参考文章：<https://www.cnblogs.com/gollong/p/9329557.html>

<https://www.cnblogs.com/eer123/p/7880789.html>

核心：wait和notify/notifyAll的区别

Wait方法将会释放锁-》否则notify方法将会无法获取锁

Notify和notifyAll都不会释放锁，只是等到同步代码块执行结束的时候，才将锁给释放

## Object类的源码分析

/\*

\* Object类是类层次结构的根。每个类都有 Object作为超类。所有对象，包括数组，实现这个类的方法。

\*/

public class Object {

/\* native:

\* 我们看到了有好几个方法是由native修饰的，那么native方法是什么意思呢？

\* native关键字说明其修饰的方法是一个原生态方法，方法对应的实现不是在当前文件，而是在用其他语言（如C和C++）实现的文件中。

\* Java语言本身不能对操作系统底层进行访问和操作，但是可以通过JNI（Java Native Interface）接口调用其他语言来实现对底层的访问。

\* JNI是Java本机接口，是一个本机编程接口，它是JDK的一部分。JNI允许Java代码使用以其他语言编写的代码和代码库。

\* InvocationAPI（JNI的一部分）可以用来将Java虚拟机（JVM）嵌入到本机应用程序中，从而允许程序员从本机代码内部调用Java代码。

\*/

private static native void registerNatives();

/\*

\* registerNatives()：

\* 顾名思义,在类初始化时调用registerNatives()方法进行本地方法的注册，也就是初始化java原生方法（native修饰）映射到对应的其他语言描述方法，比如c语言的方法

\* 我们可以发现用native修饰的方法都没有具体的方法体（类似于抽象方法），因为不需要java来实现，是使用其他语言实现的。直接调用即可，而且同时用final修饰，我们都无法重写

\* 但是hashCode()却不一样，并没有final修饰，而hashCode()的重写对于散列数据结构大有用处。所以说hashCode()既是一个native方法也是一个java方法。

\*/

static {

registerNatives();

}

/\*

\* getClass()：

\* getclass()方法返回该 Object运行时类，一般在反射时，我们想得到某个类的class对象，可以通过对象.getClass()获得，但几乎不常用，因为既然都有对象了就不用反射了。

\*/

public final native Class<?> getClass();

/\*

\* hashCode()：

\* 学习hashCode()，需要先学习equals(Object obj)，hashCode()一般不单独使用，hashCode()的功能是在比较俩个对象是否相等的时候提高比较效率，比如在一些hashSet、hashTable等

\* 数据结构中，不允许存在相同的元素，所以每次插入元素都要和已经存在的元素依次进行比较，如果依次对每个元素调用equals(Object obj)，对于大量数据来说太慢了，hashCode()便是在哈希表

\* 结构中提高比较速率的利器，如果俩个元素的hashCode()不同，则equals(Object obj)就不必比较了，因为肯定不同；如果俩个元素的hashCode()相同，则再比较equals(Object obj)

\* 于是我们得出了一个结论：hashCode()相同，equals(Object obj)不一定相同；hashCode()不同，equals(Object obj)一定不同；equals(Object obj)相同，hashCode()一点相同；

\* 但是有一个前提，必须同时重写equals(Object obj)和hashCode()。

\*/

public native int hashCode();

/\*

\* equals(Object obj)：

\* equals(Object obj)方法是比较俩个对象是否相等的主要方法，通过源码我们可以看出如果某个类不重写equals(Object obj)方法，那么equals(Object obj)就是==，

\* 而==比较的是基本类型的内容，引用类型的地址，其实无论基本类型还是引用类型，==比较的都是栈内存中的值，可以去看一下JVM的内存模型就明白了。言归正传，不重写equals(Object obj)的话

\* 和==的功能一样，我们可以通过系统的模板一键重写，重写默认比较的是所有属性的值，也可以根据自己的实际需求，去改写自己需要的功能。

\* String类默认重写了此方法。

\*/

public boolean equals(Object obj) {

return (this == obj);

