一次线上性能问题排查：Spring监听器源码解析

- **作者：黄荣鹏-互联网应用分发事业部-应用分发技术中心-应用分发服务器组**

- **关键词：性能问题、Spring监听器、源码解析**

- **摘要：最近项目中使用了Spring监听器发送异步事件，因为使用不正当，从而导致当服务请求量上涨时，整个工程全部接口可用性均下降，域名可用性最低下降至45%。本文主要是分享线上因不正当使用Spring监听器从而导致性能问题的分析过程并对Spring监听器内部实现进行源码分析。**

一、导读

最近项目中使用了Spring监听器发送异步事件，因为使用不正当，从而导致当服务请求量上涨时，整个工程全部接口可用性均下降，域名可用性最低下降至45%。
本文主要是分享线上因不正当使用Spring监听器从而导致性能问题的分析过程并对Spring监听器内部实现进行源码分析。Spring的监听器其实就是对观察者模式的一种应用，本文是假设读者对监听器模式已经有了充分的了解，所以不会深入的介绍观察者模式，如果读者对观察者模式不熟悉，建议先了解下观察者模式再阅读本文。

二、Spring监听器的使用

在对Spring监听器进行源码分析前，我们先了解下如何使用Spring监听器。关于Spring监听器的使用，读者也可以参考这篇文章Spring Events的说明，因为我们在使用Spring监听器时也是参照了这篇文章。想要了解Spring监听器，首先要明确事件监听器的三个要素：事件、事件监听器、事件源。

1、事件：ApplicationEvent

ApplicationEvent就是充当了事件的角色，我们想要自定义事件，那么只需要继承ApplicationEvent就可以了。

```
public class CustomSpringEvent extends ApplicationEvent {
    private String message;

    public CustomSpringEvent(Object source, String message) {
        super(source);
        this.message = message;
    }
    public String getMessage() {
        return message;
    }
}
```

2、事件源：ApplicationEventPublisher

事件源也就是事件的发布者，在Spring监听器模式中我们可以借助ApplicationEventPublisher来发布事件。我们可以自定义一个publisher，然后注入ApplicationEventPublisher，并通过调用publishEvent方法来发布事件。

```
@Component
public class CustomSpringEventPublisher {
    private static final Logger logger = LoggerFactory.getLogger(CustomSpringEventPublisher.class);

    @Autowired
    private ApplicationEventPublisher applicationEventPublisher;

    public void publishCustomEvent(final String message) {
        logger.info("Publishing custom event. ");
        CustomSpringEvent customSpringEvent = new CustomSpringEvent(this, message);
        applicationEventPublisher.publishEvent(customSpringEvent);
    }
}
```

3、事件监听器：ApplicationListener

ApplicationListener充当了事件监听器的角色，我们可以通过实现ApplicationListener接口来自定义监听器，ApplicationListener接口包含一个泛型参数，指定为我们需要监听的事件Class即可。

```
@Component
public class CustomSpringEventListener implements ApplicationListener<CustomSpringEvent> {
    private static final Logger logger = LoggerFactory.getLogger(CustomSpringEventListener.class);
    @Override
    public void onApplicationEvent(CustomSpringEvent event) {
        logger.info("Received spring custom event - {}", event.getMessage());
    }
}
```

当然也可以通过使用注解的方式@EventListener的方式来自定义事件监听器。

```

@Component
public class CustomSpringEventListener {
    private static final Logger logger = LoggerFactory.getLogger(CustomSpringEventListener.class);

        @EventListener
    public void onApplicationEvent(CustomSpringEvent event) {
        logger.info("Received spring custom event - {}" , event.getMessage());
    }
}
```

就这样我们就借助spring监听器实现了观察者模式，整体来说还是比较简洁的，相对于自己去实现观察者模式，能够节省一定的开发量。

4、异步事件

Spring监听器默认是同步的，在一些场景我们会希望能够通过异步的方式发布事件，使用Spring监听器实现异步发布事件也是一件非常简单的事情，只需要自定义ApplicationEventMulticaster实例指定异步的executor并注册到Spring容器中就可以了。

```
@Configuration
public class AsynchronousSpringEventsConfig {
    @Bean(name = "applicationEventMulticaster")
    public ApplicationEventMulticaster simpleApplicationEventMulticaster() {
        SimpleApplicationEventMulticaster eventMulticaster =
          new SimpleApplicationEventMulticaster();

        eventMulticaster.setTaskExecutor(new SimpleAsyncTaskExecutor());
        return eventMulticaster;
    }
}
```

