Reactor 模式详解

http://www.blogjava.net/DLevin/archive/2015/09/02/427045.html

前记

第一次听到 Reactor 模式是三年前的某个晚上，一个室友突然跑过来问我什么是 Reactor 模式？我上网查了一下，很多人都是给出 NIO 中的 Selector 的例子，而且就是 NIO 里 Selector 多路复用模型，只是给它起了一个比较 fancy 的名字而已，虽然它引入了 EventLoop 概 念，这对我来说是新的概念，但是代码实现却是一样的，因而我并没有很在意这个模式。然而最近开始读 Netty 源码，而 Reactor 模式是很多介绍 Netty 的文章中被大肆宣传的模式，因而我再次问自己，什么是 Reactor 模式？本文就是对这个问题关于我的一些理解和尝试着来解答。

什么是 Reactor 模式

要回答这个问题，首先当然是求助 Google 或 Wikipedia，其中 Wikipedia 上说：“The reactor design pattern is an event handling pattern for handling service requests delivered concurrently by one or more inputs. The service handler then demultiplexes the incoming requests and dispatches them synchronously to associated request handlers.”。从这个描述中，我们知道 Reactor 模式首先是**事件驱动的，有一个或多个并发输入源，有一个 Service Handler，有多个 Request Handlers**；这个 Service Handler 会同步的将输入的请求（Event）多路复用的分发给相应的 Request Handler。如果用图来表达：

从结构上，这有点类似生产者消费者模式，即有一个或多个生产者将事件放入一个 Queue 中，而一个或多个消费者主动的从这个 Queue 中 Poll 事件来处理；而 Reactor 模式则并没有 Queue 来做缓冲，每当一个 Event 输入到 Service Handler 之后，该 Service Handler 会主动的根据不同的 Event 类型将其分发给对应的 Request Handler 来处理。

更学术的，这篇文章（Reactor An Object Behavioral Pattern for Demultiplexing and Dispatching Handles for Synchronous Events）上说：“The Reactor design pattern handles service requests that are delivered concurrently to an application by one or more clients. Each service in an application may consistent of several methods and is represented by a separate event handler that is responsible for dispatching service-specific requests. Dispatching of event handlers is performed by an initiation dispatcher, which manages the registered event handlers. Demultiplexing of service requests is performed by a synchronous event demultiplexer. Also known as **Dispatcher, Notifier**”。这段描述和 Wikipedia 上的描述类似，有多个输入源，有多个不同的 EventHandler（RequestHandler）来处理不同的请求，Initiation Dispatcher 用于管理 EventHander，EventHandler 首先要注册到 Initiation Dispatcher 中，然后 Initiation Dispatcher 根据输入的 Event 分发给注册的 EventHandler；然而 Initiation Dispatcher 并不监听 Event 的到来，这个工作交给 Synchronous Event Demultiplexer 来处理。

Reactor 模式结构

在解决了什么是 Reactor 模式后，我们来看看 Reactor 模式是由什么模块构成。图是一种比较简洁形象的表现方式，因而先上一张图来表达各个模块的名称和他们之间的关系：

Handle：即操作系统中的句柄，是对资源在操作系统层面上的一种抽象，它可以是打开的文件、一个连接(Socket)、Timer 等。由于 Reactor 模式一般使用在网络编程中，因而这里一般指 Socket Handle，即一个网络连接（Connection，在 Java NIO 中的 Channel）。这个 Channel 注册到 Synchronous Event Demultiplexer 中，以监听 Handle 中发生的事件，对 ServerSocketChannnel 可以是 CONNECT 事件，对 SocketChannel 可以是 READ、WRITE、CLOSE 事件等。

Synchronous Event Demultiplexer：阻塞等待一系列的 Handle 中的事件到来，如果阻塞等待返回，即表示在返回的 Handle 中可以不阻塞的执行返回的事件类型。这个模块一般使用操作系统的 select 来实现。在 Java NIO 中用 Selector 来封装，当 Selector.select()返回时，可以调用 Selector 的 selectedKeys()方法获取 Set<SelectionKey>，一个 SelectionKey 表达一个有事件发生的 Channel 以及该 Channel 上的事件类型。上图的“Synchronous Event Demultiplexer ---notifies--> Handle”的流程如果是对的，那内部实现应该是 select()方法在事件到来后会先设置 Handle 的状态，然后返回。不了解内部实现机制，因而保留原图。

