引例:单线程不能满足"同时"的需求

程序不停地在屏幕上输出一句问候的语句(比如“你好”)
“同时”，当我通过键盘输入固定输入的时候，程序停止向屏幕输出问候的语句(比如说输入gun)

单线程

没有办法解决"同时"发生

```
package _20thread01.com.cskaoyan._01introduction;

import java.util.Scanner;
import java.util.concurrent.TimeUnit;

/**
 * @description:
 * @author: 景天
 * @date: 2022/7/26 14:28
 **/
/*
程序不停地在屏幕上输出一句问候的语句(比如“你好”)
“同时”，当我通过键盘输入固定输入的时候，程序停止向屏幕输出问候的语句(比如说输入gun)

 */
public class Demo {
    // 定义一个flag值
    public static boolean flag = true;
    public static void main(String[] args) {
        System.out.println("main before");
        System.out.println("hello before");

        sayHello();
        System.out.println("hello after");

        System.out.println("wait before");

        waitToStop();
        System.out.println("wait after");

        System.out.println("main after");

    }

    private static void waitToStop() {
        Scanner scanner = new Scanner(System.in);
        while (flag) {
            String s = scanner.nextLine();
            if ("gun".equals(s)) {
                flag = false;
                break;
            }
        }
    }
```

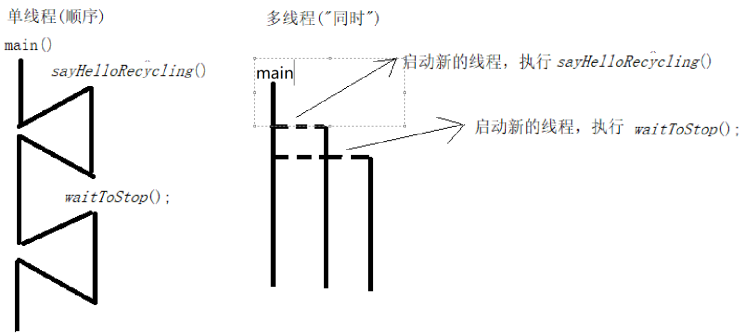
```
    private static void sayHello() {
        while (flag) {
            System.out.println("你好");
            // 暂停3s
            try {
                TimeUnit.SECONDS.sleep(3);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
}
```

多线程

```
package _20thread01.com.cskaoyan._01introduction;

import java.util.Scanner;
import java.util.concurrent.TimeUnit;

/**
 * @description:
 * @author: 景天
 * @date: 2022/7/26 14:28
 **/
/*
程序不停地在屏幕上输出一句问候的语句(比如“你好”)
“同时”，当我通过键盘输入固定输入的时候，程序停止向屏幕输出问候的语句(比如说输入gun)

 */
public class Demo2 {
    // 定义一个flag值
    public static boolean flag = true;
    public static void main(String[] args) {
        System.out.println("main before");
        System.out.println("hello before");

        sayHello();
        System.out.println("hello after");

        System.out.println("wait before");

        waitToStop();
        System.out.println("wait after");

        System.out.println("main after");

    }

    private static void waitToStop() {
        new Thread(new Runnable() {
```

```
            @Override
            public void run() {
                Scanner scanner = new Scanner(System.in);
                while (flag) {
                    String s = scanner.nextLine();
                    if ("gun".equals(s)) {
                        flag = false;
                        break;
                    }
                }
            }
        }).start();

    }

    private static void sayHello() {
        new Thread(new Runnable() {
            @Override
            public void run() {
                while (flag) {
                    System.out.println("你好");
                    // 暂停3s
                    try {
                        TimeUnit.SECONDS.sleep(3);
                    } catch (InterruptedException e) {
                        e.printStackTrace();
                    }
                }
            }
        }).start();

    }
}
```

我们的代码都是在某一条执行路径下按顺序依次执行的,单线程就是一条执行路径,多线程就是多条执行路径



操作系统基本概念

进程(process)与线程(thread)

进程

- 计算机程序在某个数据集合上的运行活动.进程是操作系统进行资源调度与分配的基本单位
- 正在运行的程序或者软件

线程

- 进程中有多个子任务,每个子任务就是一个线程. 从执行路径的角度看, 一条执行路径就是一个线程
- 线程是CPU进行资源调度与分配的基本单位

进程与线程的关系

- 线程依赖于进程而存在
- 一个进程中可以有多个线程(最少1个)
- 线程共享进程资源
- 举例: 迅雷

串行(serial),并行(parallel)与并发(concurrency)

串行

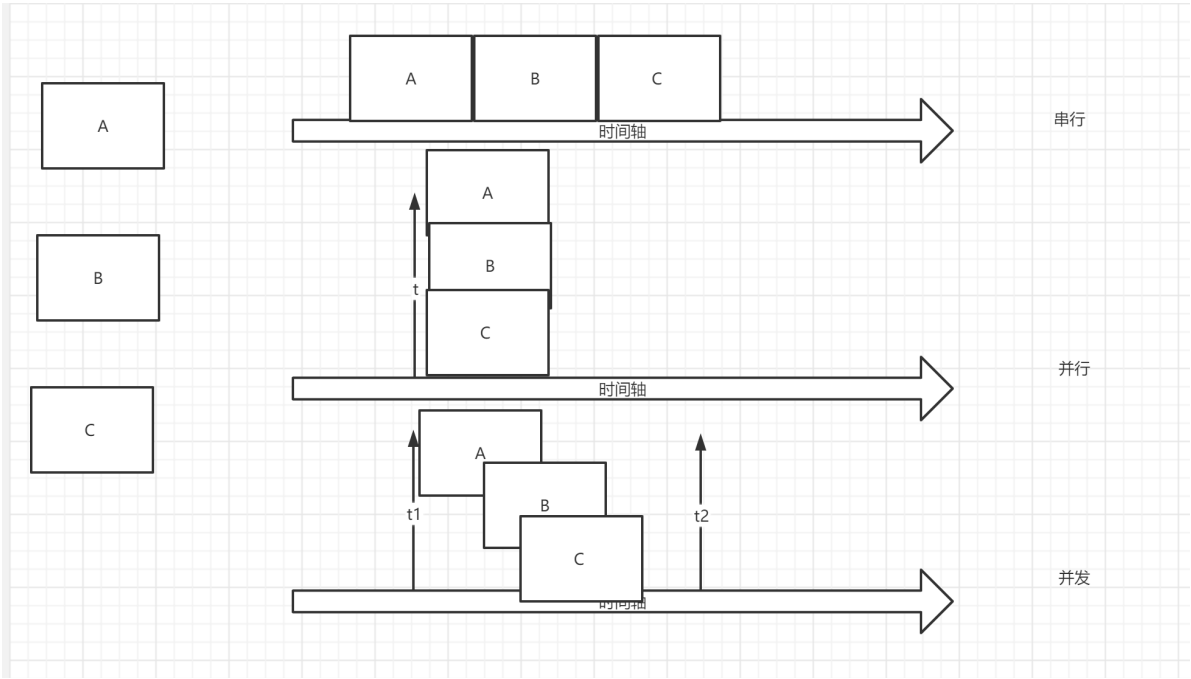
- 一个任务接一个任务按顺序执行

并行

- 在同一个时间点(时刻)上同时运行

并发

- 在同一时间段内,多个任务同时运行



同步(synchronization)与异步(asynchronization)

有2个任务(业务) A　B

同步:

- A任务执行的时候B不能执行,按顺序执行

- 你走我不走

异步:

- A任务执行的时候,B任务可以执行
- 你走你的,我走我的,互相不干扰
- 多线程是天生异步的

去书店买书,打电话问老板,有没有java书

同步: 老板接通电话,告诉我说我找一下,找到了告诉你, 电话没有挂断

异步: 老板接通电话,告诉我说我找一下,找到了告诉你, 电话挂断了. 找到了后打电话通知我

单道批处理: 内存中只有1个进程在运行

多道批处理 : 内存中有多个进程在运行 (进程的上下文切换,保护现场 , 回复现场)

现代操作系统: 引入了线程的概念

java程序运行原理

java命令+主类类名运行原理

- java命令创建一个jvm进程
- jvm会创建一个线程 (main)线程
- 执行main方法

jvm是单线程还是多线程的

结论: jvm是多线程的,除了main线程外,起码还有一个垃圾回收线程负责回收垃圾

多线程的实现方式一:继承Thread类

线程 是程序中的执行线程。Java 虚拟机允许应用程序并发地运行多个执行线程

文档示例

常。

创建新执行线程有两种方法。一种方法是将类声明为 Thread 的子类。该子类应重写 Thread 类的 run 方法。接下来可以分配并启动该子类的实例。例如，计算大于某一规定值的质数的线程可以写成：

```
class PrimeThread extends Thread {
    long minPrime;
    PrimeThread(long minPrime) {
        this.minPrime = minPrime;
    }

    public void run() {
        // compute primes larger than minPrime
         . . .
    }
}
```

然后，下列代码会创建并启动一个线程：

```
PrimeThread p = new PrimeThread(143);
p.start();
```

步骤

1. 定义一个类继承Thread
2. 重写run方法
3. 创建子类的对象
4. 通过start方法启动线程

Demo

```java
package _20thread01.com.cskaoyan._02implone;

/**
 * @description:
 * @author: 景天
 * @date: 2022/7/26 17:52
 **/
/*
多线程实现方式一
1. 定义一个类继承Thread
2. 重写run方法
3. 创建子类的对象
4. 通过start方法启动线程
 */
public class Demo {
    public static void main(String[] args) {
        MyThread t = new MyThread();

        //启动
        t.start();
    }
}

class MyThread extends Thread{
    @Override
    public void run() {
        System.out.println("线程执行了!");
    }
}
```

注意事项

多线程的执行特点是什么?

特点是随机的

```
Thread-0---------0
Thread-1---------0
Thread-1---------1
Thread-1---------2
Thread-1---------3
Thread-0---------1
Thread-0---------2
Thread-0---------3
Thread-0---------4
Thread-0---------5
Thread-0---------6
Thread-0---------7
Thread-1---------4
Thread-0---------8
Thread-0---------9
Thread-0---------10
Thread-1---------5
Thread-1---------6
Thread-1---------7
Thread-1---------8
Thread-1---------9
Thread-1---------10
Thread-1---------11
Thread-0---------11
Thread-1---------12
Thread-1---------13
Thread-1---------14
Thread-1---------15
Thread-1---------16
Thread-1---------17
Thread-1---------18
Thread-1---------19
Thread-0---------12
Thread-0---------13
Thread-0---------14
Thread-0---------15
Thread-0---------16
Thread-0---------17
Thread-0---------18
Thread-0---------19
```

start方法跟run方法有什么区别?

```
package _21thread02.com.cskaoyan._01implone;

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

```java
/**
 * @description:
 * @author: 景天
 * @date: 2022/7/27 9:39
 **/
/*
  start方法 跟run方法有啥区别?
 */
public class Demo2 {
    public static void main(String[] args) {
        System.out.println("main before");
        // 创建子类对象
        MyThread2 t1 = new MyThread2();
        // start
        t1.start();
        //t1.run();
        System.out.println("main after");

    }
}

class MyThread2 extends Thread{
    //run


    @Override
    public void run() {
        for (int i = 0; i < 10; i++) {
            // getName()
            System.out.println(getName()+"---------"+i);
            try {
                TimeUnit.SECONDS.sleep(1);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
}
```

使用run方法,效果就是顺序执行,并没有开辟新的执行路径,还是单线程的,只是普通方法调用

start方法开辟了新的执行路径,多线程的

```
run方法的执行结果
main before
Thread-0---------0
Thread-0---------1
Thread-0---------2
Thread-0---------3
Thread-0---------4
Thread-0---------5
Thread-0---------6
Thread-0---------7
```

```
Thread-0---------8
Thread-0---------9
main after

start方法的执行结果
main before
main after
Thread-0---------0
Thread-0---------1
Thread-0---------2
Thread-0---------3
Thread-0---------4
Thread-0---------5
Thread-0---------6
Thread-0---------7
Thread-0---------8
Thread-0---------9
```

同一个线程能否启动多次?

- java.lang.IllegalThreadStateException
- 不能启动多次

谁才代表一个线程?

Thread对象及其子类对象才代表一个线程

设置获取线程名称

String	getName() 返回该线程的名称。默认线程名称Thread-编号
void	setName(String name) 设置线程名称

如何获取主线程的名字?

static Thread	currentThread() 返回对当前正在执行的线程对象的引用。

