20180331_Java根类Object的方法说明

这篇文章对于Object类的方法介绍的还算挺详细的：

```
1- 类相关：hashCode, equals, toString, getClass
   线程相关：notify,  notifyAll, wait
   垃圾回收：finalize
```

Java中的Object类是所有类的父类，它提供了以下11个方法：

1. public final native Class<?> getClass()
2. public native int hashCode()
3. public boolean equals(Object obj)
4. protected native Object clone() throws CloneNotSupportedException
5. public String toString()
6. public final native void notify()
7. public final native void notifyAll()
8. public final native void wait(long timeout) throws InterruptedException
9. public final void wait(long timeout, int nanos) throws InterruptedException
10. public final void wait() throws InterruptedException
11. protected void finalize() throws Throwable { }

下面我们一个个方法进行分析，看这些方法到底有什么作用：

getClass方法

getClass方法是一个final方法，不允许子类重写，并且也是一个native方法。

返回当前运行时对象的Class对象，注意这里是运行时，比如以下代码中n是一个Number类型的实例，但是java中数值默认是Integer类型，所以getClass方法返回的是java.lang.Integer：

```
"str".getClass() // class java.lang.String
"str".getClass == String.class // true
Number n = 0;
Class<? extends Number> c = n.getClass(); // class java.lang.Integer
```

hashCode方法

hashCode方法也是一个native方法。

该方法返回对象的哈希码，主要使用在哈希表中，比如JDK中的HashMap。

哈希码的通用约定如下：

1. 在java程序执行过程中，在一个对象没有被改变的前提下，无论这个对象被调用多少次，hashCode方法都会返回相同的整数值。对象的哈希码没有必要在不同的程序中保持相同的值。
2. 如果2个对象使用equals方法进行比较并且相同的话，那么这2个对象的hashCode方法的值也必须相等。
3. 如果根据equals方法，得到两个对象不相等，那么这2个对象的hashCode值不需要必须不相同。但是，不相等的对象的hashCode值不同的话可以提高哈希表的性能。

通常情况下，不同的对象产生的哈希码是不同的。默认情况下，对象的哈希码是通过将该对象的内部地址转换成一个整数来实现的。

String的hashCode方法实现如下， 计算方法是 s[0]*31^(n-1) + s[1]*31^(n-2) + ... + s[n-1]，其中s[0]表示字符串的第一个字符，n表示字符串长度：

```
public int hashCode() {
    int h = hash;
    if (h == 0 && value.length > 0) {
        char val[] = value;

        for (int i = 0; i < value.length; i++) {
            h = 31 * h + val[i];
        }
        hash = h;
    }
    return h;
}
```

比如"fo"的hashCode = 102 *31^1 + 111 = 3273， "foo"的hashCode = 102* 31^2 + 111 * 31^1 + 111 = 101574 ('f'的ascii码为102, 'o'的ascii码为111)

hashCode在哈希表HashMap中的应用：

```
// Student类，只重写了hashCode方法
public static class Student {

    private String name;
    private int age;

    public Student(String name, int age) {
        this.name = name;
        this.age = age;
    }

    @Override
    public int hashCode() {
        return name.hashCode();
    }
}

Map<Student, String> map = new HashMap<Student, String>();
```

```
Student stu1 = new Student("fo", 11);
Student stu2 = new Student("fo", 22);
map.put(stu1, "fo");
map.put(stu2, "fo");
```

上面这段代码中，map中有2个元素stu1和stu2。但是这2个元素是在哈希表中的同一个数组项中的位置，也就是在同一串链表中。 但是为什么stu1和stu2的hashCode相同，但是两条元素都插到map里了，这是因为map判断重复数据的条件是 **两个对象的哈希码相同并且(两个对象是同一个对象或者两个对象相等[equals为true])**。 所以再给Student重写equals方法，并且只比较name的话，这样map就只有1个元素了。

```
@Override
public boolean equals(Object o) {
    if (this == o) return true;
    if (o == null || getClass() != o.getClass()) return false;
    Student student = (Student) o;
    return this.name.equals(student.name);
}
```

这个例子直接说明了hashCode中通用约定的第三点：

第三点：如果根据equals方法，得到两个对象不相等，那么这2个对象的hashCode值不需要必须不相同。但是，不相等的对象的hashCode值不同的话可以提高哈希表的性能。 -> 上面例子一开始没有重写equals方法，导致两个对象不相等，但是这两个对象的hashCode值一样，所以导致这两个对象在同一串链表中，影响性能。

