Java多线程

> 参考视频：https://www.bilibili.com/video/BV1V4411p7EF/?spm_id_from=333.337.search-card.all.click&vd_source=73f685f6515c8bcbc3bff2f9e7fd032d
>
> 编写：Sinocifeng
>
> About Me：点击进入我的Personal Page

概述

Java.Thread类(thread中文指"线、毛线")

平时联机游戏，王者荣耀，为什么可以几个人一起互相独立地操作又时刻获取其他人的状态？聊天时候为什么我给你发的时候你也可以给我发？这就需要考虑多线程。

后续内容组织如下：

- 对线程、进程、多线程进行简单介绍
- 实现一个线程（*）
- 线程的状态介绍（人有生老病死，线程也有类似的一些状态）
- 线程同步（*）——12306买火车票，一张票被a买走就不能再被b买走。
- 线程通信
- 其他

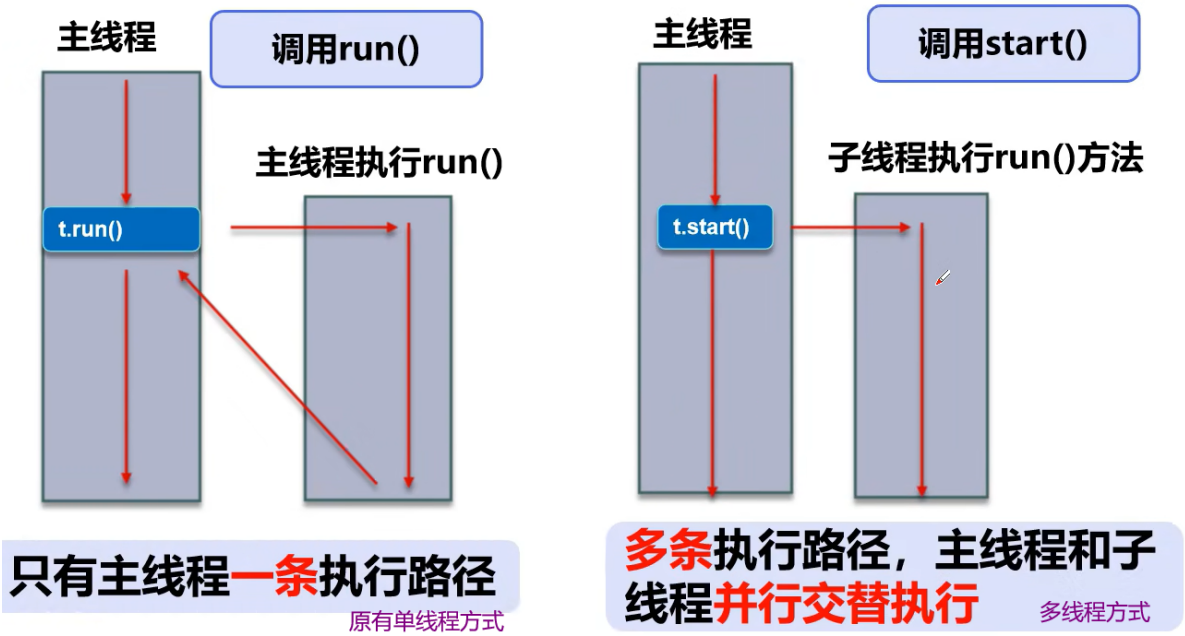
线程、进程、多进程

多任务：

现实中同时做多件事，看似多任务，但本质上大脑在同一时间只做了一件事。

多线程

由于车辆多而堵塞，为了充分使用道路，划分多个车道，解决一条道路的堵塞问题。



进程

在OS中，运行的程序就是进程，例如QQ、Wechat、IDE、game等

一个进程可以有多个线程，例如我们通过微信视频聊天不仅能看到图像，还能听到声音等。

程序：是指令和数据的有序集合，其本身没有运行含义，仅是一个静态概念。

进程：是程序的一次执行，是一个动态的过程。进程是系统进行资源分配的最小单位。

线程：通常一个进程可以包含多个线程，线程是CPU进行调度和执行的单位。

总结

- 线程是CPU独立的执行路径
- 程序运行时，即使没有人为创建线程，后台也存在线程，例如主线程、GC线程
- main()称为主线程，是系统的入口。
- 在一个进程中，如果开辟多个线程，线程调度由调度器安排调度。调度器与操作系统紧密相关，先后顺序无法人为干预
- 对于同一资源，可能存在多个线程对其进行资源争夺，需要进行并发控制
- 线程会带来额外开销，如CPU调度时间，并发控制开销等
- 每个线程在自己的工作内存区域进行交互，内存控制不当会造成数据不一致

