Netty

> Netty是Java领域有名的开源网络库，特点是高性能和高扩展性，因此很多流行的框架都是基于它来构建的，比如我们熟知的Dubbo、Rocketmq、Hadoop等，针对高性能RPC，一般都是基于Netty来构建，比如sock-bolt。

1 Netty概述

> 说明：

引用官方的一段话：**Netty是一个高性能、异步事件驱动的网络应用框架。 基于Netty，可以快速的开发和部署高性能、 高可用的网络服务端和客户端应用**

1.1 原生NIO存在的问题

1. NIO跨平台和兼容性问题：

   NIO是底层API,它的实现依赖于操作系统针对IO操作的APIs. 所以java能在所有操作系统上实现统一的接口，并用一致的行为来操作IO是很伟大的。使用NIO会经常发现代码在Linux上正常运行，但在Windows上就会出现问题。所以编写程序，特别是NIO程序，需要在程序支持的所有操作系统上进行功能测试，否则你可能会碰到一些莫明的问题。NIO2看起来很理想，但是NIO2只支持Jdk1.7+，若你的程序在Java1.6上运行，则无法使用NIO2。另外，Java7的NIO2中没有提供DatagramSocket的支持，所以NIO2只支持TCP程序，不支持UDP程序

2. NIO对缓冲区的聚合和分散操作可能会导致内存泄露

   > **分散 Scattering 对应缓冲区写入 :** 通道 (Channel) 向 缓冲区数组 中写出数据 , 按照索引从第 0 个缓冲区 (Buffer) 开始, 依次写入数据 ;
   >
   > **聚合 Gathering 对应缓冲区读取 :** 通道 (Channel) 从 缓冲区数组 中读取数据 , 按照索引从第 0 个缓冲区 (Buffer) 开始, 依次读取数据 ;
   >
   > **分散 Scattering 与 聚合 Gathering 都是 通道 (Channel) 对 缓冲区数组 (Buffer[]) 进行读写的操作 ;**

3. NIO 的类库和 API 繁杂，使用麻烦。需要熟练掌握 Selector、ServerSocketChannel、SocketChannel、ByteBuffer等。
4. 需要具备其他的额外技能：要熟悉 Java 多线程编程，因为 NIO 编程涉及到 Reactor 模式，你必须对多线程和网络编程非常熟悉，才能编写出高质量的 NIO 程序。
5. 开发工作量和难度都非常大：例如客户端面临断连重连、网络闪断、半包读写、失败缓存、网络拥塞和异常流的处理等等。
6. JDK NIO 的 Bug：著名的epoll-bug也可能会导致无效的状态选择和100%的CPU利用率。

```
com.kkb.demo.netty.example.plain.PlainNioEchoServer
```

1.2 Netty的优点

Netty的对JDK自带的NIO的API进行封装，解决上述问题，主要特点有：

- 设计
  - 针对多种传输类型的统一接口 - 阻塞和非阻塞
  - 简单但更强大的线程模型
  - 真正的无连接的数据报套接字支持
  - 链接逻辑支持复用
- 易用性
  - 大量的 Javadoc 和 代码实例
  - 除了在 JDK 1.6 + 额外的限制。（一些特征是只支持在Java 1.7 +。可选的功能可能有额外的限制。）
- 性能
  - 比核心java api 更好的吞吐量，较低的延时
  - 资源消耗更少，这个得益于共享池和重用
  - 减少内存拷贝
- 健壮性
  - 消除由于慢，快，或重载连接产生的 OutOfMemoryError
  - 消除经常发现在 NIO 在高速网络中的应用中的不公平的读/写比
- 安全
  - 完整的 SSL / TLS 和 StartTLS 的支持
  - 运行在受限的环境例如 Applet 或 OSGI
- 社区
  - 发布的更早和更频繁
  - 社区驱动

