**Object.class源码**

/\*

\* Copyright (c) 1994, 2012, Oracle and/or its affiliates. All rights reserved.

\* ORACLE PROPRIETARY/CONFIDENTIAL. Use is subject to license terms.

\*

\*/

package java.lang;

/\*\*

\* Class {@code Object} is the root of the class hierarchy.

\* Every class has {@code Object} as a superclass. All objects,

\* including arrays, implement the methods of this class.

\*

\* @author unascribed

\* @see java.lang.Class

\* @since JDK1.0

\*/

public class Object {

private static native void registerNatives();

static {

registerNatives();

}

public final native Class<?> getClass();

public native int hashCode();

public boolean equals(Object obj) {

return (this == obj);

}

protected native Object clone() throws CloneNotSupportedException;

public String toString() {

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

}

public final native void notify();

public final native void notifyAll();

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);

}

protected void finalize() throws Throwable { }

}

## 1.getClass方法

public final native Class<?> getClass();

返回此 Object 的运行时类，返回的类对象是**被表示类的static synchronized方法锁定的对象**。不可以重写，要调用的话，**一般和getName（）联合使用**。

类加载的第一阶段就是将.class文件加载到内存，并生成一个java.lang.Class对象的过程。**getClass（）方法就是获取这个对象**，这是当前类的对象在运行时类的所有信息的集合。**这个方法也是三种反射方式之一**。

反射三种方式：

1. 对象的getClass（）；
2. 类名.class；
3. Class.forName（）；

## 2.registerNatives方法

private static native void registerNatives();

static {

registerNatives();

}

该方法中的静态代码块就是**一个类在初始化过程中必定会执行的内容**，所以在类加载的时候会执行该方法，**通过该方法来注册绑定本地方法**。通过以下OpenJDK中的Thread.c的部分代码

···

static JNINativeMethod methods[] = {

{"start0", "()V", (void \*)&JVM\_StartThread},

{"stop0", "(" OBJ ")V", (void \*)&JVM\_StopThread},

{"isAlive", "()Z", (void \*)&JVM\_IsThreadAlive},

{"suspend0", "()V", (void \*)&JVM\_SuspendThread},

{"resume0", "()V", (void \*)&JVM\_ResumeThread},

{"setPriority0", "(I)V", (void \*)&JVM\_SetThreadPriority},

{"yield", "()V", (void \*)&JVM\_Yield}

};

JNIEXPORT void JNICALL

Java\_java\_lang\_Thread\_registerNatives(JNIEnv \*env, jclass cls)

{

(\*env)->RegisterNatives(env, cls, methods, ARRAY\_LENGTH(methods));

}

···

通过以上代码可知**registerNatives方法是通过JNI\_onload函数实现动态绑定**。

## 3.clone方法

protected native Object clone() throws CloneNotSupportedException;

该方法是保护方法，实现对象的浅复制，**只有实现了Cloneable接口才可以调用该方法**，否则抛出**CloneNotSupoortedException异常**。克隆的对象通常情况下满足以下三条规则：

1. x.clone（）！= x，克隆出来的对象和原来的对象不是同一个，指向不同的内存地址
2. x.clone（）.getClass（） == x.getClass（）
3. x.clone（）.equals（x）

默认的clone方法是浅拷贝。所谓浅拷贝，**指的是对象内属性引用的对象只会拷贝引用地址，而不会将引用的对象重新分配内存**。而深拷贝则是会连**引用的对象也重新创建**。

## 4.toString方法

public String toString() {

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

}

toString方法返回一个字符串，该方法中的**getClass（）.getName（）是返回对象的全类名（包含包名）**，Integer.toHexString（）是将**hash码以十六进制无符号整数形式**返回此hash码的字符串表示形式。一般在子类中我们可以对**这个方法进行重写**。

## 5.equals方法（重要）

public boolean equals(Object obj) {

return (this == obj);

}

equals方法**用来直接判断this和obj本省的值是否相等**，即用来判断调用equals的对象和形参obj所引用的对象是否同一对象，所谓同一对象就是指内存中同一块存储单元，**如果this和obj指向的是同一块内存对象，则返回true，否则返回false**。

