Java 理论与实践: 哈希

有效和正确定义hashCode()和equals()

每个Java对象都有 `hashCode()` 和 `equals()` 方法。许多类 Override 这些方法的缺省实施,以在对象实例之间提供更深层次的语义可比性。在 *Java理念和实践*这一部分，Java开发人员Brian Goetz向您介绍在创建Java类以有效和准确定义 `hashCode()` 和 `equals()` 时应遵循的规则和指南。您可以在 讨论论坛与作者和其它读者一同探讨您对本文的看法。(您还可以点击本文顶部或底部的 **讨论**进入论坛。)

Brian Goetz过去15年以来一直是专业软件开发人员。他是 Quiotix的首席顾问， Quiotix是位于加利福尼亚 Los Altos的一家软件开发和咨询公司。参阅Brian在流行行业出版物中 已经出版和即将出版的文章。可以通过 brian@quiotix.com与 Brian联系。

2003 年 8 月 11 日

虽然Java语言不直接支持关联数组 -- 可以使用任何对象作为一个索引的数组 -- 但在根 `Object` 类中使用 `hashCode()` 方法明确表示期望广泛使用 `HashMap` (及其前辈 `Hashtable`)。理想情况下基于散列的容器提供有效插入和有效检索；直接在对象模式中支持散列可以促进基于散列的容器的开发和使用。

定义对象的相等性

`Object` 类有两种方法来推断对象的标识： `equals()` 和 `hashCode()` 。一般来说，如果您 Override 了其中一种，您必须同时 Override 这两种，因为两者之间有必须维持的至关重

要的关系。特殊情况是根据 equals() 方法，如果两个对象是相等的，它们必须有相同的 hashCode() 值(尽管这通常不是真的)。

特定类的 equals() 的语义在Implementer的左侧定义；定义对特定类来说 equals() 意味着什么是其设计工作的一部分。 Object 提供的缺省实施简单引用下面等式：

```
public boolean equals(Object obj) { return (this == obj); }
```

在这种缺省实施情况下，只有它们引用真正同一个对象时这两个引用才是相等的。同样， Object 提供的 hashCode() 的缺省实施通过将对象的内存地址对映于一个整数值来生成。由于在某些架构上，地址空间大于 int 值的范围，两个不同的对象有相同的 hashCode() 是可能的。如果您 Override 了 hashCode() ，您仍旧可以使用 System.identityHashCode() 方法来接入这类缺省值。

Override equals() -- 简单实例

缺省情况下， equals() 和 hashCode() 基于标识的实施是合理的，但对于某些类来说，它们希望放宽等式的定义。例如， Integer 类定义 equals() 与下面类似：

```
public boolean equals(Object obj) {
  return (obj instanceof Integer
          && intValue() == ((Integer) obj).intValue());
}
```

在这个定义中，只有在包含相同的整数值的情况下这两个 Integer 对象是相等的。结合将不可修改的 Integer ，这使得使用 Integer 作为 HashMap 中的关键字是切实可行的。这种基于值的Equal方法可以由Java类库中的所有原始封装类使用，如 Integer 、 Float 、 Character 和 Boolean 以及 String (如果两个 String 对象包含相同顺序的字符，那它们是相等的)。由于这些类都是不可修改的并且可以实施 hashCode() 和 equals() ，它们都可以做为很好的散列关键字。

为什么 Override equals()和hashCode()?

如果 Integer 不 Override equals() 和 hashCode() 情况又将如何?如果我们从未在 HashMap 或其它基于散列的集合中使用 Integer 作为关键字的话，什么也不会发生。但是，如果我们在 HashMap中 使用这

类 Integer 对象作为关键字，我们将不能够可靠地检索相关的值，除非我们在 get() 调用中使用与 put() 调用中极其类似的 Integer 实例。这要求确保在我们的整个程序中，只能使用对应于特定整数值的 Integer 对象的一个实例。不用说，这种方法极不方便而且错误频频。

Object 的interface contract要求如果根据 equals() 两个对象是相等的，那么它们必须有相同的 hashCode() 值。当其识别能力整个包含在 equals() 中时，为什么我们的根对象类需要 hashCode()？hashCode() 方法纯粹用于提高效率。Java平台设计人员预计到了典型Java应用程序中基于散列的集合类（Collection Class)的重要性--如 Hashtable 、 HashMap 和 HashSet ，并且使用 equals() 与许多对象进行比较在计算方面非常昂贵。使所有Java对象都能够支持 hashCode() 并结合使用基于散列的集合，可以实现有效的存储和检索。