为什么用Netty

- 快，Netty正是基于NIO实现了这种Reactor模型,Boss线程用来专门处理连接的建立，SubReactor专门用来处理IO的读写以及任务的处理。这种线程模型在充分利用CPU性能的情况下支撑大量的并发连接
- 内存使用少，网络数据传输面临着大量的对象创建和销毁，Netty主要从两个方面缓解JVM的压力
  - ByteBufAllocator对象池。池化ByteBuf实例以提高性能并最小化内存碎片，后者每次调用时都返回一个新的实例
  - 零拷贝。支持DirectBuffer的使用，通过JVM的本地调用分配内存，这可避免每次调用本地I / O操作之前（或之后）将缓冲区的内容复制到（或从）中间缓冲区
- API简单，网络编程一般都比较复杂，更面临着IO读写以及线程安全问题问题要处理，Netty针对这些问题做了大量封装，使API更简单易用。基于事件模式，对网络事件进行串行化处理，在保证高效的同时，又降低了编程的复杂度
- Netty非常稳定，一般我们遇到的NIO的select空转，TCP断线重连，keep-alive检测等问题，Neety都已解决

1.3 Netty的介绍

1. `Netty` 是由 `JBOSS` 提供的一个 `Java` 开源框架，现为 `Github` 上的独立项目。
2. **Netty 是一个异步的、基于事件驱动的网络应用框架，用以快速开发高性能、高可靠性的网络 IO 程序。**
3. **Netty 主要针对在 TCP 协议下，面向 Client 端的高并发应用，或者 Peer-to-Peer 场景下的大量**

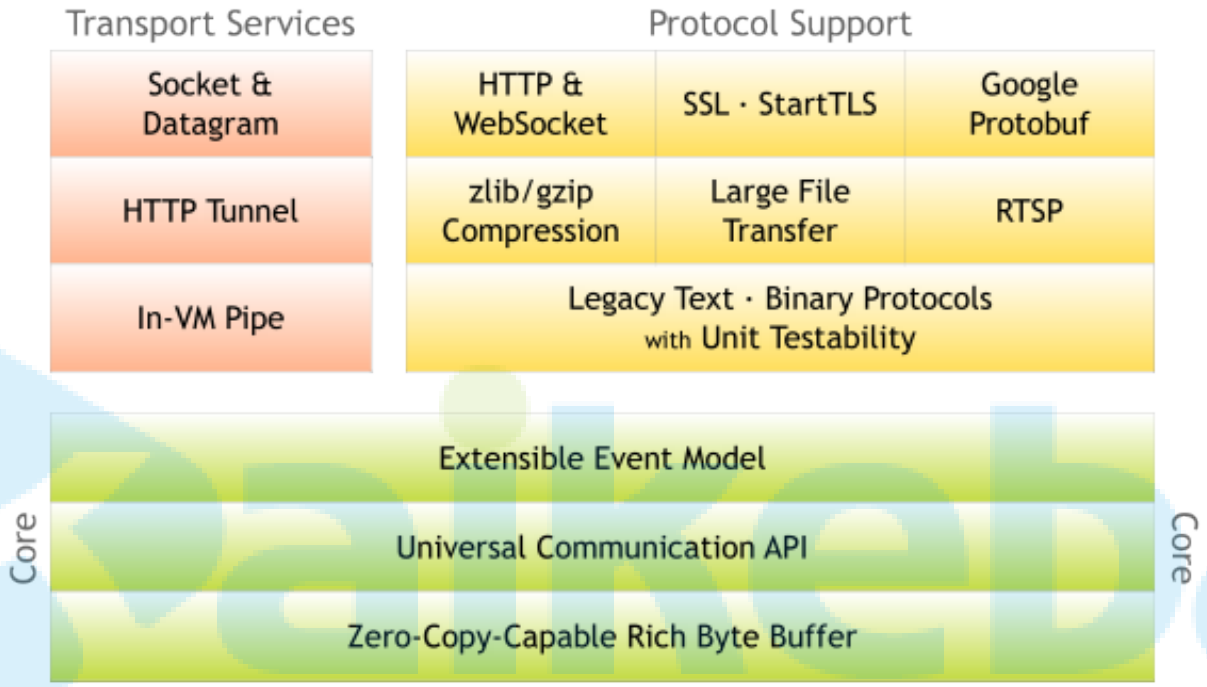
数据持续传输的应用。

4. Netty 本质是一个 NIO 框架，适用于服务器通讯相关的多种应用场景。

1.4 Netty官网说明

官网：https://netty.io/

Netty is an asynchronous event-driven network application framework for rapid development of maintainable high performance protocol servers & clients.



