Tomcat源代码分析之三：请求处理

> Tomcat的请求处理是系统中最核心的部分，我们线上性能抖动的问题就是通过阅读这部分代码找到问题原因所在的。

在 conf/server.xml 文件中默认是使用BIO模式的，但这种阻塞式的方式效率不高，线上一般会使用NIO的方式。可以在 connector 中配置 protocal="org.apache.coyote.http11.Http11NioProtocol" 来指定，Tomcat启动后就会使用 NioEndpoint.Acceptor.run() 来接收外部请求的Socket连接，然后把它注册到一个队列中：

```
- 调用countUpOrWaitConnection()，如果连接超过最大连接数，就等待
- 调用serverSocket.accept()创建一个新的SocketChannel实例，这个地方会阻塞，有连接进来后会
- 调用setSocketOptions(SocketChannel socket)
  - 调用 getPoller0().register(channel) 将channel添加到Poller中
    - 调用Poller#regiester()方法
      - 调用addEvent(PollerEvent r)
        - 调用(ConcurrentLinkedQueue<Runnable>)events.offer(r) 把它加入队列中
```

有连接进来后会往下执行

Poller会在执行 NioEndpoint#startInternal() 中创建，启动单独的线程，检测队列中是否有PollerEvent事件，如果有就进行请求的处理：

```
- 运行run()
  - 调用events()
    - 启动PollerEvent线程，将(NioChannel)socket.getPoller().getSelector()注册到Socl
  - 调用selector.selectedKeys().iterator()得到SelectionKey集合
  - 遍历这个集合，执行processKey()方法
    - 调用processSocket()方法处理连接请求
      - 调用getExecutor().execute(SocketProcessor sc) 启动一个处理线程处理请求
```