```
package _21thread02.com.cskaoyan._01implone;

/**
 * @description:
 * @author: 景天
 * @date: 2022/7/27 9:39
 **/
```

```java
/*
多线程的执行特点
 */
public class Demo3 {
    public static void main(String[] args) {
        //currentThread()
        // 返回对当前正在执行的线程对象的引用。
        Thread thread = Thread.currentThread();
        System.out.println(thread.getName());
        // 创建子类对象
        MyThread3 t1 = new MyThread3();
        // 设置名字
        t1.setName("王道吴彦祖");
        // start
        t1.start();
    }
}

class MyThread3 extends Thread{
    //run


    @Override
    public void run() {
        for (int i = 0; i < 10; i++) {
            // getName()
            System.out.println(getName()+"---------"+i);
        }
    }
}
```

线程的调度方式

什么是线程调度

概念: 给线程分配CPU处理权的过程

调度方式的分类

- 协同式线程调度
  - 线程的执行时间是由线程决定的,如果某个线程执行完报告操作系统切换到别的线程执行
- 抢占式线程调度
  - 线程的执行时间由系统决定,哪个线程抢到哪个线程执行

java中采用哪种调度方式

抢占式的调度方式

线程的优先级(了解)

操作系统优先级

- 动态优先级
  - 正在执行的线程会随着执行时间的延长优先级降低
  - 正在等待的线程会随着等待的时间的延长优先级升高
- 静态优先级: 固定数值
- 静态+动态

java中优先级

静态优先级 范围1-10

static int	**MAX_PRIORITY 线程可以具有的最高优先级。 10**
static int	MIN_PRIORITY 线程可以具有的最低优先级。 1
static int	NORM_PRIORITY 分配给线程的默认优先级。 5

设置获取优先级

int	**getPriority() 返回线程的优先级**
void	setPriority(int n) 设置线程的优先级

练习:

创建并启动2个线程

A线程打印10个数, A 线程优先级为10

B线程打印10个数, B线程优先级为1

问: A线程打印10个数后再去执行B

结论: java中优先级用处不大

然而，我们在java语言中设置的线程优先级，它仅仅只能被看做是一种"建议"(对操作系统的建议)，
实际上，操作系统本身，有它自己的一套线程优先级 (静态优先级 + 动态优先级)

java官方： 线程优先级并非完全没有用，我们Thread的优先级，它具有统计意义，总的来说，高优先级的线程
占用的cpu执行时间多一点，低优先级线程，占用cpu执行时间，短一点

```
package _21thread02.com.cskaoyan._02api;

/**
 * @description:
 * @author: 景天
 * @date: 2022/7/27 10:21
 **/

public class PriorityDemo2 {
    public static void main(String[] args) {
        // 创建子类对象
        MyThread2 t = new MyThread2();
```

```
        MyThread2 t1 = new MyThread2();

        // 设置优先级
        t.setPriority(Thread.MAX_PRIORITY);
        t1.setPriority(Thread.MIN_PRIORITY);

        // start
        t.start();
        t1.start();

    }
}

class MyThread2 extends Thread{
    // run

    @Override
    public void run() {
        for (int i = 0; i < 10; i++) {
            System.out.println(getName()+"------"+i);
        }
    }
}
```

线程控制API

线程休眠sleep

static void	sleep(long millis) 在指定的毫秒数内让当前正在执行的线程休眠（暂停执行），此操作受到系统计时器和调度程序精度和准确性的影响。

```
package _21thread02.com.cskaoyan._02api;

import java.util.concurrent.TimeUnit;

/**
 * @description: 线程休眠
 * @author: 景天
 * @date: 2022/7/27 10:31
 **/

public class SleepDemo {
    public static void main(String[] args) {
        System.out.println("main before");
        // 创建子类对象并启动
        new ThreadSleep().start();
```

```
        System.out.println("main after");

    }
}

class ThreadSleep extends Thread{
    // run

    @Override
    public void run() {
        for (int i = 0; i < 10; i++) {
            System.out.println(i);
            try {
                // TimeUnit.SECONDS.sleep(1) 跟sleep等价 效果一样
                Thread.sleep(1000);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
}
```

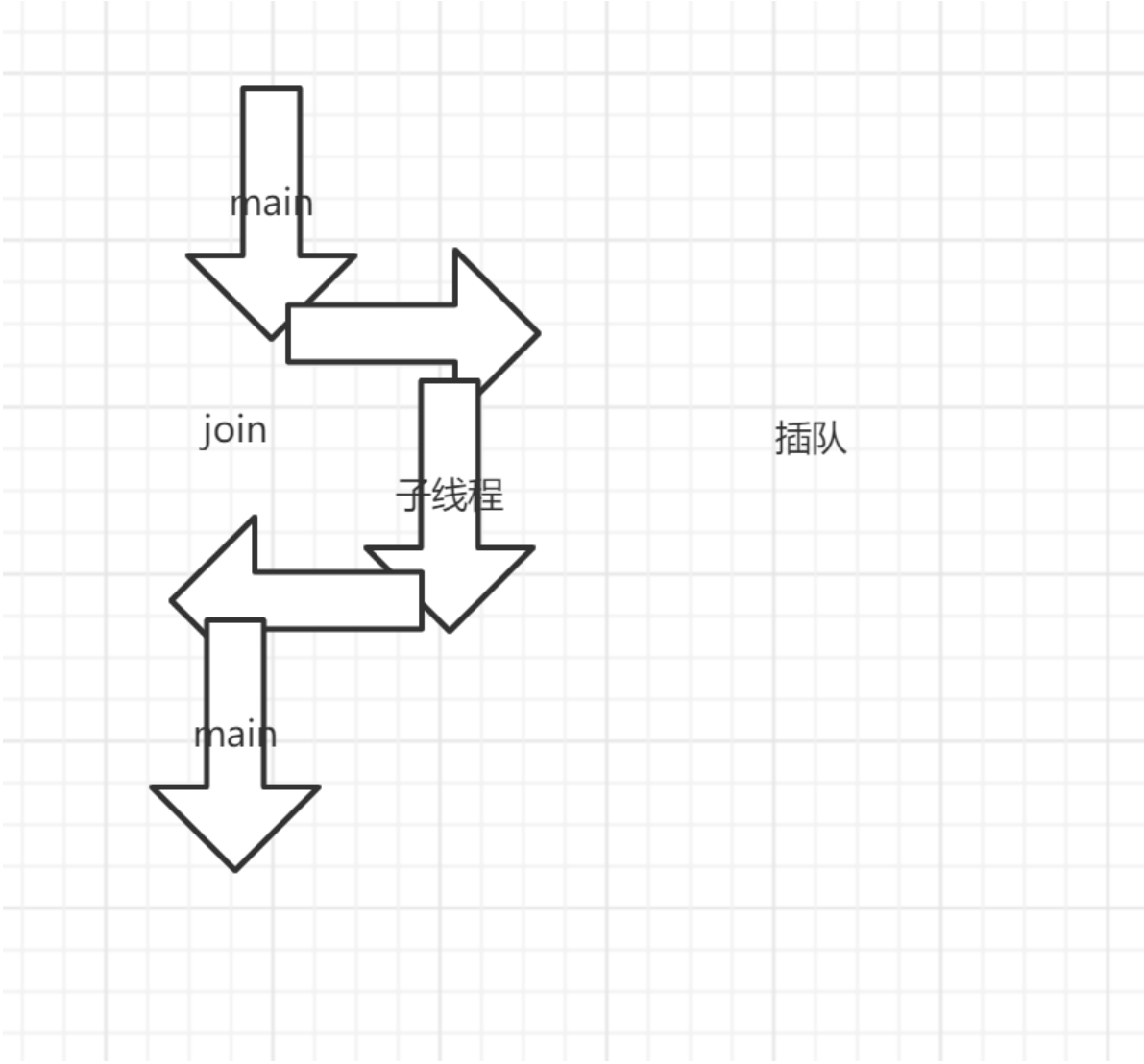
线程合并join

void	join() 等待该线程终止。

```
不使用join的执行结果
main before
0
1
2
main after
Thread-0----0
Thread-0----1
Thread-0----2
Thread-0----3
Thread-0----4
Thread-0----5
Thread-0----6
Thread-0----7
Thread-0----8
Thread-0----9

使用join的执行结果
main before
Thread-0----0
Thread-0----1
Thread-0----2
Thread-0----3
Thread-0----4
```

```
Thread-0----5
Thread-0----6
Thread-0----7
Thread-0----8
Thread-0----9
0
1
2
main after
```

```java
package _21thread02.com.cskaoyan._02api;

/**
 * @description:
 * @author: 景天
 * @date: 2022/7/27 10:54
 **/

public class JoinDemo {
    public static void main(String[] args) {
        System.out.println("main before");
        // 创建子类对象并启动
        ThreadJoin t = new ThreadJoin();
        t.start();
        // join
        try {
            t.join();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        for (int i = 0; i < 3; i++) {
            System.out.println(i);
        }
        System.out.println("main after");

    }
}

class ThreadJoin extends Thread{
    // run

    @Override
    public void run() {
        for (int i = 0; i < 10; i++) {
            System.out.println(getName()+"----"+i);
        }
    }
}
```



谁等待

- main线程在等待, join这行代码在哪个线程上运行就是哪个线程等待

等待谁

- 等待的是子线程, 哪个线程调用了join,等待的就是这个线程

线程礼让yield

static void	yield() 暂停当前正在执行的线程对象，并执行其他线程。

练习:

创建并启动2个线程

A线程B线程打印10个数

A打印一个0, B打印一个0,A打印一个1,B打印一个1....

结论: 用过yield做不到这样的功能

原因: java采用的是抢占式的调度方式, 虽然此时放弃了CPU的执行权,但是仍然可以参与下轮CPU的竞争

```java
package _21thread02.com.cskaoyan._02api;

/**
 * @description:
 * @author: 景天
 * @date: 2022/7/27 11:08
 **/
/*
线程礼让
练习:

创建并启动2个线程

A线程B线程打印10个数

A打印一个0, B打印一个0,A打印一个1,B打印一个1....
 */
public class YieldDemo {
    public static void main(String[] args) {
        // 创建并启动2个线程
        new ThreadYield("A").start();
        new ThreadYield("B").start();
    }
}

class ThreadYield extends Thread{
    // run

    public ThreadYield(String name) {
        super(name);
    }

    @Override
    public void run() {
        for (int i = 0; i < 10; i++) {
            System.out.println(getName()+"----"+i);
            // 执行yield  暂停当前正在执行的线程对象，并执行其他线程。
            Thread.yield();
        }
    }
}
```

守护线程setDaemon

线程分类

- 用户线程(默认)
    - 工作线程
- 守护线程
    - 为工作线程服务的线程(垃圾回收线程), 可以理解为工作线程的奴仆

void	setDaemon(boolean on) 将该线程标记为守护线程或用户线程
	on - 如果为 true，则将该线程标记为守护线程

```
package _21thread02.com.cskaoyan._02api;

/**
 * @description: 守护线程
 * @author: 景天
 * @date: 2022/7/27 11:20
 **/

public class DaemonDemo {
    public static void main(String[] args) {
        System.out.println("main before");
        // 创建子类对象
        ThreadDaemon t = new ThreadDaemon();
        // 设置为守护线程
        t.setDaemon(true);
        // start启动
        t.start();
        // main线程中打印3个数
        for (int i = 0; i < 3; i++) {
            System.out.println(i);
            try {
                Thread.sleep(1000);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
        System.out.println("main end");

    }
}

class ThreadDaemon extends Thread{
    // run
    @Override
    public void run() {
        for (int i = 0; i < 10; i++) {
            System.out.println(getName()+"----"+i);
            try {
                Thread.sleep(1000);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
}
```

注意:

- 当正在运行的线程都是守护线程时，Java 虚拟机退出。
- 该方法必须在启动线程前调用。 java.lang.IllegalThreadStateException

线程中断stop(已过时,了解)

void	stop() 已过时。 该方法具有固有的不安全性

安全中断线程

通过一个flag标记去控制线程中断

true 表示正常

false: 表示线程中断

练习:

启动一个子线程 打印10个数 打印一个睡1s

(如果没有中断,正常打印,如果有中断, 将中断信息保存到log.txt文件中

年月日 时分秒 哪个线程发生了中断)

main线程打印3个数 打印一个睡1s 去中断子线程 (更改flag值)