1. 即便是内容完全相等的两块不同的内存对象，也返回false。
2. 如果希望不同内存但相同内容的两个对象时，Object中的equals方法返回true，则我们需要重写父类的equals方法。
3. String类已经重写了Object类的equals方法

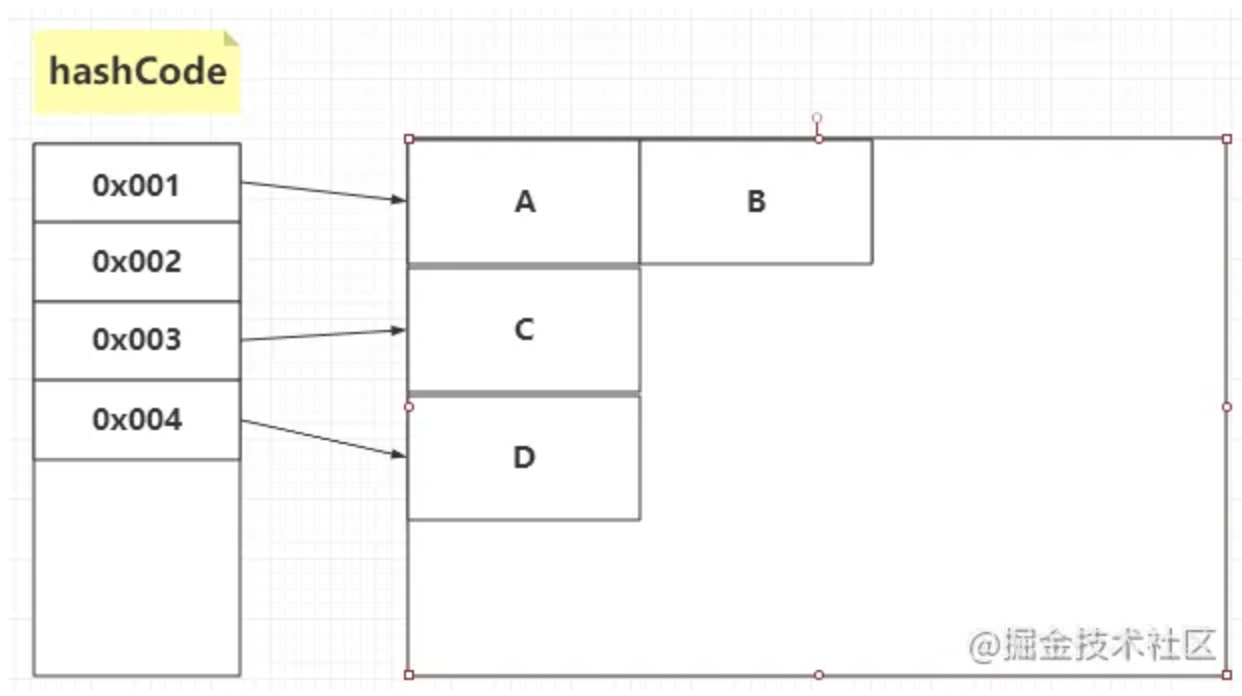
**请注意，无论何时重写此方法，通常都必须重写hashCode方法，以维护hashCode方法的一般约定，该方法声明相等对象必须具有相同的哈希代码。**

## 6.hashCode方法（重要）

public native int hashCode();

hashCode方法用一个native来声明，**返回该对象的哈希码**，是int类型的数值，用于哈希查找，**可以减少在查找中使用equals的次数**。

**哈希算法也称为散列算法**，是将数据依特定算法产生的结果直接指定到一个地址上，整个结果是由hashCode()方法产生的。



这里有 A,B,C,D四个对象，分别通过hashCode方法产生了三个值，注意A和B对象调用hashCode产生的值是相同的，即A.hashCode（）=B.hashCode（）=0x001,发生了哈希冲突，这时候由于最先是插入了A，在插入的B的时候，我们发现B是要插入到A所在的位置，而A已经插入了，**这时候就通过调用equals方法判断A和B是否相同，如果相同就不插入B，如果不同则将B插入到A后面的位置。**

## 7. wait/notify/notifyAll方法

#### wait 方法

public final void wait() throws InterruptedException {

wait(0);

