第三版：并发编程 70 道

线程 B 怎么知道线程 A 修改了变量

1、 volatile 修饰变量

2、 synchronized 修饰修改变量的方法

3、 wait/notify

4、 while 轮询

当一个线程进入一个对象的 synchronized 方法 A 之后，其它线程是否可进入此对象的 synchronized 方法 B?

不能。其它线程只能访问该对象的非同步方法，同步方法则不能进入。因为非静态方法上的 synchronized 修饰符要求执行方法时要获得对象的锁，如果已经进入 A 方法说明对象锁已经被取走，那么试图进入 B 方法的线程就只能在等锁池（注意不是等待池哦）中等待对象的锁。

synchronized、volatile、CAS 比较

1、 synchronized 是悲观锁，属于抢占式，会引起其他线程阻塞。

2、 volatile 提供多线程共享变量可见性和禁止指令重排序优化。

3、 CAS 是基于冲突检测的乐观锁（非阻塞）

synchronized 和 Lock 有什么区别?

1、 首先 synchronized 是 Java 内置关键字，在 JVM 层面，Lock 是个 Java 类；

2、 synchronized 可以给类、方法、代码块加锁；而 lock 只能给代码块加锁。

3、 synchronized 不需要手动获取锁和释放锁，使用简单，发生异常会自动释放锁，不会造成死锁；而 lock 需要自己加锁和释放锁，如果使用不当没有 unLock()去释放锁就会造成死锁。

4、 通过 Lock 可以知道有没有成功获取锁，而 synchronized 却无法办到。

synchronized 和 ReentrantLock 区别是什么?

1、 synchronized 是和 if、else、for、while 一样的关键字，ReentrantLock 是类，这是二者的本质区别。既然 ReentrantLock 是类，那么它就提供了比 synchronized 更多更灵活的特性，可以被继承、可以有方法、可以有各种各样的类变量

2、 synchronized 早期的实现比较低效，对比 ReentrantLock，大多数场景性能都相差较大，但是在 Java 6 中对 synchronized 进行了非常多的改进。

相同点：两者都是可重入锁

两者都是可重入锁。“可重入锁”概念是：自己可以再次获取自己的内部锁。比如一个线程获得了某个对象的锁，此时这个对象锁还没有释放，当其再次想要获取这个对象的锁的时候还是可以获取的，如果不可锁重入的话，就会造成死锁。

同一个线程每次获取锁，锁的计数器都自增 1，所以要等到锁的计数器下降为 0 时才能释放锁。

主要区别如下：

1、 ReentrantLock 使用起来比较灵活，但是必须有释放锁的配合动作；

2、 ReentrantLock 必须手动获取与释放锁，而 synchronized 不需要手动释放和开启锁；

3、 ReentrantLock 只适用于代码块锁，而 synchronized 可以修饰类、方法、变量等。

4、 二者的锁机制其实也是不一样的。ReentrantLock 底层调用的是 Unsafe 的 park 方法加锁，synchronized 操作的应该是对象头中 mark word

5、 Java 中每一个对象都可以作为锁，这是 synchronized 实现同步的基础：

1、 普通同步方法，锁是当前实例对象

2、 静态同步方法，锁是当前类的 class 对象

3、 同步方法块，锁是括号里面的对象

volatile 关键字的作用

1、 对于可见性，Java 提供了 volatile 关键字来保证可见性和禁止指令重排。volatile 提供 happens-before 的保证，确保一个线程的修改能对其他线程是可见的。当一个共享变量被 volatile 修饰时，它会保证修改的值会立即被更新到主内存中，当有其他线程需要读取时，它会去内存中读取新值。

2、 从实践角度而言，volatile 的一个重要作用就是和 CAS 结合，保证了原子性，详细的可以参见 java.util.concurrent.atomic 包下的类，比如 AtomicInteger。

3、 volatile 常用于多线程环境下的单次操作(单次读或者单次写)。

Java 中能创建 volatile 数组吗?

能，Java 中可以创建 volatile 类型数组，不过只是一个指向数组的引用，而不是整个数组。意思是，如果改变引用指向的数组，将会受到 volatile 的保护，但是如果多个线程同时改变数组的元素，volatile 标示符就不能起到之前的保护作用了。

volatile 变量和 atomic 变量有什么不同?

volatile 变量可以确保先行关系，即写操作会发生在后续的读操作之前，但它并不能保证原子性。例如用 volatile 修饰 count 变量，那么 count++ 操作就不是原子性的。

而 AtomicInteger 类提供的 atomic 方法可以让这种操作具有原子性如 getAndIncrement()方法会原子性的进行增量操作把当前值加一，其它数据类型和引用变量也可以进行相似操作。

volatile 能使得一个非原子操作变成原子操作吗?

1、 关键字 volatile 的主要作用是使变量在多个线程间可见，但无法保证原子性，对于多个线程访问同一个实例变量需要加锁进行同步。

2、 虽然 volatile 只能保证可见性不能保证原子性，但用 volatile 修饰 long 和 double 可以保证其操作原子性。

所以从 Oracle Java Spec 里面可以看到：

1、 对于 64 位的 long 和 double，如果没有被 volatile 修饰，那么对其操作可以不是原子的。在操作的时候，可以分成两步，每次对 32 位操作。

2、 如果使用 volatile 修饰 long 和 double，那么其读写都是原子操作

3、 对于 64 位的引用地址的读写，都是原子操作

4、 在实现 JVM 时，可以自由选择是否把读写 long 和 double 作为原子操作

5、 推荐 JVM 实现为原子操作

synchronized 和 volatile 的区别是什么?

1、 synchronized 表示只有一个线程可以获取作用对象的锁，执行代码，阻塞其他线程。

2、 volatile 表示变量在 CPU 的寄存器中是不确定的，必须从主存中读取。保证多线程环境下变量的可见性；禁止指令重排序。

区别

1、 volatile 是变量修饰符；synchronized 可以修饰类、方法、变量。

2、 volatile 仅能实现变量的修改可见性，不能保证原子性；而 synchronized 则可以保证变量的修改可见性和原子性。

3、 volatile 不会造成线程的阻塞；synchronized 可能会造成线程的阻塞。

4、 volatile 标记的变量不会被编译器优化；synchronized 标记的变量可以被编译器优化。

5、 volatile 关键字是线程同步的轻量级实现，所以 volatile 性能肯定比 synchronized 关键字要好。但是 volatile 关键字只能用于变量而 synchronized 关键字可以修饰方法以及代码块。synchronized 关键字在 JavaSE1.6 之后进行了主要包括为了减少获得锁和释放锁带来的性能消耗而引入的偏向锁和轻量级锁以及其它各种优化之后执行效率有了显著提升，实际开发中使用 synchronized 关键字的场景还是更多一些。

final 不可变对象，它对写并发应用有什么帮助?

