

19 | 比较：Jetty的线程策略 EatWhatYouKill

2019-06-22 李号双

深入拆解Tomcat & Jetty

[进入课程 >](#)

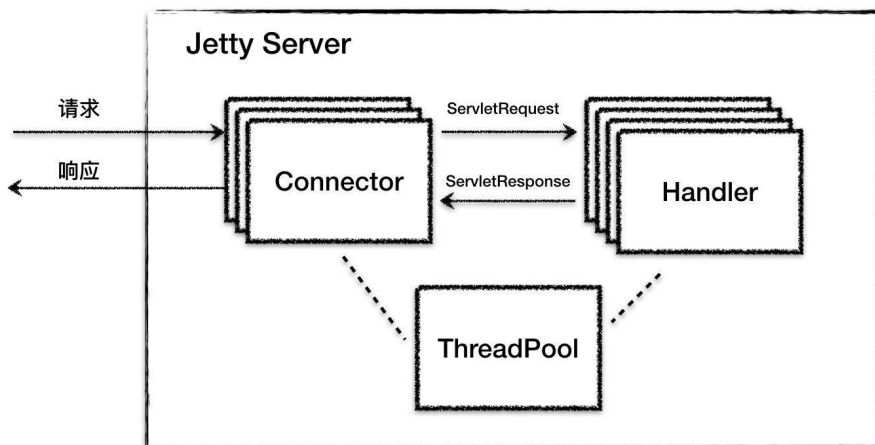


讲述：李号双

时长 10:12 大小 9.35M



我在前面的专栏里介绍了 Jetty 的总体架构设计，简单回顾一下，Jetty 总体上是由一系列 Connector、一系列 Handler 和一个 ThreadPool 组成，它们的关系如下图所示：



相比较 Tomcat 的连接器，Jetty 的 Connector 在设计上有自己的特点。Jetty 的 Connector 支持 NIO 通信模型，我们知道**NIO 模型中的主角就是 Selector**，Jetty 在 Java 原生 Selector 的基础上封装了自己的 Selector，叫作 ManagedSelector。ManagedSelector 在线程策略方面做了大胆尝试，将 I/O 事件的侦测和处理放到同一个线程来处理，充分利用了 CPU 缓存并减少了线程上下文切换的开销。

具体的数字是，根据 Jetty 的官方测试，这种名为“EatWhatYouKill”的线程策略将吞吐量提高了 8 倍。你一定很好奇它是如何实现的吧，今天我们就来看一看这背后的原理是什么。

Selector 编程的一般思路

常规的 NIO 编程思路是，将 I/O 事件的侦测和请求的处理分别用不同的线程处理。具体过程是：

启动一个线程，在一个死循环里不断地调用 select 方法，检测 Channel 的 I/O 状态，一旦 I/O 事件达到，比如数据就绪，就把该 I/O 事件以及一些数据包装成一个 Runnable，将 Runnable 放到新线程中去处理。

在这个过程中按照职责划分，有两个线程在干活，一个是 I/O 事件检测线程，另一个是 I/O 事件处理线程。我们仔细思考一下这两者的关系，其实它们是生产者和消费者的关系。I/O 事件侦测线程作为生产者，负责“生产” I/O 事件，也就是负责接活儿的老板；I/O 处理线程是消费者，它“消费”并处理 I/O 事件，就是干苦力的员工。把这两个工作用不同的线程来处理，好处是它们互不干扰和阻塞对方。

Jetty 中的 Selector 编程


然而世事无绝对，将 I/O 事件检测和业务处理这两种工作分开的思路也有缺点。当 Selector 检测读就绪事件时，数据已经被拷贝到内核中的缓存了，同时 CPU 的缓存中也有这些数据了，我们知道 CPU 本身的缓存比内存快多了，这时当应用程序去读取这些数据时，如果用另一个线程去读，很有可能这个读线程使用另一个 CPU 核，而不是之前那个检

