

27 | 新特性：Tomcat如何支持异步Servlet？

2019-07-11 李号双

深入拆解Tomcat & Jetty

[进入课程 >](#)



讲述：李号双

时长 08:27 大小 7.75M



通过专栏前面的学习我们知道，当一个新的请求到达时，Tomcat 和 Jetty 会从线程池里拿出一个线程来处理请求，这个线程会调用你的 Web 应用，Web 应用在处理请求的过程中，Tomcat 线程会一直阻塞，直到 Web 应用处理完毕才能再输出响应，最后 Tomcat 才回收这个线程。

我们来思考这样一个问题，假如你的 Web 应用需要较长的时间来处理请求（比如数据库查询或者等待下游的服务调用返回），那么 Tomcat 线程一直不回收，会占用系统资源，在极端情况下会导致“线程饥饿”，也就是说 Tomcat 和 Jetty 没有更多的线程来处理新的请求。


那该如何解决这个问题呢？方案是 Servlet 3.0 中引入的异步 Servlet。主要是在 Web 应用里启动一个单独的线程来执行这些比较耗时的请求，而 Tomcat 线程立即返回，不再等待

Web 应用将请求处理完，这样 Tomcat 线程可以立即被回收到线程池，用来响应其他请求，降低了系统的资源消耗，同时还能提高系统的吞吐量。

今天我们就来学习一下如何开发一个异步 Servlet，以及异步 Servlet 的工作原理，也就是 Tomcat 是如何支持异步 Servlet 的，让你彻底理解它的来龙去脉。

异步 Servlet 示例

我们先通过一个简单的示例来了解一下异步 Servlet 的实现。

 复制代码

```
1 @WebServlet(urlPatterns = {"/async"}, asyncSupported = true)
2 public class AsyncServlet extends HttpServlet {
3
4     //Web 应用线程池，用来处理异步 Servlet
5     ExecutorService executor = Executors.newSingleThreadExecutor();
6
7     public void service(HttpServletRequest req, HttpServletResponse resp) {
8         //1. 调用 startAsync 或者异步上下文
9         final AsyncContext ctx = req.startAsync();
10
11         // 用线程池来执行耗时操作
12         executor.execute(new Runnable() {
13
14             @Override
15             public void run() {
16
17                 // 在这里做耗时的操作
18                 try {
19                     ctx.getResponse().getWriter().println("Handling Async Servlet");
20                 } catch (IOException e) {}
21
22                 //3. 异步 Servlet 处理完了调用异步上下文的 complete 方法
23                 ctx.complete();
24             }
25
26         });
27     }
28 }
```

上面的代码有三个要点：

1. 通过注解的方式来注册 Servlet，除了 `@WebServlet` 注解，还需要加上 `asyncSupported=true` 的属性，表明当前的 Servlet 是一个异步 Servlet。
2. Web 应用程序需要调用 Request 对象的 `startAsync` 方法来拿到一个异步上下文 `AsyncContext`。这个上下文保存了请求和响应对象。
3. Web 应用需要开启一个新线程来处理耗时的操作，处理完成后需要调用 `AsyncContext` 的 `complete` 方法。目的是告诉 Tomcat，请求已经处理完成。

这里请你注意，虽然异步 Servlet 允许用更长的时间来处理请求，但是也有超时限制的，默认是 30 秒，如果 30 秒内请求还没处理完，Tomcat 会触发超时机制，向浏览器返回超时错误，如果这个时候你的 Web 应用再调用 `ctx.complete` 方法，会得到一个 `IllegalStateException` 异常。

异步 Servlet 原理

通过上面的例子，相信你对 Servlet 的异步实现有了基本的理解。要理解 Tomcat 在这个过程中都做了什么事情，关键就是要弄清楚 `req.startAsync` 方法和 `ctx.complete` 方法都做了什么。

startAsync 方法

`startAsync` 方法其实就是创建了一个异步上下文 `AsyncContext` 对象，`AsyncContext` 对象的作用是保存请求的中间信息，比如 Request 和 Response 对象等上下文信息。你来思考一下为什么需要保存这些信息呢？