当然，还有第三种情况，那就是equals方法相等，但是hashCode的值不相等。

这种情况也就是违反了通用约定的第二点：

第二点：**如果2个对象使用equals方法进行比较并且相同的话，那么这2个对象的hashCode方法的值也必须相等。** 违反这一点产生的后果就是如果一个stu1实例是Student(“fo”, 11)，stu2实例是Student(“fo”, 11)，那么这2个实例是相等的，但是他们的hashCode不一样，这样是导致哈希表中都会存入stu1实例和stu2实例，但是实际情况下，stu1和stu2是重复数据，只允许存在一条数据在哈希表中。所以这一点是非常重点的，再强调一下：**如果2个对象的equals方法相等，那么他们的hashCode值也必须相等，反之，如果2个对象hashCode值相等，但是equals不相等，这样会影响性能，所以还是建议2个方法都一起重写。**

equals方法

比较两个对象是否相等。Object类的默认实现，即比较2个对象的内存地址是否相等：

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

equals方法在非空对象引用上的特性：

1. reflexive，自反性。任何非空引用值x，对于x.equals(x)必须返回true
2. symmetric，对称性。任何非空引用值x和y，如果x.equals(y)为true，那么y.equals(x)也必须为true
3. transitive，传递性。任何非空引用值x、y和z，如果x.equals(y)为true并且y.equals(z)为true，那么x.equals(z)也必定为true
4. consistent，一致性。任何非空引用值x和y，多次调用 x.equals(y) 始终返回 true 或始终返回 false，前提是对象上 equals 比较中所用的信息没有被修改
5. 对于任何非空引用值 x，x.equals(null) 都应返回 false

Object类的equals方法对于任何非空引用值x和y，当x和y引用同一个对象时，此方法才返回true。这个也就是我们常说的地址相等。

注意点：如果重写了equals方法，通常有必要重写hashCode方法，这点已经在hashCode方法中说明了。

clone方法

创建并返回当前对象的一份拷贝。一般情况下，对于任何对象 x，表达式 x.clone() != x 为true，x.clone().getClass() == x.getClass() 也为true。

Object类的clone方法是一个protected的native方法。

由于Object本身没有实现Cloneable接口，所以不重写clone方法并且进行调用的话会发生CloneNotSupportedException异常。

toString方法

Object对象的默认实现，即输出类的名字@实例的哈希码的16进制：

```
public String toString() {
    return getClass().getName() + "@" + Integer.toHexString(hashCode());
}
```

toString方法的结果应该是一个简明但易于读懂的字符串。建议Object所有的子类都重写这个方法。

notify方法

notify方法是一个native方法，并且也是final的，不允许子类重写。

唤醒一个在此对象监视器上等待的线程(监视器相当于就是锁的概念)。如果所有的线程都在此对象上等待，那么只会选择一个线程。选择是任意性的，并在对实现做出决定时发生。一个线程在对象监视器上等待可以调用wait方法。

直到当前线程放弃对象上的锁之后，被唤醒的线程才可以继续处理。被唤醒的线程将以常规方式与在该对象上主动同步的其他所有线程进行竞争。例如，唤醒的线程在作为锁定此对象的下一个线程方面没有可靠的特权或劣势。

notify方法只能被作为此对象监视器的所有者的线程来调用。一个线程要想成为对象监视器的所有者，可以使用以下3种方法：

1. 执行对象的同步实例方法

2. 使用synchronized内置锁
3. 对于Class类型的对象，执行同步静态方法

一次只能有一个线程拥有对象的监视器。

如果当前线程不是此对象监视器的所有者的话会抛出IllegalMonitorStateException异常

注意点：

因为notify只能在拥有对象监视器的所有者线程中调用，否则会抛出IllegalMonitorStateException异常

notifyAll方法

跟notify一样，唯一的区别就是会唤醒在此对象监视器上等待的所有线程，而不是一个线程。

同样，如果当前线程不是对象监视器的所有者，那么调用notifyAll同样会发生IllegalMonitorStateException异常。

以下这段代码直接调用notify或者notifyAll方法会发生IllegalMonitorStateException异常，这是因为调用这两个方法需要当前线程是对象监视器的所有者：

```
Factory factory = new Factory();
factory.notify();
factory.notifyAll();
```

wait(long timeout) throws InterruptedException方法

wait(long timeout)方法同样是一个native方法，并且也是final的，不允许子类重写。

wait方法会让当前线程等待直到另外一个线程调用对象的notify或notifyAll方法，或者超过参数设置的timeout超时时间。

跟notify和notifyAll方法一样，当前线程必须是此对象的监视器所有者，否则还是会发生IllegalMonitorStateException异常。

wait方法会让当前线程(我们先叫做线程T)将其自身放置在对象的等待集中，并且放弃该对象上的所有同步要求。出于线程调度目的，线程T是不可用并处于休眠状态，直到发生以下四件事中的任意一件：

1. 其他某个线程调用此对象的notify方法，并且线程T碰巧被任选为被唤醒的线程
2. 其他某个线程调用此对象的notifyAll方法
3. 其他某个线程调用Thread.interrupt方法中断线程T
4. 时间到了参数设置的超时时间。如果timeout参数为0，则不会超时，会一直进行等待

所以可以理解wait方法相当于放弃了当前线程对对象监视器的所有者(也就是说释放了对象的锁)

之后，线程T会被等待集中被移除，并且重新进行线程调度。然后，该线程以常规方式与其他线程竞争，以获得在该对象上同步的权利；一旦获得对该对象的控制权，该对象上的所有其同步声明都将被恢复到以前的状态，这就是调用wait方法时的情况。然后，线程T从wait方法的调用中返回。所以，从wait方法返回时，该对象和线程T的同步状态与调用wait方法时的情况完全相同。