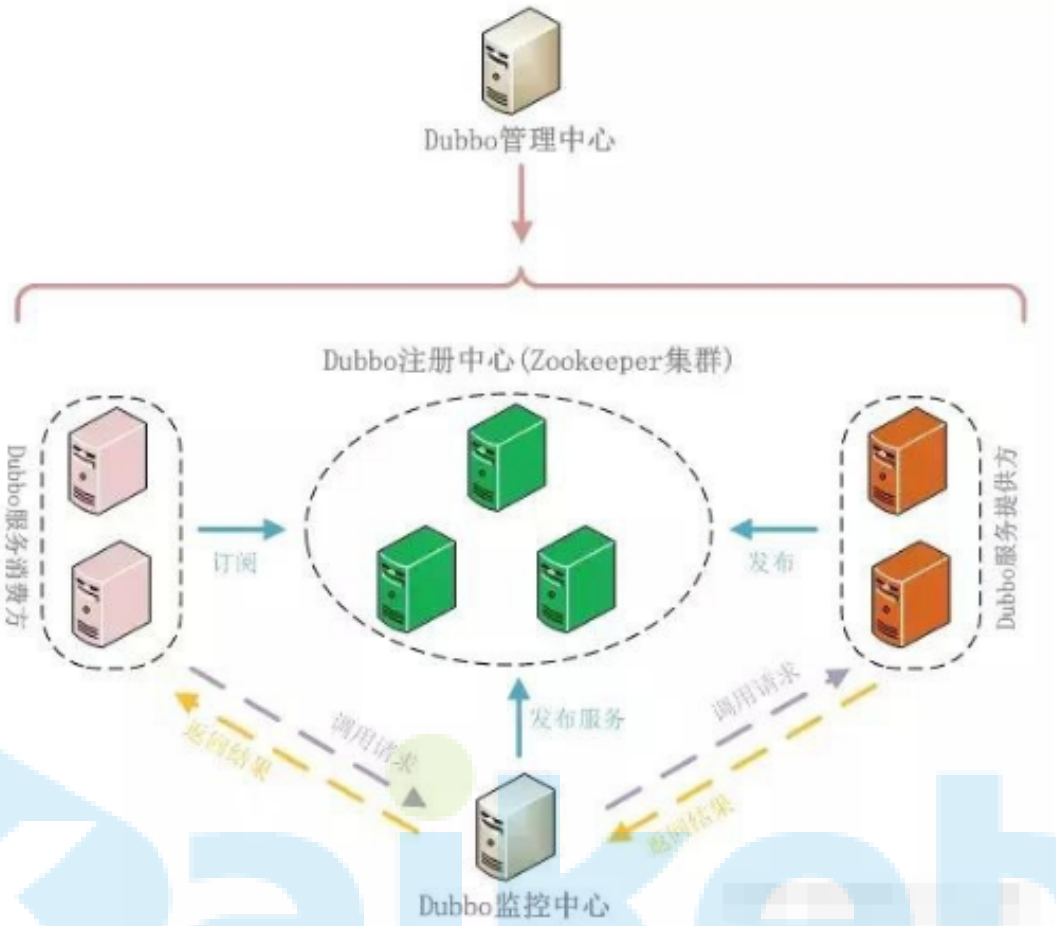
Netty 版本说明

1. Netty 版本分为 Netty 3.x 和 Netty 4.x、Netty 5.x
2. 因为 `Netty 5` 出现重大 `bug`，已经被官网废弃了，目前推荐使用的是 `Netty 4.x` 的稳定版本
3. 目前在官网可下载的版本 `Netty 3.x`、`Netty 4.0.x` 和 `Netty 4.1.x`
4. 在本次课程中，基于 4.1.58.Final 版本
5. Netty 下载地址：https://bintray.com/netty/downloads/netty/