测数据就绪的 CPU 核，这样 CPU 缓存中的数据就用不上了，并且线程切换也需要开销。

因此 Jetty 的 Connector 做了一个大胆尝试，那就是用**把 I/O 事件的生产和消费放到同一个线程来处理**，如果这两个任务由同一个线程来执行，如果执行过程中线程不阻塞，操作系统会用同一个 CPU 核来执行这两个任务，这样就能利用 CPU 缓存了。那具体是如何做的呢，我们还是来详细分析一下 Connector 中的 ManagedSelector 组件。

ManagedSelector

ManagedSelector 的本质就是一个 Selector，负责 I/O 事件的检测和分发。为了方便使用，Jetty 在 Java 原生的 Selector 上做了一些扩展，就变成了 ManagedSelector，我们先来看看它有哪些成员变量：

 复制代码

```
1 public class ManagedSelector extends ContainerLifecycle
2 {
3     // 原子变量，表明当前的 ManagedSelector 是否已经启动
4     private final AtomicBoolean _started = new AtomicBo
5
6     // 表明是否阻塞在 select 调用上
7     private boolean _selecting = false;
8
9     // 管理器的引用，SelectorManager 管理若干 ManagedSele
```


```
10     private final SelectorManager _selectorManager;
11
12     //ManagedSelector 不止一个，为它们每人分配一个 id
13     private final int _id;
14
15     // 关键的执行策略，生产者和消费者是否在同一个线程处理由'
16     private final ExecutionStrategy _strategy;
17
18     //Java 原生的 Selector
19     private Selector _selector;
20
21     //"Selector 更新任务 " 队列
22     private Deque<SelectorUpdate> _updates = new ArrayDeque<>();
23     private Deque<SelectorUpdate> _updateable = new ArrayDeque<>();
24
25     ...
26 }
```

这些成员变量中其他的都好理解，就是“Selector 更新任务”队列 `_updates` 和执行策略 `_strategy` 可能不是很直观。

SelectorUpdate 接口

为什么需要一个“Selector 更新任务”队列呢，对于 Selector 的用户来说，我们对 Selector 的操作无非是将 Channel 注册到 Selector 或者告诉 Selector 我对什么 I/O


事件感兴趣，那么这些操作其实就是对 Selector 状态的更新，Jetty 把这些操作抽象成 SelectorUpdate 接口。

 复制代码

```
1 /**
2  * A selector update to be done when the selector has b
3  */
4 public interface SelectorUpdate
5 {
6     void update(Selector selector);
7 }
```


这意味着如果你不能直接操作 ManageSelector 中的 Selector，而是需要向 ManagedSelector 提交一个任务类，这个类需要实现 SelectorUpdate 接口 update 方法，在 update 方法里定义你想要对 ManagedSelector 做的操作。

比如 Connector 中 Endpoint 组件对读就绪事件感兴趣，它就向 ManagedSelector 提交了一个内部任务类 ManagedSelector.SelectorUpdate：

 复制代码

```
1 _selector.submit(_updateKeyAction);
```

这个 `_updateKeyAction` 就是一个 `SelectorUpdate` 实例，它的 `update` 方法实现如下：

 复制代码


```
1 private final ManagedSelector.SelectorUpdate _updateKey
2 {
3     @Override
4     public void update(Selector selector)
5     {
6         // 这里的 updateKey 其实就是调用了 SelectionKey.i
7         updateKey();
8     }
9 };
```

我们看到在 `update` 方法里，调用了 `SelectionKey` 类的 `interestOps` 方法，传入的参数是 `OP_READ`，意思是现在我对这个 `Channel` 上的读就绪事件感兴趣了。

那谁来负责执行这些 `update` 方法呢，答案是 `ManagedSelector` 自己，它在一个死循环里拉取这些 `SelectorUpdate` 任务类逐个执行。