线程创建(*)

> Thread、Runnable、Callable

实现方式

- Thread Class——继承Thread类
- Runnable 接口——实现Runnable接口（实际上Thread就是实现了Runnable接口）
- Callable接口——工作之后进阶使用（了解即可）

Thread类

1. 自定义线程类继承Thread类
2. 重写run()方法，编写线程执行体
3. 创建线程对象，调用start()方法启动线程

```java
public class startThread extends Thread{
    @override
    public void run(){
        functions....
    }
}
public static void main(String[] args){
    startThread t = new startThread();
    t.start();
}
```

注意：线程开启不一定立即执行，由CPU调度执行

不推荐使用，因为OOP单继承具有局限性

Runnable 接口

1. 定义MyRunnable类实现Runnable 接口
2. 实现run()方法。编写线程执行体
3. 创建线程对象，调用start()方法启动线程

```java
public class startThread implements Runnable{
    @Override
    public void run(){
        functions....
    }
}
public static void main(String[] args){
    startThread t = new startThread();
    Thread thread = new Thread(t);
    thread.start();
}
```

推荐使用Runnable对象，因为java单继承的局限性（可以实现多个接口），便于一个对象被多个线程使用。

```java
//一份资源
StartThread station = new  StartThread();
//多个代理
new Thread(station,name:"张三").start();
new Thread(station,name:"李四").start();
new Thread(station,name:"王五").start();
```

多个线程同时操作同一个对象

```java
public class TestThread implements Runnable{

    private int tickeNums = 10;          // 车票数

    @Override
    public void run() {
        while (true) {
            if (tickeNums <= 0){
                break;
            }
            Thread.sleep(200);           //延时200ms
```

```
            System.out.println(Thread.currentThread().getName()+"买到了第"+tickeNums--+"张票")
        }
    }

    public static void main(String[] args){
        TestThread testThread = new TestThread();
        //name为线程名
        new Thread(testThread,name:"张三").start();
        new Thread(testThread,name:"李四").start();
        new Thread(testThread,name:"王五").start();
    }
}
```

结果：发现问题-多线程操作同一线程情况下，数据紊乱，有不同的人拿到同一张票。

Callable接口（了解即可）

1. 实现Callable接口，需要返回值类型
2. 重写call方法，需要抛出异常
3. 创建目标对象
4. 创建执行服务：`ExecutorService ser = Executors.newFixedThreadPool(1);`
5. 提交执行：`Future<Boolean> result = ser.submit(t1);`
6. 获取结果：`boolean r = result.get();`
7. 关闭服务：`ser.shutdownNow();`

```
public class TestCallable implements Callable{
    private String url;     //网络图片地址
    private String name;    //文件名

    public TestCallable(String url, String name){
        this.url = url;
        this.name = name;
    }

    //下载图片线程的执行体
    public Boolean call(){
        WebDownloader webDownloader = new WebDownloader();
        webDownloader.downloader(url, name);
        System.out.println("所下载文件为"+name)
        return true;
    }

    public static void main(String[] args){
        TestCallable t1 = new TestCallable("url1", "name1");
        TestCallable t2 = new TestCallable("url2", "name2");
        TestCallable t3 = new TestCallable("url3", "name3");
        //创建执行服务，生成3个线程
        ExecutorService ser = Executors.newFixedThreadPool(3);
        //提交执行
        Future<Boolean> r1 = ser.submit(t1);
        Future<Boolean> r2 = ser.submit(t2);
        Future<Boolean> r3 = ser.submit(t2);
        //获取结果
        boolean rs1 = r1.get();
        boolean rs2 = r2.get();
        boolean rs3 = r3.get();
```

```
        //关闭服务
        ser.shutdownNow();
    }
}
```