```
package _21thread02.com.cskaoyan._02api;

import java.io.FileWriter;
import java.io.IOException;
import java.text.SimpleDateFormat;
import java.util.Date;

/**
 * @description:
 * @author: 景天
 * @date: 2022/7/27 11:41
 **/
/*
通过一个flag标记去控制线程中断

true 表示正常

false: 表示线程中断

练习:

启动一个子线程 打印10个数 打印一个睡1s

(如果没有中断,正常打印,如果有中断, 将中断信息保存到log.txt文件中

年月日 时分秒 哪个线程发生了中断)

main线程打印3个数 打印一个睡1s  去中断子线程 (更改flag值)
```

```
 */
public class SecurityStop {
    public static void main(String[] args) {
        // 创建子类对象
        ThreadSecurityStop t = new ThreadSecurityStop();
        // start启动
        t.start();
        // main线程打印3个数 打印一个睡1s
        for (int i = 0; i < 3; i++) {
            System.out.println(i);
            try {
                Thread.sleep(1000);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
        //去中断子线程 (更改flag值)
        t.flag = false;
    }
}

class ThreadSecurityStop extends Thread{
    boolean flag = true;
    // run

    @Override
    public void run() {
        for (int i = 0; i < 10; i++) {
            // 对flag进行判断
            if (flag) {
                // 正常打印
                System.out.println(getName()+"----"+i);
                try {
                    Thread.sleep(1000);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }else{
                // 发生了中断
                // 保存日志信息
                // 日期对象
                SimpleDateFormat sdf = new SimpleDateFormat("yyyy-MM-dd
HH:mm:ss");
                String dateFormat = sdf.format(new Date());
                // 创建输出流对象
                FileWriter fileWriter = null;
                try {
                     fileWriter = new FileWriter("log.txt");
                    // write写数据
                    fileWriter.write(dateFormat+getName()+"-:发生了中断!");
                    // flush
                    fileWriter.flush();
                } catch (IOException e) {
                    e.printStackTrace();
                }finally {
```

```
                    // close
                    if (fileWriter != null) {
                        try {
                            fileWriter.close();
                        } catch (IOException e) {
                            e.printStackTrace();
                        }
                    }
                }
            }

        }
    }
}
```

线程的生命周期

线程的几种状态

理论层面的状态

新建

- 刚new出来的线程,此时还没有start

就绪

- start启动后

执行

- 抢到了CPU的执行权

阻塞

- 没有CPU的执行权,还缺少一些必要的条件(sleep,join)

死亡

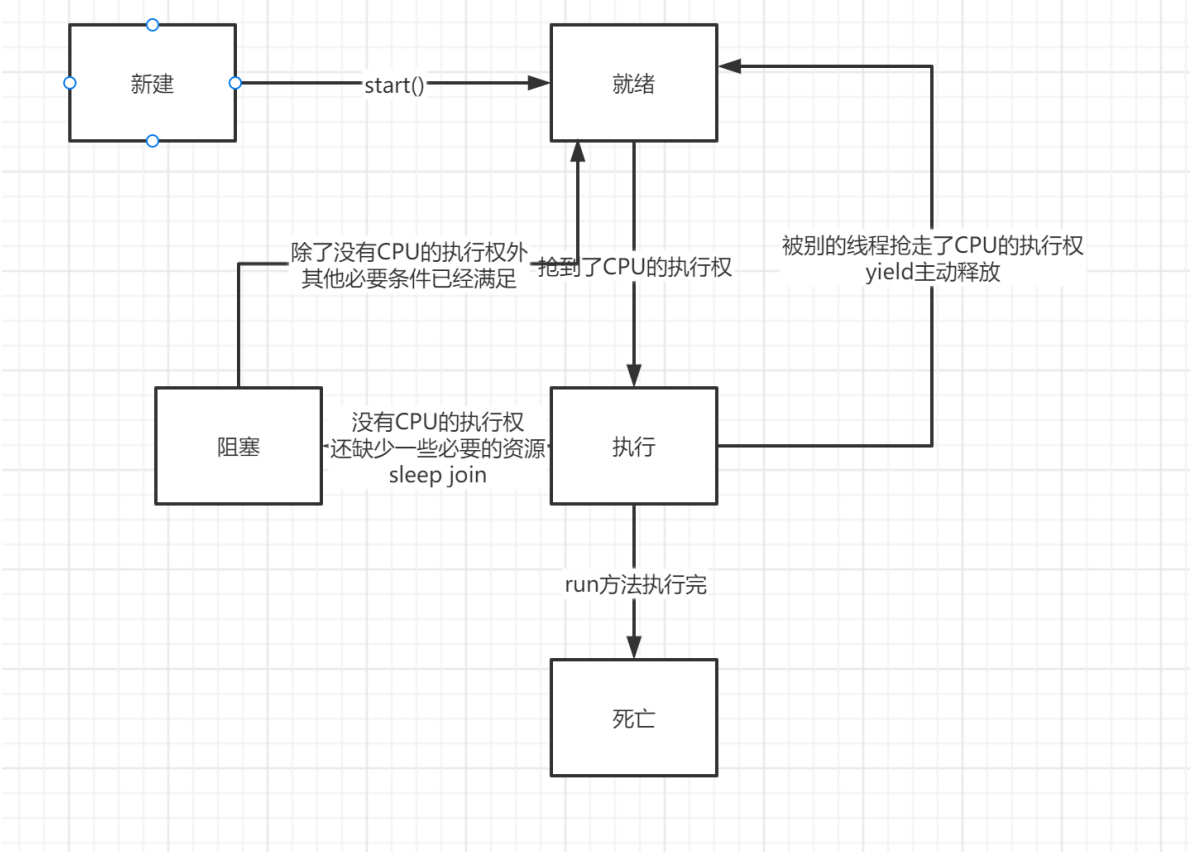
- run方法执行完

代码层面的状态

- NEW
  至今尚未启动的线程处于这种状态。
- RUNNABLE
  正在 Java 虚拟机中执行的线程处于这种状态。
- BLOCKED
  受阻塞并等待某个监视器锁的线程处于这种状态。
- WAITING
  无限期地等待另一个线程来执行某一特定操作的线程处于这种状态。
- TIMED_WAITING
  等待另一个线程来执行取决于指定等待时间的操作的线程处于这种状态。

- TERMINATED
  已退出的线程处于这种状态。

线程状态的转换



多线程实现方式二:实现Runnable接口

文档示例

创建线程的另一种方法是声明实现 Runnable 接口的类。该类然后实现 run 方法。然后可以分配该类的实例，在创建 Thread 时作为一个参数来传递并启动。采用这种风格的同一个例子如下所示：

```
class PrimeRun implements Runnable {
    long minPrime;
    PrimeRun(long minPrime) {
        this.minPrime = minPrime;
    }

    public void run() {
        // compute primes larger than minPrime
         . . .
    }
}
```

然后，下列代码会创建并启动一个线程：

```
PrimeRun p = new PrimeRun(143);
new Thread(p).start();
```

步骤

1. 定义一个类 实现Runnable接口
2. 重写run方法
3. 创建子类对象
4. 创建线程对象,把实现了Runnable接口的子类对象作为参数传递

5. 通过start方法启动

Demo

```
package _22thread03.com.cskaoyan._01impltwo;

/**
 * @description:
 * @author: 景天
 * @date: 2022/7/28 10:00
 **/
/*
1. 定义一个类 实现Runnable接口
2. 重写run方法
3. 创建子类对象
4. 创建线程对象,把实现了Runnable接口的子类对象作为参数传递
5. 通过start方法启动
 */
public class Demo {
    public static void main(String[] args) {
        MyRunnable myRunnable = new MyRunnable();
        Thread t = new Thread(myRunnable);
        t.start();
    }
}

class MyRunnable implements Runnable{

    @Override
    public void run() {
        System.out.println("子线程执行了");
    }
}
```

其他写法 匿名内部类 lambda

```
package _22thread03.com.cskaoyan._01impltwo;

/**
 * @description:
 * @author: 景天
 * @date: 2022/7/28 10:02
 **/

public class Demo2 {
    public static void main(String[] args) {
        // 使用匿名内部类
        new Thread(new Runnable() {
            @Override
            public void run() {
                System.out.println(11111);
            }
        }).start();
```

```
        // lambda
        new Thread(()->{
            System.out.println(2222222);
        }).start();
    }
}
```

为什么Runnable中的run方法会运行在子线程中

```
class Thread{
    // 定义成员变量
    private Runnable target;
    Thread(Runnable target){
        init(target);
    }
    void init(){
        // 左边这个是成员变量  右边这个是我们传的参数
        // 进行赋值
        this.target = target;
    }

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

}
```

方式一VS方式二

- 使用角度 方式一4步 方式二是5步
- 方式一是通过继承的方式,有单继承的局限性,方式二是通过实现接口的方式
- 方式二把执行路径跟执行路径要做的事情区分开(解耦)
- 方式二更便于数据共享

多线程数据安全问题

多线程仿真如下场景：
假设A电影院正在上映某电影，该电影有100张电影票可供出售，现在假设有3个窗口售票。请设计程序模拟窗口售票的场景。

分析：
3个窗口售票，互不影响，同时进行。----> 使用多线程 3个线程
3个窗口共同出售这100张电影票 ---> 3个线程共享数据 共享100票

方式一：

```
package _22thread03.com.cskaoyan._02datasecurity;

/**
 * @description:
 * @author: 景天
 * @date: 2022/7/28 10:30
 **/
/*
方式一模拟卖票过程
 */
public class Demo {
    public static void main(String[] args) {
        //创建线程对象并启动
        new SellWindow("A").start();
        new SellWindow("B").start();
        new SellWindow("C").start();
    }
}

class SellWindow extends Thread{
    public SellWindow(String name) {
        super(name);
    }

    // 定义票
    int tickets = 100;
    // run
    @Override
    public void run() {
        // 模拟卖票过程
        while (true) {
            // 进行判断
            if (tickets > 0) {
                System.out.println(getName()+"卖了第"+(tickets--)+
                        "票");
            }
        }
    }
}
```

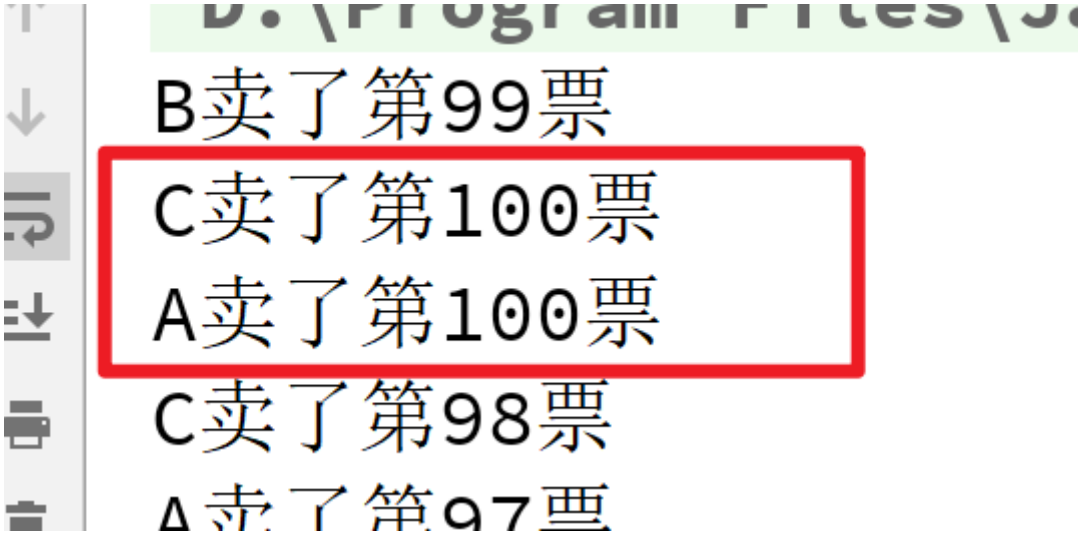
方式二：

