Dynamické datové struktury (kontejnery)

Tomáš Pitner, upravil Marek Šabo

Kontejnery

Co jsou kontejnery?

- *dynamické datové struktury* vhodné k ukládání proměnného počtu objektů (přesněji odkazů na objekty)
- kontejnery jsou datové struktury v operační paměti, nejsou automaticky ukládané do trvalé paměti (na disk)
- podle povahy mohou mít různé specifické vlastnosti

Hlavní typy kontejnerů

Seznam (List)
: každý prvek v nich uložený má svou *pozici* (číselný index),

  - podobně jako u pole, ale prvků lze uložit potenciálně neomezený počet
  - počet se může za běhu dynamicky *měnit* (i snižovat)

Množiny (Set)
: prvek lze do množiny vložit nejvýš *jedenkrát*

  - odpovídá matematické představě množiny,
  - při porovnávání rozhoduje rovnost podle výsledku volání equals

Asociativní pole (Map)
: v kontejneru jsou *dvojice* (klíč, hodnota),

  - při znalosti klíče lze v kontejneru rychle najít hodnotu,
  - klíč v mapě přibližně odpovídá databázovému klíči.

K čemu slouží

- Kontejnery slouží k ukládání odkazů na *objekty*, ne přímo hodnot primitivních typů.
- Při ukládání hodnot jako jsou čísla, booleovské hodnoty, znaky apod. se de facto ukládají jejich objektové protějšky, tzn. např. Integer, Char, Boolean, Double apod.
- V Javě byly dříve kontejnery koncipovány jako *beztypové*, to už ale od verze Java 1.5 (tedy Java 5) neplatí!
- Od Javy 5 mají tzv. *typové parametry* vyznačené ve špičatých závorkách (např. List<Person>), jimiž určujeme, jaké položky se do kontejneru smějí dostat.

Proč vůbec

- Kontejnery jsou dynamickými alternativami k poli a mají daleko širší použití.

- Slouží k uchování proměnného počtu objektů.
- Počet prvků se v průběhu existence kontejneru může měnit.
- Oproti polím nabízejí časově efektivnější algoritmy přístupu k prvkům — množiny, mapy.

Co použít

- Většinou se používají kontejnery hotové, vestavěné, tj. ty, jež jsou součástí Java Core API,
- tyto vestavěné kontejnerové třídy jsou definovány v balíku java.util,
- je možné vytvořit si vlastní implementace, obvykle ale zachovávající/implementující standardní rozhraní.

Základní kategorie

Kategorie jsou dány tím, které rozhraní příslušný kontejner implementuje. Základní jsou:

Seznam (List<E>)
: lineární struktura, každý prvek má svůj číselný index (pozici)

Množina (Set<E>)
: struktura bez duplicitních hodnot a obecně také bez uspořádání, umožňuje rychlé dotazování na přítomnost prvku

Asociativní pole, mapa (Map<K,V>)
: struktura uchovávající dvojice (klíč→hodnota), rychlý přístup přes klíč

Kolekce

- *Kolekce* jsou kontejnery implementující rozhraní Collection
- Rozhraní kolekce popisuje velmi obecný kontejner, disponující operacemi: *přidávání, rušení prvku, získání iterátoru, zjišťování prázdnosti* atd.
- Mezi kolekce patří mimo mapy všechny ostatní vestavěné kontejnery List, Set.
- Prvky kolekce nemusí mít svou pozici danou indexem, viz např. Set.

Kontejnery — rozhraní, nepovinné metody

- Funkcionalita vestavěných kontejnerů je obvykle předepsána výhradně *rozhraním*, jenž implementují.
- Rozhraní vestavěných kontejnerů připouštějí, že některé metody jsou *nepovinné*, třídy jej nemusí implementovat.
- Fakticky to v takovém případě vypadá tak, že metoda tam sice je, jinak by to typově nesouhlasilo, ale nelze ji použít, protože volání obvykle *vyhodí výjimku*.

- V praxi se totiž někdy nehodí implementovat typicky zápisové operace, protože některé kontejnery chceme mít read-only.

