**runnableTaskQueue**（任务队列）：用于**保存等待执行的任务的阻塞队列**。 可以选择以下几个阻塞队列：

**BlockingQueue**的几个注意点

【1】BlockingQueue 可以是**限定容量的**。它在任意给定时间都可以有一个remainingCapacity，超出此容量，便无法无阻塞地put 附加元素。没有任何内部容量约束的BlockingQueue 总是报告Integer.MAX\_VALUE 的剩余容量。

【2】BlockingQueue 实现**主要用于生产者-使用者队列**，但它另外还支持Collection 接口。

【3】BlockingQueue **实现是线程安全的**。

【4】BlockingQueue 实质上不支持使用任何一种“close”或“shutdown”操作来指示不再添加任何项。

**1)ArrayBlockingQueue:**规定大小的BlockingQueue,其构造函数**必须带一个int参数来指明其大小**.**其所含的对象是以FIFO(先入先出)顺序排序的.**

1：它是有界阻塞队列。**它是数组实现的，**是一个典型的“有界缓存区”。**数组大小在构造函数指定，**而且从此以后不可改变。

2:是它线程安全的，是阻塞的，具体参考BlockingQueue的“注意4”。

3:**不接受 null 元素**

4：**公平性 (fairness)**可以在构造函数中指定。如果为**true**，则**按照 FIFO** 顺序访问插入或移除时受阻塞线程的队列；如果为 **false，则访问顺序是不确定**的。

5:它实现了BlockingQueue接口。关于BlockingQueue，请参照《[BlockingQueue](https://link.jianshu.com?t=http:/hubingforever.blog.163.com/blog/static/17104057920107415915820/" \t "_blank)》

6：此类及其迭代器实现了 Collection 和 Iterator 接口的所有可选 方法。

7：其容量在构造函数中指定。容量不可以自动扩展，也没提供手动扩展的接口。

8：在JDK5/6中，LinkedBlockingQueue和ArrayBlocingQueue等对象的poll(long timeout, TimeUnit unit)存在内存泄露Leak的对象是AbstractQueuedSynchronizer.Node，

**2)LinkedBlockingQueue:**大小不定的BlockingQueue,**若**其构造函数**带一个规定大小的参数,生成的BlockingQueue有大小限制**,若**不带**大小参数,所生成的BlockingQueue的大小**由Integer.MAX\_VALUE来决定**.其所含的对象是以**FIFO(先入先出)**顺序排序的并发库中的[BlockingQueue](https://link.jianshu.com?t=http:/java.sun.com/j2se/1.5.0/docs/api/java/util/concurrent/BlockingQueue.html)是一个比较好玩的类，顾名思义，就是阻塞队列。该类主要提供了两个方法put()和take()，前者将一个对象放到队列中，如果队列已经满了，就等待直到有空闲节点；后者从head取一个对象，如果没有对象，就等待直到有可取的对象。

**3)PriorityBlockingQueue:**类似于LinkedBlockQueue,但其所含对象的排序不是FIFO,而是**依据对象的自然排序顺序或者是构造函数的Comparator决定的顺序**.

1：它是无界阻塞队列，容量是无限的，它使用与类PriorityQueue相同的顺序规则。

2:它是线程安全的，是阻塞的

3：不允许使用 null 元素。

4：对于put(E o)和offer(E o, long timeout, TimeUnit unit)，由于该队列是无界的，所以此方法永远不会阻塞。因此参数timeout和unit没意义，会被忽略掉。

5:iterator() 方法中所提供的迭代器并不保证以特定的顺序遍历 PriorityBlockingQueue 的元素。

如果需要有序地遍历，则应考虑使用 Arrays.sort(pq.toArray())。

6.至于使用和别的BlockingQueue（ArrayBlockingQueue，LinkedBlockingQueue）相似，可以参照它们。7：此类及其迭代器实现了 Collection 和 Iterator 接口的所有可选 方法。

**4)SynchronousQueue:**特殊的BlockingQueue,对其的操作必须是**放和取交替完成的**.（**每个插入操作必须等到另一个线程调用移除操作，否则插入操作一直处于阻塞状态**，）此队列不允许 null 元素。

SynchronousQueue的定义如下

public classSynchronousQueueextends[AbstractQueue](https://link.jianshu.com?t=file:/D:/ProgramFiles/JDK7-API-DOC/api/java/util/AbstractQueue.html)implements[BlockingQueue](https://link.jianshu.com?t=file:/D:/ProgramFiles/JDK7-API-DOC/api/java/util/concurrent/BlockingQueue.html),[Serializable](https://link.jianshu.com?t=file:/D:/ProgramFiles/JDK7-API-DOC/api/java/io/Serializable.html)

