目录

1. 进程和线程的区别
2. 线程和进程的区别？

一个程序至少有一个进程,一个进程至少有一个线程。

（1）定义：

进程：

进程就是在操作系统上执行的一个程序；比如：qq.exe。是具有一定独立功能的程序关于某个数据集合上的一次运行活动,进程是系统进行资源分配和调度的一个独立单位.（进程之间没有关系，都是相对独立的。每个进程独享一部分内存及其他系统资源。操作系统允许多进程（任务）处理模式）

线程：

线程是进程的一个实体表现。是进程的一个实体,是 CPU 调度和分派的基本单位,它是比进程更小的能独立运行的基本单位.线程自己基本上不拥有系统资源,只拥有一点在运行中必不可少的资源(如程序计数器,一组寄存器和栈),但是它可与同属一个进程的其他的线程共享进程所拥有的全部资源.（进程由多个线程组成。一个进程中的多个线程共享该进程的资源。一个进程中的多个线程支持并发（多线程））

（2）关系：

一个线程可以创建和撤销另一个线程;同一个进程中的多个线程之间可以并发执行.

相对进程而言，线程是一个更加接近于执行体的概念，它可以与同进程中的其他线程共享数据，但拥有自己的栈空间，拥有独立的执行序列。

（3）区别：

进程和线程的主要差别在于它们是不同的操作系统资源管理方式。

进程有独立的地址空间，一个进程崩溃后，在保护模式下不会对其它进程产生影响，而线程只是一个进程中的不同执行路径。

线程有自己的堆栈和局部变量，但线程之间没有单独的地址空间，一个线程死掉就等于整个进程死掉，所以多进程的程序要比多线程的程序健壮，但在进程切换时，耗费资源较大，效率要差一些。但对于一些要求同时进行并且又要共享某些变量的并发操作，只能用线程，不能用进程。

注：简而言之,一个程序至少有一个进程,一个进程至少有一个线程.

线程的划分尺度小于进程，使得多线程程序的并发性高。另外，进程在执行过程中拥有独立的内存单元，而多个线程共享内存，从而极大地提高了程序的运行效率。

线程在执行过程中与进程还是有区别的。每个独立的线程有一个

程序运行的入口、顺序执行序列和程序的出口。但是线程不能够独立执行，必须依存在应用程序中，由应用程序提供多个线程执行控制。

从逻辑角度来看，多线程的意义在于一个应用程序中，有多个执行部分可以同时执行。但操作系统并没有将多个线程看做多个独立的应用，来实现进程的调度和管理以及资源分配。这就是进程和线程的重要区别。

（4）优缺点：

线程和进程在使用上各有优缺点：

线程执行开销小，但不利于资源的管理和保护；

而进程正相反。同时，线程适合于在 SMP 机器上运行，而进程则可以跨机器迁移。

3. 开启线程的三种方式？

https://blog.csdn.net/longshengguoji/article/details/41126119

一、继承 Thread 类创建线程类

（1）定义 Thread 类的子类，并重写该类的 run 方法，该 run 方

法的方法体就代表了线程要完成的任务。因此把 run()方法称为执行体。

（2）创建 Thread 子类的实例，即创建了线程对象。

（3）调用线程对象的 start()方法来启动该线程。

```
package com.thread;

public class FirstThreadTest extends Thread{
        int i = 0;
        //重写run方法，run方法的方法体就是现场执行体
        public void run()
        {
                for(;i<100;i++){
                System.out.println(getName()+"  "+i);

                }
        }
        public static void main(String[] args)
        {
                for(int i = 0;i< 100;i++)
                {
                        System.out.println(Thread.currentThread().getName()+"  : "+i);
                        if(i==20)
                        {
                                new FirstThreadTest().start();
                                new FirstThreadTest().start();
                        }
                }
        }

}
```

二、通过 Runnable 接口创建线程类

（1）定义 runnable 接口的实现类，并重写该接口的 run()方法，该 run()方法的方法体同样是该线程的线程执行体。

（2）创建 Runnable 实现类的实例，并依此实例作为 Thread 的 target 来创建 Thread 对象，该 Thread 对象才是真正的线程对象。

（3）调用线程对象的 start()方法来启动该线程。

```java
package com.thread;

public class RunnableThreadTest implements Runnable
{

        private int i;
        public void run()
        {
                for(i = 0;i <100;i++)
                {
                        System.out.println(Thread.currentThread().getName()+" "+i);
                }
        }
        public static void main(String[] args)
        {
                for(int i = 0;i < 100;i++)
                {
                        System.out.println(Thread.currentThread().getName()+" "+i);
                        if(i==20)
                        {
                                RunnableThreadTest rtt = new RunnableThreadTest();
                                new Thread(rtt,"新线程1").start();
                                new Thread(rtt,"新线程2").start();
                        }
                }

        }

}
```

三、通过 Callable 和 Future 创建线程

（1）创建 Callable 接口的实现类，并实现 call()方法，该 call()方法将作为线程执行体，并且有返回值。

（2）创建 Callable 实现类的实例，使用 FutureTask 类来包装 Callable 对象，该 FutureTask 对象封装了该 Callable 对象的 call()方法的返回值。

（3）使用 FutureTask 对象作为 Thread 对象的 target 创建并启动新线程。

（4）调用 FutureTask 对象的 get()方法来获得子线程执行结束后的返回值

```
package com.thread;

import java.util.concurrent.Callable;
import java.util.concurrent.ExecutionException;
import java.util.concurrent.FutureTask;

public class CallableThreadTest implements Callable<Integer>
{

	public static void main(String[] args)
	{
		CallableThreadTest ctt = new CallableThreadTest();
		FutureTask<Integer> ft = new FutureTask<>(ctt);
		for(int i = 0;i < 100;i++)
		{
			System.out.println(Thread.currentThread().getName()+" 的循环变量i的值"+i);
			if(i==20)
			{
				new Thread(ft,"有返回值的线程").start();
			}
		}
		try
		{
			System.out.println("子线程的返回值："+ft.get());
		} catch (InterruptedException e)
		{
			e.printStackTrace();
		} catch (ExecutionException e)
		{
			e.printStackTrace();
		}

	}

	@Override
	public Integer call() throws Exception
	{
		int i = 0;
		for(;i<100;i++)
		{
			System.out.println(Thread.currentThread().getName()+" "+i);
		}
		return i;
	}

}
```

创建线程的三种方式的对比：

（1）采用实现 Runnable、Callable 接口的方式创建多线程时，

优势是：

线程类只是实现了 Runnable 接口或 Callable 接口，还可以继承其他类。在这种方式下，多个线程可以共享同一个 target 对象，所以非常适合多个相同线程来处理同一份资源的情况，从而可以将 CPU、代码和数据分开，形成清晰的模型，较好地体现了面向对象的思想。

劣势是：

编程稍微复杂，如果要访问当前线程，则必须使用 Thread.currentThread()方法。

（2）使用继承 Thread 类的方式创建多线程时

优势是：

编写简单，如果需要访问当前线程，则无需使用 Thread.currentThread()方法，直接使用 this 即可获得当前线程。

劣势是：

线程类已经继承了 Thread 类，所以不能再继承其他父类

4. 为什么要有线程，而不是仅仅用进

程？

进程还是有很多缺陷的，主要体现在两点上：

（1）进程只能在一个时间干一件事，如果想同时干两件事或多件事，进程就无能为力了。

（2）进程在执行的过程中如果阻塞，例如等待输入，整个进程就会挂起，即使进程中有些工作不依赖于输入的数据，也将无法执行。

（

如果这两个缺点理解比较困难的话，举个现实的例子也许你就清楚了：

如果把我们上课的过程看成一个进程的话，那么我们要做的是耳朵听老师讲课，手上还要记笔记，脑子还要思考问题，这样才能高效的完成听课的任务。而如果只提供进程这个机制的话，上面这三件事将不能同时执行，同一时间只能做一件事，听的时候就不能记笔记，也不能用脑子思考，这是其一；

如果老师在黑板上写演算过程，我们开始记笔记，而老师突然有一步推不下去了，阻塞住了，他在那边思考着，而我们呢，也不能干其他事，即使你想趁此时思考一下刚才没听懂的一个问题都不行，这是其二。

现在你应该明白了进程的缺陷了，而解决的办法很简单，我们完全可以让听、写、思三个独立的过程，并行起来，这样很明显可以提高听课的效率。而实际的操作系统中，也同样引入了这种类似的机制——

线程。

）

5. 什么导致线程阻塞？

导致线程阻塞的原因主要有以下几方面。

1、线程进行了休眠：线程执行了 Thread.sleep(int n)方法，线程放弃 CPU，睡眠 n 毫秒,然后恢复运行。

2、线程等待获取同步锁才能进行下一步操作：线程要执行一段同步代码，由于无法获得相关的同步锁，只好进入阻塞状态，等到获得了同步锁，才能恢复运行。

3、线程执行 wait（）进入阻塞状态：线程执行了一个对象的 wait()方法，进入阻塞状态，只有等到其他线程执行了该对象的 notify()或 notifyAll()方法，才可能将其唤醒。

4、等待相关资源：线程执行 I/O 操作或进行远程通信时，会因为等待相关的资源而进入阻塞状态。(例如,当线程执行 System.in.read()方法时，如果用户没有向控制台输入数据，则该线程会一直等读到了用户的输入数据才从 read()方法返回。进行远程通信时，在客户程序中，线程在以下情况可能进入阻塞状态。）

5、请求连接时：请求与服务器建立连接时，即当线程执行 Socket 的带参数的构造方法，或执行 Socket 的 connect()方法时，会进入阻塞状态，直到连接成功，此线程才从 Socket 的构造方法或 connect()

方法返回。

6、读取线程等待数据：线程从 Socket 的输入流读取数据时，如果没有足够的数据，就会进入阻塞状态，直到读到了足够的数据，或者到达输入流的末尾，或者出现了异常，才从输入流的 read()方法返回或异常中断。

（

输入流中有多少数据才算足够呢?这要看线程执行的 read()方法的类型。

int read()：只要输入流中有一个字节，就算足够。

int read(byte[] buff)：只要输入流中的字节数目与参数 buff 数组的长度相同，就算足够。

String readLine()：只要输入流中邮一行字符串，就算足够。值得注意的是，InputStream 类并没有 readLine 方法，在过滤流 BufferedReader 类中才有此方法。

）

7、线程写数据时可能会出现：线程向 Socket 的输出流写一批数据时，可能会进入阻塞状态，等到输出了所有的数据，或者出现异常，才从输出流的 write()方法返回或异常中断。

8、调用 Socket 关闭连接时阻塞直到发完数据：调用 Socket 的 setSoLinger()方法设置了关闭 Socket 的延迟时间，那么当线程执行 Socket 的 close 方法时，会进入阻塞状态，直到底层 Socket 发送完所有剩余数据，或者超过了 setSoLinger()方法设置的延迟时间，才

从 close()方法返回。

6. run()和 start()方法区别

多线程原理：相当于玩游戏机，只有一个游戏机（cpu），可是有很多人要玩，于是，start 是排队！等 CPU 选中你就是轮到你，你就 run（），当 CPU 的运行的时间片执行完，这个线程就继续排队，等待下一次的 run（）。

注：调用 start（）后，线程会被放到等待队列，等待 CPU 调度，并不一定要马上开始执行，只是将这个线程置于可动行状态。然后通过 JVM，线程 Thread 会调用 run（）方法，执行本线程的线程体。先调用 start 后调用 run，这么麻烦，为了不直接调用 run？就是为了实现多线程的优点，没这个 start 不行。

1. start（）方法来启动线程，真正实现了多线程运行，直接继续执行以下代码（这时无需等待 run 方法体代码执行完毕）：

```
public class Test {
        public static void main(String[] args) {
                Runner1 runner1 = new Runner1();
                Runner2 runner2 = new Runner2();
//              Thread(Runnable target) 分配新的 Thread 对象。
                Thread thread1 = new Thread(runner1);
                Thread thread2 = new Thread(runner2);
//              thread1.start();
//              thread2.start();
                thread1.run();
                thread2.run();
        }
}

class Runner1 implements Runnable { // 实现了Runnable接口，jdk就知道这个类是一个线程
        public void run() {
                for (int i = 0; i < 100; i++) {
                        System.out.println("进入Runner1运行状态————————" + i);
                }
        }
}

class Runner2 implements Runnable { // 实现了Runnable接口，jdk就知道这个类是一个线程
        public void run() {
                for (int i = 0; i < 100; i++) {
                        System.out.println("进入Runner2运行状态==========" + i);
                }
        }
}
```

（1）通过调用 Thread 类的 start()方法来启动一个线程，这时此线程是处于就绪状态，并没有运行。

（2)然后通过此 Thread 类调用方法 run()来完成其运行操作的，这里方法 run()称为线程体，它包含了要执行的这个线程的内容，Run 方法运行结束，此线程终止，而 CPU 再运行其它线程，

源码：

```
public synchronized void start() {
    //如果存在该线程，则会抛出异常
    checkNotStarted();

    hasBeenStarted = true;

    nativeCreate(this, stackSize, daemon);
}
private native static void nativeCreate(Thread t, long stackSize, boolean daemon);
```

start 是真正调用底层的代码来创建 thread，并从底层调用 run（）方法，执行我们自定义的任务。

2.run（）方法当作普通方法的方式调用，程序还是要顺序执行，还是要等待 run 方法体执行完毕后才可继续执行以下代码：

代码（跟上面一样）

而如果直接用 Run 方法，这只是调用一个方法而已，程序中依然只有主线程--这一个线程，其程序执行路径还是只有一条，这样就没有达到写线程的目的。

源码：

```
public void run() {
    if (target != null) {
        target.run();
    }
}
```

这里的 target 是初始化时候的 runnable，可见这里，只是单纯

调用 runnable 执行，而不是在子线程中运行，所以，当外部直接调用此方法时，相当于在当前线程执行任务，如遇到阻塞任务，也就会阻塞当前任务。

总结：

run()只是在当前线程中执行任务，而 start 才是真正生成 thread，并放在 cpu 中调度。

7. 如何控制某个方法允许并发访问线程的个数？

想控制允许访问线程的个数就要使用到 Semaphore。

Semaphore 有两个方法 semaphore.acquire() 和 semaphore.release()。

semaphore.acquire() ：请求一个信号量，这时候的信号量个数-1（一旦没有可使用的信号量，也即信号量个数变为负数时，再次请求的时候就会阻塞，直到其他线程释放了信号量）。

semaphore.release() 释放一个信号量，此时信号量个数+1。

8. 在 Java 中 wait 和 sleep 方法的不同；

wait 和 sleep 的主要区别:

（1）

调用 wait 方法时，线程在等待的时候会释放掉它所获得的 monitor（监控）

调用 Thread.sleep()方法时，线程在等待的时候仍然会持有 monitor 或者锁。

另外，Java 中的 wait 方法应在同步代码块中调用，但是 sleep 方法不需要。

（2）

另一个区别：

Thread.sleep()方法是一个静态方法，作用在当前线程上；

wait 方法是一个实例方法，并且只能在其他线程调用本实例的 notify()方法时被唤醒。另外，使用 sleep 方法时，被暂停的线程在被唤醒之后会立即进入就绪态（Runnable state），但是使用 wait 方法的时候，被暂停的线程会首先获得锁（译者注：阻塞态），然后再进入就绪态。

所以，根据你的需求，如果你需要暂定你的线程一段特定的时间就使用 sleep()方法，如果你想要实现线程间通信就使用 wait()方法。

下面列出 Java 中 wait 和 sleep 方法的区别：

1. wait 只能在同步（synchronize）环境中被调用，而 sleep 不需要。

2. 进入 wait 状态的线程能够被 notify 和 notifyAll 线程唤醒，但是进入 sleeping 状态的线程不能被 notify 方法唤醒。

3. wait 通常有条件地执行，线程会一直处于 wait 状态，直到某个条件变为真。但是 sleep 仅仅让你的线程进入睡眠状态。

4. wait 方法在进入 wait 状态的时候会释放对象的锁，但是 sleep 方法不会。

5. wait 方法是针对一个被同步代码块加锁的对象，而 sleep 是针对一个线程。

9. 谈谈 wait/notify 关键字的理解

wait（）：

```
public final void wait()  throws InterruptedException,IllegalMonitorStateException
```

该方法用来将当前线程置入休眠状态，直到接到通知或被中断为止。在调用 wait（）之前，线程必须要获得该对象的对象级别锁，即**只能在同步方法或同步块中**调用 wait（）方法。

进入 wait（）方法后，当前线程释放锁。在从 wait（）返回前，线程与其他线程竞争重新获得锁。如果调用 wait（）时，没有持有

适当的锁，则抛出 IllegalMonitorStateException，它是 RuntimeException 的一个子类，因此，不需要 try-catch 结构。

notify（）：

```
public final native void notify()throws IllegalMonitorStateException
```

该方法也要**在同步方法或同步块中**调用，即在调用前，线程也必须要获得该对象的对象级别锁，如果调用 notify（）时没有持有适当的锁，也会抛出 IllegalMonitorStateException。

该方法用来通知那些可能等待该对象的对象锁的其他线程。如果有多个线程等待，则线程规划器任意挑选出其中一个 wait（）状态的线程来发出通知，并使它等待获取该对象的对象锁（notify 后，当前线程不会马上释放该对象锁，wait 所在的线程并不能马上获取该对象锁，要等到程序退出 synchronized 代码块后，当前线程才会释放锁，wait 所在的线程也才可以获取该对象锁），但不惊动其他同样在等待被该对象 notify 的线程们。当第一个获得了该对象锁的 wait 线程运行完毕以后，它会释放掉该对象锁，此时如果该对象没有再次使用 notify 语句，则即便该对象已经空闲，其他 wait 状态等待的线程由于没有得到该对象的通知，会继续阻塞在 wait 状态，直到这个对象发出一个 notify 或 notifyAll。这里需要注意：它们等待的是被 notify 或 notifyAll，而不是锁。这与下面的 notifyAll（）方法执行后的情况不同。

（1）如果线程调用了对象的 wait（）方法，那么线程便会处于该对象的等待池中，等待池中的线程不会去竞争该对象的锁。

（2）当有线程调用了对象的 notifyAll（）方法（唤醒所有 wait 线程）或 notify（）方法（只随机唤醒一个 wait 线程），被唤醒的的线程便会进入该对象的锁池中，锁池中的线程会去竞争该对象锁。

（3）优先级高的线程竞争到对象锁的概率大，**假若某线程没有竞争到该对象锁，它还会留在锁池中，唯有线程再次调用 wait（）方法，它才会重新回到等待池中**。而竞争到对象锁的线程则继续往下执行，直到执行完了 synchronized 代码块，它会释放掉该对象锁，这时锁池中的线程会继续竞争该对象锁。

10. 线程如何关闭？

https://www.jianshu.com/p/536b0df1fd55

1.使用标志位

很简单地设置一个标志位，名称就叫做 isCancelled。启动线程后，定期检查这个标志位。如果 isCancelled=true，那么线程就马上结束。

```
public class MyThread implements Runnable{
   private volatile boolean isCancelled;

   public void run(){
      while(!isCancelled){
```

```
            //do something
        }
    }

    public void cancel(){   isCancelled=true;    }
}
```

注意的是：

isCancelled 需要为 volatile，保证线程读取时 isCancelled 是最新数据。

我以前经常用这种简单方法，在大多时候也很有效，

但并不完善。考虑下，如果线程执行的方法被阻塞，那么如何执行 isCancelled 的检查呢？线程有可能永远不会去检查标志位，也就卡住了。

2. 使用中断

Java 提供了中断机制，Thread 类下有三个重要方法。

(1)public void interrupt()

(2)public boolean isInterrupted()

(3)public static boolean interrupted(); // 清除中断标志，并返回原状态

每个线程都有个 boolean 类型的中断状态。当使用 Thread 的 interrupt()方法时，线程的中断状态会被设置为 true。

下面的例子启动了一个线程，循环执行打印一些信息。使用 isInterrupted()方法判断线程是否被中断，如果是就结束线程。

```
public class InterruptedExample {
```

```
public static void main(String[] args) throws Exception {
    InterruptedExample interruptedExample = new InterruptedExample();
    interruptedExample.start();
}

public void start() {
    MyThread myThread = new MyThread();
    myThread.start();

    try {
        Thread.sleep(3000);
        myThread.cancel();
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
}

private class MyThread extends Thread{

    @Override
    public void run() {
        while (!Thread.currentThread().isInterrupted()) {
```

```
                try {
                    System.out.println("test");
                    Thread.sleep(1000);
                } catch (InterruptedException e) {
                    System.out.println("interrupt");
                    //抛出 InterruptedException 后中断标志被清除，标准做法是再次调用
interrupt 恢复中断
                    Thread.currentThread().interrupt();
                }
            }
            System.out.println("stop");
        }

        public void cancel(){
            interrupt();
        }
    }
}
```

对线程调用 interrupt()方法，不会真正中断正在运行的线程，只是发出一个请求，由线程在合适时候结束自己。

```
//抛出 InterruptedException 后中断标志被清除，标准做法是再次调用 interrupt 恢复中断
Thread.currentThread().interrupt();
```

也就说，需要执行两个 interrupt（）方法，一次是发送一个请求（抛出错误后会将中断状态设为 false），一次是真正的执行调用 interrupt（）恢复中断，使中断状态保持为 true。

例如 Thread.sleep 这个阻塞方法，接收到中断请求，**会抛出 InterruptedException，让上层代码处理**。这个时候，你可以什么都不做，但等于吞掉了中断。

（因为抛出 InterruptedException 后，中断标记会被重新设置为 false！看 sleep()的注释，也强调了这点。

```
@throws  InterruptedException
    if any thread has interrupted the current thread.
    The interrupted status of the current thread is
    cleared when this exception is thrown.public static native void
sleep(long millis) throws InterruptedException;

)
```

记得这个规则：**什么时候都不应该吞掉中断！每个线程都应该有合适的方法响应中断！**

所以在 InterruptedExample 例子里，在接收到中断请求时，标准做法是执行 Thread.currentThread().interrupt()恢复中断，让线程退出。

从另一方面谈起，你不能吞掉中断，也不能中断你不熟悉的线程。如果线程没有响应中断的方法，你无论调用多少次 interrupt()方法，

也像泥牛入海。

例子：

Executor 框架提供了 Java 线程池的能力，ExecutorService 扩展了 Executor，提供了管理线程生命周期的关键能力。其中，ExecutorService.submit 返回了 Future 对象来描述一个线程任务，它有一个 cancel()方法。

```
public class InterruptByFuture {

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

        ExecutorService es = Executors.newSingleThreadExecutor();

        Future<?> task = es.submit(new MyThread());

        try {

            //限定时间获取结果

            task.get(5, TimeUnit.SECONDS);

        } catch (TimeoutException e) {

            //超时触发线程中止

            System.out.println("thread over time");

        } catch (ExecutionException e) {

            throw e;

        } finally {

            boolean mayInterruptIfRunning = true;

            task.cancel(mayInterruptIfRunning);
```

```
        }

    }
```

```
private static class MyThread extends Thread {

        @Override

        public void run() {

            while (!Thread.currentThread().isInterrupted()) {

                try {

                    System.out.println("count");

                    Thread.sleep(1000);

                } catch (InterruptedException e) {

                    System.out.println("interrupt");

                    Thread.currentThread().interrupt();

                }

            }

            System.out.println("thread stop");

        }



        public void cancel() {

            interrupt();

        }
```

```
    }
```

Future 的 get 方法可以传入时间，如果限定时间内没有得到结果，将会抛出 TimeoutException。此时，可以调用 Future 的 cancel()方法，对任务所在线程发出中断请求。

cancel()有个参数 mayInterruptIfRunning，表示任务是否能够接收到中断。

mayInterruptIfRunning=true 时，任务如果在某个线程中运行，那么这个线程能够被中断；

mayInterruptIfRunning=false 时，任务如果还未启动，就不要运行它，应用于不处理中断的任务

要注意，mayInterruptIfRunning=true 表示线程能接收中断，但线程是否实现了中断不得而知。线程要正确响应中断，才能真正被 cancel。

线程池的 shutdownNow()会尝试停止池内所有在执行的线程，原理也是发出中断请求。

11. 讲一下 java 中的同步的方法
12. 如何实现线程同步？

https://www.jianshu.com/p/6542c8a96392

https://www.jianshu.com/p/6542c8a96392

为何要使用同步？

java 允许多线程并发控制，当多个线程同时操作一个可共享的资源变量时（如数据的增删改查），将会导致数据不准确，相互之间产生冲突，

因此加入同步锁以避免在该线程没有完成操作之前，被其他线程的调用，从而保证了该变量的唯一性和准确性。

同步的方式

1. 同步方法

即有 synchronized 关键字修饰的方法。

由于 java 的每个对象都有一个内置锁，当用此关键字修饰方法时，内置锁会保护整个方法。在调用该方法前，需要获得内置锁，否则就处于阻塞状态。

```
public synchronized void save(){  }
```

