

09 | 比较：Jetty架构特点之Connector组件

2019-05-30 李号双

深入拆解Tomcat & Jetty

[进入课程 >](#)



讲述：李号双

时长 11:50 大小 10.84M

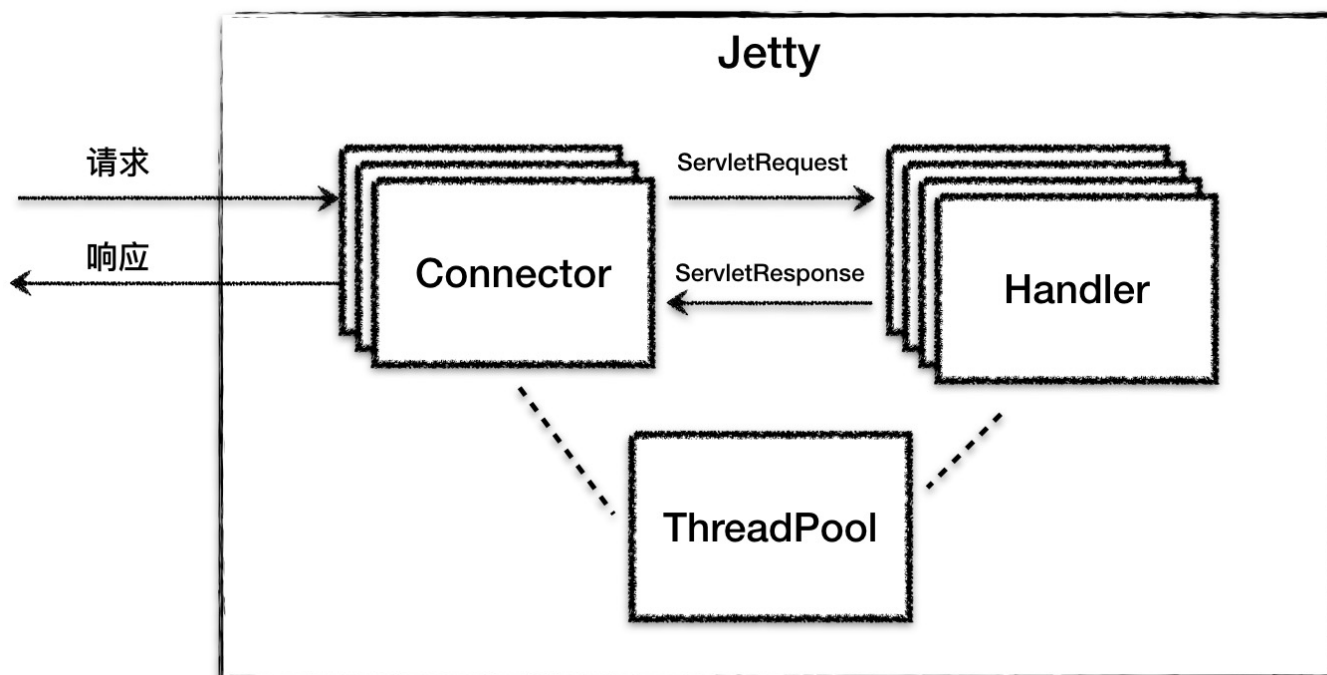


经过专栏前面几期的学习，相信你对 Tomcat 的整体架构和工作原理有了基本了解。但是 Servlet 容器并非只有 Tomcat 一家，还有别的架构设计思路吗？今天我们就来看看 Jetty 的设计特点。

Jetty 是 Eclipse 基金会的一个开源项目，和 Tomcat 一样，Jetty 也是一个“HTTP 服务器 + Servlet 容器”，并且 Jetty 和 Tomcat 在架构设计上有不少相似的地方。但同时 Jetty 也有自己的特点，主要是更加小巧，更易于定制化。Jetty 作为一名后起之秀，应用范围也越来越广，比如 Google App Engine 就采用了 Jetty 来作为 Web 容器。Jetty 和 Tomcat 各有特点，所以今天我会和你重点聊聊 Jetty 在哪些地方跟 Tomcat 不同。通过比较它们的差异，一方面希望可以继续加深你对 Web 容器架构设计的理解，另一方面也让你更清楚它们的设计区别，并根据它们的特点来选用这两款 Web 容器。

