Object 源码分析

位置：java.lang包

Object类是Java中最基本的类，是所有类的根。也就说，所有的类默认都会继承它，包括数组等，都要继承Object中的所有方法。Object类中大多数都是native方法，native就是本地方法，由关键native字修饰，这些方法不在java语言中实现，底层实现是的c/c++代码。主要的native方法如下：

```
//加载本地方法
private static native void registerNatives();
//获取对象的类型 Class
public final native Class<?> getClass();
//获取hash码
 public native int hashCode();
//对象拷贝
protected native Object clone() throws CloneNotSupportedException;

//跟线程有关的方法
//唤醒线程
public final native void notify();
//唤醒所有的线程
public final native void notifyAll();
//线程等待
public final native void wait(long timeout) throws InterruptedException;
```

private static native void registerNatives()

在Object类中，有static代码块，这个静态代码块调用了registerNatives()方法：

```
private static native void registerNatives();
static {
    registerNatives();
}
```

registerNatives方法的作用是加载和注册本地C、C++语言函数，将Java的本地方法与JVM底层的C、C++语言函数对应起来，是连接java语言与底层语言的桥梁，registerNatives方法在其他类中也可能存在，如Class类。Java类的本地方法对应C、C++函数的规则是Java_包名_方法名,包名以下划线分隔，Object类的registerNatives对应着C语言函数是Java_java_lang_Object_registerNatives，java_lang_Object是java全类名以下划线连接，registerNatives是Java中的方法，其中Java_java_lang_Object_registerNatives这个C语言函数如下：

```
static JNINativeMethod methods[] = {
    {"hashCode",    "()I",                    (void *)&JVM_IHashCode},
    {"wait",        "(J)V",                   (void *)&JVM_MonitorWait},
    {"notify",      "()V",                    (void *)&JVM_MonitorNotify},
    {"notifyAll",   "()V",                    (void *)&JVM_MonitorNotifyAll},
    {"clone",       "()Ljava/lang/Object;",   (void *)&JVM_Clone},
};

JNIEXPORT void JNICALL
Java_java_lang_Object_registerNatives(JNIEnv *env, jclass cls)
{
    (*env)->RegisterNatives(env, cls,
                              methods, sizeof(methods)/sizeof(methods[0]));
}
```

Java_java_lang_Object_registerNatives主要是注册Object类的hashCode、wait、notify、notifyAll、clone等方法，而getClass方法却不在这里加载。在Object类中，这些本地方法不是在Java层面实现的，所有在调用这些方法的时候，是调用了底层语言的具体实现。

在methods[]数组中，有多个Java本地方法对应JVM层面的函数，如Object中hashCode方法对应JVM中的JVM_IHashCode函数，返回的类型是()I，即返回的是整数类型。

public final native Class<?> getClass()

getClass()方法的作用是返回对象运行时的Class类型，java编译会将java类编译成以.class结尾的文件，这是编译生成的二进制字节码文件，用于JVM加载，Java跨平台是因为使用与平台无关的class二进制字节码文件，可以通过Jjava中的Class类获取对象的构造器、方法、属性、注解等相关信息：

```
Object object=new Object();
System.out.println(object.getClass());
Class<?> objectClass=object.getClass();
//构造器
Constructor<?>[] constructors=objectClass.getConstructors();
System.out.println("Object 类的构造器：");
for (Constructor constructor:constructors){
        System.out.print(constructor);
 }
//方法
Method[] methods=objectClass.getMethods();
System.out.println("Object 类的方法：");
for (Method method:methods){
      System.out.println(method);
}
//属性
Field[] fields=objectClass.getFields();
System.out.println("Object 类的属性：");
for (Field field:fields){
       System.out.println(field);
}
```

getClass方法对应的底层C语言函数为Java_java_lang_Object_getClass，具体实现为：

```
JNIEXPORT jclass JNICALL
Java_java_lang_Object_getClass(JNIEnv *env, jobject this)
{
    if (this == NULL) {
        JNU_ThrowNullPointerException(env, NULL);
        return 0;
    } else {
        return (*env)->GetObjectClass(env, this);
    }
}
```

当传入的对象this为null，直接抛出NullPointerException异常，否则调用GetObjectClass函数,GetObjectClass函数如下：

```
JNI_ENTRY(jclass, jni_GetObjectClass(JNIEnv *env, jobject obj))
  JNIWrapper("GetObjectClass");
#ifndef USDT2
  DTRACE_PROBE2(hotspot_jni, GetObjectClass__entry, env, obj);
#else /* USDT2 */
  HOTSPOT_JNI_GETOBJECTCLASS_ENTRY(env, obj);
#endif /* USDT2 */
//主要作用是根据java对象引用找到JVM中引用对象，将Java对象转换为JVM中的引用对象，然后调用
klass()方法找到元数据
  Klass* k = JNIHandles::resolve_non_null(obj)->klass();
  jclass ret =(jclass) JNIHandles::make_local(env, k->java_mirror());
#ifndef USDT2
  DTRACE_PROBE1(hotspot_jni, GetObjectClass__return, ret);
#else /* USDT2 */
  HOTSPOT_JNI_GETOBJECTCLASS_RETURN(
                                  ret);
#endif /* USDT2 */
  return ret;
JNI_END
```

上述的核心代码为:

```
Klass* k = JNIHandles::resolve_non_null(obj)->klass();
jclass ret =(jclass) JNIHandles::make_local(env, k->java_mirror());
```

resolve_non_null方法主要作用是根据java对象引用找到JVM中引用对象，将Java对象转换为JVM中的引用对象，然后调用klass()方法找到元数据，resolve_non_null的方法为：

```
inline oop JNIHandles::resolve_non_null(jobject handle) {
  assert(handle != NULL, "JNI handle should not be null");
  oop result = *(oop*)handle;
  assert(result != NULL, "Invalid value read from jni handle");
  assert(result != badJNIHandle, "Pointing to zapped jni handle area");
  // Don't let that private _deleted_handle object escape into the wild.
  assert(result != deleted_handle(), "Used a deleted global handle.");
  return result;
};
```

resolve_non_null方法的核心代码是oop result = *(oop*)handle;这句代码的意思是先将传入的Java对象转为oop实例，（(oop*)handle），然后再获取oop的指针，这个指针就是引用对象实例的地址（*(oop*)handle）。然后调用klass()获取对象实例所属的元数据Klass，Klass是指向Class类型的指针。

```
inline Klass* oopDesc::klass() const {
  if (UseCompressedClassPointers) {
    return Klass::decode_klass_not_null(_metadata._compressed_klass);
  } else {
  //返回元数据
    return _metadata._klass;
  }
}
```

jclass ret =(jclass) JNIHandles::make_local(env, k->java_mirror())分为两步，先调用Klass的java_mirror()方法，java_mirror方法的作用是返回Klass元数据的镜像（oop），对应着jjava/lang/Class类的实例，ava_mirror方法如下：

```
 // java/lang/Class instance mirroring this class
oop       _java_mirror;
oop java_mirror() const              { return _java_mirror; }
```