Initiation Dispatcher：用于管理 Event Handler，即 EventHandler 的容器，用以注册、移除 EventHandler 等；另外，它还作为 Reactor 模式的入口调用 Synchronous Event Demultiplexer 的

select 方法以阻塞等待事件返回，当阻塞等待返回时，根据事件发生的 Handle 将其分发给对应的 Event Handler 处理，即回调 EventHandler 中的 handle_event()方法。
Event Handler：定义事件处理方法：handle_event()，以供 InitiationDispatcher 回调使用。
Concrete Event Handler：事件 EventHandler 接口，实现特定事件处理逻辑。

Reactor 模式模块之间的交互

简单描述一下 Reactor 各个模块之间的交互流程，先从序列图开始：

1. 初始化 InitiationDispatcher，并初始化一个 Handle 到 EventHandler 的 Map。
2. 注册 EventHandler 到 InitiationDispatcher 中，每个 EventHandler 包含对相应 Handle 的引用，从而建立 Handle 到 EventHandler 的映射（Map）。
3. 调用 InitiationDispatcher 的 handle_events()方法以启动 Event Loop。在 Event Loop 中，调用 select()方法（Synchronous Event Demultiplexer）阻塞等待 Event 发生。
4. 当某个或某些 Handle 的 Event 发生后，select()方法返回，InitiationDispatcher 根据返回的 Handle 找到注册的 EventHandler，并回调该 EventHandler 的 handle_events()方法。
5. 在 EventHandler 的 handle_events()方法中还可以向 InitiationDispatcher 中注册新的 Eventhandler，比如对 AcceptorEventHandler 来，当有新的 client 连接时，它会产生新的 EventHandler 以处理新的连接，并注册到 InitiationDispatcher 中。

Reactor 模式实现

在 Reactor An Object Behavioral Pattern for Demultiplexing and Dispatching Handles for Synchronous Events 中，一直以 Logging Server 来分析 Reactor 模式，这个 Logging Server 的实现完全遵循这里对 Reactor 描述，因而放在这里以做参考。Logging Server 中的 Reactor 模式实现分两个部分：Client 连接到 Logging Server 和 Client 向 Logging Server 写 Log。因而对它的描述分成这两个步骤。
Client 连接到 Logging Server

1. Logging Server 注册 LoggingAcceptor 到 InitiationDispatcher。
2. Logging Server 调用 InitiationDispatcher 的 handle_events()方法启动。
3. InitiationDispatcher 内部调用 select()方法（Synchronous Event Demultiplexer），阻塞等待 Client 连接。
4. Client 连接到 Logging Server。
5. InitiationDisptcher 中的 select()方法返回，并通知 LoggingAcceptor 有新的连接到来。
6. LoggingAcceptor 调用 accept 方法 accept 这个新连接。
7. LoggingAcceptor 创建新的 LoggingHandler。
8. 新的 LoggingHandler 注册到 InitiationDispatcher 中(同时也注册到 Synchonous Event Demultiplexer 中)，等待 Client 发起写 log 请求。

Client 向 Logging Server 写 Log

1. Client 发送 log 到 Logging server。
2. InitiationDispatcher 监测到相应的 Handle 中有事件发生，返回阻塞等待，根据返回的 Handle 找到 LoggingHandler，并回调 LoggingHandler 中的 handle_event()方法。
3. LoggingHandler 中的 handle_event()方法中读取 Handle 中的 log 信息。
4. 将接收到的 log 写入到日志文件、数据库等设备中。

3.4 步骤循环直到当前日志处理完成。

5. 返回到 InitiationDispatcher 等待下一次日志写请求。

在 Reactor An Object Behavioral Pattern for Demultiplexing and Dispatching Handles for Synchronous Events 有对 Reactor 模式的 C++的实现版本，多年不用 C++，因而略过。

Java NIO 对 Reactor 的实现

在 Java 的 NIO 中，对 Reactor 模式有无缝的支持，即使用 Selector 类封装了操作系统提供的 Synchronous Event Demultiplexer 功能。这个 Doug Lea 已经在 Scalable IO In Java 中有非常深入的解释了，因而不再赘述，另外这篇文章对 Doug Lea 的 Scalable IO In Java 有一些简单解释，至少它的代码格式比 Doug Lea 的 PPT 要整洁一些。

需要指出的是，不同这里使用 InitiationDispatcher 来管理 EventHandler，在 Doug Lea 的版本中使用 SelectionKey 中的 Attachment 来存储对应的 EventHandler，因而不需要注册 EventHandler 这个步骤，或者设置 Attachment 就是这里的注册。而且在这篇文章中，Doug Lea 从单线程的 Reactor、Acceptor、Handler 实现这个模式出发；演化为将 Handler 中的处理逻辑多线程化，实现类似 Proactor 模式，此时所有的 IO 操作还是单线程的，因而再演化出一个 Main Reactor 来处理 CONNECT 事件(Acceptor)，而多个 Sub Reactor 来处理 READ、WRITE 等事件(Handler)，这些 Sub Reactor 可以分别再自己的线程中执行，从而 IO 操作也多线程化。这个最后一个模型正是 Netty 中使用的模型。并且在 Reactor An Object Behavioral Pattern for Demultiplexing and Dispatching Handles for Synchronous Events 的 9.5 Determine the Number of Initiation Dispatchers in an Application 中也有相应的描述。