Selectable 接口

那 I/O 事件到达时，ManagedSelector 怎么知道应该调哪个函数来处理呢？其实也是通过一个任务类接口，这个接口就是 Selectable，它返回一个 Runnable，这个 Runnable 其实就是 I/O 事件就绪时相应的处理逻辑。

 复制代码


```
1 public interface Selectable
2 {
3     // 当某一个 Channel 的 I/O 事件就绪后，ManagedSelector
4     Runnable onSelected();
5
6     // 当所有事件处理完了之后 ManagedSelector 会调的回调函数
7     void updateKey();
8 }
```

ManagedSelector 在检测到某个 Channel 上的 I/O 事件就绪时，也就是说这个 Channel 被选中了，ManagedSelector 调用这个 Channel 所绑定的附件类的 onSelected 方法来拿到一个 Runnable。

这句话有点绕，其实就是 ManagedSelector 的使用者，比如 Endpoint 组件在向 ManagedSelector 注册读就绪事件时，同时也要告诉 ManagedSelector 在事件就绪时执行什么任务，具体来说就是传入一个附件类，这个附件类需要实现 Selectable 接口。ManagedSelector 通过调用这个

onSelected 拿到一个 Runnable，然后把 Runnable 扔给线程池去执行。

那 Endpoint 的 onSelected 是如何实现的呢？


 复制代码

```
1 @Override
2 public Runnable onSelected()
3 {
4     int readyOps = _key.readyOps();
5
6     boolean fillable = (readyOps & SelectionKey.OP_READ
7     boolean flushable = (readyOps & SelectionKey.OP_WRITE
8
9     // return task to complete the job
10    Runnable task= fillable
11        ? (flushable
12            ? _runCompleteWriteFillable
13            : _runFillable)
14        : (flushable
15            ? _runCompleteWrite
16            : null);
17
18    return task;
19 }
```

上面的代码逻辑很简单，就是读事件到了就读，写事件到了就写。

ExecutionStrategy

铺垫了这么多，终于要上主菜了。前面我主要介绍了 ManagedSelector 的使用者如何跟 ManagedSelector 交互，也就是如何注册 Channel 以及 I/O 事件，提供什么样的处理类来处理 I/O 事件，接下来我们来看看 ManagedSelector 是如何统一管理和维护用户注册的 Channel 集合。再回到今天开始的讨论，ManagedSelector 将 I/O 事件的生产和消费看作是生产者消费者模式，为了充分利用 CPU 缓存，生产和消费尽量放到同一个线程处理，那这是如何实现的呢？Jetty 定义了 ExecutionStrategy 接口：

 复制代码

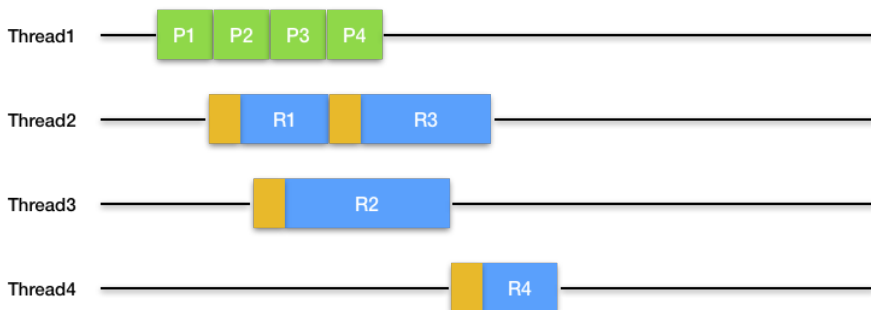
```
1 public interface ExecutionStrategy
2 {
3     // 只在 HTTP2 中用到，简单起见，我们先忽略这个方法。
4     public void dispatch();
5
6     // 实现具体执行策略，任务生产出来后可能由当前线程执行，
7     public void produce();
8
9     // 任务的生产委托给 Producer 内部接口，
10    public interface Producer
11    {
12        // 生产一个 Runnable(任务)
13        Runnable produce();
14    }
15 }
```

