发件人：**方堃** fangkun119@icloud.com
主题：CH04 Guarded Suspension
日期：2017年5月2日 上午11:59
收件人：

CH04 Guarded Suspension

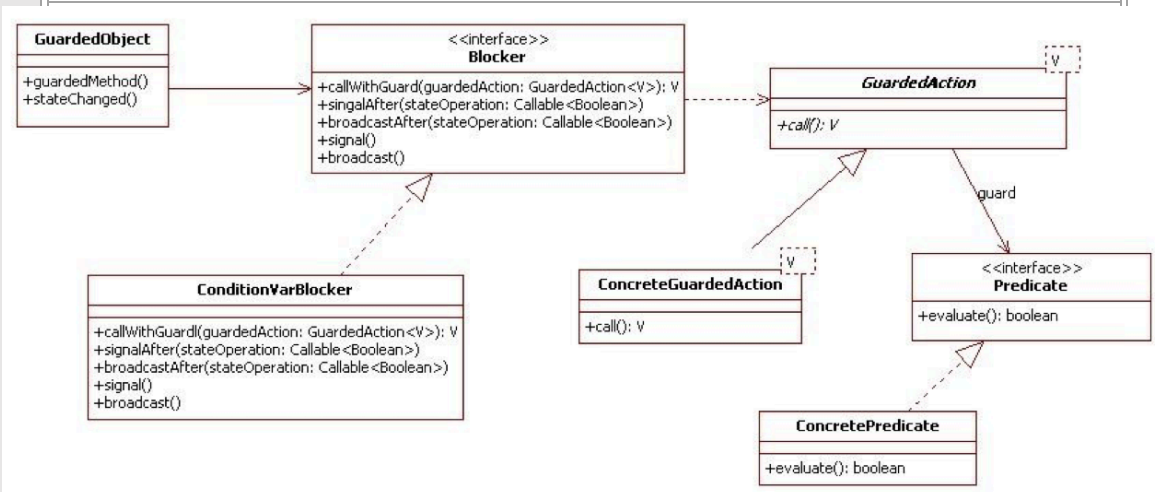
2017年4月21日 星期五
上午9:42

用途

Guarded Suspension模式的核心是一个受保护方法（Guarded Method）。该方法执行其所要真正执行的操作时需要满足特定的条件（Predicate，以下称之为保护条件）。当该条件不满足时，执行受保护方法的线程会被挂起进入等待（WAITING）状态，直到该条件满足时该线程才会继续运行。此时，受保护方法才会真正执行其所要执行的操作。为方便起见，以下称受保护方法所要真正执行的操作为目标动作。

