ThreadDemo1

```
package thread;

/**
 * 多线程：
 * 多线程可以并发执行多个代码片段
 *
 * 线程的第一种创建方式：
 * 继承Thread类重写run()方法
 */
public class ThreadDemo1 {
    public static void main(String[] args) {
        Thread t1 = new MyThread1();
        Thread t2 = new MyThread2();

        //线程运行需要调用start()方法，而不是直接调用run()方法！
        //当start()方法执行完毕，线程的run()方法很快会被自动执行。
        t1.start();
        t2.start();
    }
}

/*
    该种方式创建线程的优点：
```

```
    结构简单，利于匿名内部类形式的创建。

    缺点：
    1：存在继承冲突问题，因为java是单继承，所以继承了Thread类就无法再继承其它类去复用方法。
    2：定义线程的同时就将任务定义在里面（重写了run()方法），这导致线程与任务存在了必然的紧密耦合关系，不利于线程的复用。
 */
class MyThread1 extends Thread {
    public void run() {
        for (int i = 0; i < 1000; i++) {
            System.out.println("你是谁啊？");
        }
    }
}

class MyThread2 extends Thread {
    public void run() {
        for (int i = 0; i < 1000; i++) {
            System.out.println("开门！查水表的！");
        }
    }
}
```

ThreadDemo2

```
package thread;

/**
 * 线程的第二种创建方式:
 * 实现Runnable接口来单独定义线程任务
 */
public class ThreadDemo2 {
    public static void main(String[] args) {
        //创建线程的任务run()方法
        Runnable r1 = new MyRunnable1(); //向上造型
        Runnable r2 = new MyRunnable2();

        //创建线程
        Thread t1 = new Thread(r1);
        Thread t2 = new Thread(r2);

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

class MyRunnable1 implements Runnable {
    public void run() {
        for (int i = 0; i < 1000; i++) {
```

```
            System.out.println("你是谁啊？");
        }
    }
}

class MyRunnable2 implements Runnable {
    public void run() {
        for (int i = 0; i < 1000; i++) {
            System.out.println("我是查水表的！");
        }
    }
}
```

ThreadDemo3

```java
package thread;

/**
 * 线程的第三种创建方式：
 * 使用匿名内部类结合上述两种形式或者lambda表达式完成线程的创建
 */
public class ThreadDemo3 {
    public static void main(String[] args) {
        //使用匿名内部类，继承Thread类重写run()方法
        Thread t1 = new Thread() {
            public void run() {
                for (int i = 0; i < 1000; i++) {
                    System.out.println("你是谁啊？");
                }
            }
        };

        //使用匿名内部类，实现Runnable接口来单独定义线程任务
        Runnable r2 = new Runnable() {
            public void run() {
                for (int i = 0; i < 1000; i++) {
                    System.out.println("我是查水表的！");
                }
            }
```

```
        };
        Thread t2 = new Thread(r2);

        //lambda表达式
        Thread t3 = new Thread(() -> { //没参数也要写括号
            for (int i = 0; i < 1000; i++) {
                System.out.println("你是谁啊？");
            }
        });

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

CurrentThreadDemo

```java
package thread;

/**
 * java中所有代码都是靠线程运行的，main()方法也不例外。
 * 当JVM启动后就会创建一条线程来执行main()方法，并且该线程被赋予了一个名字也叫"main"，
 * 我们一般称它为“主线程”，但是它和我们创建的线程本身没有任何区别。
 *
 * 线程提供了一个静态方法：
 * static Thread currentThread()
 * 该方法可以获取运行该方法的线程
 */
public class CurrentThreadDemo {
    public static void main(String[] args) {
        Thread main = Thread.currentThread(); //获取运行main()方法的线程
        System.out.println("主线程：" + main);
        dosome();
        //main()方法执行完毕意味着主线程结束了
    }

    public static void dosome() {
        Thread t = Thread.currentThread(); //获取运行dosome()方法的线程
        System.out.println("运行dosome()方法的线程是：" + t);
    }
}
```

ThreadInfoDemo

```java
package thread;

//查看线程相关信息的一组方法
public class ThreadInfoDemo {
    public static void main(String[] args) {
        //获取主线程
        Thread main = Thread.currentThread();

