java多线程学习笔记

实现多线程的两种方法

- 继承Thread类

  子类继承Thread类具备多线程能力

  启动线程：子类对象.start()

  不建议使用：避免OOP单继承局限性

  ```java
  //创建线程方式一 ：继承Thread类，重写run()方法，调用start开启线程
  public class ThreadTest01 extends Thread{
      @Override
      public void run() {
          for (int i = 0; i < 10; i++) {
              System.out.println("run"+i);
          }
      }

      public static void main(String[] args) {
          //创建线程对象
          ThreadTest01 threadTest01 = new ThreadTest01();
          //调用start开启线程
          threadTest01.start();

          for (int i = 0; i < 30; i++) {
              System.out.println("main"+i);
          }
      }
  }
  ```

- 实现Runnable接口

实现接口Runnable具有多线程能力

启动线程：传入目标对象+Thread对象.start()

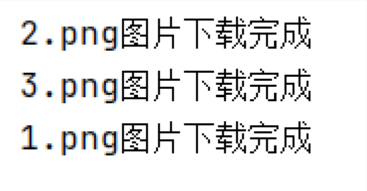
推荐使用：避免单继承局限性，方便同一个对象被多个线程使用

```java
  //创建线程方式2：实现runnable接口，重写run()方法，执行线程需要丢人runnable接口实现类，调用start方法
public class ThreadTest03 implements Runnable {
        @Override
      public void run() {
          for (int i = 0; i < 10; i++) {
```

```
            System.out.println("run" + i);
        }
    }

    public static void main(String[] args) {
      //创建runnable接口的实现对象
          ThreadTest03 threadTest03 = new ThreadTest03();
        //创建线程对象，通过线程对象来开启我们的线程
        //Thread thread = new Thread(threadTest03);
        //启动线程
        //thread.start();
        new Thread(threadTest03).start();

        for (int i = 0; i < 30; i++) {
          System.out.println("main" + i);
          }
    }
}
```

使用多线程下载图片

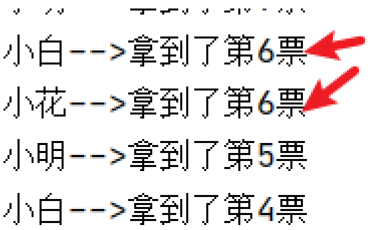
```
导入commoms-io的jar包

import org.apache.commons.io.FileUtils;

import java.io.File;
import java.io.IOException;
import java.net.URL;

//练习Thread，实现多线程同步下载图片
public class ThreadTest02 implements Runnable {
private String url;
private String name;

 public ThreadTest02(String url, String name) {
  this.url = url;
     this.name = name;
 }

 @Override
public void run() {
     WebDownloder webDownloder = new WebDownloder();
     webDownloder.downloader(url, name);
     System.out.println(name + "图片下载完成");
 }

 public static void main(String[] args) {
  ThreadTest02 thread01 = new ThreadTest02("图片URL",
"1.png");
     ThreadTest02 thread02 = new ThreadTest02("图片URL",
"2.png");
     ThreadTest02 thread03 = new ThreadTest02("图片URL",
"3.png");
     new Thread(thread01).start();
     new Thread(thread02).start();
     new Thread(thread03).start();
     //下载文件顺序并不一定是按1、2、3下载
 }
}

//下载器
class WebDownloder {
//下载方法
public void downloader(String url, String file) {
     try {
         FileUtils.copyURLToFile(new URL(url), new
File(file));
     } catch (IOException e) {
         e.printStackTrace();
         System.out.println("IO异常，downloader方法出现问题");
     }
 }
}
```



```
2.png图片下载完成
3.png图片下载完成
1.png图片下载完成
```

多线程操作同一对象

```
//多个线程同时操作同一个对象
//买车票例子
//多个线程操作同一个对象下线程不安全，数据紊乱
public class ThreadTest04 implements Runnable {
    private int ticketsNums = 10;

    @Override
    public void run() {
        while (true) {
            if (ticketsNums ⩽ 0) {
                break;
            }
            try {
                Thread.sleep(100);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            System.out.println(Thread.currentThread().getName() +
"-->拿到了第" + ticketsNums-- + "票");
        }
    }

    public static void main(String[] args) {
        ThreadTest04 threadTest04 = new ThreadTest04();
        new Thread(threadTest04, "小明").start();
        new Thread(threadTest04, "小花").start();
        new Thread(threadTest04, "小白").start();
    }
}
```



```
小白-->拿到了第6票
小花-->拿到了第6票
小明-->拿到了第5票
小白-->拿到了第4票
```

