

识别Tomcat堆栈中常见线程

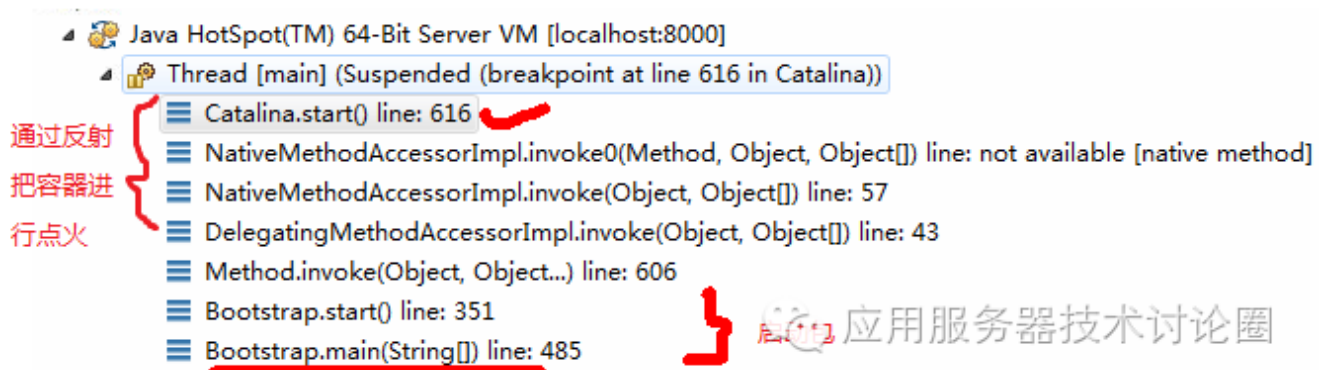
2016-06-27 09:08 feiying 0 阅读 183

Tomcat作为一个服务器来讲，必然运行着很多的线程，而每一个线程究竟是干什么的，这个需要非常的清楚，无论是打印断点，还是通过jstack进行分析锁，或者是从堆栈进行查看当前服务器的状态，这都是必须要掌握的技能。

本文带你基于Tomcat较新的版本，识别Tomcat堆栈中的线程。

1.main线程

main线程是Tomcat主要的线程，其主要的作用是通过启动包来对容器进行点火：

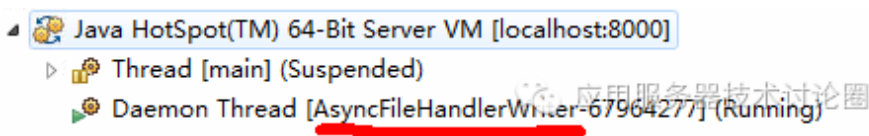


而对于该线程把容器点火完成之后，会主线程通过await进行阻塞，当stop服务器的socket发过来的时候，该main线程唤醒，最后执行清理工作，然后服务器关闭。

你如果把Tomcat看成一个java程序，main线程实际上就是main线程，也就是启动的主线程，但该线程并不参与功能，例如请求接收等，它的任务就是启动和停止Tomcat，所以无论在运行时，启动时，都会有，生命周期是一直都在的；

2.AsyncFileHandlerWriter线程

日志输入线程



下载《开发者大全》

下载 (/download/dev.apk)



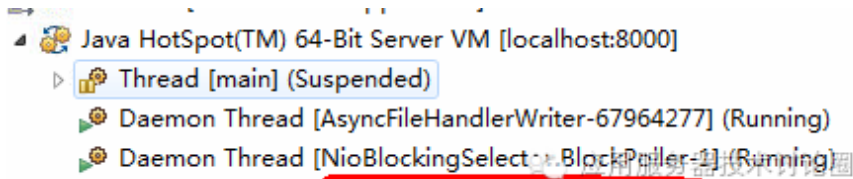
顾名思义，该线程是用于异步文件处理的，它的作用是在Tomcat级别构架出一个输出框架，然后不同的日志系统都可以对接这个框架，因为日志对于服务器来说，是非常重要的功能，如下，就是juli的配置：

```
#####  
# Facility specific properties.  
# Provides extra control for each logger.  
#####  
  
org.apache.catalina.core.ContainerBase.[Catalina].[localhost].level = INFO  
org.apache.catalina.core.ContainerBase.[Catalina].[localhost].handlers  
2localhost.org.apache.juli.AsyncFileHandler
```

