# 多线程

## 阻塞队列

### BlockingQueue

根据jdk1.7，查看源码注释得到下面的记录。

提供的方法如下：

#### boolean add(E e)

在队列中插入一个元素。如果队列**没有容量限制**或者**有可用的空间**，插入成功，并且返回true，否则，抛出异常IllegalStateException。

这个方法是返回一个boolean类型的返回值，但是注释里只说失败了会抛出异常，没说失败了会返回false，应该就是不返回false。

假如队列存在容量限制，我们推荐优先使用offer方法。

#### boolean offer(E e)

在队列中插入一个元素。如果队列**有可用的空间**，插入成功，并且返回true；如果没有可用的空间，返回false。

当队列存在容量限制的时候，我们推荐优先使用该方法。

上面两个插入方法，注释里并没有提及没有可用空间会阻塞线程的问题，所以认为是如果没有可用空间，立即返回了。

#### void put(E e)

在队列中插入一个元素，当没有可用的空间的时候，会阻塞插入线程，直到有可用空间成功插入元素为止。

#### boolean offer(E e, long timeout, TimeUnit unit)

这个和前面的offer一样，只是加了一个等待时间，当没有可用空间的时候，会等待一个时间。成功插入，返回true；如果等待时间结束，队列仍然没有可用空间，返回false。

#### E take()

返回并且一处队列的头元素，如果队列当前为空，会阻塞取元素的线程，直到队列有可用的元素为止。

#### E poll(long timeout, TimeUnit unit)

返回并且一处队列的头元素，如果队列当前为空，会阻塞取元素的线程tiemout的时间。如果成功取到，返回对应的元素；否则，返回null。

#### ArrayBlockingQueue

该类实现了接口BlockingQueue。

类注释翻译整理如下：

使用一个有界的数组缓存数据，是一个先进先出(FIFO)的队列。其它的说明，无非就是插入和取出元素时阻塞之类的。

重点说一下关于等待策略：

当有很多线程来取元素的时候，如果使用公平策略，能严格执行先进先出策略，但是这样会降低系统的吞吐量。如果使用不公平策略，则不能完全保证先进先出策略，比如消费者线程，当有可用的元素的时候，只能保证这些线程一起去争夺这个可用元素，但是不能保证先进先出。

对于添加元素和删除元素都使用**同一个锁**。

## 可重入锁

字面意思就是当一个线程获取到了某个锁以后，该线程可以多次获得该锁。

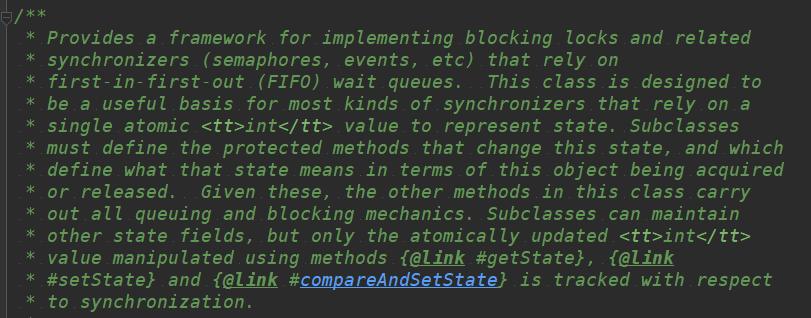
可重入锁的意义就在于可以防止死锁。

### AbstractQueuedSynchronizer

说明：只是把源码里的注释翻译了一下，并且简单写一下在使用的时候，需要重写的方法，以及方法的作用。并会提供几个Demo。

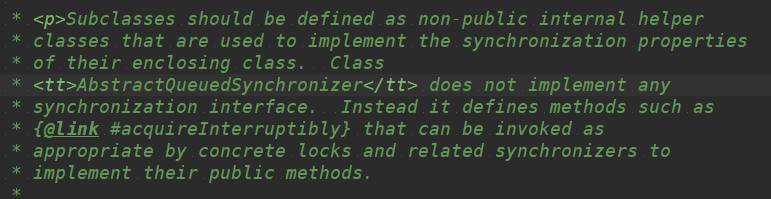
首先，说明一下这个类的作用。

1. 通过使用一个先进先出(FIFO)的队列为我们实现**阻塞锁**和**同步器**提供了一个框架。该类可以作为实际中大部分同步器的实现基础，它所依赖的是一个代表状态的int类型的值（下面用state代替）。子类必须去实现该类中的protected方法，这些方法是用来改变state的值，state的值代表了锁的释放和占有。该类里面其它的方法实现了队列以及阻塞机制。获取state的值只能使用getState，修改state的值只能使用setState或者compareAndSetState。