当前异步使用方式是全局的，也就是所有监听器的监听处理都是异步的，如果需要针对个别监听器实现异步，那么可以在listener的onApplicationEvent方法实现异步，如使用Spring的@Async注解来实现。

三、Spring监听器源码解析

通过上面的简单例子，我们基本了解了Spring监听器的使用。接下来我们就会想Spring监听器内部是怎么实现的，监听器是怎么注册的、事件是怎么发布的、如何实现事件的监听等等。要想了解Spring监听器源码的实现，我们得要先看看Spring监听器的初始化过程是如何实现的，Spring监听器的初始化是发生在ApplicationContext初始化，包括两个核心步骤：事件广播器的初始化和监听器的注册。下面是ApplicationContext初始化过程ApplicationContext#refresh方法的部分源码：

```
public void refresh() throws BeansException, IllegalStateException {
        /*
         *
         */

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

        /*
         *
         */

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

        /*
         *
         */
    }
```

1、初始化事件广播器

```
        public static final String APPLICATION_EVENT_MULTICASTER_BEAN_NAME = "applicationEventMulticaster";

        protected void initApplicationEventMulticaster() {
                ConfigurableListableBeanFactory beanFactory = getBeanFactory();
                //beanFactory,bean
                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 {
            //SimpleApplicationEventMulticaster
                        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 + "]");
                        }
                }
        }
```

我们可以看到初始化事件广播器的逻辑很简单，首先判断是否有自定义的事件广播器（bean类型为ApplicationEventMulticaster且bean名称为"applicationEventMulticaster"），有的话则使用自定义的事件广播器，否则创建默认的事件广播器SimpleApplicationEventMulticaster对象，前面我们介绍的Spring监听器使用示例里的异步实现方式，就是通过自定义事件广播器来实现异步发送事件。

2、注册监听器

注册监听器的代码逻辑也是比较简单，这个方法就做了三件事，注册通过硬编码方式加进来的listener、注册实现了ApplicationListener接口的bean、发布提前发布的事件（这个是Spring针对特殊场景做的处理，读者可以不用过多关注，这个不会影响我们对Spring监听器原理的理解）。

我们可以继续看下Spring监听器的事件广播器的抽象类AbstractApplicationEventMulticaster中addApplicationListener和addApplicationListenerBean方法的实现。

```
        public void addApplicationListener(ApplicationListener<?> listener) {
                synchronized (this.retrievalMutex) {
                        Object singletonTarget = AopProxyUtils.getSingletonTarget(listener);
                        if (singletonTarget instanceof ApplicationListener) {
                                this.defaultRetriever.applicationListeners.remove(singletonTarget);
                        }
                        this.defaultRetriever.applicationListeners.add(listener);
                        this.retrieverCache.clear();
                }
        }

        public void addApplicationListenerBean(String listenerBeanName) {
                synchronized (this.retrievalMutex) {
                        this.defaultRetriever.applicationListenerBeans.add(listenerBeanName);
                        this.retrieverCache.clear();
                }
        }
```

从上面的源码中可以看到分别添加到defaultRetriever的applicationListeners和applicationListenerBeans中，这两个集合在后面发布事件源码分析中会用到。

3、发布事件

```
        public void publishEvent(ApplicationEvent event) {
                publishEvent(event, null);
        }

        protected void publishEvent(Object event, ResolvableType eventType) {
                /*
         *
         */
                ApplicationEvent applicationEvent;
                if (event instanceof ApplicationEvent) {
                        applicationEvent = (ApplicationEvent) event;
                }
                else {
                        //ApplicationEventPayloadApplicationEvent
                        applicationEvent = new PayloadApplicationEvent<Object>(this, event);
                        if (eventType == null) {
                                eventType = ((PayloadApplicationEvent) applicationEvent).getResolvableType();
                        }
                }
                // earlyApplicationEventsnullelse
                if (this.earlyApplicationEvents != null) {
                        //publishEventearlyApplicationEvents
                        this.earlyApplicationEvents.add(applicationEvent);
                }
                else {
                        //
                        getApplicationEventMulticaster().multicastEvent(applicationEvent, eventType);
                }
                //ApplicationContextparentparentpublishEvent
                if (this.parent != null) {
                        if (this.parent instanceof AbstractApplicationContext) {
                                ((AbstractApplicationContext) this.parent).publishEvent(event, eventType);
                        }
                        else {
                                this.parent.publishEvent(event);
                        }
                }
        }
```