该线程主要的作用是通过一个event queue来与log系统对接，该线程启动的时候就有了，全生命周期。

3.NioBlockingSelector.BlockPoller线程

Nio方式的Servlet阻塞输入输出检测线程

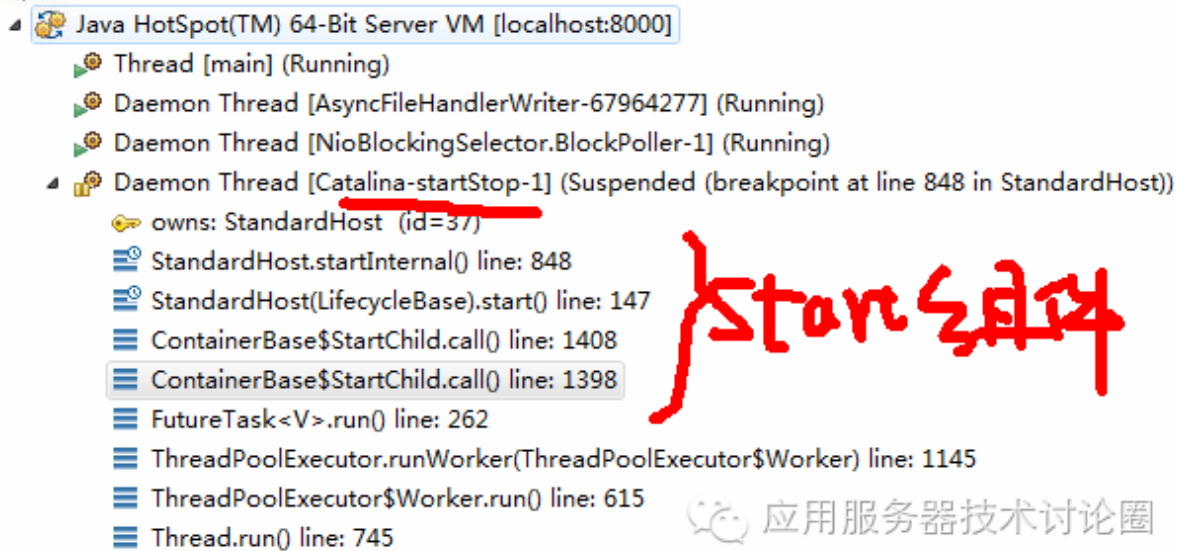


该线程在前面的NioBlockingPool中讲得很清楚了，其NIO通道的Servlet输入和输出最终都是通过NioBlockingPool来完成的，而NioBlockingPool又根据Tomcat的场景可以分成阻塞或者是非阻塞的，对于阻塞来讲，为了等待网络发出，需要启动一个线程实时监测网络socketChannel是否可以发出包，而如果不这么做的话，就需要使用一个while空转，这样会让工作线程一直损耗。

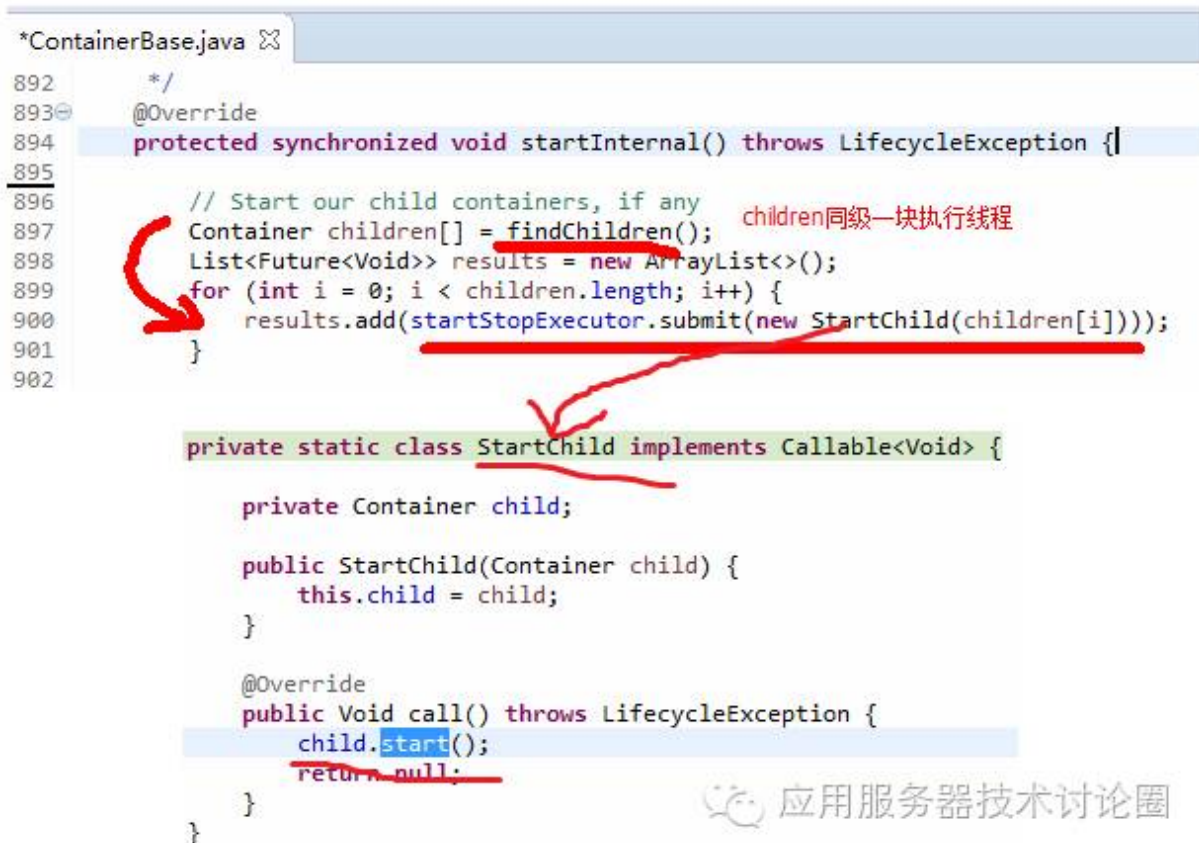
只要是阻塞模式，并且在Tomcat启动的时候，添加了一D参数 org.apache.tomcat.util.net.NioSelectorShared的话，那么就会启动这个线程。