我们看到 ExecutionStrategy 接口比较简单，它将具体任务的生产委托内部接口 Producer，而在自己的 produce 方法里来实现具体执行逻辑，**也就是生产出来的任务要么由当前线程执行，要么放到新线程中执行**。Jetty 提供了一些具体策略实现类：ProduceConsume、ProduceExecuteConsume、ExecuteProduceConsume 和 EatWhatYouKill。它们的区别是：

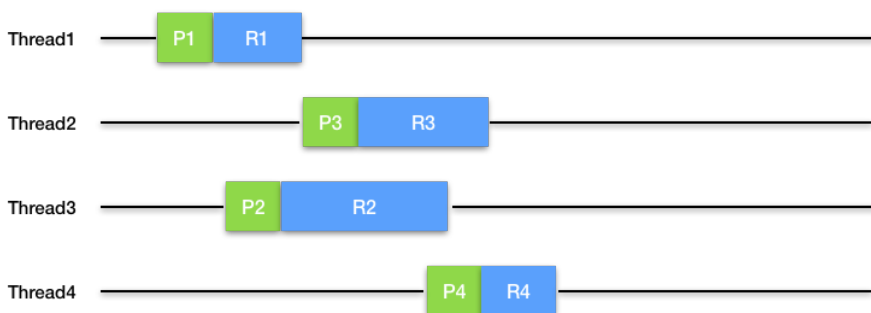
ProduceConsume：任务生产者自己依次生产和执行任务，对应到 NIO 通信模型就是用一个线程来侦测和处理一个 ManagedSelector 上所有的 I/O 事件，后面的 I/O 事件要等待前面的 I/O 事件处理完，效率明显不高。通过图来理解，图中绿色表示生产一个任务，蓝色表示执行这个任务。



ProduceExecuteConsume：任务生产者开启新线程来运行任务，这是典型的 I/O 事件侦测和处理用不同的线程来处理，缺点是不能利用 CPU 缓存，并且线程切换成本高。同样我们通过一张图来理解，图中的棕色表示线程切换。



ExecuteProduceConsume: 任务生产者自己运行任务，但是该策略可能会新建一个新线程以继续生产和执行任务。这种策略也被称为“吃掉你杀的猎物”，它来自狩猎伦理，认为一个人不应该杀死他不吃掉的东西，对应线程来说，不应该生成自己打算运行的任务。它的优点是能利用 CPU 缓存，但是潜在的问题是如果处理 I/O 事件的业务代码执行时间过长，会导致线程大量阻塞和线程饥饿。




EatWhatYouKill: 这是 Jetty 对 ExecuteProduceConsume 策略的改良，在线程池线程

充足的情况下等同于 ExecuteProduceConsume；当系统比较忙线程不够时，切换成

ProduceExecuteConsume 策略。为什么要这么做呢，原因是 ExecuteProduceConsume 是在同一线程执行 I/O 事件的生产和消费，它使用的线程来自 Jetty 全局的线程池，这些线程有可能被业务代码阻塞，如果阻塞得多了，全局线程池中的线程自然就不够用了，最坏的情况是连 I/O 事件的侦测都没有线程可用了，会导致 Connector 拒绝浏览器请求。于是 Jetty 做了一个优化，在低线程情况下，就执行 ProduceExecuteConsume 策略，I/O 侦测用专门的线程处理，I/O 事件的处理扔给线程池处理，其实就是放到线程池的队列里慢慢处理。

分析了这几种线程策略，我们再来看看 Jetty 是如何实现 ExecutionStrategy 接口的。答案其实就是实现 produce 接口生产任务，一旦任务生产出来，ExecutionStrategy 会负责执行这个任务。

