响应式编程必知必看：Reactor 和 Actor 模型

在我之前文章——这，就是响应式编程中，看到大家对响应式编程这个主题比较感兴趣，有很多评论和阅读，所以我决定在五月再继续再更新几篇响应式编程的文章，希望大家多多，多多点赞。

响应式中的概念

响应式编程是一种编程范式，它强调非阻塞和异步操作，这意味着响应式应用程序可以同时处理多个请求，而不会阻塞主线程，这使得它们非常适合处理高负载和实时应用程序。

在真正开始之前，我觉得有必要简单梳理一下几种高性能并发模中的相关概念，不然极易混淆，先来说说几个高性能并发中常用到的设计思想和技术：

1. **线程池**：池化，这个自不必多说，利用多线程加大并发，同时利用池化减少创建线程开销。
2. **任务并行**：利用 MapReduce 的思想进行并行计算，JDK 中的代表是 ForkJoinPool。
3. **管道(Pipeline)**：比较知名的代表是 Go 中的 Channel。
4. **事件驱动**：利用 Reactor 并发模型进行高性能 IO，代表是 Netty、Node、Nginx。
5. **Actor模型**：利用 Actor 并发模型进行完全异步化，代表是 Akka 框架和 Erlang 语言。

上面这几种它们设计思想并不是孤立的，可能一个框架中同时包含多种设计思想进行构建，比如 Vert.X 其实就同时使用了 Reactor + Actor，当然 VertX 并不认为它是完全引入了 Actor 模型，而是把它叫作 Like Actor，也就是说借用了其思想。

接下来我们看看关于响应式编程，Java 领域存在哪些知名的库 or 框架：

1. **RxJava**：用于编写响应式数据流风格代码的库，可以让你使用响应式风格来编码，它只是响应式操作的库，并不自带其他诸如 http 请求之类的功能。
2. **Project Reactor**：同上，不过它和 Spring 的集成比较紧密，Spring 的响应式框架都会用到它。
3. **VertX**：完整的响应式框架，以 Netty 为基础，所以它使用 Reactor 并发模型，并可以和 RxJava 共同使用。
4. **Spring WebFlux**：完整的响应式框架，同时集成了 Project Reactor 和 Reactor Netty，所以它也使用 Reactor 并发模型。
5. **Akka**：完整的响应式框架，使用 Actor 并发模型，由于 Actor 模型的特性，比较适用于分布式和微服务体系中。

在这五个库中，它们全部支持Reactive Streams 规范（最后会说），其中 VertX 需要引入 RxJava 才能支持，原因是Reactive Streams 规范其实是比较晚的产物，而前面三个库早早就开始做响应式了。

一般在使用过程中，只有 VertX、Spring WebFlux 和 Akka 是我们会用到的，前两个都基于 Netty 和 Reactor 模型，Akka 则基于Actor 模型且没有使用 Netty（早期曾使用）。

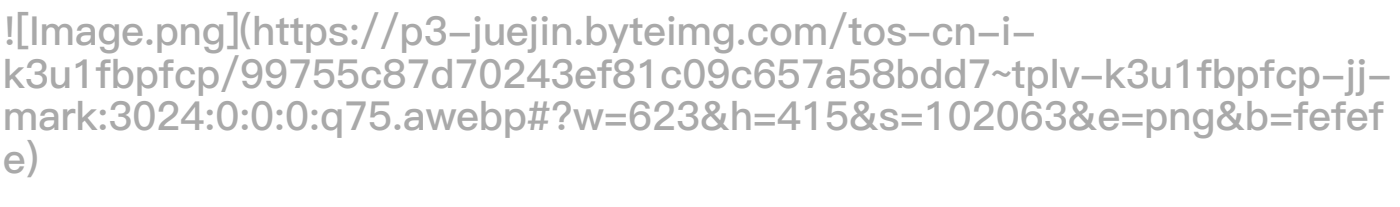
当然，无论是 Reactor 还是 Actor，底层都离不开 Linux epoll 的 IO 多路复用支持，从理论上来说，Actor 是不需要这种操作系统级别的支持的，但是 Linux epoll 能够提供高性能 IO。

