《Java 性能优化实战 21 讲》

李国 前京东、陌陌高级架构师

— 拉勾教育出品 —

15 | 案例分析：从 BIO 到 NIO，再到 AIO

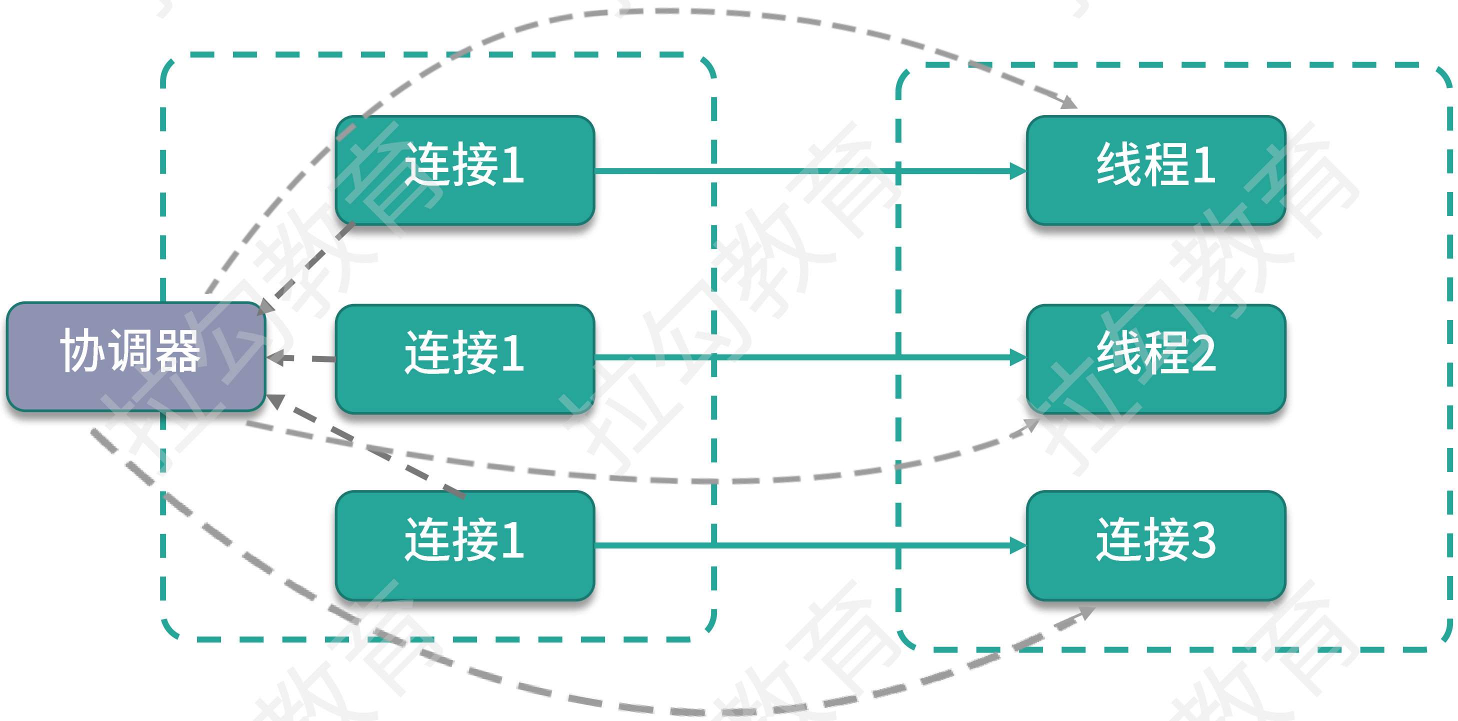
Netty 的高性能架构，是基于一个网络编程设计模式 Reactor 进行设计的

大多数与 I/O 相关的组件，都会使用 Reactor 模型，比如 Tomcat、Redis、Nginx 等

案例分析：从 BIO 到 NIO，再到 AIO

为什么 NIO 的性能就能够比传统的阻塞 I/O 性能高呢

阻塞 I/O 模型



BIO 模型图

阻塞 I/O 模型

拉勾教育
— 互联网人实战大学 —

```java
public class BIO {
  static boolean stop = false;

  public static void main(String[] args) throws Exception {
    int connectionNum = 0;
    int port = 8888;
    ExecutorService service = Executors.newCachedThreadPool();
    ServerSocket serverSocket = new ServerSocket(port);
    while (!stop) {
      if (10 == connectionNum) {
        stop = true;
      }
      Socket socket = serverSocket.accept();
      service.execute(() -> {
        try {
          Scanner scanner = new Scanner(socket.getInputStream());
          PrintStream printStream = new PrintStream(socket.getOutputStream());
          while (!stop) {
            String s = scanner.next().trim();
            printStream.println("PONG:" + s);
          }
        } catch (Exception ex) {
```