Callable接口的好处：1.可以定义返回值 2.可以抛出异常

静态代理

- 真实对象和代理对象都要实现同一接口
- 代理对象要代理真实角色

```
public class StaticPorxy {
    public static void main(String[] args){
        People people = new People();
        WeddingCompany company = new WeddingCompany(people);
        company.HappyMarry();
    }
}

interface Marry{
    void HappyMarry();
}

class People implements Marry{
    @Override
    public void HappyMarry(){
        System.out.println("Marry! ");
    }
}

class WeddingCompany implements Marry{
    private Marry customer;
    public WeddingCompany(Marry customer){
        this.customer = customer;
    }
    @Override
    public void HappyMarry(){
        before();
        this.customer.HappyMarry();
        after();
    }
    private void before(){略...}
    private void after(){略...}
}
```


静态代理的好处：

- 代理对象可以做很多真实对象无法做的事情，真实对象只需要专注于自己的事情

Thread实现线程本质上就是实现了Runnable接口，进行了静态代理。

Lambda表达式

λ表达式属于函数式编程，可以有效简化代码，可以避免匿名内部类定义过多。

> 为什么要使用Lambda表达式?

> - 避免匿名内部类定义过多
>
>> 我们写的一些类可能只用一两次，这太浪费了。为了简化，我们可以将其定义为内部类，在使用的时候再去定义这个类；而取这些类名很麻烦，因此有引入了匿名内部类；为了避免匿名内部类定义过多，又提出了Lambda表达式。
>
> - 可以让代码看上去比较简洁
> - 去掉了一堆没有意义的代码，只留下核心的逻辑

Functional Interface

理解Functional Interface(函数式接口)是学习Java 8中Lambda表达式的关键

- 任何接口，如果只包含一个抽象方法，那么其就是一个函数式接口

```
public interface Runnable{
    public abstract void run();
}
```

- 对于函数式接口，可以通过lambda表达式来创建该接口的对象

理解如何一步步提出Lambda表达式的

```
public class TestLambda{

    //3.优化一：静态内部类
    static class Like2 implements Ilike{
    @Override
    public void lambda(){
        System.out.println("I like Lambda 2");
    }
}

    public static void main(String[] args){
        Ilike like = new Like1();
        like.lambda();

        like = new Like2();
        like.lambda();

        //4.优化二：局部内部类
        class Like3 implements Ilike{
            @Override
            public void lambda(){
                System.out.println("I like Lambda 3");
            }
        }

        like = new Like3();
        like.lambda();

        //5.优化三：匿名内部类：没有类名称，必须借助接口或父类
        like = new ILike(){
            @Override
            public void lambda(){
                System.out.println("I like Lambda 4");
            }
```

```
        };
        like.lambda();

        //6.优化四：lambda表达式
        like = (/**这里放入参数*/) -> {
            System.out.println("I like Lambda 5");
        };
        like.lambda();
    }
}

//1.定义一个函数式接口
interface Ilike{
    void lambda();
}

//2.实现类
class Like1 implements Ilike{
    @Override
    public void lambda(){
        System.out.println("I like Lambda 1");
    }
}
```

