La méthode hashCode()

Philippe PRADOS

pp@philippe.prados.name

Préservez l'environnement,

n'imprimez pas ce document

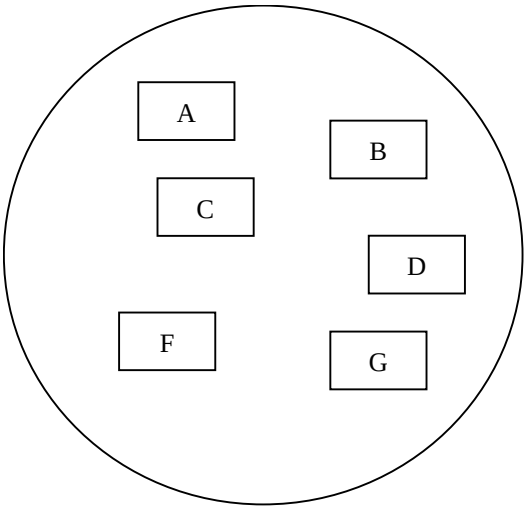
TABLE DES MATIERES

Avant-propos

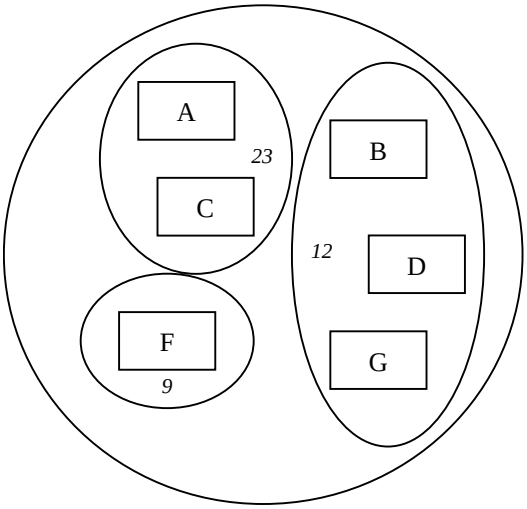
Comment rédiger la méthode `hashCode()` ? Ce document propose une démarche pour cela.

1. A QUOI SERT LA METHODE HASHCODE() ?

Pour retrouver rapidement une instance dans un ensemble, l'approche naïve consiste à parcourir toutes les instances de l'ensemble et d'appeler la méthode `equals()` sur chacun d'un. Cela prend un temps proportionnel au nombre d'élément de l'ensemble.



Pour améliorer cet algorithme, on effectue un calcul sur les instances (hash) et on classe les instances dans des sous-ensembles dépendant du résultat du calcul. Il est plus rapide de comparer les valeurs de hash que de comparer les instances.



Ainsi, pour retrouver un objet, on calcule une valeur sur l'instance à rechercher. On peut alors limiter la recherche à un sous-ensemble. Il faut rédiger l'algorithme de calcul de telle manière que les sous-ensembles aient raisonnablement la même taille. Il faut éviter d'avoir certains sous-ensembles très gros et d'autre très petit.

La méthode `hashCode()` sert à retourner un résumé d'une instance. Elle prend en entrée une donnée de longueur variable et produit un résultat de longueur fixe. Un calcul est effectué sur les valeurs des attributs et un entier est retourné. Plusieurs instances ayant des valeurs différentes peuvent avoir la même valeur de hash, mais deux instances ayants les mêmes valeurs pour les attributs doivent retourner la même valeur. Toutes classes proposant la méthode `hashCode()` doit également proposer la méthode `equals()`. La méthode `hashCode()` sert à optimiser l'utilisation de la méthode `equals()`.

```
if ((x.hashCode()==y.hashCode()) &&
    (x.equals(y))
   )
{ ...
}
```

Si les valeurs de hash de `x` et `y` ne sont pas identiques, il n'est pas nécessaire d'appeler la méthode `equals()`.

