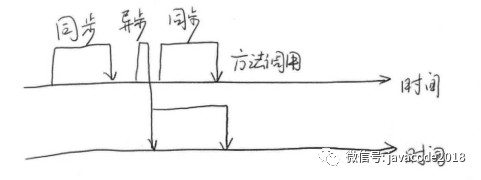
# Java并发的相关概念

## 同步（Synchronous）和异步（Asynchronous）

**同步和异步通常来形容一次方法调用**，同步方法调用一旦开始，调用者必须等到方法调用返回后，才能继续后续的行为。异步方法调用更像一个消息传递，一旦开始，方法调用就会立即返回，调用者就可以继续后续的操作。而异步方法通常会在另外一个线程中“真实”地执行。整个过程，不会阻碍调用者的工作。

如图：



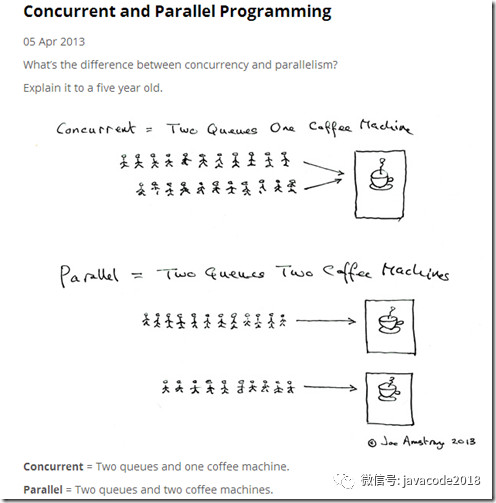
上图中显示了同步方法调用和异步方法调用的区别。对于调用者来说，异步调用似乎是一瞬间就完成的。如果异步调用需要返回结果，那么当这个异步调用真实完成时，则会通知调用者。

打个比方，比如购物，如果你去商场买空调，当你到了商场看重了一款空调，你就向售货员下单。售货员去仓库帮你调配物品。这天你热的是在不行了，就催着商家赶紧给你送货，于是你就在商店里面候着他们，直到商家把你和空调一起送回家，一次愉快的购物就结束了。这就是同步调用。

不过，如果我们赶时髦，就坐在家里打开电脑，在电脑上订购了一台空调。当你完成网上支付的时候，对你来说购物过程已经结束了。虽然空调还没有送到家，但是你的任务已经完成了。商家接到你的订单后，就会加紧安平送货，当然这一切已经跟你无关了。你已经支付完成，想干什么就能去干什么，出去溜几圈都不成问题，等送货上门的时候，接到商家的电话，回家一趟签收就完事了。这就是异步调用。

## 并发（Concurrency）和并行（Parallelism）

并发和并行是两个非常容易被混淆的概念。他们都可以表示两个或者多个任务一起执行，但是侧重点有所不同。**并发偏重于多个任务交替执行，而多个任务之间有可能还是串行的，而并行是真正意义上的“同时执行”**，下图很好地诠释了这点。



大家排队在一个咖啡机上接咖啡，交替执行，是并发；两台咖啡机上面接咖啡，是并行。

从严格意义上来说，并行的多任务是真的同时执行，而对于并发来说，这个过程只是交替的，一会执行任务A，一会执行任务B，系统会不停地在两者之间切换。但对于外部观察者来说，即使多个任务之间是串行并发的，也会造成多任务间并行执行的错觉。

**并发说的是在一个时间段内，多件事情在这个时间段内交替执行。**

**并行说的是多件事情在同一个时刻同时发生。**

实际上，如果系统内只有一个CPU，而使用多进程或者多线程任务，那么真实环境中这些任务不可能是真实并行的，毕竟一个CPU一次只能执行一条指令，在这种情况下多进程或者多线程就是并发的，而不是并行的（操作系统会不停地切换多任务）。真实的并行也只可能出现在拥有多个CPU的系统中（比如多核CPU）。