```
package _22thread03.com.cskaoyan._02datasecurity;

/**
```

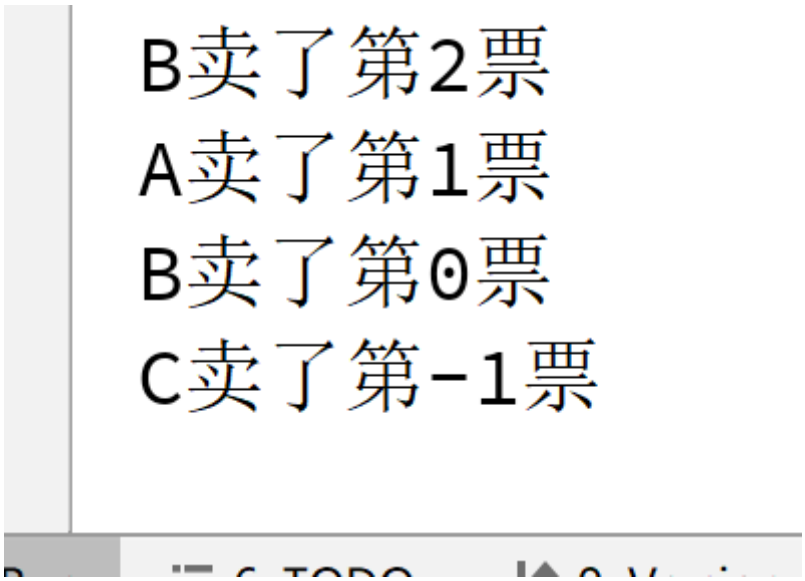
```
 * @description:
 * @author: 景天
 * @date: 2022/7/28 10:30
 **/
/*
方式一模拟卖票过程
 */
public class Demo2 {
    public static void main(String[] args) {
        //创建线程对象并启动
        SellWindow2 sellWindow2 = new SellWindow2();
        new Thread(sellWindow2, "A").start();
        new Thread(sellWindow2, "B").start();
        new Thread(sellWindow2, "C").start();
    }
}

class SellWindow2 implements Runnable{


    // 定义票
    int tickets = 100;
    // run
    @Override
    public void run() {
        // 模拟卖票过程
        while (true) {
            // 进行判断
            if (tickets > 0) {
                System.out.println(Thread.currentThread().getName()
                        +"卖了第"+(tickets--)+ "票");
            }
        }
    }
}
```

造成的现象

出现重复的票



出现不存在的票



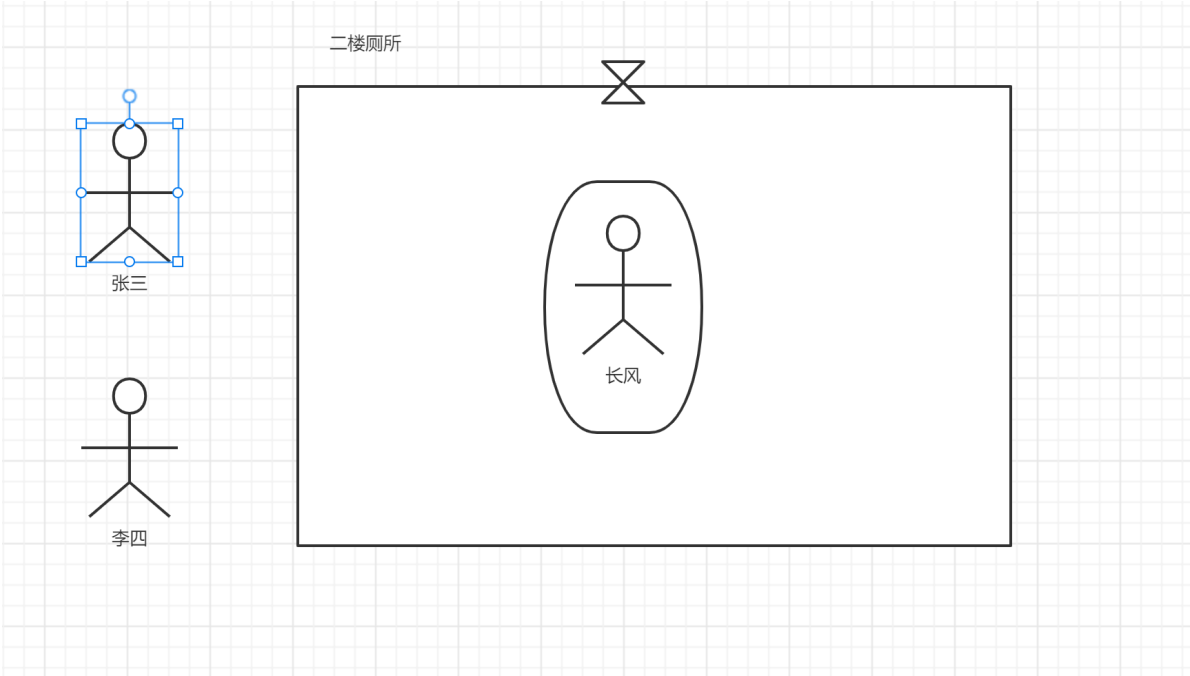
产生原因

- 多线程的运行环境(需求)
- 多线程共享数据(需求)
- 出现了非原子操作
    - 原子操作: 一个不可分割的操作(一个操作要么完成要么不完成)

解决多线程数据安全问题

思路: 破坏产生的原因

引出锁的概念



synchronized

同步代码块

同步代码块中,锁对象可以是任意的java对象,但是必须是唯一的

```
语法
    // 什么对象能够充当锁的角色
synchronized(锁对象){
    // 对共享数据的访问
}
```

```
package _22thread03.com.cskaoyan._03sync;

/**
 * @description:
 * @author: 景天
 * @date: 2022/7/28 11:34
 **/
/*
使用同步代码块
 */
public class Demo {
    public static void main(String[] args) {
        //创建线程对象并启动
        SellWindow sellWindow = new SellWindow();
        new Thread(sellWindow, "A").start();
        new Thread(sellWindow, "B").start();
        new Thread(sellWindow, "C").start();
    }
}
class SellWindow implements Runnable{


    // 定义票
```

```
    int tickets = 100;
    //Object obj = new Object();
    A obj = new A();
    // run
    @Override
    public void run() {
        // 模拟卖票过程
        while (true) {

            // 使用同步代码块
            synchronized (obj) {
                if (tickets > 0) {
                    try {
                        Thread.sleep(100);
                    } catch (InterruptedException e) {
                        e.printStackTrace();
                    }
                    System.out.println(Thread.currentThread().getName()
                            +"卖了第"+(tickets--)+ "票");

                }
            }

        }
    }
}

class A{}
```

同步方法

同步方法的锁对象是this

```
package _22thread03.com.cskaoyan._03sync;

/**
 * @description:
 * @author: 景天
 * @date: 2022/7/28 11:34
 **/
/*
使用同步代码块
 */
public class Demo2 {
    public static void main(String[] args) {
        //创建线程对象并启动
        SellWindow2 sellWindow = new SellWindow2();
        new Thread(sellWindow, "A").start();
        new Thread(sellWindow, "B").start();
        new Thread(sellWindow, "C").start();
    }
}
```

```
class SellWindow2 implements Runnable {


    // 定义票
    int tickets = 100;
    //Object obj = new Object();
    int i = 0;

    // run
    @Override
    public void run() {
        // 模拟卖票过程
        while (true) {
            if (i % 2 == 0) {
                synchronized (this) {
                    if (tickets > 0) {
                        try {
                            Thread.sleep(100);
                        } catch (InterruptedException e) {
                            e.printStackTrace();
                        }
                        System.out.println(Thread.currentThread().getName()
                                + "卖了第" + (tickets--) + "票");

                    }
                }

            } else {
                sell();

            }
            i++;

        }
    }

    private synchronized void sell() {

        if (tickets > 0) {
            try {
                Thread.sleep(100);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            System.out.println(Thread.currentThread().getName()
                    + "卖了第" + (tickets--) + "票");

        }
    }

}
```

静态同步方法

静态的同步方法的锁对象是字节码文件对象

```
package _22thread03.com.cskaoyan._03sync;

/**
 * @description:
 * @author: 景天
 * @date: 2022/7/28 11:34
 **/
/*
使用同步代码块
 */
public class Demo3 {
    public static void main(String[] args) {
        //创建线程对象并启动
        SellWindow3 sellWindow = new SellWindow3();
        new Thread(sellWindow, "A").start();
        new Thread(sellWindow, "B").start();
        new Thread(sellWindow, "C").start();
    }
}

class SellWindow3 implements Runnable {


    // 定义票
    static int tickets = 100;
    //Object obj = new Object();
    int i = 0;

    // run
    @Override
    public void run() {
        // 模拟卖票过程
        while (true) {
            if (i % 2 == 0) {
                // 静态的方法的锁对象是字节码文件对象Class对象
                // 获取字节码文件的方式
                // 对象.getClass()
                // 类名.class
                // Class.forName("全类名")
                synchronized (SellWindow3.class) {
                    if (tickets > 0) {
                        try {
                            Thread.sleep(100);
                        } catch (InterruptedException e) {
                            e.printStackTrace();
                        }
                        System.out.println(Thread.currentThread().getName()
                                + "卖了第" + (tickets--) + "票");

                    }
                }
```

```
            } else {
                sell();

            }
            i++;

        }
    }

    private static synchronized void sell() {

        if (tickets > 0) {
            try {
                Thread.sleep(100);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            System.out.println(Thread.currentThread().getName()
                    + "卖了第" + (tickets--) + "票");

        }
    }

}
```

synchronized的细节

执行流程

- 假设有2个线程A B 都要访问同步代码块中的内容
- 假设A线程先执行, 访问同步代码块,看一下锁是否可用,可以用,拿到锁对象---> 访问代码块内容, 没有执行完的时候发生了线程的切换,切换到B线程
- B线程执行, 访问同步代码块, 看一下锁是否可用, 发现锁对象被A线程持有, B线程不能访问同步代码块,B线程就需要等待, 同步阻塞.又切换A线程
- A线程接着执行,退出sync代码块, 释放锁

```
package _22thread03.com.cskaoyan._03sync;

import java.util.concurrent.TimeUnit;

/**
 * @description:
 * @author: 景天
 * @date: 2022/7/28 14:43
 **/
/*
sync执行流程
 */
```

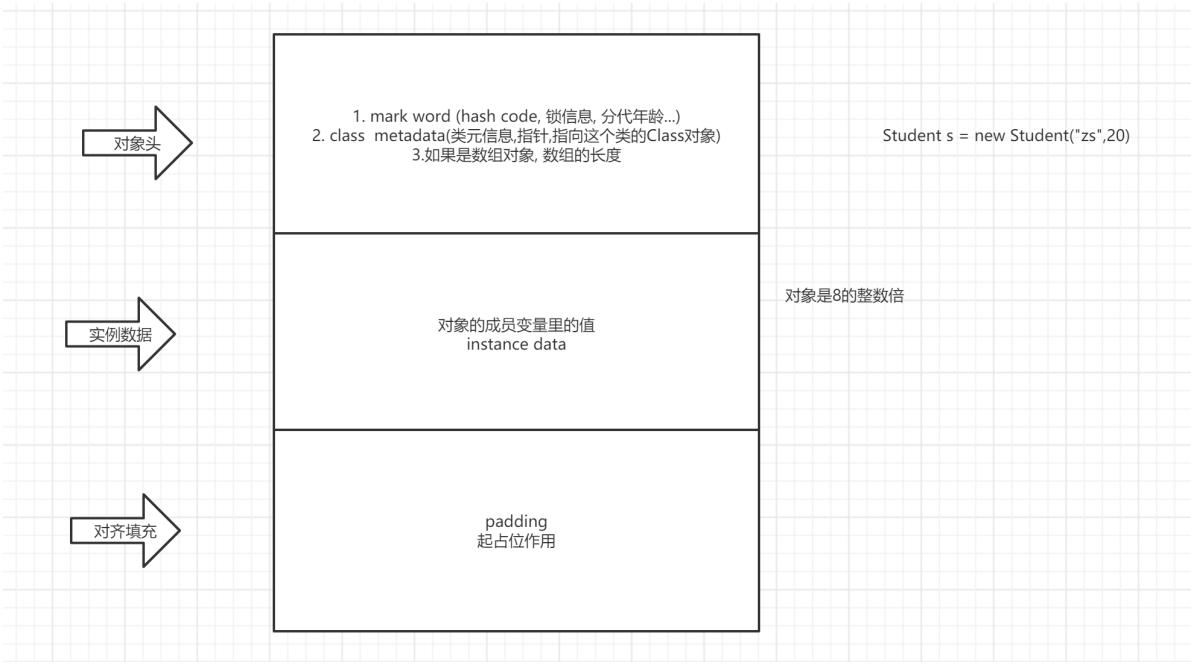
```
public class Demo4 {
    // 定义一把锁
    public static final Object OBJECT = new Object();
    public static void main(String[] args) {
        // 创建并启动一个线程
        new Thread(()->{
            // 使用同步代码块
            synchronized (OBJECT) {
                try {
                    System.out.println("t1访问到了sync");
                    TimeUnit.SECONDS.sleep(10);
                    System.out.println("t1睡醒了,退出sync");
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
        },"t1").start();
        // main休眠1s  为了让第一个线程先执行
        try {
            TimeUnit.SECONDS.sleep(1);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        // 创建并启动一个线程
        new Thread(()->{
            System.out.println("t2已经执行了");
            // 使用sync
            synchronized (OBJECT) {
                System.out.println("t2进入sync");
            }
        },"t2").start();
    }
}
```

出现异常会释放锁