La classe `Hashtable` utilise cet algorithme pour gérer les clefs des dictionnaires. Une instance servant de clef à un `HashTable` doit retourner une valeur de hash pertinente. Une `Hashtable` choisie arbitrairement un nombre de sous-ensemble. Elle répartit les instances en calculant le reste de la division de la valeur de hash par le nombre de sous-ensemble. Si le calcul de `hashCode()` retourne toujours la

même valeur, le dictionnaire fonctionne mais ne bénéficie pas de l'optimisation. Toutes les instances sont dans le même sous-ensemble. Parfois, l'algorithme décide d'augmenter le nombre de sous-ensemble pour obtenir une meilleure répartition. Dans ce cas, toutes les instances sont redistribuées.

Pour augmenter l'efficacité de la répartition en plusieurs sous-ensembles, la classe `Hashtable` choisit un nombre premier de sous-ensembles (cent un par défaut).

Attention, il ne faut pas modifier les instances servant de clefs à un dictionnaire si cela entraîne une modification de la valeur de hash code. Le dictionnaire n'en étant pas averti, l'instance n'est pas reclassée dans le bon sous-ensemble. Elle ne sera plus trouvée. Il faut alors appeler la méthode `rehash()` pour rétablir le classement des sous-ensembles.

2. Comment rediger la methode hashCode() ?

Pour rédiger la méthode `hashCode()`, il faut tenir compte de deux objectifs contradictoires. Il faut que le calcul soit rapide et il faut éviter au maximum les collisions entre deux instances.

La méthode par défaut `Object.hashCode()` retourne une valeur déduite de l'adresse mémoire de l'instance. Cela ne permet pas de comparer deux instances différentes ayants les mêmes valeurs dans tous leurs attributs. Vous pouvez obtenir le même résultat en appelant la méthode `System.identityHashCode(Object)`.

Ce choix n'est généralement pas efficace pour les classes proposant la méthode `equals()`. Il faut dans ce cas revoir la méthode `hashCode()`.

De plus, l'algorithme utilisé par cette méthode n'est pas spécifié entre les machines virtuelles. Cela veut dire que le résultat ne sera pas identique sur deux machines virtuelles différentes. En règle générale, il ne faut pas utiliser ou combiner une valeur de hash calculé par la méthode `Object.hashCode()`.

Les classes de l'API java utilisent des algorithmes spécifiques pour calculer la valeur de hash.

- La classe `String` utilise un algorithme dépendant de la taille de la chaîne. Si la chaîne à une taille inférieure à seize, la méthode calcule la somme des caractères multipliés par trente-sept (`h=h*37+buf[i]`). Si la chaîne est d'une taille supérieure, la méthode additionne un caractère tous les huit et multiplie le résultat par trente-huit. (`h=h*38+buf[i*8]`).
- La classe `java.util.Date` retourne `ht ^ (int) (ht >> 32)` déduit du nombre de seconde que représente la date.
- La classe `java.awt.Point` retourne l'expression `x ^ (y*31)` déduite de ces deux attributs.

2.1 Calculer un hash code pour les types primitifs

Pour calculer une valeur de hash à partir des types primitifs il y a plusieurs solutions. Voici un tableau de proposition.

`boolean x`	`return x ? 0x55555555 : 0x2AAAAAAA;`
`byte x`	`int h=x & 0x7F;` `return h | (h << 8) | (h << 16) | (h << 24);`
`char x`	`return x | (x << 16);`
`short x`	`int h=x & 0x7FFF;` `return h | (h << 16);`
`int x`	`return x;`
`long x`	`return (int)(x ^ (int)(x >> 32));`
`float x`	`return Float.floatToIntBits(x);`
`double x`	`long l=Double. doubleToLongBits(x);` `return (int)(l ^ (int)(l >> 32));`

Les algorithmes à utiliser pour calculer une valeur de hash dépend du contexte. Par exemple, si vous utilisez une URL comme clé à un dictionnaire, l'algorithme de hash aura du mal à séparer les instances. En effet, la méthode `hashCode()` de la classe `String` n'est pas adaptée à ce type de chaîne. Elles sont construites sous le même format (`http://www.quelquechose.fr/unePage.html`). Il peut être préférable d'adapter l'algorithme à la situation pour le calculer la valeur que sur le nom de machine et sur le nom de la page.

