Day05

新单词

- **handl**-处理
- **Thread**-线程
- **start**-开始
- **DaemonThread**-守护线程
- **sleep**-睡眠
- **interrupt**-中断
- **alive**-活着
- **priority**-优先级
- **current**-当前的

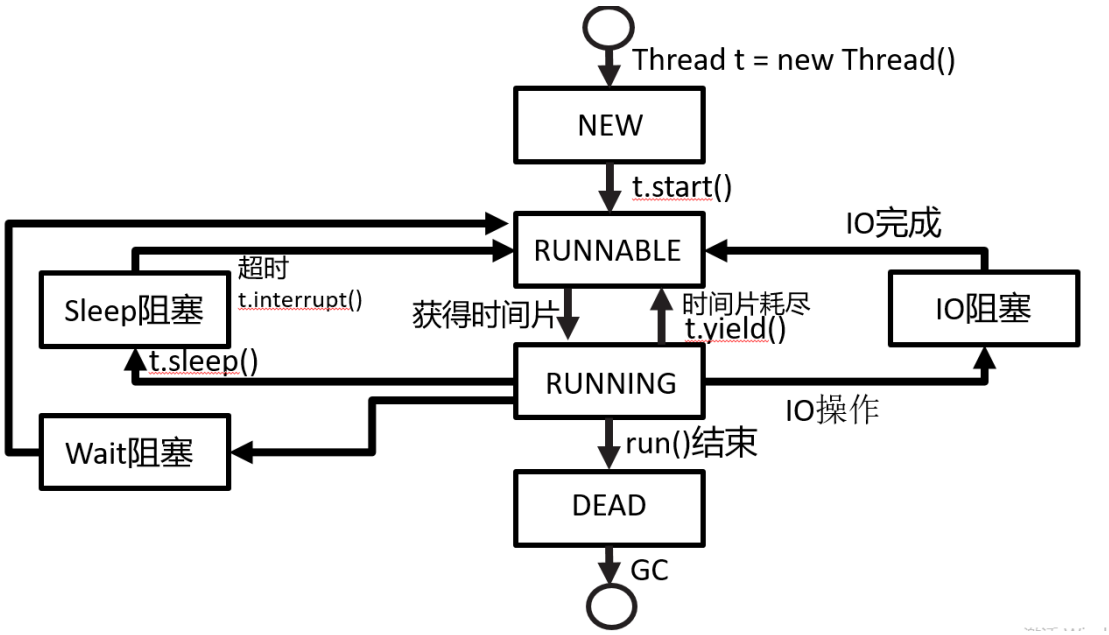
多线程

概念

- **线程:一个顺序的单一的程序执行流程就是一个线程。代码一句一句的有先后顺序的执行。**
- **多线程:多个单一顺序执行的流程并发运行。造成"感官上同时运行"的效果。**
- **并发:**

  多个线程实际运行是走走停停的。线程调度程序会将CPU运行时间划分为若干个时间片段并尽可能均匀的分配给每个线程，拿到时间片的线程被CPU执行这段时间。当超时后线程调度程序会再次分配一个时间片段给一个线程使得CPU执行它。如此反复。由于CPU执行时间在纳秒级别，我们感觉不到切换线程运行的过程。所以微观上走走停停，宏观上感觉一起运行的现象成为并发运行!
- **用途:**
  - 当出现多个代码片段执行顺序有冲突时，希望它们各干各的时就应当放在不同线程上"同时"运行
  - 一个线程可以运行，但是多个线程可以更快时，可以使用多线程运行