Reactor 模型

讲完了一些基础概念，我们来看看 Reactor 模型。

对于 Reactor 模型这个概念，我想大部分人第一次知道它还是在学习 Netty 的过程中，而且 Netty 也确实成为了在 Java 中使用 Reactor 模型构建应用的第一选择，因为它已经提供了完备高效且经过市场验证的基础组件供你使用，你只需要在上面进行二次开发就可以了，所以 VertX 和 Spring WebFlux 都集成了 Netty。

在 Netty 中，我们一般使用 Multi-Reactor 模型，就像下面这样（来自Doug Lee 的Scalable IO in Java）：



Multi-Reactor 模型由三部分组成：MainReactor、SubReactor 和 ThreadPool。

MainReactor 的职责是这样的：

1. 监听客户端请求到达服务端的入口，使用一个独立的线程来处理多个客户端Socket连接接入请求。
2. 将连接分发给 SubReactorr 线程池中的某个线程进行后续处理。
3. 实现了线程池的调度作用,避免了单线程处理所有连接的性能瓶颈。

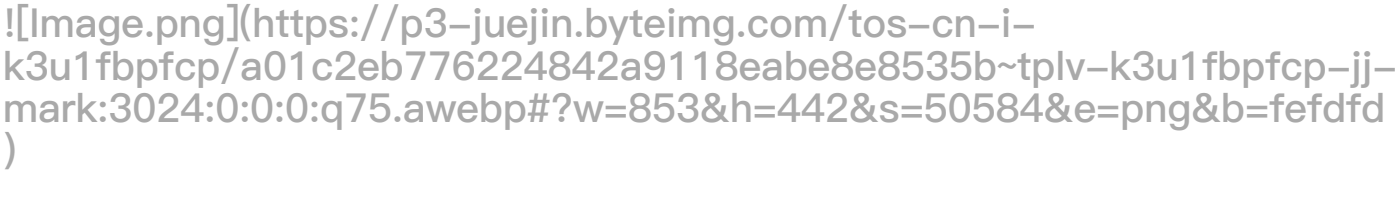
SubReactor的职责是这样的：

1. 一个高度专用化的线程池，包含N个 SubReactor 线程。
2. 每个 SubReactor 线程都使用一个独立的选择器(Selector)监听从MainReactor线程分发过来的多个连接。
3. 处理这些连接上后续的各种I/O事件，如读写事件等。

还有一个真正用来干脏活累活的 ThreadPool，它的职责是这样的：

1. 用于处理实际的业务操作，如编码、解码、计算等CPU密集型操作。
2. 接收 SubReactor 分发过来的事件，并执行实际的业务计算。

在 Netty 的实际设计中，它就采用了上面的模式，但是稍微做了一些改变：



Netty 使用两个 EventLoopGroup来对应MainReactor和SubReactor，它们分别叫作：Boos 和 Work，一般Boss中只有一个 EventLoop而Wrok 中会有cpu

* 2 个，每一个 EventLoop 都会绑定一个线程。

而 Boss 会将每个请求都分发成一个 Channel，每个 Channel 都会由 Wrok从自己掌管 EventLoop 中选取一个处理，由此，就达到了一个线程处理多个Channel的效果。

前文我们已经提到过，在 Reactor 这种模式下，我们有VertX 和 Spring WebFlux 可选，接下来讲讲区别。

VertX 给自己的定位是工具包而不是框架，它的核心只有一个 core，其他例如发送请求、消息传递之类的都需要额外引入相关生态包，也就是说它非常可插拔。

但是虽然它说自己是工具包，但是却拥有最强大的生态，关于响应式的一切你几乎都可以找到，比如主流关系型数据库支持、NoSql 数据库支持、MQ 支持等等。

当然其实 Spring WebFlux 也对响应式常用的都做了支持，而且还能和 Spring 无缝兼容，我用下来感受最多的还是 VertX 更像命令式编程，而 Spring WebFlux 是完全的响应式流风格。