阻塞 I/O 模型

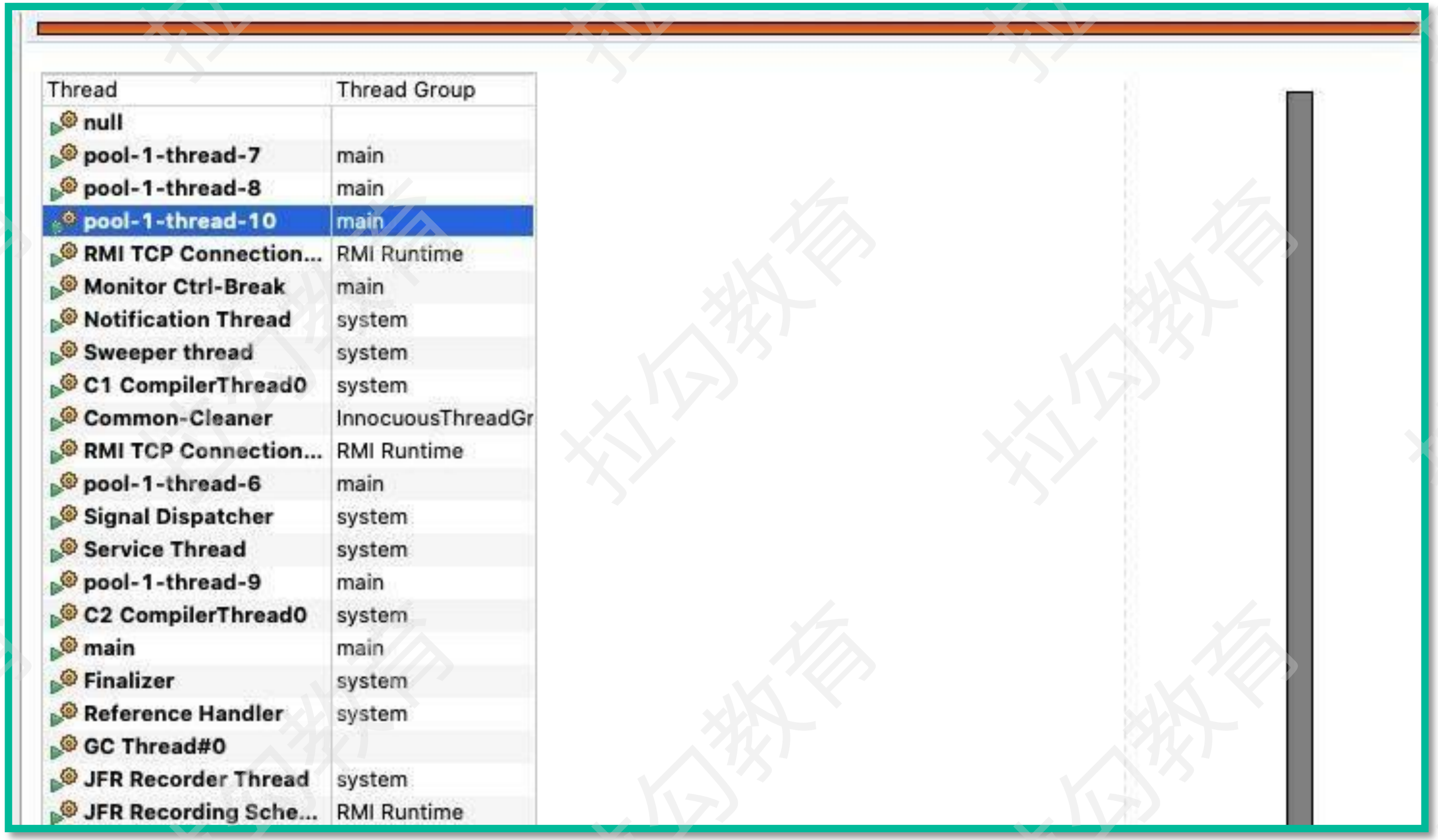
拉勾教育
— 互联网人实战大学 —

```java
        if (10 == connectionNum) {
          stop = true;
        }
        Socket socket = serverSocket.accept();
        service.execute(() -> {
          try {
            Scanner scanner = new Scanner(socket.getInputStream());
            PrintStream printStream = new PrintStream(socket.getOutputStream());
            while (!stop) {
              String s = scanner.next().trim();
              printStream.println("PONG:" + s);
            }
          } catch (Exception ex) {
            ex.printStackTrace();
          }
        });
        connectionNum++;
      }
      service.shutdown();
      serverSocket.close();
    }
}
```

阻塞 I/O 模型

```
$ nc -v localhost 8888
Connection to localhost port 8888 [tcp/ddi-tcp-1]
succeeded!
hello
PONG:hello
nice
PONG:nice
```

阻塞 I/O 模型



Thread	Thread Group
null	
pool-1-thread-7	main
pool-1-thread-8	main
pool-1-thread-10	main
RMI TCP Connection...	RMI Runtime
Monitor Ctrl-Break	main
Notification Thread	system
Sweeper thread	system
C1 CompilerThread0	system
Common-Cleaner	InnocuousThreadGr
RMI TCP Connection...	RMI Runtime
pool-1-thread-6	main
Signal Dispatcher	system
Service Thread	system
pool-1-thread-9	main
C2 CompilerThread0	system
main	main
Finalizer	system
Reference Handler	system
GC Thread#0	
JFR Recorder Thread	system
JFR Recording Sche...	RMI Runtime