## 临界区

**临界区用来表示一种公共资源或者说共享数据**，可以被多个线程使用，但是每一次只能有一个线程使用它，一旦临界区资源被占用，其他线程要想使用这个资源就必须等待。

比如，一个办公室里有一台打印机，打印机一次只能执行一个任务。如果小王和小明同时需要打印文件，很明显，如果小王先发了打印任务，打印机就开始打印小王的文件，小明的任务就只能等待小王打印结束后才能打印，这里的打印机就是一个临界区的例子。

在并行程序中，临界区资源是保护的对象，如果意外出现打印机同时执行两个任务的情况，那么最有可能的结果就是打印出来的文件是损坏的文件，它既不是小王想要的，也不是小明想要的。

## 阻塞（Blocking）和非阻塞（Non-Blocking）

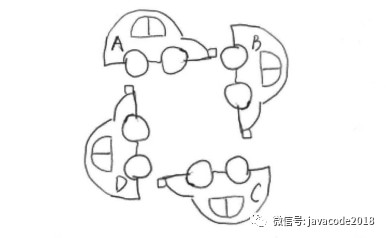
**阻塞和非阻塞通常用来形容很多线程间的相互影响。**比如一个线程占用了临界区资源，那么其他所有需要这个资源的线程就必须在这个临界区中等待。等待会导致线程挂起，这种情况就是阻塞。此时，如果占用资源的线程一直不愿意释放资源，那么其他线程阻塞在这个临界区上的线程都不能工作。

非阻塞的意思与之相反，它强调没有一个线程可以妨碍其他线程执行，所有的线程都会尝试不断向前执行。

## 死锁（Deadlock）、饥饿（Starvation）和活锁（Livelock）

**死锁、饥饿和活锁都属于多线程的活跃性问题**。如果发现上述几种情况，那么相关线程就不再活跃，也就是说它可能很难再继续往下执行了。

死锁应该是最糟糕的一种情况了（当然，其他几种情况也好不到哪里去），如下图显示了一个死锁的发生：



A、B、C、D四辆小车都在这种情况下都无法继续行驶了。他们彼此之间相互占用了其他车辆的车道，如果大家都不愿意释放自己的车道，那么这个状况将永远持续下去，谁都不可能通过，死锁是一个很严重的并且应该避免和实时小心的问题，后面的文章中会做更详细的讨论。

**饥饿是指某一个或者多个线程因为种种原因无法获得所要的资源，导致一直无法执行。**比如它的优先级可能太低，而高优先级的线程不断抢占它需要的资源，导致低优先级线程无法工作。在自然界中，母鸡给雏鸟喂食很容易出现这种情况：由于雏鸟很多，食物有限，雏鸟之间的事务竞争可能非常厉害，经常抢不到事务的雏鸟有可能被饿死。线程的饥饿非常类似这种情况。此外，某一个线程一直占着关键资源不放，导致其他需要这个资源的线程无法正常执行，这种情况也是饥饿的一种。于死锁相比，饥饿还是有可能在未来一段时间内解决的（比如，高优先级的线程已经完成任务，不再疯狂执行）。

活锁是一种非常有趣的情况。不知道大家是否遇到过这么一种场景，当你要做电梯下楼时，电梯到了，门开了，这时你正准备出去。但很不巧的是，门外一个人当着你的去路，他想进来。于是，你很礼貌地靠左走，礼让对方。同时，对方也非常礼貌的靠右走，希望礼让你。结果，你们俩就又撞上了。于是乎，你们都意识到了问题，希望尽快避让对方，你立即向右边走，同时，他立即向左边走。结果，又撞上了！不过介于人类的智慧，我相信这个动作重复两三次后，你应该可以顺利解决这个问题。因为这个时候，大家都会本能地对视，进行交流，保证这种情况不再发生。但如果这种情况发生在两个线程之间可能就不那么幸运了。如果线程智力不够。且都秉承着“谦让”的原则，主动将资源释放给他人使用，那么久会导致资源不断地在两个线程间跳动，而没有一个线程可以同时拿到所有资源正常执行。这种情况就是活锁。

### 死锁的例子

package com.jvm.visualvm;

public class Demo {

public static void main(String[] args) {

Obj1 obj1 = new Obj1();