}

该方法用来将当前线程置入休眠状态，直到接到通知或被中断为止。**在调用wait（）之前，线程必须要获得该对象的对象级别锁**，即只能在同步方法或同步块中调用wait（）方法。进入wait（）方法后，当前线程释放锁。在从wait（）返回前，线程与其他线程竞争重新获得锁。如果调用wait（）时，没有持有适当的锁，**则抛出IllegalMonitorStateException**，它是RuntimeException的一个子类，因此，不需要try-catch结构。

#### wait（long timeout）方法

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

wait（long timeout）方法是设置等待超时时间的，**如果在等待线程接到通知或被中断之前，已经超过了指定的毫秒数，则它通过竞争重新获得锁**，并从wait（long timeout）返回。

#### wait（long timeout, int nanos） 方法

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);

}

该方法导致当前线程等待，直到其他线程调用此对象的 notify() 方法或notifyAll()方法，或在指定已经过去的时间。此方法类似于 wait 方法的一个参数，但它允许更好地控制的时间等待一个通知放弃之前的量。

另外，需要知道的是 **wait（long timeout）方法和wait（long timeout, int nanos）方法**，如果设置了超时时间，当wait（）返回时，我们不能确定它是因为接到了通知还是因为超时而返回的，因为wait（）方法不会返回任何相关的信息。**但一般可以通过设置标志位来判断**，在notify之前改变标志位的值，在wait（）方法后读取该标志位的值来判断，当然为了保证notify不被遗漏，**我们还需要另外一个标志位来循环判断是否调用wait（）方法。**

#### notify 方法

public final native void notify();

notify方法用于唤醒正在等待当前对象监视器的线程，唤醒的线程是随机的。一般notify方法和wait方法配合使用来达到多线程同步的目的。

在一个线程被唤醒之后，线程必须先重新获取对象的监视器锁（线程调用对象的wait方法之后会让出对象的监视器锁），才可以继续执行。 一个线程在调用一个对象的notify方法之前必须获取到该对象的监视器(synchronized)，否则将抛出IllegalMonitorStateException异常。同样一个线程在调用一个对象的wait方法之前也必须获取到该对象的监视器。

#### notifyAll 方法

public final native void notifyAll();

notifyAll使所有原来在该对象上wait的线程统统退出wait的状态（即全部被唤醒，不再等待notify或notifyAll，但由于此时还没有获取到该对象锁，因此还不能继续往下执行），变成等待获取该对象上的锁，一旦该对象锁被释放（notifyAll线程退出调用了notifyAll的synchronized代码块的时候），他们就会去竞争。如果其中一个线程获得了该对象锁，它就会继续往下执行，在它退出synchronized代码块，释放锁后，其他的已经被唤醒的线程将会继续竞争获取该锁，一直进行下去，直到所有被唤醒的线程都执行完毕。