鸟瞰 Jetty 整体架构

简单来说，Jetty Server 就是由多个 Connector（连接器）、多个 Handler（处理器），以及一个线程池组成。整体结构请看下面这张图。

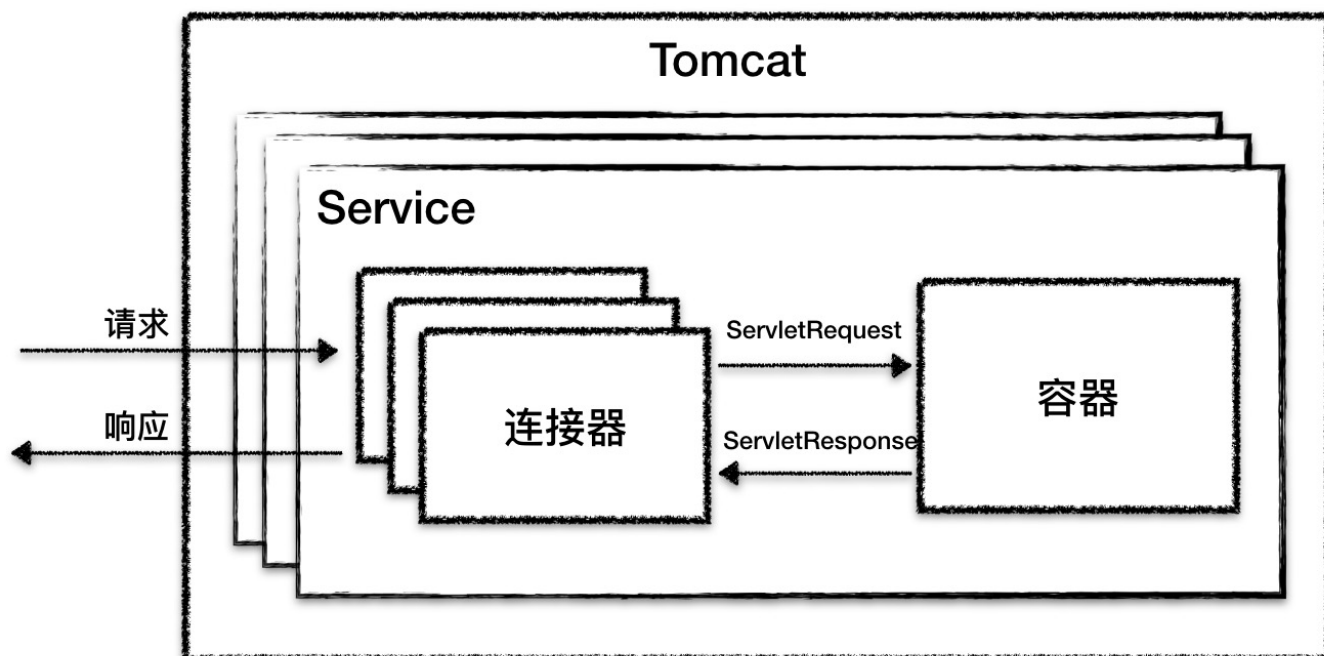


跟 Tomcat 一样，Jetty 也有 HTTP 服务器和 Servlet 容器的功能，因此 Jetty 中的 Connector 组件和 Handler 组件分别来实现这两个功能，而这两个组件工作时所需要的线程资源都直接从一个全局线程池 ThreadPool 中获取。

Jetty Server 可以有多个 Connector 在不同的端口上监听客户请求，而对于请求处理的 Handler 组件，也可以根据具体场景使用不同的 Handler。这样的设计提高了 Jetty 的灵活性，需要支持 Servlet，则可以使用 ServletHandler；需要支持 Session，则再增加一个 SessionHandler。也就是说我们可以不使用 Servlet 或者 Session，只要不配置这个 Handler 就行了。

为了启动和协调上面的核心组件工作，Jetty 提供了一个 Server 类来做这个事情，它负责创建并初始化 Connector、Handler、ThreadPool 组件，然后调用 start 方法启动它们。