从上面的源码我们可以看到ApplicationContext#publishEvent方法我们就只做了两件事情，封装ApplicationEvent对象和获取事件广播器发布事件，也就是说事件的发布核心逻辑主要是在事件广播器中实现，接下来我们可以看下Spring默认的事件广播器SimpleApplicationEventMulticaster#multicastEvent的具体实现。

```
public void multicastEvent(final ApplicationEvent event, ResolvableType eventType) {
                //eventType
                ResolvableType type = (eventType != null ? eventType : resolveDefaultEventType(event));
                //listener
                for (final ApplicationListener<?> listener : getApplicationListeners(event, type)) {
                        //executorSpringSpringexecutor
                        Executor executor = getTaskExecutor();
                        if (executor != null) {
                                //listener
                                executor.execute(new Runnable() {
                                        @Override
                                        public void run() {
                                                invokeListener(listener, event);
                                        }
                                });
                        }
                        else {
                                //listener
                                invokeListener(listener, event);
                        }
                }
        }
```

从上面的源码我们可以看出来，multicastEvent方法主要包含两个核心逻辑，第一就是根据事件和事件类型获取所有监听器getApplicationListeners(event, type)；第二就是同步或者异步唤醒所有监听器invokeListener(listener, event)。接下来我们分别深入的去了解这两块逻辑。

3.1 根据事件和事件类型获取所有监听器

```
        protected Collection<ApplicationListener<?>> getApplicationListeners(
                        ApplicationEvent event, ResolvableType eventType) {
                Object source = event.getSource();
                Class<?> sourceType = (source != null ? source.getClass() : null);
                ListenerCacheKey cacheKey = new ListenerCacheKey(eventType, sourceType);

                // listener
                ListenerRetriever retriever = this.retrieverCache.get(cacheKey);
                if (retriever != null) {
                        return retriever.getApplicationListeners();
                }

                if (this.beanClassLoader == null ||
                                (ClassUtils.isCacheSafe(event.getClass(), this.beanClassLoader) &&
                                                (sourceType == null || ClassUtils.isCacheSafe(sourceType, this.
beanClassLoader)))) {
                        // Fully synchronized building and caching of a ListenerRetriever
                        synchronized (this.retrievalMutex) {
                                retriever = this.retrieverCache.get(cacheKey);
                                if (retriever != null) {
                                        return retriever.getApplicationListeners();
                                }
                                retriever = new ListenerRetriever(true);
                                //
                                Collection<ApplicationListener<?>> listeners = retrieveApplicationListeners
(eventType, sourceType, retriever);
                                this.retrieverCache.put(cacheKey, retriever);
                                return listeners;
                        }
                }
                else {
                        // No ListenerRetriever caching -> no synchronization necessary
                        return retrieveApplicationListeners(eventType, sourceType, null);
                }
        }
```

通过上面的源码可以看出来，这里主要是对获取到的监听器做缓存的操作，其真正实现获取监听器逻辑的方法是retrieveApplicationListeners(eventType, sourceType, retriever)，所以我们再来看下这个方法的具体实现。

```
        private Collection<ApplicationListener<?>> retrieveApplicationListeners(
                        ResolvableType eventType, Class<?> sourceType, ListenerRetriever retriever) {

                List<ApplicationListener<?>> allListeners = new ArrayList<ApplicationListener<?>>();
                Set<ApplicationListener<?>> listeners;
                Set<String> listenerBeans;
                synchronized (this.retrievalMutex) {
                        //Setthis.defaultRetriever.applicationListeners
                        listeners = new LinkedHashSet<ApplicationListener<?>>(this.defaultRetriever.
applicationListeners);
                        //this.defaultRetriever.applicationListenerBeansApplicationListenerbean
                        listenerBeans = new LinkedHashSet<String>(this.defaultRetriever.
applicationListenerBeans);
                }
                //listener
                for (ApplicationListener<?> listener : listeners) {
                        //listener
                        if (supportsEvent(listener, eventType, sourceType)) {
                                if (retriever != null) {
                                        retriever.applicationListeners.add(listener);
                                }
                                allListeners.add(listener);
                        }
                }
                if (!listenerBeans.isEmpty()) {
                        BeanFactory beanFactory = getBeanFactory();
                        //listenerBeans
                        for (String listenerBeanName : listenerBeans) {
                                try {
                                        //listenerclass
                                        Class<?> listenerType = beanFactory.getType(listenerBeanName);
                                        //listener
                                        if (listenerType == null || supportsEvent(listenerType, eventType)) {
                                                //beanallListeners
                                                ApplicationListener<?> listener =
                                                                beanFactory.getBean(listenerBeanName,
ApplicationListener.class);
                                                if (!allListeners.contains(listener) && supportsEvent(listener,
eventType, sourceType)) {
                                                        if (retriever != null) {
                                                                retriever.applicationListenerBeans.add
(listenerBeanName);
                                                        }
                                                        allListeners.add(listener);
                                                }
                                        }
                                }
                                catch (NoSuchBeanDefinitionException ex) {
                                        // Singleton listener instance (without backing bean definition)
disappeared -
                                        // probably in the middle of the destruction phase
                                }
                        }
                }
                //
                AnnotationAwareOrderComparator.sort(allListeners);
                return allListeners;
        }
```

