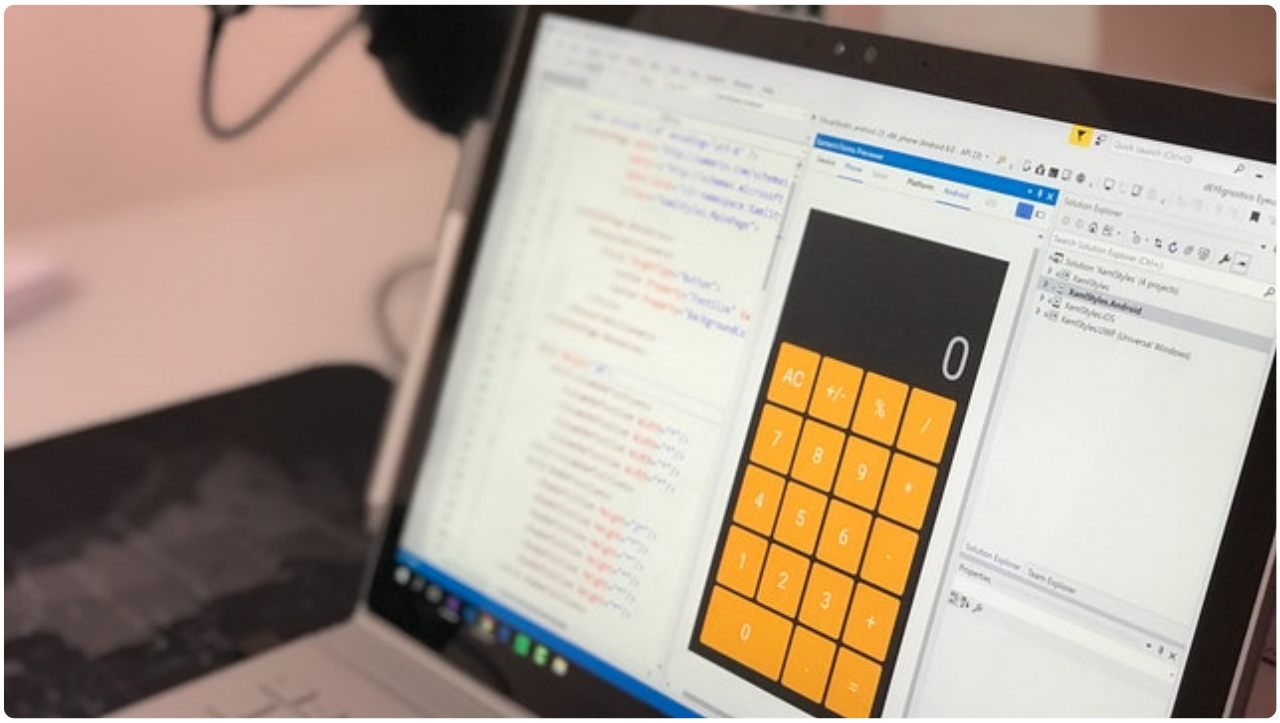
09 什么是Reactor模式

更新时间：2020-07-20 09:45:24



> 如果说我比别人看得要远一点，那是因为我站在巨人的肩上。——牛顿

前言

你好，我是彤哥。

上一节我们一起学习了 Netty 的十大核心组件，通过上一节的学习，相信你一定能从宏观上理解 Netty 的架构设计。

在上一节，学习 EventLoopGroup 的时候，我们说过服务端建议设置两个线程池，一个用于处理 Accept 事件，一个用于处理读写事件，不知道你还有没有印象呢？

本节，我们就来说说为什么需要两个线程池，也就是鼎鼎大名的 Reactor 模式。

好了，让我们进入今天的学习吧。

什么是 Reactor 模式？

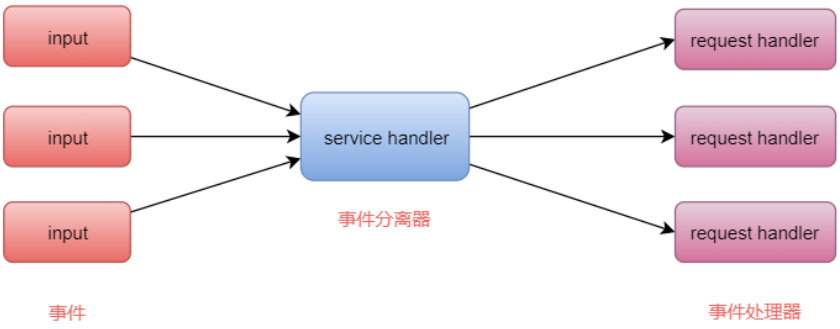
关于 Reactor 模式的定义，让我们看看维基百科怎么说：

> The reactor design pattern is an event handling pattern for handling service requests delivered concurrently to a service handler by one or more inputs. The service handler then demultiplexes the incoming requests and dispatches them synchronously to the associated request handlers.