最后通过 JNIHandles::make_local处理oop，然后返回处理过后的oop,make_local的代码如下：

```
jobject JNIHandles::make_local(oop obj) {
  if (obj == NULL) {
      //返回null
    return NULL;                // ignore null handles
  } else {
     //获取当前线程
    Thread* thread = Thread::current();
    assert(Universe::heap()->is_in_reserved(obj), "sanity check");
     //利用当前线程活跃的handles对oop进行处理。
    return thread->active_handles()->allocate_handle(obj);
  }
}
```

在JNIHandles::make_local函数中，当传入的oop实例obj为空时，直接返回空，否则将处理过的结果。

public native int hashCode()

hashCode方法返回对象的哈希码，以整数形式返回。哈希表在java集合中如HashMap、HashSet中被广泛使用，主要作用是为了提高查询效率。hashCode方法一些规定/约定如下：

1. 在不修改对象的equals()方法时，同一个对象无论何时调用hashCode方法，每次调用hashCode方法必须返回相同的整数（哈希码）。此整数不需要在应用程序的一次执行与同一应用程序的另一次执行之间保持一致。
2. 如果两个对象相等（使用equals()方法判断），那么这两个对象调用hashCode方法必须返回相同的整数结果（哈希码）。
3. 如果两个对象不相等（使用equals()方法判断），两个对象调用hashCode方法返回的结果不一定不同，也就是不同的两个对象的hashCode方法返回的结果可以相同也可以不相同。但是不同对象返回的哈希码最好不同，这样在哈希表查询时效率会更高。

hashCode方法的使用：

```
Object o1=new Object();
Object o2=new Object();
Object o3=o1;
System.out.println("o1的hash的哈希码: "+ o1.hashCode());
System.out.println("o2的hash的哈希码: "+ o2.hashCode());
System.out.println("o3的hash的哈希码: "+ o3.hashCode());
//我机器上的结果，不同机器结果不同
o1的hash的哈希码: 399573350
o2的hash的哈希码: 463345942
o3的hash的哈希码: 399573350
```

o1和o2是不同的对象，hashCode方法产生不同的哈希码， o1与o3是不同的对象，它们的哈希码不一样。hashCode的底层C++实现(参考了https://juejin.im/entry/5968876df265da6c232898c2）：

```
static inline intptr_t get_next_hash(Thread * Self, oop obj) {
  intptr_t value = 0 ;
  if (hashCode == 0) {
      //OpenJdk 6 &7的默认实现,此类方案返回一个Park-Miller伪随机数生成器生成的随机数
```

```
     value = os::random() ;
  } else
  if (hashCode == 1) {
      //此类方案将对象的内存地址，做移位运算后与一个随机数进行异或得到结果
     intptr_t addrBits = cast_from_oop<intptr_t>(obj) >> 3 ;
     value = addrBits ^ (addrBits >> 5) ^ GVars.stwRandom ;
  } else
  if (hashCode == 2) {
      //此类方案将对象的内存地址，做移位运算后与一个随机数进行异或得到结果
     value = 1 ;            // for sensitivity testing
  } else
  if (hashCode == 3) {
      //此类方案返回一个自增序列的当前值
     value = ++GVars.hcSequence ;
  } else
  if (hashCode == 4) {
      //此类方案返回当前对象的内存地址
     value = cast_from_oop<intptr_t>(obj) ;
  } else {
     //OpenJdk 8 默认hashCode的计算方法
     //通过和当前线程有关的一个随机数+三个确定值
      //运用Marsaglia's xorshift scheme随机数算法得到的一个随机数
     unsigned t = Self->_hashStateX ;
     t ^= (t << 11) ;
     Self->_hashStateX = Self->_hashStateY ;
     Self->_hashStateY = Self->_hashStateZ ;
     Self->_hashStateZ = Self->_hashStateW ;
     unsigned v = Self->_hashStateW ;
     v = (v ^ (v >> 19)) ^ (t ^ (t >> 8)) ;
     Self->_hashStateW = v ;
     value = v ;
  }

  value &= markOopDesc::hash_mask;
  if (value == 0) value = 0xBAD ;
  assert (value != markOopDesc::no_hash, "invariant") ;
  TEVENT (hashCode: GENERATE) ;
  return value;
}
```

get_next_hash函数根据hashCode的值，来采用不同的hashCode计算方法，当hashCode=0时，是OpenJDK6、7的默认实现方法，此类方案返回一个Park-Miller伪随机数生成器生成的随机数；当hashCode=1时，通过将对象的内存地址，做移位运算后与一个随机数进行异或得到结果；当hashCode == 3，返回一个自增序列的当前值；当hashCode == 4时，返回当前对象的内存地址；当hashCode 为其他值时，是openJDK8 的hashCode 方法的默认实现。

openJDK8 的默认hashCode的计算方法是通过和当前线程有关的一个随机数+三个确定值，运用Marsaglia's xorshift scheme随机数算法得到的一个随机数。xorshift算法是通过移位和与或计算，能够在计算机上以极快的速度生成伪随机数序列。算法如下：

```
unsigned long xor128(){
    static unsigned long x=123456789,y=362436069,z=521288629,w=88675123;
    unsigned long t;
    t=(xˆ(x<<11));x=y;y=z;z=w;
    return( w=(wˆ(w>>19))ˆ(tˆ(t>>8)) );
}
```

Self->_hashStateX 是随机数、Self->_hashStateY 、Self->_hashStateZ、Self->_hashStateW 是三个确认的数，分别对应xorshift 算法的x、y、z、w。

在启动JVM时，可以通过设置**-XX:hashCode**参数，改变默认的hashCode的计算方式。

public boolean equals(Object obj)

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

equals方法判断两个对象是否相等，在这个方法中，直接用this==obj进行比较，返回this和obj比较的结果，“==“符号是比较两个对象是否是同一个对象，比较的是两个对象内存地址是否相等。子类在继承Object类的时候，一般需要重写equals方法，如果不重写，那么就默认父类Object的方法，在JDK源码中，很多对象都重写equals方法，比如String类。

对于非空的对象引用，equals方法有几个性质：

- 自反性：对于任何非空对象x和y，如果x.equals(y) 的结果为true，那么y.equals(x) 的结果也为true。
- 传递性：对于任何非空对象x、y和z，如果x.equals(y)和y.equals(z)的结果都为true，那么x.equals(z)的结果也为true。
- 一致性：对于任何非空对象x和y，如果未修改equals方法，多次调用equals方法结果始终为true或者false，不能这次返回true，下次返回false。
- 对于任何非空对象x，x.equals(null)的结果是false。

子类在继承Object，重写equals方法时，必须重写hashCode方法，在hashCode约定中，如果两个对象相等，hashCode必须返回相同的哈希码。很多时候，子类在重写父类Object的equals方法时，往往会忘了重写hashCode，当重写了equals但没有重写hashCode方法时，那么该子类无法结合java集合正常运行，因为java集合如HashMap、HashSet等都是基于哈希码进行存储的。

protected native Object clone() throws CloneNotSupportedException;

clone()本地方法，创建并返回此对象的副本，该方法使用protected 修饰，Object不能直接调用clone()方法，会编译错误：

```
 Object o=new Object();
 //编译错误
 Object o1=  o.clone();
```