注： synchronized 关键字也可以修饰静态方法，此时如果调用该静态方法，将会锁住整个类

2. 同步代码块

即有 synchronized 关键字修饰的语句块。

被该关键字修饰的语句块会自动被加上内置锁，从而实现同步

代码如：

```
synchronized(object){

}
```

注：同步是一种高开销的操作，因此应该尽量减少同步的内容。通常没有必要同步整个方法，使用 synchronized 代码块同步关键代码即可。

3.使用特殊域变量(volatile)实现线程同步

a.volatile 关键字为域变量的访问提供了一种免锁机制，

b.使用 volatile 修饰域相当于告诉虚拟机该域可能会被其他线程更新，

c.因此每次使用该域就要重新计算，而不是使用寄存器中的值

d.volatile 不会提供任何原子操作，它也不能用来修饰 final 类型的变量

例如：

在上面的例子当中，只需在 account 前面加上 volatile 修饰，即可实现线程同步。

代码示例

```
class Bank {
```

```
        //需要同步的变量加上 volatile
        private volatile int account = 100;


        public int getAccount() {
            return account;
        }
        //这里不再需要 synchronized
        public void save(int money) {
            account += money;
        }
    }
```

多线程中的非同步问题主要出现在对域的读写上，如果让域自身避免这个问题，则就不需要修改操作该域的方法。

4.使用重入锁实现线程同步

在 JavaSE5.0 中新增了一个 java.util.concurrent 包来支持同步。

ReentrantLock 类是可重入、互斥、实现了 Lock 接口的锁，它与使用 synchronized 方法和快具有相同的基本行为和语义，并且扩展了其能力

ReenreantLock 类的常用方法有：

（1）**ReentrantLock()** ： 创建一个 ReentrantLock 实例

（2）lock() ： 获得锁

（3）unlock() ： 释放锁

```
class Bank {



        private int account = 100;

        //需要声明这个锁

        private Lock lock = new ReentrantLock();

        public int getAccount() {

            return account;

        }

        //这里不再需要 synchronized

        public void save(int money) {

            lock.lock();

            try{

                account += money;

            }finally{

                lock.unlock();

            }

        }

    }
```

注：关于 Lock 对象和 synchronized 关键字的选择：

a. 最好两个都不用，使用一种 java.util.concurrent 包提供

的机制，能够帮助用户处理所有与锁相关的代码。

b. 如果 synchronized 关键字能满足用户的需求，就用 synchronized，因为它能简化代码

c.如果需要更高级的功能，就用 ReentrantLock 类，此时要注意及时释放锁，否则会出现死锁，通常在 finally 代码释放锁

5.使用局部变量实现线程同步

如果使用 ThreadLocal 管理变量，则每一个使用该变量的线程都获得该变量的副本，

副本之间相互独立，这样每一个线程都可以随意修改自己的变量副本，而不会对其他线程产生影响。

ThreadLocal 类的常用方法

（1）ThreadLocal()：创建一个线程本地变量

（2）get()：返回此线程局部变量的当前线程副本中的值

（3）initialValue()：返回此线程局部变量的当前线程的"初始值"

（4）set(T value)：将此线程局部变量的当前线程副本中的值设置为 value

例如：

在上面例子基础上，修改后的代码为：

代码实例：

```
public class Bank{

        //使用 ThreadLocal 类管理共享变量 account

        private static ThreadLocal<Integer> account = new
ThreadLocal<Integer>(){

            @Override

            protected Integer initialValue(){

                return 100;

            }

        };

        public void save(int money){

            account.set(account.get()+money);

        }

        public int getAccount(){

            return account.get();

        }

    }
```

注：ThreadLocal 与同步机制

a. ThreadLocal 与同步机制都是为了解决多线程中相同变量的访问冲突问题。

b. 前者采用以”空间换时间”的方法，后者采用以”时间换空间”的方式

13. 数据一致性如何保证？

https://www.cnblogs.com/jiumao/p/7136631.html

一、基本知识：

1.基本术语

术语	英语单词	术语描述
内存屏障	Memory barriers	是一组处理器指令，用于实现对内存操作的顺序限制
缓冲行	Cache line	缓存中可以分配的最小存储单位。处理器填写缓存线时会加载整个缓存线，需要使用多个主内存读周期
原子操作	Atomic operations	不可中断的一个或一系列操作
缓存行填充	Cache line fill	当处理器识别到从内存中读取操作数可缓存的，处理器读取整个缓存行到适当的缓存（L1、L2、L3 的或所有）
缓存命中	Cache hit	如果进行高速缓存行填充操作的内存位置仍然是下次处理器访问的地址时，处理器从缓存中读取操作数，而不是从内存读取
写命中	Write hit	当处理器将操作数写回到一个内存缓存的区域时，它首先会检查这个缓存的内存地址是否在缓存行中，如果存在一个有效的缓存行，则处理器将这个操作数写回到缓存，而不是写回到内存，这个操作被称为写命中

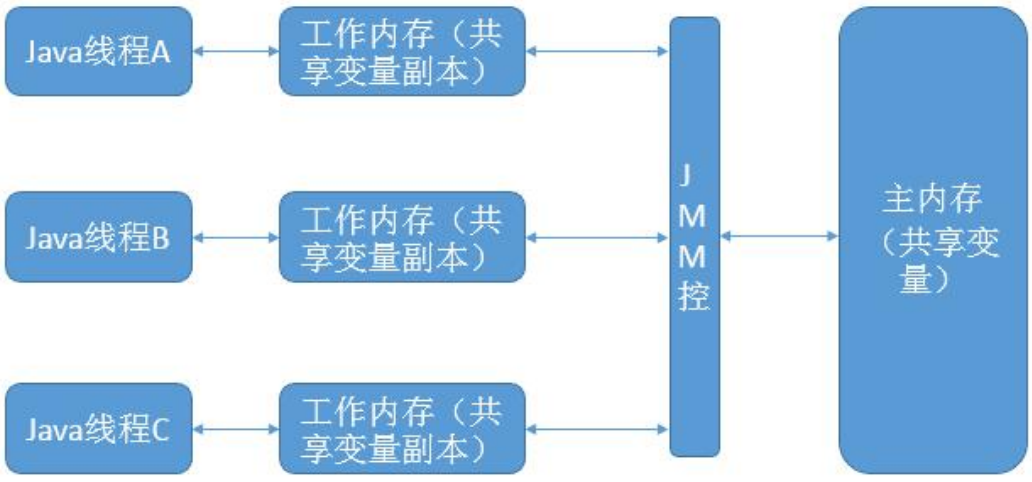
写缺失	Write misses the cache	一个有效的缓存行被写入到不存在的内存区域

2. 内存模型

每一个线程有一个工作内存和主存独立

工作内存存放主存中变量的值的拷贝

Java 内存模型规定了**所有的变量都存储在主内存中。每条线程中还有自己的工作内存，线程的工作内存中保存了被该线程所使用到的变量（这些变量是从主内存中拷贝而来）。线程对变量的所有操作（读取，赋值）都必须在工作内存中进行。不同线程之间也无法直接访问对方工作内存中的变量，线程间变量值的传递均需要通过主内存来完成。**



举个简单的例子：在 java 中，执行下面这个语句：

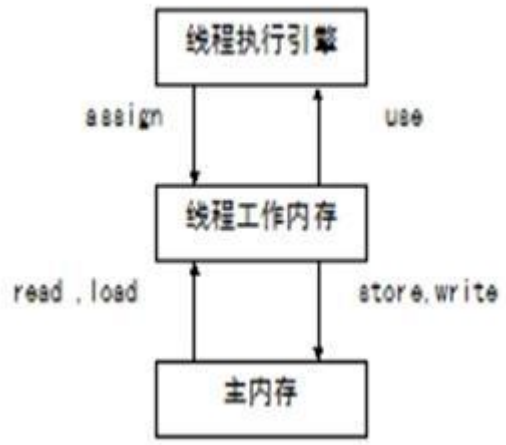
```
i  = 10;
```

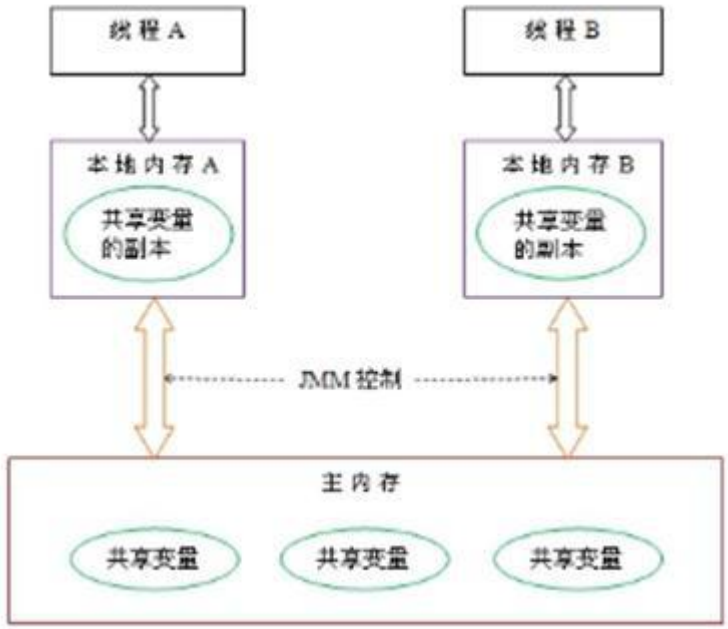
执行线程必须先在自己的工作线程中对变量 i 所在的缓存行进行赋值操作，然后再写入主存当中。而不是直接将数值 10 写入主存当中。

比如同时有 2 个线程执行这段代码，假如初始时 i 的值为 10，那么我们希望两个线程执行完之后 i 的值变为 12。但是事实会是这样吗？

可能存在下面一种情况：初始时，两个线程分别读取 i 的值存入各自所在的工作内存当中，然后线程 1 进行加 1 操作，然后把 i 的最新值 11 写入到内存。此时线程 2 的工作内存当中 i 的值还是 10，进行加 1 操作之后，i 的值为 11，然后线程 2 把 i 的值写入内存。

最终结果 i 的值是 11，而不是 12。这就是著名的缓存一致性问题。通常称这种被多个线程访问的变量为共享变量。





当数据从主内存复制到工作存储时，必须出现两个动作：

1.由主内存执行的读（read）操作；

2.由工作内存执行的相应的 load 操作；当数据从工作内存拷贝到主内存时，也出现两个操作：

1）由工作内存执行的存储（store）操作；

2）由主内存执行的相应的写（write）操作

每一个操作都是原子的，即执行期间不会被中断

对于普通变量，一个线程中更新的值，不能马上反应在其他变量中

如果需要在其他线程中立即可见，需要使用 volatile 关键字

注：CAS(Compare And Swap)

非阻塞同步指令之一，硬件指令集支持。先进行操作，如果有并发操作，则不断重试直到成功。

二、保存一致性可以使用一下几个修饰：

	原子性	可见性	有序性
Volatile	×	√	√
Synchronized	√	√	√
Final	√	√	×

1.final 不可变

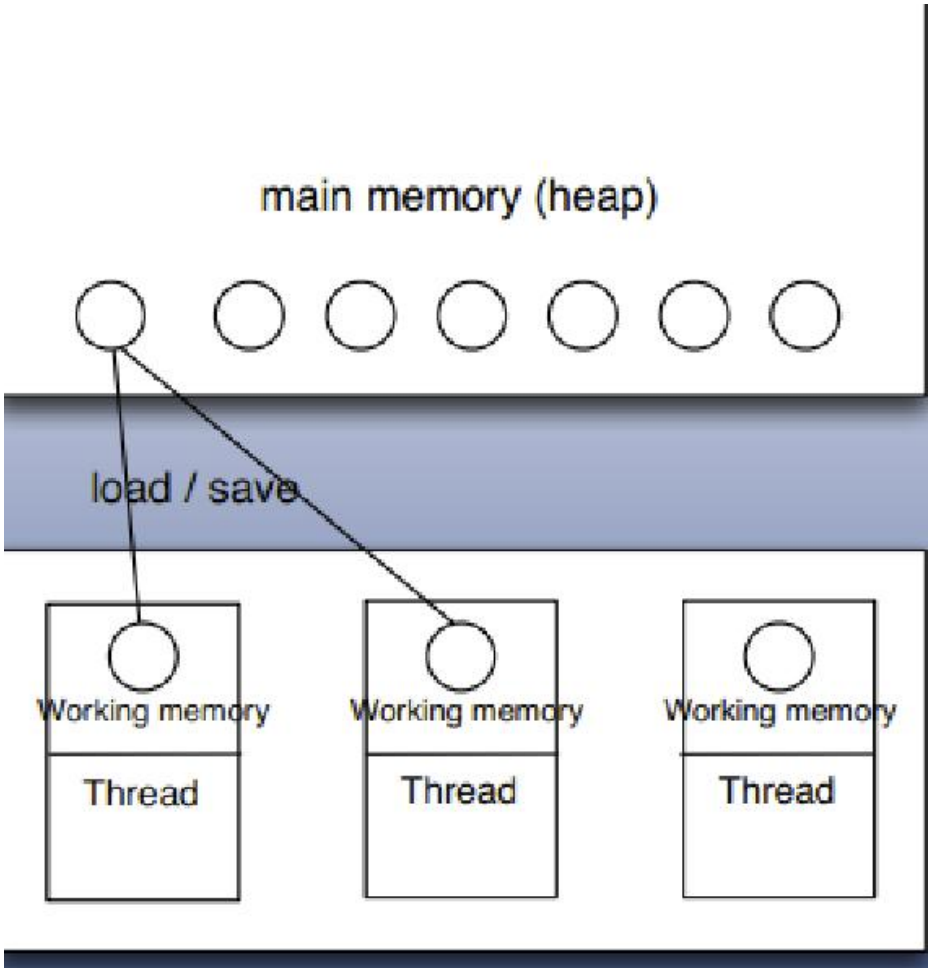
作用于类、方法、成员变量、局部变量。初始化完成后的不可变对象，其它线程可见。常量不会改变不会因为其它线程产生影响。Final 修饰的引用类型的地址不变，同时需要保证引用类型各个成员和操作的线程安全问题。因为引用类型成员可能是可变的。

2.synchronized 同步

作用域代码块、方法上。通过线程互斥，同一时间的同样操作只允许一个线程操作。通过字节码指令实现。

3.Volatile

多线程的内存模型：main memory（主存）、working memory（线程栈），在处理数据时，线程会把值从主存 load 到本地栈，完成操作后再 save 回去(volatile 关键词的作用：每次针对该变量的操作都激发一次 load and save)。



（1）volatile 修饰的变量的变化保证对其它线程立即可见。

volatile 变量的写，先发生于读。每次使用 volatile 修饰的变量个线程都会刷新保证变量一致性。但同步之前各线程可能仍有操作。（如：各个根据 volatile 变量初始值分别进行一些列操作，然后再同步写赋值。每个线程的操作有先后，当一个最早的线程给线程赋值时，其它线程同步。但这时其它线程可能根据初始值做了改变，同步的结果导致其它线程工作结果丢失。）

因此，根据 volatile 的语意使用条件：运算结果不依赖变量的当前值。

（2）volatile 禁止指令重排优化。

这个语意导致写操作会慢一些。因为读操作跟这个没关系。

并发包概述

java.util.concurrent 包含许多线程安全、测试良好、高性能的并发构建块。不客气地说，创建 java.util.concurrent 的目的就是要实现 Collection 框架对数据结构所执行的并发操作。通过提供一组可靠的、高性能并发构建块，开发人员可以提高并发类的线程安全、可伸缩性、性能、可读性和可靠性。

此包包含 locks,concurrent,atomic 三个包。

1.Atomic：原子数据的构建。

2.Locks：基本的锁的实现，最重要的 AQS 框架和 lockSupport

3.Concurrent：构建的一些高级的工具，如线程池，并发队列等。

其中都用到了 CAS（compare-and-swap）操作。

CAS 是一种低级别的、细粒度的技术，它允许多个线程更新一个内存位置，同时能够检测其他线程的冲突并进行恢复。它是许多高性能并发算法的基础。在 JDK 5.0 之前，Java 语言中用于协调线程之间的访问的惟一原语是同步，同步是更重量级和粗粒度的。公开 CAS 可以开发高度可伸缩的并发 Java 类。这些更改主要由 JDK 库类使用，而不是由开发人员使用。

CAS 操作都封装在 java 不公开的类库中，sun.misc.Unsafe。此类包含了对原子操作的封装，具体用本地代码实现。本地的 C 代码直接利用到了硬件上的原子操作。

14. 如何保证线程安全？

https://www.jianshu.com/p/fe7ed5b50933

线程安全的定义：

当多个线程访问一个对象时，如果不用考虑这些线程**在运行时环境下的调度和交替执行，也不需要进行额外的同步，或者在调用方进行任何其他的协调操作，调用这个对象的行为都可以获得正确的结果**，那这个对象是线程安全的。

线程安全类型：

为了更加深入地理解线程安全，我们将 Java 语言中各种操作共享的数据分为以下 5 类： **不可变、绝对线程安全、相对线程安全、线程兼容和线程对立。**

（1）**不可变** ：一定是线程安全的，包括 final 修饰的基本数据类型，以及不可变对象（例如 java.lang.String 类对象），对象的行为不会对其状态产生任何影响。

（2）**绝对线程安全**：绝对线程安全通常需要付出很大的，甚至是不切实际的代价，所以一般难以实现。

（3）**相对线程安全**：通常意义上的线程安全，在调用的时候不需要做额外的保障措施，但是对于一些特定顺序的连续调用，就可能需要在调用端使用额外的同步手段来保证调用的正确性

（4）**线程兼容**：指的是对象本身并不是线程安全的，但是可以通过在调用端正确地使用同步手段来保证对象在并发环境中可以安全地使用

（5）**线程对立**：无论调用端是否采用了同步措施，都无法再多线程环境中并发使用的代码，由于 Java 语言天生具备多线程特性，这种排斥多线程的代码很少出现

线程安全的实现方法：

1.互斥同步

互斥同步是常见的一种并发正确性保障手段。

同步是指在多个线程并发访问共享数据时，保证共享数据在同一

个时刻只被一个（或者是一些，使用信号量的时候）线程使用。

而**互斥**是实现同步的一种手段，临界区、互斥量、信号量都是主要的互斥实现方式。

互斥是方法，同步是目的。

在 Java 中，最基本的互斥同步手段就是 synchronized 关键字，synchronized 关键字经过编译后，会在同步块的前后分别形成 monitorenter 和 monitorexit 这两个字节码指令，这两个字节码都需要一个 reference 类型的参数来指明要锁定和解锁的对象。

（1）如果 Java 程序中 synchronized 明确制定了对象参数，那就是这个对象的 reference；

（2）如果没有明确指定，那就根据 synchronized 修饰的是实例方法还是类方法，去取对应的对象实例或 Class 对象来作为锁对象。

2.非阻塞同步

互斥同步最主要的问题就是进行线程阻塞和唤醒带来的性能问题，因此这种同步也成为阻塞同步。非阻塞同步是先进行操作，如果没有其他线程争用共享数据，那操作就成功；如果数据有争用，产生了冲突，那就采取其他的补偿措施。

3.无同步方案

对于一个方法本来就不涉及共享数据，那就自然无须同步措施来

保证正确性。

（1）**可重入代码：**在代码执行的任何时候中断它，转而去执行另一段代码，控制权返回后，原来的程序不会出现任何错误。

（2）**线程本地存储：**共享数据的代码是否能在同一个线程中执行，如“生产者-消费者”模式。

15. Synchronized 用法
16. synchronize 的原理

https://www.jianshu.com/p/19f861ab749e

https://www.jianshu.com/p/19f861ab749e

Java 中的每个对象都可以作为锁。

普通同步方法，锁是当前实例对象。

静态同步方法，锁是当前类的 class 对象。

同步代码块，锁是括号中的对象。

样例：等待 / 通知机制

```java
import java.util.concurrent.TimeUnit;

/**
 * Created by j_zhan on 2016/7/6.
 */
public class WaitNotify {
    static boolean flag = true;
    static Object lock = new Object();

    public static void main(String[] args) throws InterruptedException {
        Thread A = new Thread(new Wait(), "wait thread");
        A.start();
        TimeUnit.SECONDS.sleep(2);
        Thread B = new Thread(new Notify(), "notify thread");
        B.start();
    }

    static class Wait implements Runnable {
        @Override
        public void run() {
            synchronized (lock) {
                while (flag) {
                    try {
                        System.out.println(Thread.currentThread() + " flag is true");
                        lock.wait();
                    } catch (InterruptedException e) {

                    }
                }
                System.out.println(Thread.currentThread() + " flag is false");
            }
        }
    }

    static class Notify implements Runnable {
        @Override
        public void run() {
            synchronized (lock) {
                flag = false;
                lock.notifyAll();
                try {
                    TimeUnit.SECONDS.sleep(7);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
        }
    }
```

其相关方法定义在 java.lang.Object 上，

线程 A 在获取锁后调用了对象 lock 的 wait 方法进入了等待状

态，

线程 B 调用对象 lock 的 notifyAll()方法，

线程 A 收到通知后从 wait 方法处返回继续执行，线程 B 对共享变量 flag 的修改对线程 A 来说是可见的。

整个运行过程需要注意一下几点：

使用 wait()、notify()和 notifyAll()时需要先对调用对象加锁，调用 wait()方法后会释放锁。

调用 wait()方法之后，线程状态由 RUNNING 变为 WAITING，并将当前线程放置到对象的等待队列中。

notify()或 notifyAll()方法调用后，等待线程不会立刻从 wait()中返回，需要等该线程释放锁之后，才有机会获取锁之后从 wait()返回。

notify()方法将等待队列中的一个等待线程从等待队列中移动到同步队列中；notifyAll()方法则是把等待队列中的所有线程都移动到同步队列中；被移动的线程状态从 WAITING 变为 BLOCKED。

从 wait()方法返回的前提是，该线程获得了调用对象的锁。

实现线程之间的互斥性和可见性

互斥性

```
public class SynchronizedTest {
    private static Object object = new Object();
    public static void main(String[] args) throws Exception{
        synchronized(object) {

        }
    }
    public static synchronized void m() {}
}
```

上述代码中，使用了同步代码块和同步方法，通过使用 javap 工具查看生成的 class 文件信息来分析 synchronized 关键字的实现细节。

```
  public static void main(java.lang.String[]) throws java.lang.Exception;
    descriptor: ([Ljava/lang/String;)V
    flags: ACC_PUBLIC, ACC_STATIC
    Code:
      stack=2, locals=3, args_size=1
         0: getstatic     #2                  // Field object:Ljava/lang/Object

         3: dup
         4: astore_1
         5: monitorenter            //监视器进入，获取锁
         6: aload_1
         7: monitorexit              //监视器退出，释放锁
         8: goto          16
        11: astore_2
        12: aload_1
        13: monitorexit
        14: aload_2
        15: athrow
        16: return

    public static synchronized void m();
    descriptor: ()V
    flags: ACC_PUBLIC, ACC_STATIC, ACC_SYNCHRONIZED
    Code:
      stack=0, locals=0, args_size=0
         0: return
      LineNumberTable:
        line 9: 0
```

从生成的 class 信息中，可以清楚的看到

（1)同步代码块使用了 monitorenter 和 monitorexit 指令实现。

（2)同步方法中依靠方法修饰符上的 ACC_SYNCHRONIZED 实现。

无论哪种实现，**本质上都是对指定对象相关联的 monitor 的获取，这个过程是互斥性的**，也就是说同一时刻只有一个线程能够成功，其它失败的线程会被阻塞，并放入到同步队列中，进入 BLOCKED 状态。