这里相对于前面的代码要稍微复杂些，这里主要是做了三件事，第一获取初始化的所有监听器；第二判断监听器是否监听当前事件；第三对监听器进行排序。

3.2唤醒监听器

```
        protected void invokeListener(ApplicationListener<?> listener, ApplicationEvent event) {
                //
                ErrorHandler errorHandler = getErrorHandler();
                if (errorHandler != null) {
                        try {
                                //listener
                                doInvokeListener(listener, event);
                        }
                        catch (Throwable err) {
                                //
                                errorHandler.handleError(err);
                        }
                }
                else {
                        doInvokeListener(listener, event);
                }
        }

        @SuppressWarnings({"unchecked", "rawtypes"})
        private void doInvokeListener(ApplicationListener listener, ApplicationEvent event) {
                try {
                        //onApplicationEvent
                        listener.onApplicationEvent(event);
                }
                catch (ClassCastException ex) {
                        String msg = ex.getMessage();
                        if (msg == null || matchesClassCastMessage(msg, event.getClass())) {
                                // Possibly a lambda-defined listener which we could not resolve the generic
event type for
                                // -> let's suppress the exception and just log a debug message.
                                Log logger = LogFactory.getLog(getClass());
                                if (logger.isDebugEnabled()) {
                                        logger.debug("Non-matching event type for listener: " + listener, ex);
                                }
                        }
                        else {
                                throw ex;
                        }
                }
        }
```

可以看到唤醒listener的逻辑是比较简单，就是遍历listener调用onApplicationEvent方法。

四、线上性能问题排查

通过上面对Spring监听器使用介绍和源码分析，相信读者此时对Spring监听器已经有了比较清晰的认识，这时就可以向读者分享下因为不正当使用Spring监听器模式从而引起性能问题的案例。也希望能通过本次案例为广大读者提供宝贵的经验，避免出现类似的问题。

1、问题背景

为了让读者能够更好的理解线上性能问题产生的原因，在这里先说明下引起这次问题的版本关键改动点。

1、为了实现业务代码解耦，引入Spring监听器；

2、使用Spring监听器异步发布事件，自定义事件广播器，并指定executor为Spring自带的SimpleAsyncTaskExecutor，如下代码所示，这个与上面讲的异步示例是一致的。