非阻塞 I/O 模型

socket 连接要花费很长时间进行连接操作，在完成连接的这段时间内，它只能阻塞等待在线程中

epoll 是一个高性能的多路复用 I/O 工具

通过 epoll_create 和 epoll_ctl 等函数的操作

可以构造描述符（fd）相关的事件组合（event）

非阻塞 I/O 模型

拉勾教育

— 互联网人实战大学 —

Fd

每条连接、每个文件，都对应着一个描述符，比如端口号

event

当 fd 对应的资源，有状态或者数据变动，会更新 epoll_item 结构

非阻塞 I/O 模型

相对于 select，epoll 有哪些改进？

非阻塞 I/O 模型

01 epoll 不需要像 select 一样对 fd 集合进行轮询，也不需要在调用时将 fd 集合在用户态和内核态进行交换

02 应用程序获得就绪 fd 的事件复杂度，epoll 时 O(1)，select 是 O(n)

03 select 最大支持约 1024 个 fd，epoll 支持 65535个

04 select 使用轮询模式检测就绪事件，epoll 采用通知方式，更加高效

非阻塞 I/O 模型


```java
public class NIO {
  static boolean stop = false;

  public static void main(String[] args) throws Exception {
    int connectionNum = 0;
    int port = 8888;
    ExecutorService service = Executors.newCachedThreadPool();

    ServerSocketChannel ssc = ServerSocketChannel.open();
    ssc.configureBlocking(false);
    ssc.socket().bind(new InetSocketAddress("localhost", port));

    Selector selector = Selector.open();
    ssc.register(selector, ssc.validOps());


    while (!stop) {
      if (10 == connectionNum) {
        stop = true;
      }
```

非阻塞 I/O 模型

拉勾教育


```java
int num = selector.select();
if (num == 0) {
  continue;
}
Iterator<SelectionKey> events = selector.selectedKeys().iterator();
while (events.hasNext()) {
  SelectionKey event = events.next();

  if (event.isAcceptable()) {
    SocketChannel sc = ssc.accept();
    sc.configureBlocking(false);
    sc.register(selector, SelectionKey.OP_READ);
    connectionNum++;
  } else if (event.isReadable()) {
    try {
      SocketChannel sc = (SocketChannel) event.channel();
      ByteBuffer buf = ByteBuffer.allocate(1024);
      int size = sc.read(buf);
      if(-1==size){
        sc.close();
```

非阻塞 I/O 模型


```
            if(-1==size){
              sc.close();
            }
            String result = new String(buf.array()).trim();
            ByteBuffer wrap = ByteBuffer.wrap(("PONG:" + result).getBytes());
            sc.write(wrap);
          } catch (Exception ex) {
            ex.printStackTrace();
          }
        } else if (event.isWritable()) {
          SocketChannel sc = (SocketChannel) event.channel();
        }

        events.remove();
      }
    }
    service.shutdown();
    ssc.close();
  }
}
```

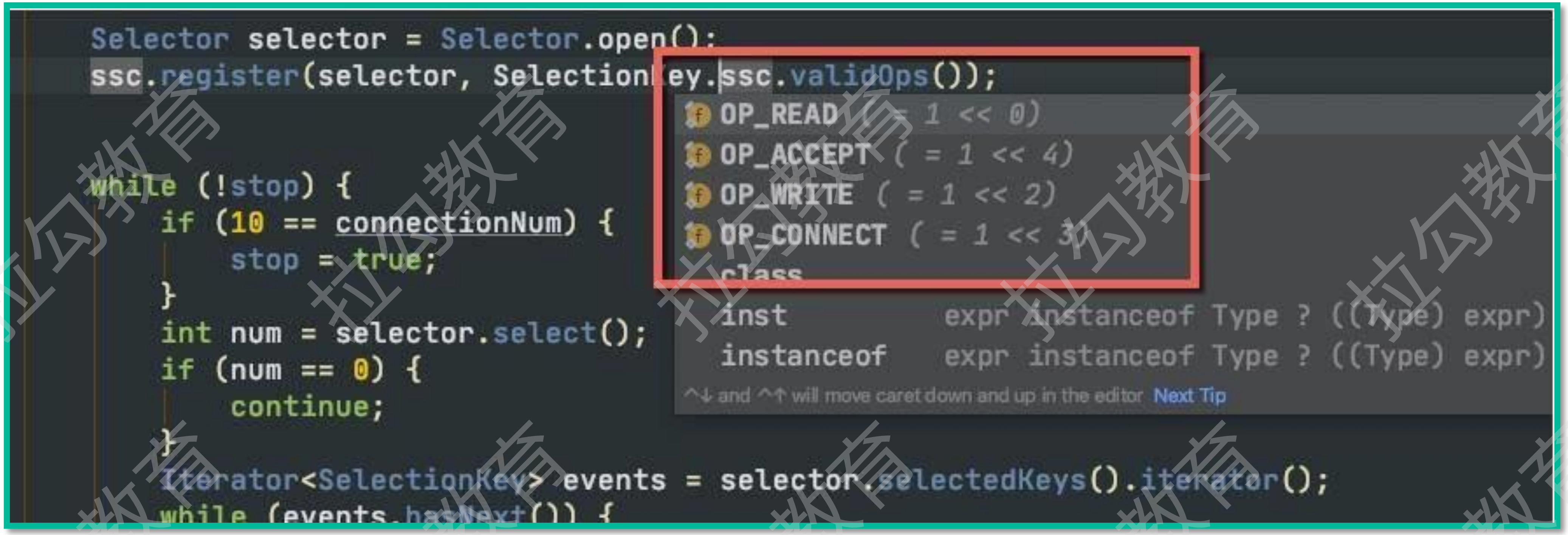
非阻塞 I/O 模型

建了一个服务端 ssc，并开启一个新的事件选择器

监听它的 OP_ACCEPT 事件

```
ServerSocketChannel ssc = ServerSocketChannel.open();
Selector selector = Selector.open();
ssc.register(selector, ssc.validOps());
```

