一: 观察者模式

背景: 当线程在运行的时候，如果没有使用线程池等包的时候，那么直接new Thread这种方法的话，是很难观察到线程目前的状态的，如果知道当前这个线程是运行还是停止，因此想要知道整个过程的生命周期的话，那么观察者模式就是一种很好的方式了。

需求分析:

1. 这里如果想观察一下运行过程的周期的话，那么肯定需要一个类来定义任务生命周期的，那么这里一般如果用枚举类型的话就比较适合了，而这个类使用接口的话就方便扩展，因此首先需要一个接口来定义任务的生命周期的枚举类型。
2. 这里如果想观察任务的生命周期，那么应该有一些公共的方法来控制任务的启动停止与报错等，因此这里需要一个类来定义这些公共方法，同样为了方便扩展，这里使用接口更好，因为后期如果用户想重写这些方法的时候就更方便一点。但是这里如果是单纯接口的话，那么如果用户想实现就不方便了，因此可以考虑在接口里面实现当前的接口，然后写空实现，这样的话就方便后期重写接口了。
3. 然后这里关键的还需一个运行任务用的，也就是任务执行的接口和一个任务监控的类，这个类需要继承Thread类，同时实现第一点的接口，这里把第二点的接口当做一个类成员变量，另外还有一些其他的方法等。见如下代码所示:

1) 任务生命周期枚举类型接口:

```
public interface Observable {
    //任务生命周期的枚举类型
    enum Cycle{
        STARTED,RUNNING,DONE,ERROR
    }

    //获取当前任务的生命周期状态
    Cycle getCycle();

  //定义启动线程的方法，主要作用是为了屏蔽Thread的其他方法
    void start();

 //定义线程的中断方法，作用与start方法都是屏蔽Thread的方法
    void interrupt();
}
```

这里主要定义了一些任务的生命周期的枚举类型

2) 生命周期控制相关接口及空实现

```
 public interface TaskLifecycle<T> {

    //任务启动时会触发onStart方法
    void onStart(Thread thread);

    //任务正在运行时会触发onRunning方法
    void onRunning(Thread thread);

    //任务运行结束时会触发onFinish方法，其中result是任务执行结束后的结果
    void onFinish(Thread thread,T result);

    //任务执行报错时会触发onError方法
    void onError(Thread thread,Exception e);

    //生命周期接口的空实现
class EmptyLifecycle<T> implements TaskLifecycle<T>{
    @Override
    public void onStart(Thread thread) { }

    @Override
    public void onRunning(Thread thread) { }

    @Override
    public void onFinish(Thread thread, T result) { }

    @Override
    public void onError(Thread thread, Exception e) { }

  }
}
```

这里的空实现的好处是后面调用测试的时候，用户就可以比较方便地重写方法了。

3) 这个接口很简单，就是实现一下任务执行的接口

```
public interface Task<T> { //任务执行接口,该接口允许有返回值 T call(); }
```

4) 这是真正地实现任务观察等代码的类

```
  public class ObservableThread<T> extends Thread implements Observable{

   private final TaskLifecycle<T> lifecycle;
   private Cycle cycle;
   private final Task<T> task;

   //指定Task的实现,默认情况下使用EmptyLifecycle
   public ObservableThread(Task<T> task) {
       this(new TaskLifecycle.EmptyLifecycle<>(),task);
   }

   public ObservableThread(TaskLifecycle<T> lifecycle,Task<T> task) {
       super();
       //Task不允许为null
       if(null == task) {        throw new IllegalArgumentException("The task is required");}
       this.lifecycle = lifecycle;
       this.task = task;
   }


   @Override
   public final void run() {
       //在执行线程逻辑单元的时候，分别触发相应的事件
       this.update(Cycle.STARTED, null,null);
       try {
           this.update(Cycle.RUNNING, null, null);
           //真正执行任务的方法
           T result = this.task.call();
           this.update(Cycle.DONE, result, null);
       }catch(Exception e) {
           this.update(Cycle.ERROR, null, e);
       }
   }

   private void update(Cycle cycle,T result,Exception e) {
       this.cycle = cycle;
       if(lifecycle == null) { return;}

       try {
           switch(cycle) {
               case STARTED:
                   this.lifecycle.onStart(currentThread());
                   break;
               case RUNNING:
                   this.lifecycle.onRunning(currentThread());
                   break;
               case DONE:
                   this.lifecycle.onFinish(currentThread(), result);
                   break;
               case ERROR:
                   this.lifecycle.onError(currentThread(), e);
                   break;
           }
       }catch(Exception ex) {
           if(cycle == Cycle.ERROR) {
               throw ex;
           }
       }
   }

     @Override
     public Cycle getCycle() { return this.cycle;  }
}
```

这里可以看到只是实现了Observable接口，其他的两个接口作为内部成员，而为什么这里只实现当前这个接口呢？因为这个接口的功能就是在任务运行状态改变的时候，需要定制化开发的，因此要重写就需要实现了。而另外两个接口，其中Task只是把运行的代码封装起来使用call调用而已，这里call其实并没有重写这个方法，因此本质上这里没有做任务的处理。但是后面可以自定义一下。而TaskLifecycle则是用于在update方法更改任务的状态的时候使用。

5) 测试接口

```
 public class ObservableTest {
   public static void main(String[] args) {
   final TaskLifecycle<String> lifecycle = new TaskLifecycle.EmptyLifecycle<String>() {
       public void onFinish(Thread thread,String result) {
           System.out.println("The result is :"+result);
       }

       public void onStart(Thread thread) {
           System.out.println(thread.getName()+"正在启动");
```

```
        }
    };

    Observable observableThread = new ObservableThread<>(lifecycle,() -> "运行任务");
    observableThread.start();
  }
  }
```

这里可以看到，在定义TaskLifecycle的时候重写了onFinish方法和onStart方法，这样的话就可以定制化开发了。比较方便。另外同时运行的结果为:

```
 Thread-0正在启动
The result is :运行任务
```

另外这里给一篇参考文档，其中也有关于普通方法的观察者模式的代码，可以对比观看一下。
https://www.cnblogs.com/luonote/p/10404316.html