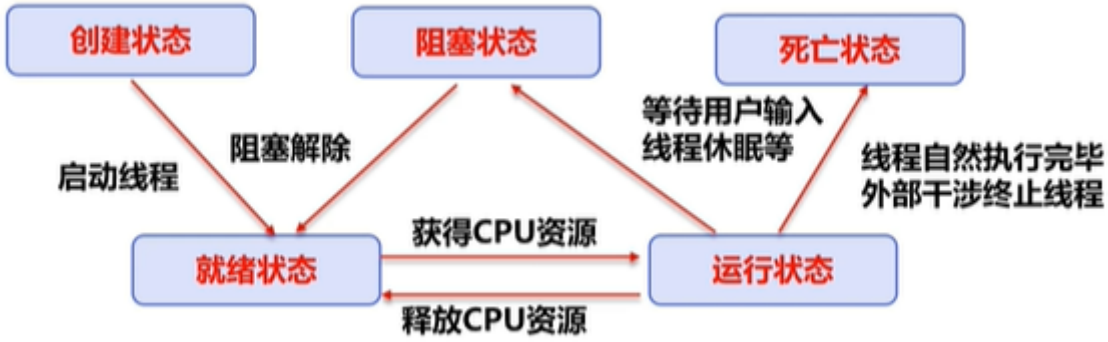
Lambda表达式简化过程

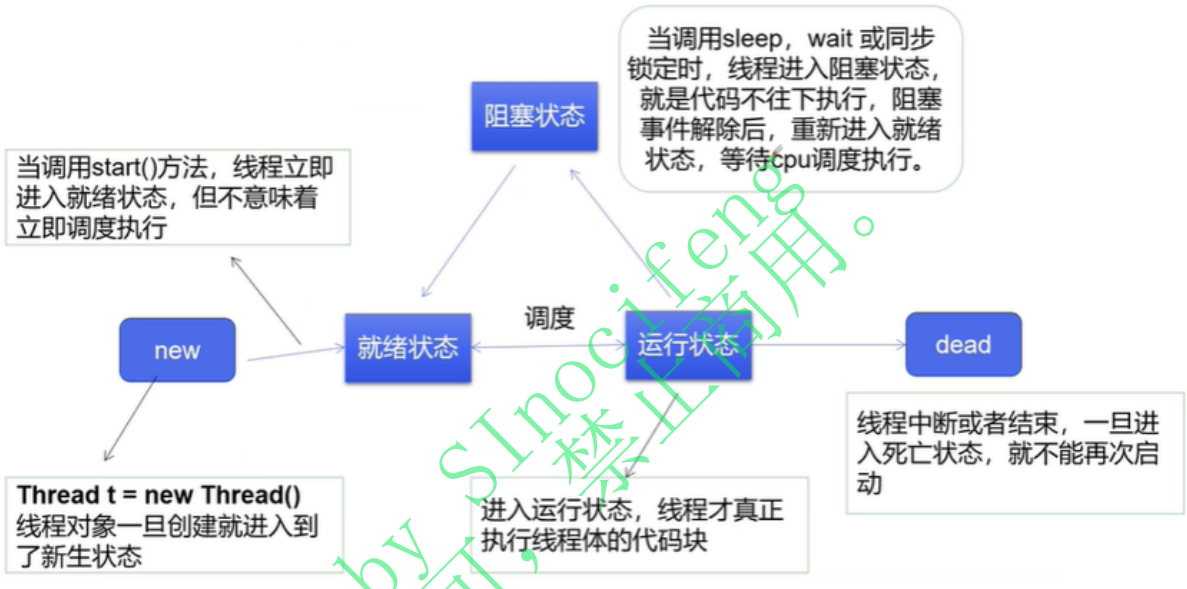
```
public class TestLambda{
    //1.Lambda表达式
    Ilove love = (int a)->{
        System.out.println("I love you for " + a +"years.");
    };
    //2.简化1：去掉参数类型
    love = (a)->{
        System.out.println("I love you for " + a +"years.");
    };
    //3.简化2：去掉括号
    love = a->{
        System.out.println("I love you for " + a +"years.");
    };
    //4.简化3：去掉花括号
    love = a-> System.out.println("I love you for " + a +"years.");

    love.love(521);
}

interface Ilove{
    void love(int a);
}
```

注意：

1. 前提必须要求接口是函数式接口
2. 简化1中去掉参数类型，如果是多个参数，则参数类型要去掉必须全部去掉
3. 简化2中如果是多个参数，则这个括号不能去掉
4. 简化3中去掉花括号是由于函数体中只包含一行代码，因此才能去掉花括号。如果函数体中包含多行代码。则这一大括号不能去掉

