются последовательно в специально отведенную для этого область памяти. Данные в массиве должны быть одного типа, а значит и размера. Это позволяет легко узнать адрес в памяти по которому хранится нужные элемент, а следовательно и время обращения к произвольному элементу массива постоянное[5]. Напомню, что в массивах Java объекты не хранятся, хранятся лишь указатели на объекты. Контейнеры на основе массивов самостоятельно следят за размером массива. Когда массив исчерпывается создается новый, как правило размер нового массива в два раба больше чем старого. Контейнеры обеспечивают дополнительные функции поиска и доступа к элементам массива, например создание итератора. Так же на основе массива реализованы хэш-таблицы, о которых мы поговорим позже.

Контейнеры на основе массивов: `ArrayList<E>`, `ArrayDeque<E>`.

Связанные списки представляют собой цепочку из объектов ссылающихся друг на друга. Рассмотри звено[6] такой цепочки из реализации `LinkedList<E>`:

```
private static class Entry<E> {
    E element;
    Entry<E> next;
    Entry<E> previous;
```

4 В версии Java 2 v1.6 Mustang

5 На самом деле не всегда. Если массив очень большой, то скорость доступа к массиву будет зависеть от того попадает ли искомый элемент в кэш, находится ли он в реальной памяти ЭВМ или страница на которой он находится выгружена на жесткий диск.

6 Обычно звено называют элементом списка. Здесь использовано слово звено с тем чтобы не вносить путаницу, т.к. элементом будет называться объект хранящийся в списке.

```
        ...
    }
```

Как видно, одно звено содержит ссылки на предыдущее и следующее звенья, а так же на объект, хранящийся в списке. Реализация `LinkedList<E>` представляет собой замкнутый двунаправленный список. Скорость доступа к произвольному элементу будет зависеть от его положения относительно головы списка (того звена на которое ссылается объект `LinkedList<E>`) и в среднем будет пропорциональна размеру списка. Однако, скорость последовательного доступа ко всем элементам списка посредством итератора не будет зависеть от положения элементов в списке. Отсюда можно заключить, что **контейнеры на основе связанных списков не стоит использовать там где необходимо часто обращаться к произвольному элементу**.

Контейнеры на основе связанных списков: `LinkedList<E>`.

Бинарные деревья служат для хранения и поиска упорядоченных объектов. Это значит, что для хранимых объектов необходимо определить отношение порядка при помощи метода `compareTo` и наследования от интерфейса `Comparable<T>` или класса реализующего интерфейс `Comparator<T>`. Скорость доступа к произвольному объекту в таких деревьях пропорциональна логарифму размера контейнера.

Контейнеры на основе бинарных деревьев: `TreeSet<E>`, `TreeMap<K,V>`.

Хэш-таблицы – контейнеры на основе массивов в которых для поиска элемента в массиве используется не индекс элемента в массиве, а его хэш-функция. Как известно для любого объекта в Java реализована хэш-функция и рекомендуется переопределять ее всегда, а для объектов-сущностей в обязательном порядке. В хэш-таблицах индекс определяется на основе хэш-функции, а в нужной позиции массива хранится указатель на связанный список элементов у которых хэш-функции совпадают. Как не трудно догадаться скорость доступа будет меньше тогда, когда связанные списки хэш-таблицы будут как можно короче, иными словами, когда хэш-функции различных объектов не будут совпадать.

Контейнеры на основе хэш-таблиц: `HashSet<E>`, `HashMap<K,V>`, `WeakHashMap<K,V>`.