4.组件-startstop线程（启动时）

Tomcat容器被点火起来后，并不是傻傻的按照次序一步一步的启动，而是将整体的children构架好，按照层级进行启动，对于每一层级的组件都是采用startstop线程进行启动，我们观察一下堆栈就可以发现：



这个startstop线程实际代码调用就是采用的JDK自带线程池来做的，请的位置就是ContainerBase的组件父类的startInternal：

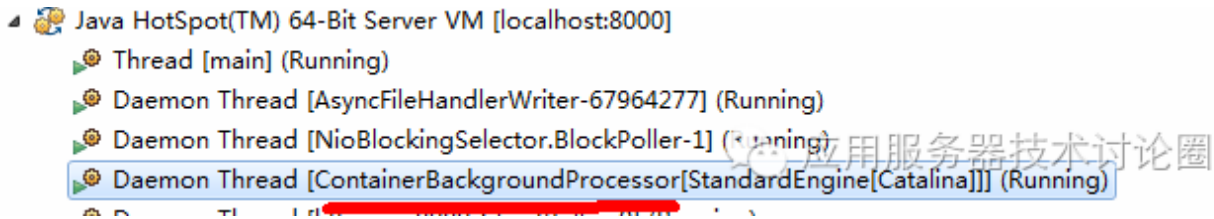


因为每一个Tomcat后端的容器组件都是继承与这个ContainerBase，所以相当于每一个组件启动的时候，除了对自身的状态进行设置，都会启动startChild线程启动自己的孩子组件。

而这个线程仅仅就是在启动时，当组件启动完成后，那么该线程就退出了，生命周期仅仅限于此。

5.ContainerbackgroundProcessor（运行时）

Tomcat在启动之后，不能说是死水一潭，很多时候可能会对Tomcat后端的容器组件做一些变化，例如部署一个应用，相当于你就需要在对应的Standardhost加上一个StandardContext，也有可能是在热部署开关开启的时候，对资源进行增删，这样应用可能会重新reload，也有可能是在生产模式下，对class进行重新替换等等，这个时候就需要在Tomcat级别中有一个线程能实时扫描Tomcat容器的变化，这个就是ContainerBackgroundProcessor线程了：



我们可以看到这个代码，也是在ContainerBase中：



这个线程是一个递归调用，也就是说，每一个容器组件其实都有一个backgroundProcessor，而整个Tomcat就点起一个线程开启扫描，扫完儿子，再扫孙子（实际上来说，主要还是用于StandardContext这一级，可以看到StandardContext这一级：


```

ContainerBase.java | StandardEngine.java | StandardContext.java
@Override
public void backgroundProcess() {
    if (!getState().isAvailable())
        return;

    Loader loader = getLoader();
    if (loader != null) {
        try {
            loader.backgroundProcess();
        } catch (Exception e) {
            Log.warn(sm.getString(
                "standardContext.backgroundProcess.loader", loader), e);
        }
    }
    Manager manager = getManager();
    if (manager != null) {
        try {
            manager.backgroundProcess();
        } catch (Exception e) {
            Log.warn(sm.getString(
                "standardContext.backgroundProcess.manager", manager),
                e);
        }
    }
    WebResourceRoot resources = getResources();
    if (resources != null) {
        try {
            resources.backgroundProcess();
        } catch (Exception e) {
            Log.warn(sm.getString(
                "standardContext.backgroundProcess.resources",
                resources), e);
        }
    }
    super.backgroundProcess();
}

