线程概念

进程是系统进行资源分配和调用的独立单位。每一个进程都有它自己的内存空间和系统资源。

线程是程序的执行单元，执行路径。是程序使用 CPU 的最基本单位。

多线程的存在，不是提高程序的执行速度。其实是为了提高应用程序的使用率。程序的执行其实都是在抢 CPU 的资源，CPU 的执行权。多个进程是在抢这个资源，而其中的某一个进程如果执行路径比较多，就会有更高的几率抢到 CPU 的执行权。我们是不敢保证哪一个线程能够在哪个时刻抢到，所以线程的执行有随机性。

线程调度

假如我们的计算机只有一个 CPU，那么 CPU 在某一个时刻只能执行一条指令，线程只有得到 CPU 时间片，也就是使用权，才可以执行指令。那么 Java 是如何对线程进行调用的呢？

线程有两种调度模型：
分时调度模型：所有线程轮流使用 CPU 的使用权，平均分配每个线程占用 CPU 的时间片
抢占式调度模型：优先让优先级高的线程使用 CPU，如果线程的优先级相同，那么会随机选择一个，优先级高的线程获取的 CPU 时间片相对多一些。

Java 使用的是抢占式调度模型。

设置和获取线程优先级的方式：
public final int getPriority()
public final void setPriority(int newPriority)

但是，即时设定了优先级也只是提高了获得 CPU 执行权的几率。

线程控制

线程休眠

当前线程休眠，不释放锁，不让其他参与同步的线程得到执行的机会

public static void sleep(long millis)

线程加入

等待该线程执行完毕
public final void join()

```
/*
 * public final void join():等待该线程终止。
 */
public class ThreadJoinDemo {
	public static void main(String[] args) {
		ThreadJoin tj1 = new ThreadJoin();
		ThreadJoin tj2 = new ThreadJoin();
		ThreadJoin tj3 = new ThreadJoin();

		tj1.setName("李渊");
		tj2.setName("李世民");
		tj3.setName("李元霸");

		tj1.start();
		try {
			tj1.join();
		} catch (InterruptedException e) {
			e.printStackTrace();
		}

		tj2.start();
		tj3.start();
	}
}
```

线程礼让

暂停当前正在执行的线程对象，并执行其他线程，让多个线程的执行更和谐，但是不能靠它保证一人一次。

public static void yield()

```
public class ThreadYield extends Thread {
	@Override
	public void run() {
```

```
            for (int x = 0; x < 100; x++) {
                System.out.println(getName() + ":" + x);
                Thread.yield();
            }
        }
}


/*
 * public static void yield():暂停当前正在执行的线程对象，并执行其他线程。
 * 让多个线程的执行更和谐，但是不能靠它保证一人一次。
 */
public class ThreadYieldDemo {
    public static void main(String[] args) {
            ThreadYield ty1 = new ThreadYield();
            ThreadYield ty2 = new ThreadYield();

            ty1.setName("林青霞");
            ty2.setName("刘意");

            ty1.start();
            ty2.start();
        }
}
```

后台线程

在 Java 中有两类线程：User Thread(用户线程)、Daemon Thread(守护线程)

用个比较通俗的比如，任何一个守护线程都是整个 JVM 中所有非守护线程的保姆：

只要当前 JVM 实例中尚存在任何一个非守护线程没有结束，守护线程就全部工作；只有当最后一个非守护线程结束时，守护线程随着 JVM 一同结束工作。Daemon 的作用是为其他线程的运行提供便利服务，守护线程最典型的应用就是 GC (垃圾回收器)，它就是一个很称职的守护者。