线程状态





线程方法

方法	说明
setPriority(int newPriority)	更改线程优先级
static void sleep(long millis)	在制定的毫秒内让当前正在执行的线程休眠
void join()	等待该线程终止
static void yield()	暂停当前正在执行的线程对象，并执行其他线程
void interrupt()	中断线程，一般不建议使用这个方式
boolean isAlive()	测试线程是否仍然处于活动状态

线程停止

- 不推荐使用JDK所提供的stop()、destory()方法【已废弃】
- 推荐让线程自己执行完成后停下来
- 建议使用一个标志位进行终止变量，当flag=false时，终止线程运行。

```
1  public class TestStop implements Runnable{
2      //1.设置线程中的线程体使用标识
3      private boolean flag = true;
4
```

```
    @override
    public void run(){
        //2.线程体中使用该标识
        while(flag){
            System.out.println("the thread is running...");
        }
    }

    //3.对外提供改变标识的方法
    public void stop(){
        this.flge = false;
    }
}
```

线程休眠

- sleep(time) 指定当前线程阻塞的毫秒数
- sleep存在异常InterruptedException
- sleep时间达到后线程进入就绪状态
- sleep可以模拟网络延时、倒计时等
- 每个对象都有一个锁，sleep不会释放锁

```
//实时获取车位信息
public void GetOnlineInfo()
{
    HttpBrowserCapabilities bc = Request.Browser;
    int hbcWidth = bc.ScreenPixelsWidth;
    //string hbcHeight = bc.ScreenPixelsHeight.ToString();

    //项目经理要求这里运行缓慢，好让客户给钱优化，并得到明显的速度提升
    Thread.Sleep(2000);
```

```
/**获取系统时间，每秒打印一次*/
public class TestSleep{
    public static void main(String[] args){
        Date currentTime = new Date(System.currentTimeMillis());
        while (1){
            try{
                Thread.sleep(1000);
                System.out.println(new
SimpleDateFormat("HH:mm:ss").format(currentTime));
                currentTime = new Date(System.currentTimeMillis());
            } catch(InterruptedException e){
                e.printStackTrace();
            }
        }
    }
}
```

线程礼让(yield)

- 礼让线程，让当前正在执行的线程暂停，但不阻塞
- 让线程从运行状态进入就绪状态
- 让CPU重新进行调度。礼让未必成功，看CPU调度结果

```java
public class TestYield{
    public static void main(String[] args){
        YieldThread myYield = new YieldThread();

        new Thread(myYield, "a").start();
        new Thread(myYield, "b").start();
    }
}

class YieldThread implements Runnable {
    @override
    public void run(){
        System.out.println(Thread.currentThread().getName() + "线程开始运行");
        Thread.yield();
        System.out.println(Thread.currentThread().getName() + "线程停止执行");
    }
}
```

线程强制执行Join

- Join合并线程，待此线程执行完成后，再执行其他线程，其他线程阻塞

```
略
```

观察线程状态

- Thread State

  线程状态，可以是以下状态之一

  - NEW 尚未启动的线程处于这一状态
  - RUBBAVLE 在Java虚拟机中执行的线程处于这一状态
  - BLOCKED 被阻塞等待监视器锁定的线程处于这一状态
  - WAITING 正在等待另一个线程执行特定动作的线程处于这一状态
  - TIMED_WAITING 正在等待另一个线程执行动作达到执行等待时间的线程处于这一状态
  - TERMINATED 已退出的线程处于这一状态

线程优先级

- Java提供一个线程调度器来监控程序中启动后进入就绪状态的所有线程，线程调度器按照其优先级决定应该调度哪个线程来执行
- 线程优先级用数字表示，范围为1~10
  - Thread.MIN_PRIORITY = 1
  - Thread.MAX_PRIORITY = 10
  - Thread.NORM_PRIORITY = 5
- 使用以下方法来改变或获取优先级
  - getPriority().setPriority(int x)

