1. 多线程编程基础

1.1 进程、线程

1.1.1 进程

狭义：进程是正在运行的程序的实例

广义：进程是一个具有一定独立功能的程序，关于某个数据集合的一次运行活动

进程是操作系统动态执行的基本单元，在传统的操作系统中，进程即是基本的分配单元，也是基本的执行单元

1.1.2 线程

线程是操作系统能够进行运算调试得最小单位。它被包含在进程中，是进程中的实际执行单位。一个线程指的是进程中得一个单一顺序的控制流，一个进程中可以并发多个线程，每个线程执行不同得任务。

1.1.3 多线程得优点

- 可以把占据时间较长得任务放在后台处理
- 程序得运行速度加快

1.1.4 主线程与子线程

JVM 启动时会创建一个主线程,该主线程负责执行main 方法. 主线程就是运行main 方法的线程 Java 中的线程不孤立的,线程之间存在一些联系. 如果在A 线程中创建了B 线程, 称B 线程为A 线程的子线程, 相应的A 线程就是B 线程的父线程

1.2 使用多线程

1.2.1 继承Thread

步骤:

- 创建一个类，继承Thread
- 重写Thread类得run方法
- 实例化创建好的线程类
- 通过实例化对象调用start方法启动线程

```java
public class Demo {
    public static void main(String[] args) {
        DemoThread thread = new DemoThread();
        thread.start(); //启动线程
        System.out.println("运行了main方法");
    }
}

```

```java
class DemoThread extends Thread{
    @Override
    public void run() {
        System.out.println("运行了run方法");
    }
}
```

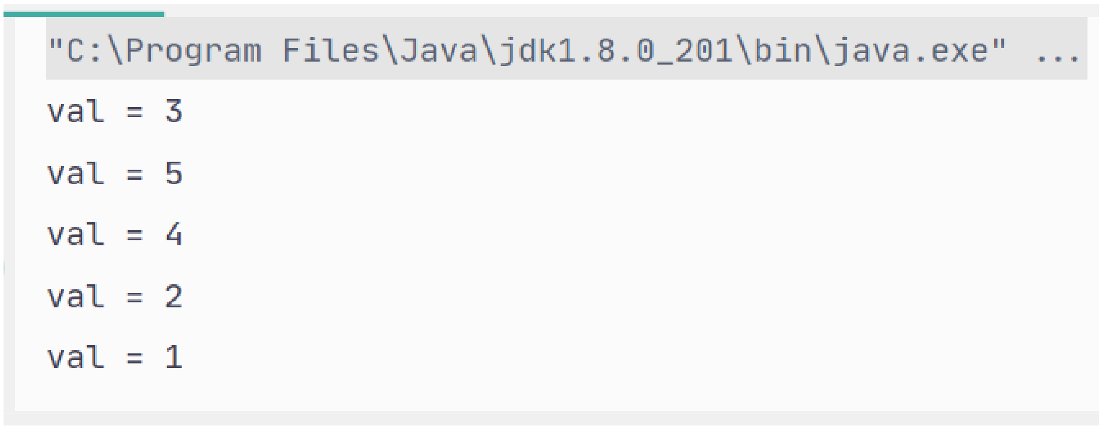
线程运行具有以下特点

1. 随机性

```java
public class Demo {
    public static void main(String[] args) {
        DemoThread thread = new DemoThread();
        thread.start(); //启动线程
        try {
            for (int i = 0; i < 10; i++) {
                System.out.println("运行了main方法");
                Thread.sleep(500);
            }
        }catch (InterruptedException e){
            e.printStackTrace();
        }

    }
}

class DemoThread extends Thread {
    @Override
    public void run() {
        try {
            for (int i = 0; i < 10; i++) {
                System.out.println("运行了run方法");
                Thread.sleep(500);
            }
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}
```