Obj2 obj2 = new Obj2();

Thread thread1 = new Thread(new SynAddRunalbe(obj1, obj2, 1, 2, true));

thread1.setName("thread1");

thread1.start();

Thread thread2 = new Thread(new SynAddRunalbe(obj1, obj2, 2, 1, false));

thread2.setName("thread2");

thread2.start();

}

/\*\*

\* 线程死锁等待演示

\*/

public static class SynAddRunalbe implements Runnable {

Obj1 obj1;

Obj2 obj2;

int a, b;

boolean flag;

public SynAddRunalbe(Obj1 obj1, Obj2 obj2, int a, int b, boolean flag) {

this.obj1 = obj1;

this.obj2 = obj2;

this.a = a;

this.b = b;

this.flag = flag;

}

@Override

public void run() {

try {

if (flag) {

synchronized (obj1) {

Thread.sleep(100);

synchronized (obj2) {

System.out.println(a + b);

}

}

} else {

synchronized (obj2) {

Thread.sleep(100);

synchronized (obj1) {

System.out.println(a + b);

}

}

}

} catch (InterruptedException e) {

e.printStackTrace();

}

}

}

public static class Obj1 {

}

public static class Obj2 {

}

}

运行上面代码，可以通过jstack查看到死锁信息:

"thread2" #13 prio=5 os\_prio=0 tid=0x0000000029225000 nid=0x3c94 waiting for monitor entry [0x0000000029c9f000]

java.lang.Thread.State: BLOCKED (on object monitor)

at com.jvm.visualvm.Demo4$SynAddRunalbe.run(Demo4.java:50)

- waiting to lock <0x00000007173d40f0> (a com.jvm.visualvm.Demo4$Obj1)

- locked <0x00000007173d6310> (a com.jvm.visualvm.Demo4$Obj2)

at java.lang.Thread.run(Thread.java:745)

Locked ownable synchronizers:

- None

"thread1" #12 prio=5 os\_prio=0 tid=0x0000000029224800 nid=0x6874 waiting for monitor entry [0x0000000029b9f000]

java.lang.Thread.State: BLOCKED (on object monitor)

at com.jvm.visualvm.Demo4$SynAddRunalbe.run(Demo4.java:43)

- waiting to lock <0x00000007173d6310> (a com.jvm.visualvm.Demo4$Obj2)

- locked <0x00000007173d40f0> (a com.jvm.visualvm.Demo4$Obj1)

at java.lang.Thread.run(Thread.java:745)

Locked ownable synchronizers:

- None

thread1持有com.jvm.visualvm.Demo4$Obj1的锁，等待获取com.jvm.visualvm.Demo4$Obj2的锁 thread2持有com.jvm.visualvm.Demo4$Obj2的锁，等待获取com.jvm.visualvm.Demo4$Obj1的锁，两个线程相互等待获取对方持有的锁，出现死锁。

### ****饥饿死锁的例子****

package com.jvm.jconsole;

import java.util.concurrent.\*;

public class ExecutorLock {

private static ExecutorService single = Executors.newSingleThreadExecutor();

public static class AnotherCallable implements Callable<String> {

@Override

public String call() throws Exception {

System.out.println("in AnotherCallable");

return "annother success";

}

}

public static class MyCallable implements Callable<String> {

@Override

public String call() throws Exception {

System.out.println("in MyCallable");

Future<String> submit = single.submit(new AnotherCallable());

return "success:" + submit.get();

}

}

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

MyCallable task = new MyCallable();

Future<String> submit = single.submit(task);

System.out.println(submit.get());

System.out.println("over");

single.shutdown();

}

}

执行代码，输出：

in MyCallable

使用jstack命令查看线程堆栈信息：

"pool-1-thread-1" #12 prio=5 os\_prio=0 tid=0x0000000028e3d000 nid=0x58a4 waiting on condition [0x00000000297ff000]

java.lang.Thread.State: WAITING (parking)

at sun.misc.Unsafe.park(Native Method)