非阻塞 I/O 模型


Selector selector = Selector.open();
ssc.register(selector, SelectionKey.ssc.validOps());
OP_READ (= 1 << 0)
OP_ACCEPT (= 1 << 4)
OP_WRITE (= 1 << 2)
OP_CONNECT (= 1 << 3)
class
inst expr instanceof Type ? ((Type) expr)
instanceof expr instanceof Type ? ((Type) expr)
^↓ and ^↑ will move caret down and up in the editor Next Tip
while (!stop) {
if (10 == connectionNum) {
stop = true;
}
int num = selector.select();
if (num == 0) {
continue;
}
Iterator<SelectionKey> events = selector.selectedKeys().iterator();
while (events.hasNext()) {

非阻塞 I/O 模型

阻塞——操作系统不再分配 CPU 时间片到当前线程中

```
int num = selector.select();
```

非阻塞 I/O 模型

如果事件不删除的话，或者漏掉了某个事件的处理，

会有什么后果？

非阻塞 I/O 模型

拉勾教育

— 互联网人实战大学 —

```java
Iterator<SelectionKey> events = selector.selectedKeys().iterator();
  while (events.hasNext()) {
    SelectionKey event = events.next();
    ...
    events.remove();
  }
}
```

非阻塞 I/O 模型


```
SocketChannel sc = ssc.accept();
sc.configureBlocking(false);
sc.register(selector, SelectionKey.OP_READ);
```

非阻塞 I/O 模型


```
SocketChannel sc = ssc.accept();
sc.configureBlocking(false);
sc.register(selector, SelectionKey.OP_READ);
```

注意：服务端和客户端的实现方式，可以是不同的

非阻塞 I/O 模型

非阻塞 I/O 模型


```java
SocketChannel sc = (SocketChannel) event.channel();

ByteBuffer buf = ByteBuffer.allocate(1024);

int size = sc.read(buf);
```

非阻塞 I/O 模型

拉勾教育

— 互联网人实战大学 —

水平触发

(level-triggered)