1.5 Netty的应用场景

1.5.1 互联网行业

1. 互联网行业：在分布式系统中，各个节点之间需要远程服务调用，高性能的 `RPC` 框架必不可少，`Netty` 作为异步高性能的通信框架，往往作为基础通信组件被这些 `RPC` 框架使用。
2. 典型的应用有：阿里分布式服务框架 Dubbo 的 RPC 框 架使用 Dubbo 协议进行节点间通信，Dubbo 协议默认使用 Netty 作为基础通信组件，用于实现各进程节点之间的内部通信。

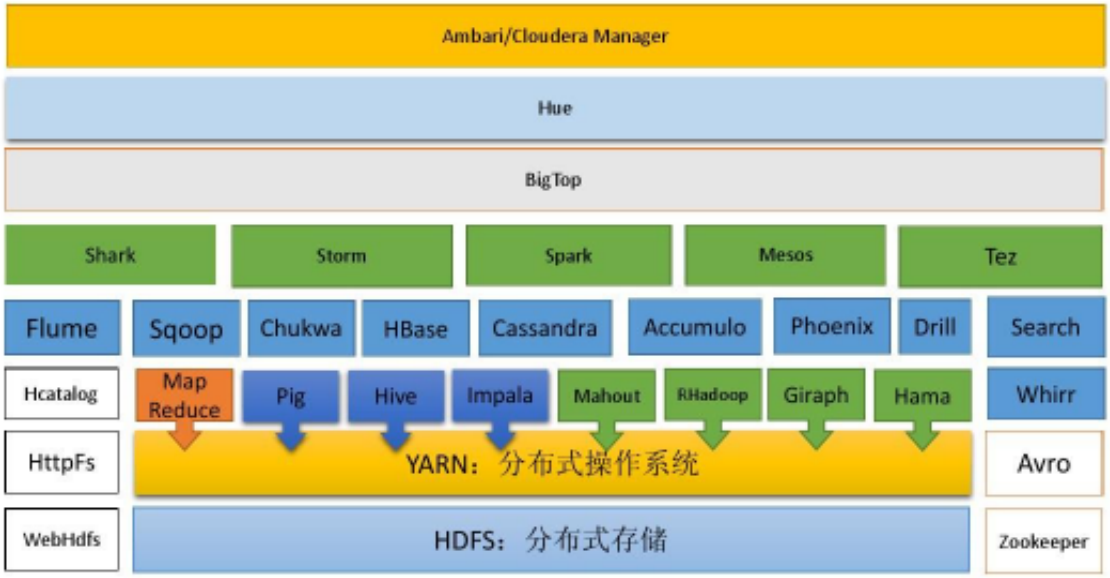


1.5.2 游戏行业

1. 无论是手游服务端还是大型的网络游戏，Java 语言得到了越来越广泛的应用。
2. `Netty` 作为高性能的基础通信组件，提供了 `TCP/UDP` 和 `HTTP` 协议栈，方便定制和开发私有协议栈，账号登录服务器。
3. 地图服务器之间可以方便的通过 Netty 进行高性能的通信。

1.5.3 大数据领域

1. 经典的 `Hadoop` 的高性能通信和序列化组件 `Avro` 的 `RPC` 框架，默认采用 `Netty` 进行跨界点通信。
2. 它的 `NettyService` 基于 `Netty` 框架二次封装实现。



