第二十四章 创建和使用线程

1. 线程的概述

1. 进程
    a. 进行就是正在执行的软件（程序）
    b. 一个进程中可以包含多条执行路径（百度网盘，上传的同时还可以下载）
        i. 上传是一条执行路径
        ii. 下载也是一条执行路径
        iii. 每个执行路径都是一个线程
        iv. 一个CPU同一个时间点，只能有一条执行路径，由于CPU执行的效率比较高，所以感觉同时在执行：
            1. 一个手画圆，一个手画方，同时进行
            2. 上班：一边听音乐，一边写代码
            3. 如上案例：外人看来，同时在进行，因为大脑切换速度比较快
2. 线程
    a. 线程是程序的执行路径
    b. 一个进程可以包含多个线程
    c. JVM就是典型的多线程
        i. main方法，主线程
        ii. 垃圾回收线程（守护线程）
3. 线程的并发和并行
    a. 并发：线程切换执行，微观是交替执行，宏观同时执行
    b. 并行：多个CPU,有多条执行路径，同时执行
4. 线程的调动策略
    a. 如果有多个线程，分配给一个CPU执行，同一个时间点，只有一个线程获得CPU执行权
    b. 那么进程中有多个线程会抢夺CPU执行权，这就是线程的抢占式调动策略
        i. 抢占式调动
            1. 让优先级高的线程大概率获得CPU执行权
            2. 如果线程的优先级相同，那么CPU会随机选择一个线程获得执行权

3. Java就是抢占式调度

ii. 分时调度

1. 所有线程轮流使用CPU执行权
2. 尽量均分CPU的时间

5. 线程的创建方式
   a. 继承Thread类
   b. 实现Runnable接口
   c. 实现Callable接口

2. 创建线程的三种方式

2.1 继承Thread类

1. 编写线程类
   a. 继承Thread类
   b. 重写run方法
2. 启动线程
   a. 创建线程实现类对象
   b. 启动线程（调用start方法）

代码块

```
package com.powernode.thread12;

/**
 * 1.编写线程类
 *   1.继承Thread类
 *   2.重写run方法
 */
class MyThread extends Thread{
    @Override
    public void run() {
        for (int i = 0; i < 10; i++) {
            System.out.println("i = " + i);
        }
    }
}
public class Test {
    public static void main(String[] args) {
        //1.创建线程实现类对象
```

```
        MyThread myThread = new MyThread();
        //2.启动线程
            //1.调用start方法启动线程（会通知run方法执行）
            //2.通知后，线程会继续往下执行
        myThread.start();//通知run方法开始执行，通知完毕start方法就执行完毕了，程序会继续往下执行，不会等run方法执行完毕
        //myThread.run();//普通的方法调用，需要等待run方法执行完毕，程序才会继续往下执行

        for (int j = 0; j < 10; j++) {
            System.out.println("--j = " + j);
        }
    }
}
```

2.2 实现Runnable接口

2.2.1 常规方式实现 Runnable接口

1. 编写类
    a. *实现Runnable接口*
    b. *重写run方法*
2. 启动线程
    a. 创建Runnable接口的实现类
    b. 创建线程
    c. 启动线程

代码块

```
package com.powernode.thread13;

/**
 * 1.实现Runnable接口
 * 2.重写run方法
 */
class MyThread implements Runnable{
    @Override
    public void run() {
        for (int i = 0; i < 10; i++) {
            System.out.println("i = " + i);
        }
    }
}
```

```
15  public class Test {
16      public static void main(String[] args) {
17          //1.创建Runnable接口的实现类
18          MyThread myThread = new MyThread();
19          //2.创建Thread类对象，构造方法中传递Runnable接口的实现类对象
20          Thread thread = new Thread(myThread);
21          //3.调用start方法开启多线程
22          thread.start();
23
24          for (int j = 0; j < 10; j++) {
25              System.out.println("--j = " + j);
26          }
27      }
28  }
```

2.2.2 匿名内部类实现

代码块

```
1   package com.powernode.thread14;
2
3   public class Test {
4       public static void main(String[] args) {
5
6           Thread thread = new Thread(new Runnable() {
7               @Override
8               public void run() {
9                   for (int i = 0; i < 10; i++) {
10                      System.out.println("i = " + i);
11                  }
12              }
13          });
14          //3.调用start方法开启多线程
15          thread.start();
16
17          for (int j = 0; j < 10; j++) {
18              System.out.println("--j = " + j);
19          }
20      }
21  }
```

2.3 实现Callable接口

- 如上两种方式都有缺陷

- 线程不可以有返回值
- 线程不可以抛出异常

2.3.1 使用 Callable接口创建一个线程

1. 编写线程类
   a. 实现Callable接口
   b. 重写call方法
2. 启动线程
   a. 创建Callable接口的实现类
   b. 创建FutureTask对象，将Callable接口的实现类作为构造方法参数传入
   c. 创建Thread对象，将FutureTask对象作为Thread的构造方法参数传入
   d. 调用Thread对象的start()方法启动线程

