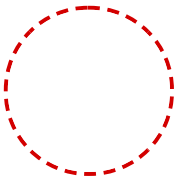
Linux 的 IO 通信 以及 Reactor 线程模型浅析

原创：许光明 杏仁技术站 2018-03-28

作者 | 许光明



杏仁后端工程师。少青年程序员，关注服务端技术和农药。

目录

随着计算机硬件性能不断提高，服务器 CPU 的核数越来越越多，为了充分利用多核 CPU 的处理能力，提升系统的处理效率和并发性能，多线程并发编程越来越显得重要。无论是 C++ 还是 Java 编写的网络框架，大多数都是基于 Reactor 模式进行设计和开发，Reactor 模式基于事件驱动，特别适合处理海量的 I/O 事件，今天我们就简单聊聊 Reactor 线程模型，主要内容分为以下几个部分：

- 经典的 I/O 通信模型；
- Reactor 线程模型详述；
- Reactor 线程模型几种模式；
- Netty Reactor 线程模型的实践；

IO 通信模型

我们先要来谈谈 I/O 通信。说到 I/O 通信，往往会提到同步（synchronous）I/O 、异步（asynchronous）I/O、阻塞（blocking）I/O 和非阻塞（non-blocking）I/O 四种。有关同步、异步、阻塞和非阻塞的区别很多时候解释不清楚，不同的人知识背景不同，对概念很难达成共识。本文讨论的背景是 Linux 环境下的 Network I/O。

一次 I/O 过程分析

对于一次 Network I/O (以 read 举例)，它会涉及到两个系统对象，一个是调用这个 I/O 的进程或线程，另一个就是系统内核 (kernel)。当一个 read 操作发生时，会经历两

个阶段（记住这两个阶段很重要，因为不同 I/O 模型的区别就是在两个阶段上各有不同的处理）：

- 第一个阶段：等待数据准备 (Waiting for the data to be ready)；
- 第二个阶段：将数据从内核拷贝到进程中 (Copying the data from the kernel to the process)；

五种 I/O 模型

Richard Stevens 的《UNIX® Network Programming Volume》提到了 5 种 I/O 模型:

1. Blocking I/O (同步阻塞 I/O)
2. Nonblocking I/O（同步非阻塞 I/O）
3. I/O multiplexing（多路复用 I/O）
4. Signal driven I/O（信号驱动 I/O，实际很少用，Java 不支持）
5. Asynchronous I/O (异步 I/O)

接下来我们对这 5 种 I/O 模型进行说明和对比。

Blocking I/O

在 Linux 中，默认情况下所有的 Socket 都是 blocking 的，也就是阻塞的。一个典型的读操作时，流程如图：

当用户进程调用了 recvfrom 这个系统调用, 这次 I/O 调用经历如下 2 个阶段:

1. 准备数据： 对于网络请求来说，很多时候数据在一开始还没有到达（比如，还没有收到一个完整的 UDP 包），这个时候 kernel 就要等待足够的数据到来。而在用户进程这边，整个进程会被阻塞。

2. 数据返回：kernel 一但等到数据准备好了，它就会将数据从 kernel 中拷贝到用户内存，然后 kernel 返回结果，用户进程才解除 block 的状态，重新运行起来。

Nonblocking IO

Linux 下，可以通过设置 socket 使其变为 non-blocking，也就是非阻塞。当对一个 non-blocking socket 执行读操作时，流程如图：

当用户进程发出 read 操作具体过程分为如下 3 个过程：

1. 开始准备数据：如果 Kernel 中的数据还没有准备好，那么它并不会 block 用户进程，而是立刻返回一个 error。
2. 数据准备中： 从用户进程角度讲，它发起一个read操作后，并不需要等待，而是马上就得到了一个结果。用户进程判断结果是一个 error 时，它就知道数据还没有准备好，于是它可以再次发送 read 操作（重复轮训）。
3. 一旦 kernel 中的数据准备好了，并且又再次收到了用户进程的 system call，那么它马上就将数据拷贝到了用户内存，然后返回。

I/O multiplexing

这种 I/O 方式也可称为 event driven I/O。Linux select/epoll 的好处就在于单个 process 就可以同时处理多个网络连接的 I/O。它的基本原理就是 select/epoll 会不断的轮询所负责的所有 socket，当某个 socket 有数据到达了，就通知用户进程。流程如图：