1、 不可变对象(Immutable Objects)即对象一旦被创建它的状态（对象的数据，也即对象属性值）就不能改变，反之即为可变对象(Mutable Objects)。

2、 不可变对象的类即为不可变类(Immutable Class)。Java 平台类库中包含许多不可变类，如 String、基本类型的包装类、BigInteger 和 BigDecimal 等。

3、 只有满足如下状态，一个对象才是不可变的；

4、 它的状态不能在创建后再被修改；

5、 所有域都是 final 类型；并且，它被正确创建（创建期间没有发生 this 引用的逸出）。

不可变对象保证了对象的内存可见性，对不可变对象的读取不需要进行额外的同步手段，提升了代码执行效率。

Lock 接口和 synchronized 对比同步它有什么优势?

Lock 接口比同步方法和同步块提供了更具扩展性的锁操作。他们允许更灵活的结构，可以具有完全不同的性质，并且可以支持多个相关类的条件对象。

它的优势有：

1、 可以使锁更公平

2、 可以使线程在等待锁的时候响应中断

3、 可以让线程尝试获取锁，并在无法获取锁的时候立即返回或者等待一段时间

4、 可以在不同的范围，以不同的顺序获取和释放锁

整体上来说 Lock 是 synchronized 的扩展版，Lock 提供了无条件的、可轮询的(tryLock 方法)、定时的(tryLock 带参方法)、可中断的(lockInterruptibly)、可多条件队列的(newCondition 方法)锁操作。另外 Lock 的实现类基本都支持非公平锁(默认)和公平锁，synchronized 只支持非公平锁，当然，在大部分情况下，非公平锁是高效的选择。

乐观锁和悲观锁的理解及如何实现，有哪些实现方式?

悲观锁：

总是假设最坏的情况，每次去拿数据的时候都认为别人会修改，所以每次在拿数据的时候都会上锁，这样别人想拿这个数据就会阻塞直到它拿到锁。传统的关系型数据库里边就用到了很多这种锁机制，比如行锁，表锁等，读锁，写锁等，都是在做操作之前先上锁。再比如 Java 里面的同步原语 synchronized 关键字的实现也是悲观锁。

乐观锁：

顾名思义，就是很乐观，每次去拿数据的时候都认为别人不会修改，所以不会上锁，但是在更新的时候会判断一下在此期间别人有没有去更新这个数据，可以使用版本号等机制。乐观锁适用于多读的应用类型，这样可以提高吞吐量，像数据库提供的类似于 write_condition 机制，其实都是提供的乐观锁。在 Java 中 java.util.concurrent.atomic 包下面的原子变量类就是使用了乐观锁的一种实现方式 CAS 实现的。

什么是 CAS

CAS 是 compare and swap 的缩写，即我们所说的比较交换。

CAS 是一种基于锁的操作，而且是乐观锁。在 java 中锁分为乐观锁和悲观锁。悲观锁是将资源锁住，等一个之前获得锁的线程释放锁之后，下一个线程才可以访问。而乐观锁采取了一种宽泛的态度，通过某种方式不加锁来处理资源，比如通过给记录加 version 来获取数据，性能较悲观锁有很大的提高。

CAS 操作包含三个操作数 —— 内存位置（V）、预期原值（A）和新值(B)。如果内存地址里面的值和 A 的值是一样的，那么就将内存里面的值更新成 B。CAS 是通过无限循环来获取数据的，若果在第一轮循环中，a 线程获取地址里面的值被 b 线程修改了，那么 a 线程需要自旋，到下次循环才有可能机会执行。

java.util.concurrent.atomic 包下的类大多是使用 CAS 操作来实现的(AtomicInteger,AtomicBoolean,AtomicLong)

CAS 的会产生什么问题?

ABA 问题：

比如说一个线程 one 从内存位置 V 中取出 A，这时候另一个线程 two 也从内存中取出 A，并且 two 进行了一些操作变成了 B，然后 two 又将 V 位置的数据变成 A，这时候线程 one 进行 CAS 操作发现内存中仍然是 A，然后 one 操作成功。尽管线程 one 的 CAS 操作成功，但可能存在潜藏的问题。从 Java1.5 开始 JDK 的 atomic 包里提供了一个类 AtomicStampedReference 来解决 ABA 问题。

循环时间长开销大：

对于资源竞争严重（线程冲突严重）的情况，CAS 自旋的概率会比较大，从而浪费更多的 CPU 资源，效率低于 synchronized。

只能保证一个共享变量的原子操作：

当对一个共享变量执行操作时，我们可以使用循环 CAS 的方式来保证原子操作，但是对多个共享变量操作时，循环 CAS 就无法保证操作的原子性，这个时候就可以用锁。

什么是原子类

java.util.concurrent.atomic 包：是原子类的小工具包，支持在单个变量上解除锁的线程安全编程 原子变量类相当于一种泛化的 volatile 变量，能够支持原子的和有条件的读-改-写操作。

比如：

AtomicInteger 表示一个 int 类型的值，并提供了 get 和 set 方法，这些 Volatile 类型的 int 变量在读取和写入上有着相同的内存语义。它还提供了一个原子的 compareAndSet 方法（如果该方法成功执行，那么将实现与读取/写入一个 volatile 变量相同的内存效果），以及原子的添加、递增和递减等方法。AtomicInteger 表面上非常像一个扩展的 Counter 类，但在发生竞争的情况下能提供更高的可伸缩性，因为它直接利用了硬件对并发的支持。

简单来说就是原子类来实现 CAS 无锁模式的算法

原子类的常用类

1、 AtomicBoolean

2、 AtomicInteger

3、 AtomicLong

4、 AtomicReference

说一下 Atomic 的原理?

Atomic 包中的类基本的特性就是在多线程环境下，当有多个线程同时对单个（包括基本类型及引用类型）变量进行操作时，具有排他性，即当多个线程同时对该

变量的值进行更新时，仅有一个线程能成功，而未成功的线程可以向自旋锁一样，继续尝试，一直等到执行成功。

死锁与活锁的区别，死锁与饥饿的区别?