User 和 Daemon 两者几乎没有区别，唯一的不同之处就在于虚拟机的离开：如果 User Thread 已经全部退出运行了，只剩下 Daemon Thread 存在了，虚拟机也就退出了。 因为没有了被守护者，Daemon 也就没有工作可做了，也就没有继续运行程序的必要了。

```
public class ThreadDaemonDemo {
```

```
    public static void main(String[] args) {
        Thread mainThread = new Thread(() -> {
            for (int i = 0; i < 100; i++) {
                System.out.println("线程 1 第" + i + "次执行！ ");
                try {
                    Thread.sleep(5);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
        });
        Thread daemonThread = new Thread(() -> {
            for (long i = 0; i < 10000; i++) {
                System.out.println("后台线程第" + i + "次执行！ ");
                try {
                    Thread.sleep(1);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
        });
        daemonThread.setDaemon(true); //设置为守护线程
        daemonThread.start();
        mainThread.start();
    }

}
```

中断线程

一个线程不应该由其他线程来强制中断或停止，而是应该由线程自己自行停止。
所以，Thread.stop, Thread.suspend, Thread.resume 都已经被废弃了。
而 Thread.interrupt 的作用其实也不是中断线程，而是「通知线程应该中断了」，具体到底中断还是继续运行，应该由被通知的线程自己处理。

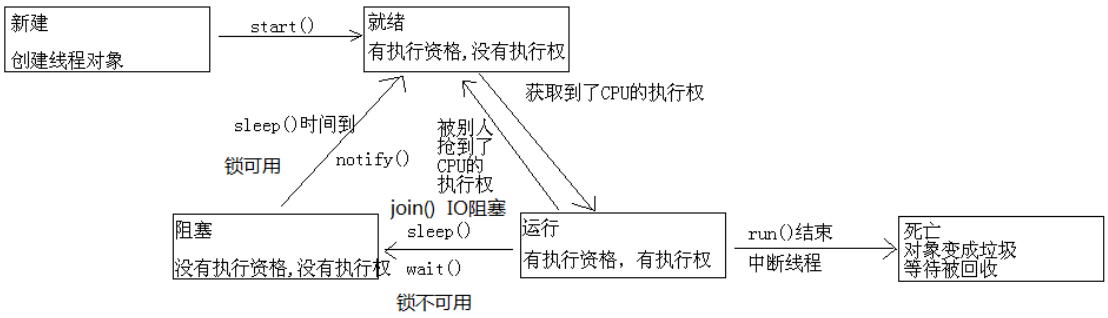
要使用中断，首先需要在可能会发生中断的线程中不断监听中断状态，一旦发生中断，就执行相应的中断处理代码。
当需要中断线程时，调用该线程对象的 interrupt 函数即可。

```
public class ThreadStopDemo {
    public static void main(String[] args) {
        Thread ts = new Thread(() -> {
            while (!Thread.currentThread().isInterrupted()) {
```

```
                System.out.println("睡觉中...");
            }
            System.out.println("我醒啦");
        });
        ts.start();

        try {
            Thread.sleep(1000);
            // ts.stop();
            System.out.println("别睡了，起来嗨");
            ts.interrupt();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}
```

线程的生命周期图



实现 Runnable 接口

```
public class MyRunnable implements Runnable {

    @Override
    public void run() {
        for (int x = 0; x < 100; x++) {
            // 由于实现接口的方式就不能直接使用 Thread 类的方法了,但是可以间接的使用
            System.out.println(Thread.currentThread().getName() + ":" + x);
        }
    }
```

```
}

/*
 * 方式 2：实现 Runnable 接口
 * 步骤：
 *      A:自定义类 MyRunnable 实现 Runnable 接口
 *      B:重写 run()方法
 *      C:创建 MyRunnable 类的对象
 *      D:创建 Thread 类的对象，并把 C 步骤的对象作为构造参数传递
 */
public class MyRunnableDemo {
    public static void main(String[] args) {
        // 创建 MyRunnable 类的对象
        MyRunnable my = new MyRunnable();

        // 创建 Thread 类的对象，并把 C 步骤的对象作为构造参数传递
        // Thread(Runnable target)
        // Thread t1 = new Thread(my);
        // Thread t2 = new Thread(my);
        // t1.setName("林青霞");
        // t2.setName("刘意");

        // Thread(Runnable target, String name)
        Thread t1 = new Thread(my, "林青霞");
        Thread t2 = new Thread(my, "刘意");

        t1.start();
        t2.start();
    }
}
```