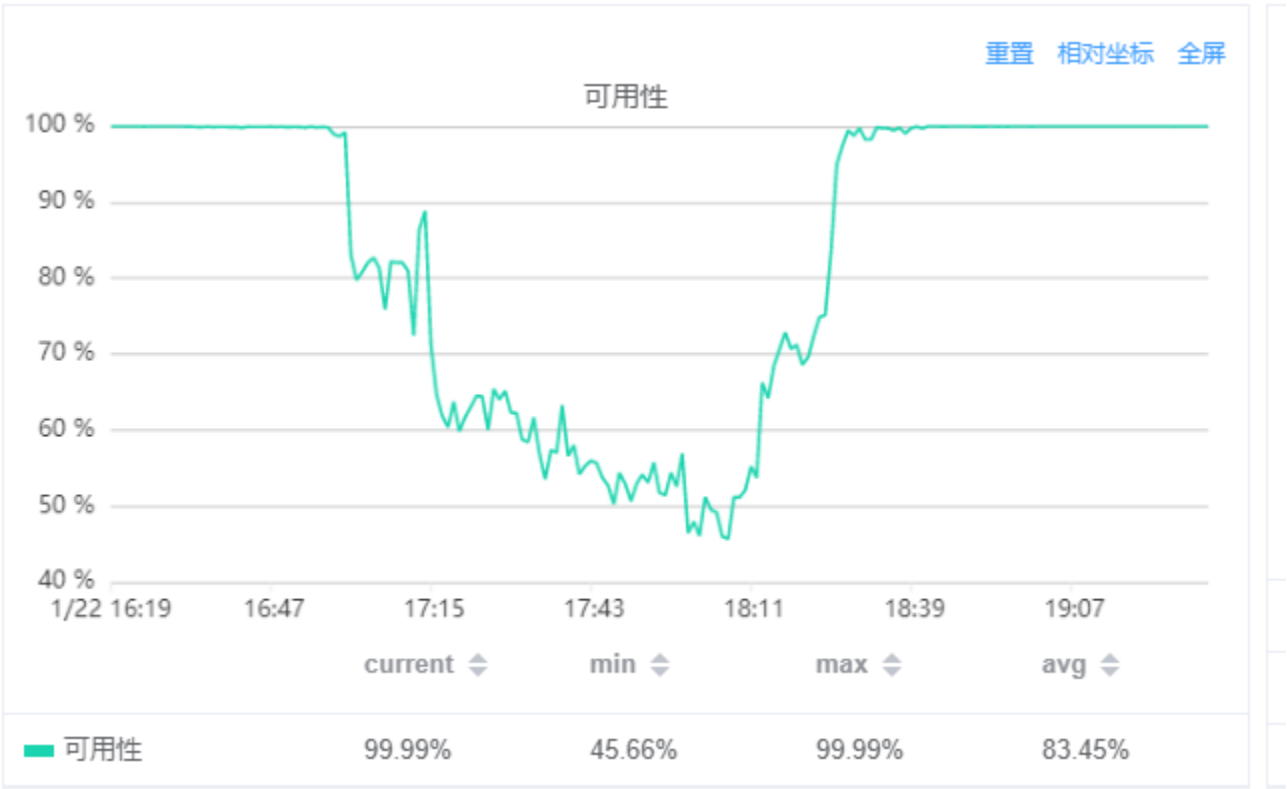
```
@Configuration
public class AsynchronousSpringEventsConfig {
    @Bean(name = "applicationEventMulticaster")
    public ApplicationEventMulticaster simpleApplicationEventMulticaster() {
        SimpleApplicationEventMulticaster eventMulticaster =
          new SimpleApplicationEventMulticaster();

        eventMulticaster.setTaskExecutor(new SimpleAsyncTaskExecutor());
        return eventMulticaster;
    }
}
```

本文不会介绍具体的业务场景，为什么需要引用Spring监听器，因为这个并不会影响读者对本次问题的理解，读者只需要记住上面列举的两点便可。

2、问题表现

当前版本是服务端线上线，客户端后放量，也就是在服务端上线时，客户端并没有上线对应版本的功能，也就是新增的业务功能并不会有流量进入，这也对我们后面排查问题的方向产生一定的影响，因为我们没有必要去怀疑没有流量的代码性能问题。服务端线上共有80台机器，服务端上线时是先灰度30%的机器跑了一天，这时线上并没有发现任何问题，于是便开始逐渐放量，全量完后，当流量高峰期时，整个域名可用性下降明显，触发了域名可用性告警，可用性最低下降至45%，而且并没有自动恢复的现象，当时的可用性折线图如下。



而且问题工程的所有接口均出现了可用性下降问题，手动调用接口基本都是超时无响应，观察机器指标可以发现线程数出现了明显上涨，无论是Runnable还是block。解决这个问题我们首先想到就是重启机器，但重启大法并没有解决问题，可用性并没有回升。我们是将全部机器回滚到上个版本后，可用性才恢复。