实施equals()和hashCode()的需求

实施 equals() 和 hashCode() 有一些限制， Object 文件中列举出了这些限制。特别是 equals() 方法必须显示以下属性：

Symmetry：两个引用， a 和 b , a.equals(b) if and only if b.equals(a)

Reflexivity：所有非空引用， a.equals(a)

Transitivity：If a.equals(b) and b.equals(c) , then a.equals(c)

Consistency with hashCode()： 两个相等的对象必须有相同的 hashCode() 值

Object 的规范中并没有明确要求 equals() 和 hashCode() 必须 一致-- 它们的结果在随后的调用中将是相同的，假设“不改变对象相等性比较中使用的任何信息。”这听起来象“计算的结果将不改变，除非实际情况如此。”这一模糊声明通常解释为相等性和散列值计算应是对象的可确定性功能，而不是其它。

对象相等性意味着什么?

人们很容易满足Object类规范对 equals() 和 hashCode() 的要求。决定是否和如何 Override equals() 除了判断以外，还要求其它。在简单的不可修值类中，如 Integer (事实上是几乎所有不可修改的类)，选择相当明显 -- 相等性应基于基本对象状态的相等性。在 Integer 情况下，对象的唯一状态是基本的整数值。

对于可修改对象来说，答案并不总是如此清楚。 equals() 和 hashCode() 是否应基于对象的标识(象缺省实施)或对象的状态(象Integer和String)? 没有简单的答案 -- 它取决于类的计划使用。对于象 List 和 Map 这样的容器来说，人们对此争论不已。Java类库中的大多数类，包括容器类，错误出现在根据对象状态来提供 equals() 和 hashCode() 实施。

如果对象的 hashCode() 值可以基于其状态进行更改，那么当使用这类对象作为基于散列的集合中的关键字时我们必须注意，确保当它们用于作为散列关键字时，我们并不允许更改它们的状态。所有基于散列的集合假设，当对象的散列值用于作为集合中的关键字时它不会改变。如果当关键字在集合中时它的散列代码被更改，那么将产生一些不可预测和容易混淆的结果。实践过程中这通常不是问题 -- 我们并不经常使用象 List 这样的可修改对象做为 HashMap 中的关键字。

一个简单的可修改类的例子是Point，它根据状态来定义 equals() 和 hashCode() 。如果两个 Point 对象引用相同的 (x, y) 座标， Point 的散列值来源于 x 和 y 座标值的IEEE 754-bit表示，那么它们是相等的。

对于比较复杂的类来说， equals() 和 hashCode() 的行为可能甚至受到superclass或interface的影响。例如， List 接口要求如果并且只有另一个对象是 List， 而且它们有相同顺序的相同的Elements(由Element上的 Object.equals() 定义)， List 对象等于另一个对象。 hashCode() 的需求更特殊--list的 hashCode() 值必须符合以下计算：

```
hashCode = 1;
Iterator i = list.iterator();
while (i.hasNext()) {
    Object obj = i.next();
    hashCode = 31*hashCode + (obj==null ? 0 : obj.hashCode());
}
```

不仅仅散列值取决于list的内容，而且还规定了结合各个Element的散列值的特殊算法。(`String` 类规定类似的算法用于计算 `String` 的散列值。)

编写自己的equals()和hashCode()方法

Override 缺省的 `equals()` 方法比较简单，但如果不违反对称（Symmetry）或传递性（Transitivity）需求，Override 已经 Override 的 `equals()` 方法极其棘手。当 Override `equals()` 时，您应该总是在 `equals()` 中包括一些Javadoc注释，以帮助那些希望能够正确扩展您的类的用户。

作为一个简单的例子，考虑以下类：

```
class A {
  final B someNonNullField;
  C someOtherField;
  int someNonStateField;
}
```

我们应如何编写该类的 `equals()` 的方法？这种方法适用于许多情况：

```
public boolean equals(Object other) {
  // Not strictly necessary, but often a good optimization
  if (this == other)
    return true;
  if (!(other instanceof A))
    return false;
  A otherA = (A) other;
  return
    (someNonNullField.equals(otherA.someNonNullField))
      && ((someOtherField == null)
          ? otherA.someOtherField == null
          : someOtherField.equals(otherA.someOtherField)));
}
```

现在我们定义了 `equals()` ，我们必须以统一的方法来定义 `hashCode()` 。一种统一但并不总是有效的定义 `hashCode()` 的方法如下：

```
public int hashCode() { return 0; }
```

这种方法将生成大量的条目并显著降低 HashMap s的性能，但它符合规范。一个更合理的 hashCode() 实施应该是这样：

```
public int hashCode() {
  int hash = 1;
  hash = hash * 31 + someNonNullField.hashCode();
  hash = hash * 31
             + (someOtherField == null ? 0 : someOtherField.hashCode());
  return hash;
}
```