如果想要使用clone()方法，子类必须重写这个方法，并用public修饰。如果子类没有实现clone(),子类默认调用父类的clone方法，如下：

```
public class ObjectCloneTest {
    public static void main(String [] args) throws CloneNotSupportedException {
        ObjectCloneTest o=new ObjectCloneTest();
        ObjectCloneTest cloneTest= (ObjectCloneTest) o.clone();
        System.out.println(cloneTest);
    }
}
```

但是运行时发生异常：

```
Exception in thread "main" java.lang.CloneNotSupportedException:
com.lingheng.java.source.ObjectCloneTest
    at java.lang.Object.clone(Native Method)
    at com.lingheng.java.source.ObjectCloneTest.main(ObjectCloneTest.java:12)
```

如果子没有实现Cloneable接口，当子类调用clone()方法时，会抛出CloneNotSupportedException异常。当子类实现Cloneable接口，程序运行会成功：

```
//实现Cloneable接口
public class ObjectCloneTest implements Cloneable{

    public static void main(String [] args) throws CloneNotSupportedException {
        ObjectCloneTest o=new ObjectCloneTest();
        ObjectCloneTest cloneTest= (ObjectCloneTest) o.clone();
        System.out.println("打印cloneTest: "+cloneTest);
    }

}
//结果
打印cloneTest: com.lingheng.java.source.ObjectCloneTest@6d6f6e28
```

所以，想要使用clone()方法，除了继承Object外（默认继承），还需要实现Cloneable接口。所有的数组默认是实现了Cloneable接口的，数组的clone()返回数组类型:

```
public void arrayClone(){
    //字符串数组
    String[] s=new String[]{"hello","world"};
    System.out.println(s);
    System.out.println(s.clone());

     //整数数组
     int[] num=new int[]{1,3};
     System.out.println(num);
     System.out.println(num.clone());
}
//结果
[Ljava.lang.String;@17d10166
[Ljava.lang.String;@1b9e1916
[I@ba8a1dc
[I@4f8e5cde
```

对于任何对象x，具有以下几个约定：

1. x.clone() != x 为true
2. x.clone().getClass() == x.getClass() 为true，但这不是必须满足的要求，也就是说可能是false。按照约定，返回的对象应该通过调用super.clone来获得。如果一个类和它所有的超类（除了Object）遵循这个约定，那就会x.clone().getClass() == x.getClass()。
3. x.clone().equals(x)通常情况下为true，这不是必须满足的要求。

对于约定1，因为拷贝是创建一个新的对象，所以拷贝的对象和原对象是不相等的。

对于约定2，这个比较好理解，一个类和它所有的超类的clone方法都调用super.clone()获取返回对象，所以这个类的Class个原对象的Claas是一样的。clone返回的类型是Object类，也就是说，在clone中可以返回只要是Object的子类就可以了，这样的话，x.clone().getClass() == x.getClass()的结果为false；

对于约定3，这不是必须满足的条件，当在clone中改变了对象的属性值，那么x.clone().equals(x)的结果就可能不是true了。

分析了Java层面的clone的约定，我们从JVM层面看看clone底层的实现，在registerNatives函数中会加载clone的JVM实现JVM_Clone，JVM_Clone的实现如下：

```
//JVM_Clone 的实现
JVM_ENTRY(jobject, JVM_Clone(JNIEnv* env, jobject handle))
  JVMWrapper("JVM_Clone");
  Handle obj(THREAD, JNIHandles::resolve_non_null(handle));
  const KlassHandle klass (THREAD, obj->klass());
  JvmtiVMObjectAllocEventCollector oam;

#ifdef ASSERT
  // Just checking that the cloneable flag is set correct
//如果是传入的是数组
  if (obj->is_array()) {
      //所有的数组都默认实现了Cloneable接口
    guarantee(klass->is_cloneable(), "all arrays are cloneable");
  } else {
      //是否是实例
    guarantee(obj->is_instance(), "should be instanceOop");
    bool cloneable = klass->is_subtype_of(SystemDictionary::Cloneable_klass());
      //判断是否是合法的cloneable
    guarantee(cloneable == klass->is_cloneable(), "incorrect cloneable flag");
  }
#endif
----------------------------------分割线1----------------------------------------
---------
// Check if class of obj supports the Cloneable interface.
 // All arrays are considered to be cloneable (See JLS 20.1.5)
    //如果没有实现Cloneable接口，抛出CloneNotSupportedException异常
  if (!klass->is_cloneable()) {
    ResourceMark rm(THREAD);
    THROW_MSG_0(vmSymbols::java_lang_CloneNotSupportedException(), klass-
>external_name());
  }

  // Make shallow object copy
    //浅拷贝
  const int size = obj->size();
  oop new_obj = NULL;
  if (obj->is_array()) {//数组
      //数组的长度
    const int length = ((arrayOop)obj())->length();
     //申请内存，创建新的数组
    new_obj = CollectedHeap::array_allocate(klass, size, length, CHECK_NULL);
  } else {
    //申请内存，创建新的对象
    new_obj = CollectedHeap::obj_allocate(klass, size, CHECK_NULL);
  }
----------------------------------分割线2----------------------------------------
---------
 // 4839641 (4840070): We must do an oop-atomic copy, because if another thread
  // is modifying a reference field in the clonee, a non-oop-atomic copy might
  // be suspended in the middle of copying the pointer and end up with parts
  // of two different pointers in the field.  Subsequent dereferences will
crash.
  // 4846409: an oop-copy of objects with long or double fields or arrays of
same
  // won't copy the longs/doubles atomically in 32-bit vm's, so we copy jlongs
instead
  // of oops.  We know objects are aligned on a minimum of an jlong boundary.
  // The same is true of StubRoutines::object_copy and the various oop_copy
  // variants, and of the code generated by the inline_native_clone intrinsic.
  assert(MinObjAlignmentInBytes >= BytesPerLong, "objects misaligned");
  //原子性拷贝，可能另外一个线程会修改clone引用的字段
  Copy::conjoint_jlongs_atomic((jlong*)obj(), (jlong*)new_obj,
                               (size_t)align_object_size(size) /
HeapWordsPerLong);
  // Clear the header
  new_obj->init_mark();

  // Store check (mark entire object and let gc sort it out)
  BarrierSet* bs = Universe::heap()->barrier_set();
  assert(bs->has_write_region_opt(), "Barrier set does not have write_region");
  bs->write_region(MemRegion((HeapWord*)new_obj, size));

  // Caution: this involves a java upcall, so the clone should be
  // "gc-robust" by this stage.
  //当java子类实现了finalize方法，已经调用finalize，注册finalizer方法
  if (klass->has_finalizer()) {
    assert(obj->is_instance(), "should be instanceOop");
    new_obj = InstanceKlass::register_finalizer(instanceOop(new_obj),
CHECK_NULL);
  }
    //创建拷贝的对象返回
  return JNIHandles::make_local(env, oop(new_obj));
JVM_END
```