注册到SocketChannel中

获取一个Executor之后，就开始真正的业务逻辑处理的，我们在业务代码中打印出来的调用栈就是这个线程的调用栈，上面的执行过程如果不看源代码是很难知道Tomcat是如何处理的，它的调用栈如下：

```
cn.fraudmetrix.forseti.api.service.RiskServiceImpl.execute(RiskServiceImpl.java)
sun.reflect.GeneratedMethodAccessor317.invoke(Unknown Source)
sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.java:
java.lang.reflect.Method.invoke(Method.java:606)
org.springframework.aop.support.AopUtils.invokeJoinpointUsingReflection(AopUtils.j
org.springframework.aop.framework.ReflectiveMethodInvocation.invokeJoinpoint(Refle
org.springframework.aop.framework.ReflectiveMethodInvocation.proceed(ReflectiveMet
org.springframework.transaction.interceptor.TransactionInterceptor$1.proceedWithIn
org.springframework.transaction.interceptor.TransactionAspectSupport.invokeWithinT
org.springframework.transaction.interceptor.TransactionInterceptor.invoke(Transact
org.springframework.aop.framework.ReflectiveMethodInvocation.proceed(ReflectiveMet
org.springframework.aop.framework.JdkDynamicAopProxy.invoke(JdkDynamicAopProxy.jav
```

```
com.sun.proxy.$Proxy238.execute(Unknown Source)
sun.reflect.GeneratedMethodAccessor317.invoke(Unknown Source)
sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.java:
java.lang.reflect.Method.invoke(Method.java:606)
org.springframework.aop.support.AopUtils.invokeJoinpointUsingReflection(AopUtils.ja
org.springframework.aop.framework.ReflectiveMethodInvocation.invokeJoinpoint(Reflec
org.springframework.aop.framework.ReflectiveMethodInvocation.proceed(ReflectiveMeth
org.springframework.transaction.interceptor.TransactionInterceptor$1.proceedWithIn
org.springframework.transaction.interceptor.TransactionAspectSupport.invokeWithinT
org.springframework.transaction.interceptor.TransactionInterceptor.invoke(Transact
org.springframework.aop.framework.ReflectiveMethodInvocation.proceed(ReflectiveMeth
org.springframework.aop.framework.JdkDynamicAopProxy.invoke(JdkDynamicAopProxy.jav
com.sun.proxy.$Proxy239.execute(Unknown Source)
cn.fraudmetrix.forseti.api.module.screen.RiskService.execute(RiskService.java:60)
cn.fraudmetrix.forseti.api.module.screen.RiskService$$FastClassByCGLIB$$7888f58c.in
net.sf.cglib.reflect.FastMethod.invoke(FastMethod.java:53)
com.alibaba.citrus.service.moduleloader.impl.adapter.MethodInvoker.invoke(MethodIn
com.alibaba.citrus.service.moduleloader.impl.adapter.DataBindingAdapter.executeAnd
com.alibaba.citrus.turbine.pipeline.valve.PerformScreenValve.performScreenModule(P
com.alibaba.citrus.turbine.pipeline.valve.PerformScreenValve.invoke(PerformScreenV
com.alibaba.citrus.service.pipeline.impl.PipelineImpl$PipelineContextImpl.invokeNe
com.alibaba.citrus.service.pipeline.impl.PipelineImpl$PipelineContextImpl.invoke(P
com.alibaba.citrus.service.pipeline.impl.valve.ChooseValve.invoke(ChooseValve.java
com.alibaba.citrus.service.pipeline.impl.PipelineImpl$PipelineContextImpl.invokeNe
com.alibaba.citrus.service.pipeline.impl.PipelineImpl$PipelineContextImpl.invoke(P
com.alibaba.citrus.service.pipeline.impl.valve.LoopValve.invokeBody(LoopValve.java
com.alibaba.citrus.service.pipeline.impl.valve.LoopValve.invoke(LoopValve.java:83)
com.alibaba.citrus.service.pipeline.impl.PipelineImpl$PipelineContextImpl.invokeNe
cn.fraudmetrix.forseti.api.valve.PrivilegeValidateValve.invoke(PrivilegeValidateVa
com.alibaba.citrus.service.pipeline.impl.PipelineImpl$PipelineContextImpl.invokeNe
com.alibaba.citrus.turbine.pipeline.valve.AnalyzeURLValve.invoke(AnalyzeURLValve.ja
com.alibaba.citrus.service.pipeline.impl.PipelineImpl$PipelineContextImpl.invokeNe
com.alibaba.citrus.turbine.pipeline.valve.SetLoggingContextValve.invoke(SetLogging
com.alibaba.citrus.service.pipeline.impl.PipelineImpl$PipelineContextImpl.invokeNe
com.alibaba.citrus.turbine.pipeline.valve.PrepareForTurbineValve.invoke(PrepareFor
com.alibaba.citrus.service.pipeline.impl.PipelineImpl$PipelineContextImpl.invokeNe
com.alibaba.citrus.service.pipeline.impl.PipelineImpl$PipelineContextImpl.invoke(P
com.alibaba.citrus.service.pipeline.impl.valve.TryCatchFinallyValve.invoke(TryCatc
com.alibaba.citrus.service.pipeline.impl.PipelineImpl$PipelineContextImpl.invokeNe
com.alibaba.citrus.service.pipeline.impl.PipelineImpl$PipelineContextImpl.invoke(P
com.alibaba.citrus.webx.impl.WebxControllerImpl.service(WebxControllerImpl.java:43
com.alibaba.citrus.webx.impl.WebxRootControllerImpl.handleRequest(WebxRootControll
com.alibaba.citrus.webx.support.AbstractWebxRootController.service(AbstractWebxRoo
com.alibaba.citrus.webx.servlet.WebxFrameworkFilter.doFilter(WebxFrameworkFilter.ja
com.alibaba.citrus.webx.servlet.FilterBean.doFilter(FilterBean.java:148)
org.apache.catalina.core.ApplicationFilterChain.internalDoFilter(ApplicationFilter
org.apache.catalina.core.ApplicationFilterChain.doFilter(ApplicationFilterChain.ja
com.alibaba.citrus.webx.servlet.SetLoggingContextFilter.doFilter(SetLoggingContext
com.alibaba.citrus.webx.servlet.FilterBean.doFilter(FilterBean.java:148)
org.apache.catalina.core.ApplicationFilterChain.internalDoFilter(ApplicationFilter
org.apache.catalina.core.ApplicationFilterChain.doFilter(ApplicationFilterChain.ja
org.apache.catalina.core.StandardWrapperValve.invoke(StandardWrapperValve.java:222
org.apache.catalina.core.StandardContextValve.invoke(StandardContextValve.java:123
org.apache.catalina.authenticator.AuthenticatorBase.invoke(AuthenticatorBase.java:
org.apache.catalina.core.StandardHostValve.invoke(StandardHostValve.java:171)
org.apache.catalina.valves.ErrorReportValve.invoke(ErrorReportValve.java:100)
```

```
org.apache.catalina.valves.AccessLogValve.invoke(AccessLogValve.java:953)
org.apache.catalina.core.StandardEngineValve.invoke(StandardEngineValve.java:118)
org.apache.catalina.connector.CoyoteAdapter.service(CoyoteAdapter.java:409)
org.apache.coyote.http11.AbstractHttp11Processor.process(AbstractHttp11Processor.ja
org.apache.coyote.AbstractProtocol$AbstractConnectionHandler.process(AbstractProtoc
org.apache.tomcat.util.net.NioEndpoint$SocketProcessor.doRun(NioEndpoint.java:1721)
org.apache.tomcat.util.net.NioEndpoint$SocketProcessor.run(NioEndpoint.java:1679)
java.util.concurrent.ThreadPoolExecutor.runWorker(ThreadPoolExecutor.java:1145)
java.util.concurrent.ThreadPoolExecutor$Worker.run(ThreadPoolExecutor.java:615)
java.lang.Thread.run(Thread.java:744)
```