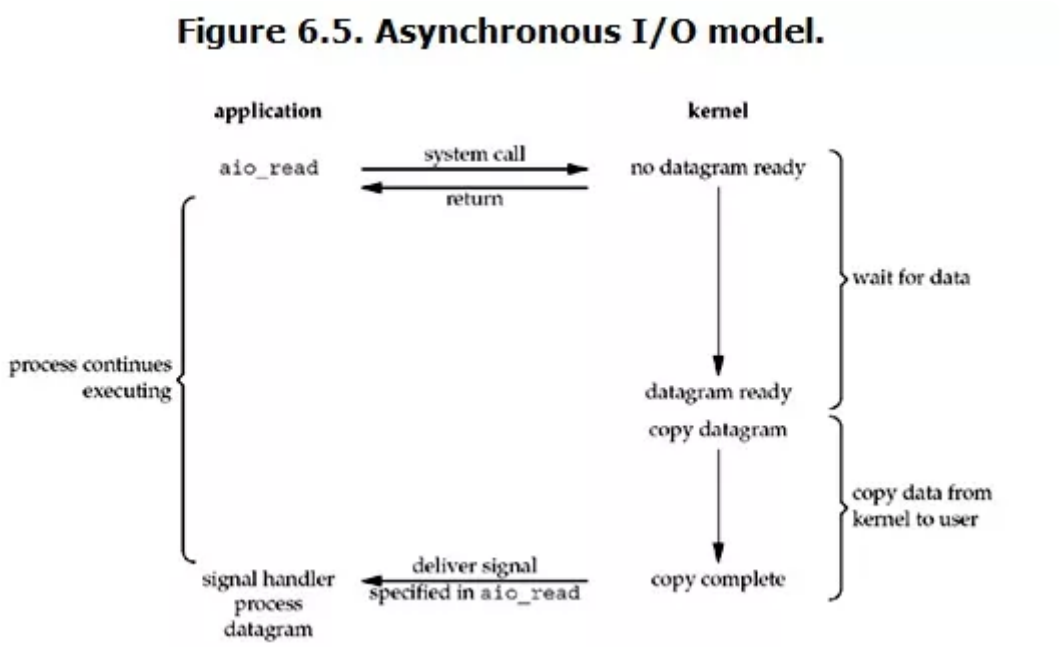
当用户进程调用了 select:

1. 整个进程会被 block，与此同时kernel 会 “监视” 所有 select 负责的 socket，当任何一个 socket 中的数据准备好了，select 就会返回。
2. 户进程再调用 read 操作，将数据从 kernel 拷贝到用户进程。这时和 blocking I/O 的图其实并没有太大的不同，事实上，还更差一些。因为这里需要使用两个 system call (select 和 recvfrom)，而 blocking I/O 只调用了一个 system call (recvfrom)。
3. 在 I/O multiplexing Model 中，实际中，对于每一个 socket，一般都设置成为 non-blocking，但是，如上图所示，整个用户的 process 其实是一直被 block 的。只不过 process 是被 select 这个函数 block，而不是被 socket I/O 给 block。

Asynchronous IO

Linux 下的 asynchronous I/O，即异步 I/O，其实用得很少（需要高版本系统支持）。

它的流程如图：



当用户进程发出 read 操作具体过程：

1. 用户进程发起 read 操作之后，并不需要等待，而是马上就得到了一个结果，立刻就可以开始去做其它的事。
2. 从 kernel 的角度，当它受到一个 asynchronous read 之后，首先它会立刻返回，所以不会对用户进程产生任何 block。然后，kernel 会等待数据准备完成，然后将数据拷贝到用户内存，当这一切都完成之后，kernel会给用户进程发送一个 signal，告诉它 read 操作完成了。

通过以上 4 种 I/O 通信模型的说明，总结一下它们各自的特点：

- Blocking I/O 的特点就是在 I/O 执行的两个阶段都被 block 了。
- Non-blocking I/O 特点是如果 kernel 数据没准备好不需要阻塞。
- I/O multiplexing 的优势在于它用 select 可以同时处理多个 connection。（如果处理的连接数不是很高的话，使用 select/epoll 的 web server 不一定比使用 multi-threading + blocking I/O 的 web server 性能更好，可能延迟还更大。select/epoll 的优势并不是对于单个连接能处理得更快，而是在于能处理更多的连接。）
- Asynchronous IO 的特点在于整个调用过程客户端没有任何 block 状态，但是需要高版本的系统支持。