代码块

```
package com.powernode.thread15;

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

class MyThread implements Callable<Integer> {
    @Override
    public Integer call() throws Exception {
        for (int i = 0; i < 10; i++) {
            System.out.println("i = " + i);
        }
        return 0;
    }
}
public class Test {
    public static void main(String[] args) {
        //1.创建Callable接口的实现类
        MyThread myThread = new MyThread();
        //2.创建FutureTask对象，将Callable接口的实现类作为构造方法参数传入
        FutureTask<Integer> integerFutureTask = new FutureTask<>(myThread);
        //3.创建Thread对象，将FutureTask对象作为Thread的构造方法参数传入
        Thread thread = new Thread(integerFutureTask);
        //4.调用Thread对象的start()方法启动线程
        thread.start();

        for (int j = 0; j < 10; j++) {
```

```
27              System.out.println("--j = " + j);
28          }
29      }
30  }
```

2.3.2 获得线程的返回值

代码块

```
1   package com.powernode.thread16;
2
3   import java.util.concurrent.Callable;
4   import java.util.concurrent.ExecutionException;
5   import java.util.concurrent.FutureTask;
6
7   class MyThread implements Callable<Integer> {
8       @Override
9       public Integer call() throws Exception {
10          int sum = 0;
11          for (int i = 0; i < 10; i++) {
12              sum+=i;
13          }
14          return sum;
15      }
16  }
17  public class Test {
18      public static void main(String[] args) throws ExecutionException,
    InterruptedException {
19          //1.创建Callable接口的实现类
20          MyThread myThread = new MyThread();
21          //2.创建FutureTask对象，将Callable接口的实现类作为构造方法参数传入
22          FutureTask<Integer> integerFutureTask = new FutureTask<>(myThread);
23          //3.创建Thread对象，将FutureTask对象作为Thread的构造方法参数传入
24          Thread thread = new Thread(integerFutureTask);
25          //4.调用Thread对象的start()方法启动线程
26          thread.start();
27          //5.获取Callable接口的实现类的返回值
28          Integer sum = integerFutureTask.get();
29          System.out.println("sum = " + sum);
30      }
31  }
```

2.3.3 get方法阻塞特性

```java
package com.powernode.thread17;

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

class MyThread implements Callable<Integer> {
    @Override
    public Integer call() throws Exception {
        int sum = 0;
        for (int i = 0; i < 10; i++) {
            sum+=i;
            System.out.println("i = " + i);
        }
        return sum;
    }
}
public class Test {
    public static void main(String[] args) throws ExecutionException,
InterruptedException {
        //1.创建Callable接口的实现类
        MyThread myThread = new MyThread();
        //2.创建FutureTask对象，将Callable接口的实现类作为构造方法参数传入
        FutureTask<Integer> integerFutureTask = new FutureTask<>(myThread);
        //3.创建Thread对象，将FutureTask对象作为Thread的构造方法参数传入
        Thread thread = new Thread(integerFutureTask);
        //4.调用Thread对象的start()方法启动线程
        thread.start();
        //5.获取Callable接口的实现类的返回值
        /**
         * 1.如果线程执行到get方法，则会阻塞当前线程，直到获取到返回值
         * 2.这就是线程的阻塞特点(阻塞主线程)
         */
        Integer sum = integerFutureTask.get();
        System.out.println("sum = " + sum);
        for (int j = 0; j < 10; j++) {
            System.out.println("--j = " + j);
        }
    }
}
```

2.3.4 get方法异常处理

代码块

```java
package com.powernode.thread18;
```

```
2
3   import java.util.concurrent.Callable;
4   import java.util.concurrent.ExecutionException;
5   import java.util.concurrent.FutureTask;
6
7   class MyThread implements Callable<Integer> {
8
9       @Override
10      public Integer call() throws Exception {
11          return 5 / 0;
12      }
13  }
14  public class Test {
15      public static void main(String[] args) {
16          MyThread myThread = new MyThread();
17          FutureTask<Integer> integerFutureTask = new FutureTask<Integer>
    (myThread);
18          new Thread(integerFutureTask).start();
19
20          try {
21              /**
22               * 1.integerFutureTask.get();可以获得call方法的返回值
23               * 2.call方法会抛出异常
24               * 3.call方法抛出的异常会被封装成ExecutionException
25               * 4.所以调用get方法时，必须处理ExecutionException
26               * 5.get方法也会抛出InterruptedException（后面将）
27               */
28              Integer num = integerFutureTask.get();
29          } catch (InterruptedException e) {
30              System.out.println("InterruptedException");
31          } catch (ExecutionException e) {
32              System.out.println("ExecutionException");
33
34          }
35
36      }
37  }
```