从上面可以看出，它实现BlockingQueue，所以是阻塞队列，从名字看，它又是同步的。

它模拟的功能类似于生活中一手交钱一手交货这种情形，像那种货到付款或者先付款后发货模型不适合使用SynchronousQueue。

首先要知道**SynchronousQueue没有容纳元素的能力，即它的isEmpty()方法总是返回true**

另外在创建SynchronousQueue时可以传递一个boolean参数来指定它是否是访问它的线程按遵守FIFO顺序处理，true表示遵守FIFO。

**5)DelayQueue**

是一个无界的BlockingQueue，用于放置实现了Delayed接口的对象，其中的**对象只能在其到期时才能从队列中取走。这种队列是有序的，**即队头对象的延迟到期时间最长。注意：不能将null元素放置到这种队列中。

**四种线程池各自的特点及所用到的队列**

**newCachedThreadPool()**缓存型池子，先查看池中有没有以前建立的线程，如果有，就reuse.如果没有，就建一个新的线程加入池中。能reuse的线程，必须是timeout IDLE内的池中线程，缺省timeout是60s,超过这个IDLE时长，线程实例将被终止及移出池。缓存型池子通常用于执行一些生存期很短的异步型任务 。**所用队列为SynchronousQueue()**

**newFixedThreadPool()**fixedThreadPool与cacheThreadPool差不多，也是能reuse就用，但不能随时建新的线程 其独特之处:任意时间点，最多只能有固定数目的活动线程存在，此时如果有新的线程要建立，只能放在另外的队列中等待，直到当前的线程中某个线程终止直接被移出池子。和cacheThreadPool不同：fixedThreadPool池线程数固定，但是0秒IDLE（无IDLE）。这也就意味着创建的线程会一直存在。所以fixedThreadPool多数针对一些很稳定很固定的正规并发线程，多用于服务器。***所用队列为*LinkedBlockingQueue()**

**newScheduledThreadPool()**调度型线程池。这个池子里的线程可以按schedule依次delay执行，或周期执行 。0秒IDLE（无IDLE）。**所用队列为DelayedWorkQueue()**

**newSingleThreadExecutor()**单例线程，任意时间池中只能有一个线程 。用的是和cache池和fixed池相同的底层池，但线程数目是1-1，0秒IDLE（无IDLE）。***所用队列为*LinkedBlockingQueue()**

# 前言

SynchronousQueue是一个比较特别的队列，由于在线程池方面有所应用，为了更好的理解线程池的实现原理，笔者花了些时间学习了一下该队列源码(JDK1.8)，此队列源码中充斥着大量的CAS语句，理解起来是有些难度的，为了方便日后回顾，本篇文章会以简洁的图形化方式展示该队列底层的实现原理。

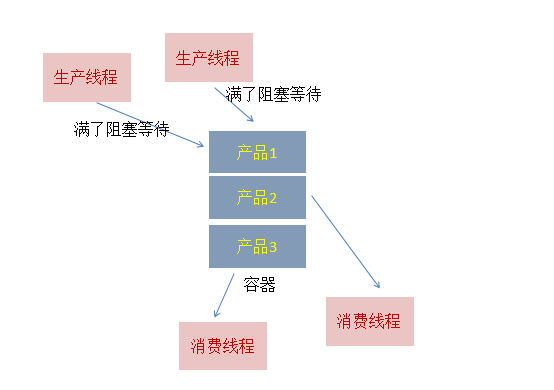
# SynchronousQueue简单使用

经典的生产者-消费者模式，操作流程是这样的：

有多个生产者，可以并发生产产品，把产品置入队列中，如果队列满了，生产者就会阻塞；

有多个消费者，并发从队列中获取产品，如果队列空了，消费者就会阻塞；

如下面的示意图所示：

 SynchronousQueue 也是一个队列来的，但它的特别之处在于它内部没有容器，一个生产线程，当它生产产品（即put的时候），如果当前没有人想要消费产品(即当前没有线程执行take)，此生产线程必须阻塞，等待一个消费线程调用take操作，take操作将会唤醒该生产线程，同时消费线程会获取生产线程的产品（即数据传递），这样的一个过程称为一次配对过程(当然也可以先take后put,原理是一样的)。