```
package _22thread03.com.cskaoyan._03sync;

import java.util.concurrent.TimeUnit;

/**
 * @description:
 * @author: 景天
 * @date: 2022/7/28 14:51
 **/
/*
异常会释放锁
 */
public class Demo5 {
    // 定义一把锁
```

```
    public static final Object OBJECT = new Object();
    // 定义一个计数器
    public static int count = 0;
    public static void main(String[] args) {
        // 创建并启动一个线程
        new Thread(()->{
            synchronized (OBJECT) {
                System.out.println("t1访问sync");
                // 死循环
                while (true) {
                    count++;
                    try {
                        TimeUnit.SECONDS.sleep(1);
                    } catch (InterruptedException e) {
                        e.printStackTrace();
                    }
                    if (count == 5) {
                        System.out.println(count);
                        // 制造异常
                        System.out.println(10/0);
                    }
                }

            }
        },"t1").start();
        // main休眠1s 保证第一个线程先执行
        try {
            TimeUnit.SECONDS.sleep(1);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }

        // 创建并启动一个线程
        new Thread(()->{
            System.out.println("t2已经执行了");
            synchronized (OBJECT) {
                System.out.println("t2访问sync");
            }
        },"t2").start();
    }
}
```

1个对象的内存布局



2条字节码指令(monitorenter/monitorexit)

monitorenter

Operation

Enter monitor for object

Format

```
monitorenter
```

Forms

monitorenter = 194 (0xc2)

Operand Stack

..., *objectref* →

...

Description

The *objectref* must be of type `reference`.

Each object is associated with a monitor. A monitor is locked if and only if it has an owner. The thread that executes *monitorenter* attempts to gain ownership of the monitor associated with *objectref*, as follows:

- If the entry count of the monitor associated with *objectref* is zero, the thread enters the monitor and sets its entry count to one. The thread is then the owner of the monitor.
- If the thread already owns the monitor associated with *objectref*, it reenters the monitor, incrementing its entry count.
- If another thread already owns the monitor associated with *objectref*, the thread blocks until the monitor's entry count is zero, then tries again to gain ownership.

monitorexit

Operation

Exit monitor for object

Format

```
monitorexit
```

Forms

monitorexit = 195 (0xc3)

Operand Stack

..., *objectref* →

...

Description

The *objectref* must be of type `reference`.

The thread that executes *monitorexit* must be the owner of the monitor associated with the instance referenced by *objectref*.

The thread decrements the entry count of the monitor associated with *objectref*. If as a result the value of the entry count is zero, the thread exits the monitor and is no longer its owner. Other threads that are blocking to enter the monitor are allowed to attempt to do so.

Lock

文档示例

```
Lock l = ...;
l.lock();
try {
    // access the resource protected by this lock
} finally {
    l.unlock();
}
```

基本使用

void	lock() 获取锁。
void	unlock() 释放锁

ReentrantLock可重入锁

```
package _22thread03.com.cskaoyan._04lock;

import java.util.concurrent.locks.Lock;
import java.util.concurrent.locks.ReentrantLock;

/**
 * @description:
 * @author: 景天
 * @date: 2022/7/28 11:34
 **/
/*
使用同步代码块
 */
public class Demo {
    public static void main(String[] args) {
        //创建线程对象并启动
        SellWindow sellWindow = new SellWindow();
        new Thread(sellWindow, "A").start();
        new Thread(sellWindow, "B").start();
        new Thread(sellWindow, "C").start();
    }
}

class SellWindow implements Runnable {

    // 定义一把锁
    Lock lock = new ReentrantLock();
    // 定义票
    int tickets = 100;

    // run
```

```
    @Override
    public void run() {
        // 模拟卖票过程
        while (true) {
            // 使用lock
            // 获取锁
            lock.lock();
            try {
                if (tickets > 0) {
                    try {
                        Thread.sleep(100);
                    } catch (InterruptedException e) {
                        e.printStackTrace();
                    }
                    System.out.println(Thread.currentThread().getName()
                            + "卖了第" + (tickets--) + "票");

                }

            } finally {
                // 释放锁
                lock.unlock();
            }
        }

    }

}
```

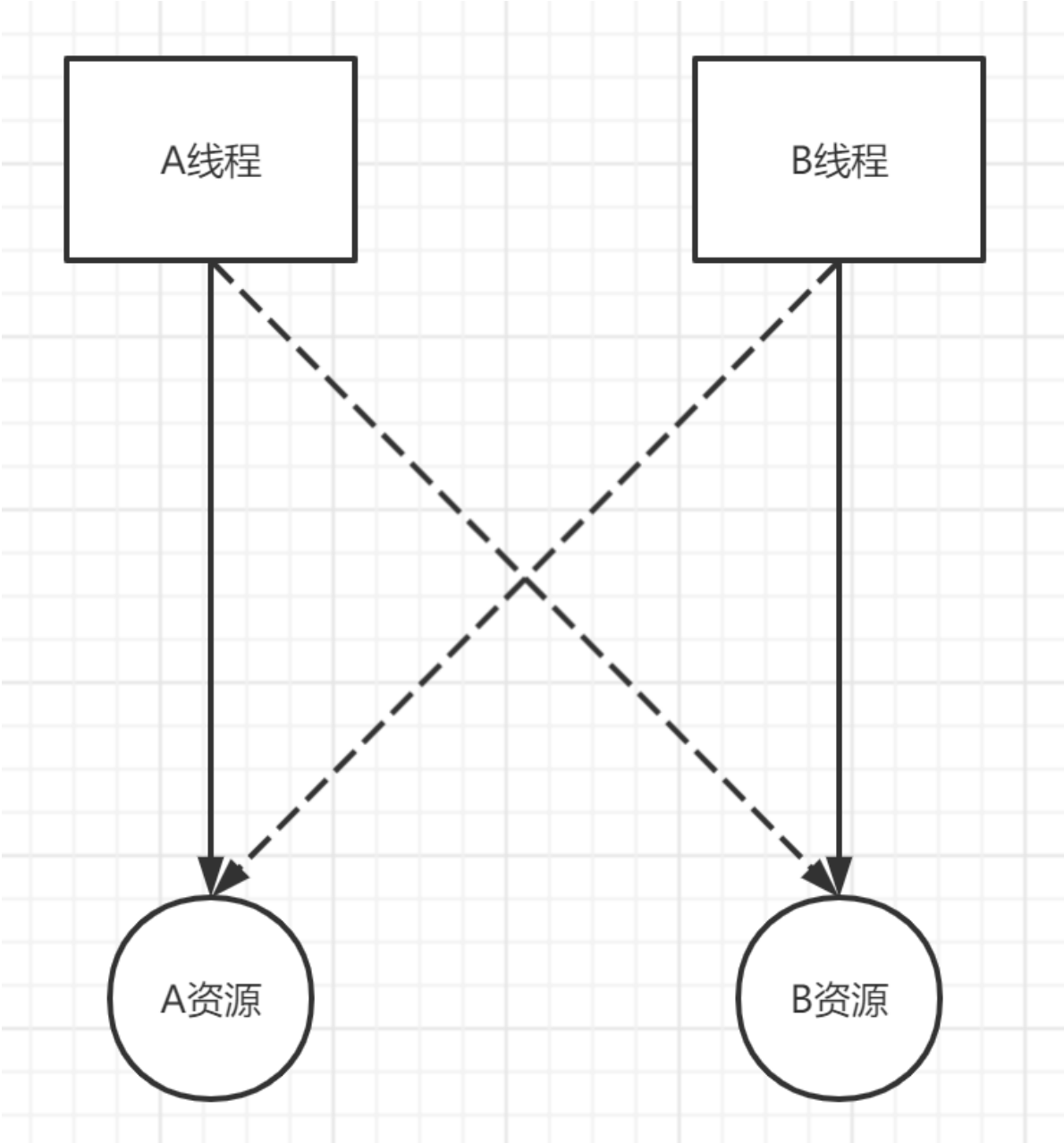
synchronized VS Lock

- sync是一把隐式的锁, 看不到加锁 释放锁的过程, 是jvm自动完成的, lock是一把真正的锁, 能够看到加锁释放锁的过程(API层面的)

死锁

什么是死锁

2个或以上的线程争抢资源而造成的互相等待的现象



死锁产生的场景

一般出现在同步代码块嵌套

```
synchronized(objA){
    synchronized(objB){

    }
}
```

```
package _22thread03.com.cskaoyan._05dielock;

/**
 * @description:
 * @author: 景天
 * @date: 2022/7/28 16:01
 **/
/*
模拟死锁场景
```

```
 */
public class Demo {
    public static void main(String[] args) {
        //创建线程并启动
        new Thread(new DieLock(true)).start();
        new Thread(new DieLock(false)).start();
    }
}
// 定义一个锁类
class MyLock{
    // 定义2把锁
    public static final Object objA = new Object();
    public static final Object objB = new Object();
}

class DieLock implements Runnable{
    // 定义成员变量
    boolean flag;

    public DieLock(boolean flag) {
        this.flag = flag;
    }

    @Override
    public void run() {
        if (flag) {
            // 同步代码块嵌套
            // 假设A线程先执行
            synchronized (MyLock.objA) {
                // A线程进来
                System.out.println("if A");
                // A想要访问里面的sync 需要B锁
                // 发生了线程切换 B线程执行
                synchronized (MyLock.objB) {
                    System.out.println("if B");
                }
            }
        }else {
            // B线程执行 B线程持有B锁
            synchronized (MyLock.objB) {
                // B线程进来
                System.out.println("else B");
                // B想要访问里面的sync
                synchronized (MyLock.objA) {
                    System.out.println("else A");
                }
            }
        }
    }
}
```

怎么解决死锁

方式一: 更改加锁顺序

```
package _22thread03.com.cskaoyan._05dielock;

/**
 * @description:
 * @author: 景天
 * @date: 2022/7/28 16:01
 **/
/*
模拟死锁场景
 */
public class Demo2 {
    public static void main(String[] args) {
        //创建线程并启动
        new Thread(new DieLock2(true)).start();
        new Thread(new DieLock2(false)).start();
    }
}