Iterátory

- *Iterátory* jsou prostředkem, jak sekvenčně procházet prvky kolekce buďto:
- v *neurčeném pořadí* nebo
- v *uspořádání* (u uspořádaných kolekcí)
- Každý iterátor musí implementovat velmi jednoduché rozhraní Iterator<E> se třemi metodami:

```
boolean hasNext()
E next()
void remove()
```

"Hromadné" operace nad kolekcemi

- Některé operace je vhodné provádět nikoli postupně po prvcích, ale jedním voláním vhodné metody:

boolean c1.addAll(c2)
: přidá do c1 všechny prvky z c2

boolean c1.containsAll(c2)
: vrátí true, právě když c1 obsahuje všechny prvky z c2, tzn. když c1 je její nadmnožinou

boolean c1.removeAll(c2)
: odstraní z c1 všechny prvky, které jsou současně v c2.

boolean c1.removeIf(Predicate filter)
: odstraní z kolekce všechny prvky, které splňují predikát (podmínku) filter [až od Java 8]

boolean c1.retainAll(c2)
: zachová v c1 jen ty prvky, které jsou současně v c2.

Seznamy

- *Seznamy* jsou lineární struktury,
- implementují rozhraní List, což je rozšíření Collection.
- Prvky lze adresovat celočíselným nezáporným indexem (typu int).
- Poskytují možnost získat dopředný i zpětný *iterátor*.
- Lze pracovat i s *podseznamy*.

Implementace seznamu: ArrayList

ArrayList

- nejpoužívanější implementace seznamu
- využívá vnitřně *pole* pro uchování prvků
- rychlé operace přístupu k prvkům dle indexu
- přičemž o něco pomalejší jsou operace přidávání a odebírání prvků blíže k začátku seznamu (pole, v němž je seznam, se musí realokovat)

Implementace seznamu: LinkedList

LinkedList

- druhá nejpoužívanější implementace seznamu
- využívá vnitřně *zřetězený seznam* pro uchování prvků
- pomalejší operace přístupu k prvkům dle indexu "uvnitř" seznamu
- rychlejší operace přidávání a odebírání prvků na začátku a na konci, resp. blízko nich

Výkonnostní porovnání seznamů

- Výsledek spuštění dema CollectionsDemo.java

```
List implemented as java.util.ArrayList test done:
   add 100000 elements took 12 ms
   remove all elements from last to first took 5 ms
   add at 0 of 100000 elements took 1025 ms
   remove all elements from 0 took 1014 ms
   add at random position of 100000 elements took 483 ms
   remove all elements at random position took 462 ms
List implemented as java.util.LinkedList test done:
   add 100000 elements took 8 ms
   remove all elements from last to first took 9 ms
   add at 0 of 100000 elements took 18 ms
   remove all elements from 0 took 10 ms
   add at random position of 100000 elements took 34504 ms
   remove all elements at random position took 36867 ms
```

Příklad použití seznamu

```
    // declaring and creating list
List<String> ls = new ArrayList<>();
// using method add
ls.add("Ahoj");
ls.add("Cheers");
ls.add("Nazdar");
// using method get
System.out.println(ls.get(0));
// using "add" at specified index
ls.add(0, "Bye");
System.out.println(ls.get(0));
System.out.println("Whole list:");
// using index
for(int i = 0; i < ls.size(); i++) {
   System.out.println(i + ". " + ls.get(i));
}
System.out.println("Whole list without indices:");
// using for-each
for(String s: ls) {
   System.out.println(s);
}
```

Další lineární struktury

Z datových struktur máme v Javě ještě např.:

zásobník
: třída Stack, struktura "LIFO" s operacemi

  - push — vložení na vrchol zásobníku
  - pop — odebrání z vrcholu zásobníku
  - peek — přečtení (neodebrání) z vrcholu zásobníku

fronta
: třída Queue, struktura "FIFO" s operacemi

  - add — přidání prvku do fronty
  - remove — vybrání prvku z fronty
  - element — přečtení (neodebrání) prvku z fronty
  - fronta může případně být *prioritní* (PriorityQueue)

oboustranná fronta
: třída Deque (čteme "deck")