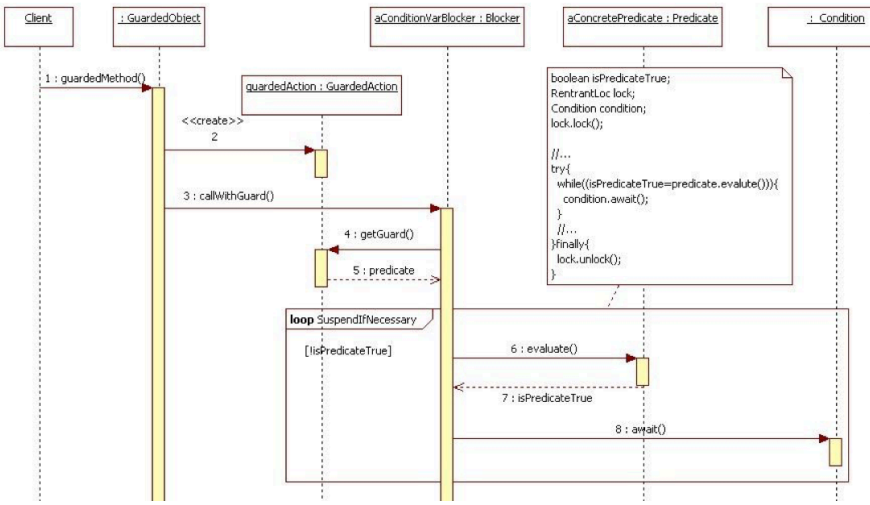
UML

GuardedObject	作用	提供给client代码的入口	
	方法	guardedMethod()是受保护的方法，stateChanged()用来触发事件、告知受保护方法条件已满足、可解除阻塞	
	交互	与GuardedAction交互	(1)创建GuardedAction (2)将GuardedAction交给Blocker (3)Blocker调用完成后检查Action结果
		与Blocker交互	执行callWithGuard(guardedAction)
Blocker	callWithGauard	作用	ClientThread A们调用callWithGuarded()，阻塞直到条件具备，执行GuardedAction，返回结果
		交互	与Predict：解除阻塞的条件是否已具备（放在cond-wait-loop的while条件检查中）
			与Condition：需要时阻塞在condition-wait上（放在cond-wait-loop的while循环内部）
			与GuardedAction：1.取出GuardedAction的Predict变量 2.条件满足时执行GuardedAction的call() 3.返回GuardedAction的执行结果
	signalAfter(..) broadcastAfter(..) signal() broadcast()	作用	ClientThread B们调用这些函数，解除ClientThread A中某些线程的阻塞，执行完GuardedAction
		交互	与Callable对象：从signalAfter/broadcastAfter传入的对象，执行前置任务
			与Condition：前置任务执行完之后，向Condition发送Signal
Condition	封装条件变量，提供await()、signal()函数，为两方线程（执行受保护方法，通知方法执行条件具备）处理条件变量等待和通知		



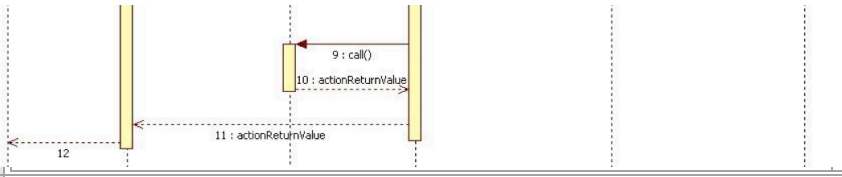
方法调用

期间会发生阻塞，等待触发条件被满足



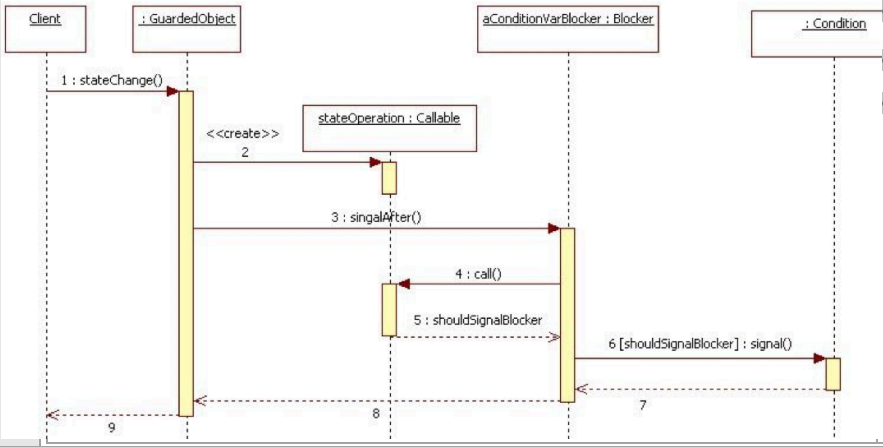
调用受保护方法

GuardedObject
guardedMehtod()是入口
所属的Predict标记前置条件
GuardedAction
封装保护操作内容
Blocker及子类
封装cond-wait-signal操作
在Condition与Predict帮助下
条件具备时执行GuardedAction