这是因为 Tomcat 的工作线程在 `Request.startAsync` 调用之后，就直接结束回到线程池中了，线程本身不会保存任何信息。也就是说一个请求到服务端，执行到一半，你的 Web 应用正在处理，这个时候 Tomcat 的工作线程没了，这就需要有个缓存能够保存原始的 Request 和 Response 对象，而这个缓存就是 `AsyncContext`。

有了 `AsyncContext`，你的 Web 应用通过它拿到 request 和 response 对象，拿到 Request 对象后就可以读取请求信息，请求处理完了还需要通过 Response 对象将 HTTP 响应发送给浏览器。

除了创建 `AsyncContext` 对象，`startAsync` 还需要完成一个关键任务，那就是告诉 Tomcat 当前的 Servlet 处理方法返回时，不要把响应发到浏览器，因为这个时候，响应还

没生成呢；并且不能把 Request 对象和 Response 对象销毁，因为后面 Web 应用还要用呢。


在 Tomcat 中，负责 flush 响应数据的是 CoyoteAdaptor，它还会销毁 Request 对象和 Response 对象，因此需要通过某种机制通知 CoyoteAdaptor，具体来说是通过下面这行代码：

 复制代码

```
1 this.request.getCoyoteRequest().action(ActionCode.ASYNC_START, this);
```

你可以把它理解为一个 Callback，在这个 action 方法里设置了 Request 对象的状态，设置它为一个异步 Servlet 请求。

我们知道连接器是调用 CoyoteAdapter 的 service 方法来处理请求的，而 CoyoteAdapter 会调用容器的 service 方法，当容器的 service 方法返回时，CoyoteAdapter 判断当前的请求是不是异步 Servlet 请求，如果是，就不会销毁 Request 和 Response 对象，也不会把响应信息发到浏览器。你可以通过下面的代码理解一下，这是 CoyoteAdapter 的 service 方法，我对它进行了简化：

 复制代码

```
1 public void service(org.apache.coyote.Request req, org.apache.coyote.Response res) {
2
3     // 调用容器的 service 方法处理请求
4     connector.getService().getContainer().getPipeline().
5         getFirst().invoke(request, response);
6
7     // 如果是异步 Servlet 请求，仅仅设置一个标志，
8     // 否则说明是同步 Servlet 请求，就将响应数据刷到浏览器
9     if (request.isAsync()) {
10         async = true;
11     } else {
12         request.finishRequest();
13         response.finishResponse();
14     }
15
16     // 如果不是异步 Servlet 请求，就销毁 Request 对象和 Response 对象
17     if (!async) {
18         request.recycle();
19         response.recycle();
20     }
21 }
```

接下来，当 CoyoteAdaptor 的 service 方法返回到 ProtocolHandler 组件时，ProtocolHandler 判断返回值，如果当前请求是一个异步 Servlet 请求，它会把当前 Socket 的协议处理器 Processor 缓存起来，将 SocketWrapper 对象和相应的 Processor 存到一个 Map 数据结构里。

[📄 复制代码](#)

```
1 private final Map<S,Processor> connections = new ConcurrentHashMap<>();
```

之所以要缓存是因为这个请求接下来还要接着处理，还是由原来的 Processor 来处理，通过 SocketWrapper 就能从 Map 里找到相应的 Processor。

complete 方法

接着我们再来看关键的 ctx.complete 方法，当请求处理完成时，Web 应用调用这个方法。那么这个方法做了些什么事情呢？最重要的就是把响应数据发送到浏览器。

这件事情不能由 Web 应用线程来做，也就是说 ctx.complete 方法不能直接把响应数据发送到浏览器，因为这件事情应该由 Tomcat 线程来做，但具体怎么做呢？

我们知道，连接器中的 Endpoint 组件检测到有请求数据达到时，会创建一个 SocketProcessor 对象交给线程池去处理，因此 Endpoint 的通信处理和具体请求处理在两个线程里运行。

在异步 Servlet 的场景里，Web 应用通过调用 ctx.complete 方法时，也可以生成一个新的 SocketProcessor 任务类，交给线程池处理。对于异步 Servlet 请求来说，相应的 Socket 和协议处理组件 Processor 都被缓存起来了，并且这些对象都可以通过 Request 对象拿到。