        //获取线程的名字
        String name = main.getName();
        System.out.println(name);

        //获取线程的唯一标识
        long id = main.getId(); //注意是long类型!
        System.out.println(id);

        //获取线程的优先级
        //是一个数字，范围在1-10之间
        //10最高优先级，1最低优先级，5普通优先级（默认）
        int priority = main.getPriority();
        System.out.println("优先级: " + priority);

        //注意：is开头的方法返回值为boolean!
```

```
        //查看是否为守护线程
        boolean daemon = main.isDaemon(); //daemon：守护线程

        //查看是否活着
        boolean alive = main.isAlive();

        //查看是否被中断了
        boolean interrupted = main.isInterrupted(); //interrupt：中断

        System.out.println("是否为守护线程：" + daemon);
        System.out.println("是否活着：" + alive);
        System.out.println("是否被中断：" + interrupted);
    }
}
```

PriorityDemo

```java
package thread;

/**
 * 线程的优先级:
 * 当一个线程调用start()方法后，就被纳入到线程调度器中被统一管理，
 * 此时线程只能被动地被分配时间片并发运行，并不能主动索取时间片。
 * 而线程调度器会尽可能均匀地分配时间片给每个线程。
 * 通过调整线程的优先级可以最大程度地改善获取时间片的概率。
 * 优先级越高的线程获取时间片的次数越多。
 *
 * 线程的优先级有10个等级，分别用整数1-10表示。其中1最低，5为默认值，10最高。
 * 也可以使用Thread提供的常量：MIN_PRIORITY，NORM_PRIORITY，MAX_PRIORITY。
 * 分别表示：最低优先级，默认优先级，最高优先级
 */
public class PriorityDemo {
    public static void main(String[] args) {
        Thread min = new Thread() {
            public void run() {
                for (int i = 0; i < 10000; i++) {
                    System.out.println("min");
                }
            }
        };
```

```
        Thread max = new Thread() {
            public void run() {
                for (int i = 0; i < 10000; i++) {
                    System.out.println("max");
                }
            }
        };

        Thread normal = new Thread() {
            public void run() {
                for (int i = 0; i < 10000; i++) {
                    System.out.println("normal");
                }
            }
        };

        min.setPriority(Thread.MIN_PRIORITY);
        max.setPriority(Thread.MAX_PRIORITY);
        //normal.setPriority(Thread.NORM_PRIORITY); //默认的，不用设置

        min.start();
        max.start();
        normal.start();
    }
}
```

DaemonThreadDemo

```
package thread;

/**
 * 守护线程:
 * 也称为“后台线程”，是通过普通线程（也称为用户线程）调用方法setDaemon(true)设置而转变的。
 * 守护线程在结束时机上与普通线程有一点不同，那就是在进程结束时。
 *
 * 进程结束:
 * 当java进程中所有普通线程都结束时，该进程就会结束，此时它会杀死所有还在运行的守护线程。
 */
public class DaemonThreadDemo {
    public static void main(String[] args) {
        Thread rose = new Thread() {
            public void run() {
                for (int i = 0; i < 5; i++) {
                    System.out.println("Rose: Let me go!");
                    try {
                        Thread.sleep(1000);
                    } catch (InterruptedException e) {
                        e.printStackTrace();
                    }
                }
                System.out.println("哼啊啊啊啊啊！ ");
                System.out.println("噗通！ ");
```

```
            }
        };

        Thread jack = new Thread() {
            public void run() {
                while (true) {
                    System.out.println("Jack: You jump! I jump!");
                    try {
                        Thread.sleep(1000);
                    } catch (InterruptedException e) {
                        e.printStackTrace();
                    }
                }
            }
        };

        //设置守护线程的工作（setDaemon()）必须在线程启动前（start()）进行
        //适合设置为守护进程的：别人停我也停，如GC（垃圾回收器）
        jack.setDaemon(true);

        rose.start();
        jack.start();

        //System.out.println("main方法执行完毕");
        while (true) ;
    }
}
```

SleepDemo

```java
package thread;

import java.util.Scanner;