线程的生命周期图



线程的创建

1. 继承Thread并重写run方法
2. 单独定义线程任务

方式一

继承Thread并重写run方法

定义一个线程类，重写run方法，在其中定义线程要执行的任务(希望和其他线程并发执行的任务)。

注:启动该线程要调用该线程的start方法，而不是run方法！！！

```
package thread;

/**
 * 多线程
 * 线程:程序中一个单一的顺序执行流程，即：线性流程
 * 多线程:多个线性流程"一起"执行。
 *
 * 线程是并发运行的。
 * 并发：线程间的代码在微观世界都是走走停停的，宏观上给我们的感受是在一起执行
 *
 * 线程的创建
 * 第一种方式:继承Thread并重写run方法
 *
 */
public class ThreadDemo1 {
    public static void main(String[] args) {
        Thread t1 = new MyThread1();
        Thread t2 = new MyThread2();
        //启动线程要调用start方法，而不是直接调用run方法!!
        /*
            当我们调用线程的start方法后，线程会纳入到线程调度器中被统一管理。
            当线程第一次分配时间片后会自动调用它的run方法开始执行。
         */
        t1.start();
        t2.start();
```

```
    }
}

/**
 * 第一种创建线程的方式的优点:结构简单，利于匿名内部类创建
 * 缺点:
 * 1:存在继承冲突问题，由于java是单继承的，这导致如果我们继承了Thread就无法再
 *    继承其他类去复用方法。这在实际开发中是极其不方便的
 * 2:当我们继承Thread并重写run方法，在run方法中定义线程要执行的任务。这会导致
 *    线程与任务存在一个必然的耦合关系，不利于线程的重用。
 */
class MyThread1 extends Thread{
    public void run(){
        for (int i = 0; i < 1000; i++) {
            System.out.println("你是谁啊?");
        }
    }
}
class MyThread2 extends Thread{
    public void run(){
        for (int i = 0; i < 1000; i++) {
            System.out.println("开门，查水表的!");
        }
    }
}
```

优缺点

优点:

在于结构简单，便于匿名内部类形式创建。

缺点:

- 1:直接继承线程，会导致不能在继承其他类去复用方法，这在实际开发中是非常不便的。
- 2:定义线程的同时重写了run方法，会导致线程与线程任务绑定在了一起，不利于线程的重用。

方式二

实现Runnable接口单独定义线程任务

```
package thread;

/**
 * 第二种创建线程的方式:单独定义线程任务
 * 定义任务:
 * 1:实现Runnable接口
 * 2:实现Callable接口(如果线程执行完毕后需要返回值时使用，多用于线程池)
 *
 */
public class ThreadDemo2 {
    public static void main(String[] args) {
        //1创建线程要执行的任务
        Runnable r1 = new MyRunnable1();
        Runnable r2 = new MyRunnable2();
        //2创建线程
```

```
        Thread t1 = new Thread(r1);
        Thread t2 = new Thread(r2);
        //3启动
        t1.start();
        t2.start();
    }
}
class MyRunnable1 implements Runnable{
    public void run(){
        for (int i = 0; i < 1000; i++) {
            System.out.println("你是谁啊?");
        }
    }
}
class MyRunnable2 implements Runnable{
    public void run(){
        for (int i = 0; i < 1000; i++) {
            System.out.println("查水表的!");
        }
    }
}
```

匿名内部类形式的线程创建

```
package thread;

/**
 * 使用匿名内部类形式完成线程的两种创建
 */
public class ThreadDemo3 {
    public static void main(String[] args) {
        //继承Thread重写run方法
        Thread t1 = new Thread(){
            public void run(){
                for (int i = 0; i < 1000; i++) {
                    System.out.println("你是谁啊?");
                }
            }
        };
        //实现Runnable接口重写run方法
//        Runnable r2 = new Runnable() {
//            public void run() {
//                for (int i = 0; i < 1000; i++) {
//                    System.out.println("我是查水表的!");
//                }
//            }
//        };
//        Thread t2 = new Thread(r2);

        //lambda表达式
//        Runnable r2 = ()->{
//            for (int i = 0; i < 1000; i++) {
//                System.out.println("我是查水表的!");
```

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

        Thread t2 = new Thread(()->{
            for (int i = 0; i < 1000; i++) {
                System.out.println("我是查水表的!");
            }
        });