模拟龟兔赛跑

```
//模拟龟兔赛跑
public class ThreadTest05 implements Runnable {
    private static String winner;

    @Override
    public void run() {
        //模拟兔子休息
        for (int i = 0; i <= 100; i++) {
            if (Thread.currentThread().getName().equals("兔子") &&
i % 10 == 0) {
                try {
                    Thread.sleep(10);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
            //判断比赛是否结束
            boolean flag = gameOver(i);
            if (flag) {
                break;
            }
            System.out.println(Thread.currentThread().getName() +
"-->跑了" + i + "m");
        }
    }

    //判断是否完成比赛
    private boolean gameOver(int steps) {
        //判断是否有胜利者
        if (winner != null) {//已经存在胜利者
            return true;
        } else if (steps >= 100) {
            winner = Thread.currentThread().getName();
            System.out.println("winner is" + " " + winner);
            return true;
        } else {
            return false;
        }
    }

    public static void main(String[] args) {
        ThreadTest05 threadTest05 = new ThreadTest05();
        new Thread(threadTest05, "兔子").start();
        new Thread(threadTest05, "乌龟").start();
    }
}
```

实现Callable接口

```
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 result = ser.submit(t1);
6. 获取结果：boolean r1 = result.get();
7. 关闭服务：ser.shutdownNow();
```

```
 @Override
    public Boolean call() throws Exception {
        WebDownloder2 webDownloder = new WebDownloder2();
        webDownloder.downloader(url, name);
        System.out.println(name + "图片下载完成");
        return true;
    }
public static void main(String[] args) throws
ExecutionException, InterruptedException {
        //创建目标对象
        CallableTest thread01 = new CallableTest("图片URL",
"1.png");
        CallableTest thread02 = new CallableTest("图片URL",
"2.png");
```

```
        CallableTest thread03 = new CallableTest("图片URL",
"3.png");
        //创建执行服务
        ExecutorService executorService =
Executors.newFixedThreadPool(3);
        //提交执行
        Future<Boolean> r1 = executorService.submit(thread01);
        Future<Boolean> r2 = executorService.submit(thread02);
        Future<Boolean> r3 = executorService.submit(thread03);
        //获取结果
        System.out.println(r1.get());
        System.out.println(r2.get());
        System.out.println(r3.get());
        //关闭服务
        executorService.shutdown();
    }
```

Lamda表达式

- 任何接口，如果只包含唯一一个抽象方法，那么它就是一个函数式接口
- Lambda表达式只能有一行代码的情况下才能化简成为一行，如果有多行，那么就用代码块包裹
- 多个参数也可以去掉参数类型，要去掉就都去掉，必须加上括号

```
//推导Lambda表达式
public class LambdaTest {
    //3.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();    //父类（接口）的引用就能够直接
调用子类（实现类）的方法。
        like.lambda();           //接口回调

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

        // 4.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.匿名内部类，没有类的名称，必须借助接口或者父类
        like = new ILike() {
            @Override
            public void lambda() {
                System.out.println("I like lambda -- 4");
            }
        };
        like.lambda();

        //6.用Lambda简化
        like = () -> {
            System.out.println("I like lambda -- 5");
        };
        like.lambda();
    }
}

//1.定义一个函数式接口，只有一个抽象方法的接口
interface ILike {
    void lambda();
}

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

静态代理

```
//静态代理
//真实对象和代理对象都要实现同一个接口
//代理对象要代理真实对象
/*
1.代理对象可以做很多真实对象做不了的事情
2.真实对象专注做自己的事情
 */
public class StaticProxy   {
```

```java
    public static void main(String[] args) {
        new Thread(()-> System.out.println("Hello"));
        new Proxy(new Me()).Eatting();
        //对比线程Thread和Proxy，发现Thread代理Runnable
    }
}

interface Eat{
    void Eatting();
}

//真实对象，我吃饭
class Me implements Eat{

    @Override
    public void Eatting() {
        System.out.println("我正在吃饭");
    }
}

//代理对象，帮助我吃饭
class Proxy implements Eat{
    //代理谁-->真实目标角色
    private Eat target;

    public Proxy(Eat target) {
        this.target = target;   //这就是真实对象
    }

    @Override
    public void Eatting() {
        before();
        this.target.Eatting();
        after();
    }

    private void before() {
        System.out.println("吃饭前摆碗筷");
    }

    private void after() {
        System.out.println("吃饭后刷锅");
    }
}
```

synchronized同步机制

- 同步方法：修饰词synchronized用来修饰要进行同步的方法，同步监视器为this

```java
//多个线程同时操作同一个对象
//买车票例子
//多个线程操作同一个对象下线程不安全，数据紊乱
public class ThreadTest04 implements Runnable {
    private int ticketsNums = 10;
    boolean flag = true;