EventHandler 接口定义

对 EventHandler 的定义有两种设计思路：single-method 设计和 multi-method 设计：
A single-method interface：它将 Event 封装成一个 Event Object，EventHandler 只定义一个 handle_event(Event event)方法。这种设计的好处是有利于扩展，可以后来方便的添加新的 Event 类型，然而在子类的实现中，需要判断不同的 Event 类型而再次扩展成 不同的处理方法，从这个角度上来说，它又不利于扩展。另外在 Netty3 的使用过程中，由于它不停的创建 ChannelEvent 类，因而会引起 GC 的不稳定。
A multi-method interface：这种设计是将不同的 Event 类型在 EventHandler 中定义相应的方法。这种设计就是 Netty4 中使用的策略，其中一个目的是避免 ChannelEvent 创建引起的 GC 不稳定，另外一个好处是它可以避免在 EventHandler 实现时判断不同的 Event 类型而有不同的实现，然而这种设计会给扩展新的 Event 类型时带来非常 大的麻烦，因为它需要该接口。

关于 Netty4 对 Netty3 的改进可以参考这里：
ChannelHandler with no event object In 3.x, every I/O operation created a ChannelEvent object. For each read / write, it additionally created a new ChannelBuffer. It simplified the internals of Netty quite a lot because it delegates resource management and buffer pooling to the JVM. However, it often was the root cause of GC pressure and uncertainty which are sometimes observed in a Netty-based application under high load.

4.0 removes event object creation almost completely by replacing the event objects with strongly typed method invocations. 3.x had catch-all event handler methods such

as handleUpstream() and handleDownstream(), but this is not the case anymore. Every event type has its own handler method now:

为什么使用 Reactor 模式

归功与 Netty 和 Java NIO 对 Reactor 的宣传，本文慕名而学习的 Reactor 模式，因而已经默认 Reactor 具有非常优秀的性能，然而慕名归慕名，到这里，我还是要不得不问自己 Reactor 模式的好处在哪里？即为什么要使用这个 Reactor 模式？在 Reactor An Object Behavioral Pattern for Demultiplexing and Dispatching Handles for Synchronous Events 中是这么说的：

Reactor Pattern 优点

Separation of concerns: The Reactor pattern decouples application-independent demultiplexing and dispatching mechanisms from application-specific hook method functionality. The application-independent mechanisms become reusable components that know how to demultiplex events and dispatch the appropriate hook methods defined by `Event Handlers`. In contrast, the application-specific functionality in a hook method knows how to perform a particular type of service.

Improve modularity, reusability, and configurability of event-driven applications: The pattern decouples application functionality into separate classes. For instance, there are two separate classes in the logging server: one for establishing connections and another for receiving and processing logging records. This decoupling enables the reuse of the connection establishment class for different types of connection-oriented services (such as file transfer, remote login, and video-on-demand). Therefore, modifying or extending the functionality of the logging server only affects the implementation of the logging handler class.

Improves application portability: The `Initiation Dispatcher`'s interface can be reused independently of the OS system calls that perform event demultiplexing. These system calls detect and report the occurrence of one or more events that may occur simultaneously on multiple sources of events. Common sources of events may in- clude I/O handles, timers, and synchronization objects. On UNIX platforms, the event demultiplexing system calls are called `select` and `poll` [1]. In the Win32 API [16], the `WaitForMultipleObjects` system call performs event demultiplexing.

Provides coarse-grained concurrency control: The Reactor pattern serializes the invocation of event handlers at the level of event demultiplexing and dispatching within a process or thread. Serialization at the `Initiation Dispatcher` level often eliminates the need for more complicated synchronization or locking within an application process.

