java并发编程笔记--Guarded Suspension模式

|  |  |
| --- | --- |
|  | 🢂 内容概览 |
|  | Why：此文档用来做什么？它存在的意义是什么？为解决什么问题？   |  | | --- | |  |   What：当前包含了那些内容？   |  | | --- | |  |   How：此文档应如何参考？   |  | | --- | |  |   Who：此文档适用于那些人员阅读参考？   |  | | --- | |  |   Summary：摘要   |  | | --- | |  |   Reference：参考文献   |  | | --- | | [死锁的三种形式：一般死锁，嵌套管程锁死，重入锁死](http://blog.csdn.net/dafeng_blog/article/details/38236305) | |

目录

[1 模式描述 3](#_Toc462681257)

[1.1 Guarded Suspension模式 3](#_Toc462681258)

[2 相关原则 3](#_Toc462681259)

[2.1 遵循原则 3](#_Toc462681260)

[2.2 违反原则 3](#_Toc462681261)

[3 模式类图 3](#_Toc462681262)

[4 模式实现 4](#_Toc462681263)

[4.1 定义守护条件：Predicate & ConcretePredicate 4](#_Toc462681264)

[4.2 定义目标操作：GuardedAction & ConcreteGuardedAction 5](#_Toc462681265)

[4.3 定义控制器：Blocker & ConcreteBlocker 6](#_Toc462681266)

[4.4 模拟客户端：Client 9](#_Toc462681267)

[5 适用场景 10](#_Toc462681268)

[5.1 业务解耦 10](#_Toc462681269)

[6 缺点&权衡点 10](#_Toc462681270)

[6.1 增加上下文切换 10](#_Toc462681271)

[6.2 守护条件内存可见性 10](#_Toc462681272)

[6.3 防止虚假唤醒 10](#_Toc462681273)

[6.4 防止嵌套加锁导致死锁发生 10](#_Toc462681274)

[7 应用案例 11](#_Toc462681275)

[7.1 BlockingQueue 11](#_Toc462681276)

[8 相关模式 11](#_Toc462681277)

[8.1 Promise模式 11](#_Toc462681278)

[8.2 Producer-Consumer模式 11](#_Toc462681279)

[9 问题思考 11](#_Toc462681280)

# 模式描述

## Guarded Suspension模式

1. Guarded Suspension（守护挂起），核心是一个要执行的目标方法。只有当特定的条件满足时，才执行方法；否则，就挂起执行的线程，直到条件满足时，才唤醒线程执行目标方法。
2. Guarded Suspension是一种基础模式，用于实现其它更为复杂的模式，比如生产者-消费者模式。
3. 在Java中，线程的挂起/唤醒通常使用wait/notify/notifyAll或者Condition的await/singal/singalAll。
4. 当守护条件中包含状态属性时，需要注意属性在多个线程的可见性，一般需要将属性声明为volitale的。

# 相关原则

## 遵循原则

单一职责原则：Guarded Suspension模式分离了数据的写操作和读操作，从而是的数据的生产者和消费者的业务分离，各自关注自己的业务实现，符合单一职责原则；

## 违反原则

# 模式类图

最简单的Guarded Suspension模式可以直接通过while循环不断对守护条件进行轮询进行实现，可以放在一个类中。下图是出于对代码可扩展性考量对Guarded Suspension的实现，其中用到了策略模式：

|  |
| --- |
|  |

**模式角色如下：**

1. **Predicate & ConcretePredicate：**定义守护条件， 其中的evaluate()方法用于判断是否满足守护条件；当满足守护条件时，返回true；反之，返回false；此处使用了策略模式，ConcretePredicate代表了根据不同的业务场景实现不同的守护条件。
2. **GuardedAction & ConcreteGuardedAction：**定义要守护的目标操作，call()方法代表要被执行的目标操作，只有当满足守护条件时，才会调用call()方法；同样使用了策略模式，ConcreteGuardedAction代表了不同业务场景下需要进行的目标操作；
3. **Blocker & ConcreteBlocker：**作为整个模式的控制器，控制守护的流程；使用了策略模式，ConcreteBlocker代表了不同实现方式的控制器；callWithGuard()方法代表了执行目标操作的方法，singal()/broadcast()代表了唤醒挂起线程的方法，区别是可以唤醒一个线程或者全部唤醒；singalOnState()/broadcastOnState()代表了在唤醒线程之前需要执行额外的操作，参数Callable<Boolean>代表了要执行的额外操作，通常是用来改变守护条件的操作；
4. **Client（客户端）：**表示执行目标操作的客户端对象，包含了执行目标操作的方法execute()以及改变守护条件的方法changeState()，两个方法在不同的线程中执行，以保证被守护条件挂起的线程能够因为守护条件的改变而被唤醒；