1、 死锁：是指两个或两个以上的进程（或线程）在执行过程中，因争夺资源而造成的一种互相等待的现象，若无外力作用，它们都将无法推进下去。

2、 活锁：任务或者执行者没有被阻塞，由于某些条件没有满足，导致一直重复尝试，失败，尝试，失败。

3、 活锁和死锁的区别在于，处于活锁的实体是在不断的改变状态，这就是所谓的“活”， 而处于死锁的实体表现为等待；活锁有可能自行解开，死锁则不能。

4、 饥饿：一个或者多个线程因为种种原因无法获得所需要的资源，导致一直无法执行的状态。

Java 中导致饥饿的原因：

1、 高优先级线程吞噬所有的低优先级线程的 CPU 时间。

2、 线程被永久堵塞在一个等待进入同步块的状态，因为其他线程总是能在它之前持续地对该同步块进行访问。

3、 线程在等待一个本身也处于永久等待完成的对象(比如调用这个对象的 wait 方法)，因为其他线程总是被持续地获得唤醒。

什么是线程池?

Java 中的线程池是运用场景最多的并发框架，几乎所有需要异步或并发执行任务的程序都可以使用线程池。在开发过程中，合理地使用线程池能够带来许多好处。

1、 降低资源消耗。通过重复利用已创建的线程降低线程创建和销毁造成的消耗。

2、 提高响应速度。当任务到达时，任务可以不需要等到线程创建就能立即执行。

3、 提高线程的可管理性。线程是稀缺资源，如果无限制地创建，不仅会消耗系统资源，还会降低系统的稳定性，使用线程池可以进行统一分配、调优和监控。但是，要做到合理利用

线程池作用?

线程池是为突然大量爆发的线程设计的，通过有限的几个固定线程为大量的操作服务，减少了创建和销毁线程所需的时间，从而提高效率。

如果一个线程所需要执行的时间非常长的话，就没必要用线程池了(不是不能作长时间操作，而是不宜。本来降低线程创建和销毁，结果你那么久我还不好控制还不如直接创建线程)，况且我们还不能控制线程池中线程的开始、挂起、和中止。

线程池有什么优点?

1、 降低资源消耗：重用存在的线程，减少对象创建销毁的开销。

2、 提高响应速度。可有效的控制最大并发线程数，提高系统资源的使用率，同时避免过多资源竞争，避免堵塞。当任务到达时，任务可以不需要的等到线程创建就能立即执行。

3、 提高线程的可管理性。线程是稀缺资源，如果无限制的创建，不仅会消耗系统资源，还会降低系统的稳定性，使用线程池可以进行统一的分配，调优和监控。

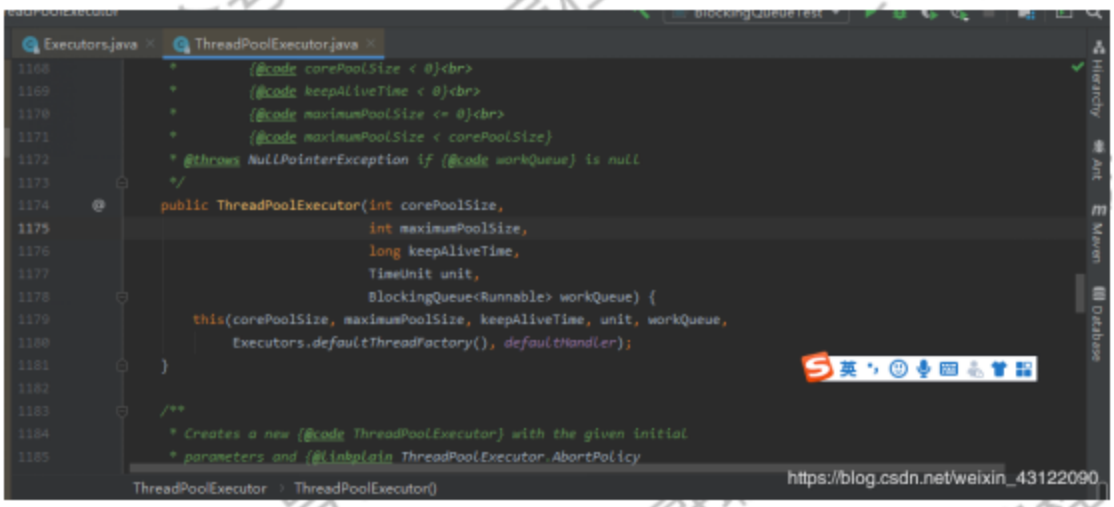
4、 附加功能：提供定时执行、定期执行、单线程、并发数控制等功能。

什么是 ThreadPoolExecutor?

ThreadPoolExecutor 就是线程池

ThreadPoolExecutor 其实也是 JAVA 的一个类，我们一般通过 Executors 工厂类的方法，通过传入不同的参数，就可以构造出适用于不同应用场景下的 ThreadPoolExecutor（线程池）

构造参数图：



构造参数参数介绍：

corePoolSize 核心线程数量
maximumPoolSize 最大线程数量
keepAliveTime 线程保持时间，N 个时间单位
unit 时间单位（比如秒，分）