JVM_Clone的实现过程比较长，我将分割成三部分，分割线1上部分主要是检查拷贝的一些标志是否正确，判断传入的是数组还是对象，在上面我讲过所有的数组都默认实现了Cloneable接口，在这里就可以证明了。然后还判断了传入的对象是否继承Cloneable接口，没有继承的话，就是非法的，这是第一段逻辑。

分割线中间部分的逻辑，主要是申请内存，供拷贝的时候使用。首先先判断是否实现Cloneable接口，如果没有抛出CloneNotSupportedException异常，然后根据传入的是数组还是对象，调用不同的方法进行申请不同大小的内存。代码的有段注释是”Make shallow object copy“，就是说这里的拷贝是浅拷贝。

最后的逻辑是，做一些检查，最后返回新创建的拷贝对象。因为对象的属性可能被另外一个线程改变，所以进行的是原子性的拷贝，最后创建拷贝的对象进行返回。这些源码的注释很详细，有兴趣的可以具体看看其他的一些逻辑。

public String toString()

toString()返回对象的字符串表示形式，最好所有Object的子类都重写这个方法，Object类的toString方法返回的是"类名@哈希码的十六进制，代码实现如下：

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

当我们在代码输入使用System.out.println输入对象的时候，会调用这个对象的toString方法，返回的是toString的字符串。一般在重写的toString的过程，返回的字符串要易读的。

notify、notifyAll、wait

这三个方法放在一起讲，都是native修饰的本地方法，另外这三个方法被final修饰，说明这三个方法是不能重写。

notify：唤醒任意一个等待的线程

notifyAll：唤醒所有等待的线程

wait：释放锁，线程进入等待。

这几个方法主要用于线程之间的通信，特别适合与生产者和消费者的场景，下面用消费者和生产者的例子看看这几个方法的使用：

```
//生产者
public class Producer implements Runnable{

    //产品容器
    private final List<Integer> container;
    //产品容器的大小
    private final int size=5;

    public Producer( List<Integer> container){
        this.container=container;
    }

    private void produce() throws InterruptedException {
        synchronized (container){
            //判断容器是否已满
            while (container.size()==size){
                System.out.println("容器已满，暂定生产。");
                container.wait();
            }

            Random random = new Random();
            int num = random.nextInt(10);
            //模拟1秒生产一个产品
            Thread.sleep(1000);
            System.out.println(Thread.currentThread().getName()+"时间"+new
Date()+" 生产产品：" + num);
            container.add(num);
            //生产一个产品就可以通知消费者消费了
            container.notifyAll();
        }
    }

    public void run() {

        while (true){
            try {
                produce();
            } catch (InterruptedException e) {
                System.out.println("生产机器异常");
                e.printStackTrace();
            }
        }

    }
}
```

生产者每秒生产一个产品，然后通知消费者消费，当容器满了，就进行等待消费者唤醒。

```java
//消费者
public class Consumer implements Runnable{

    //消费容器
    private final List<Integer> container;

    public Consumer(List<Integer> container){
        this.container=container;
    }

    private void consume() throws InterruptedException {
        synchronized(container){
            while (container.isEmpty()){
                System.out.println("没有可消费的产品");
                //等待
                container.wait();
            }

            //消费产品
            Thread.sleep(1000);
            Integer num=container.remove(0);
            System.out.println(Thread.currentThread().getName()+"时间"+new
Date()+" 消费产品: "+num);
            //消费一个就可以通知生产者消费了
            container.notifyAll();
        }
    }

    public void run() {
        while (true){
            try {
                consume();
            } catch (InterruptedException e) {
                System.out.println("消费错误");
                e.printStackTrace();
            }
        }
    }
}
```

消费者每秒消费一个产品，当容器减少一个产品就可以通知生产者生产产品了，当容器为空时，进行等待生产者唤醒。

```
//生产消费过程
public class Main {

    public static void main(String[] args){
        List<Integer> container = new ArrayList<Integer>();
        Thread producer = new Thread(new Producer(container));
        Thread consumer = new Thread(new Consumer(container));
        producer.start();
        consumer.start();
    }
}
```

启动一个消费者和一个生产者，也可以启动多个消费者和多个生产者，消费者和生产者之间共用容器，上述消费者的消费速度和生产者的生产速度都是每秒一个产品，可以改变消费者消费的速度和生产者生产的速度观察程序运行的结果。

notify、notifyAll、wait一般配合着关键synchronized 一起使用，这三个方法在synchronized 代码块中使用，否则会抛出IllegalMonitorStateException。当它们在synchronized 代码块中执行，说明当前线程一定获得了锁。优先级高的线程竞争到对象锁的概率大。notify只会唤醒一个等待的线程，而notifyAll唤醒所有的线程，唤醒所有的线程，并不意味着所有的线程会立刻执行，这些被唤醒的锁还需要竞争锁。

既然notify、notifyAll、wait这几个方法都是本地方法，那JVM层面是怎么实现的？先来看看wait方法对应的本地方法是JVM_MonitorWait，JVM_MonitorWait的实现如下：

```
JVM_ENTRY(void, JVM_MonitorWait(JNIEnv* env, jobject handle, jlong ms))
  JVMWrapper("JVM_MonitorWait");
    //resolve_non_null将传入的对象强转为oop
  Handle obj(THREAD, JNIHandles::resolve_non_null(handle));
  JavaThreadInObjectWaitState jtiows(thread, ms != 0);
    //是否已经交于monitor监控等待了
  if (JvmtiExport::should_post_monitor_wait()) {
      //触发等待事件 JVMTI [%s] montior wait event triggered
      // 将等待事件发送给线程 JVMTI [%s] monitor wait event sent
    JvmtiExport::post_monitor_wait((JavaThread *)THREAD, (oop)obj(), ms);
  }
//重点分析这句
  ObjectSynchronizer::wait(obj, ms, CHECK);
JVM_END
```

monitor指的是监控器，monitor的作用监视线程，保证在同一时间内只有一个线程可以访问共享的数据和代码（加锁对象的数据），其他线程需要访问共享数据就先进行等待，等这个线程释放锁以后，其他线程才有机会进行访问。monitor有wait set（等待池）和entry set（实例池），wait set存放处于wait状态的线程队列，entry set存放处于等待锁block状态的线程队列。

JVM_MonitorWait方法中首先判断下是否已经交于monitor进行监控等待了，如果是的话，调用post_monitor_wait方法，这个方法主要作用触发等待事件，将等待事件发送给线程。

