Spring注解驱动开发第45讲——Spring IOC容器创建源码解析(五)之初始化事件派发器

目录一览

写在前面

在上一讲中，我们已经搞清楚了如下initMessageSource方法所做的事情，它无非就是来 初始化 MessageSource组件的。

IOCTest_Ext.java | AnnotationConfigApplicationContext.class | AbstractApplicationContext.class

```
508     @Override
509     public void refresh() throws BeansException, IllegalStateException {
510         synchronized (this.startupShutdownMonitor) {
511             // Prepare this context for refreshing.
512             prepareRefresh();
513
514             // Tell the subclass to refresh the internal bean factory.
515             ConfigurableListableBeanFactory beanFactory = obtainFreshBeanFactory();
516
517             // Prepare the bean factory for use in this context.
518             prepareBeanFactory(beanFactory);
519
520             try {
521                 // Allows post-processing of the bean factory in context subclasses.
522                 postProcessBeanFactory(beanFactory);
523
524                 // Invoke factory processors registered as beans in the context.
525                 invokeBeanFactoryPostProcessors(beanFactory);
526
527                 // Register bean processors that intercept bean creation.
528                 registerBeanPostProcessors(beanFactory);
529
530                 // Initialize message source for this context.
531                 initMessageSource();
532
533                 // Initialize event multicaster for this context.
534                 initApplicationEventMulticaster();
535
536                 // Initialize other special beans in specific context subclasses.
537                 onRefresh();
538
539                 // Check for listener beans and register them
```

然后，我们让程序运行到以上第534行代码（即initApplicationEventMulticaster方法）处。顾名思义，该方法是来初始化事件派发器的。

你有没有想过这样一个问题，为什么Spring容器在创建的过程中还要调用这样一个初始化事件派发器的方法呢？没想过，就算了，这里我直接给出答案，这是因为需要一个事件派发器对我们Spring中的事件进行一些派发、管理以及通知等。