生活中通信模型

以上五种 I/0 模型的介绍，比如枯燥，其实在生活中也存在类似的“通信模型”，为了帮助理解，我们用生活中约妹纸吃饭这个不是很恰当的例子来说明这几个 I/O Model（假设我现在要用微信叫几个妹纸吃饭）：

- 发个微信问第一个妹纸好了没，妹子没回复就一直等，直到回复在发第二个（blocking I/O）。
- 发个微信问第一个妹纸好了没，妹子没回复先不管，发给第二个，但是过会要继续问之前 没有回复的妹纸有没有好（nonblocking I/O）。
- 将所有妹纸拉一个微信群，过会在群里问一次，谁好了回复一下（I/O multiplexing）。
- 直接告诉妹纸吃饭的时间地址，好了自己去就行（Asynchronous I/O）。

Reactor 线程模型

Reactor 是什么？

Reactor 是一种处理模式。 Reactor 模式是处理并发 I/O 比较常见的一种模式，用于同步 I/O，中心思想是将所有要处理的IO事件注册到一个中心 I/O 多路复用器上，同时主线程/进程阻塞在多路复用器上；一旦有 I/O 事件到来或是准备就绪(文件描述符或 socket 可读、写)，多路复用器返回并将事先注册的相应 I/O 事件分发到对应的处理器中。

Reactor 也是一种实现机制。 Reactor 利用事件驱动机制实现，和普通函数调用的不同之处在于：应用程序不是主动的调用某个 API 完成处理，而是恰恰相反，Reactor 逆置了事件处理流程，应用程序需要提供相应的接口并注册到 Reactor 上，如果相应的事件发生，Reactor 将主动调用应用程序注册的接口，这些接口又称为 “回调函数”。用 “好莱坞原则” 来形容 Reactor 再合适不过了：不要打电话给我们，我们会打电话通知你。

为什么要使用 Reactor？

一般来说通过 I/O 复用，epoll 模式已经可以使服务器并发几十万连接的同时，维持极高 TPS，为什么还需要 Reactor 模式？原因是原生的 I/O 复用编程复杂性比较高。

一个个网络请求可能涉及到多个 I/O 请求，相比传统的单线程完整处理请求生命期的方法，I/O 复用在人的大脑思维中并不自然，因为，程序员编程中，处理请求 A 的时候，假定 A 请求必须经过多个 I/O 操作 A1-An（两次 IO 间可能间隔很长时间），每经过一次 I/O 操作，再调用 I/O 复用时，I/O 复用的调用返回里，非常可能不再有 A，而是返回了请求 B。即请求 A 会经常被请求 B 打断，处理请求 B 时，又被 C 打断。这种思维下，编程容易出错。