# 模式实现

## 定义守护条件：Predicate & ConcretePredicate

|  |
| --- |
| **package** com.fsindustry.cime.dpattern.guardedSuspension.sample;  */\*\*  \* <h1>定义守护条件,在执行目标操作之前进行检查</h1>  \*/* **public interface** Predicate {   */\*\*  \* <h2>守护条件检查方法</h2>  \*  \** ***@return*** *true, 满足守护条件;false,不满足守护条件;  \*/* **boolean** evaluate(); } |

守护条件实现类，依据具体的业务场景实现，不同的业务场景守护条件不同，如下代码仅是模拟：

|  |
| --- |
| **package** com.fsindustry.cime.dpattern.guardedSuspension.sample;  */\*\*  \* <h1>守护条件实现类</h1>  \*/* **public class** ConcretePredicate **implements** Predicate {   */\*\*  \* 守护条件状态,注意:用volatile保证属性的可见性  \*/* **private volatile** Boolean **isOk**;   **public** ConcretePredicate(Boolean isOk) {  **this**.**isOk** = isOk;  }   @Override  **public boolean** evaluate() {  **return isOk**;  }   */\*\*  \* 用于模拟对守护条件的改变  \*/* **public void** setOk(Boolean ok) {  **isOk** = ok;  } } |

**注意：**上面代码中，用于标识守护状态的属性一定要确保对线程的可见性，因而声明为volatile的；

## 定义目标操作：GuardedAction & ConcreteGuardedAction

GuardedAction定义，可以是接口也可以是抽象类，如下以抽象类方式定义：

|  |
| --- |
| **package** com.fsindustry.cime.dpattern.guardedSuspension.sample;  **import** java.util.concurrent.Callable;  */\*\*  \* <h1>抽象目标动作,关联目标动作所需的守护条件</h1>  \*/* **public abstract class** GuardedAction<V> **implements** Callable<V> {   */\*\*  \* 存放当前守护条件检测对象  \*/* **protected final** Predicate **guard**;   **public** GuardedAction(Predicate guard) {  **this**.**guard** = guard;  }   */\*\*  \* <h2>目标动作的方法</h2>  \*  \** ***@return*** *\*/* **public abstract** V call(); } |

ConcreteGuardedAction实现如下：

|  |
| --- |
| **package** com.fsindustry.cime.dpattern.guardedSuspension.sample;  */\*\*  \* <h1>具体的目标操作对象,根据不同的业务实现</h1>  \* 当前实现累目标操作返回值为String  \*/* **public class** ConcreteGuardedAction **extends** GuardedAction<String> {   **public** ConcreteGuardedAction(Predicate guard) {  **super**(guard);  }   @Override  **public** String call() {  *//模拟目标方法返回值* **return "ConcreteGuardedAction目标方法已执行!"**;  } } |