从这段定义中，我们能得出以下几个信息：

1. Reactor 模式是一种事件处理模式；
2. 用于处理服务请求，把它们并发地传递给一个服务处理器（service handler）；
3. 有一个或多个输入源（inputs）；
4. 服务处理器将这些请求以多路复用的方式分离（demultiplexes ），并把它们同步地分发到相关的请求处理器（request handlers）；

总结一下，Reactor 模式包含一个或多个输入源，一个 service handler，多个 request handler，service handler 是输入源和 request handler 之间的桥梁，用于分发输入的请求，我画了一个图来表示：



为了方便描述，本节，我们把 input 叫作事件，service handler 叫作事件分离器，request handler 叫作事件处理器。

我们举个形象的例子，稍微有点 Javascript 开发经验的同学，应该都写过类似下面的代码：

```
txt.onblur(function() {
    if(!content) {
        alert("请您输入用户名");
    }
});
btn.onclick(function() {
    alert("登录成功");
});
```

没有过 Javascript 开发经验的同学也没关系，我稍微解释一下，比如登录页面，用户名输入框失去焦点时如果为空则提示 “请您输入用户名”，点击按钮的时候弹出对话框显示 “登录成功”，类似于下面这张图：

这就是典型的事件驱动模型，事件即网页上的各种事件，比如按钮点击事件、失去焦点事件、鼠标右击事件等等，事件处理器即我们编写的回调函数，即上面代码中括号中的 function，事件分离器即 Javascript 内部根据不同的事件分发到不同的回调的处理器。

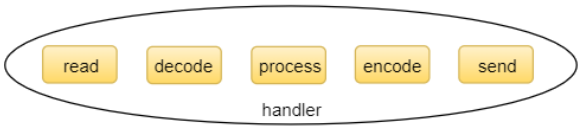
Reactor 的几种模式

上面的定义可能理解起来比较模糊，让我们再看看大神 Doug Lea 是怎么把 Reactor 模式带到 Java 中的。[1]

网络请求的处理过程

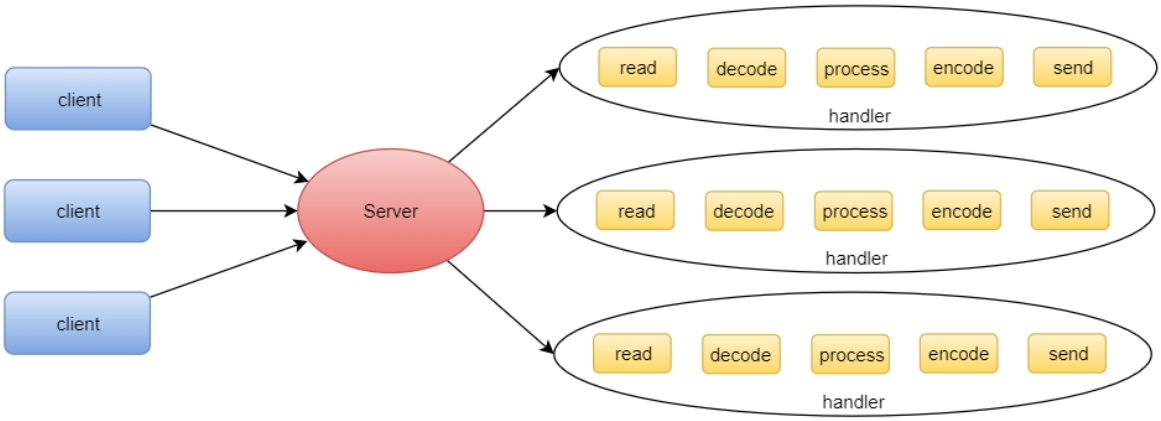
一般地，网络请求都要经过以下几个处理过程：

- Read request，读取请求
- Decode request，解码请求
- Process service，处理业务
- Encode reply，编码响应