workQueue 阻塞队列

threadFactory 线程工厂

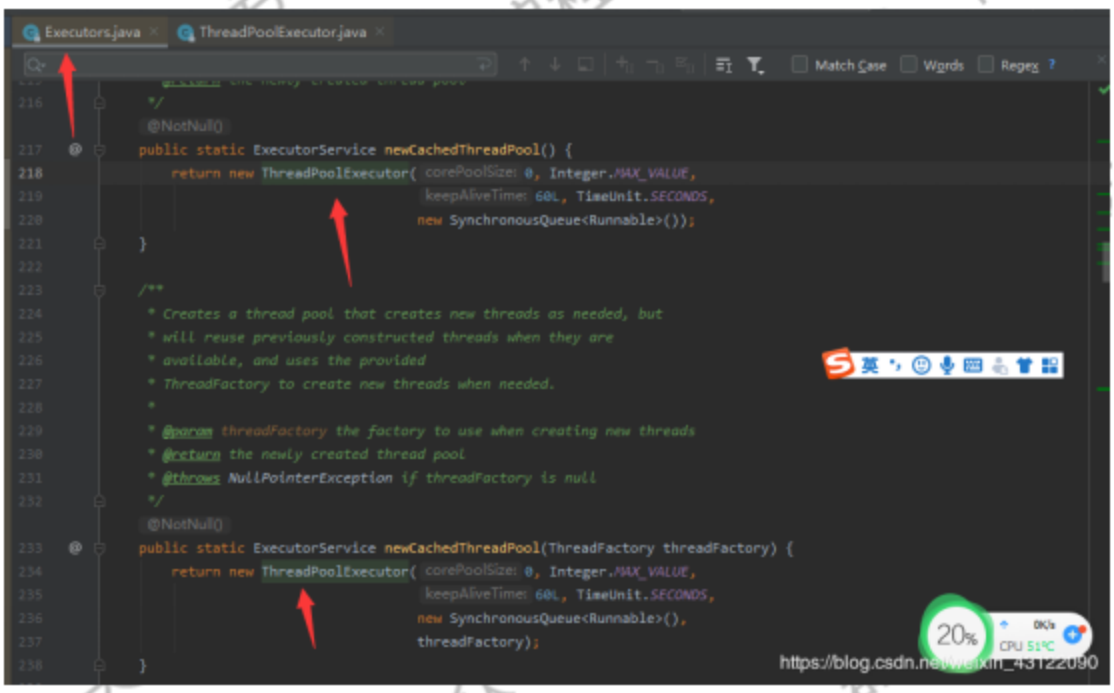
handler 线程池拒绝策略

什么是 Executors?

Executors 框架实现的就是线程池的功能

Executors 工厂类中提供的 newCachedThreadPool、newFixedThreadPool 、newScheduledThreadPool 、newSingleThreadExecutor 等方法其实也只是 ThreadPoolExecutor 的构造函数参数不同而已。通过传入不同的参数，就可以构造出适用于不同应用场景下的线程池，

Executor 工厂类如何创建线程池图：



线程池四种创建方式?

Java 通过 Executors（jdk1.5 并发包）提供四种线程池，分别为：

1、 newCachedThreadPool 创建一个可缓存线程池，如果线程池长度超过处理需要，可灵活回收空闲线程，若无可回收，则新建线程。

2、 newFixedThreadPool 创建一个定长线程池，可控制线程最大并发数，超出的线程会在队列中等待。

3、 newScheduledThreadPool 创建一个定长线程池，支持定时及周期性任务执行。

4、 newSingleThreadExecutor 创建一个单线程化的线程池，它只会用唯一的工作线程来执行任务，保证所有任务按照指定顺序(FIFO, LIFO, 优先级)执行。

在 Java 中 Executor 和 Executors 的区别?

1、 Executors 工具类的不同方法按照我们的需求创建了不同的线程池，来满足业务的需求。

2、 Executor 接口对象能执行我们的线程任务。

3、 ExecutorService 接口继承了 Executor 接口并进行了扩展，提供了更多的方法我们能获得任务执行的状态并且可以获取任务的返回值。

4、 使用 ThreadPoolExecutor 可以创建自定义线程池。

四种构建线程池的区别及特点?

newCachedThreadPool

特点：

newCachedThreadPool 创建一个可缓存线程池，如果当前线程池的长度超过了处理的需要时，它可以灵活的回收空闲的线程，当需要增加时， 它可以灵活的添加新的线程，而不会对池的长度作任何限制

缺点：

他虽然可以无线的新建线程，但是容易造成堆外内存溢出，因为它的最大值是在初始化的时候设置为 Integer.MAX_VALUE，一般来说机器都没那么大内存给它不断使用。当然知道可能出问题的点，就可以去重写一个方法限制一下这个最大值

总结：

线程池为无限大，当执行第二个任务时第一个任务已经完成，会复用执行第一个任务的线程，而不用每次新建线程

代码示例：

```
package com.lijie;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
public class TestNewCachedThreadPool {
    public static void main(String[] args) {
        // 创建无限大小线程池，由 jvm 自动回收
        ExecutorService newCachedThreadPool =
Executors.newCachedThreadPool();
        for (int i = 0; i < 10; i++) {
```

```
                final int temp = i;
                newCachedThreadPool.execute(new Runnable() {
                    public void run() {
                        try {
                            Thread.sleep(100);
                        } catch (Exception e) {}

System.out.println(Thread.currentThread().getName() + ",i==" + temp);
                    }
                });
            }
        }
}
```

newFixedThreadPool

特点：

创建一个定长线程池，可控制线程最大并发数，超出的线程会在队列中等待。定长线程池的大小最好根据系统资源进行设置。

缺点：

线程数量是固定的，但是阻塞队列是无界队列。如果有很多请求积压，阻塞队列越来越长，容易导致 OOM（超出内存空间）

总结：

请求的挤压一定要和分配的线程池大小匹配，定线程池的大小最好根据系统资源进行设置。如 Runtime.getRuntime().availableProcessors()

Runtime.getRuntime().availableProcessors()方法是查看电脑 CPU 核心数量）

代码示例

```
package com.lijie;

import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;

public class TestNewFixedThreadPool {
public static void main(String[] args) {
ExecutorService newFixedThreadPool =
Executors.newFixedThreadPool(3);
for (int i = 0; i < 10; i++) {
final int temp = i;
newFixedThreadPool.execute(new Runnable() {
public void run() {
System.out.println(Thread.currentThread().getName() + ",i==" + temp);
}
});
}
}
}
```

newScheduledThreadPool

特点：

创建一个固定长度的线程池，而且支持定时的以及周期性的任务执行，类似于 Timer（Timer 是 Java 的一个定时器类）

缺点：由于所有任务都是由同一个线程来调度，因此所有任务都是串行执行的，同一时间只能有一个任务在执行，前一个任务的延迟或异常都将会影响到之后的任务（比如：一个任务出错，以后的任务都无法继续）。

代码示例

```
package com.lijie;
import java.util.concurrent.Executors;
import java.util.concurrent.ScheduledExecutorService;
import java.util.concurrent.TimeUnit;
public class TestNewScheduledThreadPool {
    public static void main(String[] args) {
        //定义线程池大小为 3
        ScheduledExecutorService newScheduledThreadPool =
Executors.newScheduledThreadPool(3);
        for (int i = 0; i < 10; i++) {
            final int temp = i;
            newScheduledThreadPool.schedule(new Runnable() {
                public void run() {
                    System.out.println("i:" + temp);
                }
            }, 3, TimeUnit.SECONDS); //这里表示延迟 3 秒执行。
        }
    }
}
```

newSingleThreadExecutor

特点：

创建一个单线程化的线程池，它只会用唯一的工作线程来执行任务，如果这个唯一的线程因为异常结束，那么会有一个新的线程来替代它，他必须保证前一项任务执行完毕才能执行后一项。保证所有任务按照指定顺序(FIFO, LIFO，优先级)执行。