讲到这里，你可能已经猜到 ctx.complete 是如何实现的了：

[📄 复制代码](#)

```
1 public void complete() {
2     // 检查状态合法性，我们先忽略这句
3     check();
4
5     // 调用 Request 对象的 action 方法，其实就是通知连接器，这个异步请求处理完了
6     request.getCoyoteRequest().action(ActionCode.ASYNC_COMPLETE, null);
7
8 }
```

我们可以看到 complete 方法调用了 Request 对象的 action 方法。而在 action 方法里，则是调用了 Processor 的 processSocketEvent 方法，并且传入了操作码 OPEN_READ。

[📄 复制代码](#)

```
1 case ASYNC_COMPLETE: {
2     clearDispatches();
3     if (asyncStateMachine.asyncComplete()) {
4         processSocketEvent(SocketEvent.OPEN_READ, true);
5     }
6     break;
7 }
```

我们接着看 processSocketEvent 方法，它调用 SocketWrapper 的 processSocket 方法：

[📄 复制代码](#)

```
1 protected void processSocketEvent(SocketEvent event, boolean dispatch) {
2     SocketWrapperBase<?> socketWrapper = getSocketWrapper();
3     if (socketWrapper != null) {
4         socketWrapper.processSocket(event, dispatch);
5     }
6 }
```

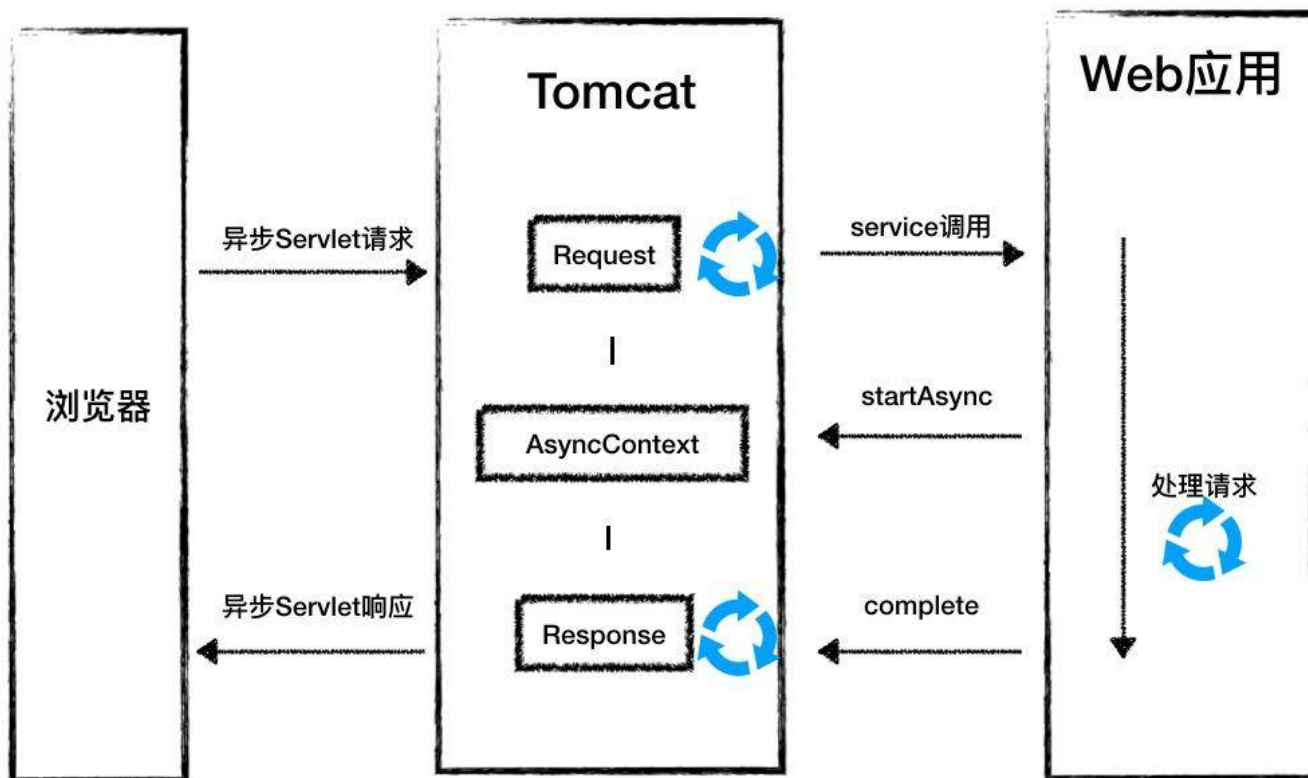
而 SocketWrapper 的 processSocket 方法会创建 SocketProcessor 任务类，并通过 Tomcat 线程池来处理：