1.5.4 其它开源项目使用到 Netty

网址：https://netty.io/wiki/related-projects.html

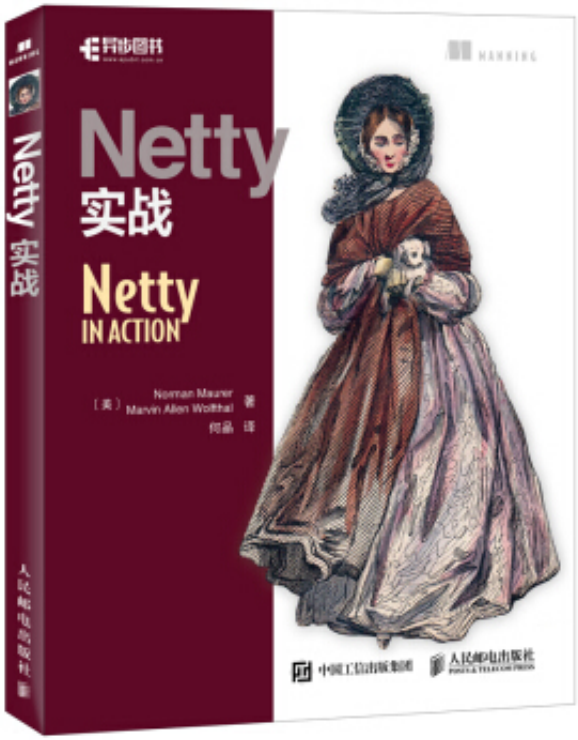
Related projects

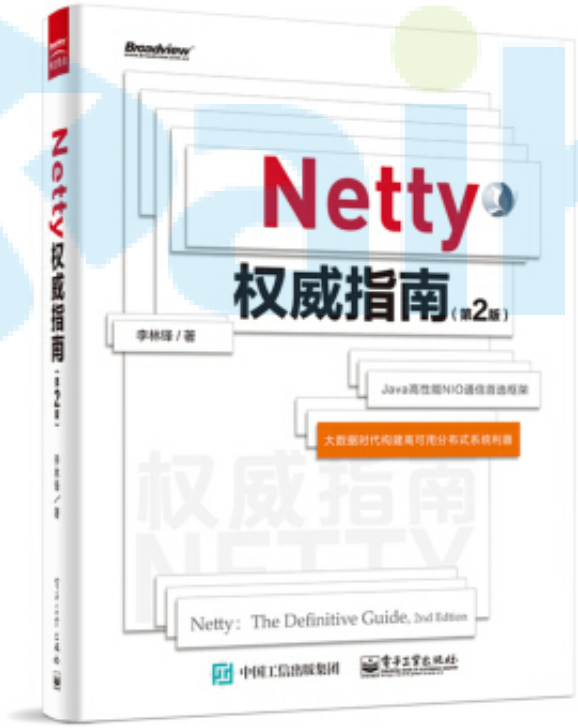
Did you know this page is automatically generated from a Github Wiki page? You can improve it by yourself here!

Make sure the list contains only open source projects documented in English and the list is sorted alphabetically. Remove unmaintained projects (no commit activity for last 12 months.)

- Akka is a Scala-based platform that provides simpler scalability, fault-tolerance, concurrency, and remoting through the actor model and software transactional
- Apache BookKeeper is a scalable, fault-tolerant, and low-latency log storage.
- Apache Cassandra is a column oriented distributed database.
- Apache Flink is a distributed, stateful stream processing framework.
- Apache James Server is a modular e-mail server platform that integrates SMTP, POP3, IMAP, and NNTP.
- Apache Pulsar is an open-source distributed pub-sub messaging system.
- Apache Spark is a fast and general purpose cluster compute framework, commonly used for "Big Data" applications.
- Apache Tajo is a distributed, fault-tolerance, low-latency, and high throughput SQL engine that provides ETL features and ad-hoc query processing on large-scale data sets.
- Arquillian is an innovative in-container testing platform for the JVM
- Async HTTP Client is a simple-to-use library that allows you to execute HTTP requests and process the HTTP responses asynchronously.
- Atomix is an event-driven framework for coordinating fault-tolerant distributed systems built on the Raft consensus algorithm.
- BungeeCord is the de facto proxy solution for combining multiple Minecraft servers into a cloud / hub system.
- ClusterVAS - A docker based node manager and an orchestrator SDK for remotely managing OpenVAS instances (communication layer is built with Netty Agents based on Netty)
- Copycat is a fault-tolerant state machine replication framework built on the Raft consensus algorithm.
- Couchbase is a distributed NoSQL document-oriented database that is optimized for interactive applications.
- Elastic Search is a distributed RESTful search engine built on top of Lucene.
- Eucalyptus is a software infrastructure for implementing on-premise cloud computing using an organization's own IT infrastructure, without modification, special-purpose hardware or reconfiguration.
- Finagle is an extensible RPC system for the JVM, used to construct high-concurrency servers.
- Forest is a general purpose friend-to-friend platform.
- Gatling is an asynchronous and efficient stress tool developed with Netty and Akka.
- gRPC is a high performance, open-source universal RPC framework.
- Hammersmith is a pure asynchronous MongoDB driver for Scala
- Higgs is a high performance, message oriented network library.
- Holmes is a Java application that implements DLNA/UPnP protocol for playing videos, music, pictures and podcasts (RSS) to compatible devices.
- HornetQ is a project to build a multi-protocol, embeddable, very high performance, clustered, asynchronous messaging system.
- http-client is a high performance and throughput oriented HTTP client library.