JVM_MonitorWait的重点在最后一句代码，这句代码才是正真进行wait方法的调用，ObjectSynchronizer::wait的代码如下：

```
//  Wait/Notify/NotifyAll
//注意:必须使用重量型monitor监控器来处理wait()
// NOTE: must use heavy weight monitor to handle wait()
void ObjectSynchronizer::wait(Handle obj, jlong millis, TRAPS) {
    //使用偏向锁
```

```
  if (UseBiasedLocking) {
      //消除偏向锁
    BiasedLocking::revoke_and_rebias(obj, false, THREAD);
    assert(!obj->mark()->has_bias_pattern(), "biases should be revoked by now");
  }
  if (millis < 0) {
    TEVENT (wait - throw IAX) ;
      //时间小于0，抛出IllegalArgumentException异常
    THROW_MSG(vmSymbols::java_lang_IllegalArgumentException(), "timeout value is
negative");
  }
    =======================================分割线
==================================
   //获取监视器
  ObjectMonitor* monitor = ObjectSynchronizer::inflate(THREAD, obj());
  DTRACE_MONITOR_WAIT_PROBE(monitor, obj(), THREAD, millis);
//执行等待方法
  monitor->wait(millis, true, THREAD);

  /* This dummy call is in place to get around dtrace bug 6254741.  Once
     that's fixed we can uncomment the following line and remove the call */
  // DTRACE_MONITOR_PROBE(waited, monitor, obj(), THREAD);
  dtrace_waited_probe(monitor, obj, THREAD);
}
```

分割线上部分代码主要是消除偏向锁、判断等待的时间是否小于0，小于0的话，抛出抛出IllegalArgumentException异常。这个方法重要代码就在分割线的下部分，ObjectMonitor就是监视器对象，ObjectMonitor的结构如下：

```
ObjectMonitor() {
  _header       = NULL;
  _count        = 0;
  _waiters      = 0,
  _recursions   = 0; //重试的次数
  _object       = NULL;
  _owner        = NULL; //指向持有ObjectMonitor对象的线程
  _WaitSet      = NULL; //存放所有wait状态的线程的对象
  _WaitSetLock  = 0 ;
  _Responsible  = NULL ;
  _succ         = NULL ;
  _cxq          = NULL ;//阻塞在Entry最近可达的线程列表，该列表其实是waitNode所构成的线
程代理
  FreeNext      = NULL ;
  _EntryList    = NULL ;//存放处于等待锁block状态的线程队列
  _SpinFreq     = 0 ;
  _SpinClock    = 0 ;
  OwnerIsThread = 0 ;
  _previous_owner_tid = 0;
}
```