// 定义一个锁类
class MyLock2{
    // 定义2把锁
    public static final Object objA = new Object();
    public static final Object objB = new Object();
}

class DieLock2 implements Runnable{
    // 定义成员变量
    boolean flag;

    public DieLock2(boolean flag) {
        this.flag = flag;
    }

    @Override
    public void run() {
        if (flag) {
            synchronized (MyLock2.objA) {
                System.out.println("if A");
                synchronized (MyLock2.objB) {
                    System.out.println("if B");
                }
            }
        }else {
            synchronized (MyLock2.objA) {
                System.out.println("else B");
                synchronized (MyLock2.objB) {
                    System.out.println("else A");
                }
            }
        }
    }
```

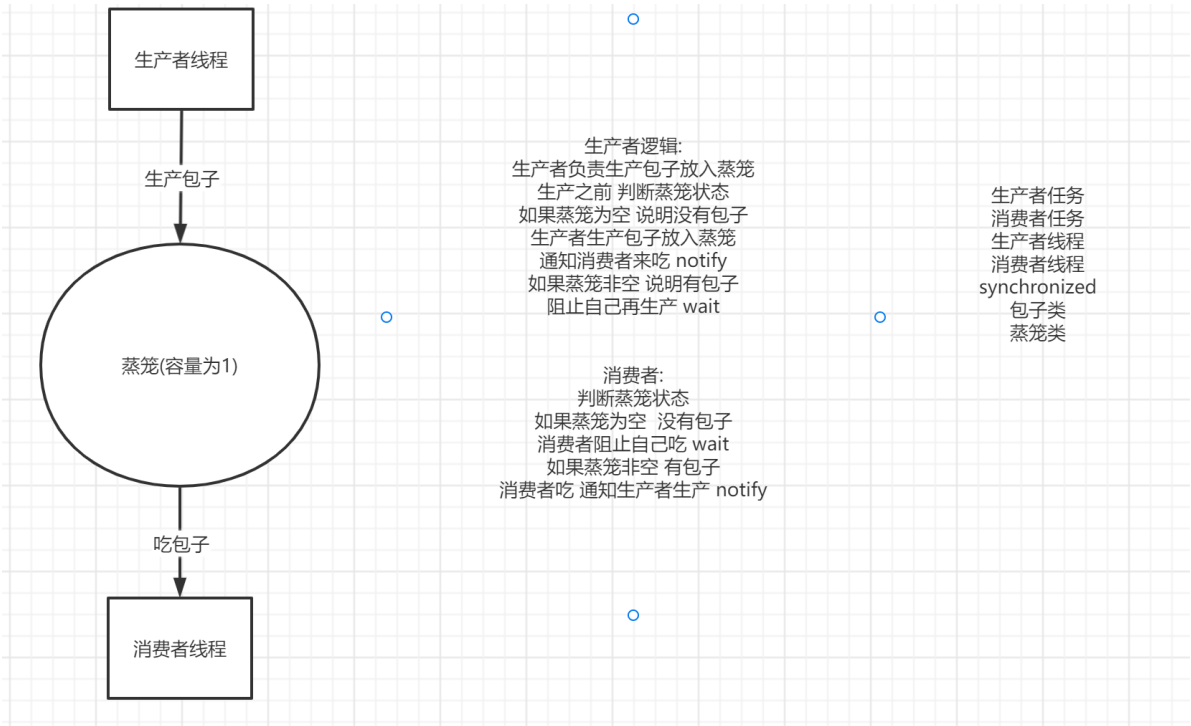
```
}
```

方式二:再加一把锁(变成原子操作)

```
package _22thread03.com.cskaoyan._05dielock;

/**
 * @description:
 * @author: 景天
 * @date: 2022/7/28 16:01
 **/
/*
解决死锁,再加一把锁
 */
public class Demo3 {
    public static void main(String[] args) {
        //创建线程并启动
        new Thread(new DieLock3(true)).start();
        new Thread(new DieLock3(false)).start();
    }
}

// 定义一个锁类
class MyLock3{
    // 定义2把锁
    public static final Object objA = new Object();
    public static final Object objB = new Object();
    public static final Object objC = new Object();

}

class DieLock3 implements Runnable{
    // 定义成员变量
    boolean flag;

    public DieLock3(boolean flag) {
        this.flag = flag;
    }

    @Override
    public void run() {
        if (flag) {
            synchronized (MyLock3.objC) {
                synchronized (MyLock3.objA) {
                    // A线程进来
                    System.out.println("if A");
                    // A想要访问里面的sync 需要B锁
                    // 发生了线程切换 B线程执行
                    synchronized (MyLock3.objB) {
                        System.out.println("if B");
                    }
                }
```

```
            }
            // 同步代码块嵌套
            // 假设A线程先执行
        }else {
            synchronized (MyLock3.objC) {
                synchronized (MyLock3.objB) {
                    // B线程进来
                    System.out.println("else B");
                    // B想要访问里面的sync
                    synchronized (MyLock3.objA) {
                        System.out.println("else A");
                    }
                }

            }
            // B线程执行 B线程持有B锁
        }
    }
}
```

生产者消费者模型



v1 使用同步代码块

```
package _22thread03.com.cskaoyan._07producer_consumer.v1;

/**
 * @description:
 * @author: 景天
 * @date: 2022/7/28 17:46
 **/
```

```java
// 蒸笼类
public class Box {
    // 定义成员变量
    Food food;

    // 定义生产包子的方法  生产者执行
    public void setFood(Food newFood) {
        // 表示包子被生产?
        food = newFood;
        System.out.println(Thread.currentThread().getName()+
                "生产了"+food);

    }

    // 定义吃包子的方法     消费者执行
    public void eatFood() {
        // 表示包子被吃
        System.out.println(Thread.currentThread().getName()+
                "吃了"+food);
        food = null;
    }

    // 定义判断蒸笼状态的方法
    public boolean isEmpty() {
        return food == null;
        // true  ---> 空
        // false ---> 非空
    }

}

// 包子类
class Food{
    String name;
    int price;

    public Food(String name, int price) {
        this.name = name;
        this.price = price;
    }

    @Override
    public String toString() {
        return "Food{" +
                "name='" + name + '\'' +
                ", price=" + price +
                '}';
    }
}



package _22thread03.com.cskaoyan._07producer_consumer.v1;

/**
```

```
 * @description: 消费者任务
 * @author: 景天
 * @date: 2022/7/28 17:52
 **/

public class ConsumerTask implements Runnable {
    // 定义成员变量
    Box box;

    public ConsumerTask(Box box) {
        this.box = box;
    }

    @Override
    public void run() {
        // 吃包子
        while (true) {
            // sync
            synchronized (box) {
                //判断蒸笼状态
                if (box.isEmpty()) {
                    //如果蒸笼为空  没有包子
                    //消费者阻止自己吃 wait
                    try {
                        box.wait();
                    } catch (InterruptedException e) {
                        e.printStackTrace();
                    }
                }else{
                    //如果蒸笼非空 有包子
                    //消费者吃 通知生产者生产 notify
                    box.eatFood();
                    box.notify();
                }
            }
        }
    }
}




package _22thread03.com.cskaoyan._07producer_consumer.v1;

import java.util.Random;

/**
 * @description: 生产者任务'
 * @author: 景天
 * @date: 2022/7/28 17:52
 **/

public class ProducerTask implements Runnable{
    // 成员变量
    Box box;
```

```
    Food[] foods = {new Food("酱肉包", 2),
            new Food("韭菜鸡蛋包", 1),
            new Food("生蚝羊腰包", 10)};
    Random random = new Random();

    public ProducerTask(Box box) {
        this.box = box;
    }

    @Override
    public void run() {
        // 生产包子
        while (true) {
            // sync
            synchronized (box) {
                // 生产之前 判断蒸笼状态
                if (box.isEmpty()) {
                    //如果蒸笼为空 说明没有包子
                    //生产者生产包子放入蒸笼
                    int index = random.nextInt(foods.length);
                    box.setFood(foods[index]);
                    //通知消费者来吃 notify
                    box.notify();
                }else{
                    //如果蒸笼非空 说明有包子
                    //阻止自己再生产 wait
                    try {
                        box.wait();
                    } catch (InterruptedException e) {
                        e.printStackTrace();
                    }
                }
            }
        }
    }
}



package _22thread03.com.cskaoyan._07producer_consumer.v1;

/**
 * @description:
 * @author: 景天
 * @date: 2022/7/28 17:46
 **/

public class Demo {
    public static void main(String[] args) {
        // 创建蒸笼对象
        Box box = new Box();
        // 创建生产者任务
        ProducerTask producerTask = new ProducerTask(box);
```

```
        // 创建消费任务
        ConsumerTask consumerTask = new ConsumerTask(box);

        // 创建生产者线程
        Thread t1 = new Thread(producerTask);

        // 创建消费者线程

        Thread t2 = new Thread(consumerTask);
        t1.setName("生产者");
        t2.setName("消费者");
        // start
        t1.start();
        t2.start();
    }
}
```

v2 使用同步方法

```
package _22thread03.com.cskaoyan._07producer_consumer.v2;

/**
 * @description:
 * @author: 景天
 * @date: 2022/7/29 9:47
 **/

public class Box {
    // 定义成员变量
    Food food;

    // 定义2个方法
    // 生产包子的方法 只有生产者执行
    public synchronized void setFood(Food newFood) {
        // 生产之前 判断蒸笼状态
        if (food == null) {
            //如果蒸笼为空 说明没有包子
            //生产者生产包子放入蒸笼
            food = newFood;
            System.out.println(Thread.currentThread().getName()+
                    "生产了"+food);
            //通知消费者来吃 notify
            this.notify();

        }else{
            //如果蒸笼非空 说明有包子
            //阻止自己再生产 wait
            try {
                this.wait();
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
```

```
        }
    }

    // 吃包子的方法 只有消费者执行
    public synchronized void eatFood() {
        //判断蒸笼状态
        if (food == null) {
            //如果蒸笼为空  没有包子
            //消费者阻止自己吃 wait
            try {
                this.wait();
            } catch (InterruptedException e) {
                e.printStackTrace();
            }

        }else{
            //如果蒸笼非空 有包子
            System.out.println(Thread.currentThread().getName()+
                    "吃了"+food);
            food = null;
            //消费者吃 通知生产者生产 notify
            this.notify();

        }
    }
}

class Food{
    String name;
    int price;

    public Food(String name, int price) {
        this.name = name;
        this.price = price;
    }

    @Override
    public String toString() {
        return "Food{" +
                "name='" + name + '\'' +
                ", price=" + price +
                '}';
    }
}



package _22thread03.com.cskaoyan._07producer_consumer.v2;

/**
 * @description:
 * @author: 景天
 * @date: 2022/7/29 9:48
 **/
```

```
public class ConsumerTask implements Runnable {
    Box box;

    public ConsumerTask(Box box) {
        this.box = box;
    }

    @Override
    public void run() {
        // 只吃包子
        while (true) {
            box.eatFood();
        }
    }
}
```

```
package _22thread03.com.cskaoyan._07producer_consumer.v2;

import java.util.Random;

/**
 * @description: 生产者任务
 * @author: 景天
 * @date: 2022/7/29 9:48
 **/

public class ProducerTask implements Runnable{
    // 成员变量
    Box box;

    Food[] foods = {new Food("杭州小笼包", 5),
            new Food("天津狗不理", 40), new Food("开封灌汤包", 20)};
    Random random = new Random();

    public ProducerTask(Box box) {
        this.box = box;
    }


    @Override
    public void run() {
        // 只生产包子
        while (true) {
            int index = random.nextInt(foods.length);
            box.setFood(foods[index]);

        }
    }
}
```

```
package _22thread03.com.cskaoyan._07producer_consumer.v2;


/**
 * @description:
 * @author: 景天
 * @date: 2022/7/28 17:46
 **/

public class Demo {
    public static void main(String[] args) {
        // 创建蒸笼对象
        Box box = new Box();
        // 创建生产者任务
        ProducerTask producerTask = new ProducerTask(box);

        // 创建消费任务
        ConsumerTask consumerTask = new ConsumerTask(box);

        // 创建生产者线程
        Thread t1 = new Thread(producerTask);

        // 创建消费者线程

        Thread t2 = new Thread(consumerTask);
        t1.setName("生产者");
        t2.setName("消费者");
        // start
        t1.start();
        t2.start();
    }
}
```

如果有多个生产者 多个消费者会出现什么情况?

会出现卡顿的现象----> 原因就是全都wait

```
c1,c2,p1,p2  都start
c1抢到 > 进入sync , 空的, wait ,释放锁
c2抢到  > 进入sync,空的, wait , 释放锁
p1抢到  > 进入sync, 空的可以生产, notify唤醒c1 , 退出sync,释放锁
p1又抢到 > 进入sync, 非空, wait, 释放锁
p2抢到> 进入sync, 非空, wait 释放锁
c1抢到 > 进入sync, 非空,吃, notify 唤醒c2 ,退出sync 释放锁
c2 > 进入 sync, 空, wait 释放锁
c1 > 进入 sync, 空 , wait
到此 所有线程都wait