  - slučuje vlastnosti zásobníku a fronty

- nabízí operace příslušné oběma typům

Množiny

- *Množiny* jsou struktury standardně bez uspořádání prvků (ale existují i uspořádané, viz dále),
- implementují rozhraní Set (což je rozšíření Collection).
- Cílem množin je mít možnost rychle (se složitostí *O(log(n))*) provádět atomické operace:
  - vkládání prvku (add)
  - odebírání prvku (remove)
  - dotaz na přítomnost prvku (contains)

Množiny — implementace

Standardní implementace množiny:

hašovací tabulka
: třída HashSet,

  - potenciálně rychlejší (ideálně konstantní, tj. sub-logaritmická složitost),
  - ale neumožňuje uspořádání hodnot

vyhledávací strom
: konkrétně *černobílý strom* (Red-Black Tree),

  - třída TreeSet,
  - uspořádané hodnoty,
  - s garantovanou logaritmickou složitostí

Uspořádané množiny

- Výše uvedená vestavěná implementace TreeSet.
- Implementují rozhraní SortedSet
- Jednotlivé prvky lze tedy iterátorem procházet v přesně definovaném pořadí — uspořádání podle *hodnot prvků*.
- černobílé stromy (Red-Black Trees)
- Uspořádání je dáno buďto:
  - standardním chováním metody compareTo vkládaných objektů — pokud implementují rozhraní Comparable
  - nebo je možné uspořádání definovat pomocí tzv. *komparátoru* (objektu impl. rozhraní Comparator) poskytnutých při vytvoření množiny.

Výkonnostní srovnání množin

- Výsledek spuštění dema CollectionsDemo.java

```
Set implemented as java.util.HashSet test done:
   add 100000 elements took 27 ms
   remove all elements from 100000 to 0 took 14 ms
   add 100000 elements took 9 ms
   remove all elements from 0 took 18 ms
   add 100000 random elements took 30 ms
   remove 100000 random elements took 17 ms
Set implemented as java.util.TreeSet test done:
   add 100000 elements took 67 ms
   remove all elements from 100000 to 0 took 50 ms
   add 100000 elements took 58 ms
   remove all elements from 0 took 41 ms
   add 100000 random elements took 84 ms
   remove 100000 random elements took 68 ms
```

Comparable

- Jednoduché rozhraní Comparable slouží k definování uspořádání na třídě objektů.
- Předepisuje jedinou metodu int compareTo(T t)
- Implementuje-li třída toto rozhraní, znamená to, že její objekty jsou vzájemně *uspořádané*.
- Lze tedy o nich říci, který je ve struktuře umístěn dříve a který později pomocí o1.compareTo(o2), která vrací celé číslo, kde rozhoduje jeho znaménko.
- Toto rozhraní definuje tzv. *přirozené uspořádání* (natural ordering).
- Využívá se zejména u uspořádaných kontejnerů (množin a asociativních polí).
- Chování by mělo být konzistentní s equals, tzn. pro si rovné objekty by compareTo měla vrátit 0.

Comparator

- Jednoduché rozhraní Comparator slouží k definování uspořádání na třídě objektů *zvnějšku*, tzn. uspořádání pomocí objektu jiné třídy.
- Předepisuje jedinou metodu int compare(T o1, T o2)
- Implementuje-li třída toto rozhraní, znamená to, že její objekty umějí definovat *uspořádání* na objektech typu T obdobně jako Comparable vracením celého čísla se znaménkem.
- Toto rozhraní funguje jako alternativa tam, kde nevyhovuje přirozené uspořádání (Comparable).
- Využívá se zejména u uspořádaných kontejnerů (množin a asociativních polí), do nichž se dá komparátor nastavit při konstrukci těchto kontejnerů.

Cyklus *for-each*

- Je rozšířenou syntaxí cyklu for.
- Umožňuje procházení všech prvků polí a dalších iterovatelných struktur, např. seznamů a množin.