Reactor 线程模型

Reactor 有三种线程模型，用户能够更加自己的环境选择适当的模型。

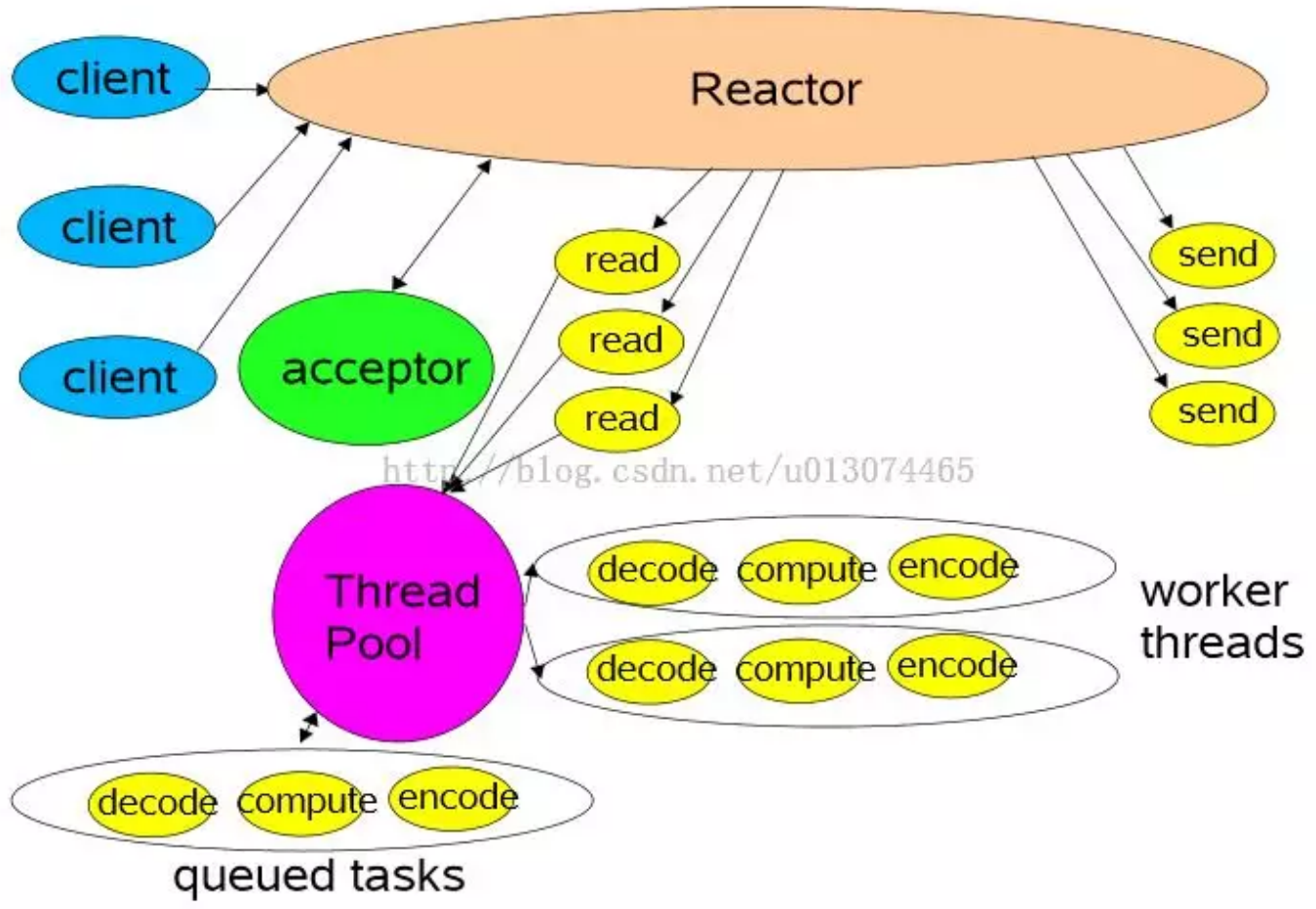
1. 单线程模型
2. 多线程模型（单 Reactor）
3. 多线程模型（多 Reactor)

单线程模式

单线程模式是最简单的 Reactor 模型。Reactor 线程是个多面手，负责多路分离套接字，Accept 新连接，并分派请求到处理器链中。该模型适用于处理器链中业务处理组件能快速完成的场景。不过这种单线程模型不能充分利用多核资源，所以实际使用的不多。