## 定义控制器：Blocker & ConcreteBlocker

|  |
| --- |
| **package** com.fsindustry.cime.dpattern.guardedSuspension.sample;  **import** java.util.concurrent.Callable;  */\*\*  \* <h1>作为整个模式的控制器，控制守护的流程；</h1>  \*/* **public interface** Blocker {   */\*\*  \* <h2>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;</h2>  \*  \** ***@param guardedAction*** *带保护条件的目标动作;  \*  \** ***@return*** *目标动作执行完成后的返回值;  \*/* <V> V callWithGuard(GuardedAction<V> guardedAction) **throws** Exception;   */\*\*  \* <h2>执行传入的操作,根据动作返回的结果决定是否执行唤醒操作;</h2>  \* 唤醒操作仅唤醒一个线程;  \*  \** ***@param stateOperation*** *用于执行判断条件的操作  \*/* **void** singalOnState(Callable<Boolean> stateOperation) **throws** Exception;   */\*\*  \* <h2>执行传入的操作,根据动作返回的结果决定是否执行唤醒操作;</h2>  \* 唤醒操作唤醒所有线程;  \*  \** ***@param stateOperation*** *用于执行判断条件的操作  \*/* **void** broadcastOnState(Callable<Boolean> stateOperation) **throws** Exception;   */\*\*  \* <h2>负责唤醒Blocker实例锁暂挂的一个线程;</h2>  \*/* **void** singal() **throws** Exception;   */\*\*  \* <h2>负责唤醒Blocker实例锁暂挂的所有线程;</h2>  \*/* **void** broadcast() **throws** Exception; } |

Blocker的实现类：

|  |
| --- |
| **package** com.fsindustry.cime.dpattern.guardedSuspension.sample;  **import** java.util.concurrent.Callable; **import** java.util.concurrent.locks.Condition; **import** java.util.concurrent.locks.Lock; **import** java.util.concurrent.locks.ReentrantLock;  */\*\*  \* <h1>基于Lock和Condition条件变量实现的控制器</h1>  \*/* **public class** ConditionVarBlocker **implements** Blocker {   */\*\*  \* 用于在阻塞/唤醒时加锁  \*/* **private final** Lock **lock**;   */\*\*  \* 作为阻塞/唤醒的条件变量  \*/* **private final** Condition **condition**;   **public** ConditionVarBlocker() {  **this**.**lock** = **new** ReentrantLock();  **this**.**condition** = **this**.**lock**.newCondition();  }   **public** ConditionVarBlocker(Lock lock) {  **this**.**lock** = lock;  **this**.**condition** = lock.newCondition();  }   @Override  **public** <V> V callWithGuard(GuardedAction<V> guardedAction) **throws** Exception {   **lock**.lockInterruptibly();  V result;  **try** {  *//获取守护条件* **final** Predicate guard = guardedAction.**guard**;   *//如果守护条件不成立,则一致阻塞等待,使用while循环防止虚假唤醒;* **while** (!guard.evaluate()) {  **condition**.await();  }   *//当条件成立时,执行要执行的方法,返回结果* result = guardedAction.call();  **return** result;  } **finally** {  **lock**.unlock();  }  }   @Override  **public void** singalOnState(Callable<Boolean> stateOperation) **throws** Exception {  **lock**.lockInterruptibly();  **try** {  *//如果传入操作返回true,则唤醒一个线程* **if** (stateOperation.call()) {  **condition**.signal();  }  } **finally** {  **lock**.unlock();  }  }   @Override  **public void** broadcastOnState(Callable<Boolean> stateOperation) **throws** Exception {  **lock**.lockInterruptibly();  **try** {  *//如果传入操作返回true,则唤醒所有线程* **if** (stateOperation.call()) {  **condition**.signalAll();  }  } **finally** {  **lock**.unlock();  }  }   @Override  **public void** singal() **throws** Exception {  **lock**.lockInterruptibly();  **try** {  **condition**.signal();  } **finally** {  **lock**.unlock();  }  }   @Override  **public void** broadcast() **throws** Exception {  **lock**.lockInterruptibly();  **try** {  **condition**.signalAll();  } **finally** {  **lock**.unlock();  }  } } |