Pour calculer une valeur de hash pour les tableaux de types primitifs, il faut combiner les valeurs de hash de chacun des éléments.

2.2 Sélectionner les attributs à combiner

Il est souvent nécessaire de combiner des valeurs de hash pour calculer une valeur résultante. Il faut sélectionner les attributs pertinents avant de les combiner. Cela dépend de la probabilité de collision des attributs les uns par rapports aux autres.

Par exemple, on peut combiner le nom d'une personne avec la première lettre de son prénom. Cela est suffisant pour améliorer les algorithmes de recherche. Il n'est pas nécessaire de combiner avec l'adresse de la personne. En effet, il y a généralement très peu d'homonyme habitant à la même adresse.

Il faut identifier les attributs caractérisant le plus la valeur d'une instance. Parfois, il faut combiner tous les attributs. C'est le cas des classes `Point` et `Date`.

2.3 Comment combiner des hash codes ?

Pour combiner les attributs, il faut rédiger une expression mathématique garantissant la bonne répartition des valeurs du résultat dans l'ensemble des valeurs possibles. Une approche simple consiste à utiliser un ou exclusif entre deux valeurs de hash.

```
class Personne
{ String nom_;
  String prenom_;

  public int hashCode()
  { return nom_.hashCode() ^ prenom_.hashCode();
  }
}
```

Cette approche est efficace, mais présente plusieurs inconvénients.

```
    x ^ x = 0
    x ^ y = y ^ x
x ^ y ^ x = y
```

Cet algorithme opère seulement bit à bit. La modification d'un bit de la valeur de hash du `nom` n'influence qu'un seul bit de la valeur de hash du résultat. Cela ne garantit pas une répartition homogène des résultats si les attributs combinés ont une forte corrélation entre eux.

On peut améliorer le résultat de telle manière que la modification d'un seul bit entraîne la modification de tous les bits du résultat. On peut par exemple, utiliser la fonction suivante.

```
static final int combine_hash(int x,int y)
{ int rc=0;

  for (int i=7;i>=0;--i)
  { if (i==4) x=y;
    rc=rc*21+(char)x;
    x >>>=8;
  }
  return rc;
}
```

La modification d'un bit d'`x` ou d'`y` entraîne un résultat très différent. Il y a une meilleure répartition des résultats dans l'ensemble des valeurs possibles.

Une classe peut proposer des méthodes statiques pour faciliter le calcul des valeurs de hash.

```
public class Hash
{
  public static int hash(boolean x)
  { return x ? 0x55555555 : 0x2AAAAAAA;
  }
  ...
}
```

Cela permet de faciliter la rédaction des méthodes `hashCode()`.

3. Cacher la valeur de hash

Quand la méthode `hashCode()` est-t'elle utilisée ? Lors de l'insertion d'un élément dans un `Hashtable`, la méthode `hashCode()` est appelée pour classer l'instance dans un sous-ensemble. Lors de la recherche d'un élément, la méthode `hashCode()`sur l'instance à rechercher sert à sélectionner le sous-ensemble correspondant. Parfois, lorsque la répartition des instances dans les sous-ensembles est déséquilibrée, de nouveaux sous-ensembles sont ajoutés. Toutes les instances présentes dans le dictionnaire sont redistribuées dans les différents sous-ensembles. A cette occasion, la méthode `hashCode()` est appelée pour toutes les instances du dictionnaire.