缺点：

缺点的话，很明显，他是单线程的，高并发业务下有点无力

总结：

保证所有任务按照指定顺序执行的，如果这个唯一的线程因为异常结束，那么会有一个新的线程来替代它

代码示例

```
package com.lijie;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
public class TestNewSingleThreadExecutor {
    public static void main(String[] args) {
        ExecutorService newSingleThreadExecutor =
Executors.newSingleThreadExecutor();
        for (int i = 0; i < 10; i++) {
            final int index = i;
            newSingleThreadExecutor.execute(new Runnable() {
                public void run() {

System.out.println(Thread.currentThread().getName() + " index:" +
index);
                    try {
```

```
                    Thread.sleep(200);
                } catch (Exception e) {}
            }
        });
    }
  }
}
```

线程池都有哪些状态?

1、 RUNNING：这是最正常的状态，接受新的任务，处理等待队列中的任务。

2、 SHUTDOWN：不接受新的任务提交，但是会继续处理等待队列中的任务。

3、 STOP：不接受新的任务提交，不再处理等待队列中的任务，中断正在执行任务的线程。

4、 TIDYING：所有的任务都销毁了，workCount 为 0，线程池的状态在转换为 TIDYING 状态时，会执行钩子方法 terminated()。

5、 TERMINATED：terminated()方法结束后，线程池的状态就会变成这个。

线程池中 submit() 和 execute() 方法有什么区别?

相同点：

相同点就是都可以开启线程执行池中的任务。

不同点：

1、 接收参数：execute()只能执行 Runnable 类型的任务。submit()可以执行 Runnable 和 Callable 类型的任务。

2、 返回值：submit()方法可以返回持有计算结果的 Future 对象，而 execute()没有

3、 异常处理：submit()方便 Exception 处理

什么是线程组，为什么在 Java 中不推荐使用?

1、 ThreadGroup 类，可以把线程归属到某一个线程组中，线程组中可以有线程对象，也可以有线程组，组中还可以有线程，这样的组织结构有点类似于树的形式。

2、 线程组和线程池是两个不同的概念，他们的作用完全不同，前者是为了方便线程的管理，后者是为了管理线程的生命周期，复用线程，减少创建销毁线程的开销。

3、 为什么不推荐使用线程组？因为使用有很多的安全隐患吧，没有具体追究，如果需要使用，推荐使用线程池。

ThreadPoolExecutor 饱和策略有哪些?

如果当前同时运行的线程数量达到最大线程数量并且队列也已经被放满了任时，ThreadPoolTaskExecutor 定义一些策略:

1、 ThreadPoolExecutor.AbortPolicy：抛出 RejectedExecutionException 来拒绝新任务的处理。

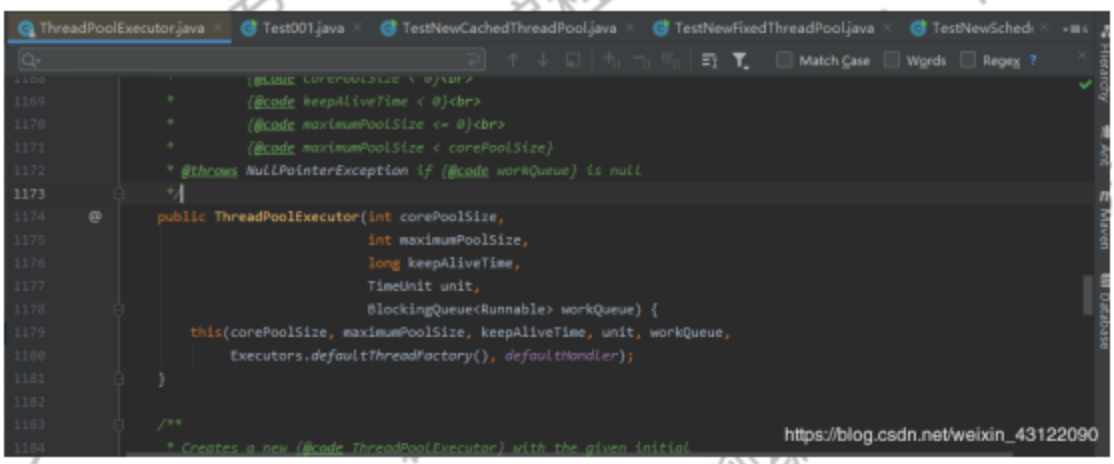
2、 ThreadPoolExecutor.CallerRunsPolicy：调用执行自己的线程运行任务。您不会任务请求。但是这种策略会降低对于新任务提交速度，影响程序的整体性能。另外，这个策略喜欢增加队列容量。如果您的应用程序可以承受此延迟并且你不能任务丢弃任何一个任务请求的话，你可以选择这个策略。

3、 ThreadPoolExecutor.DiscardPolicy：不处理新任务，直接丢弃掉。

4、 ThreadPoolExecutor.DiscardOldestPolicy： 此策略将丢弃最早的未处理的任务请求。

如何自定义线程线程池?

先看 ThreadPoolExecutor（线程池）这个类的构造参数



构造参数参数介绍：

corePoolSize 核心线程数量
maximumPoolSize 最大线程数量
keepAliveTime 线程保持时间，N 个时间单位
unit 时间单位（比如秒，分）
workQueue 阻塞队列