那么，究竟是如何来初始化事件派发器的呢？这时，我们就得来好好研究一下initApplicationEventMulticaster方法里面究竟做了些什么事了。

初始化事件派发器

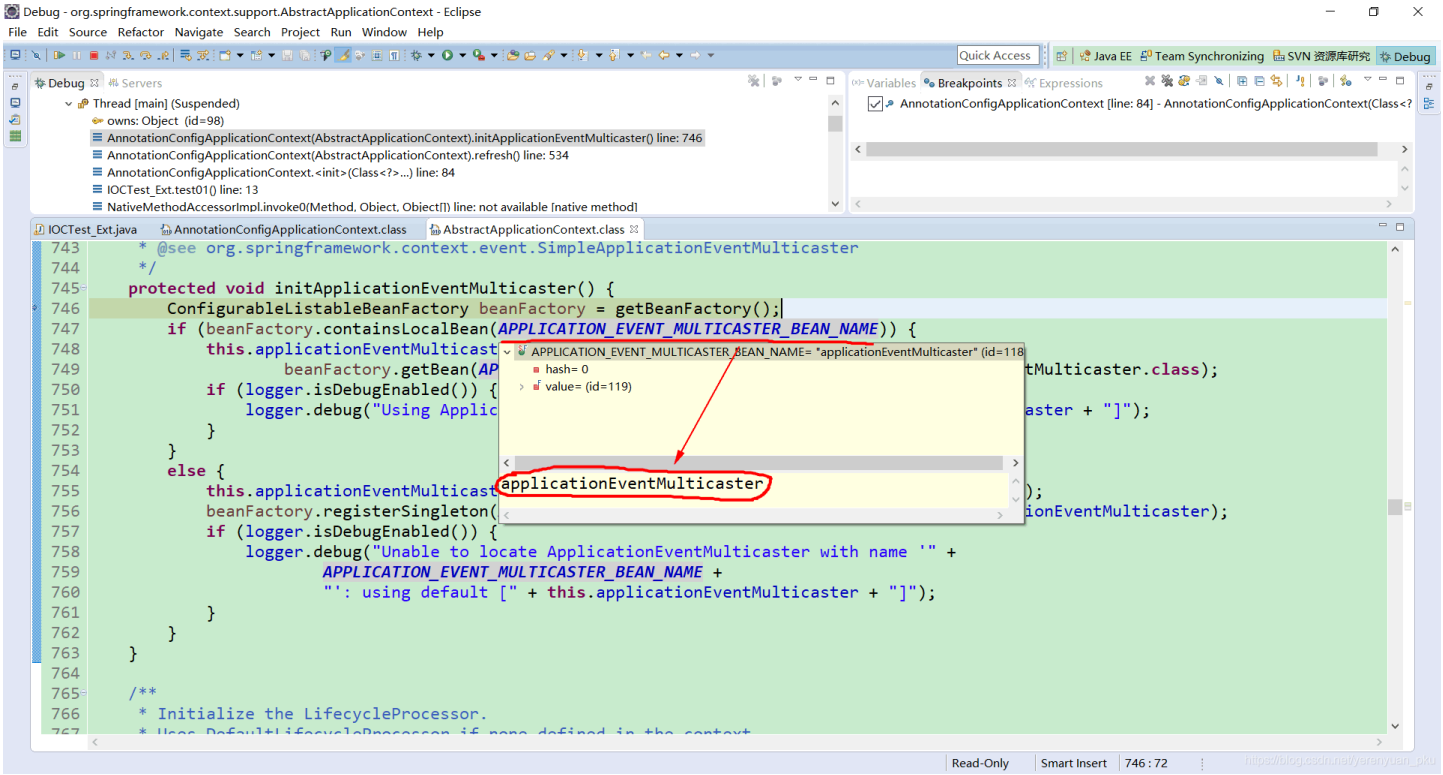
获取BeanFactory

按下 F5 快捷键进入到initApplicationEventMulticaster方法里面，如下图所示，可以看到一开始是先来获取BeanFactory的。

```java
/**
 * Initialize the ApplicationEventMulticaster.
 * Uses SimpleApplicationEventMulticaster if none defined in the context.
 * @see org.springframework.context.event.SimpleApplicationEventMulticaster
 */
protected void initApplicationEventMulticaster() {
    ConfigurableListableBeanFactory beanFactory = getBeanFactory();
    if (beanFactory.containsLocalBean(APPLICATION_EVENT_MULTICASTER_BEAN_NAME)) {
        this.applicationEventMulticaster =
                beanFactory.getBean(APPLICATION_EVENT_MULTICASTER_BEAN_NAME, ApplicationEventMulticaster.class);
        if (logger.isDebugEnabled()) {
            logger.debug("Using ApplicationEventMulticaster [" + this.applicationEventMulticaster + "]");
        }
    }
    else {
        this.applicationEventMulticaster = new SimpleApplicationEventMulticaster(beanFactory);
        beanFactory.registerSingleton(APPLICATION_EVENT_MULTICASTER_BEAN_NAME, this.applicationEventMulticaster);
        if (logger.isDebugEnabled()) {
            logger.debug("Unable to locate ApplicationEventMulticaster with name '" +
                    APPLICATION_EVENT_MULTICASTER_BEAN_NAME +
                    "': using default [" + this.applicationEventMulticaster + "]");
        }
    }
}
```


看容器中是否有id为applicationEventMulticaster，类型是ApplicationEventMulticaster的组件

按下 F6 快捷键让程序继续往下运行，会发现有一个判断，即判断BeanFactory中是否有一个id为applicationEventMulticaster的组件。我为什么会这么说呢，你只要看一下常量 APPLICATION_EVENT_MULTICASTER_BEAN_NAME 的值就知道了，如下图所示，该常量的值就是applicationEventMulticaster。



若有，则赋值给this.applicationEventMulticaster

如果有的话，那么会从BeanFactory中获取到id为applicationEventMulticaster，类型是ApplicationEventMulticaster的组件，并将其赋值给 this.applicationEventMulticaster 。这可以从下面这行代码看出。

```
743     * @see org.springframework.context.event.SimpleApplicationEventMulticaster
744     */
745    protected void initApplicationEventMulticaster() {
746        ConfigurableListableBeanFactory beanFactory = getBeanFactory();
747        if (beanFactory.containsLocalBean(APPLICATION_EVENT_MULTICASTER_BEAN_NAME)) {
748            this.applicationEventMulticaster =
749                    beanFactory.getBean(APPLICATION_EVENT_MULTICASTER_BEAN_NAME, ApplicationEventMulticaster.class);
750            if (logger.isDebugEnabled()) {
751                logger.debug("Using ApplicationEventMulticaster [" + this.applicationEventMulticaster + "]");
752            }
753        }
754        else {
755            this.applicationEventMulticaster = new SimpleApplicationEventMulticaster(beanFactory);
756            beanFactory.registerSingleton(APPLICATION_EVENT_MULTICASTER_BEAN_NAME, this.applicationEventMulticaster);
757            if (logger.isDebugEnabled()) {
758                logger.debug("Unable to locate ApplicationEventMulticaster with name '" +
759                        APPLICATION_EVENT_MULTICASTER_BEAN_NAME +
760                        "': using default [" + this.applicationEventMulticaster + "]");
761            }
762        }
763    }
764
765    /**
766     * Initialize the LifecycleProcessor.
```