我们对比一下 Tomcat 的整体架构图，你会发现 Tomcat 在整体上跟 Jetty 很相似，它们的第一个区别是 Jetty 中没有 Service 的概念，Tomcat 中的 Service 包装了多个连接器和一个容器组件，一个 Tomcat 实例可以配置多个 Service，不同的 Service 通过不同的连接器监听不同的端口；而 Jetty 中 Connector 是被所有 Handler 共享的。



它们的第二个区别是，在 Tomcat 中每个连接器都有自己的线程池，而在 Jetty 中所有的 Connector 共享一个全局的线程池。

讲完了 Jetty 的整体架构，接下来我来详细分析 Jetty 的 Connector 组件的设计，下一期我将分析 Handler 组件的设计。

Connector 组件

跟 Tomcat 一样，Connector 的主要功能是对 I/O 模型和应用层协议的封装。I/O 模型方面，最新的 Jetty 9 版本只支持 NIO，因此 Jetty 的 Connector 设计有明显的 Java NIO 通信模型的痕迹。至于应用层协议方面，跟 Tomcat 的 Processor 一样，Jetty 抽象出了 Connection 组件来封装应用层协议的差异。


Java NIO 早已成为程序员的必备技能，并且也经常出现在面试题中。接下来我们一起来看看 Jetty 是如何实现 NIO 模型的，以及它是怎么用 Java NIO 的。

Java NIO 回顾

关于 Java NIO 编程，如果你还不太熟悉，可以先学习这一[系列文章](#)。Java NIO 的核心组件是 Channel、Buffer 和 Selector。Channel 表示一个连接，可以理解为一个 Socket，通过它可以读取和写入数据，但是并不能直接操作数据，需要通过 Buffer 来中转。


Selector 可以用来检测 Channel 上的 I/O 事件，比如读就绪、写就绪、连接就绪，一个 Selector 可以同时处理多个 Channel，因此单个线程可以监听多个 Channel，这样会大量减少线程上下文切换的开销。下面我们通过一个典型的服务端 NIO 程序来回顾一下如何使用这些组件。

首先，创建服务端 Channel，绑定监听端口并把 Channel 设置为非阻塞方式。

 复制代码

```
1 ServerSocketChannel server = ServerSocketChannel.open();
2 server.socket().bind(new InetSocketAddress(port));
3 server.configureBlocking(false);
```


然后，创建 Selector，并在 Selector 中注册 Channel 感兴趣的事件 OP_ACCEPT，告诉 Selector 如果客户端有新的连接请求到这个端口就通知我。

 复制代码

```
1 Selector selector = Selector.open();
2 server.register(selector, SelectionKey.OP_ACCEPT);
```

接下来，Selector 会在一个死循环里不断地调用 select() 去查询 I/O 状态，select() 会返回一个 SelectionKey 列表，Selector 会遍历这个列表，看看是否有“客户”感兴趣的事件，如果有，就采取相应的动作。

比如下面这个例子，如果有新的连接请求，就会建立一个新的连接。连接建立后，再注册 Channel 的可读事件到 Selector 中，告诉 Selector 我对这个 Channel 上是否有新的数据到达感兴趣。

 复制代码

```
1 while (true) {
2     selector.select();// 查询 I/O 事件
3     for (Iterator<SelectionKey> i = selector.selectedKeys().iterator(); i.hasNext())
4         SelectionKey key = i.next();
5         i.remove();
6
7         if (key.isAcceptable()) {
8             // 建立一个新连接
```

```

9         SocketChannel client = server.accept();
10        client.configureBlocking(false);
11
12        // 连接建立后, 告诉 Selector, 我现在对 I/O 可读事件感兴趣
13        client.register(selector, SelectionKey.OP_READ);
14    }
15 }
16 }

```

简单回顾完服务端 NIO 编程之后, 你会发现服务端在 I/O 通信上主要完成了三件事情: **监听连接、I/O 事件查询以及数据读写**。因此 Jetty 设计了 **Acceptor**、**SelectorManager** 和 **Connection** 来分别做这三件事情, 下面我分别来说说这三个组件。

Acceptor

顾名思义, Acceptor 用于接受请求, 跟 Tomcat 一样, Jetty 也有独立的 Acceptor 线程组用于处理连接请求。在 Connector 的实现类 ServerConnector 中, 有一个 `_acceptors` 的数组, 在 Connector 启动的时候, 会根据 `_acceptors` 数组的长度创建对应数量的 Acceptor, 而 Acceptor 的个数可以配置。