称作 LT 模式

只要缓冲区有数据，事件就会一直发生

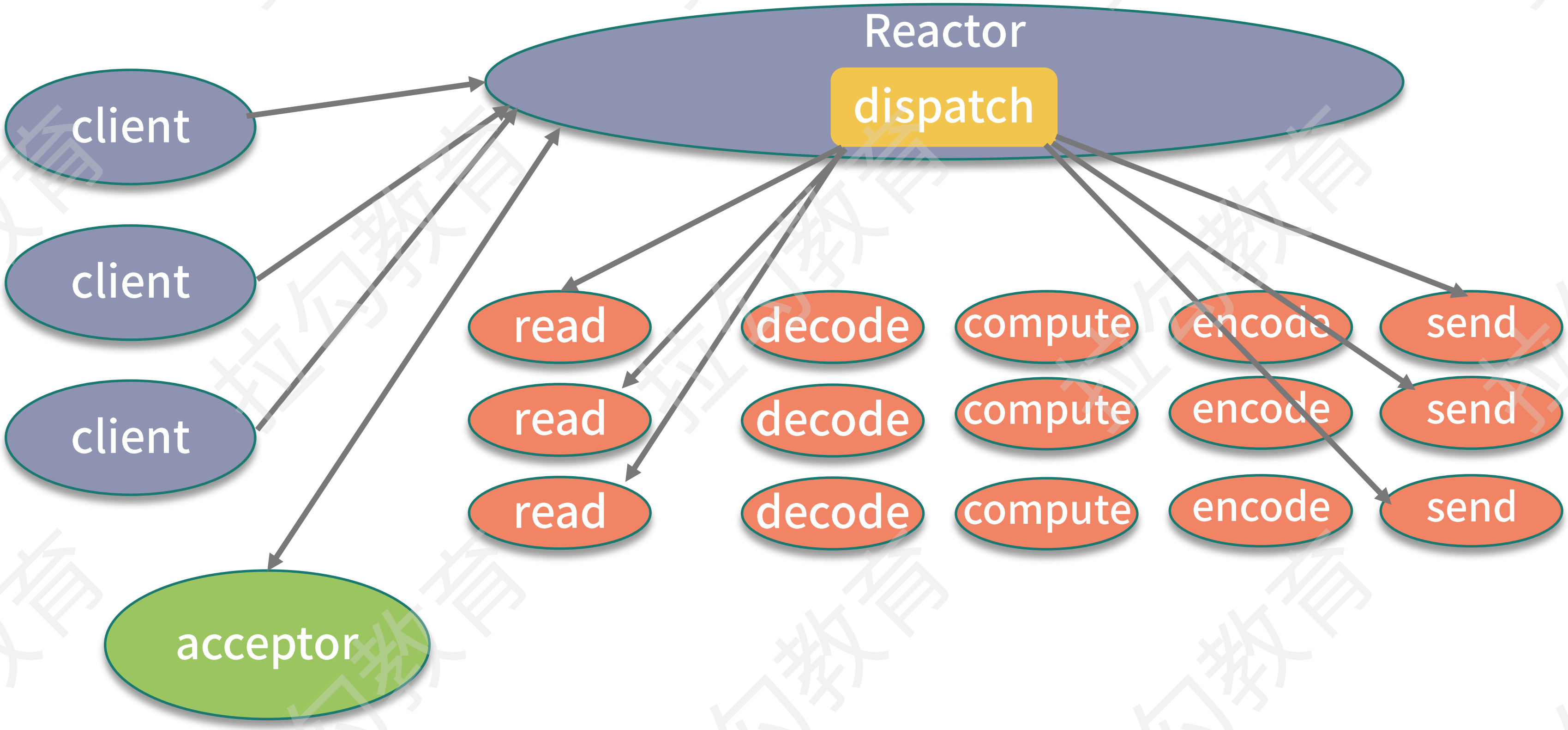
边缘触发

(edge-triggered)