在进一步深入之前，我们先认识下两个概念：**对象**头和 monitor。

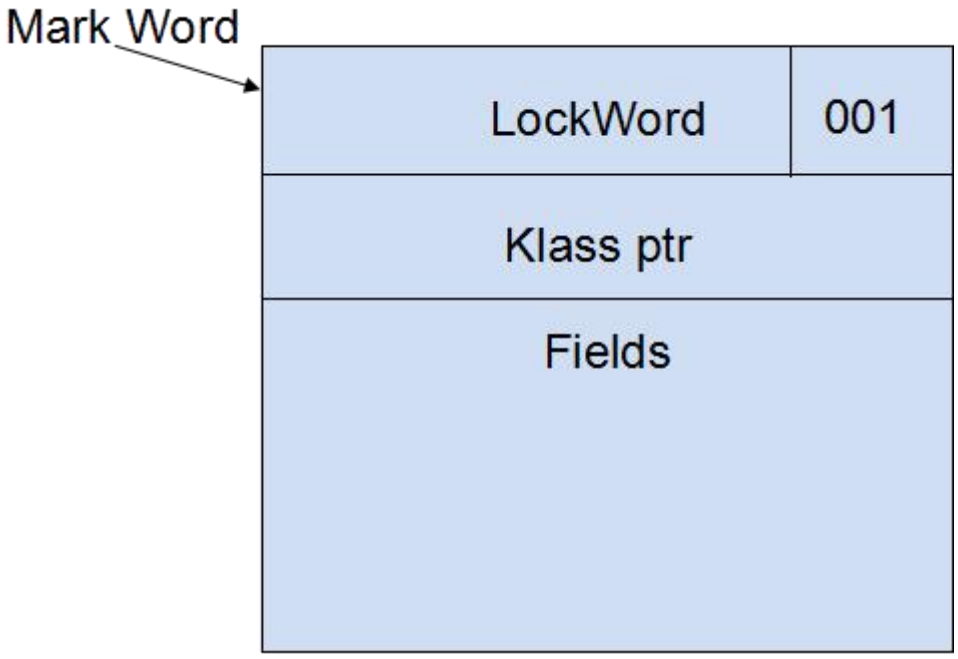
（1）对象头

在 hotspot 虚拟机中，对象在内存的分布分为 3 个部分：对象头，实例数据，和对齐填充。

mark word 被分成两部分，lock word 和标志位。

Klass ptr 指向 Class 字节码在虚拟机内部的对象表示的地址。

Fields 表示连续的对象实例字段。



mark word 被设计为非固定的数据结构，以便在及小的空间内存储更多的信息。比如：在 32 位的 hotspot 虚拟机中：如果对象处于未被锁定的情况下。mark word 的 32bit 空间中有 25bit 存储对象的哈希码、4bit 存储对象的分代年龄、2bit 存储锁的标记位、1bit 固定为 0。而在其他的状态下（轻量级锁、重量级锁、GC 标记、可偏向）下对象的存储结构为

存储内容	标志位	状态
对象哈希码、对象分代年龄	01	未锁定
指向锁记录的指针	00	轻量级锁定
指向重量级锁的指针	10	膨胀（重量级锁定）
空，不需要记录信息	11	GC 标记
偏向线程 ID、偏向时间戳、对象分代年龄	01	可偏向

（2）monitor

monitor 是线程私有的数据结构，每一个线程都有一个可用 monitor 列表，同时还有一个全局的可用列表，先来看 monitor 的内部

1）**Owner**：初始时为 NULL 表示当前没有任何线程拥有该 monitor，当线程成功拥有该锁后保存线程唯一标识，当锁被释放时又设置为 NULL；

2）**EntryQ**：关联一个系统互斥锁（semaphore），阻塞所有试图锁住 monitor 失败的线程。

3）**RcThis**：表示 blocked 或 waiting 在该 monitor 上的所有线程的个数。

4）Nest：用来实现重入锁的计数。

5）HashCode：保存从对象头拷贝过来的 HashCode 值（可能还包含 GC age）。

6）Candidate：用来避免不必要的阻塞或等待线程唤醒。因为每一次只有一个线程能够成功拥有锁，如果每次前一个释放锁的线程唤醒所有正在阻塞或等待的线程，会引起不必要的上下文切换（从阻塞到就绪然后因为竞争锁失败又被阻塞）从而导致性能严重下降。Candidate 只有两种可能的值：0 表示没有需要唤醒的线程，1 表示要唤醒一个继任线程来竞争锁。

monitor 的作用：

在 java 虚拟机中，线程一旦进入到被 synchronized 修饰的方法或代码块时，

指定的锁对象通过某些操作将对象头中的 LockWord 指向 monitor 的起始地址与之关联，

同时 monitor 中的 Owner 存放拥有该锁的线程的唯一标识，确保一次只能有一个线程执行该部分的代码，线程在获取锁之前不允许执行该部分的代码。

我们继续深入了解一下锁的内部机制

一般锁有 4 种状态：**无锁状态，偏向锁状态，轻量级锁状态，重量级锁状态。**

（1）偏向锁

下述代码中，当线程访问同步方法 method1 时，会在对象头（SynchronizedTest.class 对象的对象头）和栈帧的锁记录中存储锁偏向的线程 ID，下次该线程在进入 method2，只需要判断对象头存储的线程 ID 是 否为当前线程，而不需要进行 CAS 操作进行加锁和解锁（因为 CAS 原子指令虽然相对于重量级锁来说开销比较小但还是存在非常可观的本地延迟）。

```
/**
 * Created by j_zhan on 2016/7/6.
 */
public class SynchronizedTest {
    private static Object lock = new Object();
    public static void main(String[] args) {
        method1();
        method2();
    }
    synchronized static void method1() {}
    synchronized static void method2() {}
}
```

（2）轻量级锁

利用了 CPU 原语 Compare-And-Swap(CAS，汇编指令 CMPXCHG)。

线程可以通过两种方式锁住一个对象：

通过膨胀一个处于无锁状态（状态位 001）的对象获得该对象的锁；

对象处于膨胀状态（状态位 00），但 LockWord 指向的 monitor 的 Owner 字段为 NULL，则可以直接通过 CAS 原子指令尝试将 Owner 设置为自己的标识来获得锁。

获取锁（monitorenter）的大概过程：

1. **处于无锁状态，替换 LockWord 进行膨胀（升级）：**对象处于无锁状态时（LockWord 的值为 hashCode 等，状态位为 001），线程首先从 monitor 列表中取得一个空闲的 monitor，初始化 Nest 和 Owner 值为 1 和线程标识，一旦 monitor 准备好，**通过 CAS 替换 monitor 起始地址到 LockWord** 进行膨胀（升级）。如果存在其它线程竞争锁的情况而导致 CAS 失败，则回到 monitorenter（获取锁）重新开始获取锁的过程即可。

2. **对象已膨胀且 monitor 中的 Owner 指向当前进程，重入锁：**对象已经膨胀，monitor 中的 Owner 指向当前线程，这是重入锁的情况（reentrant），将 Nest 加 1，不需要 CAS 操作，效率高。

3. **对象已膨胀，monitor 中的 Owner 为空，竞争锁：**对象已经膨胀，monitor 中的 Owner 为 NULL，此时多个线程通过 CAS 指令试图将 Owner 设置为自己的标识获得锁，竞争失败的线程则进入第 4 种情况。

4. **对象已膨胀且 monitor 中的 Owner 指向其他线程，自旋一定次数没有获取锁，开始进入阻塞状态：**对象已经膨胀，同时 Owner 指向别的线程，在调用操作系统的重量级的互斥锁之前自旋一定的次数，当达到一定的次数如果仍然没有获得锁，则开始准备进入阻塞状态，将 rfThis 值原子加 1，由于在加 1 的过程中可能被其它线程破坏对象和 monitor 之间的联系，所以在加 1 后需要再进行一次比较确保 lock word 的值没有被改变，当发现被改变后则要重新进行 monitorenter 过程。同时再一次观察 Owner 是否为

NULL，如果是则调用 CAS 参与竞争锁，锁竞争失败则进入到阻塞状态。

注：自旋锁

自旋锁是指尝试获取锁的线程不会立即阻塞，而是采用循环的方式去尝试获取锁，这样的好处是减少线程上下文切换的消耗，缺点是循环会消耗 CPU。

释放锁（monitorexit）的大概过程：

1.检查该对象是否处于膨胀状态并且该线程是这个锁的拥有者，如果发现不对则抛出异常。

2.检查 Nest 字段是否大于 1，如果大于 1 则简单的将 Nest 减 1 并继续拥有锁，如果等于 1，则进入到步骤 3。

3.检查 rfThis 是否大于 0，设置 Owner 为 NULL 然后唤醒一个正在阻塞或等待的线程再一次试图获取锁,如果等于 0 则进入到步骤 4。

4.缩小（deflate）一个对象，通过将对象的 LockWord 置换回原来的 HashCode 等值来解除和 monitor 之间的关联来释放锁，同时将 monitor 放回到线程私有的可用 monitor 列表。

```java
/**
 * Created by j_zhan on 2016/7/6.
 */
public class SynchronizedTest implements Runnable {

    private static Object lock = new Object();
    public static void main(String[] args) {
        Thread A = new Thread(new SynchronizedTest(), "A");
        A.start();

        Thread B = new Thread(new SynchronizedTest(), "B");
        B.start();
    }

    @Override
    public void run() {
        method1();
        method2();
    }
    synchronized static void method1() {}
    synchronized static void method2() {}
}
```

（3）重量级锁

当锁处于这个状态下，其他线程试图获取锁都会被阻塞住，当持有锁的线程释放锁之后会唤醒这些线程。

内存可见性

1.线程释放锁时，JMM 会把该线程对应的本地内存中的共享变量刷新到主内存中。

2.线程获取锁时，JMM 会把该线程对应的本地内存置为无效，从而使得被监视器保护的临界区代码必须从主内存中读取共享变量。

17. 谈谈对 Synchronized 关键字,类锁，方法锁，重入锁的理解

https://blog.csdn.net/le_le_name/article/details/52348314

先了解相关的锁知识：

java 的内置锁：每个 java 对象都可以用做一个实现同步的锁，这些锁成为内置锁。线程进入**同步代码块或方法**的时候会自动获得该锁，在退出同步代码块或方法时会释放该锁。获得内置锁的唯一途径就是进入这个锁的保护的同步代码块或方法。

java 内置锁是一个互斥锁，这就是意味着最多只有一个线程能够获得该锁，当线程 A 尝试去获得线程 B 持有的内置锁时，线程 A 必须等待或者阻塞，知道线程 B 释放这个锁，如果 B 线程不释放这个锁，那么 A 线程将永远等待下去。

与之对应的还有显式锁

（1）当 synchronized 作用于普通方法是，锁对象是 this；

（2）当 synchronized 作用于静态方法是，锁对象是当前类的 Class 对象；

（3）当 synchronized 作用于代码块时，锁对象是 synchronized(obj)中的这个 obj。

方法锁:用 synchronized 修饰方法。方法锁 这个就属于对象锁。也可以说方法锁和对象锁说的是一个东西，但不准确，对象锁还有代码块。

java 的对象锁和类锁：java 的对象锁和类锁在锁的概念上基本

上和内置锁是一致的，但是，两个锁实际是有很大的区别的，**对象锁是用于对象实例方法，或者一个对象实例上的，类锁是用于类的静态方法或者一个类的 class 对象上的。**

对象锁：（synchronized 修饰方法或代码块）

```
// 对象锁：形式 1(方法锁)
public synchronized void Method1()
// 对象锁：形式 2（代码块形式）
synchronized (this)
```

类锁(synchronized 修饰静态的方法或代码块)

```
// 类锁：形式 1
public static synchronized void Method1()
// 类锁：形式 2
synchronized (Test.class)
```

我们知道，类的对象实例可以有很多个，但是每个类只有一个 class 对象，所以不同对象实例的对象锁是互不干扰的，但是每个类只有一个类锁。但是有一点必须注意的是，其实类锁只是一个概念上的东西，并不是真实存在的，它只是用来帮助我们理解锁定实例方法和静态方法的区别的

例子：下面分别分析这两种用法在对象锁和类锁上的效果。对象锁的 synchronized 修饰方法和代码块：

```
public class TestSynchronized
{
    public void test1()
    {
        synchronized(this)
        {
            int i = 5;
            while( i-- > 0)
            {
                System.out.println(Thread.currentThread().getName() + " : " + i);
                try
                {
                    Thread.sleep(500);
                }
                catch (InterruptedException ie)
                {
                }
            }
        }
    }

    public synchronized void test2()
    {
        int i = 5;
        while( i-- > 0)
        {
            System.out.println(Thread.currentThread().getName() + " : " + i);
            try
            {
                Thread.sleep(500);
            }
            catch (InterruptedException ie)
            {
            }
        }
    }
```

```
    public static void main(String[] args)
    {
        final TestSynchronized myt2 = new TestSynchronized();
        Thread test1 = new Thread(  new Runnable() {  public void run() {  myt2.test1();  } }, "test1"  );
        Thread test2 = new Thread(  new Runnable() {  public void run() { myt2.test2();   } }, "test2"  );
        test1.start();;
        test2.start();
//        TestRunnable tr=new TestRunnable();
//        Thread test3=new Thread(tr);
//        test3.start();
    }

}
```

第一个方法时用了同步代码块的方式进行同步，传入的对象实例是 this，表明是当前对象，当然，如果需要同步其他对象实例，也不可传入其他对象的实例；

第二个方法是修饰方法的方式进行同步。因为第一个同步代码块传入的 this，所以两个同步代码所需要获得的对象锁都是同一个对象锁。

下面 main 方法时分别开启两个线程，分别调用 test1 和 test2 方法，那么两个线程都需要获得该对象锁，另一个线程必须等待。

上面也给出了运行的结果可以看到：直到 test2 线程执行完毕，释放掉锁，test1 线程才开始执行。（可能这个结果有人会有疑问，代码里面明明是先开启 test1 线程，为什么先执行的是 test2 呢？这是因为 java 编译器在编译成字节码的时候，会对代码进行一个重排序，也就是说，编译器会根据实际情况对代码进行一个合理的排序，编译前代码写在前面，在编译后的字节码不一定排在前面，所以这种运行结果是正常的，　这里是题外话，最主要是检验 synchronized 的用法的正确性）

所以 synchronized 只是一个内置锁的加锁机制，当某个方法加上 synchronized 关键字后，就表明要获得该内置锁才能执行，并不能阻止其他线程访问不需要获得该内置锁的方法。

类锁修饰方法和代码块的效果和对象锁是一样的，因为类锁只是一个抽象出来的概念，只是为了区别静态方法的特点，因为静态方法是所有对象实例共用的，所以对应着 synchronized 修饰的静态方法

的锁也是唯一的，所以抽象出来个类锁。

其实这里的重点在下面这块代码，synchronized 同时修饰静态和非静态方法

Java代码

```
public class TestSynchronized
{
  public synchronized void test1()
  {
        int i = 5;
        while( i-- > 0)
        {
            System.out.println(Thread.currentThread().getName() + " : " + i);
            try
            {
                Thread.sleep(500);
            }
            catch (InterruptedException ie)
            {
            }
        }
  }

  public static synchronized void test2()
  {
      int i = 5;
      while( i-- > 0)
      {
          System.out.println(Thread.currentThread().getName() + " : " + i);
          try
          {
              Thread.sleep(500);
          }
          catch (InterruptedException ie)
          {
          }
      }
  }

  public static void main(String[] args)
  {
      final TestSynchronized myt2 = new TestSynchronized();
      Thread test1 = new Thread(  new Runnable() {  public void run() {  myt2.test1(); } }, "test1"  );
      Thread test2 = new Thread(  new Runnable() {  public void run() { TestSynchronized.test2();  } }, "test
2"  );
      test1.start();
      test2.start();
//      TestRunnable tr=new TestRunnable();
//      Thread test3=new Thread(tr);
//      test3.start();
```

上面代码 synchronized 同时修饰静态方法和实例方法，但是运行结果是交替进行的，这证明了类锁和对象锁是两个不一样的锁，控制着不同的区域，它们是互不干扰的。同样，线程获得对象锁的同时，也可以获得该类锁，即同时获得两个锁，这是允许的。

既然有了 synchronized 修饰方法的同步方式，为什么还需要 synchronized 修饰同步代码块的方式呢？

而这个问题也是 synchronized 的缺陷所在

synchronized 的缺陷：当某个线程进入同步方法获得对象锁，那么其他线程访问这里对象的同步方法时，必须等待或者阻塞，这对高并发的系统是致命的，这很容易导致系统的崩溃。如果某个线程在同步方法里面发生了死循环，那么它就永远不会释放这个对象锁，那么其他线程就要永远的等待。这是一个致命的问题。

这也是同步代码块在某种情况下要优于同步方法的方面。也就是我们可以在调用者上使用 synchronized 来修饰执行者代码块，及时阻塞，也不会影响到执行者的代码（例如在某个类的方法里面：这个类里面声明了一个对象实例，SynObject so=new SynObject()；在某个方法里面调用了这个实例的方法 so.testsy();但是调用这个方法需要进行同步，不能同时有多个线程同时执行调用这个方法。

这时如果直接用 synchronized 修饰调用了 so.testsy();代码的方法，那么当某个线程进入了这个方法之后，这个对象其他同步方法

都不能给其他线程访问了。假如这个方法需要执行的时间很长，那么其他线程会一直阻塞，影响到系统的性能。

如果这时用 synchronized 来修饰代码块：synchronized（so）{so.testsy();}，那么这个方法加锁的对象是 so 这个对象，跟执行这行代码的对象没有关系，当一个线程执行这个方法时，这对其他同步方法时没有影响的，因为他们持有的锁都完全不一样。

）

有些情况下使用同步代码块比 synchronized 修饰的方法更优：

不过这里还有一种特例，就是上面演示的第一个例子，对象锁 synchronized 同时修饰方法和代码块，这时也可以体现到同步代码块的优越性，如果 test1 方法同步代码块后面有非常多没有同步的代码，而且有一个 100000 的循环，这导致 test1 方法会执行时间非常长，那么如果直接用 synchronized 修饰方法，那么在方法没执行完之前，其他线程是不可以访问 test2 方法的，但是如果用了同步代码块，那么当退出代码块时就已经释放了对象锁，当线程还在执行 test1 的那个 100000 的循环时，其他线程就已经可以访问 test2 方法了。这就让阻塞的机会或者线程更少。让系统的性能更优越。

一个类的对象锁和另一个类的对象锁是没有关联的，当一个线程获得 A 类的对象锁时，它同时也可以获得 B 类的对象锁。

同步代码与和同步方法的不同：

1.从尺寸上讲，同步代码块比同步方法小。你可以把同步代码块看成是没上锁房间里的一块用带锁的屏风隔开的空间。

2.同步代码块还可以人为的指定获得某个其它对象的 key。就像是指定用哪一把钥匙才能开这个屏风的锁，你可以用本房的钥匙；你也可以指定用另一个房子的钥匙才能开（这样的话，你也可以要跑到另一栋房子那儿把那个钥匙拿来，并用那个房子的钥匙来打开这个房子的带锁的屏风。）

记住你获得的那另一栋房子的钥匙，并不影响其他人进入那栋房子没有锁的房间。

为什么要使用同步代码块呢？

首先对程序来讲同步的部分很影响运行效率，而一个方法通常是先创建一些局部变量，再对这些变量做一些 操作，如运算，显示等等；而同步所覆盖的代码越多，对效率的影响就越严重。因此我们通常尽量缩小其影响范围。

如何做？同步代码块。

我们只把一个方法中该同步的地方同步，比如运算。

另外，同步代码块可以指定钥匙这一特点有个额外的好处，是可

以在一定时期内霸占某个对象的 key。还记得前面说过普通情况下钥匙的使用原则吗。现在不是普通情况了。你所取得的那把钥匙不是永远不还，而是在退出同步代码块时才还。

还用前面那个想连续用两个上锁房间的家伙打比方。怎样才能在用完一间以后，继续使用另一间呢。用同步代码块吧。先创建另外一个线程，做一个同步代码 块，把那个代码块的锁指向这个房子的钥匙。然后启动那个线程。只要你能在进入那个代码块时抓到这房子的钥匙，你就可以一直保留到退出那个代码块。也就是说 你甚至可以对本房内所有上锁的房间遍历，甚至再 sleep(10*60*1000)，而房门口却还有 1000 个线程在等这把钥匙呢。很过瘾吧。

重入锁：

广义上的可重入锁指的是可重复可递归调用的锁，在外层使用锁之后，在内层仍然可以使用，并且不发生死锁（前提得是同一个对象或者 class），这样的锁就叫做可重入锁。

即线程在执行某个方法时已经持有了这个锁，那么线程在执行另一个方法时也持有该锁。首先我们来看看加锁方法 lock 的实现

18. static synchronized 方法的多线程访问和作用

https://blog.csdn.net/wangtaomtk/article/details/52318634

synchronized 与 static synchronized 的区别：

synchronized 是对类的当前实例进行加锁，防止其他线程同时访问该类的该实例的所有 synchronized 块，注意这里是“**类的当前实例**”，类的两个不同实例就没有这种约束了。

那么 static synchronized 恰好就是要控制类的所有实例的访问了，static synchronized 是限制线程同时访问 jvm 中该类的所有实例同时访问对应的代码快。

实际上，在类中某方法或某代码块中有 synchronized，那么在生成一个该类实例后，该类也就有一个监视快，放置线程并发访问该实例 synchronized 保护快，

而 static synchronized 则是所有该类的实例公用一个监视快了，也就是两个的区别了，也就是 synchronized 相当于 this.synchronized，而 staticsynchronized 相当于 Something.synchronized.