3. 线程的常用方法

3.1 基本方法

代码块

```
1   package com.powernode.thread19;
```

```java

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

class MyThread01 extends Thread {
    public MyThread01(String name) {
        super(name);
    }

    @Override
    public void run() {
        for (int i = 0; i < 10; i++) {
            System.out.println("i = " + i);
        }
    }
}
class MyThread02 implements Runnable {
    @Override
    public void run() {
        for (int i = 0; i < 10; i++) {
            System.out.println("i = " + i);
        }
    }
}
class MyThread03 implements Callable<Integer> {
    @Override
    public Integer call() throws Exception {
        for (int i = 0; i < 10; i++) {
            System.out.println("i = " + i);
        }
        return -1;
    }
}
public class Test {
    public static void main(String[] args) {
        //1.获得线程的名称和状态
        Thread thread = Thread.currentThread();
        System.out.println(thread.getName());//获得线程名称
        System.out.println(thread.getState());//获得线程状态

        //方式一
        MyThread01 myThread01 = new MyThread01("T1");
        System.out.println(myThread01.getName());
        System.out.println(myThread01.getState());
        //方式二
        MyThread02 myThread02 = new MyThread02();
        Thread thread1 = new Thread(myThread02, "T2");
```

```
        System.out.println(thread1.getName());
        System.out.println(thread1.getState());

        //方式三
        MyThread03 myThread03 = new MyThread03();
        FutureTask<Integer> integerFutureTask = new FutureTask<Integer>
(myThread03);
        Thread thread2 = new Thread(integerFutureTask, "T3");
        System.out.println(thread2.getName());
        System.out.println(thread2.getState());
    }
}
```

3.2 sleep方法

代码块

```
package com.powernode.thread20;

class MyThread extends Thread {
    @Override
    public void run() {
        for (int i = 0; i < 10; i++) {
            System.out.println("i = " + i);
            try {
                /**
                 * 1.给当前线程指定睡眠时间，让当前线程进入阻塞
                 * 2.睡眠时间结束，当前线程进入就绪状态
                 * 3.由CPU调度，由就绪状态的线程进入运行状态
                 * 4.由运行状态的线程执行完毕，进入阻塞状态
                 * 备注：单位为毫秒，1秒 = 1000毫秒
                 */
                Thread.sleep(1000);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
}
public class Test {
    public static void main(String[] args) {
        MyThread t = new MyThread();
        t.start();

        for (int j = 0; j < 10; j++) {
            System.out.println("--j = " + j);
```

```
        }
    }
}
```

3.3 interrupt方法

代码块

```
package com.powernode.thread21;
class MyThread extends Thread {
    @Override
    public void run() {
        System.out.println("线程启动了");
        try {
            Thread.sleep(5000);
        } catch (InterruptedException e) {
            e.printStackTrace();
            System.out.println("线程睡眠被中断了");
        }
        for (int i = 0; i < 10; i++) {
            System.out.println("i = " + i);
        }
    }
}
public class Test {
    public static void main(String[] args) {
        MyThread t = new MyThread();
        t.start();
        t.interrupt();//唤醒正在睡眠的线程
    }
}
```

3.4 yield方法

代码块

```
package com.powernode.thread22;

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

    @Override
    public void run() {
```

```
10          for (int i = 0; i < 10; i++) {
11              System.out.println("i = " + i);
12          }
13      }
14  }
15  public class Test {
16      public static void main(String[] args) throws Exception{
17          MyThread myThread = new MyThread("线程1");
18          myThread.start();
19          for (int j = 0; j < 10; j++) {
20              System.out.println("--j = " + j);
21              /**
22               * 1.yield()方法，提示CPU当前线程由运行状态切换到就绪状态
23               * 2.yield()方法，当前线程属于就绪状态，CPU调动时，仍然有可能调度到它
24               */
25              Thread.yield();
26              //Thread.sleep(1000);线程睡眠1秒
27          }
28      }
29  }
```

3.5 sleep和yield方法的区别

1. sleep方法
   a. 调用sleep方法后，当前线程从运行状态，进入阻塞状态，直到指定的睡眠时间结束，线程才会进入就绪状态
   b. 在睡眠的时间内，不会占用CPU资源，并且CPU不会调度它，除非指定的睡眠时间结束，或者被其他线程打断（InterruptedException）
   c. sleep方法被调用，它的效果比较明显，在没有被打断的情况下，指定的睡眠时间后，线程才进入就绪状态
2. yield方法
   a. 调用yield方法后，当前线程从运行状态，进入就绪状态，CPU在调度时，仍有可能调度到它
   b. 调用yield方法后，可能该线程又马上执行，具体什么时间执行，取决于CPU调动策略
   c. 调用yield方法后，它的效果不明显，因为它仍然有可能很快被调用