    @Override
    public void run() {
        while (flag) {
            try {
                Thread.sleep(100);
                SaleTickets();
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }

    public static void main(String[] args) {
        ThreadTest04 threadTest04 = new ThreadTest04();
        new Thread(threadTest04, "小明").start();
        new Thread(threadTest04, "小花").start();
        new Thread(threadTest04, "小白").start();
    }
    //加锁机制，保证线程安全
    private synchronized void SaleTickets() {
        if (ticketsNums <= 0) {
            flag = false;
            return;
        }
        System.out.println(Thread.currentThread().getName() +
"-->拿到了第" + ticketsNums-- + "票");
    }
}
```

- 同步块：synchronized(obj){ } obj-->线程共同访问的对象，大括号放想要进行同步的代码块

```java
public class SafeThreadTest {
    public static void main(String[] args) {
        Card card = new Card();
        card.setMoney(100);
        card.setUsrName("存款");
        Bank a = new Bank(card, 50, "a");
        Bank b = new Bank(card, 100, "b");
        a.start();
        b.start();
    }
}

//银行卡
class Card {

    private int money;
    private String usrName;

    public void setMoney(int money) {
        this.money = money;
    }

    public void setUsrName(String usrName) {
        this.usrName = usrName;
    }

    public String getUsrName() {
        return usrName;
    }

    public int getMoney() {
        return money;
    }
}

//取钱操作
class Bank extends Thread {
    Card card;
    private int outMoney;
    private int nowMoney;

    public Bank(Card card, int outMoney, String name) {
        super(name);
        this.card = card;
        this.outMoney = outMoney;
    }

    @Override
    public void run() {
        //同步块，上锁对象为公共使用对象
        synchronized (card) {
            if (card.getMoney() - outMoney < 0) {
```

```
 System.out.println(Thread.currentThread().getName() + "钱不
够，取不了");
                return;
            }
            try {
                Thread.sleep(100);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            card.setMoney(card.getMoney() - outMoney);
            nowMoney = nowMoney + outMoney;
            System.out.println(card.getUsrName() + "余额为：" +
card.getMoney());
            System.out.println(this.getName() + "手里的钱：" +
nowMoney);
        }
    }
}
```

生产者消费者问题

```
//生产者消费者模型-->缓冲区解决
public class Example {
    public static void main(String[] args) {
        Buffer buffer = new Buffer();
        Producer producer = new Producer(buffer);
        producer.setPriority(7);    //设置生产者优先级高于消费者
        producer.start();
        Consumer consumer = new Consumer(buffer);
        consumer.setPriority(6);
        consumer.start();
    }
}

//生产者
class Producer extends Thread {
    Buffer buffer;

    public Producer(Buffer buffer) {
        this.buffer = buffer;
    }

    @Override
    public void run() {
        for (int i = 1; i <= 100; i++) {
            buffer.push(new Product(i));
            System.out.println("生产了第-->" + i + "只鸡");
```

```java
        }
    }
}

//消费者
class Consumer extends Thread {
    Buffer buffer;

    public Consumer(Buffer buffer) {
        this.buffer = buffer;
    }

    @Override
    public void run() {
        for (int i = 0; i < 100; i++) {
            System.out.println("消费了第-->" + buffer.pop().id +
"只鸡");
        }
    }
}

//产品
class Product {
    int id;

    public Product(int id) {
        this.id = id;
    }
}

//缓冲区
class Buffer {
    //容器计数器
    int count = 0;
    //容器大小
    Product[] products = new Product[10];

    //生产者放入产品
    public synchronized void push(Product product) {
        while (count == products.length) {
            try {
                //等待生产
                this.wait();
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
        //如果没有满，生产者丢入产品
        products[count] = product;
        count++;
        //生产完毕，通知消费者消费
        this.notifyAll();
    }

    public synchronized Product pop() {
        while (count == 0) {
            try {
                //消费者等待
                this.wait();
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
        //如果可以消费
        count--;
        Product product = products[count];
        //吃完了，通知生产者生产
        this.notifyAll();
        return product;
    }
}
```

线程池

```
package com.wll.thread;

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

public class TreadPoolTest {
    public static void main(String[] args) {
        //创建服务，创建线程池
        ExecutorService service = 
Executors.newFixedThreadPool(10);
        //执行
        service.execute(new MyThread());
        service.execute(new MyThread());
        service.execute(new MyThread());
        service.submit(new MyThread());
        service.submit(new MyThread());
        service.submit(new MyThread());
        //关闭
        service.shutdown();
    }
}

class MyThread implements Runnable{
    @Override
    public void run() {
        for (int i = 0; i < 100; i++) {
            System.out.println(Thread.currentThread().getName()+"-
"+i);
        }
    }
}
```