 复制代码

```

1 for (int i = 0; i < _acceptors.length; i++)
2 {
3     Acceptor a = new Acceptor(i);
4     getExecutor().execute(a);
5 }

```

Acceptor 是 ServerConnector 中的一个内部类, 同时也是一个 Runnable, Acceptor 线程是通过 `getExecutor()` 得到的线程池来执行的, 前面提到这是一个全局的线程池。

Acceptor 通过阻塞的方式来接受连接, 这一点跟 Tomcat 也是一样的。

 复制代码


```

1 public void accept(int acceptorID) throws IOException
2 {
3     ServerSocketChannel serverChannel = _acceptChannel;
4     if (serverChannel != null && serverChannel.isOpen())

```

```
5  {
6    // 这里是阻塞的
7    SocketChannel channel = serverChannel.accept();
8    // 执行到这里时说明有请求进来了
9    accepted(channel);
10 }
11 }
```


接受连接成功后会调用 `accepted()` 函数，`accepted()` 函数中会将 `SocketChannel` 设置为非阻塞模式，然后交给 `Selector` 去处理，因此这也就到了 `Selector` 的地界了。

 复制代码

```
1 private void accepted(SocketChannel channel) throws IOException
2 {
3     channel.configureBlocking(false);
4     Socket socket = channel.socket();
5     configure(socket);
6     // _manager 是 SelectorManager 实例，里面管理了所有的 Selector 实例
7     _manager.accept(channel);
8 }
```

SelectorManager


Jetty 的 `Selector` 由 `SelectorManager` 类管理，而被管理的 `Selector` 叫作 `ManagedSelector`。`SelectorManager` 内部有一个 `ManagedSelector` 数组，真正干活的是 `ManagedSelector`。咱们接着上面分析，看看在 `SelectorManager` 在 `accept` 方法里做了什么。

 复制代码

```
1 public void accept(SelectableChannel channel, Object attachment)
2 {
3     // 选择一个 ManagedSelector 来处理 Channel
4     final ManagedSelector selector = chooseSelector();
5     // 提交一个任务 Accept 给 ManagedSelector
6     selector.submit(selector.new Accept(channel, attachment));
7 }
```



SelectorManager 从本身的 Selector 数组中选择一个 Selector 来处理这个 Channel，并创建一个任务 Accept 交给 ManagedSelector，ManagedSelector 在处理这个任务主要做了两步：

第一步，调用 Selector 的 register 方法把 Channel 注册到 Selector 上，拿到一个 SelectionKey。

 复制代码

```
1  _key = _channel.register(selector, SelectionKey.OP_ACCEPT, this);
```

第二步，创建一个 EndPoint 和 Connection，并跟这个 SelectionKey (Channel) 绑在一起：

 复制代码

```
1  private void createEndPoint(SelectableChannel channel, SelectionKey selectionKey) throw:
2  {
3      //1. 创建 Endpoint
4      Endpoint endPoint = _selectorManager.newEndPoint(channel, this, selectionKey);
5
6      //2. 创建 Connection
7      Connection connection = _selectorManager.newConnection(channel, endPoint, selectionKey);
8
9      //3. 把 Endpoint、Connection 和 SelectionKey 绑在一起
10     endPoint.setConnection(connection);
11     selectionKey.attach(endPoint);
12
13 }
```


上面这两个过程是什么意思呢？打个比方，你到餐厅吃饭，先点菜（注册 I/O 事件），服务员（ManagedSelector）给你一个单子（SelectionKey），等菜做好了（I/O 事件到了），服务员根据单子就知道是哪桌点了这个菜，于是喊一嗓子某某桌的菜做好了（调用了绑定在 SelectionKey 上的 EndPoint 的方法）。

这里需要你特别注意的是，ManagedSelector 并没有调用直接 EndPoint 的方法去处理数据，而是通过调用 EndPoint 的方法**返回一个 Runnable，然后把这个 Runnable 扔给线程池执行**，所以你能猜到，这个 Runnable 才会去真正读数据和处理请求。