使用 Runnable 实现线程的好处

可以避免由于 Java 单继承带来的局限性。

避免继承导致的无用的变量与方法

电影院卖票

问题

相同的票出现多次
    CPU 的一次操作必须是原子性的
还出现了负数的票
    随机性和延迟导致的

解决线程安全问题实现

```
class Sell implements Runnable {
    private int ticket = 100;//所有线程共享的票

    @Override
    public void run() {
        while (Thread.currentThread().isInterrupted()) {
            synchronized (Sell.class) {
                if (ticket > 0) {
                    try {
                        Thread.sleep(100);
                    } catch (InterruptedException e) {
                        e.printStackTrace();
                    }
                    System.out.println(Thread.currentThread().getName() +
ticket--);
                }
            }
        }
    }

}

public class SellTest {

    public static void main(String[] args) {
        Runnable r = new Sell();
        Thread t1 = new Thread(r, "1 号窗口");
        Thread t2 = new Thread(r, "2 号窗口");
        Thread t3 = new Thread(r, "3 号窗口");
```

```
        t1.start();
        t2.start();
        t3.start();
    }
}
```

JDK5 中 Lock 锁的使用

虽然我们可以理解同步代码块和同步方法的锁对象问题，但是我们并没有直接看到在哪里加上了锁，在哪里释放了锁，为了更清晰的表达如何加锁和释放锁，JDK5 以后提供了一个新的锁对象 Lock

```
Lock
    void lock()
    void unlock()

ReentrantLock

public class SellTicket implements Runnable {

    // 定义票
    private int tickets = 100;

    // 定义锁对象
    private Lock lock = new ReentrantLock();

    @Override
    public void run() {
        while (true) {
            try {
                // 加锁
                lock.lock();
                if (tickets > 0) {
                    try {
                        Thread.sleep(100);
                    } catch (InterruptedException e) {
                        e.printStackTrace();
                    }
                    System.out.println(Thread.currentThread().getName()
                            + "正在出售第" + (tickets--) + "张票");
                }
            } finally {
                // 释放锁
                lock.unlock();
```

```
            }
        }
    }

}
```

死锁问题

同步弊端

效率低

如果出现了同步嵌套，就容易产生死锁问题

死锁问题及其代码

是指两个或者两个以上的线程在执行的过程中，因争夺资源产生的一种互相等待现象

最简单的死锁例子

```
class Thread1 extends Thread {
    @Override
    public void run() {
        synchronized (Thread1.class) {
            System.out.println("1");
            synchronized (Thread2.class) {
                System.out.println("2");
            }
        }
    }
}

class Thread2 extends Thread {
    @Override
    public void run() {
        synchronized (Thread2.class) {
            System.out.println("3");
            synchronized (Thread1.class) {
                System.out.println("4");
            }
        }
    }
}
```

```
public class DeadLockTest {
	public static void main(String[] args) {
		for (int i = 0; i < 3; i++) {
			new Thread1().start();
			new Thread2().start();
		}
	}
}
```

发生死锁，或者不死锁全部输出。问题的原因就在于线程 1 拿到了线程 1 的 class 对象为锁对象，线程 2 拿到了线程 2 的 class 对象为锁对象，那么他们都无法继续执行，因为都需要对方的 class 对象为锁进入第二层 synchronized 块中。