1.6 Netty 的学习资料参考





2 Netty高性能架构设计

2.1 线程模型基本介绍

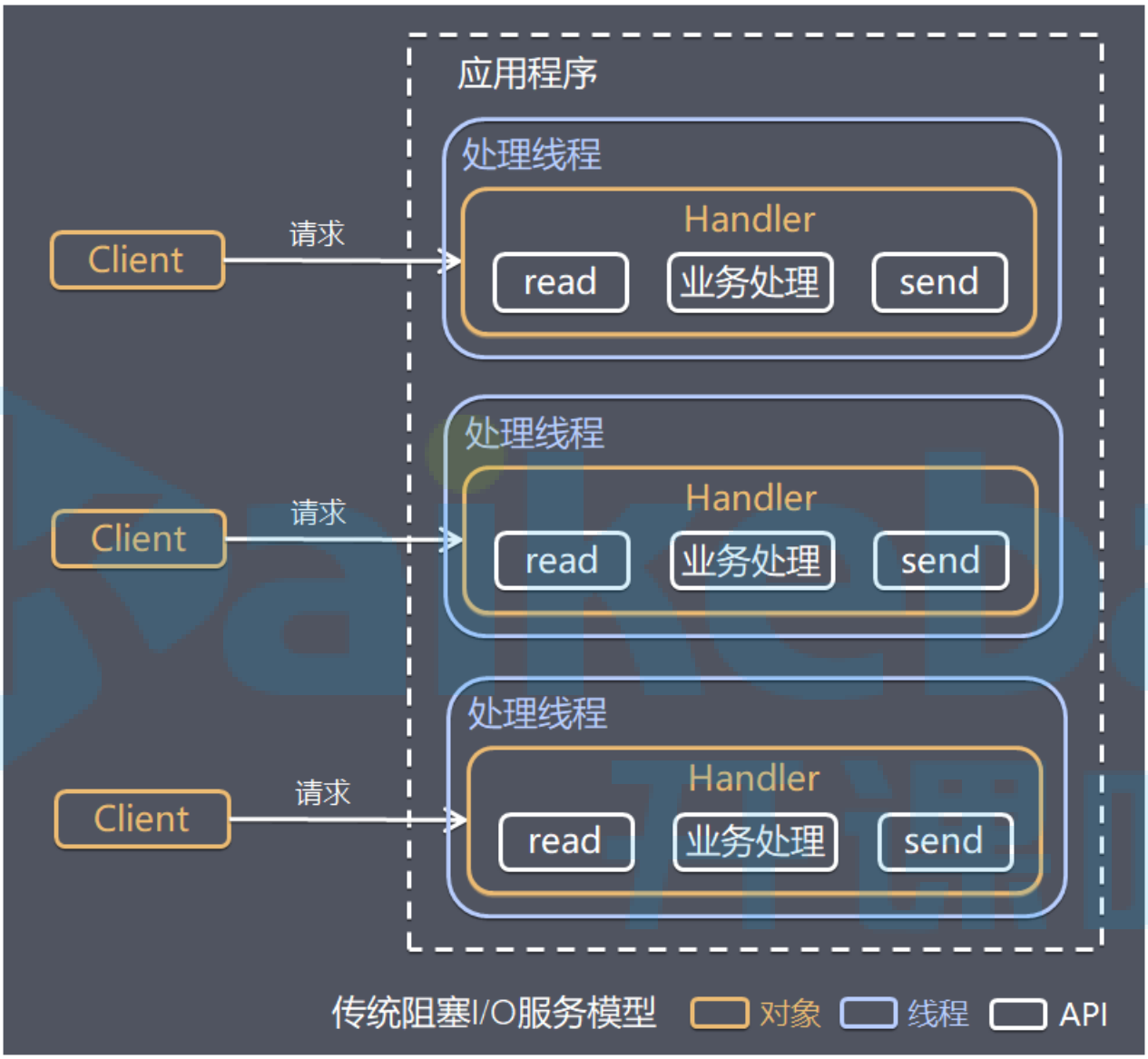
不同的线程模型对程序的性能有很大的影响，Netty是建立在Reactor模型的基础上，要搞清Netty的线程模型，需要了解一目前常见线程模型的一些概念。