2. start的执行顺序与启动顺序不一致

```java
public class Demo {
    public static void main(String[] args) {

        DemoThread thread1 = new DemoThread(1);
        DemoThread thread2 = new DemoThread(2);
        DemoThread thread3 = new DemoThread(3);
        DemoThread thread4 = new DemoThread(4);
        DemoThread thread5 = new DemoThread(5);
        thread1.start();
        thread2.start();
        thread3.start();
        thread4.start();
        thread5.start();
    }
}

class DemoThread extends Thread {
    private int val;
    public DemoThread(int val){
        this.val = val;
    }
    @Override
    public void run() {
        System.out.println("val = " + val);
    }
}
```



```
"C:\Program Files\Java\jdk1.8.0_201\bin\java.exe" ...
val = 3
val = 5
val = 4
val = 2
val = 1
```

1.2.2 实现Runnable接口

步骤:

- 创建一个类，实现Runnable接口
- 实现接口的run方法
- 实例化创建的这个类
- 实例化一个Thread类，并把第三步的对象通过Thread的构造方法传入Thread对象里面
- 调用Thread对象得start()方法

```java
public class Demo {
    public static void main(String[] args) {
        DemoRunnable r = new DemoRunnable();
        Thread t = new Thread(r);
        t.start();
        System.out.println("运行main方法");
    }
}

class DemoRunnable implements Runnable {

    @Override
    public void run() {
        System.out.println("运行run方法");
    }
}
```

1.2.3 通过Callable和FutureTask创建线程

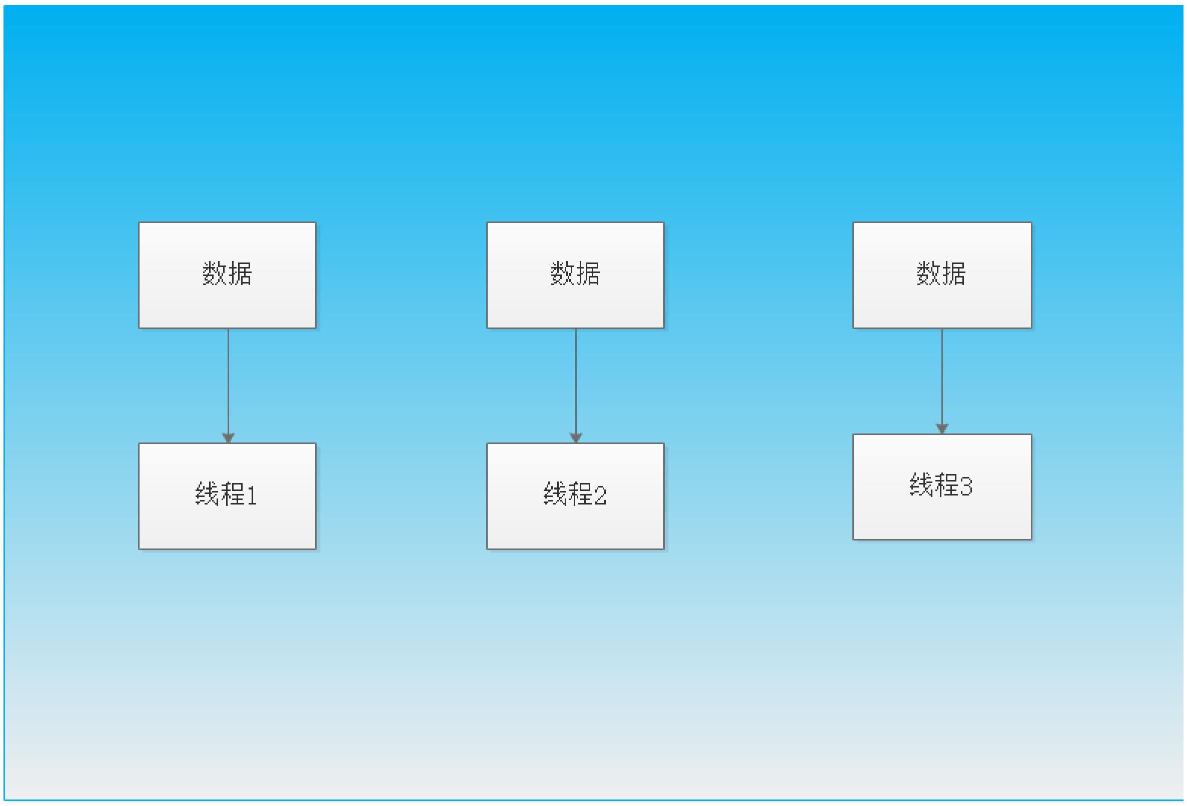
```java
public class CallableCreateTest {
    public static void main(String[] args) throws Exception {

        MyCallable callable = new MyCallable();
        FutureTask<Integer> futureTask = new FutureTask<>(callable);
        new Thread(futureTask).start();