也就是说，如果我们之前已经在容器中配置了一个事件派发器，那么此刻就能从BeanFactory中获取到该事件派发器了。

很显然，容器刚开始创建的时候，肯定是还没有的，所以程序会来到下面的else语句中。

若没有，则创建一个SimpleApplicationEventMulticaster类型的组件，并把创建好的组件注册在容器中

如果没有的话，那么Spring自己会创建一个SimpleApplicationEventMulticaster类型的对象，即一个简单的事件派发器。

然后，把创建好的事件派发器组件注册到容器中，即添加到BeanFactory中，所执行的是下面这行代码。

```
743     * @see org.springframework.context.event.SimpleApplicationEventMulticaster
744     */
745    protected void initApplicationEventMulticaster() {
746        ConfigurableListableBeanFactory beanFactory = getBeanFactory();
747        if (beanFactory.containsLocalBean(APPLICATION_EVENT_MULTICASTER_BEAN_NAME)) {
748            this.applicationEventMulticaster =
749                    beanFactory.getBean(APPLICATION_EVENT_MULTICASTER_BEAN_NAME, ApplicationEventMulticaster.class);
750            if (logger.isDebugEnabled()) {
751                logger.debug("Using ApplicationEventMulticaster [" + this.applicationEventMulticaster + "]");
752            }
753        }
754        else {
755            this.applicationEventMulticaster = new SimpleApplicationEventMulticaster(beanFactory);
756            beanFactory.registerSingleton(APPLICATION_EVENT_MULTICASTER_BEAN_NAME, this.applicationEventMulticaster);
757            if (logger.isDebugEnabled()) {
758                logger.debug("Unable to locate ApplicationEventMulticaster with name '" +
759                        APPLICATION_EVENT_MULTICASTER_BEAN_NAME +
760                        "': using default [" + this.applicationEventMulticaster + "]");
761            }
762        }
763    }
764
765    /**
766     * Initialize the LifecycleProcessor.
```

这样，我们以后其他组件要使用事件派发器，直接自动注入这个事件派发器组件即可。

别忘了，接下来还有两个方法呢

onRefresh()

按下 F6 快捷键让程序继续往下运行，直至运行到下面这行代码处。

```java
    public void refresh() throws BeansException, IllegalStateException {
        synchronized (this.startupShutdownMonitor) {
            // Prepare this context for refreshing.
            prepareRefresh();

            // Tell the subclass to refresh the internal bean factory.
            ConfigurableListableBeanFactory beanFactory = obtainFreshBeanFactory();

            // Prepare the bean factory for use in this context.
            prepareBeanFactory(beanFactory);

            try {
                // Allows post-processing of the bean factory in context subclasses.
                postProcessBeanFactory(beanFactory);

                // Invoke factory processors registered as beans in the context.
                invokeBeanFactoryPostProcessors(beanFactory);

                // Register bean processors that intercept bean creation.
                registerBeanPostProcessors(beanFactory);

                // Initialize message source for this context.
                initMessageSource();

                // Initialize event multicaster for this context.
                initApplicationEventMulticaster();

                // Initialize other special beans in specific context subclasses.
                onRefresh();

                // Check for listener beans and register them.
```

于是，我们按下 F5 快捷键进入到以上onRefresh方法里面去看一看，如下图所示，发现它里面是空的。

```java
    /**
     * Template method which can be overridden to add context-specific refresh work.
     * Called on initialization of special beans, before instantiation of singletons.
     * <p>This implementation is empty.
     * @throws BeansException in case of errors
     * @see #refresh()
     */
    protected void onRefresh() throws BeansException {
        // For subclasses: do nothing by default.
    }

    /**
     * Add beans that implement ApplicationListener as listeners.
```