_WaitSet就是monitor的等待池，存放处于wait状态的线程队列，_EntryList就是实例池，存放处于等待锁block状态的线程队列。_cxq阻塞在Entry最近可达的线程列表，该列表其实是waitNode所构成的线程代理， _owner 是指向持有ObjectMonitor对象，ObjectSynchronizer::wait方法首先调用ObjectSynchronizer::inflate获取到monitor以后才执行monitor的wait方法。由于monitor的wait方法的源码如下：

```
// Wait/Notify/NotifyAll
//
// Note: a subset of changes to ObjectMonitor::wait()
// will need to be replicated in complete_exit above
void ObjectMonitor::wait(jlong millis, bool interruptible, TRAPS) {
    //获取线程
   Thread * const Self = THREAD ;
    //断言是否是java线程
   assert(Self->is_Java_thread(), "Must be Java thread!");
    //强转为java线程
   JavaThread *jt = (JavaThread *)THREAD;

    //一些全局初始化工作、
   DeferredInitialize () ;

   // Throw IMSX or IEX.
    //检查是否拥有monitor（监视器）
   CHECK_OWNER();

   EventJavaMonitorWait event;

   // check for a pending interrupt 检查线程是否中断和挂起
   if (interruptible && Thread::is_interrupted(Self, true) &&
!HAS_PENDING_EXCEPTION) {
     // post monitor waited event.  Note that this is past-tense, we are done
waiting.
       //递交给monitor等待事件
     if (JvmtiExport::should_post_monitor_waited()) {
        // Note: 'false' parameter is passed here because the
        // wait was not timed out due to thread interrupt.
        JvmtiExport::post_monitor_waited(jt, this, false);
     }
     if (event.should_commit()) {
       post_monitor_wait_event(&event, 0, millis, false);
     }
       //监听到异常事件
     TEVENT (Wait - Throw IEX) ;
     THROW(vmSymbols::java_lang_InterruptedException());
     return ;
   }

   TEVENT (Wait) ;

   assert (Self->_Stalled == 0, "invariant") ;
   Self->_Stalled = intptr_t(this) ;
    //设置当前的等待monitor
   jt->set_current_waiting_monitor(this);

   // create a node to be put into the queue
   // Critically, after we reset() the event but prior to park(), we must check
   // for a pending interrupt.
    //将当前线程包装为ObjectWaiter
   ObjectWaiter node(Self);
    //状态设置为TS_WAIT
   node.TState = ObjectWaiter::TS_WAIT ;
   Self->_ParkEvent->reset() ;
   OrderAccess::fence();          // ST into Event; membar ; LD interrupted-flag
   // Enter the waiting queue, which is a circular doubly linked list in this
case
   // but it could be a priority queue or any data structure.
   // _WaitSetLock protects the wait queue.  Normally the wait queue is accessed
only
   // by the the owner of the monitor *except* in the case where park()
   // returns because of a timeout of interrupt.  Contention is exceptionally
rare
   // so we use a simple spin-lock instead of a heavier-weight blocking lock.

   Thread::SpinAcquire (&_WaitSetLock, "WaitSet - add") ;
    //将ObjectWaiter对象放进等待池（wait set）中
   AddWaiter (&node) ;
   Thread::SpinRelease (&_WaitSetLock) ;

   if ((SyncFlags & 4) == 0) {
      _Responsible = NULL ;
   }
    //保存旧的重试次数
   intptr_t save = _recursions; // record the old recursion count
    //增加ObjectWaiter的数量
   _waiters++;                  // increment the number of waiters
   _recursions = 0;             // set the recursion level to be 1
   exit (true, Self) ;                    // exit the monitor
   guarantee (_owner != Self, "invariant") ;

   // As soon as the ObjectMonitor's ownership is dropped in the exit()
   // call above, another thread can enter() the ObjectMonitor, do the
   // notify(), and exit() the ObjectMonitor. If the other thread's
   // exit() call chooses this thread as the successor and the unpark()
   // call happens to occur while this thread is posting a
   // MONITOR_CONTENDED_EXIT event, then we run the risk of the event
   // handler using RawMonitors and consuming the unpark().
   //
   // To avoid the problem, we re-post the event. This does no harm
   // even if the original unpark() was not consumed because we are the
   // chosen successor for this monitor.
   if (node._notified != 0 && _succ == Self) {
      node._event->unpark();
   }

   // The thread is on the WaitSet list - now park() it.
   // On MP systems it's conceivable that a brief spin before we park
   // could be profitable.
   //
   // TODO-FIXME: change the following logic to a loop of the form
   //   while (!timeout && !interrupted && _notified == 0) park()

   int ret = OS_OK ;
   int WasNotified = 0 ;
   { // State transition wrappers
     OSThread* osthread = Self->osthread();
     OSThreadWaitState osts(osthread, true);
     {
       ThreadBlockInVM tbivm(jt);
       // Thread is in thread_blocked state and oop access is unsafe.
       jt->set_suspend_equivalent();
       if (interruptible && (Thread::is_interrupted(THREAD, false) ||
HAS_PENDING_EXCEPTION)) {
           // Intentionally empty
       } else
       if (node._notified == 0) {
           //当前线程通过park()方法开始挂起(suspend)
         if (millis <= 0) {
            Self->_ParkEvent->park () ;
         } else {
            ret = Self->_ParkEvent->park (millis) ;
         }
       }

       // were we externally suspended while we were waiting?
       if (ExitSuspendEquivalent (jt)) {
          // TODO-FIXME: add -- if succ == Self then succ = null.
          jt->java_suspend_self();
       }

     } // Exit thread safepoint: transition _thread_blocked -> _thread_in_vm


     // Node may be on the WaitSet, the EntryList (or cxq), or in transition
     // from the WaitSet to the EntryList.
     // See if we need to remove Node from the WaitSet.
     // We use double-checked locking to avoid grabbing _WaitSetLock
     // if the thread is not on the wait queue.
     //
     // Note that we don't need a fence before the fetch of TState.
     // In the worst case we'll fetch a old-stale value of TS_WAIT previously
     // written by the is thread. (perhaps the fetch might even be satisfied
     // by a look-aside into the processor's own store buffer, although given
     // the length of the code path between the prior ST and this load that's
     // highly unlikely).  If the following LD fetches a stale TS_WAIT value
     // then we'll acquire the lock and then re-fetch a fresh TState value.
     // That is, we fail toward safety.

       //当node的状态为TS_WAIT时，从WaitSet中删除
     if (node.TState == ObjectWaiter::TS_WAIT) {
         Thread::SpinAcquire (&_WaitSetLock, "WaitSet - unlink") ;
         if (node.TState == ObjectWaiter::TS_WAIT) {
            DequeueSpecificWaiter (&node) ;       // unlink from WaitSet
            assert(node._notified == 0, "invariant");
             //node状态更改为TS_RUN
            node.TState = ObjectWaiter::TS_RUN ;
         }
         Thread::SpinRelease (&_WaitSetLock) ;
     }

     // The thread is now either on off-list (TS_RUN),
     // on the EntryList (TS_ENTER), or on the cxq (TS_CXQ).
     // The Node's TState variable is stable from the perspective of this
thread.
     // No other threads will asynchronously modify TState.
     guarantee (node.TState != ObjectWaiter::TS_WAIT, "invariant") ;
     OrderAccess::loadload() ;
     if (_succ == Self) _succ = NULL ;
     WasNotified = node._notified ;
     // Reentry phase -- reacquire the monitor.
     // re-enter contended monitor after object.wait().
     // retain OBJECT_WAIT state until re-enter successfully completes
     // Thread state is thread_in_vm and oop access is again safe,
     // although the raw address of the object may have changed.
     // (Don't cache naked oops over safepoints, of course).

     // post monitor waited event. Note that this is past-tense, we are done
waiting.
     if (JvmtiExport::should_post_monitor_waited()) {
       JvmtiExport::post_monitor_waited(jt, this, ret == OS_TIMEOUT);
     }

     if (event.should_commit()) {
       post_monitor_wait_event(&event, node._notifier_tid, millis, ret ==
OS_TIMEOUT);
     }

     OrderAccess::fence() ;

     assert (Self->_Stalled != 0, "invariant") ;
     Self->_Stalled = 0 ;

     assert (_owner != Self, "invariant") ;
     ObjectWaiter::TStates v = node.TState ;
     if (v == ObjectWaiter::TS_RUN) {
         enter (Self) ;
     } else {
         guarantee (v == ObjectWaiter::TS_ENTER || v == ObjectWaiter::TS_CXQ,
"invariant") ;
         ReenterI (Self, &node) ;
         node.wait_reenter_end(this);
     }

     // Self has reacquired the lock.
     // Lifecycle - the node representing Self must not appear on any queues.
     // Node is about to go out-of-scope, but even if it were immortal we
wouldn't
     // want residual elements associated with this thread left on any lists.
     guarantee (node.TState == ObjectWaiter::TS_RUN, "invariant") ;
     assert    (_owner == Self, "invariant") ;
     assert    (_succ != Self , "invariant") ;
   } // OSThreadWaitState()

   jt->set_current_waiting_monitor(NULL);

   guarantee (_recursions == 0, "invariant") ;
   _recursions = save;     // restore the old recursion count
   _waiters--;             // decrement the number of waiters

   // Verify a few postconditions
   assert (_owner == Self       , "invariant") ;
   assert (_succ  != Self       , "invariant") ;
   assert (((oop)(object()))->mark() == markOopDesc::encode(this), "invariant")
;

   if (SyncFlags & 32) {
      OrderAccess::fence() ;
   }

   // check if the notification happened
   if (!WasNotified) {
     // no, it could be timeout or Thread.interrupt() or both
     // check for interrupt event, otherwise it is timeout
     if (interruptible && Thread::is_interrupted(Self, true) &&
!HAS_PENDING_EXCEPTION) {
        TEVENT (Wait - throw IEX from epilog) ;
        THROW(vmSymbols::java_lang_InterruptedException());
     }
   }

   // NOTE: Spurious wake up will be consider as timeout.
   // Monitor notify has precedence over thread interrupt.
}
```

ObjectMonitor::wait方法的作用主要作用为：

1. 将当前线程包装为ObjectWaiter，状态设置为TS_WAIT，ObjectWaiter的结构可以参考分析下面分析ObjectSynchronizer::notify的内容。
2. 执行 AddWaiter (&node) ，将ObjectWaiter放进等待池（wait set）中，即_WaitSet，_WaitSet是ObjectWaiter的一个队列。AddWaiter方法就是将ObjectWaiter放进_WaitSet队尾中。
3. 将当前线程挂起，在上述源码中并没与释放锁。也就是释放锁的工作不在方法内。

其他逻辑，源码中有详细的注释，感兴趣的可以直接深入下。ObjectMonitor::AddWaiter作用是将线程加入等待池（wait set）中，ObjectMonitor::AddWaiter的代码为：

```
inline void ObjectMonitor::AddWaiter(ObjectWaiter* node) {
  assert(node != NULL, "should not dequeue NULL node");
  assert(node->_prev == NULL, "node already in list");
  assert(node->_next == NULL, "node already in list");
  // put node at end of queue (circular doubly linked list)
  if (_WaitSet == NULL) {
    _WaitSet = node;
    node->_prev = node;
    node->_next = node;
  } else {
    ObjectWaiter* head = _WaitSet ;
    ObjectWaiter* tail = head->_prev;
    assert(tail->_next == head, "invariant check");
    tail->_next = node;
    head->_prev = node;
    node->_next = head;
    node->_prev = tail;
  }
}
```

当_WaitSet为空，放在_WaitSet的头部，_WaitSet的_prev和_next都指向node，当_WaitSet不为空时，将node放在_WaitSet头部。

notify方法对应的JVM层面的函数是JVM_MonitorNotify，JVM_MonitorNotify的源码为：

```
JVM_ENTRY(void, JVM_MonitorNotify(JNIEnv* env, jobject handle))
  JVMWrapper("JVM_MonitorNotify");
    //转成oop，oop表示普通对象
  Handle obj(THREAD, JNIHandles::resolve_non_null(handle));
   //调用notify方法
  ObjectSynchronizer::notify(obj, CHECK);
JVM_END
```