threadFactory 线程工厂

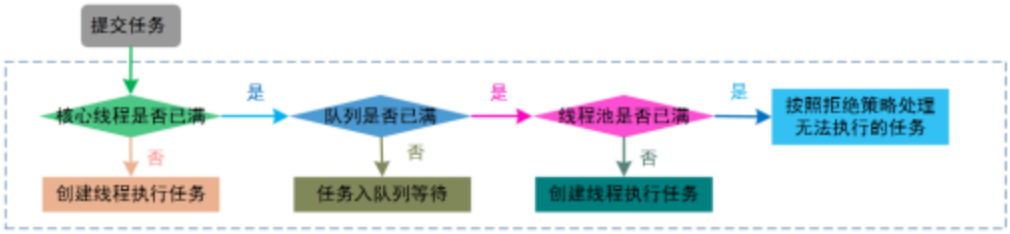
handler 线程池拒绝策略

代码示例：

```
package com.lijie;
import java.util.concurrent.ArrayBlockingQueue;
import java.util.concurrent.ThreadPoolExecutor;
import java.util.concurrent.TimeUnit;
public class Test001 {
    public static void main(String[] args) {
        //创建线程池
        ThreadPoolExecutor executor = new ThreadPoolExecutor(1, 2,
60L, TimeUnit.SECONDS, new ArrayBlockingQueue < > (3));
        for (int i = 1; i <= 6; i++) {
            TaskThred t1 = new TaskThred("任务" + i);
            //executor.execute(t1);是执行线程方法
            executor.execute(t1);
        }
        //executor.shutdown()不再接受新的任务，并且等待之前提交的任务
都执行完再关闭，阻塞队列中的任务不会再执行。
        executor.shutdown();
    }
}
class TaskThred implements Runnable {
    private String taskName;
    public TaskThred(String taskName) {
        this.taskName = taskName;
    }
    public void run() {
```

```
            System.out.println(Thread.currentThread().getName() +
taskName);
        }
}
```

线程池的执行原理?



提交一个任务到线程池中，线程池的处理流程如下：

1、 判断线程池里的核心线程是否都在执行任务，如果不是（核心线程空闲或者还有核心线程没有被创建）则创建一个新的工作线程来执行任务。如果核心线程都在执行任务，则进入下个流程。

2、 线程池判断工作队列是否已满，如果工作队列没有满，则将新提交的任务存储在这个工作队列里。如果工作队列满了，则进入下个流程。

3、 判断线程池里的线程是否都处于工作状态，如果没有，则创建一个新的工作线程来执行任务。如果已经满了，则交给饱和策略来处理这个任务。

如何合理分配线程池大小?

要合理的分配线程池的大小要根据实际情况来定，简单的来说的话就是根据 CPU 密集和 IO 密集来分配

什么是 CPU 密集

CPU 密集的意思是该任务需要大量的运算，而没有阻塞，CPU 一直全速运行。CPU 密集任务只有在真正的多核 CPU 上才可能得到加速(通过多线程)，而在单核 CPU 上，无论你开几个模拟的多线程，该任务都不可能得到加速，因为 CPU 总的运算能力就那样。

什么是 IO 密集

IO 密集型，即该任务需要大量的 IO，即大量的阻塞。在单线程上运行 IO 密集型的任务会导致浪费大量的 CPU 运算能力浪费在等待。所以在 IO 密集型任务中使用多线程可以大大的加速程序运行，即时在单核 CPU 上，这种加速主要就是利用了被浪费掉的阻塞时间。

分配 CPU 和 IO 密集：

1、 CPU 密集型时，任务可以少配置线程数，大概和机器的 cpu 核数相当，这样可以使得每个线程都在执行任务

2、 IO 密集型时，大部分线程都阻塞，故需要多配置线程数，2*cpu 核数

精确来说的话的话：

从以下几个角度分析任务的特性：

1、 任务的性质：CPU 密集型任务、IO 密集型任务、混合型任务。

2、 任务的优先级：高、中、低。

3、 任务的执行时间：长、中、短。

4、 任务的依赖性：是否依赖其他系统资源，如数据库连接等。

可以得出一个结论：

1、 线程等待时间比 CPU 执行时间比例越高，需要越多线程。

2、 线程 CPU 执行时间比等待时间比例越高，需要越少线程。

你经常使用什么并发容器，为什么?

答：Vector、ConcurrentHashMap、HasTable

一般软件开发中容器用的最多的就是 HashMap、ArrayList，LinkedList ，等等

但是在多线程开发中就不能乱用容器，如果使用了未加锁（非同步）的的集合，你的数据就会非常的混乱。由此在多线程开发中需要使用的容器必须是加锁（同步）的容器。

什么是 Vector

Vector 与 ArrayList 一样，也是通过数组实现的，不同的是它支持线程的同步，即某一时刻只有一个线程能够写 Vector，避免多线程同时写而引起的不一致性，但实现同步需要很高的花费，访问它比访问 ArrayList 慢很多

ArrayList 是最常用的 List 实现类，内部是通过数组实现的，它允许对元素进行快速随机访问。当从 ArrayList 的中间位置插入或者删除元素时，需要对数组进行复

制、移动、代价比较高。因此，它适合随机查找和遍历，不适合插入和删除。ArrayList 的缺点是每个元素之间不能有间隔。

ArrayList 和 Vector 有什么不同之处?

Vector 方法带上了 synchronized 关键字，是线程同步的

1、 ArrayList 添加方法源码

```
public boolean add(E e) {
    modCount++;
    add(e, elementData, size);
    return true;
}
```

2、 Vector 添加源码（加锁了 synchronized 关键字）

```
public synchronized boolean add(E e) {
    modCount++;
    add(e, elementData, elementCount);
    return true;
}
```

为什么 HashTable 是线程安全的?

因为 HasTable 的内部方法都被 synchronized 修饰了，所以是线程安全的。其他的都和 HashMap 一样

1、 HashMap 添加方法的源码

```
public V put(K key, V value) { return putVal(hash(key), key, value, onlyIfAbsent: false, evict: true); }

/**
 * Implements Map.put and related methods.
```

2、 HashTable 添加方法的源码

```
public synchronized V put(K key, V value) {
    // Make sure the value is not null
    if (value == null) {
        throw new NullPointerException();
    }
```

用过 ConcurrentHashMap，讲一下他和 HashTable 的不同之处?

ConcurrentHashMap 是 Java5 中支持高并发、高吞吐量的线程安全 HashMap 实现。它由 Segment 数组结构和 HashEntry 数组结构组成。Segment 数组在 ConcurrentHashMap 里扮演锁的角色，HashEntry 则用于存储键-值对数据。一个 ConcurrentHashMap 里包含一个 Segment 数组，Segment 的结构和 HashMap 类似，是一种数组和链表结构；一个 Segment 里包含一个 HashEntry 数组，每个 HashEntry 是一个链表结构的元素；每个 Segment 守护着一个 HashEntry 数组里的元素，当对 HashEntry 数组的数据进行修改时，必须首先获得它对应的 Segment 锁。

看不懂？？？很正常，我也看不懂

总结：

1、 HashTable 就是实现了 HashMap 加上了 synchronized，而 ConcurrentHashMap 底层采用分段的数组+链表实现，线程安全

2、 ConcurrentHashMap 通过把整个 Map 分为 N 个 Segment，可以提供相同的线程安全，但是效率提升 N 倍，默认提升 16 倍。

3、 并且读操作不加锁，由于 HashEntry 的 value 变量是 volatile 的，也能保证读取到最新的值。

4、 Hashtable 的 synchronized 是针对整张 Hash 表的，即每次锁住整张表让线程独占，ConcurrentHashMap 允许多个修改操作并发进行，其关键在于使用了锁分离技术

5、 扩容：段内扩容（段内元素超过该段对应 Entry 数组长度的 75%触发扩容，不会对整个 Map 进行扩容），插入前检测需不需要扩容，有效避免无效扩容

Collections.synchronized 是什么?