/**
 * sleep阻塞:
 *
 * 线程提供了一个静态方法:
 * static void sleep(long ms)
 * 当一个线程调用sleep()方法后就会进入指定毫秒数的阻塞状态
 */
public class SleepDemo {
    public static void main(String[] args) {
        //倒计时程序:
        //程序启动后要求输入一个数字，从该数字开始每秒递减，到0时输出时间到。

        System.out.println("程序开始了");

        Scanner scanner = new Scanner(System.in);
        System.out.println("请输入一个数字: ");
        int num = scanner.nextInt();
        for (int i = num; i > 0; i--) {
            try {
                Thread.sleep(1000); //java没办法精准控制时间，误差比较大
```

```
            } catch (InterruptedException e) {
                throw new RuntimeException(e);
            }
        }
        System.out.println("时间到");

        System.out.println("程序结束了");
    }
}
```

SleepDemo2

```
package thread;

/**
 * sleep方法要求我们必须处理中断异常：InterruptedException
 * 当一个线程调用sleep()方法处于睡眠阻塞的过程时，若该线程的interrupt()方法被调用，
 * 那么此时会强制中断它的睡眠阻塞并抛出异常InterruptedException。
 */
public class SleepDemo2 {
    public static void main(String[] args) {
        Thread lin = new Thread() {
            public void run() {
                System.out.println("sleep~");
                try {
                    Thread.sleep(500000000);
                } catch (InterruptedException e) {
                    //强制唤醒后的代码放这里
                    System.out.println("what?");
                }
                System.out.println("wake!");
            }
        };

        Thread huang = new Thread() {
            public void run() {
```

```
                System.out.println("big 80,small 40,begin!");
                for (int i = 0; i < 5; i++) {
                    System.out.println("80!");
                    try {
                        Thread.sleep(1000);
                    } catch (InterruptedException e) {
                        //不能直接throw抛异常，因为这是额外异常，Runnable没有
                    }
                }
                System.out.println("down!");
                System.out.println("bro!ok!");
                lin.interrupt(); //强行打断lin的sleep
            }
        };

        lin.start();
        huang.start();
    }
}
```

Test2

```
package thread;

/**
 * 实验：
 * 在main()方法中定义两个线程，各自循环10000次，每次循环时对num++。
 * 当两个线程执行完毕后，查看num的值是多少？
 */
public class Test2 {
    public static int num = 0;

    public static void main(String[] args) {
        Thread t1 = new Thread() {
            public void run() {
                for (int i = 0; i < 10000; i++) {
                    num++;
                }
                System.out.println("线程结束了！");
            }
        };

        Thread t2 = new Thread() {
            public void run() {
                for (int i = 0; i < 10000; i++) {
                    num++;
```

```
                }
                System.out.println("线程结束了！");
            }
        };

        t1.start();
        t2.start();

        try {
            Thread.sleep(2000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        System.out.println(num);
    }
}
```

SyncDemo1

```
package thread;

/**
 * 多线程并发安全问题：
 * 当多个线程并发操作同一临界资源，由于线程切换实际不确定，导致操作顺序出现混乱，从而导致程序bug或更严重的程序问题。
 * 通俗的讲就是当多个线程同时“抢”同一个东西产生了混乱。
 * <p>
 * 临界资源：同一时刻只能被单线程操作的资源称为临界资源。
 */
public class SyncDemo1 { //Sync：同步
    public static void main(String[] args) {
        Table table = new Table();
        Thread t1 = new Thread() {
            public void run() {
                while (true) {
                    int bean = table.getBean();
                    Thread.yield();
                    System.out.println(getName() + ":" + bean);
                }
            }
        };
        Thread t2 = new Thread() {
            public void run() {
```

```
                while (true) {
                    int bean = table.getBean();
                    Thread.yield();
                    System.out.println(getName() + ":" + bean);
                }
            }
        };
        t1.start();
        t2.start();
    }
}

class Table {
    private int beans = 20; //桌子上有20个豆子

    /*
        当一个方法使用synchronized修饰后，该方法称为同步方法。即：多个线程不能同时在内部一起执行
        将多个线程并发操作同一临界资源改为同步执行（排队，有先后顺序的）就可以有效的解决多线程的并发安全问题。
     */
    public synchronized int getBean() { //synchronized：同步的
        //异步
        //线程1执行完了，外面等的线程2、3、4、5谁先进，不是按顺序来，而是看下次时间片先分配给谁
        if (beans == 0) { //桌子上没有豆子了
            throw new RuntimeException("没豆子了！");
        }
        Thread.yield(); //让线程主动放弃当前时间片
        return beans--;
    }
```

