前言

阅读和学习本文内容，请先阅读《Java 编程思想（第四版）》中的第十章：内部类。

我一直觉得，往往学 100 个知识点才能获得一个新的思想升华。只有把内部类的方方面面都学完了，山穷水尽疑无路 柳暗花明又一村，只有这样才能获得突破和飞跃。

函数转内部类实现异步执行的设计思想

我一直觉得:只做 Java 的人,永远成不了 Java 高手。下面内容是顿悟出来的,是基于 Python 语言的特性：函数，也是一种对象。立足这个思想，将 Java 里面函数，升华成内部类，而且让原功能带上线程池+异步执行的特性。具体如何实现呢？请看下文：

类里面包括两种物质：属性和函数。在某些编程语言里面，将函数也当做一个对象。从这个角度来看，函数就像一个内部类。

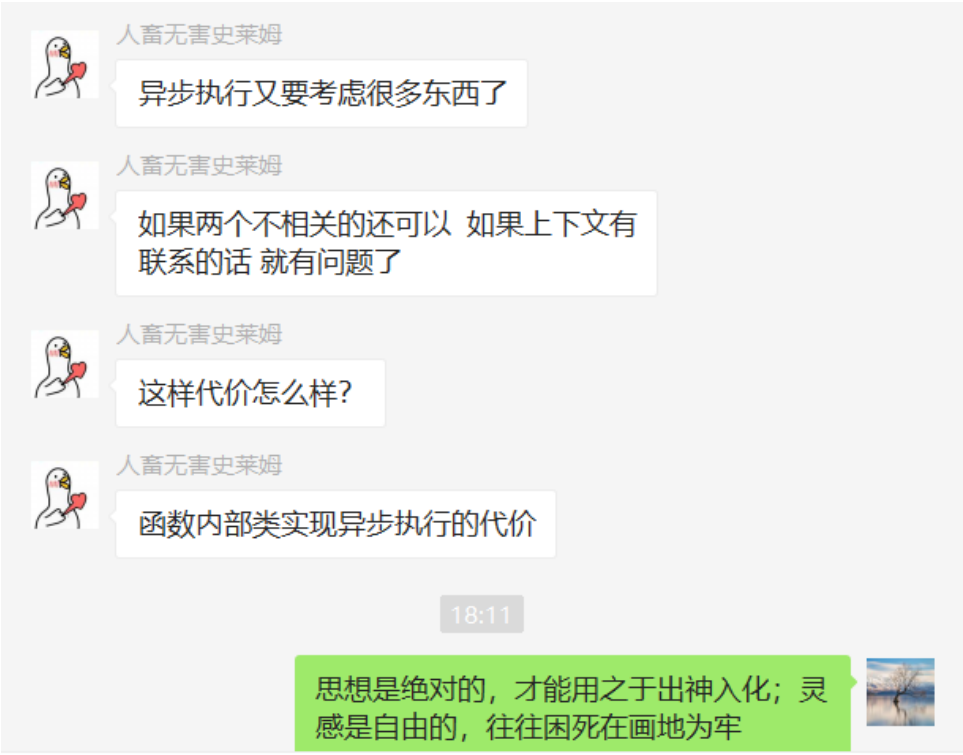
把函数当做内部类，这种想法别出心裁，挺新颖的，但是并没有特别使用之处。可以将其彻底升华一下，从而将这种想法发挥出巨大威力，以提升代码的设计能力和思想境界：

（1）函数变成内部类，而这个内部类 implements Runnable，给函数的执行增加了异步的功能。这个内部类，可以是普通的类，也可以看做一个事件类
（2）函数变成内部类之后，之前函数的参数变成内部类的属性
（3）函数变成内部类之后，之前函数的函数体不变，而此时内部类提供 run 方法，去调用旧的函数体。
（4）函数变成内部类之后，外部类增加线程池，外部类执行函数的过程变成了：将参数封装成内部类，然后扔到线程池，进行异步执行的过程。

以上思想的代码落地，与《编码手法：函数调用变事件队列的异步处理的写作手法鉴赏》类似，见下文介绍。

另外，有的人提了一堆的问题，对此类异议不再一一回复，统一回答为：

思想是绝对的，才能用之于出神入化；灵感是自由的，往往困死在画地为牢。



编码手法：函数调用变事件队列的异步处理的写作手法鉴赏

1、函数的深入分析

首先，我们要把函数当做一个高度复杂的生命体来看待，把它解剖一下，看看其内部的各个零件或者器官。以下面函数为例：

```
public class Printer
{
    public void print(String msg)
    {
        System.out.println(msg);
    }
}
```