 复制代码

```
1 private class SelectorProducer implements ExecutionStra
2 {
3     private Set<SelectionKey> _keys = Collections.empty
4     private Iterator<SelectionKey> _cursor = Collection
5
6     @Override
7     public Runnable produce()
```

```
8      {
9          while (true)
10         {
11             // 如果 Channel 集合中有 I/O 事件就绪，调用前
12             Runnable task = processSelected();
13             if (task != null)
14                 return task;
15
16             // 如果没有 I/O 事件就绪，就干点杂活，看看有没有
17             processUpdates();
18             updateKeys();
19
20             // 继续执行 select 方法，侦测 I/O 就绪事件
21             if (!select())
22                 return null;
23         }
24     }
25 }
```

SelectorProducer 是 ManagedSelector 的内部类，SelectorProducer 实现了 ExecutionStrategy 中的 Producer 接口中的 produce 方法，需要向 ExecutionStrategy 返回一个 Runnable。在这个方法里 SelectorProducer 主要干了三件事情

1. 如果 Channel 集合中有 I/O 事件就绪，调用前面提到的 Selectable 接口获取 Runnable，直接返回给 ExecutionStrategy 去处理。

2. 如果没有 I/O 事件就绪，就干点杂活，看看有没有客户提交了更新 Selector 上事件注册的任务，也就是上面提到的 SelectorUpdate 任务类。
3. 干完杂活继续执行 select 方法，侦测 I/O 就绪事件。

本期精华

多线程虽然是提高并发的法宝，但并不是说线程越多越好，CPU 缓存以及线程上下文切换的开销也是需要考虑的。Jetty 巧妙设计了 EatWhatYouKill 的线程策略，尽量用同一个线程侦测 I/O 事件和处理 I/O 事件，充分利用了 CPU 缓存，并减少了线程切换的开销。

课后思考

文章提到 ManagedSelector 的使用者不能直接向它注册 I/O 事件，而是需要向 ManagedSelector 提交一个 SelectorUpdate 事件，ManagedSelector 将这些事件 Queue 起来由自己来统一处理，这样做有什么好处呢？

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 18 | 新特性：Tomcat如何支持WebSocket?

精选留言 (8)

写留言



nightmare

2019-06-22

第一selectorUpdate可以统一抽象封装注册的io事件，坐到面相抽象编程，将来如果nio的api接口有变动，也不需要改动ManangerSelector的代码 只需要新建一下 selectorUpdate的子类来实现变更 第二 作为缓冲，，如

果高并发的话，一下子很多channel注册到linux的epoll...



👍 3



QQ怪

2019-06-23

课后问题就有点像分布式服务为什么爱用消息中间件一样，一切都是为了解耦，服务类比线程，服务与服务可以直接通讯，线程与线程也可以直接通讯，服务有时候会比较忙或者挂掉了，会导致该请求消息丢失，线程与线程之间的上下文切换同样会带来很大的性能消耗，如果此时...



👍 1



magicnum

2019-06-22

不直接注册I/O事件是为了解耦吧，而且队列平衡了生产者消费者的处理能力



👍 1



-W.LI-

2019-06-24

李老师!SelectorUpdate接口的update方法有个入参完了。硬是没看见哪用了这个入参。泪流满面老师。





-W.LI-

2019-06-24

李老师好!越看越不明白了, 我承认我基础差也没去看源码。不晓得别的同学看不看的懂。

老师能不能画点图方便理解。managedSelector怎么工作的还是不清楚。

只看懂了四种生成消费模型...



-W.LI-

2019-06-24

老师好!我有个问题, 我是先把学校教的计算机基础的书操作系统, 计算机组成原理这些再看一遍补基础。还是直接上手撸源码啊?



-W.LI-

2019-06-24

老实好!

检测到就绪的时候数据已被拷贝到了内核缓存中。CPU的缓存中也有这些数据。

这句话怎么理解啊, CPU的缓存说的是高速缓存么?然后内核缓存是什么呢? 这方面的知识需要看啥书补啊(是操...





802.11

2019-06-23

老师，在NIO2中服务器端的几个概念不是很清晰，比如WindowsAsynchronousChannelProvider, Invoker等。如果需要系统了解这几个概念是怎么串起来的，怎么办呢。