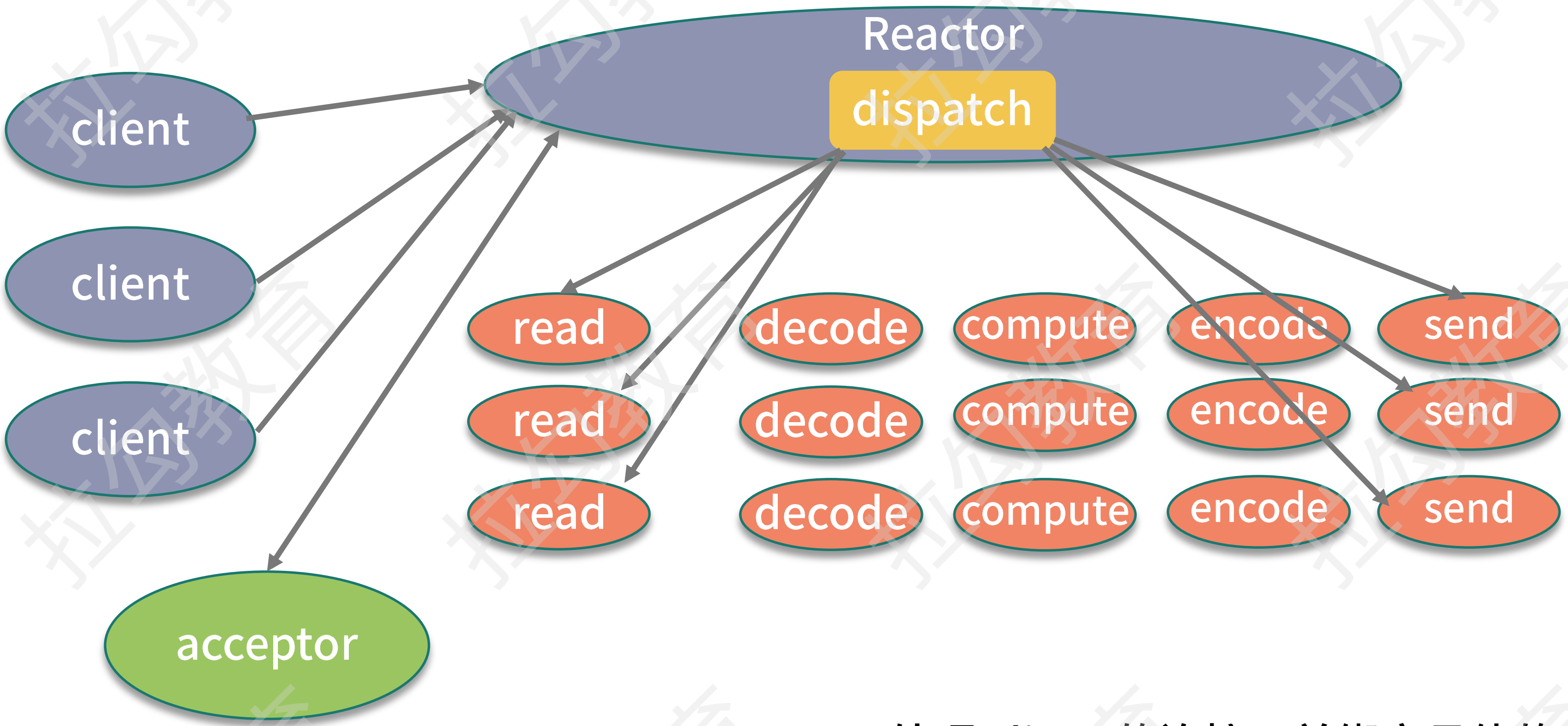
称作 ET 模式

缓冲区有数据，仅会触发一次

Reactor 模式

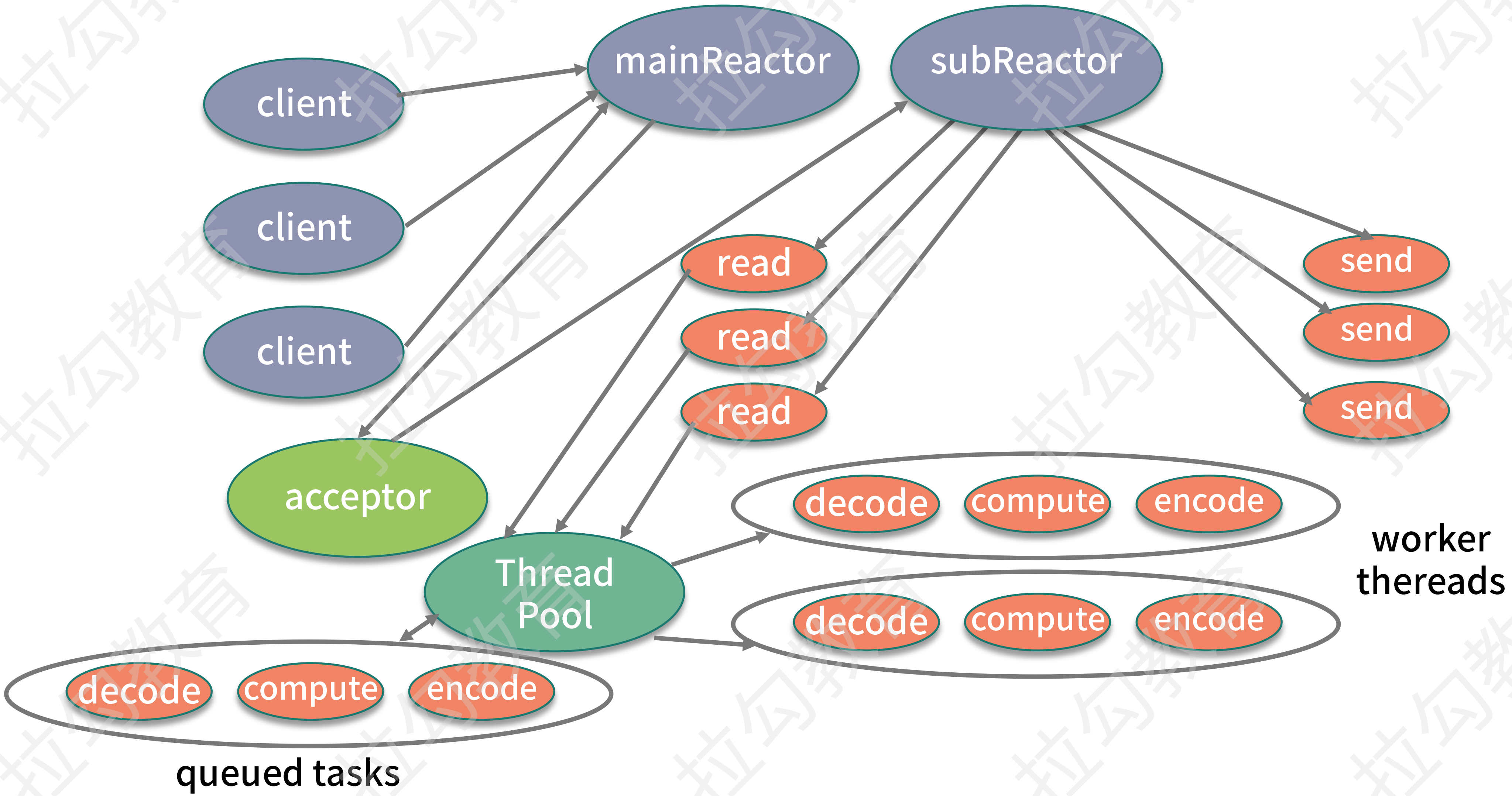

Reactor
dispatch
client
client
client
acceptor
read
decode
compute
encode
send
read
decode
compute
encode
send
read
decode
compute
encode
send

Reactor 模式



- Acceptor 处理 client 的连接，并绑定具体的事件处理器
- Event 具体发生的事件，比如图中 s 的 read、send 等
- Handler 执行具体事件的处理者，比如处理读写事件的具体逻辑
- Reactor 将具体的事件分配（dispatch）给 Handler