注意：

1. 优先级设置应该在start()调度之前
2. 优先级低是获得调度的概率低，而不是优先级低就一定之后才会被调度。取决于CPU的调度。

守护(daemon)线程

- 线程分为**用户线程**（如main线程）和**守护线程**
- 虚拟机不许确保用户线程执行完毕

- 虚拟机不需要等到守护线程执行完毕（如果用户线程执行完毕，守护线程即使还在执行，虚拟机就可以停止了）
- 典型的守护线程有：后台记录操作日志、监控内存、垃圾回收等待

设置守护线程：`setDaemon(true);//默认为false，用户线程`

线程同步(*)

> 多个线程操作同一个资源。同步需要队列+锁。

并发：同一个对象被多个线程同时操作。如12306上抢票。

- 处理多线程问题。多个线程访问同一对象，且某些线程还想要修改这一对象。此时就需要线程同步机制。

  > 线程同步实际上就是一种等待机制。多个需要同时访问该对象的线程进入这一**对象的等待池**形成队列，等待前面的线程使用完毕后，下一个线程才能继续使用。

- 由于同一进程的多个线程共享同一存储空间，在带来方便的同时，也引发了访问冲突问题。为了保障数据在方法中被访问时的正确性，在访问时加入**锁机制Synochronized**，当一个线程获得对象的排他锁，独占资源，其他线程必须等待，使用完后释放锁即可，存在以下问题：
  - 一个线程持有锁会导致其他所有需要该锁的其他线程挂起
  - 多线程竞争下，加锁、释放锁会导致较多的上下文切换和调度延时，引发性能问题
  - 如果一个优先级高的线程等待一个优先级低的线程释放锁，会导致优先级倒置，产生性能问题

线程不安全案例

```java
public class UnsafeList{
    public static void main(String[] args){
        List<String> list = new ArrayList<String>();
        for(int i=0; i<10000 ; i++){
            new Thread(() -> {
                list.add(Thread.currentThread().getName());
            }).start();
        }
        System.out.println(list.size());
    }
}
```

可能会观察到 list 的 size 小于10000.

同步方法与同步块

- 为了保护类中的数据对象，我们采用private关键字来保护其只能通过方法进行访问。同样，为了进行方法同步，采用来synchronized关键字机制。其包含两种用法：synchronized方法和synchronized块。

  > 同步方法： public synchronized void method(int args){ }
  >
  > 同步块：synchronized (obj){}
  >
  > > - Obj称为**同步监视器**
  > >   - obj可以是任何对象，但是推荐采用共享资源作为同步监视器
  > >   - 同步方法中无需指定同步监视器，因为在同步方法中同步监视器就是this，是这个对象本身，或者是class(反射中讲)
  > > - 同步监视器的执行过程
  > >   - 第一个线程访问，锁定同步监视器，执行其中代码

> - 第二个线程访问，发现同步监视器被锁定，无法访问
> - 第一个线程执行完毕，解锁同步监视器
> - 第二个线程访问，发现同步监视器没有锁，然后锁定并访问

- synchronized方法控制对`对象`的访问，每个对象对应一把锁，每个synchronized方法都必须获得调用该方法的对象的锁才能执行，否则线程会阻塞。方法一旦执行，就独占该锁，直到方法返回才释放锁，后面被阻塞的线程才能获得这个锁，继续执行。

> 缺陷：若将一个大的方法申明为synchronized 将会影响效率

注意：

1. 方法中如果需要修改内容，才上锁，锁的太多会影响效率。
2. synchronized默认锁的是this，如果this不是临界资源，则需要说明 - 见同步块。

JUC介绍——CopyOnWriteArrayList

```
//JUC指 java.util.concurrent 这个包，其支持并发
public class TestJUC{
    public static void main(String[] args){
        //这里并没有手动锁控制，但由于使用了JUC，这里最终还是线程安全的，list.size()为10000
        CopyOnWriteArrayList<String> list = new CopyOnWriteArrayList<String>();
        for (int i=0; i<10000; i++){
            new thread(()->{
                list.add(Thread.currentThread().getName());
            });
        }
        System.out.println(list.size());
    }
}
```