在没有被通知、中断或超时的情况下，线程还可以唤醒一个所谓的虚假唤醒 (spurious wakeup)。虽然这种情况在实践中很少发生，但是应用程序必须通过以下方式防止其发生，即对应该导致该线程被提醒的条件进行测试，如果不满足该条件，则继续等待。换句话说，等待应总是发生在循环中，如下面的示例：

```
synchronized (obj) {
    while (<condition does not hold>)
        obj.wait(timeout);
        ... // Perform action appropriate to condition
}
```

如果当前线程在等待之前或在等待时被任何线程中断，则会抛出InterruptedException异常。在按上述形式恢复此对象的锁定状态时才会抛出此异常。

wait(long timeout, int nanos) throws InterruptedException方法

跟wait(long timeout)方法类似，多了一个nanos参数，这个参数表示额外时间（以毫微秒为单位，范围是 0-999999）。 所以超时的时间还需要加上nanos毫秒。

需要注意的是 wait(0, 0)和wait(0)效果是一样的，即一直等待。

wait() throws InterruptedException方法

跟之前的2个wait方法一样，只不过该方法一直等待，没有超时时间这个概念。

以下这段代码直接调用wait方法会发生IllegalMonitorStateException异常，这是因为调用wait方法需要当前线程是对象监视器的所有者：

```
Factory factory = new Factory();
factory.wait();
```

一般情况下，wait方法和notify方法会一起使用的，wait方法阻塞当前线程，notify方法唤醒当前线程，一个使用wait和notify方法的生产者消费者例子代码如下：

```
public class WaitNotifyTest {

    public static void main(String[] args) {
        Factory factory = new Factory();
        new Thread(new Producer(factory, 5)).start();
        new Thread(new Producer(factory, 5)).start();
        new Thread(new Producer(factory, 20)).start();
        new Thread(new Producer(factory, 30)).start();
        new Thread(new Consumer(factory, 10)).start();
        new Thread(new Consumer(factory, 20)).start();
        new Thread(new Consumer(factory, 5)).start();
        new Thread(new Consumer(factory, 5)).start();
```

```
        new Thread(new Consumer(factory, 20)).start();
    }

}

class Factory {

    public static final Integer MAX_NUM = 50;

    private int currentNum = 0;

    public void consume(int num) throws InterruptedException {
        synchronized (this) {
            while(currentNum - num < 0) {
                this.wait();
            }
            currentNum -= num;
            System.out.println("consume " + num + ", left: " + currentNum);
            this.notifyAll();
        }
    }

    public void produce(int num) throws InterruptedException {
        synchronized (this) {
            while(currentNum + num > MAX_NUM) {
                this.wait();
            }
            currentNum += num;
            System.out.println("produce " + num + ", left: " + currentNum);
            this.notifyAll();
        }
    }

}

class Producer implements Runnable {
    private Factory factory;
    private int num;
    public Producer(Factory factory, int num) {
        this.factory = factory;
        this.num = num;
    }
    @Override
    public void run() {
        try {
            factory.produce(num);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
```

```
    }
}


class Consumer implements Runnable {
    private Factory factory;
    private int num;
    public Consumer(Factory factory, int num) {
        this.factory = factory;
        this.num = num;
    }
    @Override
    public void run() {
        try {
            factory.consume(num);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}
```

注意的是Factory类的produce和consume方法都将Factory实例锁住了，锁住之后线程就成为了对象监视器的所有者，然后才能调用wait和notify方法。

输出：

```
produce 5, left: 5
produce 20, left: 25
produce 5, left: 30
consume 10, left: 20
produce 30, left: 50
consume 20, left: 30
consume 5, left: 25
consume 5, left: 20
consume 20, left: 0
```

finalize方法

finalize方法是一个protected方法，Object类的默认实现是不进行任何操作。

该方法的作用是实例被垃圾回收器回收的时候触发的操作，就好比 “死前的最后一波挣扎”。

直接写个弱引用例子：

```
Car car = new Car(9999, "black");
WeakReference<Car> carWeakReference = new WeakReference<Car>(car);

int i = 0;
```

```
while(true) {
    if(carWeakReference.get() != null) {
        i++;
        System.out.println("Object is alive for "+i+" loops -
"+carWeakReference);
    } else {
        System.out.println("Object has been collected.");
        break;
    }
}

class Car {
    private double price;
    private String colour;

    public Car(double price, String colour){
        this.price = price;
        this.colour = colour;
    }

    // get set method

    @Override
    protected void finalize() throws Throwable {
        System.out.println("i will be destroyed");
    }
}
```

输出:

```
....
Object is alive for 26417 loops - java.lang.ref.WeakReference@7c2f1622
Object is alive for 26418 loops - java.lang.ref.WeakReference@7c2f1622
Object is alive for 26419 loops - java.lang.ref.WeakReference@7c2f1622
Object is alive for 26420 loops - java.lang.ref.WeakReference@7c2f1622
Object is alive for 26421 loops - java.lang.ref.WeakReference@7c2f1622
Object is alive for 26422 loops - java.lang.ref.WeakReference@7c2f1622
Object has been collected.
i will be destroyed
```