例子：

```
pulbic class Something(){
        public synchronized void isSyncA(){}
        public synchronized voidisSyncB(){}
        public static synchronizedvoid cSyncA(){}
        public static synchronizedvoid cSyncB(){}
}
```

那么，假如有 Something 类的两个实例 a 与 b，那么下列组方法可以被 1 个以上线程同时访问呢

a. x.isSyncA()与 x.isSyncB()

b. x.isSyncA()与 y.isSyncA()

c. x.cSyncA()与 y.cSyncB()

d. x.isSyncA()与 Something.cSyncA()

这里，很清楚的可以判断：

a，都是对同一个实例的 synchronized 域访问，因此不能被同时访问

b，是针对不同实例的，因此可以同时被访问

c，因为是 staticsynchronized，所以不同实例之间仍然会被限制,相当于 Something.isSyncA()与 Something.isSyncB()了，因此不能被同时访问。

d，是可以被同时访问的，理由是 synchronzied 的是实例方法与 synchronzied 的类方法由于锁定（lock）不同的原因。

个人分析也就是 synchronized 与 static synchronized 相当于两帮派，各自管各自，相互之间就无约束了，可以被同时访问。

结论：

1. synchronized static 是某个类的范围，synchronized static cSync{}防止多个线程同时访问这个类中的 synchronized static 方法。它可以对类的所有对象实例起作用。

2. synchronized 是某实例的范围，synchronized isSync(){}防止多个线程同时访问这个实例中的 synchronized 方法。

19. 同一个类里面两个 synchronized 方法，两个线程同时访问的问题

https://blog.csdn.net/aiyawalie/article/details/53261823

```
public class ThreadA extends Thread {
    private MyObject object;

    public ThreadA(MyObject object){
        this.object = object;
    }

    public void run(){
        super.run();
        object.menthodA();
    }
}
```

```
public class ThreadB extends Thread {
    private MyObject object;

    public ThreadB(MyObject object){
        this.object = object;
    }

    public void run(){
        super.run();
        object.methodB();
    }
}
```

```
public class Run {
    public static void main(String args[]){
        MyObject myObject = new MyObject();
        ThreadA threadA = new ThreadA(myObject);
        threadA.setName("A");
        threadA.start();
        ThreadB threadB = new ThreadB(myObject);
        threadB.setName("B");
        threadB.start();
    }
}
```

代码如上所示，MyObject 类有两个方法，分别创建两个线程调用方法 A 和方法 B：

会有以下几种情况：

1、**两个方法都没有 synchronized 修饰，调用时都可进入：**方法

A 和方法 B 都没有加 synchronized 关键字时，调用方法 A 的时候可进入方法 B；

2、**一个方法有 synchronized 修饰，另一个方法没有，调用时都可进入：**方法 A 加 synchronized 关键字而方法 B 没有加时，调用方法 A 的时候可以进入方法 B；

3、**两个方法都加了 synchronized 修饰，一个方法执行完才能执行另一个：**方法 A 和方法 B 都加了 synchronized 关键字时，调用方法 A 之后，必须等 A 执行完成才能进入方法 B；

4、**两个方法都加了 synchronized 修饰，其中一个方法加了 wait（）方法，调用时都可进入：**方法 A 和方法 B 都加了 synchronized 关键字时，且方法 A 加了 wait()方法时，调用方法 A 的时候可以进入方法 B；

5、**一个添加了 synchronized 修饰，一个添加了 static 修饰，调用时都可进入：**方法 A 加了 synchronized 关键字，而方法 B 为 static 静态方法时，调用方法 A 的时候可进入方法 B；

6、**两个方法都是静态方法且还加了 synchronized 修饰，一个方法执行完才能执行另一个：**方法 A 和方法 B 都是 static 静态方法，且都加了 synchronized 关键字，则调用方法 A 之后，需要等 A 执行完成才能进入方法 B；

7、**两个方法都是静态方法且还加了 synchronized 修饰，分别在不同线程调用不同的方法，还是需要一个方法执行完才能执行另一个：**方法 A 和方法 B 都是 static 静态方法，且都加了 synchronized

关键字，创建不同的线程分别调用 A 和 B，需要等 A 执行完成才能执行 B（因为 static 方法是单实例的，A 持有的是 Class 锁，Class 锁可以对类的所有对象实例起作用）

总结：

同一个 object 中多个方法都加了 synchronized 关键字的时候，其中调用任意方法之后需等该方法执行完成才能调用其他方法，即同步的，阻塞的；

此结论同样适用于对于 object 中使用 synchronized(this)同步代码块的场景；

synchronized 锁定的都是当前对象！

20. synchronized 和 volatile 关键字的区别

https://blog.csdn.net/suifeng3051/article/details/52611233

volatile 和 synchronized 特点

首先需要理解线程安全的两个方面：**执行控制和内存可见**。

（1）执行控制的目的是：**控制代码执行（顺序）及是否可以并发执行。**

（2）内存可见控制的是：**线程执行结果在内存中对其它线程的可见性。**（根据 Java 内存模型的实现，线程在具体执行时，会先拷贝主存数据到线程本地（CPU 缓存），操作完成后再把结果从线程本地刷到主存。）

synchronized 关键字解决的是**执行控制的问题**，它会阻止其它线程获取当前对象的监控锁，这样就使得当前对象中被 synchronized 关键字保护的代码块无法被其它线程访问，也就无法并发执行。

更重要的是，synchronized 还会创建一个内存屏障，内存屏障指令保证了所有 CPU 操作结果都会直接刷到主存中，从而**保证了操作的内存可见性**，同时也使得先获得这个锁的线程的所有操作，都 happens-before 于随后获得这个锁的线程的操作。

volatile 关键字解决的是内存可见性的问题，会使得所有对 volatile 变量的读写都会直接刷到主存，即保证了变量的可见性。这样就能满足一些对变量可见性有要求而对读取顺序没有要求的需求。

使用 volatile 关键字仅能实现对原始变量(如 boolen、short 、int 、long 等)操作的原子性，但需要特别注意， volatile 不能保

证复合操作的原子性，即使只是 i++（实际上也是由多个原子操作组成：read i; inc; write i），假如多个线程同时执行 i++，volatile 只能保证他们操作的 i 是同一块内存，但依然可能出现写入脏数据的情况。

在 Java 5 提供了原子数据类型 atomic wrapper classes，对它们的 increase 之类的操作都是原子操作，不需要使用 sychronized 关键字。

对于 volatile 关键字，当且仅当满足以下所有条件时可使用：

1. 对变量的写入操作不依赖变量的当前值，或者你能确保只有单个线程更新变量的值。

2. 该变量没有包含在具有其他变量的不变式中。

volatile 和 synchronized 的区别

1. **本质原理不同：**volatile 本质是在告诉 jvm 当前变量在寄存器(工作内存)中的值是不确定的，需要从主存中读取；synchronized 则是锁定当前变量，只有当前线程可以访问该变量，其他线程被阻塞住。

2. **使用范围不同：**volatile 仅能使用在变量级别；synchronized 则可以使用在变量、方法、和类级别的

3. **实现功能：**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

4.**会造成的线程阻塞情况：**volatile 不会造成线程的阻塞；synchronized 可能会造成线程的阻塞。

5.**是否会被编译优化：**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

21. synchronized 与 Lock 的区别

区别：

1.用法上的不同：

（1）使用范围不同：synchronized 既可以加在方法上，也可以加载特定代码块上，而 lock 需要显示地指定起始位置和终止位置。

（2）实现方式不同：synchronized 是托管给 JVM 执行的，lock 的锁定是通过代码实现的，它有比 synchronized 更精确的线程语义。

2.性能上的不同：

lock 接口的实现类 ReentrantLock，不仅具有和 synchronized 相同的并发性和内存语义，还多了锁投票、定时锁、等候和中断锁等。在竞争不是很激烈的情况下，synchronized 的性能优于 ReentrantLock，

竞争激烈的情况下 synchronized 的性能会下降的非常快，而 ReentrantLock 则基本不变。

3.锁机制不同：

synchronized 获取锁和释放锁的方式都是在块结构中，当获取多个锁时，必须以相反的顺序释放，并且是自动解锁。

而 Lock 则需要开发人员手动释放，并且必须在 finally 中释放，否则会引起死锁。

22. 两个进程同时要求写或者读，能不能实现？如何防止进程的同步？

使用 ConcurrentHashMap，加锁？FileWrite?

23. 线程间操作 List

（暂无）

24. Java 中对象的生命周期

https://blog.csdn.net/sodino/article/details/38387049

Java 对象的生命周期

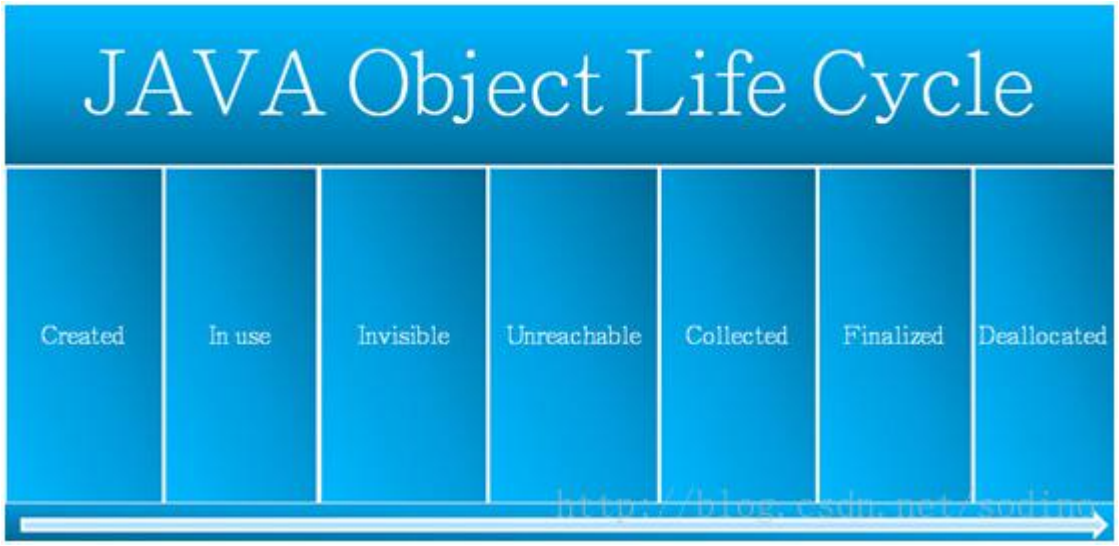
在 Java 中，对象的生命周期包括以下几个阶段：

1. 创建阶段(Created)
2. 应用阶段(In Use)
3. 不可见阶段(Invisible)
4. 不可达阶段(Unreachable)

5.收集阶段(Collected)

6.终结阶段(Finalized)

7.对象空间重分配阶段(De-allocated)



1.创建阶段(Created)

在创建阶段系统通过下面的几个步骤来完成对象的创建过程

（1）为对象分配存储空间

（2）开始构造对象

（3）从超类到子类对 static 成员进行初始化

（4）超类成员变量按顺序初始化，递归调用超类的构造方法

（5）子类成员变量按顺序初始化，子类构造方法调用

一旦对象被创建，并被分派给某些变量赋值，这个对象的状态就切换到了应用阶段

2.应用阶段(In Use)

对象至少被一个强引用持有着。

3.不可见阶段(Invisible)

当一个对象处于不可见阶段时，说明程序本身不再持有该对象的任何强引用，虽然该这些引用仍然是存在着的。

简单说就是程序的执行已经超出了该对象的作用域了。

4.不可达阶段(Unreachable)

对象处于不可达阶段是指该对象不再被任何强引用所持有。

注：与“不可见阶段”相比，“不可见阶段”是指程序不再持有该对象的任何强引用，这种情况下，该对象仍可能被 JVM 等系统下的某些已装载的静态变量或线程或 JNI 等强引用持有着，这些特殊的强引用被称为”GC root”。存在着这些 GC root 会导致对象的内存泄露情况，无法被回收。

5.收集阶段(Collected)

当垃圾回收器发现该对象已经处于“不可达阶段”并且垃圾回收器已经对该对象的内存空间重新分配做好准备时，则对象进入了“收集阶段”。如果该对象已经重写了 finalize()方法，则会去执行该方法的终端操作。

注：不要重载 finazlie()方法！原因有两点：

（1）会影响 JVM 的对象分配与回收速度

在分配该对象时，JVM **需要在垃圾回收器上注册该对象**，以便在回收时能够执行该重载方法；在该方法的执行时**需要消耗 CPU 时间且在执行完该方法后才会重新执行回收操作**，即至少需要垃圾回收器对该对象执行两次 GC。

（2）可能造成该对象的再次“复活”

在 finalize()方法中，如果有其它的强引用再次持有该对象，则会导致对象的状态由“收集阶段”又重新变为“应用阶段”。这个已经破坏了 Java 对象的生命周期进程，且“复活”的对象不利用后续的代码管理。

6.终结阶段

当对象执行完 finalize()方法后仍然处于不可达状态时，则该对象进入终结阶段。在该阶段是等待垃圾回收器对该对象空间进行回收。

7.对象空间重新分配阶段

垃圾回收器对该对象的所占用的内存空间进行回收或者再分配了，则该对象彻底消失了，称之为“对象空间重新分配阶段”。

25. volatile 的原理

https://www.jianshu.com/p/7c614ac4dd92

Volatile 工作原理分析

1.主要作用是保证变量在多线程之间的可见性；

2.volatile 在 concurrent 包中起着举足轻重的作用，为大量的并发类提供了有力的援助；

接下来我们从了解 CPU 缓存开始，然后再深入原理剖析，循序渐进的了解 volatile；

基础知识：

CPU 缓存

1.传输链路

CPU（线程） --》 CPU 缓存（一级、二级、三级缓存等） --》 主内存

大致的传输方向就这样，而且还是必须是双向传输。

2.CPU 缓存

(1)**适应 CPU 与内存之间的速度：**CPU 缓存解决了 CPU 运算速度与内存读取速度不匹配的问题；

(2)**提升数据之间的传输速度：**因为主内存访问通常比较慢，访问时间大概在几十到几百个时钟，而 CPU 缓存还有一二三级之分，每个级别的读取速度虽然很快，但还是有访问速度的区分，至少比主内存的读取速度快很多，所以 CPU 缓存它的出现在很大程度上提高了数据之间的传输；

(3)**数据呈金字塔结构，下面的数据有上面的数据：**每一级缓存中所存储的数据全部都是下一级缓存中的一部分，这三种缓存的技术难度和制造成本是相对递减的，所以其容量也相对递增；

(4)**利用等级缓存能提高查找效率：**当 CPU 要读取一个数据时，首先从一级缓存中查找，如果没有再从二级缓存中查找，如果还是没有再从三级缓存中或内存中查找。

一般来说每级缓存的命中率大概都有 80%左右，也就是说全部数据量的 80%都可以在一级缓存中找到；

Volatile 原理特性

1.可见性

(1)不论线程是如何如何的访问带 volatile 字段的对象，都会访问到内存中最新的一份值；

(2)当我们在 java 代码中书写的那行对 volatile 对象进行写操作时，JVM 会向处理器发送一条 Lock 指令，Lock 指令锁住（锁总线）确保变量对象所在缓存行数据会更新到主内存中去，确保更新后如果再有其他线程访问该对象，其他线程一律强制从主内存中重新读取最新的值。

（3）因为所有内存的传输都发生在一条共享的总线上，并且所有的处理器都能看到这条总线，那么既然所有处理器都能看到这条总线， 总不至于看见了不干点啥吧？

没错，每个处理器都会通过一种嗅探技术，不停的嗅探总线上传

输的数据，以便来检查自己缓存中的数据是否过期。

当处理器发现高速缓存中的数据对应的内存地址被修改，会将该缓存数据置为失效，当处理器下次访问该内存地址数据时，将强制重新从系统内存中读取。

而且 CPU 制造商也曾制定了一个这样的规则：当一个 CPU 修改缓存中的字节对象时，服务器中其他 CPU 会被通知，它们的缓存将视为无效。

当那些被视为无效变量所在的线程再次访问字节对象时，则强制再次从主内存中获取最新值。

（4）至于第 2 点提到 Lock 锁总线，其实最初采用锁总线，虽说能解决问题，但是效率低下，一旦锁总线，其他 CPU 就得干等着，光看不干效率不行嘛。

所以后来优化成了锁缓存，效率也高了，开销也自然就少了，总之 Lock 目的很明确，确保锁住的那份值最新，且其他持有该缓存的备份处都得失效，其实这种锁缓存过程的思想也正是缓存一致性协议的核心思想。

综上所述，所以不论何时不论何地在哪种多线程环境下，只要你想获取被 volatile 修饰过的字段，都能看到最新的一份值，这就是可见性的终极描述。

2.有序性

例子：浅显的讲，A1，A2，A3 三块代码先后执行，A2 有一行代码被 volatile 修饰过，那么在被反编译成指令进行重排序时，A2 必须等到A1 执行完了才能开始,但是A1 内部的指令可以支持重排指令；而 A3 代码块的执行必须等到 A2 执行完了才能开始，但是 A3 内部的指令可以支持重排指令，这就是有序性，只要 A2 夹在中间，A2 必须等 A1 执行完才能干活，A2 没干完活，A3 是不允许开工的。

具体的讲，Lock 前缀指令实际上相当于一个内存屏障（也成内存栅栏），它确保指令重排序时不会把其后面的指令排到内存屏障之前的位置，也不会把前面的指令排到内存屏障的后面；即在执行到内存屏障这句指令时，在它前面的操作已经全部完成。

综上所述，有序不是我们通常说的自然顺序，而是在有 volatile 修饰时，存在类似尊卑等级的先后有序这么一说。

3.非原子性

本不该拿到台面上讲是不是属于 volatile 的特性，因为我们不能认为仅仅只是因为可见性随处都是最新值，那么就认为是原子性操作。

因为可见性只是将 volatile 变量这回主内存并使得其他 CPU 缓存失效，但是不带代表对 volatile 变量

回写主内存的动作和对 volatile 变量的逻辑操作是捆绑在一起

的。因此既要逻辑操作，又要写回主内存，这本来就违背了 volatile 特性的本意，所以 volatile 并不是原子操作的。

26. 谈谈 volatile 关键字的用法
27. 谈谈 volatile 关键字的作用

https://www.jianshu.com/p/7798161d7472

https://www.jianshu.com/p/7798161d7472

volatile 关键字经常在并发编程中使用，其特性是保证可见性以及有序性

1.volatile 保证可见性

一旦一个共享变量（类的成员变量、类的静态成员变量）被 volatile 修饰之后，那么就具备了两层语义：

1）保证了**不同线程对这个变量进行操作时的可见性**，即一个线程修改了某个变量的值，这新值对其他线程来说是立即可见的。

2）**禁止进行指令重排序**。

例子：

```
//线程 1boolean stop = false;while(!stop){

    doSomething();

}

 //线程 2
```

```
stop = true;
```

没使用 vloatile 之前：

每个线程在运行过程中都有自己的工作内存，那么线程 1 在运行的时候，会将 stop 变量的值拷贝一份放在自己的工作内存当中。

那么当线程 2 更改了 stop 变量的值之后，但是还没来得及写入主存当中，线程 2 转去做其他事情了，那么线程 1 由于不知道线程 2 对 stop 变量的更改，因此还会一直循环下去。

使用 volatile 修饰之后就变得不一样了：

第一：使用 volatile 关键字会**强制将修改的值立即写入主存**；

第二：使用 volatile 关键字的话，当线程 2 进行修改时，**会导致线程 1 的工作内存中缓存变量 stop 的缓存行无效**（反映到硬件层的话，就是 CPU 的 L1 或者 L2 缓存中对应的缓存行无效）；

第三：由于线程 1 的工作内存中缓存变量 stop 的缓存行无效，所以线程 1 再次读取变量 stop 的值时会去主存读取。

那么在线程 2 修改 stop 值时（当然这里包括 2 个操作，修改线程 2 工作内存中的值，然后将修改后的值写入内存），会使得线程 1 的工作内存中缓存变量 stop 的缓存行无效，然后线程 1 读取时，发现自己的缓存行无效，它会等待缓存行对应的主存地址被更新之后，然后去对应的主存读取最新的值。

那么线程 1 读取到的就是最新的正确的值。

2.Volatile 无法保证原子性

volatile 关键字能保证可见性没有错，但是上面的程序错在没能保证原子性。可见性只能保证每次读取的是最新的值，但是 volatile 没办法保证对变量的操作的原子性。

在前面已经提到过，**自增操作是不具备原子性的，它包括读取变量的原始值、进行加 1 操作、写入工作内存**。那么就是说自增操作的三个子操作可能会分割开执行，就有可能导致下面这种情况出现：

假如某个时刻变量 inc 的值为 10，

线程 1 对变量进行自增操作，线程 1 先读取了变量 inc 的原始值，然后线程 1 被阻塞了；

然后线程 2 对变量进行自增操作，线程 2 也去读取变量 inc 的原始值，**由于线程 1 只是对变量 inc 进行读取操作，而没有对变量进行修改操作，所以不会导致线程 2 的工作内存中缓存变量 inc 的缓存行无效，也不会导致主存中的值刷新**，所以线程 2 会直接去主存读取 inc 的值，发现 inc 的值时 10，然后进行加 1 操作，并把 11 写入工作内存，最后写入主存。

然后线程 1 接着进行加 1 操作，由于已经读取了 inc 的值，注意此时在线程 1 的工作内存中 inc 的值仍然为 10，所以线程 1 对 inc 进行加 1 操作后 inc 的值为 11，然后将 11 写入工作内存，最后写入主存。

那么两个线程分别进行了一次自增操作后，inc 只增加了 1。

根源就在这里，自增操作不是原子性操作，而且 volatile 也无法保

证对变量的任何操作都是原子性的。

解决方案：

可以通过 synchronized 或 lock，进行加锁，来保证操作的原子性。也可以通过 AtomicInteger。

3. volatile 保证有序性

在前面提到 volatile 关键字能禁止指令重排序，所以 volatile 能在一定程度上保证有序性。

volatile 关键字禁止指令重排序有两层意思：

1）**前面的操作已经全部进行了：**当程序执行到 volatile 变量的读操作或者写操作时，**在其前面的操作的更改肯定全部已经进行，且结果已经对后面的操作可见；在其后面的操作肯定还没有进行；**

2）**volatile 变量的读写操作不能放前也不能放后：**在进行指令优化时，**不能将在对 volatile 变量的读操作或者写操作的语句放在其后面执行，也不能把 volatile 变量后面的语句放到其前面执行。**

那么我们回到前面举的一个例子：

```
//线程 1:
context = loadContext();   //语句 1
inited = true;             //语句 2

 //线程 2:while(!inited ){
  sleep()
}
doSomethingwithconfig(context);
```

前面举这个例子的时候，提到有可能语句 2 会在语句 1 之前执行（因为变量在编译时会根据优化自动排序来编译，不能确保顺序），

那么久可能导致 context 还没被初始化，而线程 2 中就使用未初始化的 context 去进行操作，导致程序出错。

这里如果用 volatile 关键字对 inited 变量进行修饰，就不会出现这种问题了，**因为当执行到语句 2 时，必定能保证 context 已经初始化完毕。**

volatile 的实现原理

1.可见性

处理器为了提高处理速度，不直接和内存进行通讯，而是将系统内存的数据独到内部缓存后再进行操作，但操作完后不知什么时候会写到内存。

（1）**锁住缓存内存，让数据直接写到系统内存：如果对声明了 volatile 变量进行写操作时，JVM 会向处理器发送一条 Lock 前缀的指令，将这个变量所在缓存行的数据写会到系统内存。** 这一步确保了如果有其他线程对声明了 volatile 变量进行修改，则立即更新主内存中数据。

（2）**一旦发现总线上传播的数据是变量，会检查是否过期更新：**但这时候其他处理器的缓存还是旧的，所以在多处理器环境下，为了保证各个处理器缓存一致，每个处理会通过嗅探在总线上传播的数据来检查自己的缓存是否过期，当处理器发现自己缓存行对应的内存地址被修改了，就会将当前处理器的缓存行设置成无效状态，**当处理器要对这个数据进行修改操作时**，会强制重新从系统内存把数据读到处理器缓存里。这一步确保了其他线程获得的声明了 volatile 变量都

是从主内存中获取最新的。

2.有序性

Lock 前缀指令实际上相当于一个内存屏障（也成内存栅栏），它确保**指令重排序时不会把其后面的指令排到内存屏障之前的位置，也不会把前面的指令排到内存屏障的后面**；即在执行到内存屏障这句指令时，在它前面的操作已经全部完成。

volatile 的应用场景

synchronized 关键字是防止多个线程同时执行一段代码，那么就会很影响程序执行效率，而 volatile 关键字在某些情况下性能要优于 synchronized，

但是要注意 volatile 关键字是无法替代 synchronized 关键字的，因为 volatile 关键字无法保证操作的原子性。

通常来说，使用 volatile 必须具备以下 2 个条件：

1）对变量的写操作不依赖于当前值

2）该变量没有包含在具有其他变量的不变式中

下面列举几个 Java 中使用 volatile 的几个场景。

①.状态标记量

```
volatile boolean flag = false;
 //线程1
while(!flag){
    doSomething();
}
  //线程2
public void setFlag() {
    flag = true;
}
```

根据状态标记，终止线程。

②.单例模式中的double check

```
class Singleton{
    private volatile static Singleton instance = null;

    private Singleton() {

    }

    public static Singleton getInstance() {
        if(instance==null) {
            synchronized (Singleton.class) {
                if(instance==null)
                    instance = new Singleton();
            }
        }
        return instance;
    }
}
```

为什么要使用 volatile 修饰 instance?