Connection

这个 Runnable 是 EndPoint 的一个内部类，它会调用 Connection 的回调方法来处理请求。Jetty 的 Connection 组件类比就是 Tomcat 的 Processor，负责具体协议的解析，得到 Request 对象，并调用 Handler 容器进行处理。下面我简单介绍一下它的具体实现类 HttpConnection 对请求和响应的处理过程。

请求处理：HttpConnection 并不会主动向 EndPoint 读取数据，而是向在 EndPoint 中注册一堆回调方法：

 复制代码

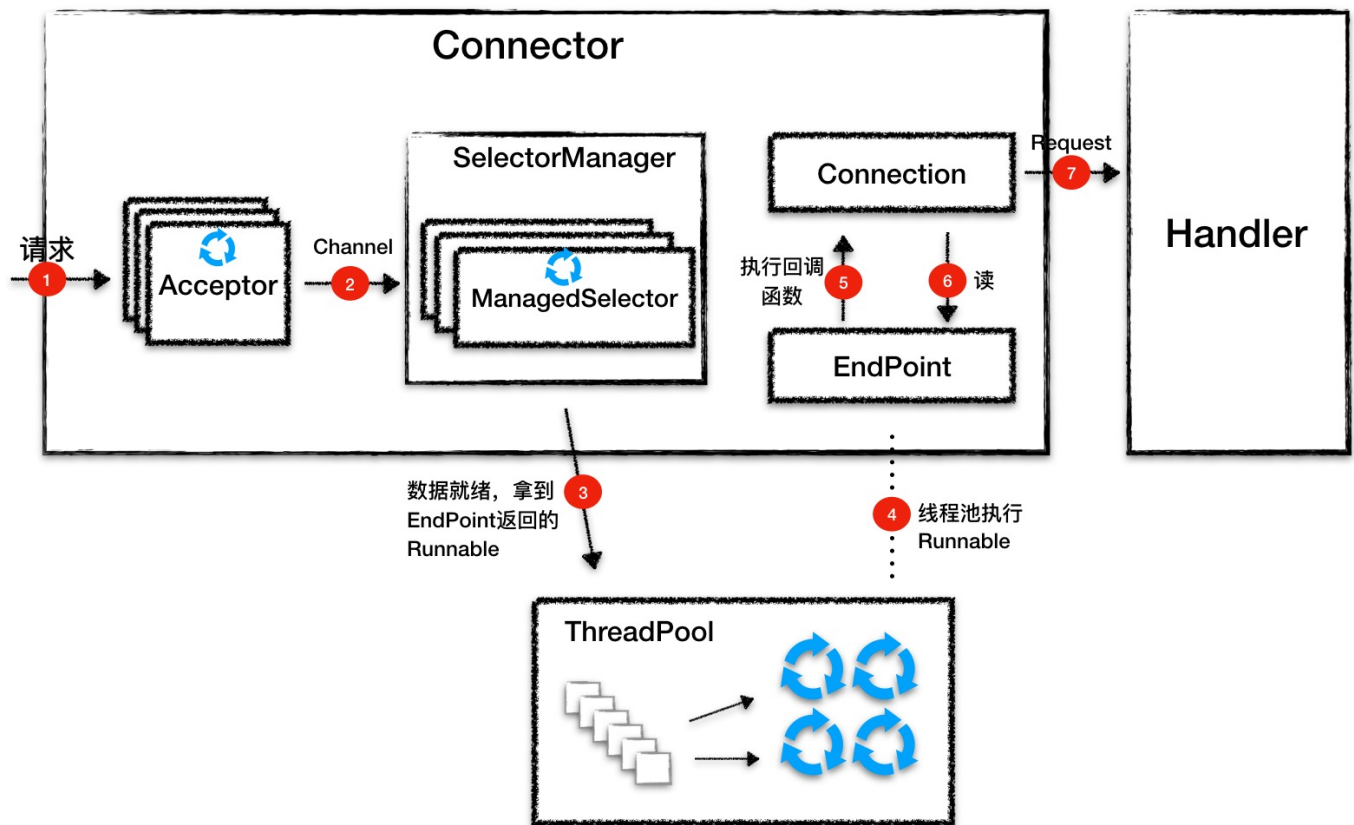
```
1 getEndPoint().fillInterested(_readCallback);
```

这段代码就是告诉 EndPoint，数据到了你就调我这些回调方法 _readCallback 吧，有点异步 I/O 的感觉，也就是说 Jetty 在应用层面模拟了异步 I/O 模型。