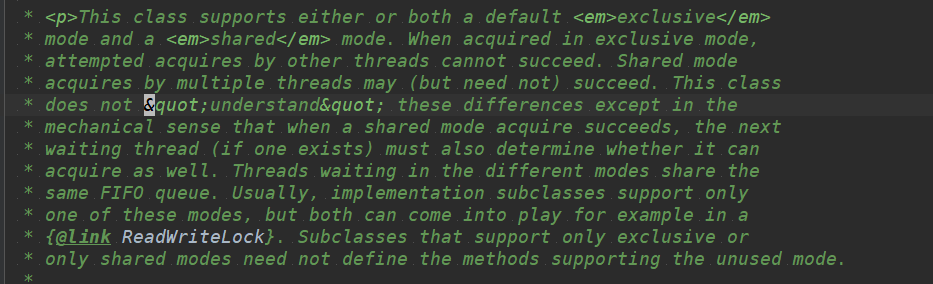


1. 继承该类的子类被推荐为声明为一个**非静态的内部类(non-public internal helper classes)**。

~~该类的内里机制并不知道，按照官方推荐实现就好，另外这一部分缺失了一部分翻译。~~



1. 该框架类支持**独占锁**和**共享锁**。独占模式下，锁被一个线程获取到后，其它线程只能等待；共享模式下，如果锁被一个线程获取到了，其它线程再次尝试获取锁，也需要判断，不是直接就获取成功了。通常，在实际应用中，我们只需要实现一个模式就好了，要么使用独占锁，要么使用共享锁。但是也有同时实现的，比如ReadWriteLock，写是独占锁，读是共享锁。



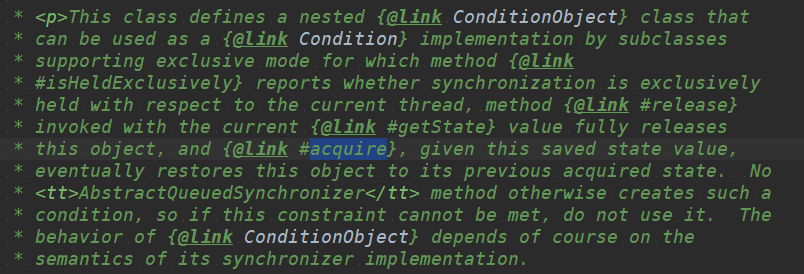
1. 该框架类内部还实现了一个ConditionObject(实现接口Condition)。

ConditionObject里的几个简单的方法：

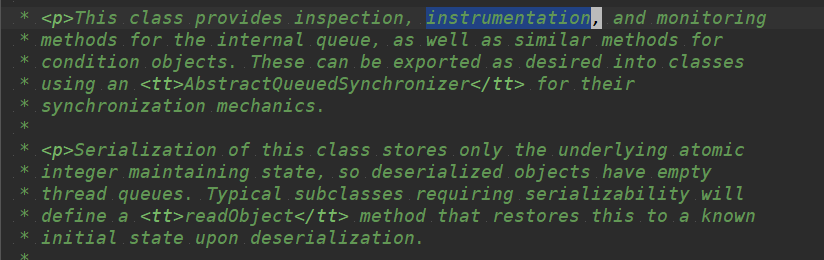
isHeldExclusively：当前线程是独占模式还是共享模式

release：释放锁

acquire：获取锁



1. 剩下的两段无所谓了。



接下来的就是官方提供的用法了。

需要重写的方法：

**特别提示**：获取和修改状态的方法。上面已经说过了。

<ul>

<li> {@link #tryAcquire}

<li> {@link #tryRelease}

<li> {@link #tryAcquireShared}

<li> {@link #tryReleaseShared}

<li> {@link #isHeldExclusively}

</ul>

好了，接下来就是该框架类里其它辅助的内部类的解释了。

#### Node

这是等待队列的Node类。

等待队列是一个锁的队列。**锁是自旋锁(其实就是个死循环，释放锁的当前线程通知一下下一个线程。还给自己起了一个这么高大上的名字，这个方法都经常用好吧)。**

这个类比较简单，简单介绍一下里面的几个成员变量和成员方法。

成员变量：

所有的成员变量都是volatile修饰的。

waitStatus：有以下几个状态值。

SIGNAL: 当前线程的**继承者**处于阻塞(通过park方法)状态。当当前线程释放锁

或者被中断以后，需要释放继承者(unpark)。

CANCELLED: 当前线程由于超时或者中断被取消了。处于该状态的线程不会被

阻塞（鬼知道阻塞到底是几个意思，可能就是被取消的永远不会获

取到锁了？）。

CONDITION: This node is currently on a condition queue.