**注意： *号代表后面是还有内容的*

此方法是干什么的呢，他完完全全的可以把 List、Map、Set 接口底下的集合变成线程安全的集合

Collections.synchronized * ： 原理是什么，我猜的话是代理模式：

Java 中 ConcurrentHashMap 的并发度是什么?

ConcurrentHashMap 把实际 map 划分成若干部分来实现它的可扩展性和线程安全。这种划分是使用并发度获得的，它是 ConcurrentHashMap 类构造函数的一个可选参数，默认值为 16，这样在多线程情况下就能避免争用。

在 JDK8 后，它摒弃了 Segment（锁段）的概念，而是启用了一种全新的方式实现,利用 CAS 算法。同时加入了更多的辅助变量来提高并发度，具体内容还是查看源码吧。

什么是并发容器的实现?

何为同步容器：可以简单地理解为通过 synchronized 来实现同步的容器，如果有多个线程调用同步容器的方法，它们将会串行执行。比如 Vector，Hashtable，以及 Collections.synchronizedSet，synchronizedList 等方法返回的容器。可以通过查看 Vector，Hashtable 等这些同步容器的实现代码，可以看到这些容器实现线程安全的方式就是将它们的状态封装起来，并在需要同步的方法上加上关键字 synchronized。

并发容器使用了与同步容器完全不同的加锁策略来提供更高的并发性和伸缩性，例如在 ConcurrentHashMap 中采用了一种粒度更细的加锁机制，可以称为分段锁，在这种锁机制下，允许任意数量的读线程并发地访问 map，并且执行读操作的线程和写操作的线程也可以并发的访问 map，同时允许一定数量的写操作线程并发地修改 map，所以它可以在并发环境下实现更高的吞吐量。

Java 中的同步集合与并发集合有什么区别?

同步集合与并发集合都为多线程和并发提供了合适的线程安全的集合，不过并发集合的可扩展性更高。在 Java1.5 之前程序员们只有同步集合来用且在多线程并发的时候会导致争用，阻碍了系统的扩展性。Java5 介绍了并发集合像 ConcurrentHashMap，不仅提供线程安全还用锁分离和内部分区等现代技术提高了可扩展性。

SynchronizedMap 和 ConcurrentHashMap 有什么区别?

1、 SynchronizedMap 一次锁住整张表来保证线程安全，所以每次只能有一个线程来访为 map。

2、 ConcurrentHashMap 使用分段锁来保证在多线程下的性能。

3、 ConcurrentHashMap 中则是一次锁住一个桶。ConcurrentHashMap 默认将 hash 表分为 16 个桶，诸如 get，put，remove 等常用操作只锁当前需要用到的桶。

4、 这样，原来只能一个线程进入，现在却能同时有 16 个写线程执行，并发性能的提升是显而易见的。

5、 另外 ConcurrentHashMap 使用了一种不同的迭代方式。在这种迭代方式中，当 iterator 被创建后集合再发生改变就不再是抛出 ConcurrentModificationException，取而代之的是在改变时 new 新的数据从而不影响原有的数据，iterator 完成后再将头指针替换为新的数据 ，这样 iterator 线程可以使用原来老的数据，而写线程也可以并发的完成改变。

CopyOnWriteArrayList 是什么?

CopyOnWriteArrayList 是一个并发容器。有很多人称它是线程安全的，我认为这句话不严谨，缺少一个前提条件，那就是非复合场景下操作它是线程安全的。

CopyOnWriteArrayList(免锁容器)的好处之一是当多个迭代器同时遍历和修改这个列表时，不会抛出 ConcurrentModificationException。在 CopyOnWriteArrayList 中，写入将导致创建整个底层数组的副本，而源数组将保留在原地，使得复制的数组在被修改时，读取操作可以安全地执行。

CopyOnWriteArrayList 的使用场景?

合适读多写少的场景。

CopyOnWriteArrayList 的缺点?

1、 由于写操作的时候，需要拷贝数组，会消耗内存，如果原数组的内容比较多的情况下，可能导致 young gc 或者 full gc。

2、 不能用于实时读的场景，像拷贝数组、新增元素都需要时间，所以调用一个 set 操作后，读取到数据可能还是旧的，虽然 CopyOnWriteArrayList 能做到最终一致性,但是还是没法满足实时性要求。

3、 由于实际使用中可能没法保证 CopyOnWriteArrayList 到底要放置多少数据，万一数据稍微有点多，每次 add/set 都要重新复制数组，这个代价实在太高昂了。在高性能的互联网应用中，这种操作分分钟引起故障。

CopyOnWriteArrayList 的设计思想?

1、 读写分离，读和写分开

2、 最终一致性

3、 使用另外开辟空间的思路，来解决并发冲突

什么是并发队列：

消息队列很多人知道：

消息队列是分布式系统中重要的组件，是系统与系统直接的通信。

并发队列是什么：

并发队列多个线程以有次序共享数据的重要组件

并发队列和并发集合的区别：

那就有可能要说了，我们并发集合不是也可以实现多线程之间的数据共享吗，其实也是有区别的：

队列遵循“先进先出”的规则，可以想象成排队检票，队列一般用来解决大数据量采集处理和显示的。

并发集合就是在多个线程中共享数据的

怎么判断并发队列是阻塞队列还是非阻塞队列

在并发队列上 JDK 提供了 Queue 接口，一个是以 Queue 接口下的 BlockingQueue 接口为代表的阻塞队列，另一个是高性能（无堵塞）队列。

阻塞队列和非阻塞队列区别

1、 当队列阻塞队列为空的时，从队列中获取元素的操作将会被阻塞。

2、 或者当阻塞队列是满时，往队列里添加元素的操作会被阻塞。

3、 或者试图从空的阻塞队列中获取元素的线程将会被阻塞，直到其他的线程往空的队列插入新的元素。

4、 试图往已满的阻塞队列中添加新元素的线程同样也会被阻塞，直到其他的线程使队列重新变得空闲起来

常用并发列队的介绍：

非堵塞队列：

ArrayDeque，（数组双端队列）

ArrayDeque （非堵塞队列）是 JDK 容器中的一个双端队列实现，内部使用数组进行元素存储，不允许存储 null 值，可以高效的进行元素查找和尾部插入取出，是用作队列、双端队列、栈的绝佳选择，性能比 LinkedList 还要好。