死锁

- 多个线程各自占有一些共享资源，并且相互等待其他线程占用资源才可以运行，从而导致两个或多个线程都在等待对方释放资源，都停止执行的情况。某一个同步块同时拥有**两个以上对象的锁**时，就有可能发生死锁。
- 死锁产生的四个必要条件
    1. 互斥条件：一个资源每次职能被一个进程使用
    2. 请求与保持条件：一个进程请求资源阻塞时，对已获得资源保持不放
    3. 不剥夺条件：进程已获得的资源，在未使用完之前，不可被强行剥夺
    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
- 死锁避免（思路：破坏四个条件中的任意一个或多个即可）

Lock锁

前面提到的`synchronized`提供的是隐式的锁，我们难以对其进行观察。在JDK 5.0后，Java提供了一种更加强大的线程同步机制——通过显式定义同步锁对象来实现同步。同步锁使用Lock对象充当。

- java.util.concurrent.locks.Lock接口是控制多个线程对共享资源进行访问的工具。锁提供了对共享资源的独占访问，每次只能有一个线程对Lock对象加锁，线程开始访问共享资源之前应该先获得Lock对象
- `ReentrantLock`类实现了 Lock，它拥有与`synchronized`相同的并发性和内存语义，在实现线程安全控制中，常用的是ReentrantLock，可以显式加锁、释放锁。（可重入锁）

```
Class A{
    private final ReentrantLock lock = new ReentrantLock();
    public void m(){
        lock.lock();//加锁
        try{
            //需要保证线程安全的代码
        }finally{
            lock.unlock();//释放锁
            //如果同步代码有异常，需要讲unlock()写入到finally语句块
        }
    }
}
```

synchronized和Lock的对比

- Lock为显式锁（手动开启和关闭锁），synchronized是隐式锁（出了作用域自动释放锁）
- Lock指引代码块锁，synchronized有代码块锁和方法锁
- 使用Lock锁，JVM花费较少时间来进行线程调度，性能更好，且其具有更好的扩展性（提供更多子类）
- 优先使用顺序：
  - Lock > 同步代码块(已经进入方法体，分配了相应资源) > 同步方法(在方法体之外)

线程通信

生产者-消费者问题

- 生产者，在没有生产产品前，需要通知消费者等待，生产产品后，又需要马上统治消费者消费
- 消费者，在消费之后，要通知生产者已经结束消费，需要生产新的产品以供消费
- 在生产者-消费者问题中，仅有synchronized是不够的
  - synchronized可以阻止并发更新同一个共享资源，实现了同步
  - synchronized不能用于实现不同线程之间的消息传递（通信）

java提供了一些方法来解决线程之间通信问题

方法名	作用
wait()	表示线程会一直等待，直到其他线程通知，与sleep()不同，会释放锁
wait(long timeout)	指定等待的毫秒数
notify()	唤醒一个处于等待状态的线程
notifyAll()	唤醒同一个对象上所有调用wait()方法的线程，优先级高的线程优先调度

注意：均是Object类的方法，都只能在同步方法或同步代码中使用，否则会抛出异常IllegalMonitorStateException

问题解决方式梳理

1. 管程
   - 生产者：负责生产数据（可能是方法、对象、线程、进程）
   - 消费者：负责处理数据（可能是方法、对象、线程、进程）
   - 缓冲区：消费者不能直接使用生产者数据，二者之间存在一个"缓冲区"