Il est parfois intéressant de mémoriser la valeur de hash dans un cache. C'est le cas des instances immuables.

```
public class Immuable
{
  private int cacheHashCode_;
  public Immuable(...
  { ...
    cacheHashCode_=...
  }
  public int hashCode()
  { return cacheHashCode_;
  }
}
```

Le calcul de la valeur de hash est effectué une seule fois dans le constructeur. Ensuite, la méthode `hashCode()` retourne la valeur précédemment calculée. Cela permet également d'optimiser la méthode `equals()`.

```
public class Immuable
{
  ...
```

```
  public boolean equals(Object x)
  {
    if (x instancof Immuable)
    { Immuable obj=(Immuable)x;
      if (cacheHashCode_!=obj.cacheHashCode_) return false;
      return ...
    }
    return false;
  }
}
```

Si on construit une instance `Immuable` sans jamais avoir besoin d'appeler la méthode `hashCode()`, le calcul aura été effectué pour rien. Suivant le principe : "Je ne paye que pour ce que j'utilise", on peut améliorer la classe en supprimant le calcul de la valeur de hash dans le constructeur. Pour cela, on choisit une valeur de hash arbitraire signifiant que le calcul n'a pas été fait. La méthode `hashCode()` calculera la valeur seulement la première fois.

```
public class Immuable
{
  private transiant int cacheHashCode_=0;
  public Immuable(...
  { ...
  }
  public int hashCode()
  { if (cacheHashCode_==0) cacheHashCode_=...
    return cacheHashCode_;
  }
}
```

Le cache est déclaré `transiant` car il peut être recalculé si nécessaire. Il n'est pas utile de le sauver dans un fichier.

Il est possible, bien qu'exceptionnel, que le résultat du calcule donne justement la valeur zéro. Dans ce cas, le calcul sera effectué à chaque fois. Uniquement dans ce cas extrême, le cache n'apportera pas davantage. Pour garantir une utilisation optimum du cache, il faut faire en sorte que le calcul de la valeur de hash ne retourne jamais la valeur zéro. Cela peut être réglé facilement en modifiant la valeur dans ce cas.

```
public class Immuable
{ ...
  public int hashCode()
  { if (cacheHashCode_==0)
    {  cacheHashCode_=...
       if (cacheHashCode_==0) cacheHashCode_=-1;
    }
    return cacheHashCode_;
  }
}
```

Ainsi, la valeur `-1` à un peu plus de chance de sortie que les autres valeurs, mais on est garanti que le calcul de la valeur de hash ne sera effectué qu'une seule fois, si et seulement si, on utilise la méthode `hashCode()`.

Les instances modifiables peuvent également cacher la valeur de hash. Pour cela, elles doivent maintenir un drapeau indiquant la validité de la valeur du cache.

```
public class MaClass
{ private transiant int cacheHashCode_=0;
  private void calcHashCode()
  { ...
  }
  public void modifieInstance(...
  { cacheHashCode_=0;
    ...
  }
  public int hashCode()
  { if (cacheHashCode_==0)
    { cacheHashCode_=...
      if (cacheHashCode_==0) cacheHashCode_=-1;
    }
    return cacheHashCode_;
  }
}
```

Les méthodes de modifications de l'instance forcent la valeur du cache à zéro pour signaler qu'elle devra être recalculée.

Le cache consomme de la place mémoire (un entier). S'il existe de nombreuses instances de la classe, il est peut-être préférable de ne pas garder le cache et de recalculer la valeur à chaque demande. C'est le choix fait pour l'implantation de la classe `String`. La méthode `hashCode()` est généralement invoquée peu souvent.

4. Conclusion

Il est impossible de rédiger une méthode `hashCode()` générique. Cette méthode est trop dépendante de la sémantique de chaque objet et de chaque attribut de l'objet. Il faut parfois combiner les valeurs de hash de tous les attributs, parfois de certains seulement. Si la méthode

`hashCode()`retourne toujours la même valeur, l'algorithme `Dictionary` sera dans la situation la plus mauvaise. Tous les autres cas améliore les algorithmes de recherche. Pour obtenir le meilleur résultat, il faut que chaque valeur est autant de chance de sortir.