```

类
JMX
资源

应用服务器技术讨论圈

我们可以看到，每一次backgroundProcessor，都会对该应用进行一次全方位的扫描，这个时候，当你开启了热部署的开关，一旦class和资源发生变化，立刻就会reload。

6.Poller线程（运行时）

NIO和APR模式下的Tomcat前端，都会有Poller线程：

```

Java HotSpot(TM) 64-Bit Server VM [localhost:8000]
  Thread [main] (Running)
  Daemon Thread [AsyncFileHandlerWriter-67964277] (Running)
  Daemon Thread [NioBlockingSelector.BlockPoller-1] (Running)
  Daemon Thread [ContainerBackgroundProcessor[StandardEngine[Catalina]]] (Running)
  Daemon Thread [http-nio-8080-ClientPoller-0] (Running)
  Daemon Thread [http-nio-8080-ClientPoller-1] (Running)

```

应用服务器技术讨论圈

对于Poller线程实际就是继续接着Acceptor进行处理，展开Selector，然后遍历key，将后续的任务转接给工作线程，起到的是一个缓冲，转接，和NIO事件遍历的作用

下载《开发者大全》

下载 (/download/dev.apk)



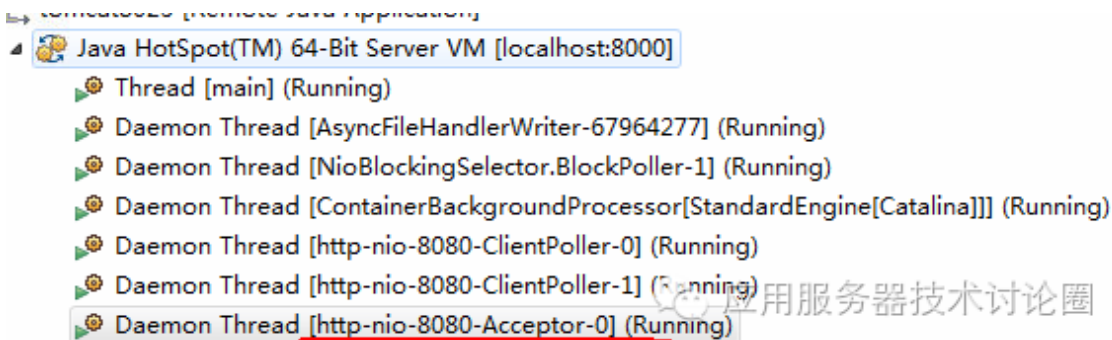
```
// Start poller threads
pollers = new Poller[getPollerThreadCount()];
for (int i=0; i<pollers.length; i++) {
    pollers[i] = new Poller();
    Thread pollerThread = new Thread(pollers[i], getName() + "-ClientPoller-"+i);
    pollerThread.setPriority(threadPriority);
    pollerThread.setDaemon(true);
    pollerThread.start();
}
```

应用服务器技术讨论圈

上述的代码在NioEndpoint的startInternal中，默认开始开启2个Poller线程，后期再随着压力增大增长，可以在Connector中进行配置。

7.Acceptor线程（运行时）

无论是NIO还是BIO通道，都会有Acceptor线程，该线程就是进行socket接收的，它不会继续处理，如果是NIO的，无论是新接收的还是包继续发的，直接就会交给Poller，而BIO模式，Acceptor线程直接把活就给工作线程了：