}

/\*

\* 返回一个此类的克隆对象，二者具有相同的属性，二者拥有独立的属性存储空间，分别位于堆内存中的俩快空间中，不相互影响，但是也不是绝对的，因为存在浅克隆和深克隆的区别，我会用一篇文章来介绍深克隆和浅克隆

\* 这里注意区别a = b 和 a = b.clone(),他们在堆内存中的存储不一样。

\*/

protected native Object clone() throws CloneNotSupportedException;

/\*

\* toString()：

\* 最最最常用的方法，如果你定义一个类，而不去重写toString()的话，那你每一次想看看该类的某个对象的某些属性，你就不得不写一堆getXxx()方法，简直了。其实在输出一个字符串对象a时，输出的就是

\* a.toString();我们在类中重写此方法后，直接输出就是调用此方法，也就是依次遍历对象的所有属性。但是System.out.println(a)和System.out.println(a.toString())是有区别的

\* 当a对象没有实例化时，其中一个会报出NullPointException，至于是哪个，自己动手去试试不就知道了嘛。

\*/

public String toString() {

return getClass().getName() + "@" + Integer.toHexString(hashCode());

}

/\*

\* notify()：

\* 接下来的俩个方法（notify()和wait()）都是在多线程中使用的，至于为什么多线程的方法要写在Object中，这就难以琢磨了！

\* notify()唤醒一个在这个对象的监视器上等待的单个线程。什么是监视器呢，我们知道一个CPU一次只能处理一个进程（线程），那么是谁保证只能有一个进程进入CPU呢，

\* 别忘了，每个进程都是自私的，他可不管里面有没有其他同类。所以我们需要一套规则，需要一个上帝来制约所有的进程，监视器就是这个上帝，他负责某些进程，来维护他们之间的秩序。

\* 唤醒的线程不是得到CPU的使用权，而是和其他就绪线程竞争。也就是将阻塞状态转换为就绪状态，并不是立即执行。

\*

\*/

public final native void notify();

/\*

\* notifyAll()：

\* notifyAll()唤醒一个在这个对象的监视器上等待的所有线程，只要记住唤醒的线程不是得到CPU的使用权，而是和其他就绪线程竞争。

\*/

public final native void notifyAll();

/\*

\* wait(long timeout)：

\* wait(long timeout)使当前线程等待另一个线程调用此对象的方法或 notify() notifyAll()方法，或一个指定的时间流逝,API上的谷歌翻译，有点拗口，其实我也不明白什么意思

\* 我猜测大致是，wait方法是一个线程进行等待状态，等待多长时间有参数决定，时间到后自动唤醒，或者在等待期间又其他进程调用其的notify或者notifyAll方法进行唤醒。

\* 而wait(0)代表无限期的等待。因为等待0秒毫无意义！

\* 通过下面的源码，我们可以很明确的看出：wait()==wait(0)==wait(0,0)都是无限期的等待，直到被notify() notifyAll()唤醒

\*/

public final native void wait(long timeout) throws InterruptedException;

/\*

\*

\*/

public final void wait(long timeout, int nanos) throws InterruptedException {

if (timeout < 0) {

throw new IllegalArgumentException("timeout value is negative");

}

if (nanos < 0 || nanos > 999999) {

throw new IllegalArgumentException("nanosecond timeout value out of range");

}

if (nanos > 0) {

timeout++;

}

wait(timeout);

}

/\*

\*

\*/

public final void wait() throws InterruptedException {

wait(0);

}

/\*

\* finalize()：

\* 当垃圾收集确定没有对对象的引用时，由对象上的垃圾收集器调用。子类重写 finalize方法配置系统资源或执行其他清理。垃圾回收机制中的方法，我会用一章来详细讲解垃圾回收机制

\*/

protected void finalize() throws Throwable {

}

}