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

主线程

java中的代码都是靠线程运行的，执行main方法的线程称为"主线程"。

线程提供了一个方法:

- static Thread currentThread()

  该方法可以获取运行这个方法的线程

```
package thread;

/**
 * 主线程
 * java程序启动后，JVM会创建一条线程来执行main方法。并且JVM为该线程取名为"main"
 * 因此我们称执行main方法的线程为主线程
 *
 * 线程提供了一个静态方法:
 * static Thread currentThread()
 * 该方法可以获取运行这个方法的线程
 */
public class CurrentThreadDemo {
    public static void main(String[] args) {
        //让主线程执行该方法，此时该方法返回的就是主线程
        Thread main = Thread.currentThread();
        System.out.println(main);
        dosome();//主线程执行dosome方法

        Thread myThread = new Thread(){
            public void run(){
                dosome();//自定义线程执行dosome
            }
        };
        myThread.start();
    }

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

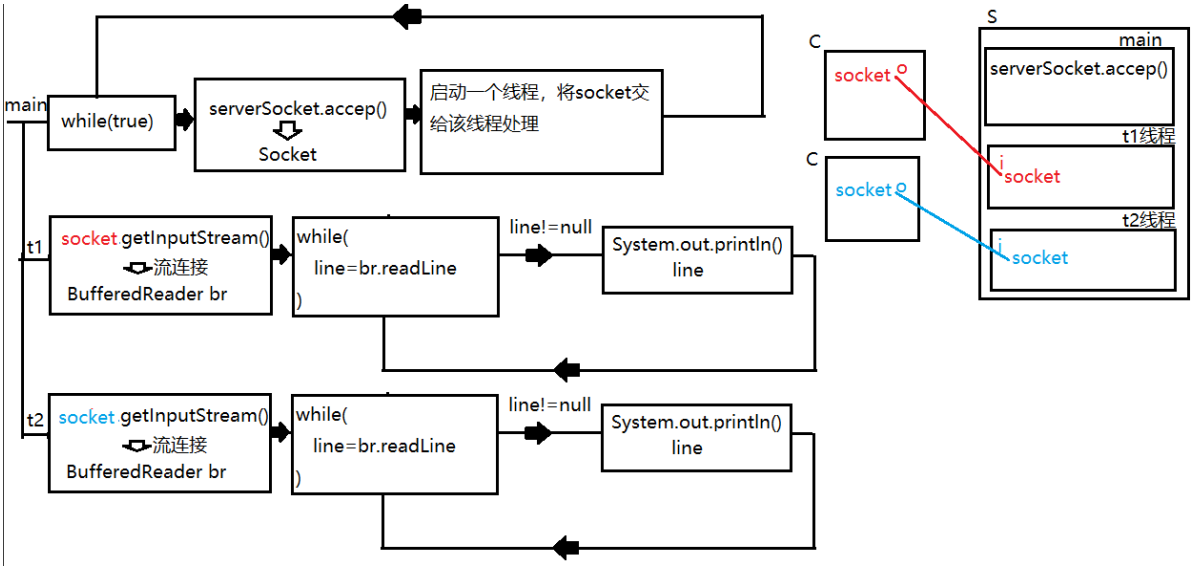
```
    }
}
```

使用多线程实现多客户端连接服务端

若想使一个服务端可以支持多客户端连接，我们需要解决以下问题

- 循环调用accept方法侦听客户端的连接
- 使用线程来处理单一客户端的数据交互

因为需要处理多客户端，所以服务端要循环调用accept方法，但该方法会产生阻塞，且读取客户端的消息也是依靠一个循环完成的，这会导致它接近是一个死循环，不结束的话也会影响服务端再次调用accept方法。所以与某个客户端的交互就需要使用线程来并发处理。



服务端代码改造

```
package socket;

import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStream;
import java.io.InputStreamReader;
import java.net.ServerSocket;
import java.net.Socket;
import java.nio.charset.StandardCharsets;