而在回调方法 _readCallback 里，会调用 EndPoint 的接口去读数据，读完后让 HTTP 解析器去解析字节流，HTTP 解析器会将解析后的数据，包括请求行、请求头相关信息存到 Request 对象里。

响应处理：Connection 调用 Handler 进行业务处理，Handler 会通过 Response 对象来操作响应流，向流里面写入数据，HttpConnection 再通过 EndPoint 把数据写到 Channel，这样一次响应就完成了。

到此你应该了解了 Connector 的工作原理，下面我画张图再来回顾一下 Connector 的工作流程。



1.Acceptor 监听连接请求，当有连接请求到达时就接受连接，一个连接对应一个 Channel，Acceptor 将 Channel 交给 ManagedSelector 来处理。

2.ManagedSelector 把 Channel 注册到 Selector 上，并创建一个 EndPoint 和 Connection 跟这个 Channel 绑定，接着就不断地检测 I/O 事件。

3.I/O 事件到了就调用 EndPoint 的方法拿到一个 Runnable，并扔给线程池执行。

4. 线程池中调度某个线程执行 Runnable。

5.Runnable 执行时，调用回调函数，这个回调函数是 Connection 注册到 EndPoint 中的。

6. 回调函数内部实现，其实就是调用 EndPoint 的接口方法来读数据。

7.Connection 解析读到的数据，生成请求对象并交给 Handler 组件去处理。

本期精华

Jetty Server 就是由多个 Connector、多个 Handler，以及一个线程池组成，在设计上简洁明了。

Jetty 的 Connector 只支持 NIO 模型，跟 Tomcat 的 NioEndpoint 组件一样，它也是通过 Java 的 NIO API 实现的。我们知道，Java NIO 编程有三个关键组件：Channel、Buffer 和 Selector，而核心是 Selector。为了方便使用，Jetty 在原生 Selector 组件的基础上做了一些封装，实现了 ManagedSelector 组件。

在线程模型设计上 Tomcat 的 NioEndpoint 跟 Jetty 的 Connector 是相似的，都是用一个 Acceptor 数组监听连接，用一个 Selector 数组侦测 I/O 事件，用一个线程池执行请求。它们的不同点在于，Jetty 使用了一个全局的线程池，所有的线程资源都是从线程池来分配。

Jetty Connector 设计中的一大特点是，使用了回调函数来模拟异步 I/O，比如 Connection 向 EndPoint 注册了一堆回调函数。它的本质**将函数当作一个参数来传递**，告诉对方，你准备好了就调这个回调函数。

课后思考

Jetty 的 Connector 主要完成了三件事：接收连接、I/O 事件查询以及数据读写。因此 Jetty 设计了 Acceptor、SelectorManager 和 Connection 来做这三件事情。今天的思考题是，为什么要把这些组件跑在不同的线程里呢？

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 08 | Tomcat的“高层们”都负责做什么？

下一篇 10 | 比较：Jetty架构特点之Handler组件

精选留言 (26)

写留言



姬晓东

2019-05-30

4

源码都是怎么导入，怎么编译，怎么看呢

展开

作者回复: 下载这里的源码，直接IDE打开，设断点就行。

<https://github.com/jetty-project/embedded-jetty-jsp>



-W.LI-

2019-06-01

3

老师好!在线程模型设计上 Tomcat 的 NioEndpoint 跟 Jetty 的 Connector 是相似的,都是用一个 Acceptor 数组监听连接,用一个 Selector 数组侦测 I/O 事件。这句话怎么理解啊?

问题1:Acceptor 数组监听连接,监听的是一次TCP链接么?

...

展开 ∨

作者回复: 简单可以这样理解:

1. Acceptor就是不停的调accept函数,接收新的连接
2. Selector不停的调select函数,查询某个Channel上是否有数据可读
3. 同一个浏览器发过来的请求会重用TCP连接,也就是用同一个Channel

Channel是非阻塞的,连接器里维护了这些Channel实例,过了一段时间超时到了channel还没数据到来,表面用户长时间没有操作浏览器,这时Tomcat才关闭这个channel。

◀ ▶



锦

2019-05-30

👍 3

使用不同的线程是为了合理的使用全局线程池。

我有两个问题请教老师:

问题一:负责读写的socket与handle线程是什么对应关系呢?多对1,还是1对1?