比如下面这个例子中，请求的结果是一个 Future，它很像使用JDK 中的 CompletableFuture 的感觉，因为它的返回值都是 Future，并且通过 Future 进行链式调用：

```
WebClient client = WebClient.create(vertx);

// 发送GET请求
client
  .get(8080, ”myserver.mycompany.com”, ”/some-uri”)
  .send()
  .onSuccess(response -> System.out
    .println(”Received response with status code” +
response.statusCode()))
```

```
  .onFailure(err ->
    System.out.println("Something went wrong " + err.getMessage()));
```

如果你使用 Spring WebFlux，会引入流式处理的概念：

```
@RestController
@RequestMapping("/api")
public class MyController {

    private final WebClient webClient;

    @GetMapping("/data")
    public Mono<String> getData() {
        return webClient.get()
                .uri("https://api.example.com/users") // 目标 API URL
                .retrieve()
                .bodyToMono(String.class); // 将响应体转换为 Mono<String>
                .doOnNext(data -> System.out.println("Received data: " +
data));
    }
}
```

从上面的例子中可以看到响应数据被 Mono 包裹了起来，Spring WebFlux 会订阅这个响应流，然后内部处理，在这个过程中你要被迫学习一下 Reactive Streams 规范，对流式处理有一些认识之后，才能搞懂：Flux、Mono和 Subscribe 这些东西起到的作用。

虽然 Spring WebFlux 上手难度高了点，但是假如我现在开发响应式应用，我的首选还会是 Spring WebFlux，因为你必不可免的会在代码中使用其他 Spring 组件，而 Spring WebFlux 无缝兼容 Spring。

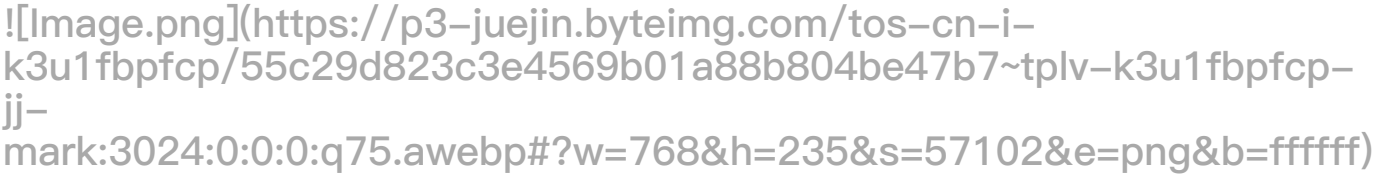
Actor 模型

> Actor模型(Actor model)首先是由Carl Hewitt在1973定义， 由Erlang OTP 推广开来。

> Actor属于并发组件模型，通过组件方式定义并发编程范式的高级阶段，避免使用者直接接触多线程并发或线程池等基础概念。

上面的定义看起来有点云里雾里的，但是我觉得可以用 **消息传递** 来对它做精准的描述。

在 Actor 模型中，组件（Actor）与组件之间没有直接的联系，也就是完全解耦，组件之间全部通过消息而非共享内存来通信并完成业务逻辑，想象一下，你的所有服务逻辑都是通过 MQ 通信的，这种形式就是 Actor 了。



