Java 多线程的实现

- 继承 Thread
- 实现 Runnable

线程调度

- 线程休眠

让当前线程暂停执行，从运行状态进入阻塞状态，将 CPU 资源让给其他线程的调度方式，通过 sleep() 来实现。

sleep(long millis)，调用时需要传入休眠时间，单位为豪秒。

```
package com.southwind.test;

public class MyThread extends Thread{
  @Override
  public void run() {
    // TODO Auto-generated method stub
    for(int i=0;i<10;i++) {
      if(i == 5) {
        try {
          sleep(5000);
        } catch (InterruptedException e) {
          // TODO Auto-generated catch block
          e.printStackTrace();
        }
      }
      System.out.println(i+"---------MyThread");
    }
  }
}
```

也可以在类的外部调用 sleep 方法。

```
MyThread2 thread = new MyThread2();
try {
  thread.sleep(5000);
} catch (InterruptedException e) {
  // TODO Auto-generated catch block
  e.printStackTrace();
}
thread.start();
```

在外部调用需要注意，休眠一定要放在启动之前。

如何让主线程休眠？直接通过静态方式调用 sleep 方法。

```
package com.southwind.test;

public class Test2 {
  public static void main(String[] args) {
    for(int i=0;i<10;i++) {
      if(i == 5) {
        try {
          Thread.sleep(3000);
        } catch (InterruptedException e) {
          // TODO Auto-generated catch block
          e.printStackTrace();
        }
      }
      System.out.println(i+"+++++Test2+++++");
    }
  }
}
```

```
public static native void sleep(long millis) throws InterruptedException;
```

sleep 是静态本地方法，可以通过类调用，也可以通过对象调用，方法定义抛出 InterruptedException，InterruptedException 继承 Exception，外部调用时必须手动处理异常。

- 线程合并

合并是指将指定的某个线程加入到当前线程中，合并为一个线程，由两个线程交替执行变成一个线程中的两个自线程顺序执行。

通过调用 join 方法来实现合并，具体如何合并?

线程甲和线程乙，线程甲执行到某个时间点的时候调用线程乙的 join方法，则表示从当前时间点开始 CPU 资源被线程乙独占，线程甲进入阻塞状态，直到线程乙执行完毕，线程甲进入就绪状态，等待获取 CPU 资源进入运行状态。

join 方法重载，join() 表示乙线程执行完毕之后才能执行其他线程，join(long millis) 表示乙线程执行 millis 毫秒之后，无论是否执行完毕，其他线程都可以和它争夺 CPU 资源。

```java
package com.southwind.test;

public class JoinRunnable implements Runnable {

  @Override
  public void run() {
    // TODO Auto-generated method stub
    for(int i=0;i<200;i++) {
      System.out.println(i+"------JoinRunnable");
    }
  }

}
```

```java
package com.southwind.test;

public class JoinTest {
  public static void main(String[] args) {
    /**
     * 两个线程，主线程、join线程
     * 主线程的逻辑：当i==10，join线程合并到主线程中
     */
    JoinRunnable joinRunnable = new JoinRunnable();
    Thread thread = new Thread(joinRunnable);
    thread.start();
    for(int i=0;i<100;i++) {
      if(i == 10) {
        try {
          thread.join();
        } catch (InterruptedException e) {
          // TODO Auto-generated catch block
          e.printStackTrace();
        }
      }
      System.out.println(i+"main+++++++++");
    }
  }
}
```

```java
package com.southwind.test;

public class JoinRunnable2 implements Runnable {

  @Override
  public void run() {
    // TODO Auto-generated method stub
```

```
    for(int i=0;i<20;i++) {
      try {
        Thread.sleep(1000);
      } catch (InterruptedException e) {
        // TODO Auto-generated catch block
        e.printStackTrace();
      }
      System.out.println(i+"--------JoinRunnable");
    }
  }

}
```

```
package com.southwind.test;

public class Test2 {
  public static void main(String[] args) {
    for(int i=0;i<10;i++) {
      if(i == 5) {
        try {
          Thread.sleep(3000);
        } catch (InterruptedException e) {
          // TODO Auto-generated catch block
          e.printStackTrace();
        }
      }
      System.out.println(i+"+++++Test2+++++");
    }
  }
}
```