事件触发

执行前置任务，cond-signal，让受保护方法的阻塞被解除



执行

GuardedObject
statChange()是入口
Callable
是前置任务
Condition
signal()来解除条件变量阻塞

例子

见电子书

优点

优点	易错细节封装	容易出错的细节封装在框架中，避免用户出错，同时这部分
	高内聚	框架提供

<<interface>>Blocker	ConditionVarBlocker	GuardedAction	<<interface>>Predict
为客户代码定义阻塞/通知接口	阻塞/通知操作的具体实现 (容易出错的细节封装在这里)	前置操作基类	前置操作检查接口

用户实现

GuardedObject	ConcreteGuardedAction	ConcretePredict
整合业务场景	前置操作具体实现	前置操作检查的具体实现

缺点	新生代垃圾回收	GuardedAction闭包来封装受保护操作，导致原本的函数变成了对象。如果GuardedAction生成非常频繁、同时新生代内存少，容易增加JVM负担
注意	线程上下文切换	频繁出现保护方法被调用，而保护条件不成立时，容易引发过多的上下文切换

实现要点

防止锁泄漏(没有释放)	//获得锁
	lock.lockInterruptibly();
	try {
	//临界区代码
	} finally {
	//在finally块中释放锁，保证锁总是会被释放的
	lock.unlock();
	}
		1.用ReentrantLock并借助finally确保锁一定会被释放

防止线程被过早唤醒	lock.lockInterruptibly();
	try {
	while (保护条件不成立) {
	condition.await();
	}
	执行目标动作
	} finally {
	lock.unlock();
	}
		1.condition.await()放在while循环中
		这样condition.await()生效之前发生的signal()可以被while捕捉到
		2.lock加锁，condition.await()应当能够将lock释放
		因为发signal也需要占用lock
		所以lock到condition.await()这段时间是被锁保护的

防止监视器锁死	当guardedMethod()和stateChanged()被声明为synchronized时会死锁。	callWithGuard与signalAfter/signal/broadcast/...共用同一把显式锁，来进行互斥。Condition#await(), Condition#signal()可以在调用时把锁释放 当guardedMethod()和stateChanged()被声明为synchronized时会死锁。因为guardedMethod()阻塞在Codition#await()时，需要stateChanged()来调用Condition#signal()来解除阻塞。而synchronized使得stateChanged()无法被调用

附: Eclipse查看锁阻塞的状况

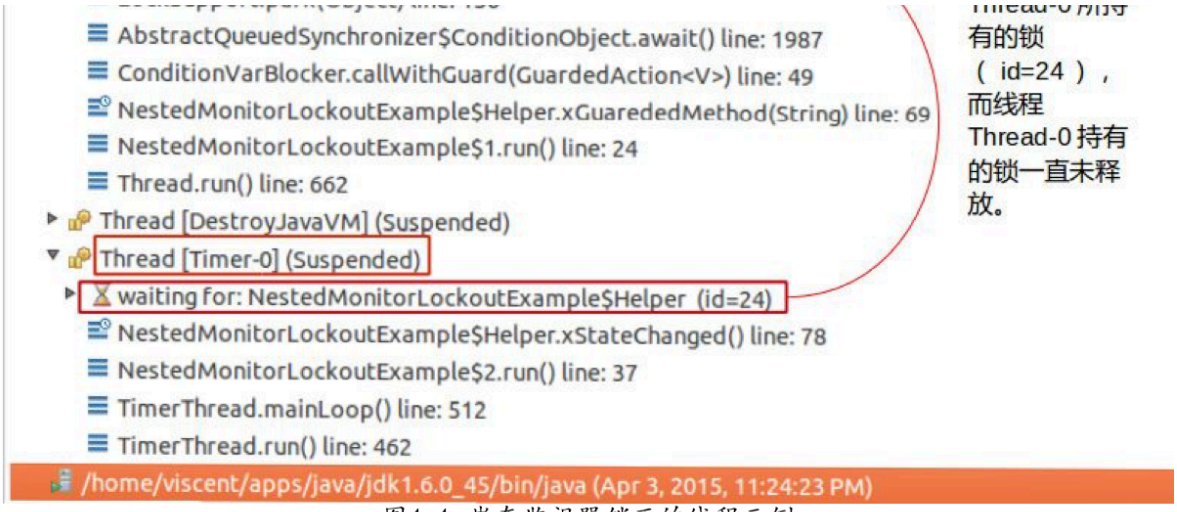


图4-4. 嵌套监视器锁死的线程示例

可复用代码

目录

框架提供（可复用）	<<interface>>Blocker	ConditionVarBlocker	GuardedAction	<<interface>>Predict
	为客户代码定义阻塞/通知接口	阻塞/通知操作的具体实现（容易出错的细节封装在这里）	前置操作基类	前置操作检查接口