怎么解决?
使用notifyAll()
```

线程间通信

wait与notify机制

wait与notify机制

拥有相同锁的线程才可以实现wait/notify机 制，所以后面的描述中都是假定操作同一个锁。

- wait()方法是Object类的方法，它的作用是使当前执行wait()方法的线程**等待**，在wait()所在的 代码行处暂停执行，并释放锁，直到接到通知被唤醒。在调用wait()之前，线程必须获得锁对象，即只能在同步方法或同步块中调用wait()方法。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语句捕捉异常。
- notify()方法要在同步方法或同步块中调用， 即在调用前，线程必须获得锁对象，如果调用notify()时没有持有适当的锁，则会抛IllegalMonitorStateException。该方法用来通知那 些可能等待该锁对象的其他线程，如果有多个线程等待，则唤醒其中随机一个线程，并使该线程重新获取锁。
- 需要说明的是，执行notify()方法后，当 前线程不会马上释放该锁，因wait方法而阻塞的线程也 并不能马上获取该对象锁，要等到执行notify()方 法的线程将程序执行完，也就是退出synchronized同步区域后，当前线程才会释放锁，而处于阻塞状 态的线程才可以获取该对象锁。当第一个获得了该对象锁的wait线程运行完毕后，它会释放该对 象锁，此时如果没有再次使用notify语句，那么其他呈阻塞状态的线程因为没有得到通知，会继续 处于阻塞状态。

总结：wait()方法使线程暂停运行，而notify() 方法通知暂停的线程继续运行

wait()

1. 阻塞功能：
   当在某线程中，对象上.wait()， 在哪个线程中调用wait()， 导致哪个线程处于阻塞状态
   当某线程，因为调用执行某对象的wait(),而处于阻塞状态，我们说，该线程在该对象上阻塞。
2. 唤醒条件
   当某线程，因为某对象A的wait()， 而处于阻塞状态时，如果要唤醒该线程，只能在其他线程中，再同一个对象(即对象A)上调用其notify()或notifyAll()
   即在线程的阻塞对象上，调用notify或notifyAll方法，才能唤醒，在该对象上阻塞的线程
3. 运行条件
   当前线程必须拥有此对象监视器。
   监视器：指synchronized代码块中的锁对象
   即我们只能在，当前线程所持有的synchronized代码块中的，锁对象上调用wait方法，才能正常执行
   如果没有锁对象就会有这样一个异常 IllegalMonitorStateException
4. 执行特征
   a.该线程发布(release)对此监视器的所有权
   b.等待(阻塞)
   注意：Thread的sleep方法，执行的时候：
   该线程不丢失任何监视器的所属权

使用条件

必须要有用锁对象,跟sync结合

```
package _22thread03.com.cskaoyan._06waitnotify;

/**
 * @description:
 * @author: 景天
 * @date: 2022/7/28 16:32
 **/

/*
wait使用条件
 */
public class Demo {
    public static void main(String[] args) {
        Object o = new Object();
        // wait
        try {
            o.wait();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}
```

执行特点

```
package _22thread03.com.cskaoyan._06waitnotify;

/**
 * @description:
 * @author: 景天
 * @date: 2022/7/28 16:32
 **/

/*
wait使用条件
 */
public class Demo2 {
    public static void main(String[] args) {
        Object o = new Object();
        // wait
        // sync
        synchronized (o) {
            try {
                System.out.println("wait before");
                // 在锁对象上调用wait
                o.wait();
                System.out.println("wait after");
```

```
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            /*
            执行结果
            wait before
            分析: 执行wait后导致当前线程(main) 处于阻塞状态
            Q:
            wait after输出?
            需要再别的线程 同一个锁对象是使用notify  notifyAll唤醒

             */

        }
    }
}
```

验证wait释放锁

```
package _22thread03.com.cskaoyan._06waitnotify;

import java.util.concurrent.TimeUnit;

/**
 * @description:
 * @author: 景天
 * @date: 2022/7/28 16:38
 **/

/*
验证wait会释放锁
 */
public class Demo3 {
    // 定义一把锁
    public static final Object OBJECT = new Object();
    public static void main(String[] args) {
        // 创建并启动一个线程
        new Thread(()->{
            // sync
            synchronized (OBJECT) {
                System.out.println("t1访问了sync");
                try {
                    TimeUnit.SECONDS.sleep(10);
                    System.out.println("wait before");
                    OBJECT.wait();
                    System.out.println("wait after");
                    /*
                    Q:怎么让程序执行完?需要notify
                     */
                } catch (InterruptedException e) {
                    e.printStackTrace();
```

```
                }
            }
        },"t1").start();
        // main休眠1s
        try {
            TimeUnit.SECONDS.sleep(1);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }

        // 创建并启动一个线程
        new Thread(()->{
            System.out.println("t2已经执行了");
            // sync
            synchronized (OBJECT) {
                System.out.println("t2访问sync");
            }
        },"t2").start();
    }
}
```

wait与notify的基本使用

```
package _22thread03.com.cskaoyan._06waitnotify;

import java.util.concurrent.TimeUnit;

/**
 * @description:
 * @author: 景天
 * @date: 2022/7/28 16:47
 **/
/*
wait notify基本使用
 */
public class Demo4 {
    // 定义一把锁
    public static final Object OBJECT = new Object();
    public static void main(String[] args) {
        // 创建并启动一个线程
        new Thread(()->{
            // sync
            synchronized (OBJECT) {
                // wait
                try {
                    TimeUnit.SECONDS.sleep(10);
                    System.out.println("wait before");
                    OBJECT.wait();
                    System.out.println("wait after");

                } catch (InterruptedException e) {
                    e.printStackTrace();
```

```
                }
            }
        },"t1").start();
        // main休眠1s
        try {
            TimeUnit.SECONDS.sleep(1);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        // 创建并启动一个线程
        new Thread(()->{
            System.out.println("t2已经执行了");
            // sync
            synchronized (OBJECT) {
                System.out.println("t2访问sync");
                System.out.println("notify before");
                OBJECT.notify();
                System.out.println("notify after");

            }
        },"t2").start();
    }
}
```

练习

创建2个线程 A B

A线程打印1, B线程打印2, A线程大印3, B打印4.... 100

```
package _22thread03.com.cskaoyan._06waitnotify;

/**
 * @description:
 * @author: 景天
 * @date: 2022/7/28 17:18
 **/
/*
练习

创建2个线程 A B

A线程打印1, B线程打印2, A线程大印, B打印4.... 100
 */
public class Ex {
    // 定义一把锁
    public static final Object OBJECT = new Object();
    public static void main(String[] args) {
        // 创建并启动一个线程 A线程打印奇数
        new Thread(()->{
            synchronized (OBJECT) {
                for (int i = 1; i < 100; i+=2) {
                    // 先唤醒另一个线程
                    OBJECT.notify();
                    System.out.println(Thread.currentThread().getName()+
```

```
                            "-----"+i);
                    // 阻止自己打印
                    try {
                        OBJECT.wait();
                    } catch (InterruptedException e) {
                        e.printStackTrace();
                    }
                }
                // notify
                OBJECT.notify();

            }

        },"A").start();

        // 创建并启动一个线程 B线程打印偶数
        new Thread(()->{
            synchronized (OBJECT) {
                for (int i = 2; i <= 100; i+=2) {
                    // 先唤醒另一个线程
                    OBJECT.notify();
                    System.out.println(Thread.currentThread().getName()+
                            "-----"+i);
                    // 阻止自己打印
                    try {
                        OBJECT.wait();
                    } catch (InterruptedException e) {
                        e.printStackTrace();
                    }
                }
                // notify
                OBJECT.notify();

            }

        },"B").start();
    }
}
```

sleep VS wait

```
Thread.sleep VS Object.wait()

1. 所属不同：
   a. sleep定义在Thread类，静态方法
   b. wait定义在 Object类中，非静态方法

2. 唤醒条件不同
   a. sleep: 休眠时间到
   b. wait: 在其他线程中，在同一个锁对象上，调用了notify或notifyAll方法

3. 使用条件不同：
```

a. sleep 没有任何前提条件
b. wait()，必须当前线程，持有锁对象，锁对象上调用wait()

4. 休眠时，对锁对象的持有，不同：（最最核心的区别）
a. 线程因为sleep方法而处于阻塞状态的时候，在阻塞的时候不会放弃对锁的持有
b. 但是wait()方法，会在阻塞的时候，放弃锁对象持有

sleep wait对于锁的持有

```
package _22thread03.com.cskaoyan._06waitnotify;

/**
 * @description:
 * @author: 景天
 * @date: 2022/7/28 17:32
 **/
/*
sleep wait对于锁的持有
 */
public class Demo5 {
    // 定义一把锁
    public static final Object OBJECT = new Object();
    public static void main(String[] args) {
        // 创建并启动2个线程
        // 验证sleep
        //new ThreadA().start();
        //new ThreadA().start();

        // 验证wait
        new ThreadB().start();
        new ThreadB().start();
    }
}

// 验证sleep方法
class ThreadA extends Thread{
    @Override
    public void run() {
        synchronized (Demo5.OBJECT) {
            // sleep
            try {
                System.out.println(getName()+"sleep before");
                Thread.sleep(10000);
                System.out.println(getName()+"sleep after");

            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
}

// 验证wait方法
class ThreadB extends Thread{
    @Override
```

```
    public void run() {
        synchronized (Demo5.OBJECT) {
            // wait
            try {
                System.out.println(getName()+"wait before");
                Demo5.OBJECT.wait();
                System.out.println(getName()+"wait after");

            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
}
```

notify()

- 唤醒在此对象监视器上等待的单个线程。
- 如果所有线程都在此对象上等待，则会选择唤醒其中一个线程
- 选择是任意性的

notifyAll()

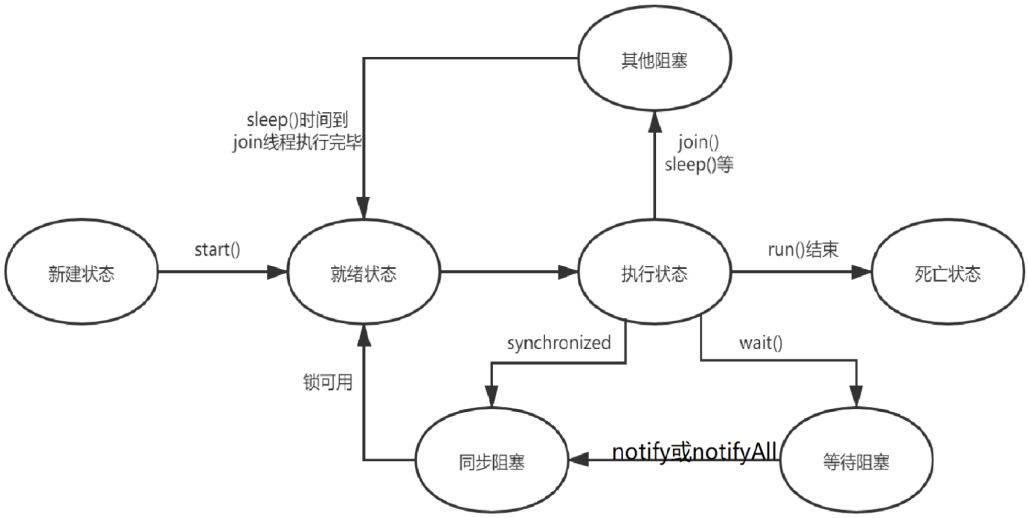
唤醒在此对象监视器上等待的所有线程。

为什么wait,notify,notifyAll方法不定义在Thread类中?

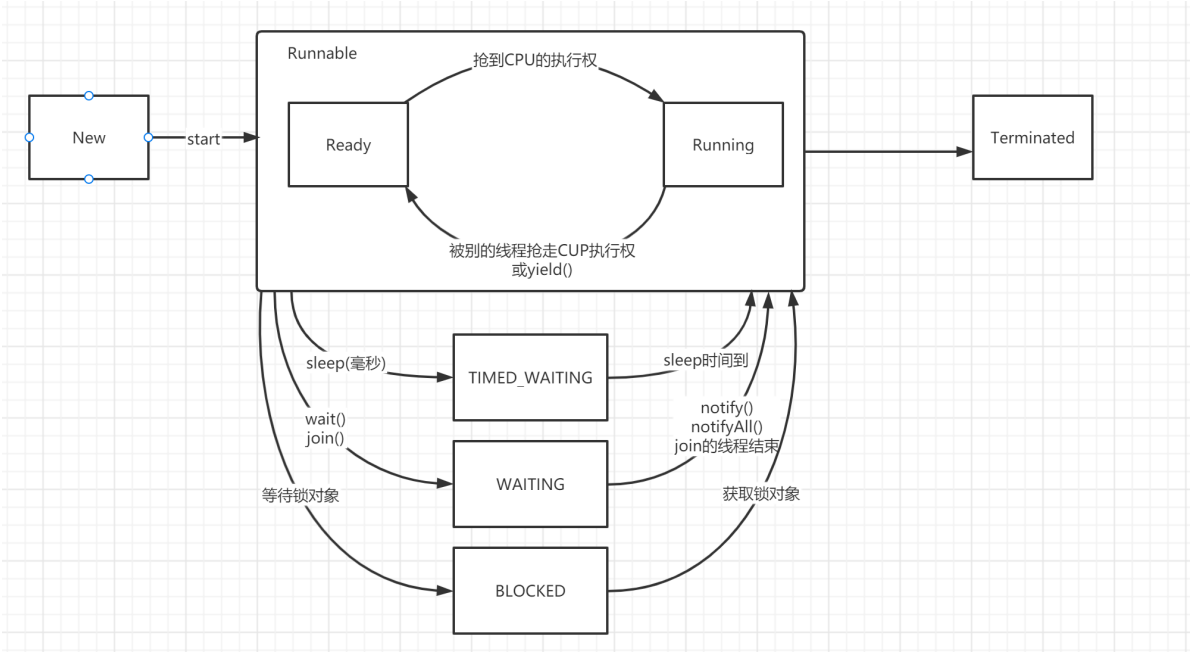
任意java对象都能充当锁的角色

完整的线程状态转换图

理论层面



代码层面



多线程工具

线程池