在 `getExecutor().execute(SocketProcessor sc)` 是有个坑的， `conf/server.xml` 中配置Executor的参数时通常有如下几个：

```
<Executor
    name="tomcatThreadPool"
    namePrefix="catalina-exec-"
    maxThreads="150"
    minSpareThreads="4"
    maxIdleTime="60000"
    maxQueueSize="100"/>
```

最关键的后面四个参数， `maxThreads` 是线程池的最大活动线程数，可以把它理解为线程池最多能同时开启的线程数。 `minSpareThreads` 是保留的最小活动空闲线程数，也是说不管有没有请求，活动线程数最少是4。 `maxIdleTime` 是指超过4的线程最大空闲时间，超过这个时间线程还没有使用将会被关闭。 `maxQueueSize` 是指当已经有4个线程在运行中了，如果再有请求进来会将它放到队列中，最多能放100，默认是 `Integer.MAX_VALUE` ，基本上等同于无界队列了。最开始没有完全搞清楚 `minSpareThreads` 的含义，从官方文档上看以为是保留的最小活动线程数，如果有请求超过这个数目，还会创建新的线程来执行，只要不超过最大值150就行，如果超过最大值就会放入队列，超过队列的长度就会被拒绝。但看源代码后才发现不是这么回事：

```
public abstract class AbstractEndpoint {
    ...
    public void createExecutor() {
        internalExecutor = true;
        TaskQueue taskqueue = new TaskQueue();
        TaskThreadFactory tf = new TaskThreadFactory(getName() + "-exec-", daemon
        executor = new ThreadPoolExecutor(getMinSpareThreads(), getMaxThreads(),
        taskqueue.setParent( (ThreadPoolExecutor) executor);
    }
    ...
}

public class ThreadPoolExecutor extends java.util.concurrent.ThreadPoolExecutor {
    ...
    public ThreadPoolExecutor(int corePoolSize, int maximumPoolSize, long keepAliv
        super(corePoolSize, maximumPoolSize, keepAliveTime, unit, workQueue, thre
    }
    ...
```

```
}
```

问题根源：

从上面的代码来看，Exector最终还是用了Jdk的 `ThreadPoolExecutor` ， `minSpareThreads` 实际上指的就是 `corePoolSize` ，对它了解的人都知道，将请求数小于等于它时会马上创建一个新的线程，如果大于它就会把新的请求放到队列中，如果请求数超过 `corePoolSize` 加上队列的长度但又小于 `maxPoolSize` 时也会马上创建新的线程。而Tomcat中默认是使用了一个无界队列 `TaskQueue` ，它继承自 `LinkedBlockingQueue<Runnable>` ,在上面设置 `maxThreads` 和 `maxQueueSize` 其实是没有意义的，因为默认创建的是无界队列根本没有读取配置的参数几乎永远不会满，自然不存在超过需要用 `maxThreads` 来判断了。 **上面最后一句理解有误**：maxThreads实际上executor.setMaximumPoolSize(maxThreads)来实现，肯定有效果。（见AbstractEndpoint<S>.setMaxThreads(int maxThreads)方法实现）

解决方法： maxQueueSize参数见StandardThreadExecutor的setMaxQueueSize(int size)和startInternal()方法实现

像我们的业务场景对执行时间要求非常高，如果线程无法马上创建而被放入队列等待前面的请求释放资源就意味着可能会超时，再执行已毫无意义，因此需要将 `minSpareThreads` 值设置的大一点，一有请求就马上创建线程，不要等待下去。

同时，还要将 maxQueueSize 值设置为1
maxQueueSize="1"

通过阅读源代码可以更好的理解系统的执行原理，便于参数的合理设置和快速定位问题。

yikebocai / 2014-11-27