```

```
            Integer sum = futureTask.get();
            System.out.println(Thread.currentThread().getName() + Thread.currentThread().getId() + "=" + sum);
    }
}


class MyCallable implements Callable<Integer> {

    @Override
    public Integer call() throws Exception {
        System.out.println(Thread.currentThread().getName() + "\t" + Thread.currentThread().getId() + "\t" + new Date() + " \tstarting...");

            int sum = 0;
            for (int i = 0; i <= 100000; i++) {
                sum += i;
            }
            Thread.sleep(5000);


   System.out.println(Thread.currentThread().getName() + "\t" + Thread.currentThread().getId() + "\t" + new Date() + " \tover...");
            return sum;
        }
}
```

1.2.4 成员变量与线程安全

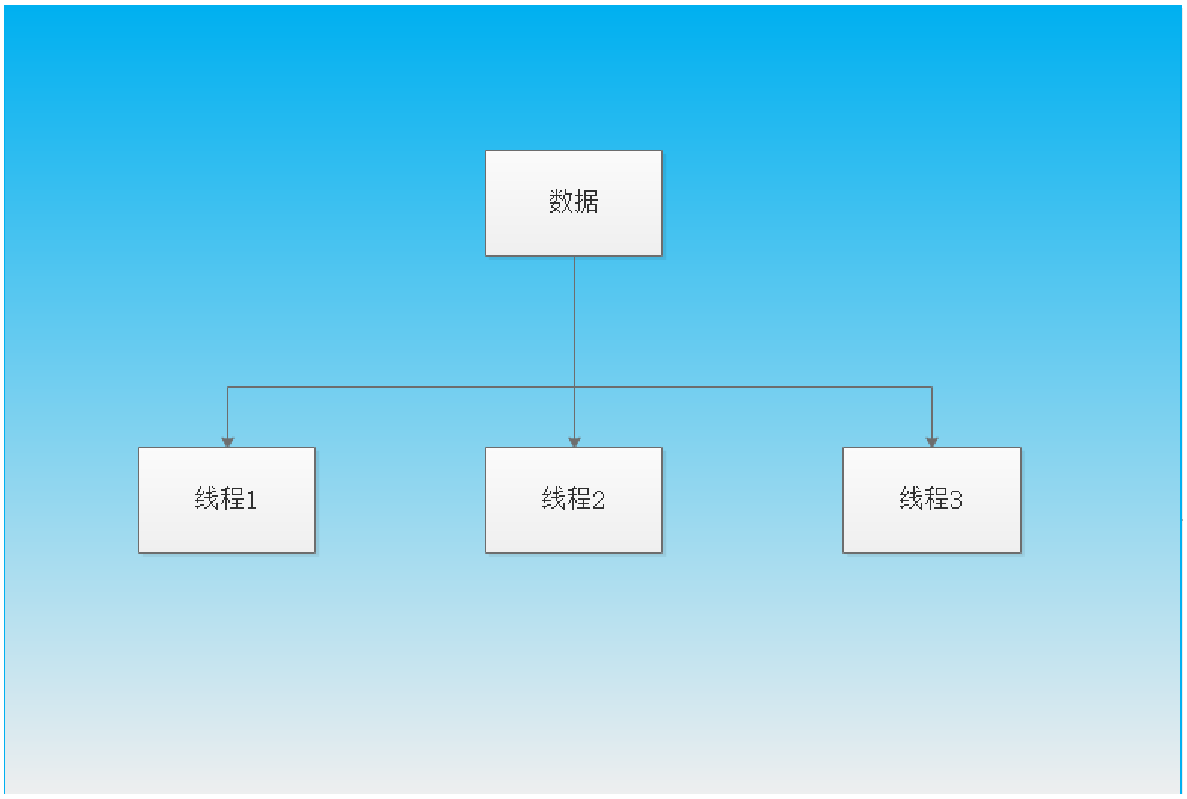
自定义线程类中得成员变量针对其它线程可以分为共享和不共享，这个多线程之间交互时很重要得一个技术点