用户实现	GuardedObject	ConcreteGuardedAction	ConcretePredict
	整合业务场景	前置操作具体实现	前置操作检查的具体实现

Predict

```
public interface Predicate {
 boolean evaluate();
}
```

GuardedAction

```
public abstract class GuardedAction<V> implements Callable<V> {
 protected final Predicate guard;

 public GuardedAction(Predicate guard) {
  this.guard = guard;
 }
}
```

Blocker

```
public interface Blocker {

 /**
  * 在保护条件成立时执行目标动作；否则阻塞当前线程，直到保护条件成立。
  * @param guardedAction 带保护条件的目标动作
  * @return
  * @throws Exception
  */
 <V> V callWithGuard(GuardedAction<V> guardedAction) throws Exception;

 /**
  * 执行stateOperation所指定的操作后，决定是否唤醒本Blocker
  * 所暂挂的所有线程中的一个线程。
  *
  * @param stateOperation
  *            更改状态的操作，其call方法的返回值为true时，该方法才会唤醒被暂挂的线程
  */
 void signalAfter(Callable<Boolean> stateOperation) throws Exception;

 void signal()  throws InterruptedException;

 /**
  * 执行stateOperation所指定的操作后，决定是否唤醒本Blocker
  * 所暂挂的所有线程。
  *
  * @param stateOperation
  *            更改状态的操作，其call方法的返回值为true时，该方法才会唤醒被暂挂的线程
  */
 void broadcastAfter(Callable<Boolean> stateOperation) throws Exception;
}
```

ConditionVarBroker

```
public class ConditionVarBlocker implements Blocker {
 private final Lock lock;
```

```
 private final Lock lock;
private final Condition condition;
public ConditionVarBlocker(Lock lock) {
 this.lock = lock;
 this.condition = lock.newCondition();
}
public ConditionVarBlocker() {
 this.lock = new ReentrantLock();
 this.condition = lock.newCondition();
}
public <V> V callWithGuard(GuardedAction<V> guardedAction) throws Exceptio
n {
 lock.lockInterruptibly();
 V result;
 try {
  final Predicate guard = guardedAction.guard;
  while (! guard.evaluate()) {
   condition.await();
  }
  result = guardedAction.call();
  return result;
 } finally {
  lock.unlock();
 }
}
public void signalAfter(Callable<Boolean> stateOperation) throws Exception
 {
 lock.lockInterruptibly();
 try {
  if (stateOperation.call()) {
   condition.signal();
  }
 } finally {
  lock.unlock();
 }
}
public void broadcastAfter(Callable<Boolean> stateOperation) throws Except
ion {
 lock.lockInterruptibly();
 try {
  if (stateOperation.call()) {
   condition.signalAll();
  }
 } finally {
  lock.unlock();
 }
}
public void signal()  throws InterruptedException {
 lock.lockInterruptibly();
 try {
  condition.signal();
 } finally {
   lock.unlock();
  }

 }
}
```

Java标准例	JDK 1.5开始提供的阻塞队列类java.util.concurrent.LinkedBlockingQueue就使用了Guarded Suspension模式。该类的take方法用于从队列中取出一个元素。如果take方法被调用时，队列是空的，则当前线程会被阻塞；直到队列不为空时，该方法才返回一个出队列的元素。只不过LinkedBlockingQueue在实现Guarded Suspension模式时，直接使用了java.concurrent.locks.Condition。