线程礼让

线程礼让是指在某个特定的时间点，让线程暂停抢占 CPU 资源的行为，运行状态/就绪状态---》阻塞状态，将 CPU 资源让给其他线程来使用。

假如线程甲和线程乙在交替执行，某个时间点线程甲做出了礼让，所以在这个时间节点线程乙拥有了 CPU 资源，执行业务逻辑，但不代表线程甲一直暂停执行。

线程甲只是在特定的时间节点礼让，过了时间节点，线程甲再次进入就绪状态，和线程乙争夺 CPU 资源。

通过 yield 方法实现。

```
package com.southwind.yield;

public class YieldThread1 extends Thread {
```

```
  @Override
  public void run() {
    // TODO Auto-generated method stub
    for(int i = 0; i < 10;i++) {
      if(i == 5) {
        yield();
      }
      System.out.println(Thread.currentThread().getName()+"-----"+i);
    }
  }
}
```

```
package com.southwind.yield;

public class YieldThread2 extends Thread {
  @Override
  public void run() {
    // TODO Auto-generated method stub
    for(int i=0;i<10;i++) {
      System.out.println(Thread.currentThread().getName()+"======"+i);
    }
  }
}
```

```
package com.southwind.yield;

public class Test {
  public static void main(String[] args) {
    YieldThread1 thread = new YieldThread1();
    thread.setName("线程1");
    YieldThread2 thread2 = new YieldThread2();
    thread2.setName("线程2");
    thread.start();
    thread2.start();
  }
}
```

线程中断

有很多种情况会造成线程停止运行：

线程执行完毕自动停止

线程执行过程中遇到错误抛出异常并停止

线程执行过程中根据需求手动停止

Java 中实现线程中断有如下几个常用方法：

- public void stop()

- public void interrupt()
- public boolean isInterrupted()

stop 方法在新版本的 JDK 已经不推荐使用，重点关注后两个方法。

interrupt 是一个实例方法，当一个线程对象调用该方法时，表示中断当前线程对象。

每个线程对象都是通过一个标志位来判断当前是否为中断状态。

isInterrupted 就是用来获取当前线程对象的标志位：true 表示清除了标志位，当前线程已经中断；false 表示没有清除标志位，当前对象没有中断。

当一个线程对象处于不同的状态时，中断机制也是不同的。

创建状态：实例化线程对象，不启动。

```
package com.southwind.interrupted;

public class Test {
  public static void main(String[] args) {
    Thread thread = new Thread();
    System.out.println(thread.getState());
    thread.interrupt();
    System.out.println(thread.isInterrupted());
  }
}
```

```
<terminated> Test (153) [Java Application] /Library/Jav
NEW
false
```

NEW 表示当前线程对象为创建状态，false 表示当前线程并未中断，因为当前线程没有启动，不存在中断，不需要清除标志位。

匿名内部类

```
Thread thread = new Thread(new Runnable() {

  @Override
  public void run() {
    // TODO Auto-generated method stub
    for(int i = 0; i < 10;i++) {
      System.out.println(i+"---main");
    }
  }
});
thread.start();
```

```
package com.southwind.interrupted;

public class Test2 {
  public static void main(String[] args) {
//    MyRunnable runnable = new MyRunnable();
//    Thread thread = new Thread(runnable);
//    thread.start();

    Thread thread = new Thread(new Runnable() {

      @Override
      public void run() {
        // TODO Auto-generated method stub
        for(int i = 0; i < 10;i++) {
          System.out.println(i+"---main");
        }
      }
    });
    thread.start();
    System.out.println(thread.getState());
    thread.interrupt();
    System.out.println(thread.isInterrupted());
    System.out.println(thread.getState());
  }
}
```