问题二:如果有很多tcp建立连接后迟迟没有写入数据导致连接请求堵塞,或者如果有很多handle在处理耗时io操作时,...

展开 ∨

作者回复: 1) 一个Socket上可以接收多个HTTP请求,每次请求跟一个Handler线程是一一对应的关系,因为keepalive,一次请求处理完成后Socket不会立即关闭,下一次再来请求,会分配一个新的Handler线程。

2) 很好的问题,这就是为什么Servlet3.0中引入了异步Servlet的概念,就是说遇到耗时的IO操作, Tomcat的线程会立即返回,当业务线程处理完后,再调用Tomcat的线程将响应发回给浏览器,异步Servlet的原理后面有专门的一篇来介绍。

◀ ▶



Li Shundu...

2019-06-01

👍 2

有两个问题请教老师：

问题一：根据文章看，Jetty中有多个Acceptor组件，请问这些Acceptor背后是共享同一个ServerSocketChannel？还是每个Acceptor有自己的ServerSocketChannel？如果有多个ServerSocketChannel的话，这些ServerSocketChannel如何做到监听同一个端口？连接到来时如何决定分配到哪一个ServerSocketChannel？...

展开 ▾

作者回复: 不错的問題。

- 1) 多个Acceptor共享同一个ServerSocketChannel。多个Acceptor线程调用同一个ServerSocketChannel的accept方法，由操作系统保证线程安全
- 2) 直接调用accept方法，编程上简单一些，否则每个Acceptor又要自己维护一个Selector。
- 3) 每个ManagedSelector都有自己的Selector，多个Selector可以并行管理大量的channel，提高并发，连接请求到达时采用Round Robin的方式选择ManagedSelector。



802.11

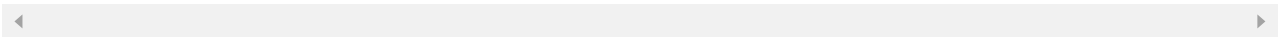
2019-06-02

👍 1

老师，在一个类里再写接口或者类，一般是基于什么样的设计思想呢

展开 ▾

作者回复: 内部类，作为一个类的实现细节，外部不需要知道，通过内部类的方式实现显得整洁干净。



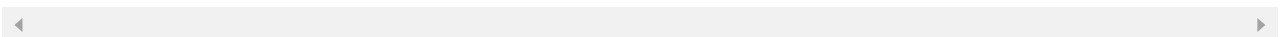
kxkq2000

2019-06-01

👍 1

分在不同的线程里我认为是这样分工明确好比工厂流水线最大化提升处理能力。
我有个疑问是用全局线程池真的好吗，不是应该根据任务类型分配线程池的吗？用全局的不会互相干扰吗？

作者回复: 全局线程池和多个隔离的线程池各有优缺点。全局的线程池方便控制线程总数，防止过多的线程导致大量线程切换。隔离的线程池可以控制任务优先级，确保低优先级的任务不会去抢高优先级任务的线程。



Lrwin



2019-05-30



感觉jetty就是一个netty模型

展开 ∨

作者回复: 说的很对, Tomcat和Jetty相比, Jetty的I/O线程模型更像Netty, 后面会讲到Jetty的EatWhatYouKill线程策略, 其实就是Netty 4.0中的线程模型。



强哥

2019-05-30

1

说了各自的特点。但是感觉缺少关键性的对比, 以及背后设计的理念, 建议再深入探讨各自的主要差异及场景

作者回复: 随着讲解的深入, 会涉及这部分内容。Jetty和Tomcat没有本质区别, 一般来说Jetty比较小巧, 又可以高度裁剪和定制, 因此适合放在嵌入式设备等对内存资源比较紧张场合。而Tomcat比较成熟稳定, 对企业级应用支持比较好。



Monday

2019-05-30

1

tomcat还没学透, 就学jetty, 两者还相互比较, 生怕自己混淆了

展开 ∨

作者回复: 这个模块还只是架构层面的学习, 下一个模块会深入到细节。



Lc

2019-05-30

1

Jetty作为后起之秀, 跟tomcat相比, 它的优势在哪儿? 他们的设计思路不同, 我们自己在设计的时候应该依据什么来确定使用哪种呢?