It will not be used as a sync queue node

until transferred, at which time the status

will be set to 0. (Use of this value here has

nothing to do with the other uses of the

field, but simplifies mechanics.)

PROPAGATE: A releaseShared should be propagated to other

nodes. This is set (for head node only) in

doReleaseShared to ensure propagation

continues, even if other operations have

since intervened.

0: None of the above

prev：

next：

thread：

nextWaiter：

#### Sync

# Java基础

## Java内存管理以及GC原理

### 内存管理

Java虚拟机在运行时，会把内存分为若干个不同数据区域，这几个区域有各自的用途，以及创建销毁的时间，有的区域随着虚拟机的启动销毁而启动销毁，有的随着线程的启动销毁而启动销毁。

主要有5个区域：方法区、虚拟机栈、本地方法栈、程序计数器、堆。（运行时常量池是方法区的一部分）

#### 程序计数器

这是一块很小的区域，它可以被看作是当前线程执行的字节码的行号指示器。在虚拟机的模型里，字节码指示器就是通过改变程序计数器的值来指定下一条需要执行的指令。

由于虚拟机的多线程是通过轮流切换分配处理器来完成的，所以同一时刻只能有一个线程在执行，在同一时刻也只能执行一个线程的一条指令。因此，在时间片结束后，为了线程能够恢复正确的执行位置，所以每个线程都有一个单独的程序计数器。所以**程序计数器是线程私有的。**

如果当前正在执行的是一个Java方法，那么程序计数器指示的就是字节码指令的地址；如果执行的是一个本地方法，那么程序计数器指示为空。

程序计数器是唯一一个没有定义OutOfMemory的区域。

#### 虚拟机栈

该区域是线程私有的，它是Java方法执行时的内存模型。每一个java方法被执行时，都会创建一个栈帧用于存储java方法的局部变量表、操作栈、方法出口等。每一个java方法的执行就是一个栈帧在虚拟机栈中入栈和出栈的过程。

**局部变量表**存放的是基本数据类型、对象的引用、方法的出口。局部变量表所需要的空间在编译的时候就会分配完成，需要多大的空间是完全确定的，运行期间不会改变。

可能出现的异常：

1. StackOverflowError
2. OutOfMemory

#### 本地方法栈

和虚拟机栈基本一样，本地方法栈执行的本地方法。

#### 堆

堆是Java虚拟机管理的最大的一块内存区域，也是GC管理的主要区域，堆的主要作用就是存放对象实例。

#### 方法区

方法区是各个线程共享的内存区域。用来存放已经被虚拟机加载的类信息（包括类的名字、类的方法、类的成员变量）、常量、静态变量、编译器编译后的代码等数据。当方法区无法满足内存分配需求时，抛出OutOfMemoryError错误。

##### 运行时常量池

**运行时常量池**也是方法区的一部分。存放的数据一般是如下的两种：

1. .class文件里的字面值以及常量（这些是在编译期间产生的）

例如：字面值：private String name = “name”;

常量：private final int age = 11;

1. 常量不一定是在编译期间产生，在运行过程中也会产生。用的最多就是

String str = “abc”.intern()

## String StringBuffer StringBuilder

首先大致的说一下这三者的区别，然后再详细的进行解释。

String类型的值是不可变的，每次对String对象的操作都会产生新的对象，这样不仅效率低下，而且浪费内存空间。

StringBuffer和StringBuilder大致是一样的。

相同点：

1. 对象指向的字符串对象是可变的，也就是说任何对该对象指向的字符串的操作都不会产生新的对象，省去了因为重新分配内存和新建对象带来的问题。

2. 有一定的缓冲容量，当字符串的大小没有超过该容量时，不会重新分配容量。当字符串的大小超过了这个容量，会自动增加容量。

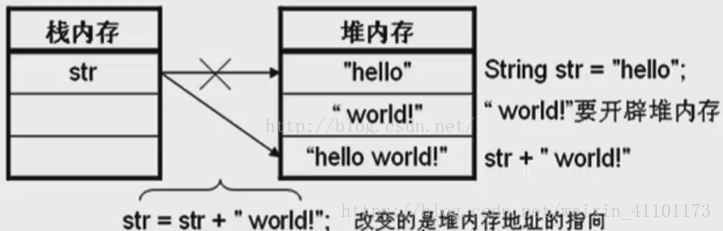
不同点：

StringBuffer是线程安全的，StringBuilder是线程不安全的。

下面详细的说明一下各自的特点。

### String

例如：



这个说明的很明显了，如果不明白，还做什么Java开发。

### StringBuffer

### StringBuilder

# 设计模式

## 代理模式

### 静态代理

### 动态代理