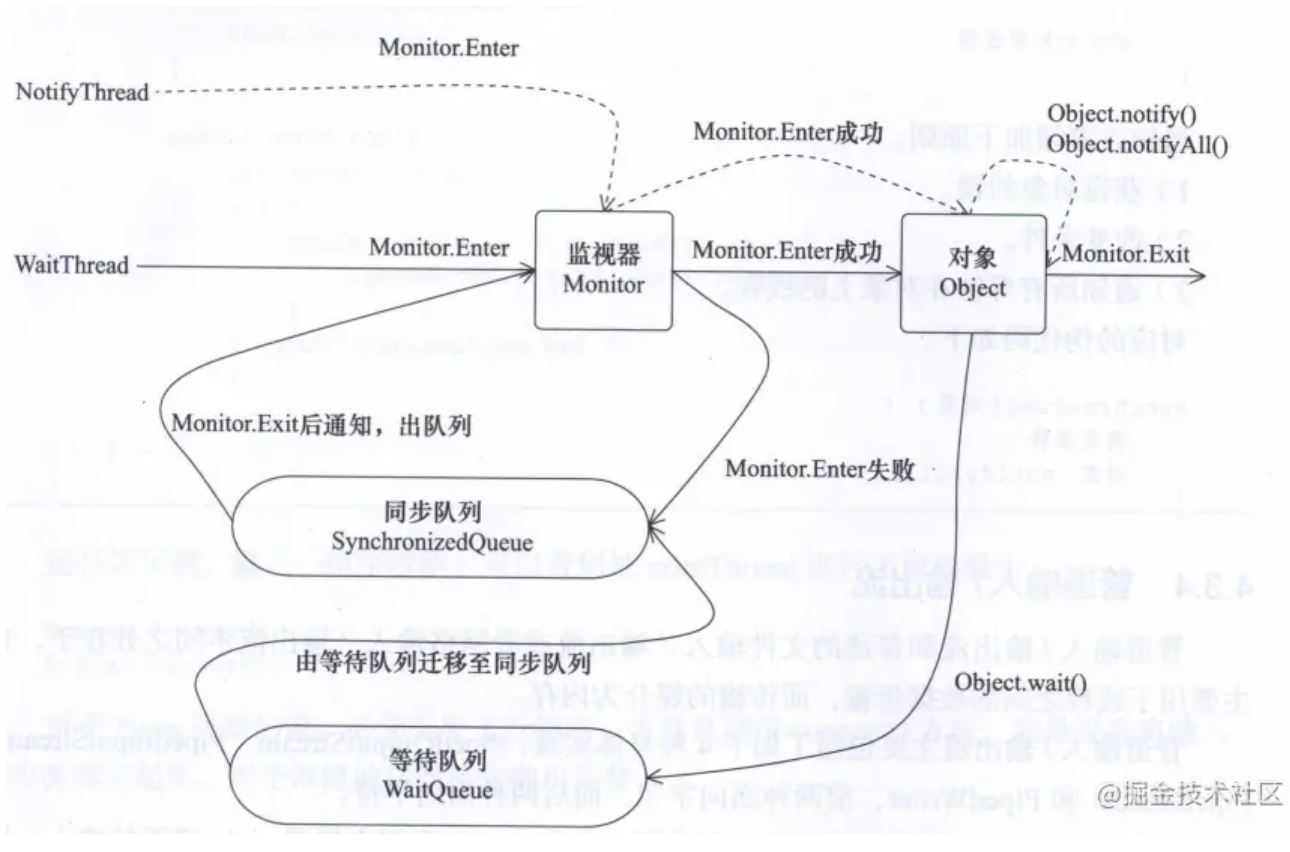
 使用wait（）、notify（）、notifyAll（）时需要先对调用对象加锁。

 调用wait（）方法后，线程状态有RUNNING变成WAITING，并将当前线程放置到对象的等待队列。

 notify（）或notifyAll（）方法调用后，等待线程依旧不会从wait（）返回，需要调用notify（）或notifyAll（）的线程释放锁之后，等待线程才有机会从wait（）返回。

 notify（）方法将等待队列中的一个等待线程从等待队列中移到同步队列中，而notifyAll（）方法则是将等待队列中所有线程全部移到同步队列，被移动的线程状态由WAITING变为BLOCKED。