首先将传入的对象转换成oop，上述源码中，最重要的是调用了 ObjectSynchronizer::notify进行唤醒等待的线程， ObjectSynchronizer::notify的源码如下：

```
void ObjectSynchronizer::notify(Handle obj, TRAPS) {
    //是否使用偏向锁
 if (UseBiasedLocking) {
     //消除偏向锁
    BiasedLocking::revoke_and_rebias(obj, false, THREAD);
    assert(!obj->mark()->has_bias_pattern(), "biases should be revoked by now");
  }

  markOop mark = obj->mark();
    //如果已经获取了锁和获取了监视器
  if (mark->has_locker() && THREAD->is_lock_owned((address)mark->locker())) {
    return;
  }
    //获取monitor（监视器），监视THREAD，然后调用notify方法
  ObjectSynchronizer::inflate(THREAD, obj())->notify(THREAD);
}
```

ObjectSynchronizer::notify方法先判断是否使用偏向锁，如果使用了就消除偏向锁。ObjectSynchronizer::inflate是获取monitor，在调用wait、notify、notifyAll方法的时候，首先需要获取monitor（监视器），获取了monitor可以看成是获取了锁，monitor相当于是边界，没有monitor监控的线程就不能进来monitor中。获取了monitor，然后调用notify方法，notify在JVM层面的源码为：

```
void ObjectMonitor::notify(TRAPS) {
    //OWNES就是ObjectMonitor的_owner，指向持有ObjectMonitor对象的线程
  CHECK_OWNER();
   //判断等待池（Wait Set）是是否为空，如果为空，直接返回
  if (_WaitSet == NULL) {
      //触发Empty的Notify
     TEVENT (Empty-Notify) ;
     return ;
  }
    //追踪监视器探针，获取java线程的id以及获取java当前class的名字、字节大小、长度等
  DTRACE_MONITOR_PROBE(notify, this, object(), THREAD);

  int Policy = Knob_MoveNotifyee ;

   ===============================分割线1==============================

   //线程自旋
  Thread::SpinAcquire (&_WaitSetLock, "WaitSet - notify") ;
  //等待池队列，将等待池（wait set）队列的第一个值取出并返回
  ObjectWaiter * iterator = DequeueWaiter() ;
  if (iterator != NULL) {
     TEVENT (Notify1 - Transfer) ;
      //ObjectWaiterd的状态
     guarantee (iterator->TState == ObjectWaiter::TS_WAIT, "invariant") ;
     guarantee (iterator->_notified == 0, "invariant") ;
      //如果Policy!=4 ,状态设置为在获取锁队列的状态
     if (Policy != 4) {
        iterator->TState = ObjectWaiter::TS_ENTER ;
     }
     iterator->_notified = 1 ;
     Thread * Self = THREAD;
      //线程id
     iterator->_notifier_tid = Self->osthread()->thread_id();
    //获取等待锁block状态的线程队列
     ObjectWaiter * List = _EntryList ;
     if (List != NULL) {
         //断言,进行List前指针、状态的断言
        assert (List->_prev == NULL, "invariant") ;
        assert (List->TState == ObjectWaiter::TS_ENTER, "invariant") ;
        assert (List != iterator, "invariant") ;
     }

     if (Policy == 0) {       // prepend to EntryList
         //为空，等待锁block状态的线程队列为空
         if (List == NULL) {
             //如果为空，将队列设置为空
             iterator->_next = iterator->_prev = NULL ;
             _EntryList = iterator ;
         } else {
             //放入_EntryList队列的排头位置
             List->_prev = iterator ;
             iterator->_next = List ;
             iterator->_prev = NULL ;
             _EntryList = iterator ;
        }
     } else if (Policy == 1) {      // append to EntryList
         //Policy == 1: 放入_EntryList队列的末尾位置;
         if (List == NULL) {
             //如果为空，队列设置为空
             iterator->_next = iterator->_prev = NULL ;
             _EntryList = iterator ;
         } else {
            // CONSIDER:  finding the tail currently requires a linear-time walk
of
            // the EntryList.  We can make tail access constant-time by
converting to
            // a CDLL instead of using our current DLL.
             //放入_EntryList队列的末尾位置;
            ObjectWaiter * Tail ;
            for (Tail = List ; Tail->_next != NULL ; Tail = Tail->_next) ;
            assert (Tail != NULL && Tail->_next == NULL, "invariant") ;
            Tail->_next = iterator ;
            iterator->_prev = Tail ;
            iterator->_next = NULL ;
        }
         //Policy == 2 时，将List放在_cxq队列的排头位置
     } else if (Policy == 2) {      // prepend to cxq
         // prepend to cxq
         if (List == NULL) {
              //如果为空，队列设置为空
```

```
              iterator->_next = iterator->_prev = NULL ;
              _EntryList = iterator ;
          } else {
             iterator->TState = ObjectWaiter::TS_CXQ ;
             //放进_cxq队列时，CAS操作，有其他线程竞争
             for (;;) {
                 ObjectWaiter * Front = _cxq ;
                 iterator->_next = Front ;
                 if (Atomic::cmpxchg_ptr (iterator, &_cxq, Front) == Front) {
                     break ;
                 }
             }
          }
          //Policy == 3：放入_cxq队列中，末尾位置；
      } else if (Policy == 3) {      // append to cxq
         iterator->TState = ObjectWaiter::TS_CXQ ;
          //同样CAS操作
         for (;;) {
             ObjectWaiter * Tail ;
             Tail = _cxq ;
             //尾指针为空，设置为空
             if (Tail == NULL) {
                 iterator->_next = NULL ;
                 if (Atomic::cmpxchg_ptr (iterator, &_cxq, NULL) == NULL) {
                    break ;
                 }
             } else {
                     //尾指针不为空，添加在队尾
                 while (Tail->_next != NULL) Tail = Tail->_next ;
                 Tail->_next = iterator ;
                 iterator->_prev = Tail ;
                 iterator->_next = NULL ;
                 break ;
             }
         }
      } else {
          //Policy等于其他值，立即唤醒ObjectWaiter对应的线程；
         ParkEvent * ev = iterator->_event ;
         iterator->TState = ObjectWaiter::TS_RUN ;
         OrderAccess::fence() ;
         ev->unpark() ;
      }

       //Policy<4,等待重试
      if (Policy < 4) {
        iterator->wait_reenter_begin(this);
      }

      // _WaitSetLock protects the wait queue, not the EntryList.  We could
      // move the add-to-EntryList operation, above, outside the critical section
      // protected by _WaitSetLock.  In practice that's not useful.  With the
      // exception of  wait() timeouts and interrupts the monitor owner
      // is the only thread that grabs _WaitSetLock.  There's almost no
contention
      // on _WaitSetLock so it's not profitable to reduce the length of the
      // critical section.
  }
```

```
    //线程自旋释放
   Thread::SpinRelease (&_WaitSetLock) ;

   if (iterator != NULL && ObjectMonitor::_sync_Notifications != NULL) {
      ObjectMonitor::_sync_Notifications->inc() ;
   }
 }
```