不共享



```java
public class Demo {
    public static void main(String[] args) {
        DemoThread t1 = new DemoThread();
        t1.start();
        DemoThread t2 = new DemoThread();
        t2.start();
        DemoThread t3 = new DemoThread();
        t3.start();
    }
}

class DemoThread extends Thread{
    private int count = 5;
    @Override
    public void run() {
        while (count > 0){
            count--;
            System.out.println(Thread.currentThread().getName() + " count=" + count);
        }
    }
}
```

共享



```java
public class Demo {
    public static void main(String[] args) {
        DemoThread t1 = new DemoThread();
        Thread thread1 = new Thread(t1);
        Thread thread2 = new Thread(t1);
        Thread thread3 = new Thread(t1);
        Thread thread4 = new Thread(t1);
        Thread thread5 = new Thread(t1);
        thread1.start();
        thread2.start();
        thread3.start();
        thread4.start();
```

```java
            thread5.start();
        }
}

class DemoThread extends Thread{
        private int count = 5;
        @Override
        public void run() {
                count--;
                
        System.out.println(Thread.currentThread().getName() + " count=" + count);
        }
}
```

1.3 线程常用API

- **currentThread方法**
    - 获取当前正在执行的线程
- **isAlive方法**
    - 判断当前线程是否处于活动状态

    ```java
    public class Demo {
        public static void main(String[] args) {
            DemoThread t1 = new DemoThread();
    ```

```java
        System.out.println("准备开启线程: " + t1.isAlive());
        t1.start();
        System.out.println("启动线程后: " + t1.isAlive());
    }
}

class DemoThread extends Thread{
    @Override
    public void run() {
        System.out.println("run方法运行状态: " + this.isAlive());
    }
}
```

- **sleep方法**
  - 让当前正在执行的线程暂停（休眠）多少毫秒数
- **getId方法**
  - 获取当前线程唯一标识

1.4 停止线程

停止一个线程意味着在线程处理完成任务之前结束正在执行的操作

- 使用退出标志，使线程正常退出，也就是当run()方法完成后线程终止

```java
public class Demo {
    public static void main(String[] args) throws InterruptedException {
        DemoThread t1 = new DemoThread();
        t1.start();
        Thread.sleep(3000);
        t1.stopThread();
    }
}

class DemoThread extends Thread {
    private boolean flag = true;

    @Override
    public void run() {
        try {
            while (flag) {
                System.out.println(Thread.currentThread());
                Thread.sleep(1000);
            }
```

```
                System.out.println("线程执行结束");
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
    }

    public void stopThread(){
        flag = false;
    }
}
```

- 使用stop方法（过时）
- interrupt方法

```
public class Demo {
    public static void main(String[] args) throws InterruptedException {
        DemoThread t1 = new DemoThread();
        t1.start();
        t1.interrupt();
    }
}

class DemoThread extends Thread {
```

```java

    @Override
    public void run() {
        for (int i = 0; i < 1000; i++) {
            System.out.println(i);
        }
    }

}
```

调用interrupt方法不会真正的结束线程，而是在当前线程中打上一个停止的标记

Thread类提供了interrupt方法测试当前线程是否中断，isInterrupt方法测试线程是否已经中断

```java
public class Demo {
    public static void main(String[] args) throws InterruptedException {
        DemoThread t1 = new DemoThread();
        t1.start();
         t1.interrupt();System.out.println("是否停止1：" + t1.isInterrupted());
        System.out.println("是否停止1：" + t1.isInterrupted());
        System.out.println("是否停止2: " + Thread.interrupted());

    }
}

class DemoThread extends Thread {

    @Override
    public void run() {
        for (int i = 0; i < 1000; i++) {
            System.out.println(i);
        }
    }

}
```

```java
public class Demo {
    public static void main(String[] args) throws InterruptedException {
        Thread.currentThread().interrupt();
        System.out.println("是否停止1： " + Thread.interrupted());
        System.out.println("是否停止2： " + Thread.interrupted());
    }
}
```