主要在于 instance = new Singleton()这句，这并非是一个原子操作，事实上在 JVM 中这句话大概做了下面 3 件事情:

1.给 instance 分配内存

2.调用 Singleton 的构造函数来初始化成员变量

3.将 instance 对象指向分配的内存空间（执行完这步 instance 就为非 null 了）。

但是在 JVM 的即时编译器中存在指令重排序的优化。也就是说上面的第二步和第三步的顺序是不能保证的，最终的执行顺序可能是 1-2-3 也可能是 1-3-2。

如果是后者，则在 3 执行完毕、2 未执行之前，被线程二抢占了，这时 instance 已经是非 null 了（但却没有初始化），所以线程二会直接返回 instance，然后使用，然后顺理成章地报错。（即已经指向分配的内存空间了，但还没初始化构造函数，此时调用会报错）

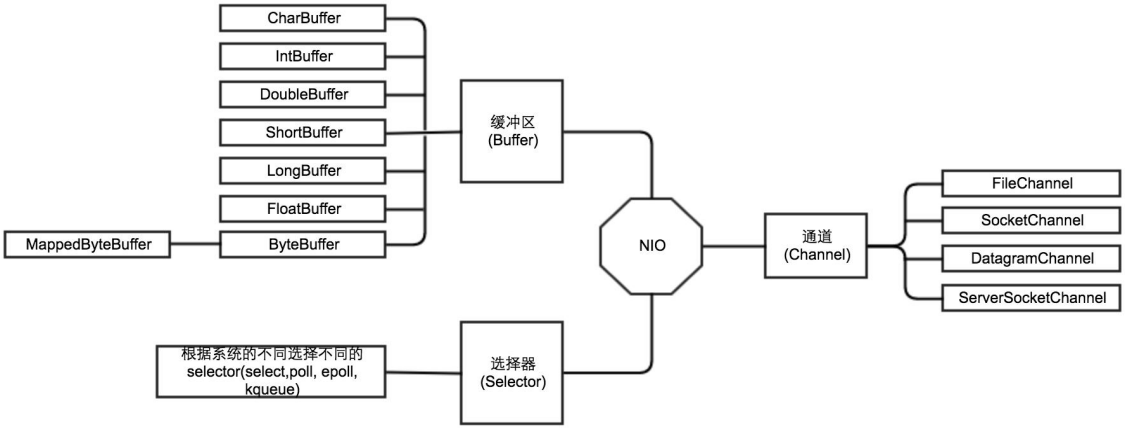
28. 谈谈 NIO 的理解

https://www.jianshu.com/p/a33f741fe450

NIO(Non-Blocking IO)的 IO 处理机制与以往的标准 IO 机制(BIO,Blocking IO)不同的是，新的机制把重点放在了如何缩短抽象与现实之间的距离上面。

NIO 中提出了一种新的抽象，NIO 弥补了原来的 BIO 的不足，它在标准 Java 代码中提供了高速的、面向块的 I/O。

NIO 的包括三个核心概念:缓冲区(Buffer)、通道(Channel)、选择器(Selector)。思维导图如下：



BIO 与 NIO

BIO 与 NIO 之间的共同点是他们都是同步的。而非异步的。

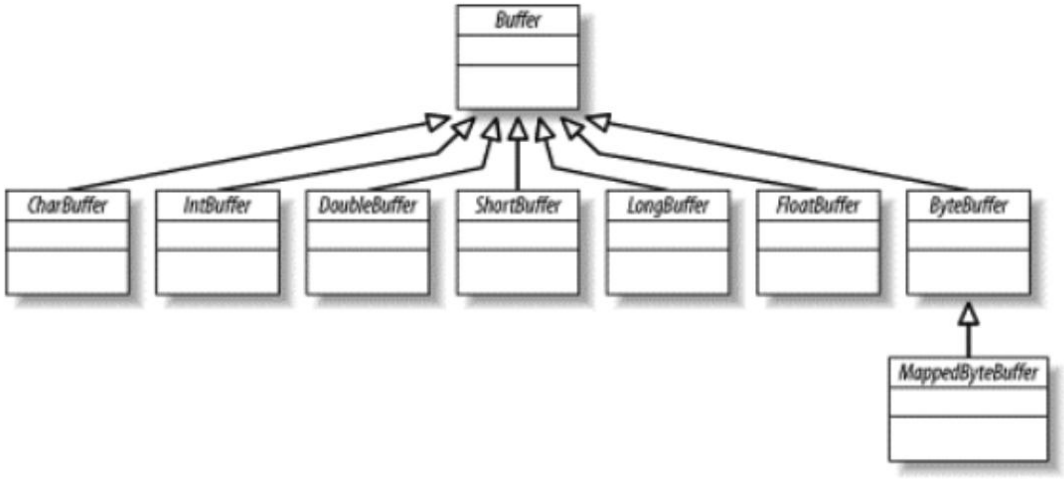
（1）BIO 是阻塞的(当前线程必须等待感兴趣的事情发生)，NIO 是非阻塞的(事件选择，感兴趣的事情发生可以通知线程，而不必一直在哪等待)；

（2）BIO 是面向流式的 IO 抽象(一次一个字节地处理数据)，NIO 是面向块的 IO 抽象(每一个操作都在一步中产生或者消费一个数据块(Buffer))；

（3）BIO 的服务器实现模式为一个连接一个线程，NIO 服务器实现模式为一个请求一个线程；

缓冲区

一个 Buffer 对象是固定数量的数据的容器。其作用是**一个存储器，或者分段运输区**，在这里数据可被存储并在之后用于检索。缓冲区的工作与通道紧密联系。 Buffer 的类层次图:



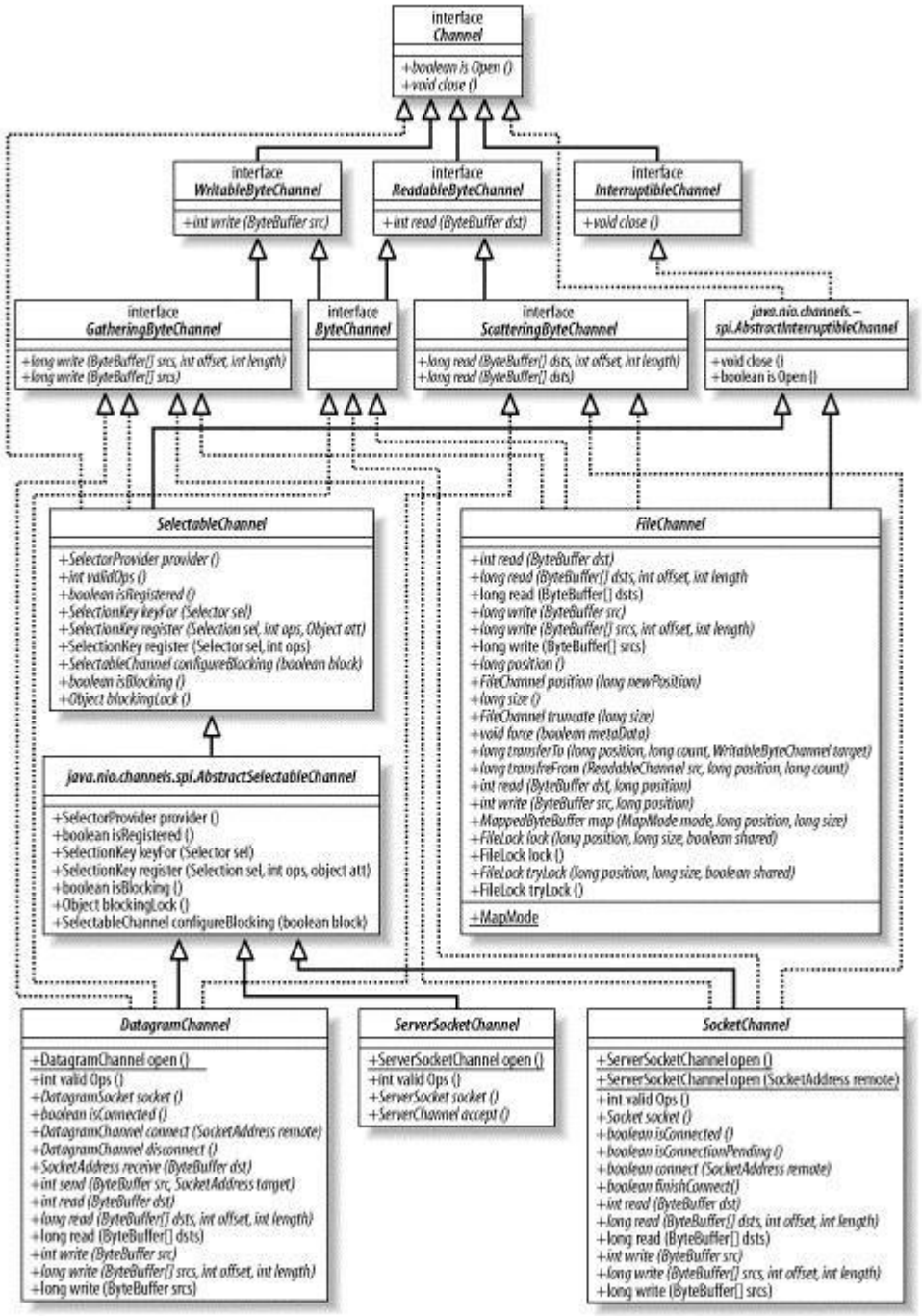
通道

通道用于在字节缓冲区和位于通道另一边的实体(通常是一个文件或套接字)之间有效地传输数据。

例子：

通道可以形象地比喻为银行出纳窗口使用的动导管。您的薪水支票就是您要传送的信息，载体(Carrier)就好比一个缓冲区。您先填充缓冲区(将您的薪水支票放到载体上)，接着将缓冲“写”到通道中(将载体进导管中)，然后信息负载就被传递到通道另一边的 I/O 服务(银行出纳员)。

channel 类的继承关系如下:

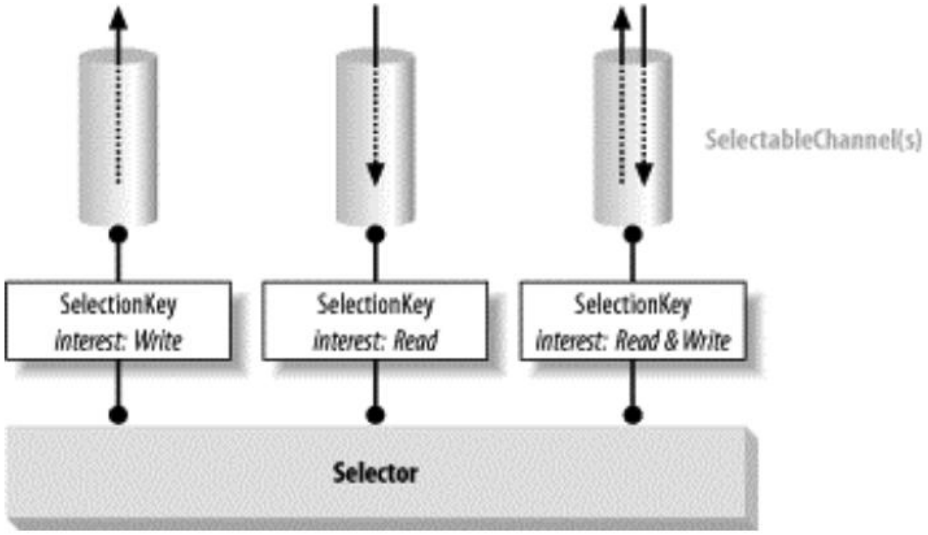


选择器

选择器提供选择执行已经就绪的任务的能力，这使得多元 I/O 成为可能。

选择器类管理着一个被注册的通道集合的信息和它们的就绪状

态。通道是和选择器一起被注册的，并且使用选择器来更新通道的就绪状态。当这么做的时候，可以选择将被激发的线程挂起，直到有就绪的的通道。



29. ReentrantLock 、synchronized 和 volatile 比较

http://heaven-arch.iteye.com/blog/1738212

Lock

作用：显式加锁

原理

(1)通过同步器 AQS（AbstractQueuedSynchronized 类）来实现的，AQS 根本上是通过一个双向队列来实现的

(2)线程构造成一个节点，一个线程先尝试获得锁，如果获取锁

失败，就将该线程加到队列尾部

(3)非公平锁的lock方法,调用的sync(NonfairSync和fairSync的父类）的lock方法

```
public ReentrantLock(boolean fair) {
        sync = fair ? new FairSync() : new NonfairSync();
    }
// ReentrantLock的lock方法
public void lock() {
        sync.lock();
    }
```

ReentrantLock

可重入锁是Lock接口的一个重要实现类。所谓可重入锁即线程在执行某个方法时已经持有了这个锁，那么线程在执行另一个方法时也持有该锁。首先我们来看看加锁方法lock的实现

```
public void lock() {
    sync.lock();
}
```

sync是ReentrantLock中静态内部接口Sync的实例对象。在ReentrantLock中提供了两种Sync的具体实现，FairSync与NonfairSync。故名思意，两种不同的Sync分别用于公平锁和非公平锁。

Synchronized

原理

(1)synchronized 关键字是通过字节码指令来实现的

(2)synchronized 关键字编译后会在同步块前后形成 monitorenter 和 monitorexit 两个字节码指令

(3)执行 monitorenter 指令时需要先获得对象的锁(每个对象有一个监视器锁 monitor)，如果这个对象没被锁或者当前线程已经获得此锁（也就是重入锁），那么锁的计数器+1。如果获取失败，那么当前线程阻塞，直到锁被对另一个线程释放

(4)执行 monitorexit 指令时，计数器减一，当为 0 的时候锁释放

volatile

作用:保证变量对所有的线程的可见性，当一个线程修改了这个变量的值,其他线程可以立即知道这个新值(之所以有可见性的问题，是因为 java 的内存模型)

原理:

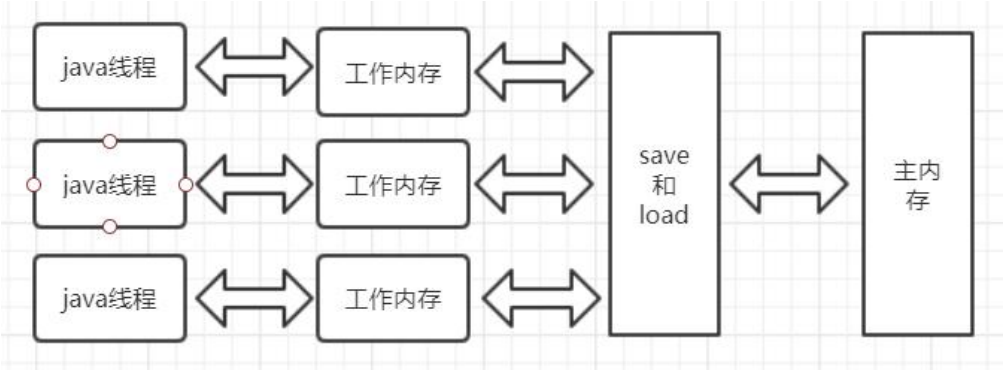
(1)所有变量都存在主内存，每条线程有自己的工作内存，工作内存保存了被该线程使用的变量的主内存副本拷贝

(2)线程对变量的所有操作都必须在工作内存中进行，不能直接读写主内存的变量，也就是必须先通过工作内存

(3)一个线程不能访问另一个线程的工作内存

(4)volatile 保证了变量更新的时候能够立即同步到主内存，使用变量的时候能立即从主内存刷新到工作内存，这样就保证了变量的可见性

(5)实际上是通过内存屏障来实现的。语义上，内存屏障之前的所有写操作都要写入内存；内存屏障之后的读操作都可以获得同步屏障之前的写操作的结果。



30. ReentrantLock 的内部实现

https://blog.csdn.net/yanyan19880509/article/details/52345422

java 可重入锁-ReentrantLock 实现细节

ReentrantLock 支持两种获取锁的方式，一种是**公平模型**，一种是**非公平模型**。在继续之前，咱们先把故事元素转换为程序元素。

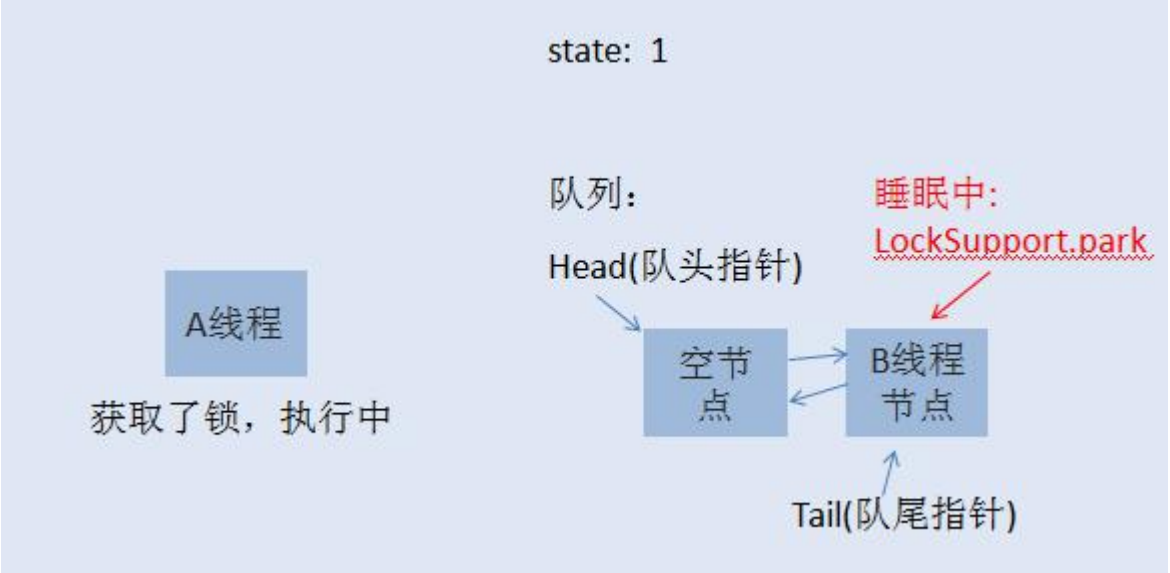
公平锁模型：

（占用锁时，状态 state +1 ，其他想成想要使用，要排队，当在使用的时候再次进入，状态 state 再加 1。释放一次锁就状态 state -1 ，完全释放时状态 state 再 -1，然后让排队的线程竞争锁 ）

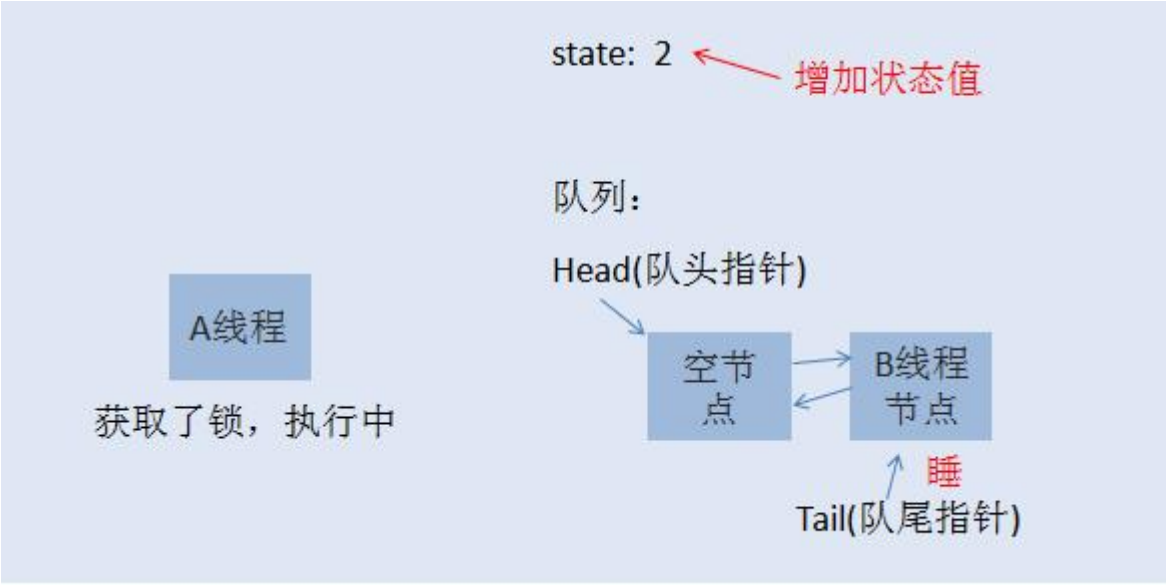
初始化时， state=0，表示无人抢占了打水权。这时候，村民 A 来打水(A 线程请求锁)，占了打水权，把 state+1，如下所示：

线程 A 取得了锁，把 state 原子性+1,这时候 state 被改为 1，A 线程继续执行其他任务，然后来了村民 B 也想打水（线程 B 请求锁），

线程 B 无法获取锁，生成节点进行排队，如下图所示：

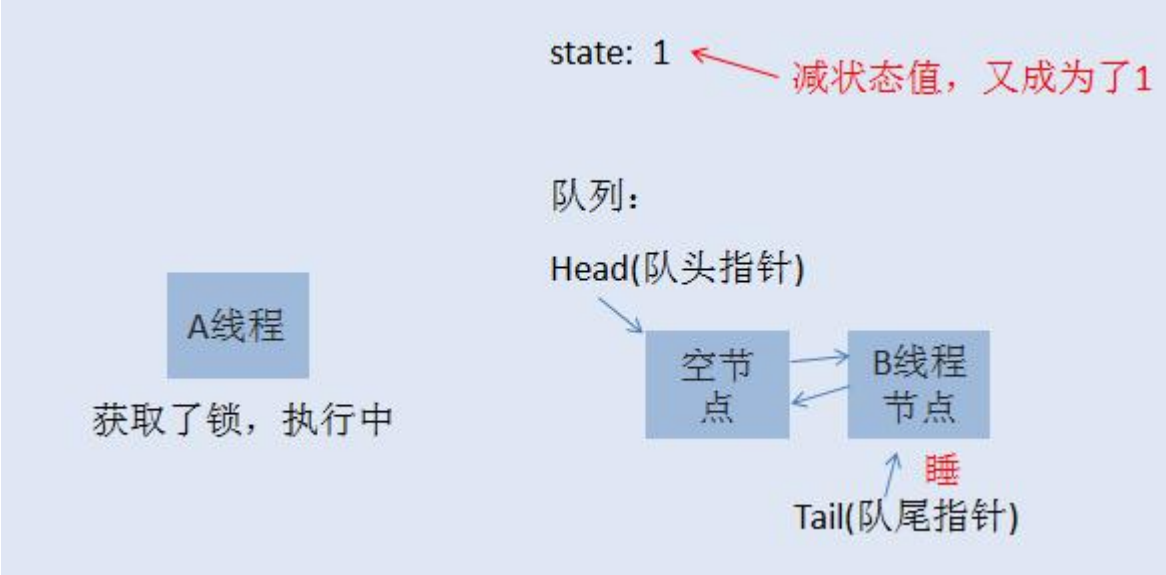


初始化的时候，会生成一个空的头节点，然后才是 B 线程节点，这时候，如果线程 A 又请求锁，是否需要排队？答案当然是否定的，否则就直接死锁了。当 A 再次请求锁，就相当于是打水期间，同一家人也来打水了，是有特权的，这时候的状态如下图所示：



到了这里，相信大家应该明白了什么是可重入锁了吧。就是一个线程在获取了锁之后，再次去获取了同一个锁，这时候**仅仅是把状态**

值进行累加。如果线程 A 释放了一次锁，就成这样了：



仅仅是把状态值减了，只有线程 A 把此锁全部释放了，状态值减到 0 了，其他线程才有机会获取锁。

当 A 把锁完全释放后，state 恢复为 0，然后会通知队列唤醒 B 线程节点，使 B 可以再次竞争锁。当然，如果 B 线程后面还有 C 线程，C 线程继续休眠，除非 B 执行完了，通知了 C 线程。注意，当一个线程节点被唤醒然后取得了锁，对应节点会从队列中删除。

非公平锁模型

（在释放锁之后的切换过程中，本该让排队线程获取锁，却被其他线程抢到锁）

如果你已经明白了前面讲的公平锁模型，那么非公平锁模型也就非常容易理解了。当线程 A 执行完之后，要唤醒线程 B 是需要时间的，而且线程 B 醒来后还要再次竞争锁，所以如果在切换过程当中，来了

一个线程 C，那么线程 C 是有可能获取到锁的，如果 C 获取到了锁，B 就只能继续乖乖休眠了。这里就不再画图说明了。

31. lock 原理

https://blog.csdn.net/endlu/article/details/51249156

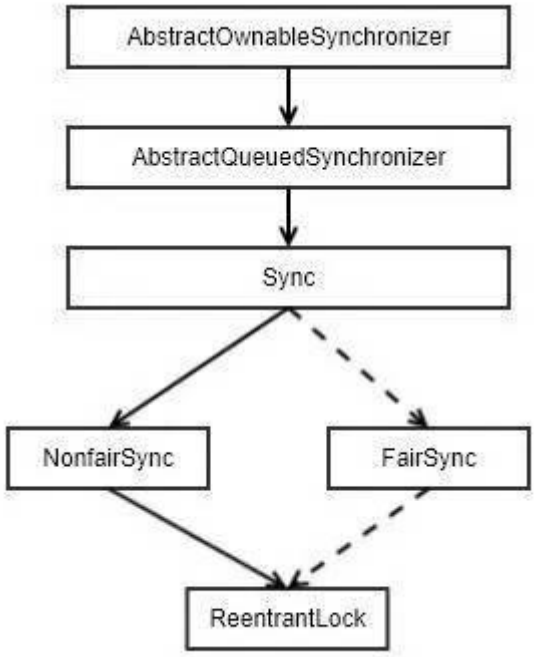
ReentrantLock 的调用过程

经过观察 ReentrantLock 把所有 Lock 接口的操作都委派到一个 Sync 类上，该类继承了 AbstractQueuedSynchronizer：

```
static abstract class Sync extends AbstractQueuedSynchronizer
```