这些貌似是很多模式的共性：解耦、提升复用性、模块化、可移植性、事件驱动、细力度的并发控制等，因而并不能很好的说明什么，特别是它鼓吹的对性能的提升，这里并没有体现出来。当然在这篇文章的开头有描述过另一种直观的实现：Thread-Per-Connection，即传统的实现，提到了这个传统实现的以下问题：

Thread Per Connection 缺点

Efficiency: Threading may lead to poor performance due to context switching, synchronization, and data movement [2];

Programming simplicity: Threading may require complex concurrency control schemes;

Portability: Threading is not available on all OS platforms.对于性能，它其实就是第一点关于 Efficiency 的描述，即线程的切换、同步、数据的移动会引起性能问题。也就是说从性能的角度上，它最大的提升就是减少了性能的使用，即不需要每个 Client 对应一个线程。我的理解，其他业务逻辑处理很多时候也会用到相同的线程，IO 读写操作相对 CPU 的操作还是要慢很多，即使 Reactor 机制中每次读写已经能保证非阻塞读写，这里可以减少一些线程的使用，但是这减少的线程使用对性能有那么大的影响吗？答案貌似是肯定的，这篇论文(SEDA: Staged Event-Driven Architecture - An Architecture for Well-Conditioned, Scalable Internet Service)对随着线程的增长带来性能降低做了一个统计：

在这个统计中，每个线程从磁盘中读 8KB 数据，每个线程读同一个文件，因而数据本身是缓存在操作系统内部的，即减少 IO 的影响；所有线程是事先分配的，不会有线程启动的影响；所有任务在测试内部产生，因而不会有网络的影响。该统计数据运行环境：Linux 2.2.14，2GB 内存，4-way 500MHz Pentium III。从图中可以看出，随着线程的增长，吞吐量在线程数为 8 个左右的时候开始线性下降，并且到 64 个以后而迅速下降，其相应事件也在线程达到 256 个后指数上升。即 1+1<2，因为线程切换、同步、数据移动会有性能损失，线程数增加到一定数量时，这种性能影响效果会更加明显。

对于这点，还可以参考 C10K Problem，用以描述同时有 10K 个 Client 发起连接的问题，到 2010 年

的时候已经出现 10M Problem 了。

当然也有人说：Threads are expensive are no longer valid.在不久的将来可能又会发生不同的变化，或者这个变化正在、已经发生着？没有做过比较仔细的测试，因而不敢随便断言什么，然而本人观点，即使线程变的影响并没有以前那么大，使用 Reactor 模式，甚至时 SEDA 模式来减少线程的使用，再加上其他解耦、模块化、提升复用性等优点，还是值得使用的。

Reactor 模式的缺点

Reactor 模式的缺点貌似也是显而易见的：
1. 相比传统的简单模型，Reactor 增加了一定的复杂性，因而有一定的门槛，并且不易于调试。
2. Reactor 模式需要底层的 Synchronous Event Demultiplexer 支持，比如 Java 中的 Selector 支持，操作系统的 select 系统调用支持，如果要自己实现 Synchronous Event Demultiplexer 可能不会有那么高效。
3. Reactor 模式在 IO 读写数据时还是在同一个线程中实现的，即使使用多个 Reactor 机制的情况下，那些共享一个 Reactor 的 Channel 如果出现一个长时间的数据读写，会影响这个 Reactor 中其他 Channel 的相应时间，比如在大文件传输时，IO 操作就会影响其他 Client 的相应时间，因而对这种操作，使用传统的 Thread-Per-Connection 或许是一个更好的选择，或则此时使用 Proactor 模式。

参考

Reactor Pattern WikiPedia
Reactor An Object Behavioral Pattern for Demultiplexing and Dispatching Handles for Synchronous Events
Scalable IO In Java
C10K Problem WikiPedia

posted on 2015-09-02 15:14 DLevin 阅读(5374) 评论(4) 编辑 收藏 所属分类: Architecture

.

FeedBack:

re: Reactor 模式详解

2015-09-11 16:29 | 李强强

拜读 Jetty 容器源码 ing~ 回复 更多评论

re: Reactor 模式详解[未登录]

2015-09-24 23:42 | Rick

从原理上看，SocketServer 应该就是此类设计。 回复 更多评论

re: Reactor 模式详解

2016-03-28 17:00 | 李华峰

"它其实就是第一点关于 Efficiency 的描述，即线程的切换、同步、数据的移动会引起性能问题"，作者这段话是本篇文章的题眼，因为 Jakob 在博客里说过"它适用于连接数很多但每个连接的流量很小(处理时间很短)的情况"，两种表述一个思想.

另外, reactor pattern 和 observer pattern 两种模式类似，区别在于前者与多个事件源关联，后者与多个事件源关联. 这点区别，又反过来印证了上述思想, reactor pattern 和多个事件源关联，每个事件的处理时间很短，所以，大家复用线程，避免线程切换/同步/数据移动带来的性能问题; observer pattern 和单个事件源关联，不同事件的处理时间不一致，各类事件独享线程，避免处理时间长的影响处理时间短的响应速度.

回复 更多评论

re: Reactor 模式详解

2016-04-08 16:35 | 无

@李华峰

好 回复 更多评论