传统的服务设计

基于以上处理过程，传统的服务是如何设计的呢？



每次来一个客户端都启动一个新的线程来处理，每一个 handler 都在它自己的线程中。是不是很像我们前面介绍 BIO 的模型？没错，它们是相通的。

使用这种服务设计自然有它的优点：

1. 编码简单；
2. 每一个 handler 都在自己的线程中，不存在线程切换的问题，不需要考虑线程安全的问题；

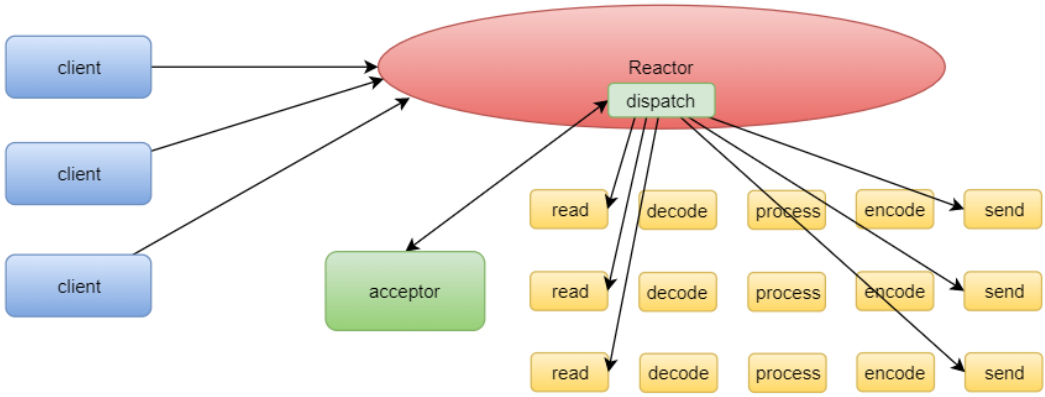
但是，随着服务请求量越来越大，启动的线程数量会越来越多，最后，会导致服务端的线程无限增多，然而，其实大部分的线程可能都处于 IO 阻塞状态，并没有使用到 CPU，无法充分利用 CPU。

那么，怎么改进呢？

采用基于事件驱动的设计，当有事件触发时，才调用相应的处理器来处理事件。

Reactor 单线程模式

Reactor 单线程模式应运而生，使用一个线程就可以处理大量的事件。



Reactor 单线程模式，就像一个饭店只有老板一个人一样，既要负责接待客人，又要当厨师，又要当服务员，一个人干所有的事，效率势必非常低下。

在服务端，对于网络请求有三种不同的事件：Accept 事件、Read 事件、Write 事件，对应于上图中的 acceptor、read、send。

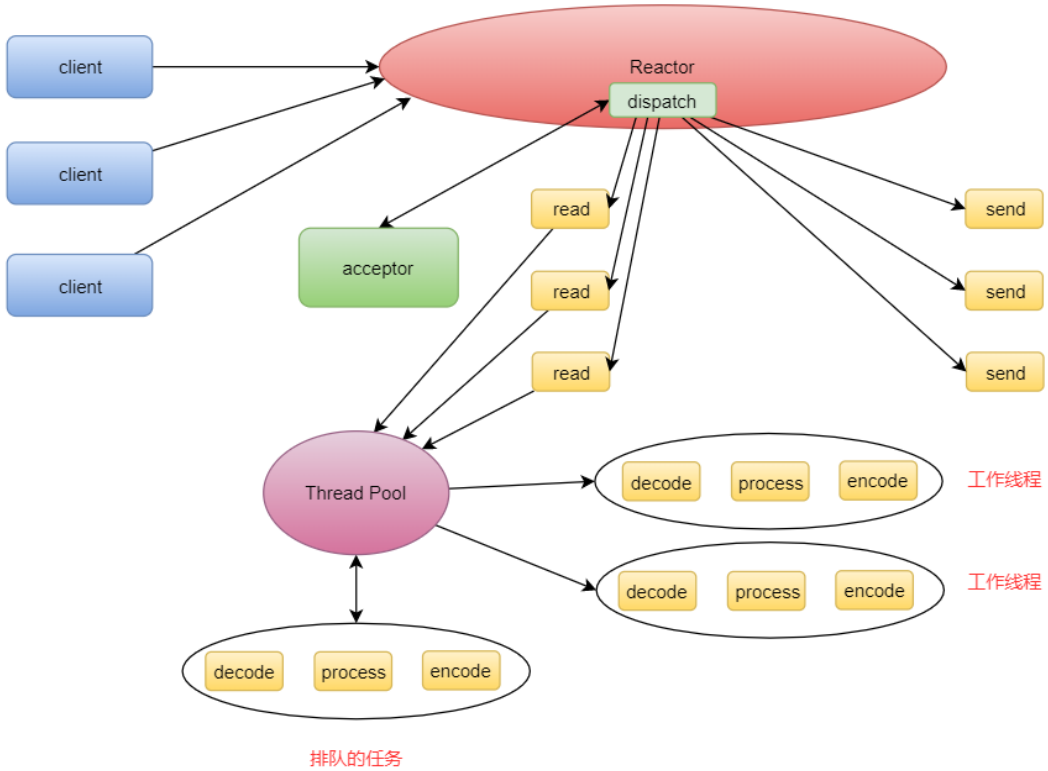
> Connect 事件属于客户端事件。
>
> 为什么 acceptor（Accept 事件处理器）是双向箭头，而 read 和 send 是单向箭头呢？因为服务端启动的时候是先注册 Accept 事件到 Reactor 上，当收到客户端连接时，也就是 Accept 事件时，才会注册 Read 和 Write 事件，所以 acceptor 是双向的，Reactor 不仅要向 acceptor 分发 Accept 事件，acceptor 也要向 Reactor 注册 Read 和 Write 事件。