你是不是觉得很熟悉，因为我们之前就见到过两次类似这样的空方法，一次是我们在做容器刷新前的 预处理 工作时，可以让子类自定义个性化的属性设置，另一次是在BeanFactory创建并预处理完成以后，可以让子类做进一步的设置。我的朋友，你现在记起来了吗? 😂

同理，以上onRefresh方法就是留给子类来重写的，这样是为了给我们留下一定的弹性，当子类（也可以说是子容器）重写该方法后，在容器刷新的时候就可以再自定义一些逻辑了，比如给容器中多注册一些组件之类的。

registerListeners()

继续按下 F6 快捷键让程序继续往下运行，直至运行到下面这行代码处。

```java
            try {
                // Allows post-processing of the bean factory in context subclasses.
                postProcessBeanFactory(beanFactory);

                // Invoke factory processors registered as beans in the context.
                invokeBeanFactoryPostProcessors(beanFactory);

                // Register bean processors that intercept bean creation.
                registerBeanPostProcessors(beanFactory);

                // Initialize message source for this context.
                initMessageSource();

                // Initialize event multicaster for this context.
                initApplicationEventMulticaster();

                // Initialize other special beans in specific context subclasses.
                onRefresh();

                // Check for listener beans and register them.
                registerListeners();

                // Instantiate all remaining (non-lazy-init) singletons.
                finishBeanFactoryInitialization(beanFactory);
```

按照registerListeners方法上面的注释来说，该方法是来检查监听器并注册它们的。也就是说，该方法会将我们项目里面的监听器（也即咱们自己编写的ApplicationListener）注册进来。

我们可以按下 F5 快捷键进入到以上registerListeners方法里面去看一看，如下图所示。

IOCTest_Ext.java | AnnotationConfigApplicationContext.class | AbstractApplicationContext.class

```
803     /**
804      * Add beans that implement ApplicationListener as listeners.
805      * Doesn't affect other listeners, which can be added without being beans.
806      */
807     protected void registerListeners() {
808         // Register statically specified listeners first.
809         for (ApplicationListener<?> listener : getApplicationListeners()) {
810             getApplicationEventMulticaster().addApplicationListener(listener);
811         }
812
813         // Do not initialize FactoryBeans here: We need to leave all regular beans
814         // uninitialized to let post-processors apply to them!
815         String[] listenerBeanNames = getBeanNamesForType(ApplicationListener.class, true, false);
816         for (String listenerBeanName : listenerBeanNames) {
817             getApplicationEventMulticaster().addApplicationListenerBean(listenerBeanName);
818         }
819
820         // Publish early application events now that we finally have a multicaster...
821         Set<ApplicationEvent> earlyEventsToProcess = this.earlyApplicationEvents;
822         this.earlyApplicationEvents = null;
823         if (earlyEventsToProcess != null) {
824             for (ApplicationEvent earlyEvent : earlyEventsToProcess) {
825                 getApplicationEventMulticaster().multicastEvent(earlyEvent);
826             }
827         }
828     }
829
830     /**
831      * Finish the initialization of this context's bean factory,
832      * initializing all remaining singleton beans.
833      */
834     protected void finishBeanFactoryInitialization(ConfigurableListableBeanFactory beanFactory) {
```

遍历从容器中获取到的所有的*ApplicationListener*，然后将遍历出的每一个监听器添加到事件派发器中

可以看到一开始会有一个for循环，该for循环是来遍历从容器中获取到的所有的ApplicationListener的，然后将遍历出的每一个监听器添加到事件派发器中。