PriorityQueue，（优先级队列）

PriorityQueue （非堵塞队列） 一个基于优先级的无界优先级队列。优先级队列的元素按照其自然顺序进行排序，或者根据构造队列时提供的 Comparator 进行排序，具体取决于所使用的构造方法。该队列不允许使用 null 元素也不允许插入不可比较的对象

ConcurrentLinkedQueue，（基于链表的并发队列）

ConcurrentLinkedQueue （非堵塞队列）：是一个适用于高并发场景下的队列，通过无锁的方式，实现了高并发状态下的高性能。ConcurrentLinkedQueue 的性能要好于 BlockingQueue 接口，它是一个基于链接节点的无界线程安全队列。该队列的元素遵循先进先出的原则。该队列不允许 null 元素。

堵塞队列：

DelayQueue，（基于时间优先级的队列，延期阻塞队列）

DelayQueue 是一个没有边界 BlockingQueue 实现，加入其中的元素必需实现 Delayed 接口。当生产者线程调用 put 之类的方法加入元素时，会触发 Delayed 接口中的 compareTo 方法进行排序，也就是说队列中元素的顺序是按到期时间排序的，而非它们进入队列的顺序。排在队列头部的元素是最早到期的，越往后到期时间越晚。

ArrayBlockingQueue，（基于数组的并发阻塞队列）

ArrayBlockingQueue 是一个有边界的阻塞队列，它的内部实现是一个数组。有边界的意思是它的容量是有限的，我们必须在其初始化的时候指定它的容量大小，容量大小一旦指定就不可改变。ArrayBlockingQueue 是以先进先出的方式存储数据

LinkedBlockingQueue，（基于链表的 FIFO 阻塞队列）

LinkedBlockingQueue 阻塞队列大小的配置是可选的，如果我们初始化时指定一个大小，它就是有边界的，如果不指定，它就是无边界的。说是无边界，其实是采用了默认大小为 Integer.MAX_VALUE 的容量 。它的内部实现是一个链表。

LinkedBlockingDeque，（基于链表的 FIFO 双端阻塞队列）

LinkedBlockingDeque 是一个由链表结构组成的双向阻塞队列，即可以从队列的两端插入和移除元素。双向队列因为多了一个操作队列的入口，在多线程同时入队时，也就减少了一半的竞争。

相比于其他阻塞队列，LinkedBlockingDeque 多了 addFirst、addLast、peekFirst、peekLast 等方法，以 first 结尾的方法，表示插入、获取获移除双端队列的第一个元素。以 last 结尾的方法，表示插入、获取获移除双端队列的最后一个元素。

LinkedBlockingDeque 是可选容量的，在初始化时可以设置容量防止其过度膨胀，如果不设置，默认容量大小为 Integer.MAX_VALUE。

PriorityBlockingQueue，（带优先级的无界阻塞队列）

priorityBlockingQueue 是一个无界队列，它没有限制，在内存允许的情况下可以无限添加元素；它又是具有优先级的队列，是通过构造函数传入的对象来判断，传入的对象必须实现 comparable 接口。

SynchronousQueue （并发同步阻塞队列）

SynchronousQueue 是一个内部只能包含一个元素的队列。插入元素到队列的线程被阻塞，直到另一个线程从队列中获取了队列中存储的元素。同样，如果线程尝试获取元素并且当前不存在任何元素，则该线程将被阻塞，直到线程将元素插入队列。

将这个类称为队列有点夸大其词。这更像是一个点。

并发队列的常用方法

不管是那种列队，是那个类，当是他们使用的方法都是差不多的

方法名	描述
add()	在不超出队列长度的情况下插入元素，可以立即执行，成功返回 true，如果队列满了就抛出异常。
offer()	在不超出队列长度的情况下插入元素的时候则可以立即在队列的尾部插入指定元素,成功时返回 true，如果此队列已满，则返回 false。
put()	插入元素的时候，如果队列满了就进行等待，直到队列可用。
take()	从队列中获取值，如果队列中没有值，线程会一直阻塞，直到队列中有值，并且该方法取得了该值。
poll(long timeout, TimeUnit unit)	在给定的时间里，从队列中获取值，如果没有取到会抛出异常。
remainingCapacity()	获取队列中剩余的空间。
remove(Object o)	从队列中移除指定的值。
contains(Object o)	判断队列中是否拥有该值。

drainTo(Collection c)	将队列中值，全部移除，并发设置到给定的集合中。

常用的并发工具类有哪些?

CountDownLatch

CountDownLatch 类位于 java.util.concurrent 包下，利用它可以实现类似计数器的功能。比如有一个任务 A，它要等待其他 3 个任务执行完毕之后才能执行，此时就可以利用 CountDownLatch 来实现这种功能了。

CyclicBarrier (回环栅栏) CyclicBarrier 它的作用就是会让所有线程都等待完成后才会继续下一步行动。

CyclicBarrier 初始化时规定一个数目，然后计算调用了 CyclicBarrier.await()进入等待的线程数。当线程数达到了这个数目时，所有进入等待状态的线程被唤醒并继续。

CyclicBarrier 初始时还可带一个 Runnable 的参数， 此 Runnable 任务在 CyclicBarrier 的数目达到后，所有其它线程被唤醒前被执行。

Semaphore (信号量) Semaphore 是 synchronized 的加强版，作用是控制线程的并发数量（允许自定义多少线程同时访问）。就这一点而言，单纯的 synchronized 关键字是实现不了的。

Semaphore 是一种基于计数的信号量。它可以设定一个阈值，基于此，多个线程竞争获取许可信号，做自己的申请后归还，超过阈值后，线程申请许可信号将会被阻塞。Semaphore 可以用来构建一些对象池，资源池之类的，比如数据库连接池，我们也可以创建计数为 1 的 Semaphore，将其作为一种类似互斥锁的机制，这也叫二元信号量，表示两种互斥状态。它的用法如下：