我们用一个简单的代码来验证一下，如下所示：

package com.concurrent;

import java.util.concurrent.SynchronousQueue;

public class SynchronousQueueDemo {

public static void main(String[] args) throws InterruptedException {

final SynchronousQueue<Integer> queue = new SynchronousQueue<Integer>();

Thread putThread = new Thread(new Runnable() {

@Override

public void run() {

System.out.println("put thread start");

try {

queue.put(1);

} catch (InterruptedException e) {

}

System.out.println("put thread end");

}

});

Thread takeThread = new Thread(new Runnable() {

@Override

public void run() {

System.out.println("take thread start");

try {

System.out.println("take from putThread: " + queue.take());

} catch (InterruptedException e) {

}

System.out.println("take thread end");

}

});

putThread.start();

Thread.sleep(1000);

takeThread.start();

}

}

put thread start

take thread start

take from putThread: 1

put thread end

take thread end

从结果可以看出，put线程执行queue.put(1) 后就被阻塞了，只有take线程进行了消费，put线程才可以返回。可以认为这是一种线程与线程间一对一传递消息的模型。

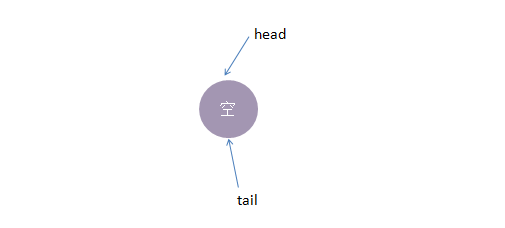
# SynchronousQueue实现原理

不像ArrayBlockingQueue、LinkedBlockingDeque之类的阻塞队列依赖AQS实现并发操作，SynchronousQueue直接使用CAS实现线程的安全访问。由于源码中充斥着大量的CAS代码，不易于理解，所以按照笔者的风格，接下来会使用简单的示例来描述背后的实现模型。

队列的实现策略通常分为公平模式和非公平模式，接下来将分别进行说明。

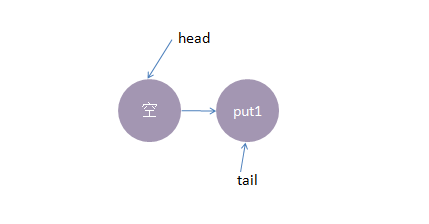
### 公平模式下的模型：

公平模式下，底层实现使用的是TransferQueue这个内部队列，它有一个head和tail指针，用于指向当前正在等待匹配的线程节点。   
初始化时，TransferQueue的状态如下：

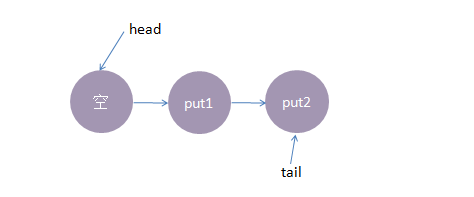
 

接着我们进行一些操作：

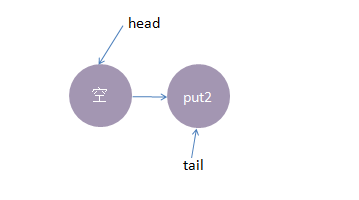
1、线程put1执行 put(1)操作，由于当前没有配对的消费线程，所以put1线程入队列，自旋一小会后睡眠等待，这时队列状态如下：

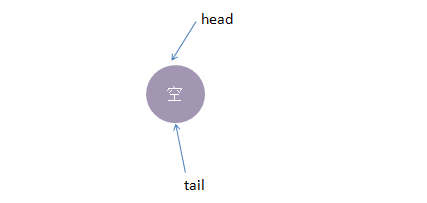
2、接着，线程put2执行了put(2)操作，跟前面一样，put2线程入队列，自旋一小会后睡眠等待，这时队列状态如下：

3、这时候，来了一个线程take1，执行了 take操作，由于tail指向put2线程，put2线程跟take1线程配对了(一put一take)，这时take1线程不需要入队，但是请注意了，这时候，要唤醒的线程并不是put2，而是put1。为何？ 大家应该知道我们现在讲的是公平策略，所谓公平就是谁先入队了，谁就优先被唤醒，我们的例子明显是put1应该优先被唤醒。至于读者可能会有一个疑问，明明是take1线程跟put2线程匹配上了，结果是put1线程被唤醒消费，怎么确保take1线程一定可以和次首节点(head.next)也是匹配的呢？其实大家可以拿个纸画一画，就会发现真的就是这样的。   
公平策略总结下来就是：队尾匹配队头出队。   
执行后put1线程被唤醒，take1线程的 take()方法返回了1(put1线程的数据)，这样就实现了线程间的一对一通信，这时候内部状态如下：