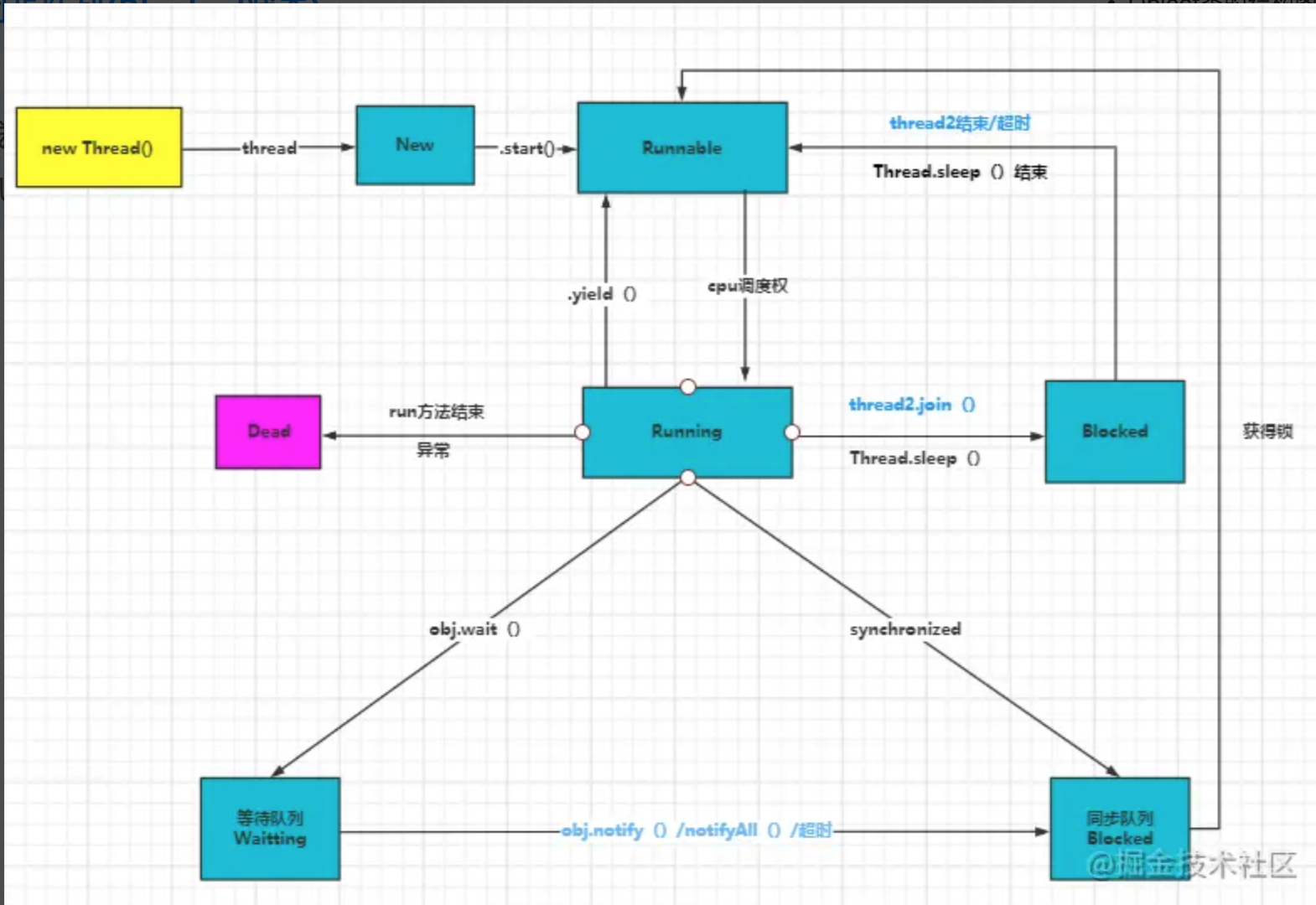
 从wait（）方法返回的前提是获得了调用对象的锁。



在**线程t1**调用**A对象**的wait()方法,会释放**t1持有的锁**，让线程t1进入**等待队列(Waiting状态)**

直到其他线程调用A对象的**notify()** 方法或**notifyAll()** 方法，线程t1进入**同步队列(Blocked状态)**

当线程t1获得锁后会进入**就绪状态Runnable**,获取CPU的调度权后会继续执行。



如果线程调用了对象的wait（）方法，那么线程便会处于该对象的等待池中，等待池中的线程不会去竞争该对象的锁。

当有线程调用了对象的notifyAll（）方法（唤醒所有wait线程）或notify（）方法（只随机唤醒一个wait线程），被唤醒的的线程便会进入该对象的锁池中，锁池中的线程会去竞争该对象锁。

优先级高的线程竞争到对象锁的概率大，假若某线程没有竞争到该对象锁，它还会留在锁池中，唯有线程再次调用wait（）方法，它才会重新回到等待池中。而竞争到对象锁的线程则继续往下执行，直到执行完了synchronized代码块，它会释放掉该对象锁，这时锁池中的线程会继续竞争该对象锁。

## 8.finalize方法

protected void finalize() throws Throwable { }

该方法用于释放资源。Java允许在类中定义一个名为finalize()的方法。它的工作原理是：**一旦垃圾回收器准备好释放对象占用的存储空间，将首先调用其finalize()方法**。并且在下一次垃圾回收动作发生时，才会真正回收对象占用的内存。

由于finalize方法不一定被执行，那么我们想清理某各类里打开的资源时，**则不要用finalize方法**。