作者回复: Jetty的优势是轻巧, 代码量小, 比如它只支持非阻塞IO, 这意味着把它加载到内存后占用内存空间也小, 另外还可以把它裁剪的更小, 比如不需要Session支持, 可以方便的去掉相应的Handler。





Geek_28b75...

2019-06-04



老师，麻烦您给说说bio和nio的区别，表面上的差别我看过了，能不能从操作系统角度给讲解一下呢？麻烦您了，实在有点难理解

作者回复: 第14篇会详细介绍，图文并茂。



木木木

2019-06-04



从理论上，Accept接受请求，SelectorManager处理客户端socket组的轮询，Connection负责数据的收发，三者关系是异步的，可以分成不同的线程。特别是jetty的Accept是阻塞的更没办法。

有几点疑问，按我的理解，虽然是全局线程池，这三者应该有独立的线程最大数量配置吧，这样应该更可控点，否则相互抢占，会不会有问题。...

展开 ∨

作者回复: 1) 对的，全局线程池有最大线程数限制，的确会互相争抢，一般把线程池设大点，如果还发生线程争抢和线程饥饿，要考虑web应用实现是不是有问题，导致大量线程阻塞。

2) 不是BIO呢，BIO说的是接受到连接后，对这个连接上的请求的处理一直用同一个线程。这里只是接收连接这一步是阻塞的，后续用Selector检测IO事件，应用线程没有阻塞。

3) 合成一个Selector负担太重了



Geek_00d56...

2019-06-03



放在不同线程里，提高并发量。

展开 ∨



Lrwin

2019-06-03



创建多个Acceptor 真的有必要吗？一个线程一个selector就可以用于监听所有的连接呀、。

netty中bossgroup也就只有一个eventloop来监听所有连接。

作者回复: Netty中BossGroup的个数可以设置，也可以启多个。



-W.LI-

2019-06-02



老师好!源码里面有些全局变量希望给下注释解释下。都搞不清楚acceptor怎么拿到selectormanager对象的。然后_manager.accept(channel)调用的是一个入参，下面是两个入参的重载方法。基础比较差希望老师照顾下。

展开 ∨

作者回复: ☺，_manager是ServerConnector直接new出来的:

```
_manager = newSelectorManager(getExecutor(), getScheduler(),selectors);
```

如果你对某个问题感兴趣想弄清楚，还是需要自己打开源码看一看，因为专栏的篇幅实在有限，你可以根据专栏的分析先弄清楚大概流程，然后再去学习细节。



-W.LI-

2019-06-02



老师好，跑在不同的线程里是为了解耦么?实在想不出，告诉答案吧

展开 ∨

作者回复: 反过来想，如果等待连接到达，接收连接、等待数据到达、数据读取和请求处理（等待应用处理完）都在一个线程里，这中间线程可能大部分时间都在“等待”，没有干活，而线程资源是很宝贵的。并且线程阻塞会发生线程上下文切换，浪费CPU资源。



why

2019-06-02



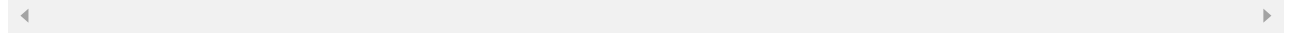
- Jetty 也是 Http 服务器 + Servlet 容器, 更小巧, 更易于定制
- Jetty 架构: 多个 Connector + 多个 Handler + 一个全局线程池(Connector 和 Handler

共享)

- 多个 Connector 在不同端口监听请求, 可以根据应用场景选择 Handler :
ServletHandler 和 SessionHandler...

展开 ▾

作者回复: 📬



非洲铜

2019-05-31



一直没用过jetty当web容器, 看的有点懵逼

展开 ▾



天天向上

2019-05-30



为了分工协作, IO工作, 业务处理工作分成两大流程环节, 互不干扰, 分工明确,
高效复用IO线程

展开 ▾



朱晋君

2019-05-30



acceptor、connector和selector各自承担不同工作, 用不同线程执行,

- 1.用异步和通知机制, 效率更高
- 2.个线程干一个事, 代码实现更加简单
- 3.更容易定位和分析故障