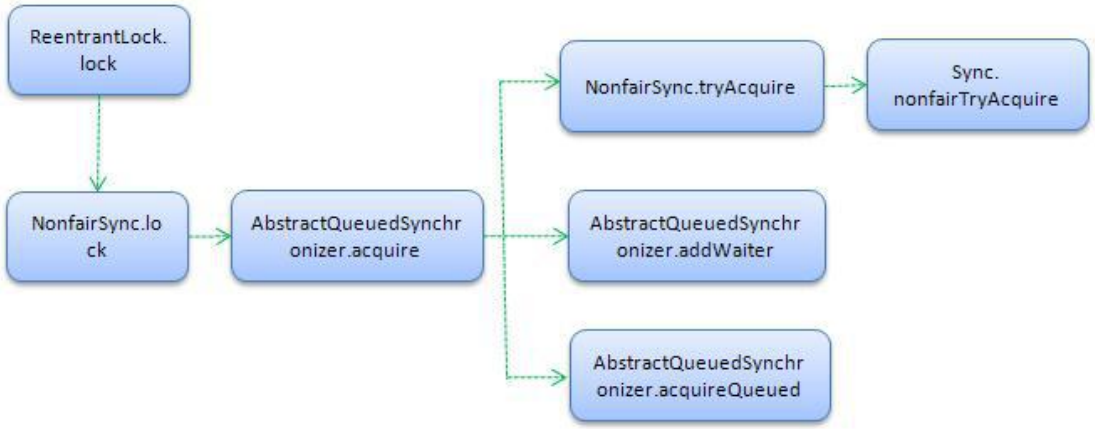
Sync 又有两个子类：

```
final static class NonfairSync extends Sync
final static class FairSync extends Sync
```



显然是为了支持公平锁和非公平锁而定义，默认情况下为非公平锁。

先理一下 Reentrant.lock()方法的调用过程（默认非公平锁）：



这些讨厌的 Template（模板）模式导致很难直观的看到整个调用过程，

其实通过上面调用过程及 AbstractQueuedSynchronizer 的注释可以发现，

AbstractQueuedSynchronizer 中抽象了绝大多数 Lock 的功能，而只把 tryAcquire 方法延迟到子类中实现。

tryAcquire 方法的语义在于用**具体子类判断请求线程是否可以获得锁**，无论成功与否 AbstractQueuedSynchronizer 都将处理后面的流程。

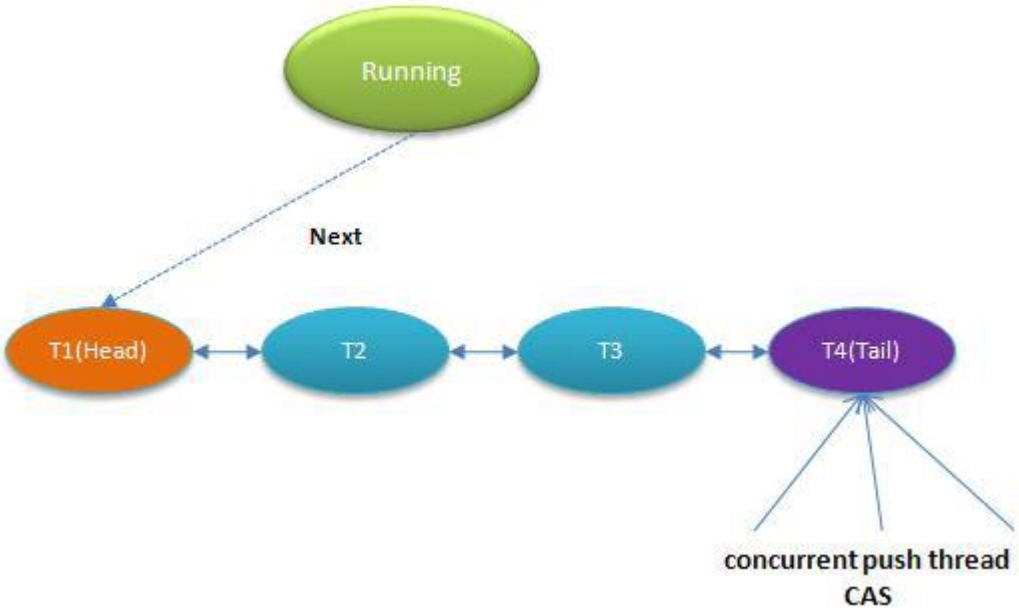
锁实现（加锁）

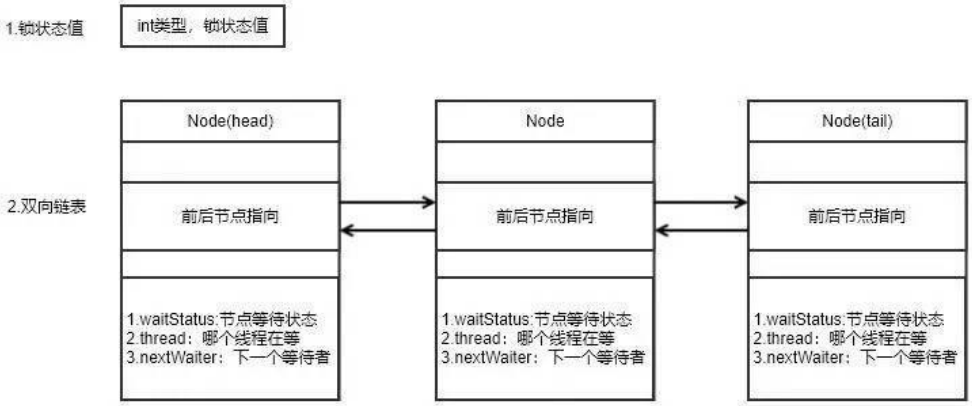
简单说来，AbstractQueuedSynchronizer 会把所有的请求线程构成一个 CLH 队列，当一个线程执行完毕（lock.unlock()）时会激活自己的后继节点，但正在执行的线程并不在队列中，而那些等待执行的线程全部处于阻塞状态，

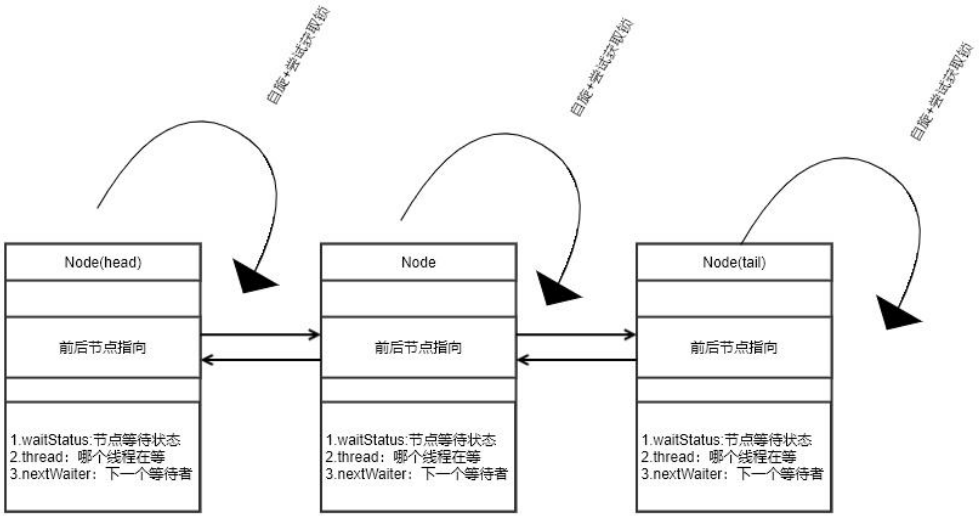
经过调查线程的显式阻塞是通过调用 LockSupport.park()完成，而 LockSupport.park()则调用 sun.misc.Unsafe.park()本地方法，再进一步，HotSpot 在 Linux 中中通过调用 pthread_mutex_lock 函数把线程交给系统内核进行阻塞。

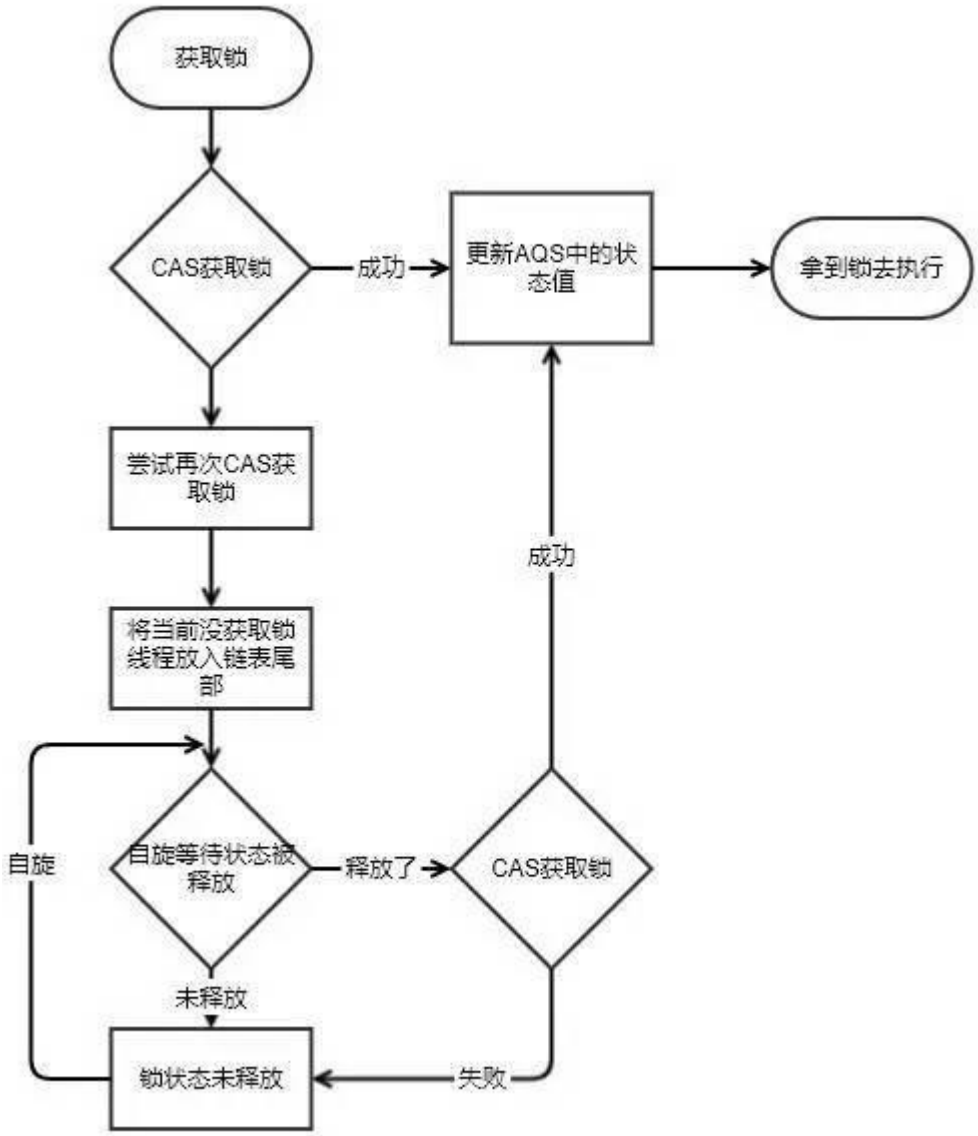
AbstractQueuedSynchronizer:发现锁的存储结构就两个东西:"双向链表" + "int 类型状态"。需要注意的是，他们的变量都被 " transient 和 volatile 修饰。

该队列如图：


Running
Next
T1(Head)
T2
T3
T4(Tail)
concurrent push thread
CAS


1.锁状态值
int类型，锁状态值
2.双向链表
Node(head)
Node
Node(tail)
前后节点指向
前后节点指向
前后节点指向
1.waitStatus:节点等待状态
2.thread：哪个线程在等
3.nextWaiter：下一个等待者
1.waitStatus:节点等待状态
2.thread：哪个线程在等
3.nextWaiter：下一个等待者
1.waitStatus:节点等待状态
2.thread：哪个线程在等
3.nextWaiter：下一个等待者


自旋+尝试获取锁
自旋+尝试获取锁
自旋+尝试获取锁
Node(head)
Node
Node(tail)
前后节点指向
前后节点指向
前后节点指向
1.waitStatus:节点等待状态
2.thread：哪个线程在等
3.nextWaiter：下一个等待者
1.waitStatus:节点等待状态
2.thread：哪个线程在等
3.nextWaiter：下一个等待者
1.waitStatus:节点等待状态
2.thread：哪个线程在等
3.nextWaiter：下一个等待者



与 synchronized 相同的是，这也是一个虚拟队列，不存在队列实例，仅存在节点之间的前后关系。

令人疑惑的是为什么采用 CLH 队列呢？原生的 CLH 队列是用于自旋锁，但 Doug Lea 把其改造为阻塞锁。

当有线程竞争锁时，该线程会首先尝试获得锁，这对于那些已经在队列中排队的线程来说显得不公平，这也是非公平锁的由来，与 synchronized 实现类似，这样会极大提高吞吐量。

如果已经存在 Running 线程，则新的竞争线程会被追加到队尾，具体是采用基于 CAS（乐观锁）的 Lock-Free 算法，因为线程并发对 Tail 调用 CAS 可能会导致其他线程 CAS 失败，解决办法是循环 CAS 直至成功。

lock 的基本操作还是通过乐观锁来实现的。

解锁

释放锁就是对 AQS 中的状态值 State 进行修改。同时更新下一个链表中的线程等待节点。

public void unlock() --> NonfairSync.release() -- > NonfairSync.tryRelease()

Lock 和 Synchronize 区别:

1.两者所处层面不同

synchronized 是 Java 中的一个关键字，当我们调用它时会从在虚拟机指令层面加锁，关键字为 monitorenter 和 monitorexit

Lock 是 Java 中的一个接口,它有许多的实现类来为它提供各种功能，加锁的关键代码为大体为 Lock 和 unLock;

2、获锁方式

synchronized 可对实例方法、静态方法和代码块加锁，相对应的，加锁前需要获得实例对象的锁或类对象的锁或指定对象的锁。说到底就是要先获得对象的监视器（即对象的锁）然后才能够进行相关

操作。

Lock 的使用离不开它的实现类 AQS，而它的加锁并不是针对对象的，而是针对当前线程的，并且 AQS 中有一个原子类 state 来进行加锁次数的计数

3、获锁失败

使用关键字 synchronized 加锁的程序中，获锁失败的对象会被加入到一个虚拟的等待队列中被阻塞，直到锁被释放；1.6 以后加入了自旋操作

使用 Lock 加锁的程序中，获锁失败的线程会被自动加入到 AQS 的等待队列中进行自旋，自旋的同时再尝试去获取锁，等到自旋到一定次数并且获锁操作未成功，线程就会被阻塞

4、偏向或重入

synchronized 中叫做偏向锁

当线程访问同步块时，会使用 CAS 将线程 ID 更新到锁对象的 Mark Word 中，如果更新成功则获得偏向锁，并且之后每次进入这个对象锁相关的同步块时都不需要再次获取锁了。

Lock 中叫做重入锁

AQS 的实现类 ReentrantLock 实现了重入的机制，即若线程 a 已经获得了锁，a 再次请求锁时则会判断 a 是否持正有锁，然后会将原子值 state+1 来实现重入的计数操作

5、Lock 独有的队列

condition 队列是 AQS 中的一个 Lock 的子接口的内部现类，它

一般会和 ReentrantLock 一起使用来满足除了加锁和解锁以外的一些附加条件，比如对线程的分组和临界数量的判断（阻塞队列）

6、解锁操作

synchronized：不能指定解锁操作，执行完代码块的对象会自动释放锁

Lock：可调用 ulock 方法去释放锁比 synchronized 更灵活

32. 死锁的四个必要条件？

产生死锁的原因主要是：

（1） 因为系统资源不足。

（2） 进程运行推进的顺序不合适。

（3） 资源分配不当等。

如果系统资源充足，进程的资源请求都能够得到满足，死锁出现的可能性就很低，否则

就会因争夺有限的资源而陷入死锁。其次，进程运行推进顺序与速度不同，也可能产生死锁。

产生死锁的四个必要条件：

（1）**互斥条件：**一个资源每次只能被一个进程使用。

（2）**请求与保持条件：**一个进程因请求资源而阻塞时，对已获得的资源保持不放。

（3）**不剥夺条件**:进程已获得的资源，在未使用完之前，不能强行剥夺。

（4）**循环等待条件**:若干进程之间形成一种头尾相接的循环等待资源关系。

这四个条件是死锁的必要条件，只要系统发生死锁，这些条件必然成立，而只要上述条件之一不满足，就不会发生死锁。

死锁的解除与预防：

理解了死锁的原因，尤其是产生死锁的四个必要条件，就可以最大可能地避免、预防和解除死锁。

所以，在系统设计、进程调度等方面注意如何不让这四个必要条件成立，如何确定资源的合理分配算法，避免进程永久占据系统资源。

此外，也要防止进程在处于等待状态的情况下占用资源。

因此，对资源的分配要给予合理的规划。

33. 怎么避免死锁？.

https://blog.csdn.net/ls5718/article/details/51896159

一、死锁的定义

多线程以及多进程改善了系统资源的利用率并提高了系统的处理能力。然而，并发执行也带来了新的问题——死锁。所谓死锁是指多个线程因竞争资源而造成的一种僵局（互相等待），若无外力作用，

这些进程都将无法向前推进。

下面我们通过一些实例来说明死锁现象。

先看生活中的一个实例，2 个人一起吃饭但是只有一双筷子，2 人轮流吃（同时拥有 2 只筷子才能吃）。某一个时候，一个拿了左筷子，一人拿了右筷子，2 个人都同时占用一个资源，等待另一个资源，这个时候甲在等待乙吃完并释放它占有的筷子，同理，乙也在等待甲吃完并释放它占有的筷子，这样就陷入了一个死循环，谁也无法继续吃饭。。。

二、死锁产生的原因

1.系统资源的竞争

通常系统中拥有的不可剥夺资源，其数量不足以满足多个进程运行的需要，使得进程在运行过程中，会因争夺资源而陷入僵局，如磁带机、打印机等。只有对不可剥夺资源的竞争 才可能产生死锁，对可剥夺资源的竞争是不会引起死锁的。

2.进程推进顺序非法

进程在运行过程中，请求和释放资源的顺序不当，也同样会导致死锁。例如，并发进程 P1、P2 分别保持了资源 R1、R2，而进程 P1 申请资源 R2，进程 P2 申请资源 R1 时，两者都 会因为所需资源被占用而阻塞。

信号量使用不当也会造成死锁。进程间彼此相互等待对方发来的消息，结果也会使得这些进程间无法继续向前推进。(例如，进程 A 等待进程 B 发的消息，进程 B 又在等待进程 A 发的消息，可以看出进程 A 和 B 不是因为竞争同一资源，而是在等待对方的资源导致死锁。)

3.死锁产生的必要条件

产生死锁必须同时满足以下四个条件，只要其中任一条件不成立，死锁就不会发生。

(1)互斥条件：进程要求对所分配的资源（如打印机）进行排他性控制，即在一段时间内某 资源仅为一个进程所占有。此时若有其他进程请求该资源，则请求进程只能等待。

(2)不剥夺条件：进程所获得的资源在未使用完毕之前，不能被其他进程强行夺走，即只能由获得该资源的进程自己来释放(只能是主动释放)。

(3)请求和保持条件：进程已经保持了至少一个资源，但又提出了新的资源请求，而该资源已被其他进程占有，此时请求进程被阻塞，但对自己已获得的资源保持不放。

(4)循环等待条件：存在一种进程资源的循环等待链，链中每一个进程已获得的资源同时被链中下一个进程所请求。即存在一个处于等待状态的进程集合{Pl, P2, ..., pn},其中Pi等 待的资源被P(i+1)

占有（i=0, 1, ..., n-1)，Pn 等待的资源被 P0 占有，如图 2-15 所示。

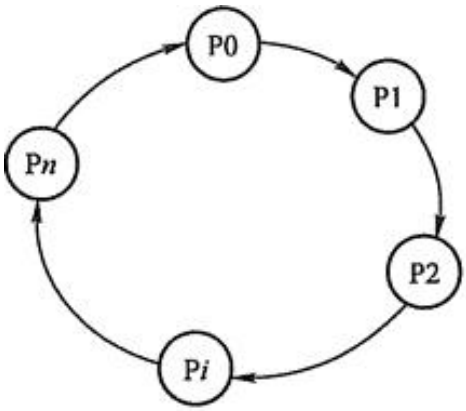


图2-15 循环等待

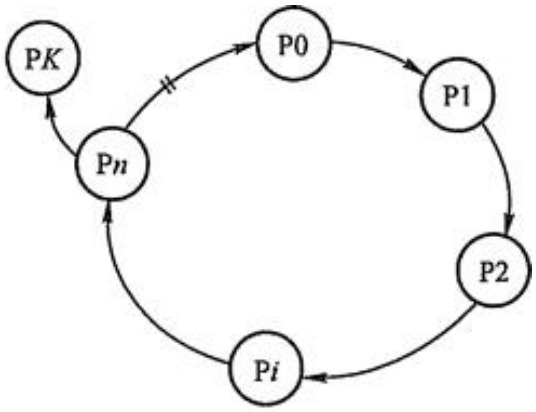


图2-16 满足条件但无死循环

直观上看，循环等待条件似乎和死锁的定义一样，其实不然。按死锁定义构成等待环所要求的条件更严，它要求 Pi 等待的资源必须由 P(i+1)来满足，而循环等待条件则无此限制。(例如，系统中有两台输出设备,P0 占有一台,PK 占有另一台,且 K 不属于集合{0, 1, ..., n}。Pn 等待一台输出设备，它可以从 P0 获得，也可能从 PK 获得。因此，虽然 Pn、P0 和其他 一些进程形成了循环等待圈，但 PK 不在圈内，若 PK 释放了输出设备，则可打破循环等待，如图 2-16 所示。因此循环等待只是死锁的必要条件。)

资源分配图含圈而系统又不一定有死锁的原因是同类资源数大于 1。但若系统中每类资源都只有一个资源，则资源分配图含圈就变成了系统出现死锁的充分必要条件。

```java
/**
* 一个简单的死锁类
* 当DeadLock类的对象flag==1时（td1），先锁定o1,睡眠500毫秒
* 而td1在睡眠的时候另一个flag==0的对象（td2）线程启动，先锁定o2,睡眠500毫秒
* td1睡眠结束后需要锁定o2才能继续执行，而此时o2已被td2锁定；
* td2睡眠结束后需要锁定o1才能继续执行，而此时o1已被td1锁定；
* td1、td2相互等待，都需要得到对方锁定的资源才能继续执行，从而死锁。
*/
public class DeadLock implements Runnable {
    public int flag = 1;
    //静态对象是类的所有对象共享的
    private static Object o1 = new Object(), o2 = new Object();
    @Override
    public void run() {
        System.out.println("flag=" + flag);
        if (flag == 1) {
            synchronized (o1) {
                try {
                    Thread.sleep(500);
                } catch (Exception e) {
                    e.printStackTrace();
                }
                synchronized (o2) {
                    System.out.println("1");
                }
            }
        }
        if (flag == 0) {
            synchronized (o2) {
                try {
                    Thread.sleep(500);
                } catch (Exception e) {
                    e.printStackTrace();
                }
                synchronized (o1) {
                    System.out.println("0");
                }
            }
        }
    }

    public static void main(String[] args) {

        DeadLock td1 = new DeadLock();
        DeadLock td2 = new DeadLock();
        td1.flag = 1;
        td2.flag = 0;
        //td1,td2都处于可执行状态，但JVM线程调度先执行哪个线程是不确定的。
        //td2的run()可能在td1的run()之前运行
        new Thread(td1).start();
        new Thread(td2).start();

    }
```