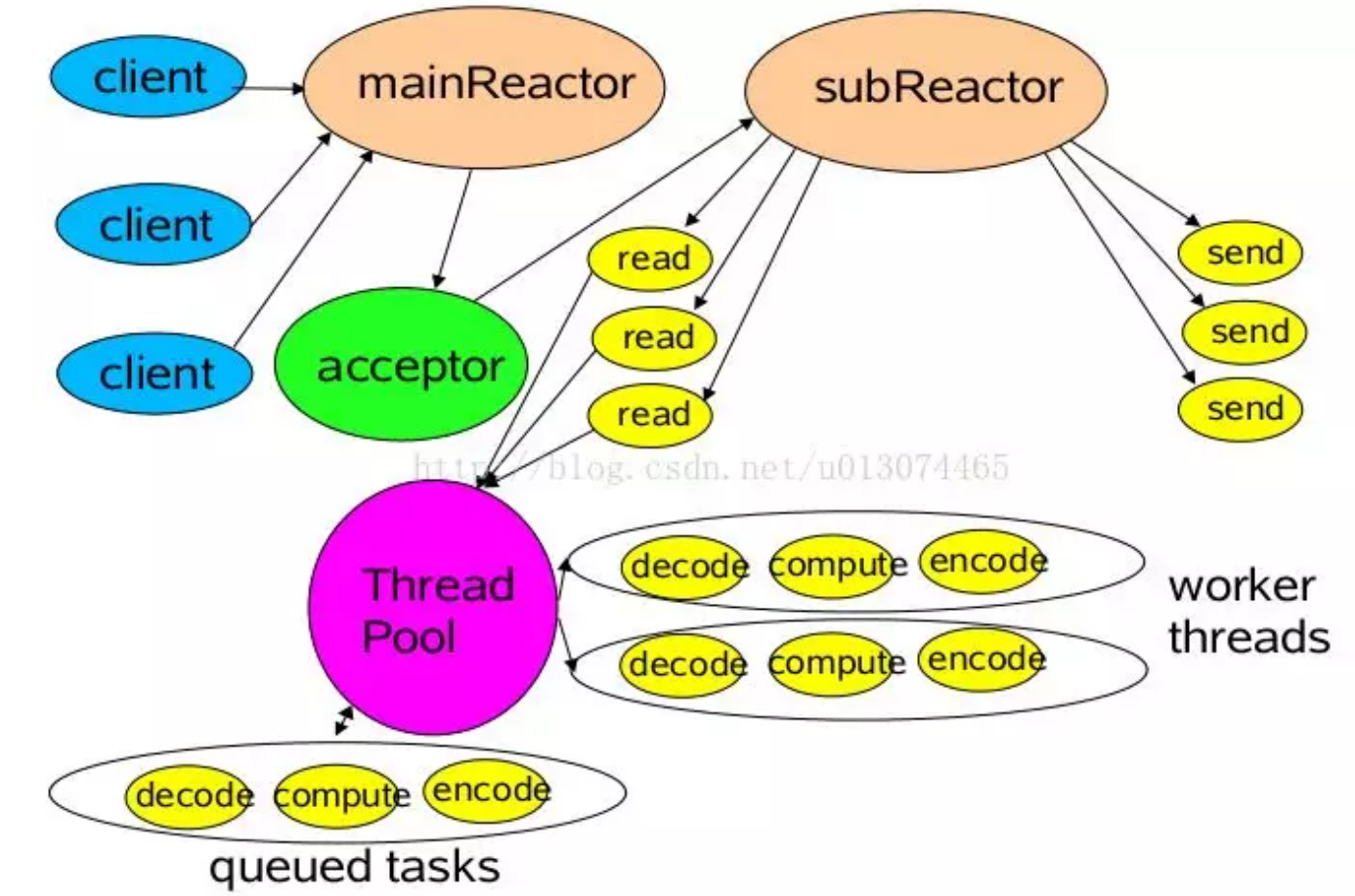
多线程模式(单 Reactor)

该模型在事件处理器（Handler）链部分采用了多线程（线程池），也是后端程序常用的模型。



多线程模式(多 Reactor)

比起多线程单 Rector 模型，它是将 Reactor 分成两部分，mainReactor 负责监听并 Accept 新连接，然后将建立的 socket 通过多路复用器（Acceptor）分派给 subReactor。subReactor 负责多路分离已连接的 socket，读写网络数据；业务处理功能，其交给 worker 线程池完成。通常，subReactor 个数上可与 CPU 个数等同。



Reactor 使用

软件领域很多开源的产品使用了 Ractor 模型，比如 Netty。

Netty Reactor 实践

服务端线程模型

服务端监听线程和 I/O 线程分离，类似于 Reactor 的多线程模型，它的工作原理图如下：

服务端用户线程创建

- 创建服务端的时候实例化了 2 个 EventLoopGroup。bossGroup 线程组实际就是 Acceptor 线程池，负责处理客户端的 TCP 连接请求。workerGroup 是真正负责 I/O 读写操作的线程组。通过这里能够知道 Netty 是多 Reactor 模型。
- ServerBootstrap 类是 Netty 用于启动 NIO 的辅助类，能够方便开发。通过 group 方法将线程组传递到 ServerBootstrap 中，设置 Channel 为 NioServerSocketChannel，接着设置 NioServerSocketChannel 的 TCP 参数，最后绑定 I/O 事件处理类 ChildChannelHandler。
- 辅助类完成配置之后调用 bind 方法绑定监听端口，Netty 返回 ChannelFuture，f.channel().closeFuture().sync() 对同步阻塞的获取结果。
- 调用线程组 shutdownGracefully 优雅推出，释放资源。

```
public class TimeServer {
	public void bind(int port) {
		// 配置服务端的NIO线程组
		EventLoopGroup bossGroup = new NioEventLoopGroup();
		EventLoopGroup workGroup = new NioEventLoopGroup();
		try {
			ServerBootstrap b = new ServerBootstrap();
			b.group(bossGroup, workGroup).channel(NioServerSocketChannel.class)
					.option(ChannelOption.SO_BACKLOG, 1024)
					.childHandler(new ChildChannelHandler());
			// 绑定端口，同步等待成功
			ChannelFuture f = b.bind(port).sync();
			// 等待服务端监听端口关闭
			f.channel().closeFuture().sync();
		} catch (Exception e) {
			e.printStackTrace();
		} finally {
			// 释放线程池资源
			bossGroup.shutdownGracefully();
			workGroup.shutdownGracefully();
		}
	}
	private class ChildChannelHandler extends ChannelInitializer<SocketChannel> {
		@Override

		protected void initChannel(SocketChannel ch) throws Exception {
			ch.pipeline().addLast(new TimeServerHandler());
		}
	}
}
```