一个 Reactor 就相当于一个事件分离器，而单线程模式下，所有客户端的所有事件都在一个线程中完成，这就出现了一个新的问题，如果哪个请求有阻塞，直接影响了所有请求的处理速度，所以，自然而然就进化出了 Reactor 的多线程模式。

> 早期都是单核 CPU，一个请求阻塞会影响所有请求，注意，是阻塞，而不是处理缓慢，处理缓慢是有大量的计算，这时候即使启动多个线程也无法提高其它请求处理的速度。

Reactor 多线程模式

Reactor 多线程模式，还是把 IO 事件放在 Reactor 线程中来处理，同时，把业务处理逻辑放到单独的线程池中来处理，这个线程池我们称为工作线程池（Worker Thread Pool）或者业务线程池。



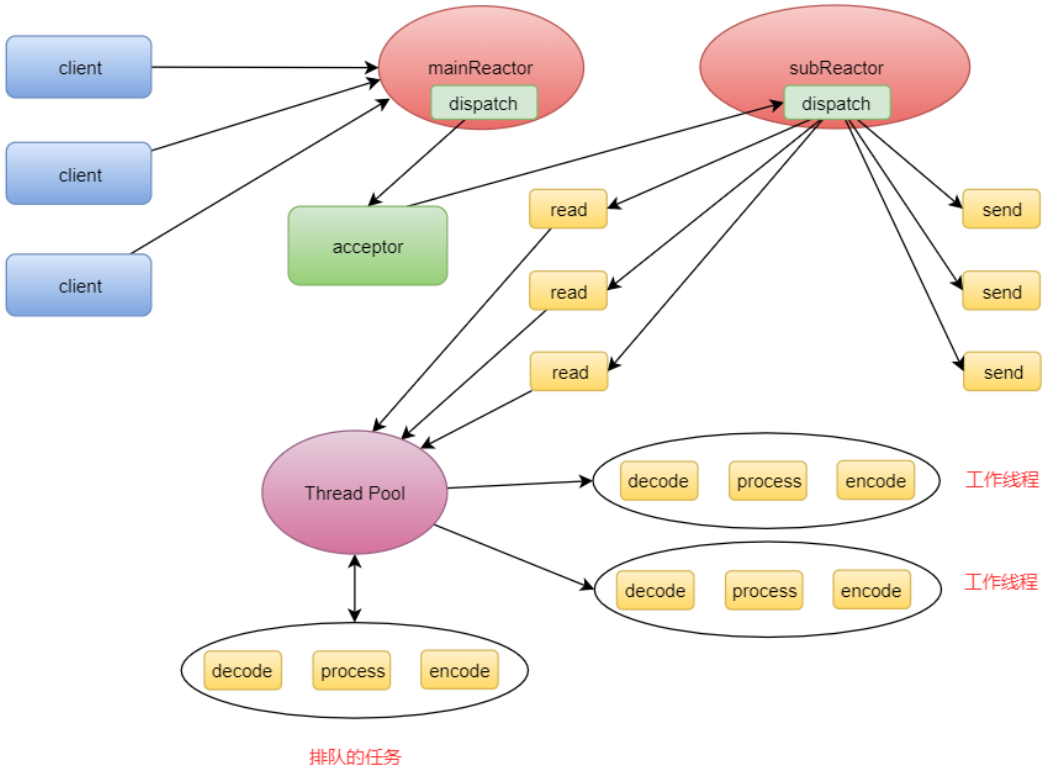
此时，如果业务处理逻辑中有 IO 阻塞，则不会影响其它请求的处理，能很大程度提高系统的并发量。