[📄 复制代码](#)

```
1 public boolean processSocket(SocketWrapperBase<S> socketWrapper,
2     SocketEvent event, boolean dispatch) {
3
4     if (socketWrapper == null) {
5         return false;
6     }
7
8     SocketProcessorBase<S> sc = processorCache.pop();
9     if (sc == null) {
10         sc = createSocketProcessor(socketWrapper, event);
11     } else {
12         sc.reset(socketWrapper, event);
13     }
14     // 线程池运行
15     Executor executor = getExecutor();
16     if (dispatch && executor != null) {
17         executor.execute(sc);
18     } else {
19         sc.run();
20     }
21 }
```

请你注意 `createSocketProcessor` 函数的第二个参数是 `SocketEvent`，这里我们传入的是 `OPEN_READ`。通过这个参数，我们就能控制 `SocketProcessor` 的行为，因为我们不需要再把请求发送到容器进行处理，只需要向浏览器端发送数据，并且重新在这个 `Socket` 上监听新的请求就行了。

最后我通过一张在帮你理解一下整个过程：



本期精华

非阻塞 I/O 模型可以利用很少的线程处理大量的连接，提高了并发度，本质就是通过一个 Selector 线程查询多个 Socket 的 I/O 事件，减少了线程的阻塞等待。

同样，异步 Servlet 机制也是减少了线程的阻塞等待，将 Tomcat 线程和业务线程分开，Tomcat 线程不再等待业务代码的执行。

那什么样的场景适合异步 Servlet 呢？适合的场景有很多，最主要的还是根据你的实际情况，如果你拿不准是否适合异步 Servlet，就看一条：如果你发现 Tomcat 的线程不够了，大量线程阻塞在等待 Web 应用的处理上，而 Web 应用又没有优化的空间了，确实需要长时间处理，这个时候你不妨尝试一下异步 Servlet。

课后思考

异步 Servlet 将 Tomcat 线程和 Web 应用线程分开，体现了隔离的思想，也就是把不同的业务处理所使用的资源隔离开，使得它们互不干扰，尤其是低优先级的业务不能影响高优先级的业务。你可以思考一下，在你的 Web 应用内部，是不是也可以运用这种设计思想呢？

不知道今天的内容你消化得如何？如果还有疑问，请大胆的在留言区提问，也欢迎你把你的课后思考和心得记录下来，与我和其他同学一起讨论。如果你觉得今天有所收获，欢迎你把

它分享给你的朋友。



深入拆解 Tomcat & Jetty

从源码角度深度探索 Java 中间件

李号双

eBay 技术主管



新版升级：点击「 请朋友读」，20位好友免费读，邀请订阅更有**现金**奖励。

© 版权归极客邦科技所有，未经许可不得传播售卖。页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇 26 | Context容器（下）：Tomcat如何实现Servlet规范？

下一篇 28 | 新特性：Tomcat和Jetty如何处理Spring Boot应用？

精选留言 (5)

写留言



echo_陈

2019-07-11

我感觉，异步servlet只能说让tomcat有机会接受更多的请求，但并不能提升服务的并发吞吐量，因为如果业务操作本身还是慢的话，业务线程池仍然会被占满，后面提交的任务会等待。

作者回复: 同意，还有就是业务处理一般阻塞在io等待上，越是IO密集型应用，越需要配置更多线程。

1

2



非想

2019-07-14

老师您好，请问下怎么理解tomcat线程和servlet线程，它们有什么区别，又是怎么关联的呢？



Feng.X

2019-07-13

老师，请问对Map<S,Processor> connections里的Processor的取出操作是在SocketWrapper的processSocket 方法里吗？

展开 ∨



梁中华

2019-07-12

异步sevlet内部的业务应用中的IO也需要异步IO支持吧，就像vertx的异步模式，否则都堵塞在业务线程上就没意义了

展开 ∨



nightmare

2019-07-11

异步servlet相当于用户控制开启和完成，在protohandler通过类似future的机制来完成异步操作