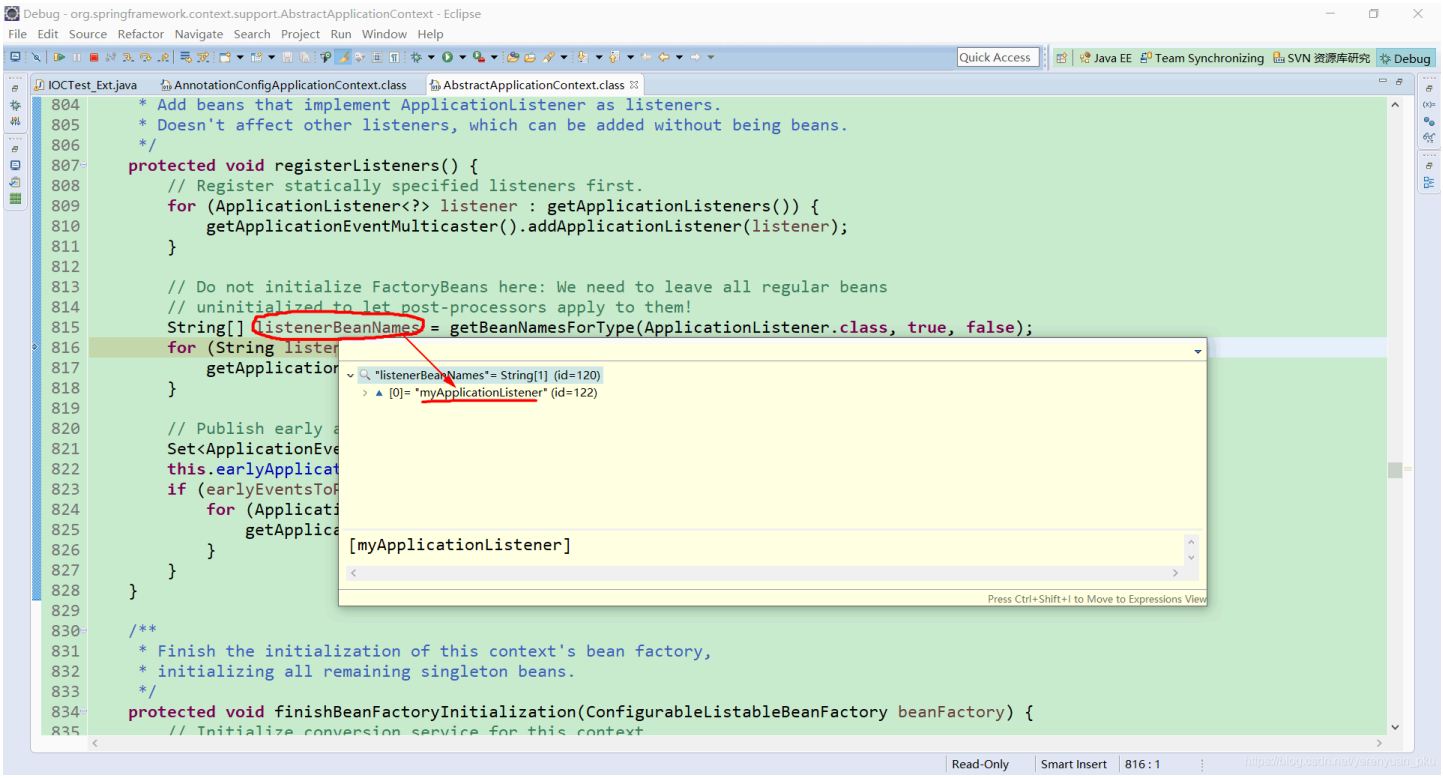
当我们按下 F6 快捷键让程序继续往下运行时，发现并没有进入for循环中，而是来到了下面这行代码处。

IOCTest_Ext.java | AnnotationConfigApplicationContext.class | AbstractApplicationContext.class

```
803     /**
804      * Add beans that implement ApplicationListener as listeners.
805      * Doesn't affect other listeners, which can be added without being beans.
806      */
807     protected void registerListeners() {
808         // Register statically specified listeners first.
809         for (ApplicationListener<?> listener : getApplicationListeners()) {
810             getApplicationEventMulticaster().addApplicationListener(listener);
811         }
812
813         // Do not initialize FactoryBeans here: We need to leave all regular beans
814         // uninitialized to let post-processors apply to them!
815         String[] listenerBeanNames = getBeanNamesForType(ApplicationListener.class, true, false);
816         for (String listenerBeanName : listenerBeanNames) {
817             getApplicationEventMulticaster().addApplicationListenerBean(listenerBeanName);
818         }
819
820         // Publish early application events now that we finally have a multicaster...
821         Set<ApplicationEvent> earlyEventsToProcess = this.earlyApplicationEvents;
822         this.earlyApplicationEvents = null;
823         if (earlyEventsToProcess != null) {
824             for (ApplicationEvent earlyEvent : earlyEventsToProcess) {
825                 getApplicationEventMulticaster().multicastEvent(earlyEvent);
826             }
827         }
828     }
829
830     /**
831      * Finish the initialization of this context's bean factory,
832      * initializing all remaining singleton beans.
833      */
834     protected void finishBeanFactoryInitialization(ConfigurableListableBeanFactory beanFactory) {
```