Reactor 多线程模式，就像饭店中老板只负责主要事务，比如，接待客人、接收客人的下单请求等，具体的事务交给服务员去处理。

但是，这种模式还不够完美，一个客户端连接过程需要三次握手，是一个比较耗时的操作，将 Accept 事件和 Read 事件与 Write 事件放在一个 Reactor 中来处理，明显降低了 Read 和 Write 事件的响应速度。而且，一个 Reactor 只有一个线程，也无法利用多核 CPU 的性能提升。因此，又自然而然的出现了 Reactor 主从模式。

Reactor 主从模式

Reactor 主从模式把 Accept 事件的处理单独拿出来放到主 Reactor 中来处理，把 Read 和 Write 事件放到子 Reactor 中来处理，而且，像这样的子 Reactor 我们可以启动多个，充分利用多核 CPU 的资源。

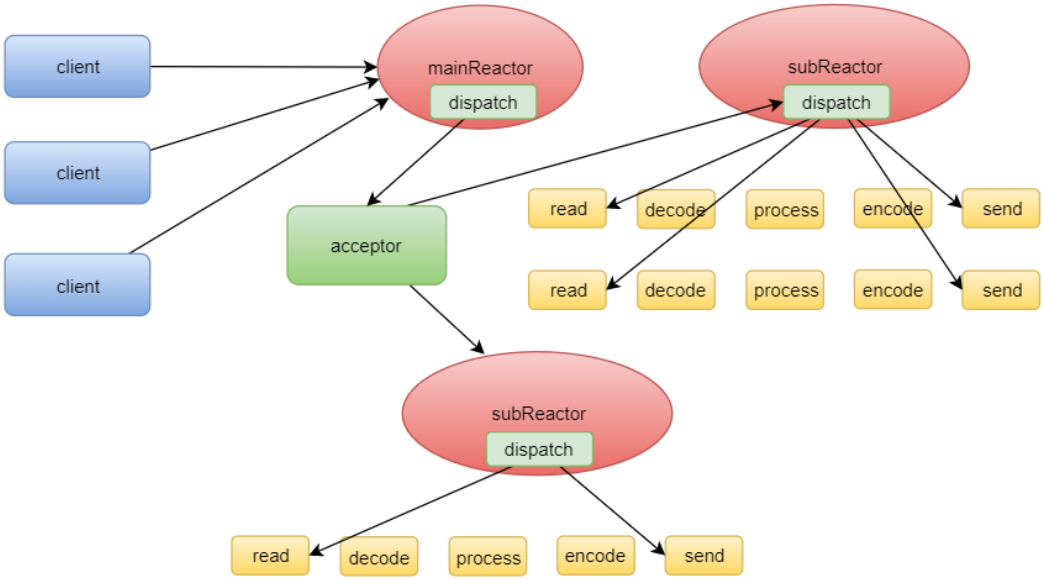


Reactor 主从模式，就像饭店中的老板只负责客人接待这一件事，其它事务全部交给服务员来处理，而且服务员也可以按区域划分，比如 1 号服务员负责 1 到 5 号包厢，2 号服务员负责 6 到 10 号包厢，极大地提高了效率。

在 Reactor 主从模式中，我们依然把业务逻辑的处理放到业务线程池中来处理，但是，既然子 Reactor 本身就可以启动多个，所以，我们直接让子 Reactor 池化，利用子 Reactor 本身的线程来处理业务逻辑，可不可以呢？