/**
 * 聊天室服务端
 */
public class Server_V2 {
    /**
     * java.net.ServerSocket
     * 运行在服务端的ServerSocket主要有两个工作:
     * 1:打开服务端口，客户端就是通过这个端口与服务端建立连接的
     * 2:监听服务端口，一旦一个客户端连接，则立即返回一个Socket实例
     */
```

```
    private ServerSocket serverSocket;

    public Server_V2(){
        try {
            /*
                ServerSocket实例化的同时指定服务端口
                如果该端口被其他程序占用则会抛出异常:
                java.net.BindException:address already in use
                此时我们需要更换端口，或者杀死占用该端口的进程。

                端口号范围:0-65535
             */
            System.out.println("正在启动服务端...");
            serverSocket = new ServerSocket(8088);
            System.out.println("服务端启动完毕!");
        } catch (IOException e) {
            e.printStackTrace();
        }
    }

    /**
     * 服务端开始工作的方法
     */
    public void start(){
        try {
            /*
                ServerSocket的一个重要方法:
                Socket accept()
                该方法用于接受客户端的连接。这是一个阻塞方法，调用后会"卡住"，直到
                一个客户端与ServerSocket连接，此时该方法会立即返回一个Socket实例
                通过这个Socket实例与该客户端对等连接并进行通讯。
                相当于"接电话"的动作
             */
            while(true) {
                System.out.println("等待客户端连接...");
                Socket socket = serverSocket.accept();
                System.out.println("一个客户端连接了!");
                //启动一个线程负责与该客户端交互
                ClientHandler handler = new ClientHandler(socket);
                //创建线程
                Thread t = new Thread(handler);
                //启动线程
                t.start();

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

    }

    public static void main(String[] args) {
        Server_V2 server = new Server_V2();
        server.start();
    }
```

```
    /**
     * 第二种创建线程的方式:实现Runnable接口单独定义线程任务
     * 这个线程任务就是让一个线程与指定的客户端进行交互
     */
    private class ClientHandler implements Runnable{
        private Socket socket;

        public ClientHandler(Socket socket){
            this.socket = socket;
        }

        public void run(){
            try {
                InputStream in = socket.getInputStream();
                InputStreamReader isr = new InputStreamReader(in,
StandardCharsets.UTF_8);
                BufferedReader br = new BufferedReader(isr);
                String line;
                while ((line = br.readLine()) != null) {
                    System.out.println("客户端说:" + line);
                }
            } catch (IOException e) {
                e.printStackTrace();
            }
        }
    }

}
```

线程API

获取线程相关信息的方法

```
package thread;

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

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

        long id = main.getId();//获取该线程的唯一标识
        System.out.println("id:"+id);

        int priority = main.getPriority();//获取该线程的优先级
        System.out.println("优先级:"+priority);

        boolean isAlive = main.isAlive();//该线程是否活着
```

```
        System.out.println("是否活着:"+isAlive);

        boolean isDaemon = main.isDaemon();//是否为守护线程
        System.out.println("是否为守护线程:"+isDaemon);

        boolean isInterrupted = main.isInterrupted();//是否被中断了
        System.out.println("是否被中断了:"+isInterrupted);

    }
}
```

线程优先级

线程start后会纳入到线程调度器中统一管理,线程只能被动的被分配时间片并发运行,而无法主动索取时间片.线程调度器尽可能均匀的将时间片分配给每个线程.

线程有10个优先级,使用整数1-10表示

- 1为最小优先级,10为最高优先级.5为默认值
- 调整线程的优先级可以最大程度的干涉获取时间片的几率.优先级越高的线程获取时间片的次数越多,反之则越少.
- Thread提供了对应的常量:MAX_PRIORITY表示最高优先级10，MIN_PRIORITY表示最低优先级，NORM_PRIORITY表示默认优先级5