生产者将生产好的数据放入缓冲区，消费者从缓冲区拿取数据

2. 信号量

设置一个标志位flag，如果flag为true，让线程等待；如果flag为false，让线程通知另一个线程。

用管程处理生产者-消费者问题

```
Public class TestMonitor{
    public static void main(String[] args){
        SynContainer container = new SynContainer();
        new Productor(container).start();
        new Consumer(container).start();
    }
}
//生产者
class Productor extends Thread{
    synchronized container;
    public Productor(SynContainer container){
        this.container = container;
    }
    //生产产品
    @Override
    pulic void run(){
        for(int i=0; i<100; i++){
            container.push(new Food(i));
            System.out.println("生产者生产了"+i+"food");
        }
    }
}
//消费者
class Consumer extends Thread{
    synchronized container;
    public Consumer(SynContainer container){
        this.container = container;
    }
    //消费产品
    @Override
    pulic void run(){
        for(int i=0; i<100; i++){
            System.out.println("消费者消费了"+container.pop().id+"food");
        }
    }
}
//产品
class Food{
    int id;  //产品编号
    public Food(int id){
        this.id = id;
    }
}
//缓冲区--管程
class SynContainer{
    Food[] foods = new Food[10];  //大小为10的缓冲区
    int count = 0;                //缓冲区中已使用大小
    //生产者放入产品
    public synchronized void push(Food food){
        if(count==foods.length){  //生产者等待
            this.wait();
```

```java
        }
        //放入产品
        foods[count] = food;
        count++;
        //通知消费者消费
        this.notifyAll();

    }
    //消费者消费产品
    public synchronized Food pop(){
        if(count==0){              //消费者等待
            this.wait();
        }
        count--;
        Food food = foods[count];
        //通知生产者生产
        this.notifyAll();
        return food;
    }
}
```

用信号量处理生产者-消费者问题

```java
Public class TestSignal{
    public static void main(String[] args){
        Commodity commodity = new Commodity();
        new Productor(commodity).start();
        new Consumer(commodity).start();
    }
}
//生产者
class Productor extends Thread{
    Commodity commodity;
    public produce(commodity){
        this.commodity = commodity;
    }
    @Override
    pulic void run(){
        for(int i=0; i<20; i++){
            if (i%2 == 0) this.commodity.put("香蕉牛奶");
            else this.commodity.put("燕麦面包");
        }
    }
}
//消费者
class Consumer extends Thread{
    Commodity commodity;
    public consume(commodity){
        this.commodity = commodity;
    }
    @Override
    pulic void run(){
        for(int i=0; i<20; i++){
            commodity.get();
        }
    }
}
```

```java
//商品
class Commodity{
    String name;  //产品名称
    boolean flag = true;//true表示生产者生产，false表示消费者消费

    //生产
    public synchronized void put(String name){
        if(!flag){
            this.wait();
        }
        System.out.println("生产者生产了商品："+name);
        //通知消费者消费
        this.notifyAll();
        this.name = name;
        this.flag = !this.flag;
    }
    //消费
    public synchronized void get(){
        if(flag){
            this.wait();
        }
        System.out.println("消费者消费了商品："+name);
        //通知生产者生产
        this.notifyAll();
        this.flag = !this.flag;
    }
}
```

线程池

- 背景：经常创建和销毁、使用量特别大的资源，比如并发情况下的线程，对性能影响很大。
- 思路：提前创建很多个线程，放入到线程池，使用时直接获取，用完后放回池中。从而避免频繁创建销毁，实现重复利用。（类似公共交通工具）
- 好处：
  - 提高想要速度（减少创建新线程时间）
  - 降低资源消耗（重复利用线程池中线程，无需每次都创建）
  - 便于线程管理
    - corePoolSize: 核心池的大小
    - maximumPoolSize: 最大线程数
    - keepAliveTime: 线程没有任务时最多保持多久会终止

JDK5.0起提供了线程池相关的API: `ExecutorService`和`Executors`

- ExecutorService：真正的线程池。常见子类为ThreadPoolExecutor
  - `void execute(Runnable command)`：执行任务/命令，没有返回值，一般用于执行Runnable
  - `<T>Future<T> submit(Callable<T> task)`：执行任务/命令，有返回值，一般用于执行Callable
  - `void shutdown()`：关闭连接池
- Executors：工具类，线程池的工厂类，用于创建并返回不同类型的线程池

```java
//测试线程
public class TestPool{
```

```
    public static void main(String[] args){
        //1.创建线程池,大小为10
        ExecutorService service = Executors.newFixedThreadPool(10);
            //执行
        service.execute(new MyThread);
        //若是Callable，使用service.submit(threadObj)来执行

        //2.关闭连接
        service.shutdown();

    }
}
class MyThread implements Runnable{
    @Override
    pulic void run(){
        for(int i=0; i<5; i++){
            System.out.println(Thread.currentThread().getName + i);
        }
    }
}
```