变异的 Reactor 模式

基于主从模式可以有很多种变异的模式，比如使用子 Reactor 线程池来处理业务逻辑。



正常情况下，在 Netty 中，我们也是这么使用的，当然，依据不同的业务场景也可以有不同的变异。

如果说，正常的 Reactor 主从模式下，一批服务员负责不同包厢的下单请求（多个子 Reactor），另外一批服务员负责包厢的其它事务，比如上菜、端茶、倒水（业务线程池）。那么，变异的 Reactor 主从模式下，就是一个服务员负责几个包厢的所有事务，不管下单请求，还是上菜、端茶、倒水，另一个服务员再负责另几个包厢的所有事务，海底捞貌似就是这种变异的 Reactor 模式。

好了，Reactor 的几种模式介绍完了，那么，在 Netty 中怎么使用以上几种模式呢？

Netty 中使用 Reactor 的不同模式

Reactor 单线程模式的使用

Reactor 单线程模式，只有一个 Reactor，也就是一个线程处理所有事务，所以，在 Netty 中，只需要声明一个 EventLoopGroup 就可以了。

```
EventLoopGroup bossGroup = new NioEventLoopGroup(1);
ServerBootstrap serverBootstrap = new ServerBootstrap();
serverBootstrap.group(bossGroup);
```

Reactor 多线程模式的使用

Reactor 多线程模式，实际上还是只有一个 Reactor，但是这个 Reactor 只负责处理 IO 事件，而不负责处理业务逻辑，所以，在 Netty 中，需要将业务逻辑的处理，也就是 Handler，放到另外的线程池中。

```
EventLoopGroup bossGroup = new NioEventLoopGroup(1); // 一个Reactor
ServerBootstrap serverBootstrap = new ServerBootstrap();
serverBootstrap.group(bossGroup);
// Handler使用线程池处理
```

Reactor 主从模式的使用

Reactor 主从模式，有一个主 Reactor 和多个子 Reactor，但是，业务逻辑的处理还是在线程池中，所以，在 Netty 中，需要声明两个不同的 EventLoopGroup，Handler 依然使用线程池处理。

```
EventLoopGroup bossGroup = new NioEventLoopGroup(1); // 一个主Reactor
EventLoopGroup workerGroup = new NioEventLoopGroup(); // 多个子Reactor
ServerBootstrap serverBootstrap = new ServerBootstrap();
serverBootstrap.group(bossGroup, workerGroup);
// Handler使用线程池处理
```

Reactor 变异主从模式的使用

Reactor 变异主从模式，业务线程池和子 Reactor 池合并为一，所以，在 Netty 中，Handler 放在子 Reactor 池中处理即可，默认情况，Netty 也是使用的这种模式。

```
EventLoopGroup bossGroup = new NioEventLoopGroup(1); // 一个主Reactor
EventLoopGroup workerGroup = new NioEventLoopGroup(); // 多个子Reactor
ServerBootstrap serverBootstrap = new ServerBootstrap();
serverBootstrap.group(bossGroup, workerGroup);
```

看了这几种模式的使用，你可能会有个疑问：为什么只能有一个主 Reactor 呢？启动多个主 Reactor 可不可以呢？

答案是，可以，但没必要，因为底层的 Accept 事件的处理依然要排队处理，具体可以查看源码 `sun.nio.ch.ServerSocketChannelImpl#accept()`：

```
public SocketChannel accept() throws IOException {
    Object var1 = this.lock;
    synchronized(this.lock) {
        // 省略具体代码
    }
}
```

可以看到，accept () 方法中使用了一个 synchronized 锁来控制同时只能处理一个客户端的连接请求，使用一个线程来处理，相应地，还能减少线程的切换，提高一定的性能，有兴趣的同学，可以去查查 synchronized 的偏向锁、轻量级锁、重量级锁相关的内容。

Reactor 模式的优点和缺点

好了，Reactor 的几种模式介绍完了，但是，Reactor 并不是一剂万能药，所以我们有必要了解它的的优点和缺点，综合对比，我们才能决定要不要使用它。

首先，我们来看看它的优点，也是 Reactor 的主要卖点：

1. 能够解耦模块，将 IO 操作与业务逻辑解耦；
2. 能够提高并发量，充分利用 CPU 资源；
3. 可扩展性好，简单地增加子 Reactor 的数量就能很好地扩展；
4. 可复用性好，Reactor 框架本身不与具体的业务逻辑挂钩，复用性好；

