并发与多线程

1，进程和线程

进程让操作系统的并发成为可能，而线程让进程内部的并发成为可能。一个进程虽然包括包括多个线程，但是这些线程是共享进程占有的资源和地址空间的。

进程是操作系统进行资源分配的基本单位，而线程是操作系统进行调度的基本单位

2，上下文切换

cpu在运行一个线程的过程中，转而去运行另外一个线程，这个叫上下文切换.一般来说，线程的上下文切换中会记录程序计数器，CPU寄存器状态等数据。虽然多线程可以使得人物的执行效率得到提高，但是由于线程的切换时同样会带来一定的开销代价，并且多个线程会导致系统资源占用的增加。

3，线程创建的几种方式

```
public class _Thread{
    public static void main(String args[]) {
        new Thread(){
            @Override
            public void run() {
                System.out.println("通过Thread类来创建线程！！！");
            }
        }.start();
        Thread y=new Thread(new Runnable(){
            @Override
            public void run() {
                System.out.println("通过实现Runnable接口来创建线程！！！");
            }
        });
        y.start();

    }
}
```

使用Callable接口也可实现线程，Callable可以从线程中返回内容

```
package test1;

import java.util.concurrent.Callable;
import java.util.concurrent.ExecutionException;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.Future;
import java.util.concurrent.FutureTask;
```

```
public class T1 {
    public static void main(String[] args) throws InterruptedException,
ExecutionException {
        Mycallable aMycallable=new Mycallable();
        ExecutorService executorService=Executors.newCachedThreadPool();
        //使用Future
        Future<String> xFuture=executorService.submit(aMycallable);
        System.out.println(xFuture.get());

        //使用FutureTask
        FutureTask<String> xFutureTask=new FutureTask<String>(aMycallable);
        new Thread(xFutureTask).start();
        System.out.println(xFutureTask.get());


    }
}
class Mycallable implements Callable<String>{
    @Override
    public String call() throws Exception {
        // TODO Auto-generated method stub
        return "hello";
    }
}
```

通过start()方法启动一个线程之后，若线程获得了CPU执行的时间，便进入run()方法去执行具体的任务。start()方法的作用就是通知"线程规划器"，该线程已经准备就绪，以便让系统安排一个时间来调用其run()方法，也就是使线程得到运行。

4，Thread类中API函数

1，start()方法

用来启动一个线程，当调用该方法时，相应的线程就会进入就绪状态，该线程中的run()方法会在某个时机被调用。

2，run()方法

首先Thread类是继承了Runnable接口，并且在用Thread创建线程的时候，一定要重写run函数，重写有两种方式：

- 直接重写：也就是说直接重写run()方法，就是上面代码中线程创建的第一种方法。
- 间接重写：重新写Runable中run的方法，可以从Thread的run()方法中可以看出，当没有重写run方法是，那么就会执行创建线程时传入的Runable对象的run方法。

  Thread的run方法如下：

  ```
  public void run() {
        if (target != null) {
            target.run();//target就是我们创建线程的时候传入的Runnable对象
        }
    }
  ```

3，sleep()方法

该方法时让当前正在执行的线程睡眠，并交出CPU让其去执行其他的任务，但是不会交出锁资源当线程睡眠时间到了之后，不一定就会立即执行，因为此时CPU可能在执行其他的任务，所以说：调用sleep方法相当于让线程进入阻塞状态。

4,yield()方法

调用该方法会让当前线程交出CPU资源，让CPU去执行其他的线程。不能控制具体交出的时间，该方法也不会交出锁资源，注意：

- yield()方法只能让拥有相同优先级的线程有机会获取CPU执行的机会
- 调用yield()方法不会让该线程进入到阻塞状态，而是进入到就绪状态，它只需要重新得到CPU的执行。

5，join()方法

当主线程中出现Thread.join()的时候，主线程会释放锁资源主线程会等待Thread线程执行完或者执行一段时间之后在会接着执行。

join()方法其实是通过wait()方法来实现的,理解：main线程去获取子线程对象的锁，但是由于子线程再运行，那么main线程就会一直处于阻塞状态，直到子线程运行完之后，主线程才会接着运行

6，interrupt()方法

单独调用interrupt方法可以使得处于阻塞状态的线程抛出一个异常，因此可以用该方法来判断一个线程是否处于阻塞状态，interrupted()和isinterrupted()方法可以停止正在运行的线程。interrupt()方法可以中断阻塞状态中的线程，但是无法中断正在运行的线程，但是可以通过interrupt()方法来对线程的中断标志位变成true，通过isInterrupted()方法来对线程中断。

中断线程方法：

第一种：通过interrupt()方法和isInterrupted()方法来对线程进行终中断。

```
public class _interrupt{
    public static void main(String[] args){
        Thread x=new Thread(){

            @Override
            public void run() {
                int i=0;
                while(!isInterrupted()&&i<=Integer.MAX_VALUE){
                    System.out.print(i+"->");
                    i++;
                }
            }
        };
        x.start();
        try{
            Thread.currentThread().sleep(100);
        } catch(Exception e){

        }
        x.interrupt();  //令x线程的interrupt标志为true，通过isInterrupted()方法来进行判断线程的中断标志位是否为true来中止线程的运行。
    }
```

```
}
```

第二种：使用一个volatitle的变量来进行对线程的中止

```
public class _interrupt2{
    public static void main(String[] args){
        MyThread x=new MyThread();
        x.start();
        try{
            Thread.currentThread().sleep(10);
        } catch(Exception e){
        }
        x.StopThread();    //停止线程
    }
}
class MyThread extends Thread{
    public volatile boolean flog=false;    //volatitle可以实现线程之间的数据的可见性。
    @Override
    public void run() {
        int i=0;
        while(!flog){
            System.out.print(i+"->");
            i++;
        }
    }
    public void StopThread(){
        flog=true;
    }
}
```

7,守护进程的函数(setDaemon函数)

守护线程是一种特殊的线程，当进程中不存在非守护线程的时候，那么守护线程就会自动销毁。典型的守护线程就是垃圾回收线程，main线程是由JVM建立的非守护线程。

```
public class _Test{
    public static void main(String[] srgs){
        Thread x=new Thread(){
          @Override
          public void run() {
              while(true){
                  System.out.println("ALive");
              }
          }
        };
        x.setDaemon(true);    //将x设置成守护进程，当main线程结束的时候，在进程也会跟着结束。
        x.start();
        try{
            Thread.currentThread().sleep(1000);
        } catch(Exception e){

        }
```

```
    }
}
```

5,线程的优先级

- 线程的优先级具有继承性，比如A线程启动B线程，那么B线程的优先级和A的优先级高。
- 线程的优先级具有一定的规则性，CPU尽量将执行资源让给优先级较高的线程