```
}

/*
Thread-0:20
Thread-1:19
Thread-0:18
Thread-1:17
Thread-0:16
Thread-1:15
Thread-0:14
Thread-1:13
Thread-0:12
Thread-1:11
Thread-0:10
Thread-1:9
Thread-0:8
Thread-1:7
Thread-0:6
Thread-1:5
Thread-0:4
Thread-1:3
Thread-1:1
Thread-0:2
Exception in thread "Thread-0" Exception in thread "Thread-1" java.lang.RuntimeException: 没豆子了!
    at thread.Table.getBean(SyncDemo1.java:45)
    at thread.SyncDemo1$2.run(SyncDemo1.java:25)
java.lang.RuntimeException: 没豆子了!
```

```
    at thread.Table.getBean(SyncDemo1.java:45)
    at thread.SyncDemo1$1.run(SyncDemo1.java:16)
*/
```

SyncDemo2

```
package thread;

/**
 * 同步块的使用:
 * 有效的缩小同步范围可以在保证并发安全的前提下尽可能的提高并发效率。
 * <p>
 * 语法:
 * synchronized(同步监视器对象){
 * 需要多个线程同步执行的代码片段
 * }
 * <p>
 * 同步和异步执行：这两种执行都是指多线程的执行。
 * 同步执行：多个线程的执行过程出现了先后顺序。
 * 异步执行：多个线程各执行各的，之间没有顺序。
 */
public class SyncDemo2 {
    public static void main(String[] args) {
        //"hello"（字符串字面量）作为锁对象的例子
        //Shop shop1 = new Shop();
        //Shop shop2 = new Shop();

        Shop shop = new Shop();
        Thread t1 = new Thread("范传奇") { //给线程起名字，getName()可以获取名字
            public void run() {
```

```
                //shop1.buy();
                shop.buy();
            }
        };
        Thread t2 = new Thread("王克晶") {
            public void run() {
                //shop2.buy();
                shop.buy();
            }
        };
        t1.start();
        t2.start();
    }
}

class Shop {
    //在成员方法上使用synchronized时，同步监视器对象就是当前方法的所属对象this。
    //public synchronized void buy(){
    public void buy() {
        try {
            Thread t = Thread.currentThread(); //获取运行buy方法的线程
            System.out.println(t.getName() + ":正在挑衣服...");
            Thread.sleep(2000);
            /*
                在使用同步块时要指定同步监视器对象(上锁的对象)
                该对象必须同时满足以下两个条件：
                1：必须是一个引用类型
                2：多个需要同步执行该代码片段的线程看到的这个锁对象必须为【同一个】！
```

```
         */

        synchronized (this) {
            //可以，因为t1和t2调用的是同一个shop的buy方法，因此方法内的this是同一个shop对象
            //作用的对象是调用这个代码块的对象，谁调就是谁
            //Shop对象调的buy()方法，所以this指的是Shop对象

            //synchronized (t) {
            //不可以，因为t1和t2调用buy方法时获取的就是线程自身，因此t是不同的线程。
            //如果这里写t，编译也能通过，但实际运行没实现同步的效果
            //因为t是buy()方法定义的当前线程，可能是t1线程也可能是t2线程，
            //相当于t1进来了锁自己，t2进来了锁自己，没实现同步的效果

            //synchronized (new Object()) {
            //不可以，有new一定不行！因为new一定产生新对象

            //synchronized ("hello") {
            /*
                当"hello"（字符串字面量）作为锁对象时，由于java对字符串的优化机制，所以字面量始终是同一个对象，

                这会导致当t1和t2线程调用不同shop的buy方法时（相当于两个人进不同的商店买衣服），
                由于在这里看到的锁对象相同，所以仍然需要排队执行，这就是不合理的操作了。
                排队执行的前提是存在并发安全问题（有"抢"这件事发生）。
             */