等等。

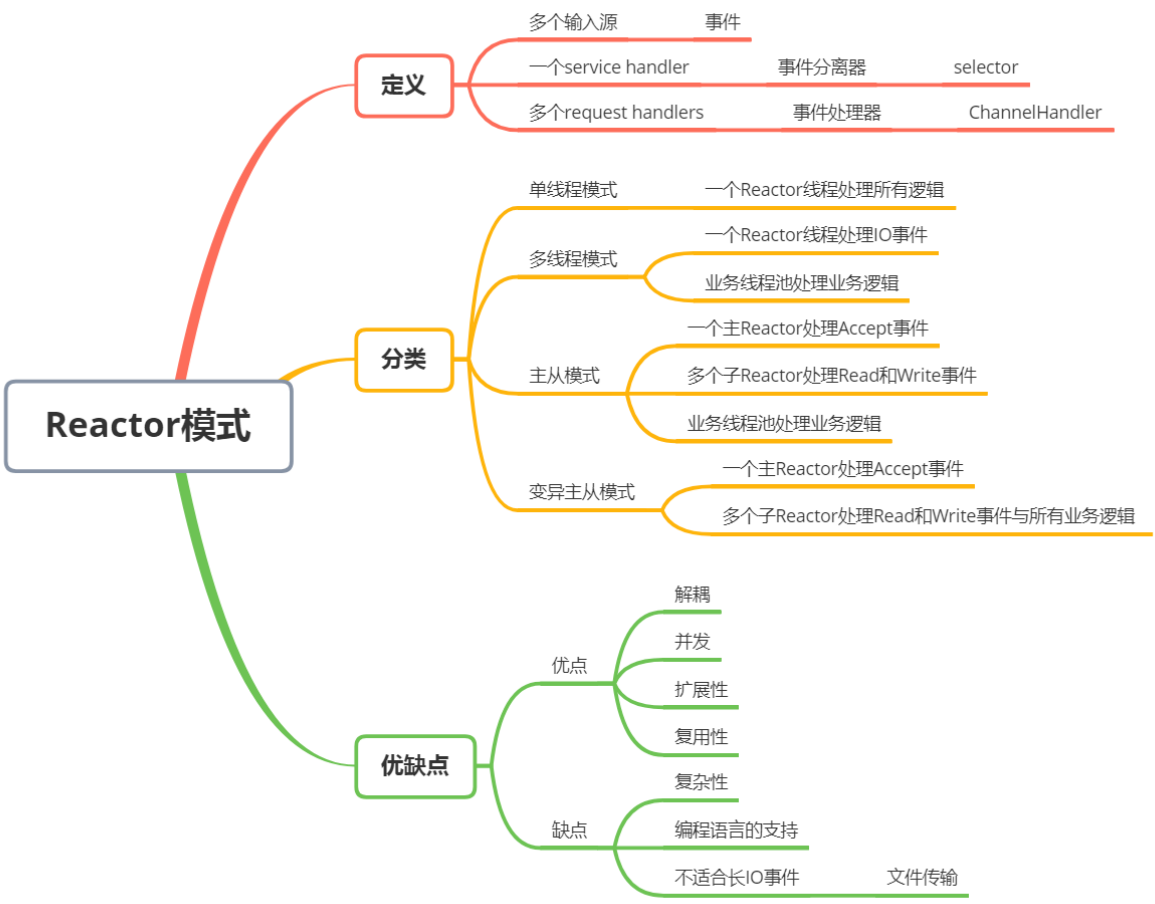
然而 ，同样地，它也有一些缺点

1. 相比于传统的简单模式，Reactor 增加了一定复杂度，增加了学习成本、试错成本和调试成本；
2. 需要编程语言支持事件分离器，比如 Java 中的 Selector，如果自己实现不现实；
3. 多个客户端共用同一个 Reactor，如果有文件传输这种耗时的 IO 操作， 不适合使用 Reactor 模式；

后记

本节，我们一起学习了 Reactor 的几种模式以及它们在 Netty 中的使用，总结下来，我们在 Netty 中，一般使用变异的主从模式就够了，除非有比较耗时的 IO 阻塞，我们才需要使用主从模式那种更复杂的情形。Netty 本身默认使用的也是这种变异的主从模式。

思维导图



参考

1. Doug Lea 大神关于 Reactor 在 Java NIO 中运用的讲述：http://gee.cs.oswego.edu/dl/cpjslides/nio.pdf

}

08 Netty的核心组件有哪些

10 如何解决粘包半包问题