如果不配置，Acceptor线程默认开始就开启1个，后期再随着压力增大增长：

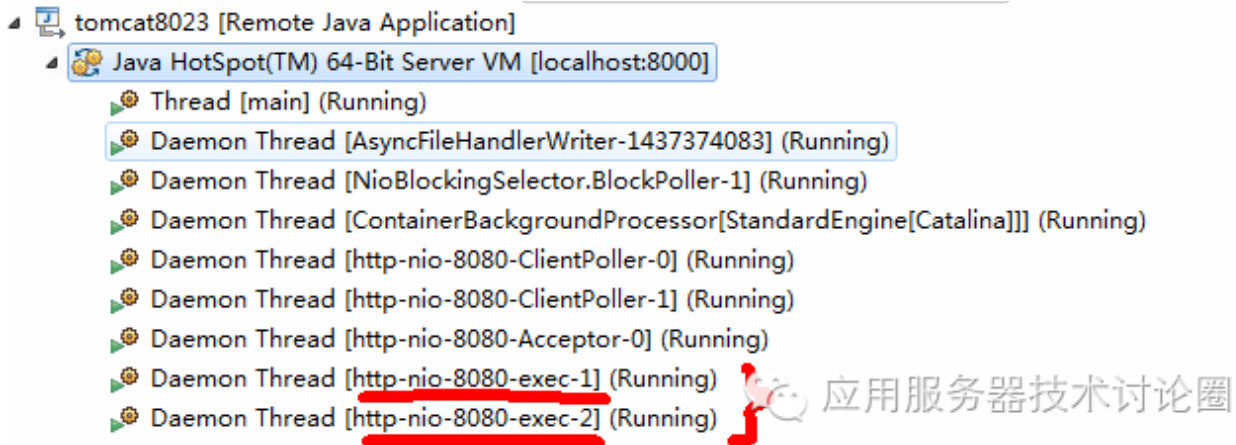
```
protected final void startAcceptorThreads() {
    int count = getAcceptorThreadCount();
    acceptors = new Acceptor[count];

    for (int i = 0; i < count; i++) {
        acceptors[i] = createAcceptor();
        String threadName = getName() + "-Acceptor-" + i;
        acceptors[i].setThreadName(threadName);
        Thread t = new Thread(acceptors[i], threadName);
        t.setPriority(getAcceptorThreadPriority());
        t.setDaemon(getDaemon());
        t.start();
    }
}
```

应用服务器技术讨论圈

8.工作线程（运行时）

也就是SocketProcessor，NIO模式下，Poller线程将解析好的socket交给SocketProcessor处理，它主要是http协议分析，攒出Response和Request，然后调用Tomcat后端的容器：



该线程的重要性不言而喻，Tomcat主要的时间都耗在这个线程上，所以我们可以看到Tomcat里面有很多的优化啊，配置啊，都是基于这个线程的，尽可能让这个线程少阻塞，少线程切换，甚至少创建，多利用。

下面就是NIO模式下创建的工作线程：

```
*NioEndpoint.java
@Override
public void startInternal() throws Exception
{
    // Create worker collection
    if ( getExecutor() == null ) {
        createExecutor();
    }

    public void createExecutor() {
        internalExecutor = true;
        TaskQueue taskqueue = new TaskQueue();
        TaskThreadFactory tf = new TaskThreadFactory(getName() + "-exec-", daemon, getThreadPriority());
        executor = new ThreadPoolExecutor(getMinSpareThreads(), getMaxSpareThreads(), 60, TimeUnit.SECONDS, taskqueue, tf);
        taskqueue.setParent( (ThreadPoolExecutor) executor);
    }
}
```

实际上也是JDK的线程池，只不过基于Tomcat的不同环境参数，对JDK线程池进行了定制化而已，本质上还是JDK的线程池。

这块在前面的文章已经详细分析过了，这里就不再赘余了。


8.其它线程

Tomcat本身还有很多其它的线程，远远不止这些，例如如果开启了sendfile，那么对sendfile就是开启线程来进行，例如这种功能的线程开启还有很多。

Tomcat作为一个自身的服务器，不可能就是1个线程搞定，肯定要多搞几个线程，而很多功能处理能做异步就异步，能尽可能的减少线程切换，也不是每一处都要开启线程，线程越多越好。

因此，线程的控制也尤为关键。

分享 :

阅读 183  0

应用服务器技术讨论圈 更多文章

东方通加码大数据业务 拟募资8亿收购微智信业 (/html/308/201504/206211355/1.html)

玩转Netty – 从Netty3升级到Netty4 (/html/308/201504/206233287/1.html)

金蝶中间件2015招聘来吧！Come on！ (/html/308/201505/206307460/1.html)

GlassFish 4.1 发布，J2EE 应用服务器 (/html/308/201505/206323120/1.html)

Tomcat对keep-alive的实现逻辑 (/html/308/201505/206357679/1.html)

猜您喜欢

用Swashbuckle给ASP.NET Core的项目自动生成Swagger的API帮助文档 (/html/391/201608/2654067819/1.html)

周末大讲堂 | 网络营销的小伙伴，千万别错过这堂课~ (/html/372/201605/2652953793/1.html)

数据挖掘浅析 (/html/323/201501/203151822/1.html)

有了这样的编程学习工具，再学不好，就是你不努力了！ (/html/370/201607/2651474714/1.html)

郑人元：美国四年本科后对中国教育的感悟 (/html/200/201312/100081387/1.html)