Reactor 模式

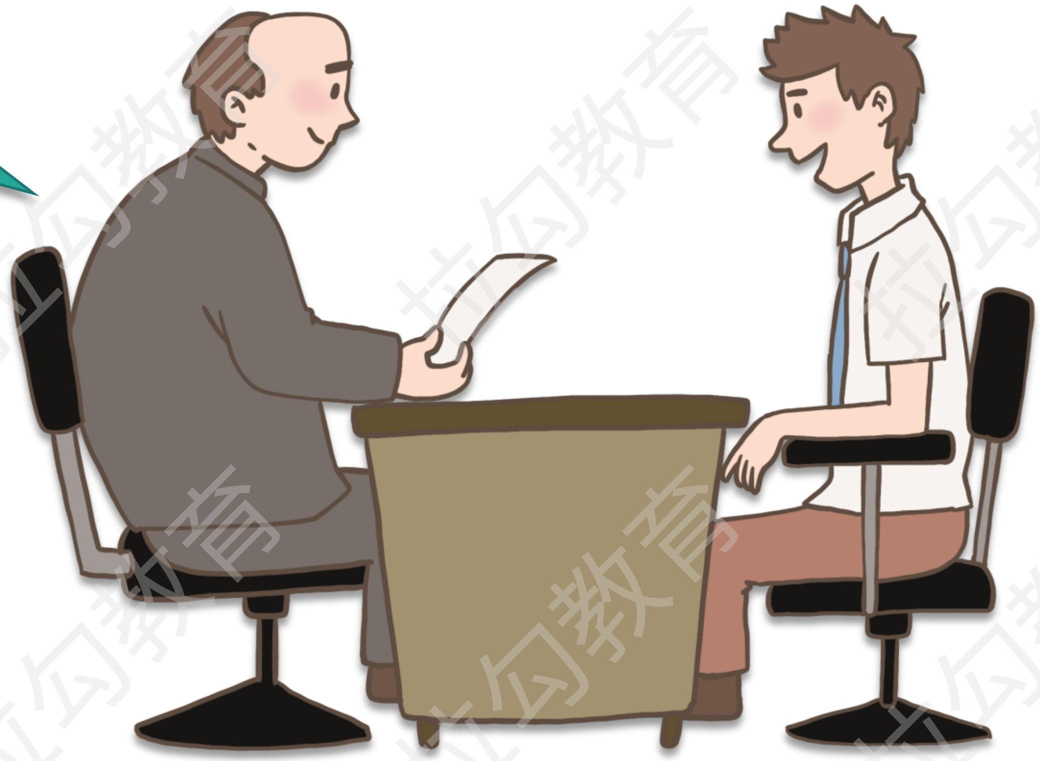


client
client
client
mainReactor
subReactor
read
read
read
send
send
send
acceptor
Thread Pool
decode
compute
encode
decode
compute
encode
worker thereads
decode
compute
encode
queued tasks

AIO

为什么我在使用 NIO 时，使用 Channel 进行读写，socket 的操作依然是阻塞的？NIO 的作用主要体现在哪里？

```
//这行代码是阻塞的
int size = sc.read(buf);
```



AIO


```
//这行代码是阻塞的
int size = sc.read(buf);
```

NIO 只负责对发生在 fd 描述符上的事件进行通知

事件的获取和通知部分是非阻塞的，但收到通知之后的操作

却是阻塞的，即使使用多线程去处理这些事件

它依然是阻塞的

AIO


```java
public class AIO {
  public static void main(String[] args) throws Exception {
    int port = 8888;
    AsynchronousServerSocketChannel ssc = AsynchronousServerSocketChannel.open();
    ssc.bind(new InetSocketAddress("localhost", port));
    ssc.accept(null, new CompletionHandler<AsynchronousSocketChannel, Object>() {
      void job(final AsynchronousSocketChannel sc) {
        ByteBuffer buffer = ByteBuffer.allocate(1024);
        sc.read(buffer, buffer, new CompletionHandler<Integer, ByteBuffer>() {
          @Override
          public void completed(Integer result, ByteBuffer attachment) {
            String str = new String(attachment.array()).trim();
            ByteBuffer wrap = ByteBuffer.wrap(("PONG:" + str).getBytes());
            sc.write(wrap, null, new CompletionHandler<Integer, Object>() {
              @Override
              public void completed(Integer result, Object attachment) {
                job(sc);
              }
              @Override
              public void failed(Throwable exc, Object attachment) {
```