注意：这两种实施都降低了类状态字段的 equals() 或 hashCode() 方法一定比例的计算能力。根据您使用的类，您可能希望降低superclass的 equals() 或 hashCode() 功能一部分计算能力。对于原始字段来说，在相关的封装类中有helper功能，可以帮助创建散列值，如 Float.floatToIntBits 。

编写一个完美的 equals() 方法是不现实的。通常，当扩展一个自身 Override 了 equals() 的 instantiable类时，Override equals() 是不切实际的，而且编写将被 Override 的 equals() 方法(如在抽象类中)不同于为具体类编写 equals() 方法。关于实例以及说明的更详细信息请参阅 *Effective Java Programming Language Guide*, Item 7 (参考资料) 。

有待改进?

将散列法构建到Java类库的根对象类中是一种非常明智的设计折衷方法 -- 它使使用基于散列的容器变得如此简单和高效。但是，人们对Java类库中的散列算法和对象相等性的方法和实施提出了许多批评。java.util 中基于散列的容器非常方便和简便易用，但可能不适用于需要非常高性能的应用程序。虽然其中大部分将不会改变，但当您设计严重依赖于基于散列的容器效率的应用程序时必须考虑这些因素，它们包括：

太小的散列范围。使用 int 而不是 long 作为 hashCode() 的返回类型增加了散列冲突的几率。

糟糕的散列值分配。短strings和小型integers的散列值是它们自己的小整数，接近于其它“邻近”对象的散列值。一个循规导矩（Well-behaved）的散列函数将在该散列范围内更均匀地分配散列值。

无定义的散列操作。虽然某些类，如 `String` 和 `List` ，定义了将其Element的散列值结合到一个散列值中使用的散列算法，但语言规范不定义将多个对象的散列值结合到新散列值中的任何批准的方法。我们在前面 编写自己的equals()和hashCode()方法中讨论的 `List` 、 `String` 或实例类 `A` 使用的诀窍都很简单，但算术上还远远不够完美。类库不提供任何散列算法的方便实施，它可以简化更先进的 `hashCode()` 实施的创建。

当扩展已经 Override 了 `equals()` 的 instantiable类时很难编写 `equals()` 。当扩展已经 Override 了 `equals()` 的 instantiable类时，定义 `equals()` 的“显而易见的”方式都不能满足 `equals()` 方法的对称或传递性需求。这意味着当 Override `equals()` 时，您必须了解您正在扩展的类的结构和实施详细信息，甚至需要暴露基本类中的机密字段，它违反了面向对象的设计的原则。

结束语

通过统一定义 `equals()` 和 `hashCode()`， 您可以提升类作为基于散列的集合中的关键字的使用性。有两种方法来定义对象的相等性和散列值：基于标识，它是 `Object` 提供的缺省方法；基于状态，它要求 Override `equals()` 和 `hashCode()` 。当对象的状态更改时如果对象的散列值发生变化，确信当状态作为散列关键字使用时您不允许更更改其状态。

参考资料

您可以参阅本文在 developerWorks 全球站点上的 英文原文.

参加本文的 讨论论坛。(您还可以点击本文顶部或底部的 **讨论**进入论坛。)

阅读Brian Goetz撰写的一整套 *Java理论和实践文章*。尤其是2003年2月“ Java理论与实践：变还是不变？ ，”它讨论使用可变对象作为散列关键字的危害。

Joshua Bloch杰作的第 7和 8部分 *Java编程语言指南*，详细阐述围绕 `equals()` 和 `hashCode()` 的问题。

Tony Sintes在 *JavaWorld*提供的本文中解释 基于散列的容器是如何工作的以及如何使用 `equals()` 和 `hashCode()` (2002年7月)。

在幻灯片中，新西兰奥克兰大学计算机科学系的Robert Uzgalis介绍一些 Java 散列模式的批评意见，解释一些散列函数背后的问题。

Mark Roulo 在自己的文章“如何避免陷阱和正确 Override java.lang.Object的方法”(*JavaWorld*, 1999年1月)一文中提供了一些 Override `equals()` 和 `hashCode()` 的实例程序代码 。

新西兰坎特伯雷大学计算机科学系提供的这一份技术报告详细描述了 what makes an effective hash function(PDF)。

IBM软件实验室软件工程师Sreekanth Iyer 探讨了Java对象相等性的各种不同意义(*developerWorks*, 2002年9月)。

JavaWorld(获得许可后才可再版)的提示略微谈到 相等性比较的缺陷。

在 *developerWorks*Java技术专区 可以找到数百篇有关 Java 技术的参考资料。.