线程间通信

针对同一个资源的操作有多个线程，需要使用到 wait 与 notify 方法进行线程间的通信

例子

通过设置线程(生产者)和获取线程(消费者)针对同一个学生对象进行操作。

目的：生产者产生一个学生，消费者才能去消费这个学生，不然等待。

```
public class Student {
    private String name;
    private int age;
    private boolean flag; // 默认情况是没有数据，如果是 true，说明有数据

    public synchronized void set(String name, int age) {
        // 如果有数据，就等待
        if (this.flag) {
            try {
                this.wait();
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }

        // 2.get 在等待设置数据
        this.name = name;
        this.age = age;
```

```
        //  修改标记
        this.flag = true;
        this.notify();
    }

    public synchronized void get() {
        // 1.如果没有数据，就开始等待
        if (!this.flag) {
            try {
                this.wait();
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }

        // 3.set 中 notify 后获取数据
        System.out.println(this.name + "---" + this.age);

        //  修改标记
        this.flag = false;
        this.notify();
    }
}

public class SetThread implements Runnable {

    private Student s;
    private int x = 0;

    public SetThread(Student s) {
        this.s = s;
    }

    @Override
    public void run() {
        while (true) {
            if (x % 2 == 0) {
                s.set("林青霞", 27);
            } else {
                s.set("刘意", 30);
            }
            x++;
        }
```

```
    }
}

public class GetThread implements Runnable {
    private Student s;

    public GetThread(Student s) {
        this.s = s;
    }

    @Override
    public void run() {
        while (true) {
            s.get();
        }
    }
}
public class StudentDemo {
    public static void main(String[] args) {
        //创建资源
        Student s = new Student();

        //设置和获取的类
        SetThread st = new SetThread(s);
        GetThread gt = new GetThread(s);

        //线程类
        Thread t1 = new Thread(st);
        Thread t2 = new Thread(gt);

        //启动线程
        t1.start();
        t2.start();
    }
}
```

线程组

Java 中使用 ThreadGroup 来表示线程组，它可以对一批线程进行分类管理，Java 允许程序直接对线程组进行控制。

默认情况下，所有的线程都属于主线程组。
public final ThreadGroup getThreadGroup()

我们也可以给线程设置分组
Thread(ThreadGroup group, Runnable target, String name)

```
package cn.itcast_06;

/*
 * 线程组： 把多个线程组合到一起。
 * 它可以对一批线程进行分类管理，Java 允许程序直接对线程组进行控制。
 */
public class ThreadGroupDemo {
    public static void main(String[] args) {
        // method1();

        // 我们如何修改线程所在的组呢?
        // 创建一个线程组
        // 创建其他线程的时候，把其他线程的组指定为我们自己新建线程组
        method2();

        // t1.start();
        // t2.start();
    }

    private static void method2() {
        // ThreadGroup(String name)
        ThreadGroup tg = new ThreadGroup("这是一个新的组");

        MyRunnable my = new MyRunnable();
        // Thread(ThreadGroup group, Runnable target, String name)
        Thread t1 = new Thread(tg, my, "林青霞");
        Thread t2 = new Thread(tg, my, "刘意");

        System.out.println(t1.getThreadGroup().getName());
        System.out.println(t2.getThreadGroup().getName());

        //通过组名称设置后台线程，表示该组的线程都是后台线程
        tg.setDaemon(true);
    }

    private static void method1() {
        MyRunnable my = new MyRunnable();
        Thread t1 = new Thread(my, "林青霞");
        Thread t2 = new Thread(my, "刘意");
        // 我不知道他们属于那个线程组,我想知道，怎么办
        // 线程类里面的方法：public final ThreadGroup getThreadGroup()
```

```
        ThreadGroup tg1 = t1.getThreadGroup();
        ThreadGroup tg2 = t2.getThreadGroup();
        // 线程组里面的方法：public final String getName()
        String name1 = tg1.getName();
        String name2 = tg2.getName();
        System.out.println(name1);
        System.out.println(name2);
        // 通过结果我们知道了：线程默认情况下属于 main 线程组
        // 通过下面的测试，你应该能够看到，默任情况下，所有的线程都属于同一个组
        System.out.println(Thread.currentThread().getThreadGroup().getName());
    }
}
```

线程池

程序启动一个新线程成本是比较高的，因为它涉及到要与操作系统进行交互。而使用线程池可以很好的提高性能，尤其是当程序中要创建大量生存期很短的线程时，更应该考虑使用线程池。