```
Thread t = new Thread();
t.start();
```

3中线程池

Executors : 工具类 产生线程池

ExecutorService: 代表线程池对象

```
//JDK5提供了一Executors来产生线程池，有如下方法：
ExecutorService newCachedThreadPool()
// 特点:
// 1.会根据需要创建新线程，也可以自动删除，60s处于空闲状态的线程
// 2.线程数量可变，立马执行提交的异步任务（异步任务：在子线程中执行的任务）
ExecutorService newFixedThreadPool(int nThreads)
// 特点:
// 1.线程数量固定
// 2.维护一个无界队列（暂存已提交的来不及执行的任务）
// 3.按照任务的提交顺序，将任务执行完毕
ExecutorService newSingleThreadExecutor()
// 特点:
// 1.单个线程
// 2.维护了一个无界队列（暂存已提交的来不及执行的任务）
// 3.按照任务的提交顺序，将任务执行完毕
```

基本使用

Future<?> submit(Runnable task)

Future submit(Callable task)

Future

`Future` 表示异步计算的结果 Runnable类型的任务没有返回值 Callable类型的任务是有返回值的

Future用来存储Callable类型任务的返回值

V	get() 如有必要，等待计算完成，然后获取其结果。

提交Runnable类型的任务

```
package _22thread03.com.cskaoyan._08threadpool;

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

/**
 * @description:
 * @author: 景天
 * @date: 2022/7/29 11:15
 **/

public class Demo {
    public static void main(String[] args) {
        // 创建线程池对象
        ExecutorService pool = Executors.newCachedThreadPool();
        // 向线程池提交任务
        // submit(Runnable task)
        pool.submit(new MyTask());
        pool.submit(new MyTask());
    }
}

class MyTask implements Runnable{

    @Override
    public void run() {
        for (int i = 0; i < 10; i++) {
            System.out.println(Thread.currentThread().getName()+
                    "------"+i);
        }
    }
}
```

提交Callable类型的任务

```
package _22thread03.com.cskaoyan._08threadpool;

import java.util.concurrent.*;
```

```java
/**
 * @description:
 * @author: 景天
 * @date: 2022/7/29 11:19
 **/

public class Demo2 {
    public static void main(String[] args) throws ExecutionException,
InterruptedException {
        // 创建线程池对象
        ExecutorService pool = Executors.newCachedThreadPool();
        // 向线程池提交任务
        // submit(Callable task)
        Future<String> future = pool.submit(new MyCallableTask());
        // 获取返回值 通过get()方法
        System.out.println("get before");
        String s = future.get();
        System.out.println("get after");

        System.out.println(s);
    }
}

class MyCallableTask implements Callable<String> {
    @Override
    public String call() throws Exception {
        for (int i = 0; i < 10; i++) {
            System.out.println(i);
        }
        // sleep
        TimeUnit.SECONDS.sleep(10);
        return "哈哈哈哈";
    }

    //@Override
    //public Object call() throws Exception {
    //    return null;
    //}
}
```

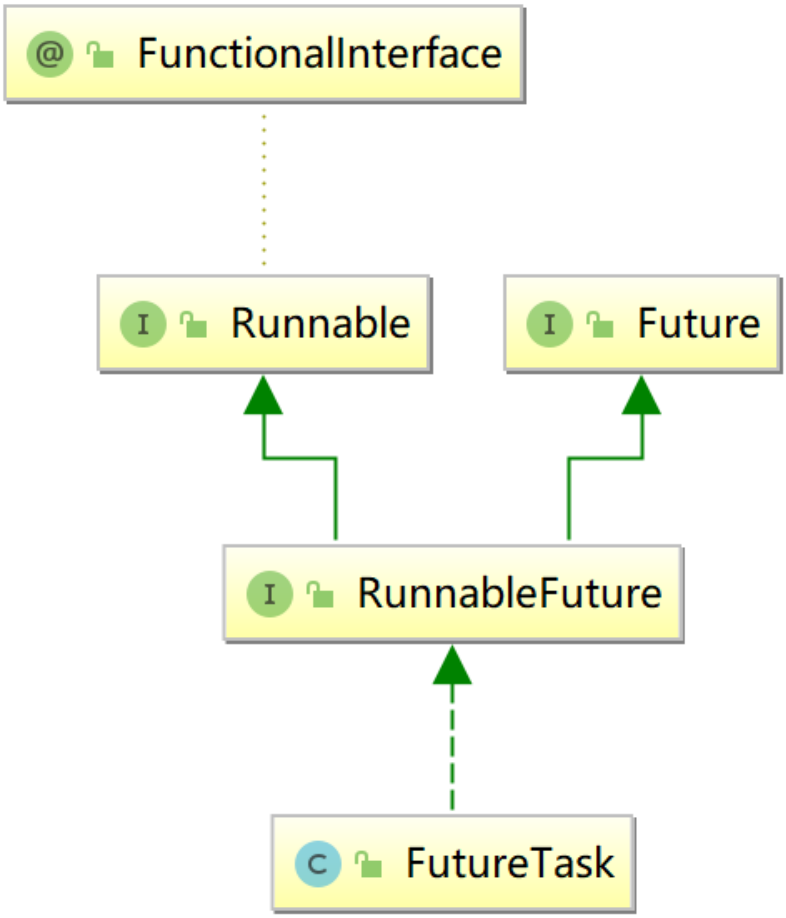
关闭线程池

void	**shutdown() 启动一次顺序关闭，执行以前提交的任务，但不接受新任务。**
List	shutdownNow() 试图停止所有正在执行的活动任务，暂停处理正在等待的任务，并返回等待执行的任务列表。

多线程的实现方式三:实现Callable接口

不使用线程池,借助于FutureTask

FutureTask = Future+Task（既可以作为任务执行, 又可以存储返回值)



构造方法

FutureTask(Callable callable)　　创建一个 FutureTask，一旦运行就执行给定的 Callable。

Demo

```
package _22thread03.com.cskaoyan._08threadpool;

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

/**
 * @description:
 * @author: 景天
 * @date: 2022/7/29 11:34
 **/
/*
使用Callable FutureTask
 */
```

```
public class Demo3 {
    public static void main(String[] args) throws ExecutionException,
InterruptedException {
        // 创建FutureTask对象
        FutureTask<String> futureTask = new FutureTask<>(new MyCallable());

        // 让其运行在线程中
        Thread t = new Thread(futureTask);
        t.start();

        // 获取这个返回值
        String s = futureTask.get();
        System.out.println(s);
    }
}

class MyCallable implements Callable<String>{

    @Override
    public String call() throws Exception {
        System.out.println("111111");
        return "call 执行完毕";
    }
}
```

练习:

使用Callable 做运算 返回运算结果

创建2个线程 1个线程计算1+2+3...100 另一个线程计算1+2+3+...200

```
package _22thread03.com.cskaoyan._08threadpool;

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

/**
 * @description:
 * @author: 景天
 * @date: 2022/7/29 11:40
 **/
/*
练习:

使用Callable 做运算 返回运算结果

创建2个线程  1个线程计算1+2+3...100     另一个线程计算1+2+3+...200
 */
public class Ex {
    public static void main(String[] args) throws ExecutionException,
InterruptedException {
        // 创建2个Callable类型的任务
```

```
        // 创建2个FutureTask
        FutureTask<Integer> futureTask1 = new FutureTask<>(new
MyCallable2(100));
        FutureTask<Integer> futureTask2 = new FutureTask<>(new
MyCallable2(200));
        // 创建线程对象
        // start
        new Thread(futureTask1).start();
        new Thread(futureTask2).start();

        // get获取返回值
        Integer result1 = futureTask1.get();
        Integer result2 = futureTask2.get();
        System.out.println(result1);
        System.out.println(result2);

    }
}

class MyCallable2 implements Callable<Integer> {

    // 成员变量
    int num;

    public MyCallable2(int num) {
        this.num = num;
    }

    @Override
    public Integer call() throws Exception {
        int sum = 0;
        for (int i = 0; i <= num; i++) {
            sum += i;
        }

        return sum;
    }
}
```

Runnable VS Callable

- Runnable是run方法 Callable里面是call方法
- 最大的区别,Runnable没有返回值,Callable有返回值

定时器与定时任务

定时器Timer

一种工具，线程用其安排以后在后台线程中执行的任务。可安排任务执行一次，或者定期重复执行。

Timer的调度

```
schedule(TimerTask task, Date time)
schedule(TimerTask task, long delay, long period)
schedule(TimerTask task, Date firstTime, long period)
scheduleAtFixedRate(TimerTask task, long delay, long period)

// 追赶特性
schedule    scheduleAtFixedRate
```

定时任务TimerTask

由 Timer 安排为一次执行或重复执行的任务。

```
package _22thread03.com.cskaoyan._09timer;

import java.text.ParseException;
import java.text.SimpleDateFormat;
import java.util.Date;
import java.util.Timer;
import java.util.TimerTask;

/**
 * @description:
 * @author: 景天
 * @date: 2022/7/29 14:31
 **/

public class Demo {
    public static void main(String[] args) throws ParseException {
        // 创建定时器
        // Timer()
        // 创建一个新计时器。
        Timer timer = new Timer();
        // schedule(TimerTask task, Date time)
        SimpleDateFormat sdf = new SimpleDateFormat("yyyy-MM-dd HH:mm:ss");
        String time = "2022-07-29 14:39:20";
        Date date = sdf.parse(time);
        //timer.schedule(new MyTimerTask(), date);

        //schedule(TimerTask task, long delay, long period)
        //timer.schedule(new MyTimerTask(),5000,3000);

        //schedule(TimerTask task, Date firstTime, long period)
        //timer.schedule(new MyTimerTask(),date, 3000);

        //scheduleAtFixedRate(TimerTask task, long delay, long period)
        timer.scheduleAtFixedRate(new MyTimerTask(), 3000, 3000);
    }
```

```
}

// 定义定时任务
// 继承TimerTask
// 重写run方法
class MyTimerTask extends TimerTask{

    @Override
    public void run() {
        System.out.println("炸弹爆炸了!Boom!");
    }
}
```