如果连续两次调用interrupted()方法，第二次将返回false

```java
public class Demo {
    public static void main(String[] args) throws InterruptedException {
        DemoThread t = new DemoThread();
        t.start();
        Thread.sleep(2);
        t.interrupt();
    }
}

class DemoThread extends Thread {

    @Override
    public void run() {
        try {
            for (int i = 0; i < 1000; i++) {
                if (this.isInterrupted()){
                    System.out.println("已经是停止状态了，我要退出了。");
//                  break;
                    throw new InterruptedException();
                }
                System.out.println(i);
            }
        }catch (InterruptedException e){
            e.printStackTrace();
        }

    }
}
```

1.5 暂停线程

1、暂停线程使用suspend方法，重启暂停线程使用resume方法

```java
public class Demo {
    public static void main(String[] args) throws InterruptedException {
        DemoThread t = new DemoThread();
        t.start();
        Thread.sleep(1000);

        t.suspend();  //暂停线程
        System.out.println("A=" + System.currentTimeMillis() + ",i= " + t.getI());
        Thread.sleep(1000);
        System.out.println("A=" + System.currentTimeMillis() + ",i= " + t.getI());

        t.resume();  //恢复暂停的线程
        Thread.sleep(1000);

        t.suspend();  //暂停线程
        System.out.println("B=" + System.currentTimeMillis() + ",i= " + t.getI());
        Thread.sleep(1000);
        System.out.println("B=" + System.currentTimeMillis() + ",i= " + t.getI());
    }
}

class DemoThread extends Thread {
    private long i = 0;


    public long getI() {
        return i;
    }

    public void setI(long i) {
        this.i = i;
    }

    @Override
    public void run() {
        while (true){
            i++;
        }
    }
}
```

为什么suspend方法和resume方法被弃用了？

1、suspend如果独占公共的同步对象，使其它线程无法访问公共同步对象

```
public class Demo {
    public static void main(String[] args) throws InterruptedException {
        Print print = new Print();
        Thread t1 = new Thread(){
            @Override
            public void run() {
                print.printString();
            }
        };
        t1.setName("A");
        t1.start();
        Thread.sleep(10);
        Thread t2 = new Thread(){
            @Override
            public void run() {

                print.printString();

            }
```

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

class Print {
    public void printString(){
        System.out.println("线程" + Thread.currentThread().getName() + "开始");
        if ("A".equals(Thread.currentThread().getName())){
            System.out.println("A线程永远suspend了");
            Thread.currentThread().suspend();
        }
        System.out.println("线程" +Thread.currentThread().getName() + "结束");
    }
}
```

运行结果

```
"C:\Program Files\Java\jdk1.8.0_201\bin\java.exe" ...
线程A开始
A线程永远suspend了
线程Thread-1开始
线程Thread-1结束
```

这个时候在printString方法上加同步锁

```java
public class Demo {
    public static void main(String[] args) throws InterruptedException {
        Print print = new Print();
        Thread t1 = new Thread(){
            @Override
            public void run() {
                print.printString();
            }
        };
        t1.setName("A");
        t1.start();
        Thread.sleep(10);
        Thread t2 = new Thread(){
            @Override
            public void run() {
                print.printString();
            }
        };
        t2.start();
    }
}

class Print {
    public synchronized void printString(){
        System.out.println("线程" + Thread.currentThread().getName() + "开始");
        if ("A".equals(Thread.currentThread().getName())){
            System.out.println("A线程永远suspend了");
            Thread.currentThread().suspend();
        }
        System.out.println("线程" +Thread.currentThread().getName() + "结束");
    }
}

```

运行结果

```
"C:\Program Files\Java\jdk1.8.0_201\bin\java.exe" ...
线程A开始
A线程永远suspend了
```

2、suspend会造成共享对象数据不同步