```
package thread;

public class PriorityDemo {
    public static void main(String[] args) {
        Thread max = new Thread(){
            public void run(){
                for(int i=0;i<10000;i++){
                    System.out.println("max");
                }
            }
        };
        Thread min = new Thread(){
            public void run(){
                for(int i=0;i<10000;i++){
                    System.out.println("min");
                }
            }
        };
        Thread norm = new Thread(){
            public void run(){
                for(int i=0;i<10000;i++){
                    System.out.println("nor");
                }
            }
        };
        min.setPriority(Thread.MIN_PRIORITY);
        max.setPriority(Thread.MAX_PRIORITY);
        min.start();
```

```
        norm.start();
        max.start();
    }
}
```

sleep阻塞

线程提供了一个静态方法:

- static void sleep(long ms)
- 使运行该方法的线程进入阻塞状态指定的毫秒,超时后线程会自动回到RUNNABLE状态等待再次获取时间片并发运行.

```
package thread;

public class SleepDemo {
    public static void main(String[] args) {
        System.out.println("程序开始了!");
        try {
            Thread.sleep(5000);//主线程阻塞5秒钟
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        System.out.println("程序结束了!");
    }
}
```

完成一个倒计时程序

```
package thread;

import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.util.Scanner;

/**
 * sleep阻塞
 * 线程提供了一个静态方法:
 * static void sleep(long ms)
 * 该方法可以让执行该方法的线程处于阻塞状态指定毫秒，超时后线程会再次回到RUNNABLE状态
 * 再次并发
 */
public class SleepDemo {
    public static void main(String[] args) throws IOException {
        System.out.println("程序开始了");
        /*
            简易的倒计时程序
            程序启动后在控制台上输入一个整数，从该数字开始每秒递减，到0时输出时间到
         */
//        BufferedReader br = new BufferedReader(
//                new InputStreamReader(
//                        System.in
```

```
//                )
//        );
//        System.out.println("请输入一个数字");
//        int num = Integer.parseInt(br.readLine());

        Scanner scanner = new Scanner(System.in);
        System.out.println("请输入一个数字");
        for(int num = scanner.nextInt();num>0;num--) {
            System.out.println(num);
            try {
                Thread.sleep(1000);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
        System.out.println("程序结束了");
    }
}
```

sleep阻塞(续)

sleep方法处理异常:InterruptedException.

当一个线程调用sleep方法处于睡眠阻塞的过程中,该线程的interrupt()方法被调用时,sleep方法会抛出该异常从而打断睡眠阻塞.

```
package thread;

/**
 * sleep方法要求必须处理中断异常
 * 当一个线程调用sleep方法处于睡眠阻塞的过程中，如果此时该线程的interrupt()方法被
 * 调用，此时会中断该线程的睡眠阻塞，那么sleep方法就会抛出中断异常。
 */
public class SleepDemo2 {
    public static void main(String[] args) {
        Thread lin = new Thread("林永健"){
            public void run(){
                System.out.println(getName()+":刚美完容，睡一会吧...");
                try {
                    Thread.sleep(500000000);
                } catch (InterruptedException e) {
                    System.out.println(getName()+":干嘛呢!干嘛呢!干嘛呢!都破了相
了!");
                }
                System.out.println(getName()+":醒了");
            }
        };

        Thread huang = new Thread("黄宏"){
            public void run(){
                System.out.println(getName()+":大锤80，小锤40，开始砸墙! ");
                for(int i=0;i<5;i++){
                    System.out.println(getName()+":80!");
                    try {
```

```
                    Thread.sleep(1000);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
            System.out.println("咣当！！！");
            System.out.println(getName()+":大哥！搞定！");
            lin.interrupt();//中断lin线程的睡眠阻塞
        }
    };
    lin.start();
    huang.start();
  }
}
```

守护线程

概念

守护线程也称为:后台线程

- 守护线程是通过普通线程调用setDaemon(boolean on)方法设置而来的,因此创建上与普通线程无异.
- 守护线程的结束时机上有一点与普通线程不同,即:进程的结束.
- 进程结束:当一个进程中的所有普通线程都结束时,进程就会结束,此时会杀掉所有正在运行的守护线程.

用途

- GC就是运行在守护线程上的
- 当我们有一个任务不需要关注何时结束，当程序需要结束时可以一起结束的任务就可以放在守护线程上执行

例