- parking to wait for <0x0000000717921bf0> (a java.util.concurrent.FutureTask)

at java.util.concurrent.locks.LockSupport.park(LockSupport.java:175)

at java.util.concurrent.FutureTask.awaitDone(FutureTask.java:429)

at java.util.concurrent.FutureTask.get(FutureTask.java:191)

at com.jvm.jconsole.ExecutorLock$MyCallable.call(ExecutorLock.java:25)

at com.jvm.jconsole.ExecutorLock$MyCallable.call(ExecutorLock.java:20)

at java.util.concurrent.FutureTask.run(FutureTask.java:266)

at java.util.concurrent.ThreadPoolExecutor.runWorker(ThreadPoolExecutor.java:1142)

at java.util.concurrent.ThreadPoolExecutor$Worker.run(ThreadPoolExecutor.java:617)

at java.lang.Thread.run(Thread.java:745)

Locked ownable synchronizers:

- <0x00000007173f2690> (a java.util.concurrent.ThreadPoolExecutor$Worker)

"main" #1 prio=5 os\_prio=0 tid=0x00000000033e4000 nid=0x5f94 waiting on condition [0x00000000031fe000]

java.lang.Thread.State: WAITING (parking)

at sun.misc.Unsafe.park(Native Method)

- parking to wait for <0x00000007173f1d48> (a java.util.concurrent.FutureTask)

at java.util.concurrent.locks.LockSupport.park(LockSupport.java:175)

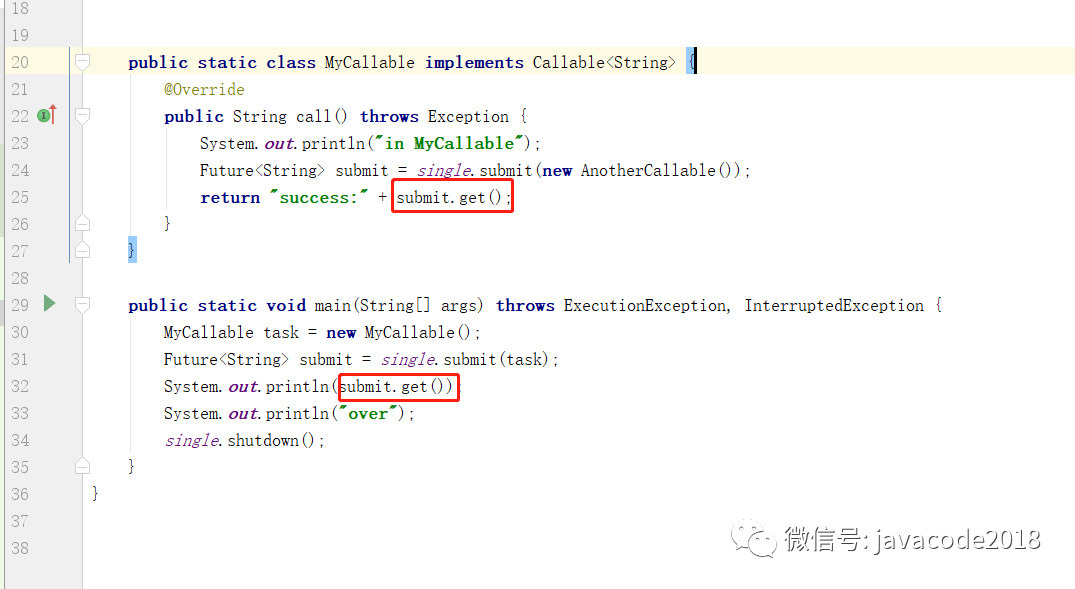
at java.util.concurrent.FutureTask.awaitDone(FutureTask.java:429)

at java.util.concurrent.FutureTask.get(FutureTask.java:191)

at com.jvm.jconsole.ExecutorLock.main(ExecutorLock.java:32)

Locked ownable synchronizers:

- None



堆栈信息结合图中的代码，可以看出主线程在32行处于等待中，线程池中的工作线程在25行处于等待中，等待获取结果。由于线程池是一个线程，AnotherCallable得不到执行，而被饿死，最终导致了程序死锁的现象。