```java
public class Demo {
    public static void main(String[] args) throws InterruptedException {
        User user = new User();
        Thread t1 = new Thread(){
            @Override
            public void run() {
                user.updateUserNameAndPassword("b","bb");
            }
        };
        t1.setName("A");
        t1.start();
        Thread.sleep(10);
       user.getUserNameAndPassword();
    }
}

class User {

    private String userName = "a";
    private String password = "aa";

    public void updateUserNameAndPassword(String userName, String password){
        this.userName = userName;
        if ("A".equals(Thread.currentThread().getName())){
            Thread.currentThread().suspend();
        }
        this.password = password;
    }
    public void getUserNameAndPassword(){
        System.out.println("userName: " + userName + " password: " + password);
    }

}

```

1.6 yield方法

yield方法的作用使放弃当前CPU资源，将资源让其他任务去占用CPU执行，但放弃时间不确定，有可能刚放弃，马上有获取CPU资源

```java
public class MyThread extends Thread{
    @Override
    public void run() {
        long begin = System.currentTimeMillis();
        long sum = 0;
        for(int i = 1; i <= 1000000; i++){
            sum += i;
            Thread.yield(); //线程让步，放弃CPU 执行权
        }
        long end = System.currentTimeMillis();
        System.out.println("用时: " + (end - begin));
    }
}
```

```java
public class Test {
    public static void main(String[] args) {

        //开启子线程,计算累加和
        MyThread t = new MyThread();
        t.start();
        //在main 线程中计算累加和
        long begin = System.currentTimeMillis();
        long sum = 0;
        for(int i = 1; i <= 1000000; i++){
            sum += i;
        }
        long end = System.currentTimeMillis();
        System.out.println("main 方法, 用时: " + (end - begin));
    }
}
```

1.7 线程的优先级

在操作系统中，线程可以划分优先级，优先级较高的线程得到更多的CPU资源，也就是CPU会优先执行优先级较高的线程对象中的任务。设置线程优先级有助于帮助线程调度器确定在下一次选择哪一个线程优先执行

设置线程的优先级使用setPriority方法，优先级分为1-10个级别，如果优先级小于1或大于10，JDK会抛出IllegalArgumentException。Thread默认设置三个优先级常量。MIN_PRIORITY（1），NORM_PRIORITY（5），MAX_PRIORITY （10）

获取线程的优先级使getPriority方法

1.8 守护线程

在Java线程中有两种线程，一种是用户线程，另一种是守护线程。

守护线程是一种特殊的线程，特殊指的是当进程中不存在用户线程时，守护线程会自动销毁。典型的守护线程垃圾回收线程。

通过setDaemon(true)设置成守护线程

设置守护线程必须要在调用start方法之前设置，否则会产生IllegalThreadStateException异常

```java
public class Demo {
    public static void main(String[] args) throws InterruptedException {
        DemoThread t = new DemoThread();
        t.setDaemon(true); //设置线程为守护线程
        t.start();
        Thread.sleep(3000);
        System.out.println("主线程结束");
    }
}

class DemoThread extends Thread {

    @Override
    public void run() {
        try {
            while (true){
                System.out.println(System.currentTimeMillis());
                Thread.sleep(1000);
            }
        }catch (InterruptedException e){
            e.printStackTrace();
        }

    }
}
```

1.9 加入一个线程

一个线程可以在其它线程之上调用**join()**方法，其效果是等待一段时间直到第二个线程结束才继续执行。如果某一个线程在另一个线程t上调用t.join(),此线程就被挂起，直到目标线程t结束才恢复（即**t.isAlive()**返回假）

```java
public class JoinThread {
    public static void main(String[] args) {
        Thread  t = new Thread(){
            @Override
            public void run() {
                for (int i = 0; i < 100; i++) {
                    try {
                        sleep(50);
                    } catch (InterruptedException e) {
                        e.printStackTrace();
                    }
                    System.out.println(getName() + ",i的值为: " + i);
                }
            }
        };
        t.start();

        for (int i = 0; i < 100; i++) {
            System.out.println(Thread.currentThread().getName() + ",i的值为： " + i);
            if (i == 20){
                try {
                    t.join();
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
        }
    }
}
```