三、如何避免死锁

在有些情况下死锁是可以避免的。三种用于避免死锁的技术：

1.加锁顺序（线程按照一定的顺序加锁）

2.加锁时限（线程尝试获取锁的时候加上一定的时限，超过时限则放弃对该锁的请求，并释放自己占有的锁）

3.死锁检测

(1)加锁顺序：

当多个线程需要相同的一些锁，但是按照不同的顺序加锁，死锁就很容易发生。

如果能确保所有的线程都是按照相同的顺序获得锁，那么死锁就不会发生。

(2)加锁时限：

另外一个可以避免死锁的方法是在尝试获取锁的时候加一个超时时间，这也就意味着在尝试获取锁的过程中若超过了这个时限该线程则放弃对该锁请求。若一个线程没有在给定的时限内成功获得所有需要的锁，则会进行回退并释放所有已经获得的锁，然后等待一段随机的时间再重试。这段随机的等待时间让其它线程有机会尝试获取相同的这些锁，并且让该应用在没有获得锁的时候可以继续运行(加锁超时后可以先继续运行干点其它事情，再回头来重复之前加锁的逻辑)。

(3)死锁检测：

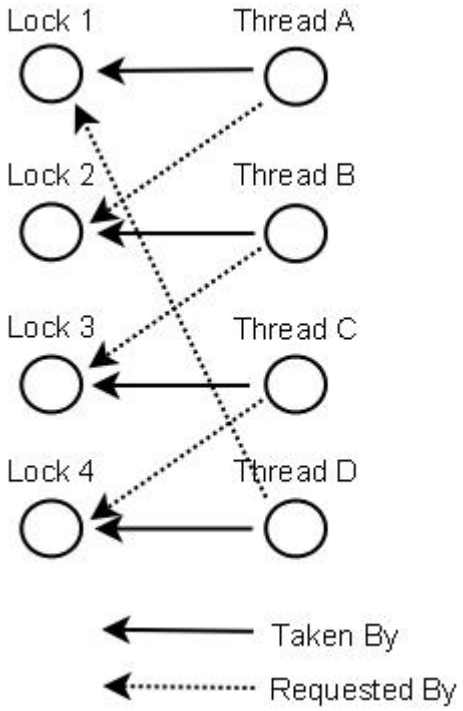
死锁检测是一个更好的死锁预防机制，它主要是针对那些不可能实现按序加锁并且锁超时也不可行的场景。

每当一个线程获得了锁，会在线程和锁相关的数据结构中（map、graph 等等）将其记下。除此之外，每当有线程请求锁，也需要记录在这个数据结构中。

当一个线程请求锁失败时，这个线程可以遍历锁的关系图看看是否有死锁发生。（例如，线程 A 请求锁 7，但是锁 7 这个时候被线程 B 持有，这时线程 A 就可以检查一下线程 B 是否已经请求了线程 A 当前所持有的锁。如果线程 B 确实有这样的请求，那么就是发生了死锁（线程 A 拥有锁 1，请求锁 7；线程 B 拥有锁 7，请求锁 1））。

当然，死锁一般要比两个线程互相持有对方的锁这种情况要复杂的多。线程 A 等待线程 B，线程 B 等待线程 C，线程 C 等待线程 D，线程 D 又在等待线程 A。线程 A 为了检测死锁，它需要递进地检测所有被 B 请求的锁。从线程 B 所请求的锁开始，线程 A 找到了线程 C，然后又找到了线程 D，发现线程 D 请求的锁被线程 A 自己持有着。这是它就知道发生了死锁。

下面是一幅关于四个线程（A,B,C 和 D）之间锁占有和请求的关系图。像这样的数据结构就可以被用来检测死锁。



那么当检测出死锁时，这些线程该做些什么呢？

(1)一个可行的做法是释放所有锁，回退，并且等待一段随机的时间后重试。这个和简单的加锁超时类似，不一样的是只有死锁已经发生了才回退，而不会是因为加锁的请求超时了。虽然有回退和等待，但是如果有大量的线程竞争同一批锁，它们还是会重复地死锁（原因同超时类似，不能从根本上减轻竞争）。

34. 对象锁和类锁是否会互相影响？

https://blog.csdn.net/codeharvest/article/details/70649375

对象锁：

Java 的所有对象都含有 1 个互斥锁，这个锁由 JVM 自动获取和释放。

线程进入 synchronized 方法的时候获取该对象的锁，当然如果已经有线程获取了这个对象的锁，那么当前线程会等待；

synchronized 方法正常返回或者抛异常而终止，JVM 会自动释放对象锁。

这里也体现了用 synchronized 来加锁的 1 个好处，方法抛异常的时候，锁仍然可以由 JVM 来自动释放。

类锁：

对象锁是用来控制实例方法之间的同步，类锁是用来控制静态方法（或静态变量互斥体）之间的同步。

其实类锁只是一个概念上的东西，并不是真实存在的，它只是用来帮助我们理解锁定实例方法和静态方法的区别的。

我们都知道，java 类可能会有很多个对象，但是只有 1 个 Class 对象，也就是说类的不同实例之间共享该类的 Class 对象。Class 对象其实也仅仅是 1 个 java 对象，只不过有点特殊而已。

由于每个 java 对象都有 1 个互斥锁，而类的静态方法是需要 Class 对象。所以所谓的类锁，不过是 Class 对象的锁而已。获取类的 Class 对象有好几种，最简单的就是 MyClass.class 的方式。

不会相互影响：类锁和对象锁不是同 1 个东西，一个是类的 Class 对象的锁，一个是类的实例的锁。也就是说：1 个线程访问静态 synchronized 的时候，允许另一个线程访问对象的实例 synchronized 方法。反过来也是成立的，因为他们需要的锁是不同的。

35. 什么是线程池，如何使用?

https://www.jianshu.com/p/210eab345423

一、为什么用线程池

1、创建/销毁线程伴随着系统开销，过于频繁的创建/销毁线程，会很大程度上影响处理效率

例如：

记创建线程消耗时间 T1，执行任务消耗时间 T2，销毁线程消耗时间 T3

如果 T1+T3>T2，那么是不是说开启一个线程来执行这个任务太不划算了！

正好，线程池缓存线程，可用已有的闲置线程来执行新任务，避免了 T1+T3 带来的系统开销

2、线程并发数量过多，抢占系统资源从而导致阻塞

我们知道线程能共享系统资源，如果同时执行的线程过多，就有

可能导致系统资源不足而产生阻塞的情况。运用线程池能有效的控制线程最大并发数，避免以上的问题

3、对线程进行一些简单的管理。比如：延时执行、定时循环执行的策略等。运用线程池都能进行很好的实现

线程池 ThreadPoolExecutor

既然 Android 中线程池来自于 Java，那么研究 Android 线程池其实也可以说是研究 Java 中的线程池

在 Java 中，线程池的概念是 Executor 这个接口，具体实现为 ThreadPoolExecutor 类，学习 Java 中的线程池，就可以直接学习他了

对线程池的配置，就是对 ThreadPoolExecutor 构造函数的参数的配置，既然这些参数这么重要，就来看看构造函数的各个参数吧

一、ThreadPoolExecutor 提供了四个构造函数

```
public ThreadPoolExecutor(int corePoolSize,
                          int maximumPoolSize,
                          long keepAliveTime,
                          TimeUnit unit,
                          BlockingQueue<Runnable> workQueue)
```

```
public ThreadPoolExecutor(int corePoolSize,
                          int maximumPoolSize,
                          long keepAliveTime,
                          TimeUnit unit,
                          BlockingQueue<Runnable> workQueue,
                          ThreadFactory threadFactory)
```

```
public ThreadPoolExecutor(int corePoolSize,
                          int maximumPoolSize,
                          long keepAliveTime,
                          TimeUnit unit,
                          BlockingQueue<Runnable> workQueue,
                          RejectedExecutionHandler handler)
```

```
public ThreadPoolExecutor(int corePoolSize,
                          int maximumPoolSize,
                          long keepAliveTime,
                          TimeUnit unit,
                          BlockingQueue<Runnable> workQueue,
                          ThreadFactory threadFactory,
```

```
RejectedExecutionHandler handler)
```

（1）int corePoolSize => **该线程池中核心线程数最大**值

核心线程：

线程池新建线程的时候，如果当前线程总数小于 corePoolSize，则新建的是核心线程，如果超过 corePoolSize，则新建的是非核心线程

核心线程默认情况下会一直存活在线程池中，即使这个核心线程啥也不干(闲置状态)。

如果指定 ThreadPoolExecutor 的 allowCoreThreadTimeOut 这个属性为 true，那么核心线程如果不干活(闲置状态)的话，超过一定时间(时长下面参数决定)，就会被销毁掉

很好理解吧，正常情况下你不干活我也养你，因为我总有用到你的时候，但有时候特殊情况(比如我自己都养不起了)，那你不干活我就要把你干掉了

（2）int maximumPoolSize

该线程池中**线程总数最大值**

线程总数 = 核心线程数 + 非核心线程数。非核心线程：不是核心线程的线程

（3）long keepAliveTime

该线程池中**非核心线程闲置超时时长**

一个非核心线程，如果不干活(闲置状态)的时长超过这个参数所设定的时长，就会被销毁掉

如果设置 allowCoreThreadTimeOut = true，则会作用于核心线程

（4）TimeUnit unit

keepAliveTime 的单位，TimeUnit 是一个枚举类型，其包括：

1) NANOSECONDS ： 1 微毫秒 = 1 微秒 / 1000

2) MICROSECONDS ： 1 微秒 = 1 毫秒 / 1000

3) MILLISECONDS ： 1 毫秒 = 1 秒 /1000

4) SECONDS ： 秒

5) MINUTES ： 分

6) HOURS ： 小时

7) DAYS ： 天

（5）BlockingQueue<Runnable> workQueue

该线程池中的任务队列：维护着等待执行的 Runnable 对象

当所有的核心线程都在干活时，新添加的任务会被添加到这个队列中等待处理，如果队列满了，则新建非核心线程执行任务

常用的 workQueue 类型：

1）SynchronousQueue：（这个消息队列相当于不存在一样）这个队列接收到任务的时候，会直接提交给线程处理，而不保留它，如果所有线程都在工作那就新建一个线程来处理这个任务！

所以为了保证不出现<线程数达到了 maximumPoolSize 而不能新建线程>的错误，使用这个类型队列的时候，maximumPoolSize 一般

指定成 Integer.MAX_VALUE，即无限大

2）**LinkedBlockingQueue：**（这个线程队列只给核心线程处理）这个队列接收到任务的时候，如果当前线程数小于核心线程数，则新建线程(核心线程)处理任务；如果当前线程数等于核心线程数，则进入队列等待。

由于这个队列没有最大值限制，即所有超过核心线程数的任务都将被添加到队列中，这也就导致了 maximumPoolSize 的设定失效，因为总线程数永远不会超过 corePoolSize

3）**ArrayBlockingQueue：**（标注消息队列）可以限定队列的长度，接收到任务的时候，如果没有达到 corePoolSize 的值，则新建线程(核心线程)执行任务，如果达到了，则入队等候，如果队列已满，则新建线程（非核心线程）执行任务，又如果总线程数到了 maximumPoolSize，并且队列也满了，则发生错误

4）**DelayQueue：**（会给线程延迟的消息队列）队列内元素必须实现 Delayed 接口，这就意味着你传进去的任务必须先实现 Delayed 接口。这个队列接收到任务时，首先先入队，只有达到了指定的延时时间，才会执行任务

（6）ThreadFactory threadFactory

创建线程的方式，这是一个接口，你 new 他的时候需要实现他的 Thread newThread(Runnable r)方法

例子：AsyncTaskk 新建线程池的 threadFactory 参数源码：

```
new ThreadFactory() {
        private final AtomicInteger mCount = new AtomicInteger(1);
        public Thread newThread(Runnable r) {
        return new Thread(r,"AsyncTask #" +
mCount.getAndIncrement());
        }
}
```

上述代码就给线程起了名。

（7）RejectedExecutionHandler handler

这是跑出异常专用的。比如上面提到的两个错误发生了，就会由这个 handler 抛出异常，你不指定他也有个默认的

一般用不上，新建一个线程池的时候，一般只用 5 个参数的构造函数。

二、向 ThreadPoolExecutor 添加任务

向线程池提交一个要执行的任务：通过 ThreadPoolExecutor.execute(Runnable command)方法即可向线程池内添加一个任务

三、ThreadPoolExecutor 的策略

上面介绍参数的时候其实已经说到了 ThreadPoolExecutor 执行的策略，这里给总结一下，当一个任务被添加进线程池时：

1.线程数量未达到 corePoolSize，则新建一个线程(核心线程)执行任务

2.线程数量达到了 corePools，则将任务移入队列等待

3.队列已满，新建线程(非核心线程)执行任务

4.队列已满，总线程数又达到了 maximumPoolSize，就会由上面那位星期天(RejectedExecutionHandler)抛出异常

常见四种线程池

（1）CachedThreadPool()（可缓存线程池）：

1.线程数无限制

2.有空闲线程则复用空闲线程，若无空闲线程则新建线程

3.一定程序减少频繁创建/销毁线程，减少系统开销

（2）FixedThreadPool()（定长线程池）：

1.可控制线程最大并发数（同时执行的线程数）

2.超出的线程会在队列中等待

（3）ScheduledThreadPool()（定长线程池）

1.支持定时及周期性任务执行。

（4）SingleThreadExecutor()（单线程化的线程池）

1.有且仅有一个工作线程执行任务

2.所有任务按照指定顺序执行，即遵循队列的入队出队规则

36. Java 的并发、多线程、线程模型（复杂，暂时先不管）

37. 谈谈对多线程的理解

https://www.jianshu.com/p/40d4c7aebd66

用多线程只有一个目的，那就是更好的利用 cpu 的资源，因为所有的多线程代码都可以用单线程来实现。

说这个话其实只有一半对，因为反应“多角色”的程序代码，最起码每个角色要给他一个线程吧，否则连实际场景都无法模拟，当然也没法说能用单线程来实现：比如最常见的“生产者，消费者模型”。

基本知识：

（1）**多线程**：指的是这个程序（一个进程）运行时产生了不止一个线程

（2）**并行与并发**：

1）并行：多个 cpu 实例或者多台机器同时执行一段处理逻辑，是真正的同时。

2）并发：通过 cpu 调度算法，让用户看上去同时执行，实际上从 cpu 操作层面不是真正的同时。并发往往在场景中有公用的资源，那么针对这个公用的资源往往产生瓶颈，我们会用 TPS 或者 QPS 来反应这个系统的处理能力。

（3）**线程安全**：经常用来描绘一段代码。指在并发的情况之下，

该代码经过多线程使用，线程的调度顺序不影响任何结果。这个时候使用多线程，我们只需要关注系统的内存，cpu 是不是够用即可。反过来，线程不安全就意味着线程的调度顺序会影响最终结果（如不加事务的转账代码）。

（4）同步：Java 中的同步指的是通过人为的控制和调度，保证共享资源的多线程访问成为线程安全，来保证结果的准确。如上面的代码简单加入@synchronized 关键字。在保证结果准确的同时，提高性能，才是优秀的程序。线程安全的优先级高于性能。

多线程的内容分类：

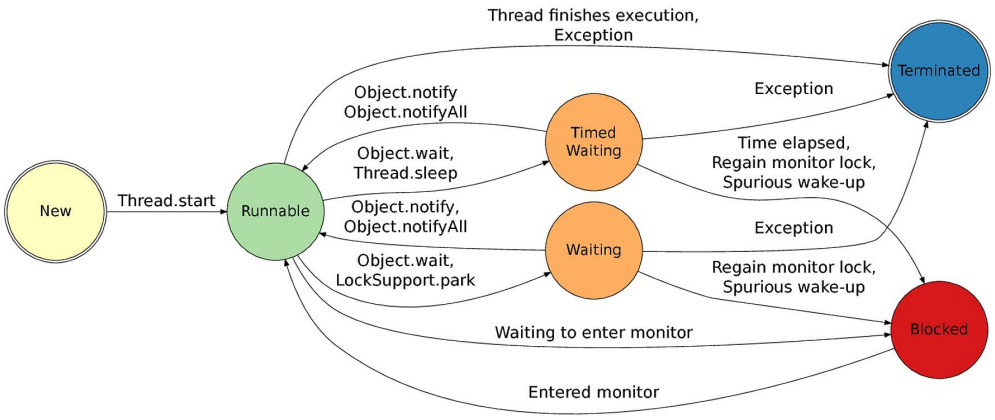
1.线程的状态

2.每个对象都有的方法（机制）

3.基本线程类

4.高级多线程控制类

（1）线程的状态



各种状态一目了然，值得一提的是"Blocked"和"Waiting"这两个状态的区别：

线程在 Running 的过程中可能会遇到阻塞(Blocked)情况：

对 Running 状态的线程加同步锁(Synchronized)使其进入(lock blocked pool)，同步锁被释放进入可运行状态(Runnable)。（从 jdk 源码注释来看，blocked 指的是对 monitor 的等待（可以参考下文的图）即该线程位于等待区。）

线程在 Running 的过程中可能会遇到等待（Waiting）情况：

线程可以主动调用 object.wait 或者 sleep，或者 join（join 内部调用的是 sleep，所以可看成 sleep 的一种）进入。（从 jdk 源码注释来看，waiting 是等待另一个线程完成某一个操作，如 join 等待另一个完成执行，object.wait()等待 object.notify()方法执行。）

Waiting 状态和 Blocked 状态有点费解，我个人的理解是：

Blocked 其实也是一种 wait，等待的是 monitor，

但是和 Waiting 状态不一样，

举个例子，有三个线程进入了同步块，其中两个调用了 object.wait()，进入了 waiting 状态，这时第三个调用了 object.notifyAll()，这时候前两个线程就一个转移到了 Runnable，一个转移到了 Blocked。

从下文的 monitor 结构图来区别：

每个 Monitor 在某个时刻，只能被一个线程拥有，该线程就是“Active Thread”，而其它线程都是 “Waiting Thread”，分别在两个队列 “ Entry Set”和 “Wait Set”里面等候。

在 “Entry Set”中等待的线程状态 Blocked,从 jstack 的 dump 中来看是 “Waiting for monitor entry”，

在 “Wait Set”中等待的线程状态是 Waiting，表现在 jstack 的 dump 中是 “in Object.wait()”。

此外，在 runnable 状态的线程是处于被调度的线程，此时的调度顺序是不一定的。Thread 类中的 yield 方法可以让一个 running 状态的线程转入 runnable。

（2）每个对象都有的方法（机制）

synchronized, wait, notify 是任何对象都具有的同步工具。让我们先来了解他们

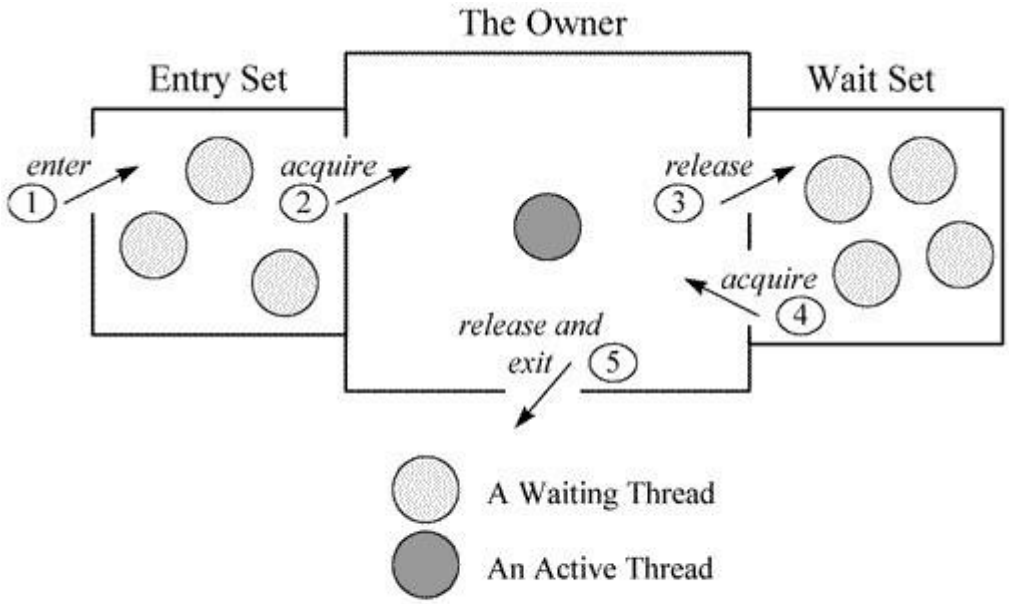


Figure 20-1. A Java monitor.

他们是**应用于同步问题的人工线程调度工具**。

讲其本质，首先就要明确 monitor 的概念，Java 中的每个对象都有一个监视器，来监测并发代码的重入。在非多线程编码时该监视器不发挥作用，反之如果在 synchronized 范围内，监视器发挥作用。

wait/notify 必须存在于 synchronized 块中。并且，这三个关键字针对的是同一个监视器（某对象的监视器）。这意味着 wait 之后，其他线程可以进入同步块执行。

当某代码并不持有监视器的使用权时去 wait 或 notify，会抛出 java.lang.IllegalMonitorStateException。也包括在 synchronized 块中去调用另一个对象的 wait/notify，因为不同对象的监视器不同，同样会抛出此异常。

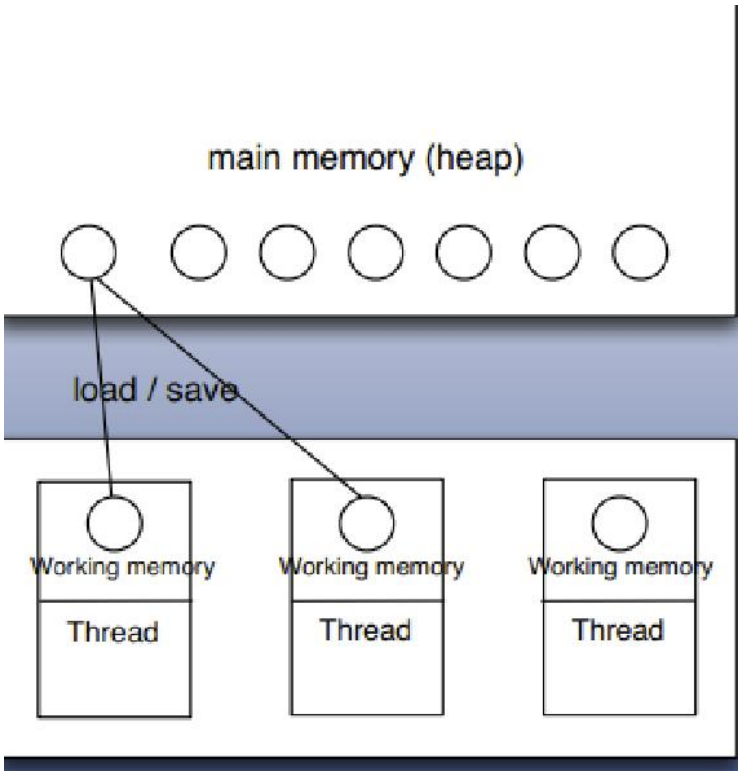
再讲用法：

（1）synchronized 单独使用：

（2）代码块：如下，在多线程环境下，synchronized 块中的方法获取了 lock 实例的 monitor，如果实例相同，那么只有一个线程能执行该块内容

volatile

多线程的内存模型：main memory（主存）、working memory（线程栈），在处理数据时，线程会把值从主存 load 到本地栈，完成操作后再 save 回去(volatile 关键词的作用：每次针对该变量的操作都激发一次 load and save)。



针对多线程使用的变量如果不是 volatile 或者 final 修饰的，

很有可能产生不可预知的结果（另一个线程修改了这个值，但是之后在某线程看到的是修改之前的值）。其实道理上讲同一实例的同一属性本身只有一个副本。但是多线程是会缓存值的，本质上，volatile 就是不去缓存，直接取值。在线程安全的情况下加 volatile 会牺牲性能。

（3）基本线程类

基本线程类指的是 Thread 类(Thread 类实现了 Runnable 接口)、Runnable 接口、Callable 接口

Thread：

关于中断：它并不像 stop 方法那样会中断一个正在运行的线程。线程会不时地检测中断标识位，以判断线程是否应该被中断（中断标识值是否为 true）。终端只会影响到 wait 状态、sleep 状态和 join 状态。被打断的线程会抛出 InterruptedException。