3、问题排查

3.1、线程栈分析

最终我们回退了版本并保留一台问题机器进行对比观察。从公司的调用链中能明显看出问题机器相对其他机器线程数有翻倍的增长。于是便对问题机器dump了线程栈。dump下来后我们发现其中有大量如下所示的线程栈信息：

java.lang.Thread.State: RUNNABLE
at java.lang.Thread.setPriority0(Native Method)
at java.lang.Thread.setPriority(Thread.java:1095)
at org.springframework.util.CustomizableThreadCreator.createThread(CustomizableThreadCreator.java:150)
at org.springframework.core.task.SimpleAsyncTaskExecutor.doExecute(SimpleAsyncTaskExecutor.java:232)
at org.springframework.core.task.SimpleAsyncTaskExecutor.execute(SimpleAsyncTaskExecutor.java:191)
at org.springframework.core.task.SimpleAsyncTaskExecutor.execute(SimpleAsyncTaskExecutor.java:170)
at org.springframework.context.event.SimpleApplicationEventMulticaster.multicastEvent(SimpleApplicationEventMulticaster.java:131)
at org.springframework.context.support.AbstractApplicationContext.publishEvent(AbstractApplicationContext.java:393)
at org.springframework.context.support.AbstractApplicationContext.publishEvent(AbstractApplicationContext.java:347)
at org.springframework.web.servlet.FrameworkServlet.publishRequestHandledEvent(FrameworkServlet.java:1073)
at org.springframework.web.servlet.FrameworkServlet.processRequest(FrameworkServlet.java:1005)
at org.springframework.web.servlet.FrameworkServlet.doGet(FrameworkServlet.java:861)
at javax.servlet.http.HttpServlet.service(HttpServlet.java:624)
at org.springframework.web.servlet.FrameworkServlet.service(FrameworkServlet.java:846)
at javax.servlet.http.HttpServlet.service(HttpServlet.java:731)
at org.apache.catalina.core.ApplicationFilterChain.internalDoFilter(ApplicationFilterChain.java:303)
at org.apache.catalina.core.ApplicationFilterChain.doFilter(ApplicationFilterChain.java:208)
at org.apache.tomcat.websocket.server.WsFilter.doFilter(WsFilter.java:52)
at org.apache.catalina.core.ApplicationFilterChain.internalDoFilter(ApplicationFilterChain.java:241)
at org.apache.catalina.core.ApplicationFilterChain.doFilter(ApplicationFilterChain.java:208)

此时可以发现标红的代码就是在发布事件，这个非常契合本篇文章的主题，所以问题很大可能就是出在这里，最终得出来的结论也确实是这里导致工程可用性下降。但当时的我们还不是事后诸葛亮，还没有意识到是Spring监听器的问题，因为从整个线程栈来看，这并没有涉及到任何的业务代码，这只是Spring Mvc本身发送的事件，并不是业务代码中发布的事件。这与我们一开始的想法是不一致，在当时我们都认为肯定是这个问题版本加入了“问题代码”，从而导致整个工程处理能力下降，所以当看到这个线程栈中没有我们想要的“问题代码”时，我们没有对此产生过多的怀疑，也没有深入的去分析。但读者可以结合上面介绍的问题背景，思考下是什么原因导致的性能问题，为了便于读者更好的理解，贴出当前线程栈对应的部分源码 FrameworkServlet#processRequest：

```
protected final void processRequest(HttpServletRequest request, HttpServletResponse response)
                        throws ServletException, IOException {
                /*
         *
         */
                try {
                        doService(request, response);
                }
                /*
         * catch
         */
                finally {
                        /*
                 *
                 */

                        //
                        publishRequestHandledEvent(request, response, startTime, failureCause);
                }
        }

private void publishRequestHandledEvent(
                        HttpServletRequest request, HttpServletResponse response, long startTime, Throwable
failureCause) {

                if (this.publishEvents) {
                        // Whether or not we succeeded, publish an event.
                        long processingTime = System.currentTimeMillis() - startTime;
                        int statusCode = (responseGetStatusAvailable ? response.getStatus() : -1);
                        //
                        this.webApplicationContext.publishEvent(
                                        new ServletRequestHandledEvent(this,
                                                        request.getRequestURI(), request.getRemoteAddr(),
                                                        request.getMethod(), getServletConfig().
getServletName(),
                                                        WebUtils.getSessionId(request), getUsernameForRequest
(request),
                                                        processingTime, failureCause, statusCode));
                }
        }
```