4、最后，再来一个线程take2，执行take操作，这时候只有put2线程在等候，而且两个线程匹配上了，线程put2被唤醒，   
take2线程take操作返回了2(线程put2的数据)，这时候队列又回到了起点，如下所示：

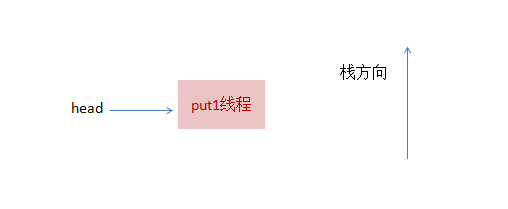
 

以上便是公平模式下，SynchronousQueue的实现模型。总结下来就是：队尾匹配队头出队，先进先出，体现公平原则。

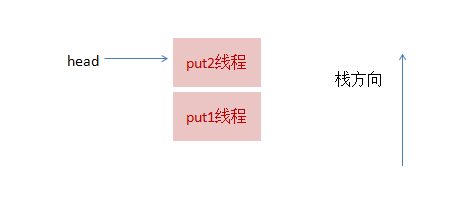
### 非公平模式下的模型：

我们还是使用跟公平模式下一样的操作流程，对比两种策略下有何不同。非公平模式底层的实现使用的是TransferStack，   
一个栈，实现中用head指针指向栈顶，接着我们看看它的实现模型:

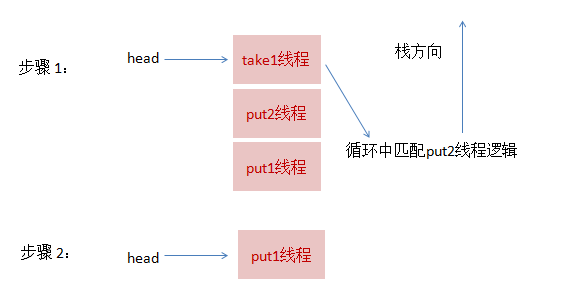
1、线程put1执行 put(1)操作，由于当前没有配对的消费线程，所以put1线程入栈，自旋一小会后睡眠等待，这时栈状态如下：

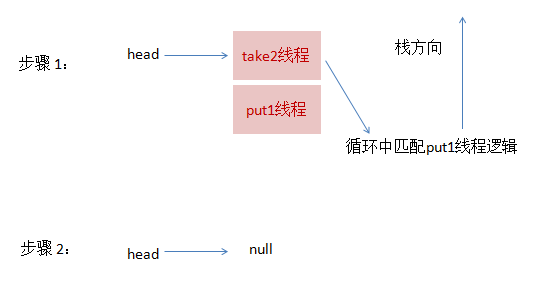
2、接着，线程put2再次执行了put(2)操作，跟前面一样，put2线程入栈，自旋一小会后睡眠等待，这时栈状态如下：



3、这时候，来了一个线程take1，执行了take操作，这时候发现栈顶为put2线程，匹配成功，但是实现会先把take1线程入栈，然后take1线程循环执行匹配put2线程逻辑，一旦发现没有并发冲突，就会把栈顶指针直接指向 put1线程

4、最后，再来一个线程take2，执行take操作，这跟步骤3的逻辑基本是一致的，take2线程入栈，然后在循环中匹配put1线程，最终全部匹配完毕，栈变为空，恢复初始状态，如下图所示：

可以从上面流程看出，虽然put1线程先入栈了，但是却是后匹配，这就是非公平的由来。

# 总结

SynchronousQueue由于其独有的线程一一配对通信机制，在大部分平常开发中，可能都不太会用到，但线程池技术中会有所使用，由于内部没有使用AQS，而是直接使用CAS，所以代码理解起来会比较困难，但这并不妨碍我们理解底层的实现模型，在理解了模型的基础上，有兴趣的话再查阅源码，就会有方向感，看起来也会比较容易，希望本文有所借鉴意义。

**SynchronousQueue其实就是比其他LinkedBlockingQueue这些少了一个容器，装不了产品，只有指针来指向线程，一存一取的模式，使得要不就是生产者（put）阻塞，要不就是消费者（take）阻塞。**