这是调用getBeanNamesForType方法从容器中拿到ApplicationListener类型的所有bean的名字的。也就是说，首先会从容器中拿到所有的ApplicationListener组件。

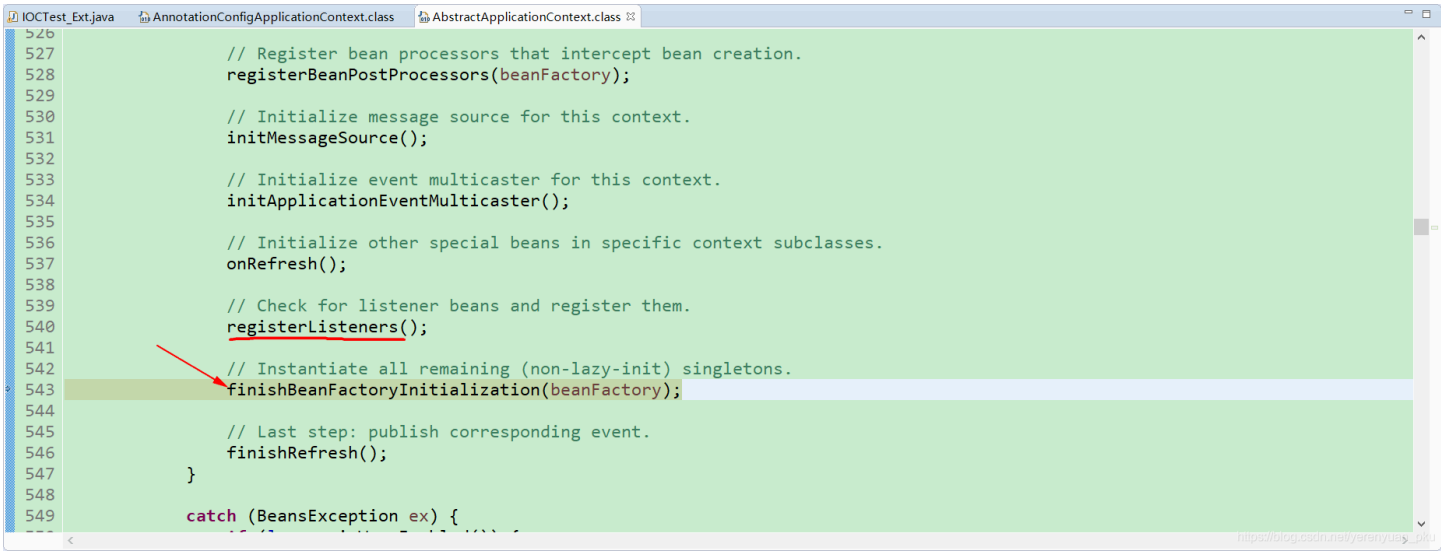
按下 F6 快捷键让程序继续往下运行，运行一步即可，这时Inspect一下listenerBeanNames变量的值，你就能看到确实是获取到了咱们自己编写的ApplicationListener了，如下图所示。



然后，将获取到的每一个监听器添加到事件派发器中。

当早期我们容器中有一些事件时，会将这些事件保存在名为earlyApplicationEvents的Set集合中。这时，会先获取到事件派发器，再利用事件派发器将这些事件派发出去。也就是说，派发之前步骤产生的事件。

而现在呢，容器中默认还没有什么事件，所以，程序压根就不会进入到下面的for循环中去派发事件。当程序运行至下面这行代码处时，registerListeners方法就执行完了，它所做的事情很简单，无非就是**从容器中拿到所有的ApplicationListener组件，然后将每一个监听器添加到事件派发器中**。



以上finishBeanFactoryInitialization方法是非常非常重要的，顾名思义，它是来初始化所有剩下的单实例bean的。执行完该方法之后，就完成BeanFactory的初始化了。我们下一讲来着重分析一下它，敬请期待哟~~~