ObjectMonitor::notify函数的逻辑主要分为两部分，第一部分做一些检查和准备唤醒操作过程需要的一些信息，第二部所及是根据Policy的大小将需要唤醒的线程放进等待锁block状态的线程队列，即ObjectMonitor的_EntryList和_cxq队列中,这两个队列的线程将等待获取锁。在分割线上部分，CHECK_OWNER()检测是否拥有monitor，只有拥有monitor，才可以唤醒等待的线程，当等待池（wait set）为空，说明没有等待池中没有需要唤醒的线程，直接返回。如果等待池不为空，则准备获取java线程以及获取java当前class（JVM层面读取的是class文件）的名字、字节大小、长度等。分割线下部分第二部分逻辑，当等待池中线程不为空的时候，首先调用 ObjectWaiter * iterator = DequeueWaiter() 从等待池中将第一个等待的线程取出来，DequeueWaiter() 的源代码为：

```
inline ObjectWaiter* ObjectMonitor::DequeueWaiter() {
  // dequeue the very first waiter
  //将wait set赋值给waiter
  ObjectWaiter* waiter = _WaitSet;
  if (waiter) {
    DequeueSpecificWaiter(waiter);
  }
  return waiter;
}
```

ObjectMonitor::DequeueWaiter()中，当_WaitSet不为空时，调用ObjectMonitor::DequeueSpecificWaiter方法返回_WaitSet的第一个元素，ObjectMonitor::DequeueSpecificWaiter方法的源代码为：

```
//将_WaitSet中第一个线程返回回来
inline void ObjectMonitor::DequeueSpecificWaiter(ObjectWaiter* node) {
  assert(node != NULL, "should not dequeue NULL node");
  assert(node->_prev != NULL, "node already removed from list");
  assert(node->_next != NULL, "node already removed from list");
  // when the waiter has woken up because of interrupt,
  // timeout or other spurious wake-up, dequeue the
  // waiter from waiting list
  ObjectWaiter* next = node->_next;
  //
  if (next == node) {
    assert(node->_prev == node, "invariant check");
    _WaitSet = NULL;
  } else {
    ObjectWaiter* prev = node->_prev;
    assert(prev->_next == node, "invariant check");
    assert(next->_prev == node, "invariant check");
    next->_prev = prev;
    prev->_next = next;
    if (_WaitSet == node) {
      _WaitSet = next;
    }
  }
  node->_next = NULL;
```

```
    node->_prev = NULL;
  }
```

ObjectMonitor::DequeueSpecificWaiter方法中，首先判断ObjectWaiter的_next是否等于_WaitSet，如果是否，则说明_WaitSet为空，将_WaitSet设置为NULL,如果不是，则将第一元素返回。

ObjectWaiter是JVM层面的C++类，ObjectWaiter类为：

```
// ObjectWaiter serves as a "proxy" or surrogate thread.
// TODO-FIXME: Eliminate ObjectWaiter and use the thread-specific
// ParkEvent instead.  Beware, however, that the JVMTI code
// knows about ObjectWaiters, so we'll have to reconcile that code.
// See next_waiter(), first_waiter(), etc.
class ObjectWaiter : public StackObj {
 public:
    //状态
  enum TStates { TS_UNDEF, TS_READY, TS_RUN, TS_WAIT, TS_ENTER, TS_CXQ } ;
  enum Sorted  { PREPEND, APPEND, SORTED } ;
   //下一个ObjectWaiter指针
  ObjectWaiter * volatile _next;
    //前一个ObjectWaiter指针
  ObjectWaiter * volatile _prev;
    //线程
  Thread*       _thread;
    //被唤醒的线程id
  jlong         _notifier_tid;
  ParkEvent *   _event;
  volatile int  _notified ;
  volatile TStates TState ;
  Sorted        _Sorted ;           // List placement disposition
  bool          _active ;           // Contention monitoring is enabled
 public:
  ObjectWaiter(Thread* thread);

  void wait_reenter_begin(ObjectMonitor *mon);
  void wait_reenter_end(ObjectMonitor *mon);
};
```

ObjectWaiter类充当“代理”或代理线程，也就是ObjectWaiter充当_thread的代理角色，负责与其他对外的工作对接。_next指向下一个ObjectWaiter指针，_prev指向前一个ObjectWaiter指针。回到ObjectMonitor::notify函数的第二部分逻辑，当从线程池中取出第一个ObjectWaiter（线程代理），根据Policy的值不同，将取出的线程放入等待锁的_EnterList或者_cxq队列中的起始或末尾位置。当Policy == 0时，将等待池取出的iterator（线程或者线程代理）放进 _EntryList中的排头位置；当Policy == 1时，将等待池取出的iterator放进_EntryList中的末尾位置；当Policy == 2时，将等待池中取出的iterator放进放在_cxq队列的排头位置。因为有其他线程的竞争，当放入_cxq队列时，进行CAS操作保证线程的安全；在讲解ObjectMonitor结构出现过，_cxq是阻塞在_EnterList最近可达的线程列表，该列表其实是waitNode所构成的线程代理；当Policy == 3时，将等待池中取出的iterator放入_cxq队列中的末尾位置；当Policy等于其他值，立即唤醒ObjectWaiter对应的线程，唤醒线程以后并没有释放锁。经过上面的分析，我们知道java中调用notify方法时，不一定是立即唤醒线程，可能先将等待池中取出的线程放在获取锁阻塞池中（_EntryList或_cxq）。

java的notifyAll方法在JVM中的实现跟java的notify方法基本一样，这里就不贴源码了，主要区别是遍历ObjectWaiter * iterator = DequeueWaiter() ，重复java的notify方法的JVM实现过程，把所有的_WaitSet中的ObjectWaiter对象放入到_EntryList中。

JVM中wait、notify、notifyAll 方法中都没有释放锁，锁的释放是在Synchronizer同步块结束的时候释放的。

释放锁调用ObjectMonitor::exit方法，主要将ObjectWaiter从_cxq或者_EntryList中取出后唤醒，唤醒的线程会继续执行挂起前的代码，通过CAS去竞争锁，exit方式释放锁后，被唤醒的线程占用了该锁。

protected void finalize()

Object类中最后一个方法是finalize()，由protected 修饰，由子类进行重写。当对象的引用不再被使用时，垃圾回收器进行调用finalize。这个方法是由垃圾回收器进行调用的，所以该方法可能不会被触发，finalize方法不会被任何对象调用多次。当子类重写了finalize方法，并且这个方法体不为空时，JVM层面则会调用register_finalizer函数进行注册这个方法，finalize方法是在Java对象初始化过程中注册的，当进行垃圾回收时，对象被回收并且在finalize中引用自身时，会逃过一次回收，这样对象不一定会被回，finalize方法也不可能调用第二次。