AIO

拉勾教育

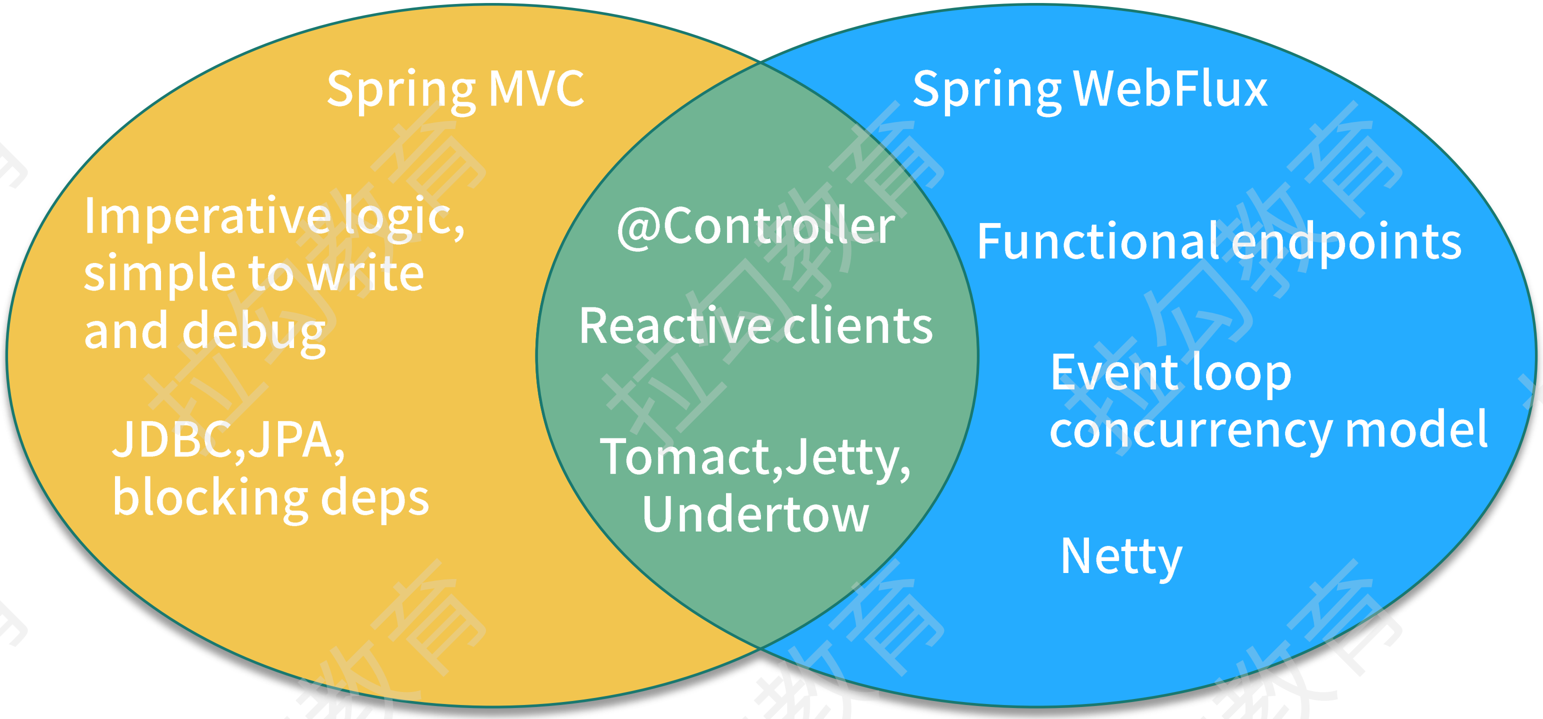
— 互 联 网 人 实 战 大 学 —

```
            System.out.println("error");
        }
      });
    }
    @Override
    public void failed(Throwable exc, ByteBuffer attachment) {
      System.out.println("error");
    }
  });
}
@Override
public void completed(AsynchronousSocketChannel sc, Object attachment) {
  ssc.accept(null, this);
  job(sc);
}
@Override
public void failed(Throwable exc, Object attachment) {
  exc.printStackTrace();
  System.out.println("error");
}
```

AIO

```
            @Override
            public void failed(Throwable exc, ByteBuffer attachment) {
                System.out.println("error");
            }
        });
    }
    @Override
    public void failed(Throwable exc, Object attachment) {
        exc.printStackTrace();
        System.out.println("error");
    }
    @Override
    public void completed(AsynchronousSocketChannel sc, Object attachment) {
        ssc.accept(null, this);
        job(sc);
    }
    @Override
    public void failed(Throwable exc, Object attachment) {
        exc.printStackTrace();
        System.out.println("error");
    }
  });
  Thread.sleep(Integer.MAX_VALUE);
 }
}
```

响应式编程



Spring MVC
Imperative logic, simple to write and debug
JDBC,JPA, blocking deps
@Controller
Reactive clients
Tomact,Jetty, Undertow
Spring WebFlux
Functional endpoints
Event loop concurrency model
Netty

响应式编程

响应式编程	背压（Backpressure）
一种面向数据流和变化传播的编程范式	生产者与消费者之间的流量控制，通过将操作全面异步化，来减少无效的等待和资源消耗

响应式编程


```
public RouteLocator customerRouteLocator(RouteLocatorBuilder builder) {
    return builder.routes()
        .route(r -> r.path("/market/**")
            .filters(f -> f.filter(new RequestTimeFilter())
                .addResponseHeader("X-Response-Default-Foo", "Default-Bar"))
            .uri("http://localhost:8080/market/list")
            .order(0)
            .id("customer_filter_router")
        )
        .build();
}
```

响应式编程


```
public RouteLocator customerRouteLocator(RouteLocatorBuilder builder) {
    return builder.routes()
        .route(r -> r.path("/market/**")
            .filters(f -> f.filter(new RequestTimeFilter())
                .addResponseHeader("X-Response-Default-Foo", "Default-Bar"))
            .uri("http://localhost:8080/market/list")
            .order(0)
            .id("customer_filter_router")
        )
        .build();
}
```

小结

BIO 的线程模型是一个连接对应一个线程的，非常浪费资源

NIO 通过对关键事件的监听，通过主动通知的方式完成非阻塞操作，但它对事件本身的处理依然是非阻塞的

AIO 完全是异步非阻塞的，但现实中使用很少

小结

BIO 的线程模型是一个连接对应一个线程的，非常浪费资源

NIO 通过对关键事件的监听，通过主动通知的方式完成非阻塞操作，但它对事件本身的处理依然是非阻塞的

AIO 完全是异步非阻塞的，但现实中使用很少

Netty 的事件触发机制使用了高效的 ET 模式，使得支持的连接更多，性能更高

Next：16 |《案例分析：常见 Java 代码优化法则》