每个组件都可以将消息传递给一个或者多个组件，这种完全解耦的性能使其天然具有高性能、高并发的特点。

在 Java 领域，使用了 Actor 模型的知名框架只有 Akka 框架一个，相较于 Spring WebFlux，我估计大部分人都没有听说过这个框架。

写这篇文章之前，我还问了一下身边会用到响应式编程的网友，它们也基本上都是使用 VertX 和 Spring WebFlux，几乎没有人使用 Akka，原因就是太繁琐，下面给一个超级简单的例子：

```
// 创建ActorSystem
        ActorSystem system = ActorSystem.create("SimpleSystem");

        // 创建PrintActor
        ActorRef printActor =
system.actorOf(Props.create(PrintActor.class), "printActor");

        // 向PrintActor发送消息
        printActor.tell("Hello", ActorRef.noSender());
        printActor.tell("World", ActorRef.noSender());
```

假设你有一个 PrintActor 用于记录日志，那么你需要通过上面两步才能把它创建出来，然后调用 tell 命令发送消息，然后通过对应的 Actor 实例进行处理。

所以在 Akka 编程中，你需要将业务逻辑拆成一个个的 Actor，然后引入一个协调者的角色，对所有的 Actor 聚合之后进行手动流程编排，或者为当前 Actor 引入其他 Actor，形成依赖路径。

这种模式直接将代码编写难度提升一个量级，如果遇到互相依赖等待更要小心处理。

虽然比较繁琐，但是也正因为它的这种特性，它天然适合分布式系统，因为我们传统的分布式系统也是通过 Http 或 rpc 来通信的，也是基于消息传递来通信的。

但是在同一个 JVM 进程内，因为所有的 Actor 都在 JVM 内存中，这样通信起来反而有点画蛇添足，当你的消息传递的对象不是一个不可变对象时，每次传递都会创建这个对象的副本，并传递副本的引用。

还记得我们开头说过的 VertX 也借用了 Actor 的思想嘛，它和 Actor 不同的是消息传递形式是通过消息总线（EventBus），也就是有一个统一的消息传递组件进行消息传递。

而 VertX 也并不强制通过这种方式创建应用，它是可选的，相对的，在 VertX 中有一个和 Akka 的Actor 相对的东西，它叫做： **Verticle**，写法上也几乎一样，需要定义多个 Verticle，然后引入一个协调者将它们编排起来：

```
import io.vertx.core.AbstractVerticle;
import io.vertx.core.Promise;
import io.vertx.core.Vertx;
import io.vertx.core.json.JsonObject;

public class MainVerticle extends AbstractVerticle {
```

```
  @Override
  public void start(Promise<Void> startPromise) {
    vertx.deployVerticle(new UserVerticle());
    vertx.deployVerticle(new AddressVerticle());
    vertx.deployVerticle(new QueryVerticle(), startPromise);
  }
}

class QueryVerticle extends AbstractVerticle {

  @Override
  public void start(Promise<Void> startPromise) {
    vertx.eventBus().consumer("query.address", message -> {
      String userId = message.body().toString();
      queryUser(userId, res -> {
        if (res.succeeded()) {
          JsonObject userJson = res.result();
          queryAddress(userId, addrRes -> {
            if (addrRes.succeeded()) {
              JsonObject addressJson = addrRes.result();
              JsonObject vo = new JsonObject()
                  .mergeIn(userJson)
                  .mergeIn(addressJson);
              message.reply(vo);
              // 记录登录成功日志
            } else {
              message.fail(500, addrRes.cause().getMessage());
            }
          });
        } else {
          message.fail(500, res.cause().getMessage());
        }
      });
    });
    startPromise.complete();
  }

  private void queryUser(String userId, Promise<JsonObject> promise) {
    vertx.eventBus().request("user.get", userId, reply -> {
      if (reply.succeeded()) {
        promise.complete((JsonObject) reply.result().body());
      } else {
        promise.fail(reply.cause());
      }
    });
  }

  private void queryAddress(String userId, Promise<JsonObject>
```

```
promise) {
    vertx.eventBus().request("address.get", userId, reply -> {
      if (reply.succeeded()) {
        promise.complete((JsonObject) reply.result().body());
      } else {
        promise.fail(reply.cause());
      }
    });
  }
}

class UserVerticle extends AbstractVerticle {

  @Override
  public void start(Promise<Void> startPromise) {
    vertx.eventBus().consumer("user.get", message -> {
      String userId = message.body().toString();
      // 实现查询用户逻辑
      JsonObject user = new JsonObject().put("userId",
userId).put("name", "User" + userId);
      message.reply(user);
    });
  }
}

class AddressVerticle extends AbstractVerticle {

  @Override
  public void start(Promise<Void> startPromise) {
    vertx.eventBus().consumer("address.get", message -> {
      String userId = message.body().toString();
      // 实现查询地址逻辑
      JsonObject address = new JsonObject().put("userId",
userId).put("city", "City" + userId);
      message.reply(address);
    });
  }
}