其组成部分包括：

（1）函数的参数，即：String msg

（2）参数的名称，即：print

以上两部分可以包装成事件和事件类型，如下所示：

```
//事件类型
public enum EventType
{
    PRINT, //打印
    COPY //扫描
}

public class Event<T> extends EventObject implements Serializable
{
    private static final long serialVersionUID = 1L;

    private EventType eventType = EventType.PRINT;

    public Event(T param, EventType type)
    {//注意：param表示函数的参数
        super(param);
        this.eventType = type;
    }

    public EventType getEventType()
    {
        return eventType;
    }

    @SuppressWarnings("unchecked")

    public T getParam()
    {
        return (T) super.getSource();

    }
}
```

（3）函数的执行体，可以包装成事件处理器：

```
//事件处理器接口
public interface IEventHandler
{
    <T> void handleEvent(Event<T> event);
}


//具体的事件处理器实现类
```

```
public class PrintEventHandler implements IEventHandler
{

    private Printer printer = new Printer();

    public <T> void handleEvent(Event<T> event)
    {
        String param = (String) event.getParam();

        printer.print(param);
    }
}
```

（4）函数的调用过程：函数参数+函数执行，可以包装成事件运行器：

```
//事件运行器
public class EventRunner implements Runnable
{
    private final IEventHandler handler;

    private final Event<?> event;

    EventRunner(IEventHandler hand, Event<?> event)
    {
        this.handler = hand;
        this.event = event;
    }

    public void run()
    {
        handler.handleEvent(event);
    }
}
```

（5）函数的附着体，原先是 Printer，而现在变成了事件队列：

```
//事件队列
public class EventQueue
{
    private static final String THREAD_PREFIX = "EventQueue";

    private boolean destroyed = false;
```

```
    private LinkedBlockingQueue<Runnable> queue;

    private ThreadPoolExecutor queueProcessor;

    public EventQueue()
    {
        queue = new LinkedBlockingQueue<Runnable>();

        queueProcessor = new ThreadPoolExecutor(1, 1, 0L,
TimeUnit.MILLISECONDS, queue,
                new EventThreadFactory(THREAD_PREFIX));

    }

    public void addEventRunner(EventRunner runner)
    {

        if (!destroyed)
        {
            queueProcessor.execute(runner);
        }
    }

    public void dispose()
    {
        if (!destroyed)
        {
            destroyed = true;

            queueProcessor.shutdownNow();
            queueProcessor = null;
        }
    }
}
```

（6）补充：线程工厂类

```
import java.util.concurrent.ThreadFactory;

public class EventThreadFactory implements ThreadFactory
{
    private String prefix;
    private boolean threadIsDaemon = true;
    private int threadPriority = Thread.NORM_PRIORITY;
```

```
    public EventThreadFactory(String prefix)
    {
        this(prefix, Thread.NORM_PRIORITY);
    }

    public EventThreadFactory(String prefix, int threadPriority)
    {
        this.prefix = prefix;
        this.threadPriority = threadPriority;
    }

    @Override
    public Thread newThread(Runnable runner)
    {
        Thread thread = new Thread(runner);
        String name = thread.getName();
        thread.setName(prefix + name);
        thread.setDaemon(threadIsDaemon);
        thread.setPriority(threadPriority);
        return thread;
    }
}
```

2、函数的用法对比

在未改造之前，也就是未将函数变事件改造之前，函数的用法是这样的：

```
public class Client
{
    public static void main(String[] args)
    {
        Printer printer = new Printer();
        printer.print("hell world!");
    }
}
```

而改造后，函数变成了事件，异步处理的过程是这样的：

```
public class ClientEvent
{
    public static void main(String[] args)
    {
```

```
        EventQueue eventQueue = new EventQueue();

        IEventHandler handler = new PrintEventHandler();

        Event<String> event = new Event<String>("hello world",
EventType.PRINT);

        EventRunner runner = new EventRunner(handler, event);

        eventQueue.addEventRunner(runner);

        eventQueue.dispose();
    }
}
```