**注意：**对于守护条件的判断一定放在while循环而非if语句中，防止虚假唤醒。

## 模拟客户端：Client

客户端模拟了执行带守护条件的目标操作以及对守护条件改变的操作，这两个操作一定是在不同的线程当中，可以在同一个类中也可以在不同的类中，依据实际设计确定。

|  |
| --- |
| **package** com.fsindustry.cime.dpattern.guardedSuspension.sample;  **import** java.util.concurrent.Callable;  */\*\*  \* <h1>模拟客户端对象,用来触发守护等待操作和唤醒操作</h1>  \*/* **public class** Client {   **public static void** main(String[] args) {   *//创建目标操作对象* ConcretePredicate predicate = **new** ConcretePredicate(**false**);  GuardedAction<String> action = **new** ConcreteGuardedAction(predicate);   *//创建控制器对象* Blocker blocker = **new** ConditionVarBlocker();  *guardedMethod*(blocker, action);   *//触发唤醒操作  stateChange*(blocker, predicate);   }   */\*\*  \* 执行守护等待操作  \*/* **public static void** guardedMethod(**final** Blocker blocker, GuardedAction<String> action) {   *//在子线程中调用带守护条件的目标操作* **new** Thread(**new** Runnable() {  @Override  **public void** run() {   *//新建线程,调用目标操作;* **try** {  String result = blocker.callWithGuard(action);  System.***out***.println(**"result:"** + result);  } **catch** (Exception e) {  e.printStackTrace();  }  }  }).start();  }   */\*\*  \* 更改守护条件状态,唤醒操作  \*/* **public static void** stateChange(**final** Blocker blocker, ConcretePredicate predicate) {   *//触发唤醒操作* **try** {  *//模拟操作耗时* System.***out***.println(**"5秒后改变触发条件。。。"**);  Thread.*sleep*(5 \* 1000);  blocker.singalOnState(**new** Callable<Boolean>() {  @Override  **public** Boolean call() **throws** Exception {  *//模拟触发条件的改变* predicate.setOk(**true**);  System.***out***.println(**"触发条件改变。。。"**);  **return true**;  }  });  } **catch** (Exception e) {  e.printStackTrace();  }  } } |

# 适用场景

## 业务解耦

能够将业务参与者之间的关系解耦，各自关注各自的业务，做到低耦合高内聚。

# 缺点&权衡点

## 增加上下文切换

根据守护条件实现方式的不同（比如：使用cas+自旋的方式），可能会造成对执行目标操作线程的频繁阻塞和唤醒，增加上下文切换，消耗CPU资源。

## 守护条件内存可见性

用于检查守护条件的变量，同时受到状态检测线程和状态修改线程的访问，故需要保证内存可见性，通常对于单个变量控制的守护条件而言，可以通过volatile关键字声明来确保可见性；对于一系列变量控制的守护条件而言，可以直接通过lock保证可见性。

## 防止虚假唤醒

因为涉及到对线程的阻塞/唤醒操作，故需要注意”虚假唤醒”的情况，使用while语句而非if语句进行条件检查。

## 防止嵌套加锁导致死锁发生

执行守护操作和状态修改操作，有时可能被放在同一个类中。如果在客户端使用时，再一次加锁实现对两个操作进行互斥，可能导致死锁（嵌套监视器锁死）的发生，比如：当守护操作获取锁，并执行守护条件的判断时，一直持有锁。而执行状态操作的线程无法获取到锁，也就无法更改状态，导致死锁。

# 应用案例

## BlockingQueue

java.util.concurrent.BlockingQueue阻塞队列，当队列为空时，提供可阻塞的出队操作；当队列满时，提供可阻塞的入队操作；这两个操作的实现，实际上就是Guarded Suspension模式的实现。

# 相关模式

## Promise模式

Promise模式，使用Guarded Suspension模式作为实现基础，结果访问操作和结果设置操作放在不同线程执行，当结果访问时，如果结果设置操作还未完成时，线程进入等待状态，直到结果设置线程唤醒为止。Java的Futrue变量即使用该模式实现。

## Producer-Consumer模式

生产者-消费者模式同样使用Guarded Suspension模式作为实现基础，分别对消息的生产和消费进行控制。在没有消息时，阻塞消费者，直到有消息来了才唤醒；在消息数量达到上限时，阻塞生产者，直到消息数量低于上限才唤醒。

# 问题思考

1. **如何保证守护状态的可见性？**
2. **如下这段代码，如何理解？因为执行await操作时，线程释放掉了已经持有的lock，此时当while条件不满足时，需要再次执行await，则执行之前需要再次获取锁吗？**

|  |
| --- |
| *//如果守护条件不成立,则一致阻塞等待,使用while循环防止虚假唤醒;* **while** (!guard.evaluate()) {  **condition**.await(); } |