```

上面代码示例中，MainVerticle 就是协调者，其他的都是业务组件，当然你也可以选择将所有逻辑写在同一个 **Verticle** 里面，其实这样你代码运行起来也不会有问题，只是复用性会很差。

Reactive Streams

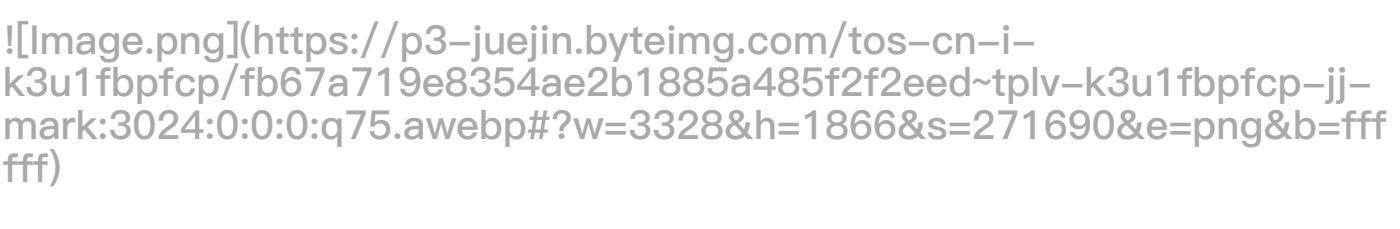
开头的时候我们曾经提到过 **Reactive Streams**，但是由于它出现的比较晚，所以这个词并没有那么流行，甚至在 JDK 中，也是在JDK9 中才引入了FlowAPI，实现了Reactive Streams 规范。

Reactive Streams 是一套处理异步数据流的规范，它定义了一组接口和协议，用于在异步组件之间进行安全、高效的数据交换，并解决数据流中的背压问题。

一般来说，它需要有四个核心组件来描述数据流：

* **Publisher:** 发布者，负责发布数据流中的元素。
* **Subscriber:** 订阅者，接收并处理发布者发布的元素。
* **Subscription:** 订阅，表示订阅者与发布者之间的连接，订阅者可以使用它来请求元素或取消订阅。
* **Processor:** 处理器，既是发布者又是订阅者，可以转换或处理数据流。

借用网上一张图，它的流程是这样的：



看起来就是一个生产者投递消费者接收的模式，但是在序号为 3 的动作中，它有一个 request 的动作，这个动作是用来处理背压的，也就是说当订阅者消费压力大的时候，拉取的消息就少一点，当订阅者消费压力小的时候，拉取的消息就适当多一些。

除此之外，上面几个名词只是标准概念，实际框架中可能并不叫这个名字，比如 Spring WebFlux 中的生产者就是 Flux 和 Mono，分别代表多个和一个。

所以对于 **Reactive Streams** 这个规范，我感觉只需要理解其是怎么回事就可以了，实际编码过程中还是需要实际了解你在用的框架。

最后提醒一句，对于响应式应用，永远不要写阻塞代码。

好了，以上就是本篇文章的全部内容了，希望大家多多点赞支持，我将更快提供更好更优质的内容。

点赞过 100，快速更新下一章：响应式编程会给我们带来哪些问题。

原文链接: https://juejin.cn/post/7368421137917657126