服务端 I/O 线程处理（TimeServerHandler）

- exceptionCaught 方法：当 I/O 处理发生异常时被调用，关闭 ChannelHandlerContext，释放资源。
- channelRead 方法: 是真正处理读写数据的方法，通过 buf.readBytes 读取请求数据。通过 ctx.write(resp) 将相应报文发送给客户端。
- channelReadComplete 方法: 为了提高性能，Netty write 是将数据先写到缓冲数组，通过 flush 方法可以将缓冲数组的所有消息发送到 SocketChannel 中。

```
public class TimeServerHandler extends ChannelHandlerAdapter {

	@Override
	public void exceptionCaught(ChannelHandlerContext ctx, Throwable cause)
			throws Exception {
		ctx.close();
	}

	@Override
	public void channelRead(ChannelHandlerContext ctx, Object msg)
			throws Exception {
		// msg转Buf
		ByteBuf buf = (ByteBuf) msg;
		// 创建缓冲中字节数的字节数组
		byte[] req = new byte[buf.readableBytes()];
		// 写入数组
		buf.readBytes(req);
		String body = new String(req, "UTF-8");
		String currenTime = "QUERY TIME ORDER".equalsIgnoreCase(body) ? new Date(
				System.currentTimeMillis()).toString() : "BAD ORDER";
		// 将要返回的信息写入Buffer
		ByteBuf resp = Unpooled.copiedBuffer(currenTime.getBytes());
		// buffer写入通道
		ctx.write(resp);
	}

	@Override
	public void channelReadComplete(ChannelHandlerContext ctx) throws Exception {
		// write读入缓冲数组后通过invoke flush写入通道
		ctx.flush();
	}
}
```

总结

通过以上大概了解 Reactor 相关知识。最后做个总结一下使用 Reactor 模型的优缺点。

- 优点
  - 响应快，虽然 Reactor 本身依然是同步的，不必为单个同步时间所阻塞。
  - 编程相对简单，可以最大程度的避免复杂的多线程及同步问题，并且避免了多线程/进程的切换开销。

    - 可扩展性，通过并发编程的方式增加 Reactor 个数来充分利用 CPU 资源。
    - 可复用性，Reactor 框架本身与具体事件处理逻辑无关，具有很高的复用性。
- 缺点
    - 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，调试相对复杂。
    - Reactor 模式需要底层的 Synchronous Event Demultiplexer 支持，例如 Java 中的 Selector，操作系统的 select 系统调用支持。
    - 单线程 Reactor 模式在 I/O 读写数据时还是在同一个线程中实现的，即使使用多 Reactor 机制的情况下，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，I/O 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。

全文完

以下文章您可能也会感兴趣：

- 微服务环境下的集成测试探索（一）—— 服务 Stub & Mock
- 微服务环境下的集成测试探索（二）—— 契约式测试
- 乐高式微服务化改造（上）
- 乐高式微服务化改造（下）
- 一个创业公司的容器化之路（一）- 容器化之前
- 一个创业公司的容器化之路（二）- 容器化
- 一个创业公司的容器化之路（三）- 容器即未来
- 响应式编程（上）：总览
- 响应式编程（下）：Spring 5
- 复杂业务状态的处理：从状态模式到 FSM
- 后端的缓存系统浅谈
- 谈谈到底什么是抽象，以及软件设计的抽象原则
- 所谓 Serverless，你理解对了吗？

我们正在招聘 Java 工程师，欢迎有兴趣的同学投递简历到 rd-hr@xingren.com 。

杏仁技术站

长按左侧二维码关注我们，这里有一群热血青年期待着与您相会。