Thread.interrupted()检查当前线程是否发生中断，返回 boolean

synchronized 在获锁的过程中是不能被中断的。

中断是一个状态！interrupt()方法只是将这个状态置为 true 而已。所以说正常运行的程序不去检测状态，就不会终止，而 wait 等阻塞方法会去检查并抛出异常。如果在正常运行的程序中添加 while(!Thread.interrupted()) ，则同样可以在中断后离开代码体

Runnable

与 Thread 类似

Callable

future 模式：并发模式的一种，可以有两种形式，即无阻塞和阻塞，分别是 isDone 和 get。其中 Future 对象用来存放该线程的返回值以及状态

（4）高级多线程控制类

1.ThreadLocal 类

用处：保存线程的独立变量。对一个线程类（继承自 Thread）当使用 ThreadLocal 维护变量时，ThreadLocal 为每个使用该变量的线程提供独立的变量副本，所以每一个线程都可以独立地改变自己的副本，而不会影响其它线程所对应的副本。常用于用户登录控制，如记录 session 信息。

实现：每个 Thread 都持有一个 TreadLocalMap 类型的变量（该类是一个轻量级的 Map，功能与 map 一样，区别是桶里放的是 entry 而不是 entry 的链表。功能还是一个 map。）以本身为 key，以目标为 value。

主要方法是 get()和 set(T a)，set 之后在 map 里维护一个 threadLocal -> a，get 时将 a 返回。ThreadLocal 是一个特殊的容器。

2.原子类（AtomicInteger、AtomicBoolean……）

如果使用 atomic wrapper class 如 atomicInteger，或者使用

自己保证原子的操作，则等同于 synchronized

```
//返回值为 boolean
AtomicInteger.compareAndSet(int expect,int update)
```

该方法可用于实现乐观锁，考虑文中最初提到的如下场景：a 给 b 付款 10 元，a 扣了 10 元，b 要加 10 元。此时 c 给 b2 元，但是 b 的加十元代码约为：

```
if(b.value.compareAndSet(old, value)){
  return ;
}else{
  //try again
  // if that fails, rollback and log
}
```

AtomicReference

对于 AtomicReference 来讲，也许对象会出现，属性丢失的情况，即 oldObject == current，但是 oldObject.getPropertyA != current.getPropertyA。

这时候，AtomicStampedReference 就派上用场了。这也是一个很常用的思路，即加上版本号

3.Lock 类

lock: 在 java.util.concurrent 包内。共有三个实现：

（1）ReentrantLock

（2）ReentrantReadWriteLock.ReadLock

（3）ReentrantReadWriteLock.WriteLock

主要目的是和 synchronized 一样， 两者都是为了解决同步问

题，处理资源争端而产生的技术。功能类似但有一些区别。

区别如下：

1. lock 更灵活，可以自由定义多把锁的枷锁解锁顺序，而 synchronized 要按照先加的后解顺序。

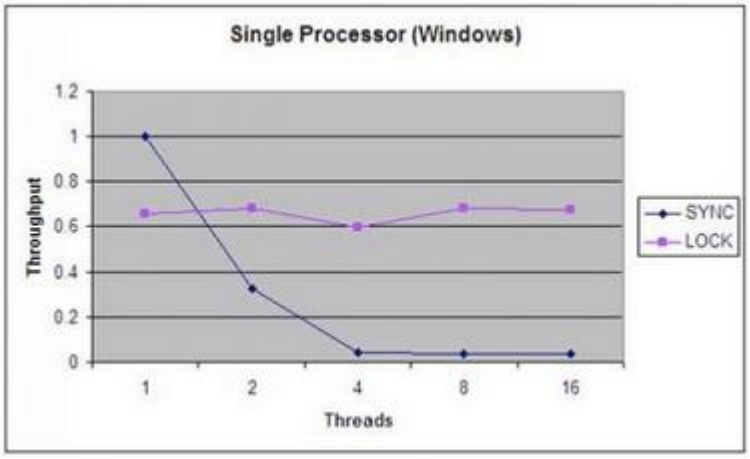
2. 提供多种加锁方案，lock 阻塞式，trylock 无阻塞式，lockInterruptily 可打断式， 还有 trylock 的带超时时间版本。

3. 本质上和监视器锁（即 synchronized）是一样的

4. 能力越大，责任越大，必须控制好加锁和解锁，否则会导致灾难。

5. 和 Condition 类的结合。

6. 性能更高



（1）ReentrantLock

可重入的意义在于持有锁的线程可以继续持有，并且要释放对等的次数后才真正释放该锁。

使用方法是：

1.先 new 一个实例

```
static ReentrantLock r=new ReentrantLock();
```

2.加锁

```
r.lock()或r.lockInterruptibly();
```

此处也是个不同，后者可被打断。当 a 线程 lock 后，b 线程阻塞，此时如果是 lockInterruptibly，那么在调用 b.interrupt()之后，b 线程退出阻塞，并放弃对资源的争抢，进入 catch 块。（如果使用后者，必须 throw interruptable exception 或 catch）

3.释放锁

```
r.unlock()
```

必须做！何为必须做呢，要放在 finally 里面。以防止异常跳出了正常流程，导致灾难。（finally 是可以信任的：经过测试，哪怕是发生了 OutofMemoryError，finally 块中的语句执行也能够得到保证。）

（2）ReentrantReadWriteLock

可重入读写锁（读写锁的一个实现）

```
ReentrantReadWriteLock lock = new ReentrantReadWriteLock()
ReadLock r = lock.readLock();
WriteLock w = lock.writeLock();
```

两者都有 lock,unlock 方法。写写，写读互斥；读读不互斥。可以实现并发读的高效线程安全代码

4.容器类

这里就讨论比较常用的两个：

（1）BlockingQueue

（2）ConcurrentHashMap

（1）BlockingQueue

阻塞队列。该类是 java.util.concurrent 包下的重要类。

这个 queue 是单向队列，可以在队列头添加元素和在队尾删除或取出元素。类似于一个管道，特别适用于先进先出策略的一些应用场景。普通的 queue 接口主要实现有 PriorityQueue（优先队列）

（2）ConcurrentHashMap

高效的线程安全哈希 map。

5.管理类

管理类的概念比较泛，用于管理线程，本身不是多线程的，但提供了一些机制来利用上述的工具做一些封装。

了解到的值得一提的管理类：ThreadPoolExecutor 和 JMX 框架下的系统级管理类 ThreadMXBean

ThreadPoolExecutor

如果不了解这个类，应该了解前面提到的 ExecutorService，开一个自己的线程池非常方便：

```
ExecutorService e = Executors.newCachedThreadPool();
    ExecutorService e = Executors.newSingleThreadExecutor();
    ExecutorService e = Executors.newFixedThreadPool(3);
    // 第一种是可变大小线程池，按照任务数来分配线程，
    // 第二种是单线程池，相当于 FixedThreadPool(1)
    // 第三种是固定大小线程池。
```

```
// 然后运行
e.execute(new MyRunnableImpl());
```

该类内部是通过 ThreadPoolExecutor 实现的，掌握该类有助于理解线程池的管理，本质上，他们都是 ThreadPoolExecutor 类的各种实现版本。

38. 多线程同步机制

1. 在需要同步的方法的方法签名中加上 synchronized 关键字
2. 使用 synchronized 关键字对需要进行同步的代码块进行同步
3. 使用 java.util.concurrent.lock 包中 Lock 对象（JDK1.8）

39. 多线程有什么要注意的问题？

并发问题，安全问题，效率问题。

40. 谈谈你对并发编程的理解并举例说明

https://www.jianshu.com/p/053943a425c3

一、为什么需要并发

并发其实是一种解耦合的策略，它帮助我们把做什么（目标）和什么时候做（时机）分开。这样做可以明显改进应用程序的吞吐量（获得更多的 CPU 调度时间）和结构（程序有多个部分在协同工作）。

二、误解和正解

最常见的对并发编程的误解有以下这些：

1.（错误）并发总能改进性能。

真相：并发在 CPU 有很多空闲时间时能明显改进程序的性能，但当线程数量较多的时候，线程间频繁的调度切换反而会让系统的性能下降

2.（错误）编写并发程序无需修改原有的设计。

真相：目的与时机的解耦往往会对系统结构产生巨大的影响

3.（错误）在使用 Web 或 EJB 容器时不用关注并发问题。

真相：只有了解了容器在做什么，才能更好的使用容器

下面的这些说法是对并发编程比较客观的认识：

A. 编写并发程序会在代码上增加额外的开销。

B. 正确的并发是非常复杂的，即使对于很简单的问题。

C. 并发中的缺陷因为不易重现也不容易被发现。

D. 并发往往需要对设计策略从根本上进行修改。

三、并发编程的原则和技巧

（1）单一职责原则：分离并发相关代码和其他代码（并发相关代码有自己的开发、修改和调优生命周期）。

（2）限制数据作用域：两个线程修改共享对象的同一字段时可能会相互干扰，导致不可预期的行为，解决方案之一是构造临界区，但是必须限制临界区的数量。

（3）使用数据副本：数据副本是避免共享数据的好方法，复制出来的对象只是以只读的方式对待。

Java 5 的 java.util.concurrent 包中增加一个名为 CopyOnWriteArrayList 的类，它是 List 接口的子类型，所以你可以认为它是 ArrayList 的线程安全的版本，它使用了写时复制的方式创建数据副本进行操作来避免对共享数据并发访问而引发的问题。

线程应尽可能独立：让线程存在于自己的世界中，不与其他线程共享数据。

四、Java 7 的并发编程

Java 7 中引入了 TransferQueue，它比 BlockingQueue 多了一个叫 transfer 的方法，如果接收线程处于等待状态，该操作可以马上将任务交给它，否则就会阻塞直至取走该任务的线程出现。可以用 TransferQueue 代替 BlockingQueue，因为它可以获得更好的性能。

伴随着 Java 7 的到来，Java 中默认的数组排序算法已经不再是经典的快速排序（双枢轴快速排序）了，新的排序算法叫 TimSort，它是归并排序和插入排序的混合体，TimSort 可以通过分支合并框架充分利用现代处理器的多核特性，从而获得更好的性能（更短的排序时间）。

41. 谈谈你对多线程同步机制的理解？

https://www.jianshu.com/p/592ef5642513

线程同步是为了确保线程安全，所谓线程安全指的是多个线程对同一资源进行访问时，有可能产生数据不一致问题，导致线程访问的资源并不是安全的。（如果多线程程序运行结果和单线程运行的结果是一样的，且相关变量的值与预期值一样，则是线程安全的。）

java 中与线程同步有关的关键字/类包括：

volatile、synchronized、Lock、AtomicInteger 等 concurrent 包下的原子类。。。等

线程安全

概述：当多个线程访问一个对象时，如果不用考虑这些线程在运行时环境下的调度和交替执行，也不需要进行额外的同步，或者在调用方进行任何其他的协调操作，调用这个对象的行为都可以获得正确

的结果，那这个对象是线程安全的。

注：这意味着如若要实现线程安全，代码本身必须要封装所有必要的正确性保障手段（比如锁的实现），以确保程序无论在多线程环境下如何调用该方法，将始终保持返回正确的结果。

Java 的线程安全

我们在讨论 Java 的线程安全，实际上讨论的是“相对线程安全”。需要保证的是单独对象操作是线程安全的，调用过程中不需要额外的保障措施，但是涉及到某些业务场景需要特定顺序连续调用，就可能需要调用者考虑使用额外的同步手段保证同步。

例如使用 ：

```
synchronized
```

Java 中与线程同步有关的关键词/类：

1.Synchronized

同步原理：

JVM 规范规定 JVM 基于进入和退出 Monitor 对象来实现方法同步和代码块同步，但两者的实现细节不一样。

代码块同步是使用 monitorenter 和 monitorexit 指令实现，而方法同步是使用另外一种方式实现的，细节在 JVM 规范里并没有详细说明，但是方法的同步同样可以使用这两个指令来实现。

monitorenter 指令是在编译后插入到同步代码块的开始位置，

而 monitorexit 是插入到方法结束处和异常处， JVM 要保证每个 monitorenter 必须有对应的 monitorexit 与之配对。

任何对象都有一个 monitor 与之关联，当且一个 monitor 被持有后，它将处于锁定状态。线程执行到 monitorenter 指令时，将会尝试获取对象所对应的 monitor 的所有权，即尝试获得对象的锁。

Sychronized 采取的同步策略是**互斥同步**

通常情况下，临界区（Critical Section）、互斥量（Mutex）和信号量（Semaphore）都是主要的互斥实现形式。在每次获取资源之前，都需要检查是否有线程占用该资源。这里有两个关键点需要注意：

（1）Sychronized 是可重入的；

（2）已经进入的线程尚未执行完，将会阻塞后面其他线程；

锁的本质是对象实例

对于非静态方法来说，Synchronized 有两种呈现形式，Synchronized 方法体和 Synchronized 语句块。两种呈现形式本质上的锁都是对象实例。

```
synchronized (synchronizeDemo)

public synchronized void doSth3() {
```

我们可以看出，从本质上而非呈现形式上看，synchronized 同步也分两种。

（1）锁类的对象实例，针对于某个具体实例普通方法/语句块的

互斥；

（2）锁类的 Class 对象，针对于 Class 类静态方法/语句块的互斥；

进程切换导致的系统开销：

Java 的线程是直接映射到操作系统线程之上的，线程的挂起、阻塞、唤醒等都需要操作系统的参与，因此在线程切换的过程中是有一定的系统开销的。

在多线程环境下调用 Synchronized 方法，有可能需要多次线程状态切换，因此可以说 Synchronized 是在 Java 语言中一个**重量级操作**。

虽然如此，JDK1.6 版本后还是对 Synchronized 关键字做了相关优化，加入**锁自旋**特性减少系统线程切换导致的开销，几乎与 ReentrantLock 的性能不相上下，因此建议在能满足业务需求的前提下，优先使用 Sychronized。

2.ReentrantLock（可重入锁）

与 Synchronized 的实现原理类似，采用的都是互斥同步策略，用法和实现效果上来说也很相似，也具备可重入的特性。

高级特性

（1）**公平锁**是指多个线程在等待同一个锁时，必须按照申请锁的时间顺序依次获得锁；而非公平锁则不保证这一点，在锁被释放时，任何一个等待锁的线程都有机会获得锁。**Sychronized 的锁是非公平**

的,ReentrantLock 默认情况下也是非公平的,但可以通过带 boolean 值的构造函数要求使用公平锁;

（2）**锁绑定多个条件**是指一个 ReentrantLock 对象可以同时绑定多个 Condition 对象，而在 Synchronized 中，锁对象的 wait()和 notify()或 notifyAll()方法可以实现一个隐含的条件，如果要多于一个条件关联的时候，就不得不额外添加一个锁，而 ReentrantLock 无需这样做，只需要多次调用 newCondition()方法即可。

AbstractQueuedSynchronizer（简称 AQS）

ReentrantLock 实现是基于 AQS 这个抽象方法的

简单来说，AQS 是通过管理状态的方式来实现相对线程安全的。Java 中信号量（Semaphore）、读写锁（ReadWriteLock）、计数器（CountDownLatch）以及 FutureTask 等都是基于 AQS 实现的，可见这个抽象类的地位多么不一般。

与其说 ReentrantLock 性能更好不如说 Synchronized 优化空间更大

上面介绍过，Synchronized 在 JDK1.6 以后性能有所增强，因此在能满足业务复杂度需求的情况下，采用 Synchronized 也未尝不可。然而**互斥同步**终究属于悲观的并发策略，在对性能要求极高的业务场景下使用以上**互斥同步**策略并不合适。

3. volatile 关键字

volatile 在多线程环境下保证了共享变量内存可见性。（意思就是线程 A 修改了 volatile 修饰的共享变量，线程 B 能够感知修改。）如果 volatile 合理使用的话，将会比 Synchronized 的执行成本更低。

从底层的角度来说，为了提高处理速度，CPU 不直接和内存进行通信，而是先将数据读入到 CPU 缓存后在进行操作，但不知何时将会更新到内存。声明变量加入 volatile 关键字后，每次修改该变量，JVM 就会通知处理器将 CPU 缓存内的值强制更新到内存中，这就是所谓的“可见性”。

4. Java 中的非阻塞同步策略

CAS 指令与原子性

原子操作的业务表现形式是“不可被中断或不可被分割操作”。所谓 CAS（Compare And Swap）比较并交换就是一种原子操作。简单来说执行 CAS 需要两个参数，一个新值，一个旧值，当比较内存的值与旧值相符时，则替换为新值，否则不执行替换操作。CPU 如何实现，这里不多说，Java 若要实现 CAS 则需要 CPU 指令集配合。

```
couter.compareAndSet(0, 1);

        System.out.println("结果为" + couter.get());// 结果为 1

        couter.compareAndSet(0, 3);

        System.out.println("结果为" + couter.get());// 结果为 1
```

除了 Integer 以外，还支持包括 CAS 更新实例、更新实例的属性等功能。

阅读源码不难发现，Java 是通过一个 sun.misc.Unsafe 的类，完成 CAS 指令操作的，然而我们从 AQS 的源码中也发现了 sun.misc.Unsafe 类的踪影。

其实不难理解，AQS 负责管理状态（也可以理解为互斥资源）——这里狭义来说可以是锁是否被线程占用的标记，当然，状态的判定规则以及互斥资源数目由 AQS 的继承者们负责实现，而状态的更新只能是通过 CAS 指令完成，以确保线程安全。

5.无同步策略

这就比较容易理解了，同步只是线程安全的一个手段，无同步并不意味着线程不安全。大致两种方法的代码可以保证没有使用同步方案的前提下的线程安全。

（1）可重入代码：例如纯计算的函数之类的，方法运行间不需要获取外部资源就可以进行计算。

（2）线程本地存储资源：线程本地维护自己的资源，根本不存在与其他线程资源冲突的可能。

42. 如何保证多线程读写文件的安全？

https://blog.csdn.net/baple/article/details/23857485

我们看多个线程共用一个 FileWriter 写入同一个文件的情况：

在 Writer 抽象类里面有一个 protected 类型的 lock 属性，是一个简单 Object 对象。

JDK 里对这个 lock 属性的描述如下：“用于同步针对此流的操作的对象。为了提高效率，字符流对象可以使用其自身以外的对象来保护关键部分。因此，子类应使用此字段中的对象，而不是 this 或者同步的方法。”——看来，多线程共用同一个 writer 的方案有戏。

FileWriter 的 JDK 说明:“某些平台一次只允许一个 FileWriter（或其他文件写入对象）打开文件进行写入”

多个线程各自开启一个 FileWriter 写入同一个文件，会发生文本串行现象。

```
//多线程争抢写入同一个文件的测试，一次一行
//多个线程公用一个FileWriter
//测试结果：
private void multiThreadWriteFile2() throws IOException{
    File file=new File(basePath+jumpPath+fileName);
    file.createNewFile();
    FileWriter fw=new FileWriter(file);

    //创建10个线程
     int totalThreads=10;
     WriteFileThread2[] threads=new WriteFileThread2[totalThreads];
     for(int i=0;i<totalThreads;i++){
        WriteFileThread2 thread=new WriteFileThread2(fw,i);
        threads[i]=thread;
     }

     //启动10个线程
     for(Thread thread: threads){
        thread.start();
     }

     //主线程休眠100毫秒
     try {
        Thread.sleep(100);
     } catch (InterruptedException e) {
        e.printStackTrace();
     }

     //所有线程停止
     for(WriteFileThread2 thread: threads){
        thread.setToStop();
     }
     System.out.println("还楞着干什么，去看一下文件结构正确与否啊！");
  }
```

FileWriter 在写入时，同步在了对应的 FileOutputStream 对象上——依此分析，多个线程共用一个 FileWriter 写入同一个文件，一次一行的情况下，不会出现串行。

```
FileWriter  fw=new  FileWriter(file);
```

线程内

WriteFileThread2 thread=new WriteFileThread2(fw,i);

没有出现任何串行现象。

BufferedWriter 又如何呢？

按道理 BufferedWriter 只是把别的 Writer 装饰了一下，在底层写的时候也是同步的。

BufferedWriter.write 和 BufferedWriter.flushBuffer 的方法同步在了被包装的 Writer 这个对象上。

也就是说，BufferedWriter.write 和 BufferedWriter.flushBuffer 都有同步块包围，说明按上述环境测试时，是不会出现串行现象的。

结论:

1.可以开多个线程操作多个 FileWriter 写入同一个文件，多个 FileWriter 切换时，会导致相互交错，破坏字符串结构的完整性。

2.多个线程操作同一个 FileWriter 或者 BufferedWriter 时，每一次写入操作都是可以保证原子性的，也即：FileWriter 或者 BufferedWriter 是线程安全的

3.由于第 2 条中的线程安全，写入速度下降超过一半。

43. 多线程断点续传原理
44. 断点续传的实现

https://blog.csdn.net/ghost_Programmer/article/details/51923895

https://blog.csdn.net/ghost_Programmer/article/details/51923895

断点续存类似于游戏里的存档

（1）如果游戏不能存档，那么则意味着我们下次游戏的时候，这次已经通过的 4 关的进度将会丢失，无法接档。

（2）如果游戏不能存档，那么则意味着我们下次游戏的时候，这次已经通过的 4 关的进度将会丢失，无法接档。

“断点续传”最最基础的原理：**我们要在下载行为出现中断的时候，记录下中断的位置信息，然后在下次行为中读取。**

在新的下载行为开始的时候，直接从记录的这个位置开始下载内容，而不再从头开始。

（1）当“上传(下载)的行为”出现中断，我们需要记录本次上传(下载)的位置(position)。

（2）当“续”这一行为开始，我们直接跳转到 postion 处继续上传(下载)的行为。

回归二进制，因为这里的本质无非就是文件的读写。

（1）那么剩下的工作就很简单了，先是记录 position，这似乎都没什么值得说的，因为只是数据的持久化而已(内存，文件，数据库)，我们有很多方式。

（2）另一个关键在于当“续传”的行为开始，我们需要需要从上次记录的 position 位置开始读写操作，所以我们需要一个类似于“指针”功能的东西。

我们当然也可以自己想办法去实现这样一个“指针”，但高兴地是，Java 已经为我们提供了这样的一个类，那就是 RandomAccessFile。

这个类的功能从名字就很直观的体现了，能够随机的去访问文件。我们看一下 API 文档中对该类的说明：

此类的实例支持对随机访问文件的读取和写入。随机访问文件的行为类似存储在文件系统中的一个大型 byte 数组。

如果随机访问文件以读取/写入模式创建，则输出操作也可用；输出操作从文件指针开始写入字节，并随着对字节的写入而前移此文件指针。

写入隐含数组的当前末尾之后的输出操作导致该数组扩展。该文件指针可以通过 getFilePointer 方法读取，并通过 seek 方法设置。

例子具体步骤：

（1）**准备存储数值：**首先，我们定义了一个变量 position，记录在发生中断的时候，已完成读写的位置。（这是为了方便，实际来说肯定应该讲这个值存到文件或者数据库等进行持久化）

（2）**中断发生：**在文件读写的 while 循环中，我们去模拟一个中断行为的发生。这里是当 targetFile 的文件长度为 3 个字节则模拟抛出一个我们自定义的异常。(我们可以想象为实际下载中，已经上传(下载)了”x”个字节的内容，这个时候网络中断了，那么我们就在网络中断抛出的异常中将”x”记录下来)。

（3）**续传：**剩下的就如果我们之前说的一样，在“续传”行为开始后，通过 RandomAccessFile 类来包装我们的文件，然后通过 seek 将指针指定到之前发生中断的位置进行读写就搞定了。

（实际的文件下载上传，我们当然需要将保存的中断值上传给服务器，这个方式通常为 httpConnection.setRequestProperty(“RANGE”,”bytes=x”);）

实现步骤：

（1）中断时记录 position

```
position = 3;
```

（2）当再次开启时使用 RandomAccessFile，从断点处开始读起

```
RandomAccessFile readFile = new RandomAccessFile(source, "rw");
RandomAccessFile writeFile = new RandomAccessFile(target, "rw");
readFile.seek(position);
writeFile.seek(position);

//数据缓冲区
byte[] buf = new byte[1];
//数据读写
while (readFile.read(buf) != -1) {
    writeFile.write(buf);
}
```