            System.out.println(t.getName() + ":正在试衣服...");
            Thread.sleep(2000);
        }
```

```
                System.out.println(t.getName() + ":结账离开");
            } catch (InterruptedException e) {

            }
        }
    }
```

SyncDemo3

```
package thread;

//若在静态方法上使用synchronized，则该方法一定具有同步效果，一定是分开做。
public class SyncDemo3 {
    public static void main(String[] args) {
        Boo b1 = new Boo();
        Boo b2 = new Boo();
        Thread t1 = new Thread() {
            public void run() {
                b1.dosome();
            }
        };
        Thread t2 = new Thread() {
            public void run() {
                b2.dosome();
            }
        };
        t1.start();
        t2.start();
    }
}

class Boo {
    /*
```

```
        静态方法上使用synchronized后，那么同步监视器对象为当前类的类对象。即：Class的实例。
        知识引入：JVM中每一个被加载的类都有且只有一个Class的实例与之对应，该实例称为这个类的类对象。
                获取类对象的方式：类名.class
        该知识点会在后期反射中详细说明
     */
    //public static synchronized void dosome(){
    public static void dosome() {
        //静态方法的同步块中依然应当使用当前类的类对象作为锁对象
        synchronized (Boo.class) {
            try {
                Thread t = Thread.currentThread();
                System.out.println(t.getName() + ":正在执行dosome方法...");
                Thread.sleep(2000);
                System.out.println(t.getName() + ":执行dosome方法完毕!");
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
}
```

SyncDemo4

```
package thread;

/**
 * 互斥锁：
 * 当使用多个synchronized锁定多个代码片段，但是指定的锁对象相同时，这些代码片段就是互斥的
 * 即：多个线程不能同时执行这些代码片段
 */
public class SyncDemo4 {
    public static void main(String[] args) {
        Foo foo = new Foo();
        Thread t1 = new Thread() {
            public void run() {
                foo.methodA();
            }
        };
        Thread t2 = new Thread() {
            public void run() {
                foo.methodB();
            }
        };
        t1.start();
        t2.start();
    }
}
```

```
class Foo {
    //在成员方法上使用synchronized时，同步监视器对象就是当前方法的所属对象this。
    public synchronized void methodA() {
        try {
            Thread t = Thread.currentThread();
            System.out.println(t.getName() + ":正在执行A方法...");
            Thread.sleep(2000);
            System.out.println(t.getName() + ":执行A方法完毕!");
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }

    public void methodB() {
        synchronized (this) {
            try {
                Thread t = Thread.currentThread();
                System.out.println(t.getName() + ":正在执行B方法...");
                Thread.sleep(2000);
                System.out.println(t.getName() + ":执行B方法完毕!");
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
}
```

ThreadPoolDemo

```java
package thread;

import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors; //concurrent：并发

/*
线程池：
线程池是线程的管理机制，主要解决两方面问题：
1、控制线程的数量
2、重用线程
 */
public class ThreadPoolDemo {
    public static void main(String[] args) {
        ExecutorService threadPool = Executors.newFixedThreadPool(2); //线程数量

        for (int i = 0; i < 5; i++) {
            /*
            以下三行：
            Runnable r = new Runnable() {
                @Override
                public void run() {
            可替换为：
            Runnable r = () -> {
             */
```

```
            Runnable r = new Runnable() {
                @Override
                public void run() {
                    Thread t = Thread.currentThread(); //获取当前线程
                    System.out.println(t.getName() + "正在执行任务。。。");
                    try {
                        Thread.sleep(2000);
                    } catch (InterruptedException e) {
                        //当阻塞的方法收到中断请求时会抛出InterruptedException异常
                        e.printStackTrace();
                    }
                    System.out.println(t.getName() + "任务执行完毕！");
                }
            };
            threadPool.execute(r); //将任务指派给线程池
        }

        threadPool.shutdown(); //关闭线程池，但是会把正在执行的线程执行完
        threadPool.shutdownNow(); //关闭线程池，立刻停止所有线程
        System.out.println("线程池停止了！");
    }
}
```