从上面的源码可以看出来，Spring Mvc在处理完一个请求后（无论成功与否）都会发送一个事件。

3.2、压测

因为通过线上问题机器的线程栈没有立马发现具体的问题，于是就在压测环境对问题版本进行了压测。在压测过程中也确实发现当前版本存在性能问题，最终压测发现当前版本单机最大的qps是500（正常水平在1500~2000之间），当qps超过500，服务基本不可用，dump压测机器的线程栈，发现有198个线程处在Spring监听器的发布事件中，这次和线上环境不同的是这个事件是业务代码发布的事件，而不再是Spring Mvc发布的事件。其线程栈信息如下所示，为了避免涉及信息安全问题，对于业务代码做了一些马赛克处理（用“xxx”替换掉跟业务相关的单词）。

java.lang.Thread.State: RUNNABLE
at java.lang.Thread.setPriority0(Native Method)
at java.lang.Thread.setPriority(Thread.java:1095)
at java.lang.Thread.init(Thread.java:417)
at java.lang.Thread.init(Thread.java:349)
at java.lang.Thread.<init>(Thread.java:599)
at org.springframework.util.CustomizableThreadCreator.createThread(CustomizableThreadCreator.java:149)
at org.springframework.core.task.SimpleAsyncTaskExecutor.doExecute(SimpleAsyncTaskExecutor.java:232)
at org.springframework.core.task.SimpleAsyncTaskExecutor.execute(SimpleAsyncTaskExecutor.java:191)
at org.springframework.core.task.SimpleAsyncTaskExecutor.execute(SimpleAsyncTaskExecutor.java:170)
at org.springframework.context.event.SimpleApplicationEventMulticaster.multicastEvent(SimpleApplicationEventMulticaster.java:131)
at org.springframework.context.support.AbstractApplicationContext.publishEvent(AbstractApplicationContext.java:393)
at org.springframework.context.support.AbstractApplicationContext.publishEvent(AbstractApplicationContext.java:347)
at com.xxx.xxx.xxx.xxx.service.impl.XxxServiceImpl.afterXxxSuccess(XxxServiceImpl.java:150)
at com.xxx.xxx.xxx.xxx.service.impl.XxxServiceImpl.doXxx(XxxServiceImpl.java:115)

这时候可以更明显的发现Spring的事件发布是存在性能问题的，再进行问题分析前，先说明下读者会产生的一个疑惑。

```
1Spring Mvc
```

虽然线上和压测环境的线程栈不完全一致，但至少可以反映出一个问题，那就是发布事件存在性能问题。而我们确实也是在新版本中在业务代码中使用了Spring监听器。于是我们便开始思考这其中是否有什么关联，有没有可能是我们做了什么改动导致Spring监听器出现了性能问题。

为了验证我们的思考，于是便开始对Spring监听器原理进行深入的了解。终于发现了问题点，我们为了实现异步发送事件，使用了自定义的事件广播器，在自定义的事件广播器中我们使用了SimpleAsyncTaskExecutor来实现异步，而这个改动是全局的，也就是所有使用了Spring监听器的地方都会由原本默认的同步变成异步，而Spring提供的SimpleAsyncTaskExecutor并不是通过线程池实现的，而是每次唤醒一个listener则会创建一个线程，这就是为什么线上线程数会增加的原因。这也就解释了为什么线上没有走到新版本的代码逻辑，仍然会存在性能问题，那是因为我们改变了Spring监听器的实现方式（同步改成异步），从而导致Spring Mvc自身的事件发布逻辑创建过多的线程，可用性下降，再回过头来看线上的线程栈信息，我们也会发现大量的线程都处在创建新线程过程中。所以当我们把异步去掉，调整成同步模式，整个服务的可用性就恢复到了正常水平。

五、总结

每一次线上问题都会让我们精神紧张、心情压抑，但同时也是宝贵的成长经历，为了避免重蹈覆辙，我们也总结了如下经验：
1、不要滥用异步，异步能给我们带来好处，同时也会有代码复杂、问题排查困难等问题，所以在使用前需要进行合理的评估，如本案例中其实完全没有必要使用异步，因为listener并没有做什么耗时的io操作；
2、使用异步时优先使用线程池，线程池可以避免创建过多线程和减少创建销毁线程的损耗。如本案例中就是因为没有使用线程池从而导致创建线程过多，当然这不代表使用线程池就不会有问题，使用线程池也需要根据具体场景评估具体的参数指标；
3、对于业务中使用的技术原理要有充分的了解，这样能让我们更合理的使用和进行问题排查。如本案例中如果在使用Spring监听器时，了解到自定义事件广播器的影响是全局的，那么就不会采用当前方式去实现异步，比如可以针对耗时较长的listener使用异步处理；
4、重视性能压测，很多性能问题是很难通过肉眼识别，所以在版本上线前可以进行一次性能压测，尤其是改动较大的版本。