```
package thread;

/**
 * 守护线程
 * 线程提供了一个方法:
 * void setDaemon(boolean on)
 * 如果参数为true，则会将当前线程设置为守护线程。
 *
 * 守护线程与普通的用户线程(线程创建出来时默认就是用户线程)的区别在于进程结束
 *
 * 进程结束:
 * 当一个JAVA进程中所有的用户线程都结束时，进程就会结束，此时会强制杀死所有还在运行
 * 的守护线程。
 *
 * GC就是运行在一条守护线程上的。
 */
public class DaemonThreadDemo {
    public static void main(String[] args) {
        Thread rose = new Thread("rose"){
```

```
            public void run(){
                for(int i=0;i<5;i++){
                    System.out.println(getName()+":let me go !!!");
                    try {
                        Thread.sleep(1000);
                    } catch (InterruptedException e) {
                    }
                }
                System.out.println(getName()+":啊啊啊啊啊AAAAAAaaaaa....");
                System.out.println("噗通！");
            }
        };

        Thread jack = new Thread("jack"){
            public void run(){
                while(true){
                    System.out.println(getName()+":you jump!i jump!");
                    try {
                        Thread.sleep(1000);
                    } catch (InterruptedException e) {
                    }
                }
            }
        };

        rose.start();
        jack.setDaemon(true);//设置守护线程必须在线程启动前进行
        jack.start();


    }
}
```

总结

多线程

线程:单一的顺序执行流程就是一个线程，顺序执行:代码一句一句的先后执行。

多线程:多个线程并发执行。线程之间的代码是快速被CPU切换执行的，造成一种感官上"同时"执行的效果。

线程的创建方式

1. 继承Thread，重写run方法，在run方法中定义线程要执行的任务

   优点:

   - 结构简单，便于匿名内部类创建

   缺点:

   - 继承冲突:由于java单继承，导致如果继承了线程就无法再继承其他类去复用方法
   - 耦合问题:线程与任务耦合在一起，不利于线程的重用。
2. 实现Runnable接口单独定义线程任务

   优点:

- 犹豫是实现接口，没有继承冲突问题
- 线程与任务没有耦合关系，便于线程的重用

缺点:

- 创建复杂一些(其实也不能算缺点)

线程Thread类的常用方法

void run():线程本身有run方法，可以在第一种创建线程时重写该方法来定义线程任务。

void start():**启动线程**的方法。调用后线程被纳入到线程调度器中统一管理，并处于RUNNABLE状态，等待分配时间片开始并发运行。

> 注:线程第一次获取时间片开始执行时会自动执行run方法。
>
> **启动线程一定是调用start方法，而不能调用run方法!**

String getName():获取线程名字

long getId():获取线程唯一标识

int getPriority():获取线程优先级，对应的是整数1-10

boolean isAlive():线程是否还活着

boolean isDaemon():是否为守护线程

boolean isInterrupted():是否被中断了

void setPriority(int priority):设置线程优先级，参数可以传入整数1-10。1为最低优先级，5为默认优先级，10为最高优先级

> 优先级越高的线程获取时间片的次数越多。可以使用Thread的常量
> MIN_PRIORITY,NORM_PRIORITY,MAX_PRIORITY。
> 他们分别表示最低，默认，最高优先级

static void sleep(long ms):静态方法sleep可以让运行该方法的线程阻塞参数ms指定的毫秒。

static Thread currentThread():获取运行该方法的线程。

void setDaemon(boolean on):设置线程是否为守护线程，当参数为true时当前线程被设置为守护线程。**该操作必须在线程启动前进行**

守护线程与普通线程的区别主要体现在当java进程中所有的普通线程都结束时进程会结束，在结束前会杀死所有还在运行的守护线程。