线程池里的每一个线程代码结束后，并不会死亡，而是再次回到线程池中成为空闲状态，等待下一个对象来使用。

在 JDK5 之前，我们必须手动实现自己的线程池，从 JDK5 开始，Java 内置支持线程池

JDK5 新增了一个 Executors 工厂类来产生线程池，有如下几个方法

```
public static ExecutorService newCachedThreadPool()
public static ExecutorService newFixedThreadPool(int nThreads)
public static ExecutorService newSingleThreadExecutor()
```

这些方法的返回值是 ExecutorService 对象，该对象表示一个线程池，可以执行 Runnable 对象或者 Callable 对象代表的线程。它提供了如下方法

```
Future<?> submit(Runnable task)
<T> Future<T> submit(Callable<T> task)
```

```
package cn.itcast_08;

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

/*
 * 线程池的好处：线程池里的每一个线程代码结束后，并不会死亡，而是再次
```

```
回到线程池中成为空闲状态，等待下一个对象来使用。
 *
 * 如何实现线程的代码呢?
 *      A:创建一个线程池对象，控制要创建几个线程对象。
 *          public static ExecutorService newFixedThreadPool(int nThreads)
 *      B:这种线程池的线程可以执行：
 *          可以执行 Runnable 对象或者 Callable 对象代表的线程
 *          做一个类实现 Runnable 接口。
 *      C:调用如下方法即可
 *          Future<?> submit(Runnable task)
 *          <T> Future<T> submit(Callable<T> task)
 *      D:我就要结束，可以吗?
 *          可以。
 */
public class ExecutorsDemo {
    public static void main(String[] args) {
        // 创建一个线程池对象，控制要创建几个线程对象。
        // public static ExecutorService newFixedThreadPool(int nThreads)
        ExecutorService pool = Executors.newFixedThreadPool(2);

        // 可以执行 Runnable 对象或者 Callable 对象代表的线程
        pool.submit(new MyRunnable());
        pool.submit(new MyRunnable());

        //结束线程池
        pool.shutdown();
    }
}
```

定时器

定时器是一个应用十分广泛的线程工具，可用于调度多个定时任务以后台线程的方式执行。在 Java 中，可以通过 Timer 和 TimerTask 类来实现定义调度的功能

```
package cn.itcast_12;

import java.util.Timer;
import java.util.TimerTask;

/*
 * 定时器：可以让我们在指定的时间做某件事情，还可以重复的做某件事情。
 * 依赖 Timer 和 TimerTask 这两个类：
```

```
 * Timer:定时
 *        public Timer()
 *        public void schedule(TimerTask task,long delay)
 *        public void schedule(TimerTask task,long delay,long period)
 *        public void cancel()
 * TimerTask:任务
 */
public class TimerDemo2 {
    public static void main(String[] args) {
        // 创建定时器对象
        Timer t = new Timer();
        // 3 秒后执行爆炸任务第一次，如果不成功，每隔 2 秒再继续炸
        t.schedule(new MyTask2(), 3000, 2000);
    }
}

// 做一个任务
class MyTask2 extends TimerTask {
    @Override
    public void run() {
        System.out.println("beng,爆炸了");
    }
}
```

多线程面试题

1:多线程有几种实现方案，分别是哪几种?
两种。

继承 Thread 类
实现 Runnable 接口

扩展一种：实现 Callable 接口。这个得和线程池结合。

2:同步有几种方式，分别是什么?
两种。

同步代码块
同步方法

3:启动一个线程是 run()还是 start()?它们的区别?
start();

run():封装了被线程执行的代码,直接调用仅仅是普通方法的调用
start():启动线程，并由 JVM 自动调用 run()方法

4:sleep()和 wait()方法的区别
sleep():必须指时间;不释放锁。
wait():可以不指定时间，也可以指定时间;释放锁。

5:为什么 wait(),notify(),notifyAll()等方法都定义在 Object 类中
因为这些方法的调用是依赖于锁对象的，而同步代码块的锁对象是任意锁。
而 Object 代码任意的对象，所以，定义在这里面。

6:线程的生命周期图
新建 -- 就绪 -- 运行 -- 死亡
新建 -- 就绪 -- 运行 -- 阻塞 -- 就绪 -- 运行 -- 死亡
建议：画图解释。