Příklad:

```
Set<String> strings = ...
for(String s: strings) {
   System.out.println(s);
}
```

Mapy

- *Mapy* (*asociativní pole*) fungují v podstatě na stejných principech a požadavcích jako Set:
- Ukládají ovšem dvojice (klíč, hodnota) a umožnují rychlé vyhledání dvojice podle hodnoty klíče.
- Základními metodami jsou:
  - dotazy na přítomnost klíče v mapě (containsKey),
  - výběr hodnoty odpovídající zadanému klíči (get),
  - možnost získat zvlášť *množiny klíčů*, *hodnot* nebo *dvojic* (klíč, hodnota).
  - možnost iterace po těchto dvojicích (forEach)

Mapy — implementace

Mapy mají podobné implementace jako množiny, tj.:

hašovací tabulky
: třída HashMap

  - potenciálně rychlejší,
  - ale neuspořádané klíče

černobílé stromy
: třída TreeMap

  - uspořádané klíče,
  - s garantovanou logaritmickou složitostí

Mapy — složitosti

- Složitost základních operací (put, remove, containsKey):
  - Mapy implementované jako stromy mají nejvýš *logaritmickou* složitost základních operací.
  - U map implementovaných hašovací tabulkou složitost v praxi závisí na kvalitě hašovací funkce (metody hashCode) na ukládáných objektech,
  - teoreticky se blíží složitosti *konstantní*.

Uspořádané mapy

- Implementují rozhraní SortedMap
- Dvojice (klíč, hodnota) jsou v nich *uspořádané podle hodnot klíče*.
- Existuje vestavěná implementace třídou TreeMap
- Uspořádání lze ovlivnit naprosto stejným postupem jako u uspořádané množiny.

Získání nemodifikovatelných kontejnerů

- Kontejnery jsou standardně modifikovatelné (read/write).
- Nemodifikovatelné kontejnery se často používají při vracení hodnot z metod.
- Dají se získat pomocí volání příslušné metody třídy Collections.

Souběžný přístup

- Moderní kontejnery jsou *nesynchronizované*, nepřipouštějí souběžný přístup z více vláken.
- Standardní, nesynchronizovaný, kontejner lze však *zabalit synchronizovanou obálkou*.

Pomocná třída Collections

- Java Core API nabízí třídu Collections.
- Je to tzv. *utility class*, nabízí jen statické metody a proměnné, nelze od ní vytvářet instance.
- Nabízí škálu užitečných metod pro práci s kontejnery, jako např.:
  - binární vyhledávání v kontejneru
  - vracení nemodifikovatelných kopií
  - rychlé vracení prázdných kontejnerů
  - agregační funkce (maximální, mininmální prvek)
  - vyplňování seznamu
  - obracení (reverse) nebo prohazování (shuffle) pořadí
  - uspořádání

Kontejnery a výjimky

- Při práci s kontejnery může vzniknout řada *výjimek*, např. IllegalStateException apod.
- Většina má charakter výjimek *běhových*, tudíž není povinností je odchytávat, pokud věříme, že nevzniknou.

Starší typy kontejnerů

- Existují tyto starší typy kontejnerů (za → uvádíme náhradu):
  - Hashtable → HashMap, HashSet (podle účelu)
  - Vector → List
  - Stack → List nebo lépe Queue či Deque

Enumeration

- Roli iterátoru plnil dříve *výčet* (Enumeration) se dvěma metodami:
  - boolean hasMoreElements()
  - Object nextElement()

Srovnání implementací kontejnerů

Seznamy

- na bázi pole (ArrayList) - rychlý přímý přístup (přes index)
- na bázi lineárního zřetězeného seznamu (LinkedList) — rychlý sekvenční přístup (přes iterátor)
- téměř vždy se používá ArrayList — stejně rychlý a paměťově efektivnější

Množiny a mapy

- na bázi hašovacích tabulek (HashMap, HashSet) — rychlejší, ale neuspořádané (lze získat iterátor procházející klíče uspořádaně)
- na bázi vyhledávacích stromů (TreeMap, TreeSet) - pomalejší, ale uspořádané
- spojení výhod obou — LinkedHashSet, LinkedHashMap

Odkazy

- Demo efektivity práce kontejnerů CollectionsDemo
- Velmi podrobné a kvalitní seznámení s kontejnery najdete na Oracle Trail: Collections