目前存在的线程模型有：

- 传统阻塞IO服务模型
- Reactor模型
- Proactor模型

Netty的整体架构，基于了一个著名的模式——Reactor模式。Reactor模式，是高性能网络编程的必知必会模式（Netty 主要基于主从 Reactor 多线程模型做了一定的改进，其中主从 Reactor 多线程模型有多个 Reactor）

2.2 传统阻塞IO服务模型



> 黄色的框表示对象、蓝色的框表示线程、白色的框表示方法（API）

采用阻塞IO模型获取输入的数据。 每个连接需要独立的完成数据的输入，业务的处理，数据返回。当并发数大的时候，会创建大量的线程，占用系统资源，如果连接创建后，当前线程没有数据可读，会阻塞，造成线程资源浪费。

最最原始的网络编程思路就是服务器用一个while循环，不断监听端口是否有新的套接字连接，如果有，那么就调用一个处理函数处理，类似：

```
while(true){

socket = accept();

handle(socket)

}
```

这种方法的最大问题是无法并发，效率太低，如果当前的请求没有处理完，那么后面的请求只能被阻塞，服务器的吞吐量太低。

之后，想到了使用多线程，也就是很经典的connection per thread，每一个连接用一个线程处理，类似：

```
package com.kkb.demo.netty.example.base;

import java.io.IOException;
import java.net.ServerSocket;
import java.net.Socket;

public class BasicModel implements Runnable {
    @Override
    public void run() {
        try {
            ServerSocket ss =
                    new ServerSocket(7007);
            while (!Thread.interrupted())
                new Thread(new Handler(ss.accept())).start();
            //创建新线程来handle
            // or, single-threaded, or a thread pool
        } catch (IOException ex) { /* ... */ }
    }

    static class Handler implements Runnable {
        final Socket socket;
        Handler(Socket s) { socket = s; }
        public void run() {
            try {
                byte[] input = new byte[1024];
                socket.getInputStream().read(input);
                byte[] output = process(input);
                socket.getOutputStream().write(output);
            } catch (IOException ex) { /* ... */ }
        }
        private byte[] process(byte[] input) {
            byte[] output=null;
            /* ... */
            return output;
        }
```

```
    }
}
```

对于每一个请求都分发给一个线程，每个线程中都独自处理上面的流程。

tomcat服务器的早期版本确实是这样实现的。

多线程并发模式，一个连接一个线程的优点是：

一定程度上极大地提高了服务器的吞吐量，因为之前的请求在read阻塞以后，不会影响到后续的请求，因为他们在不同的线程中。这也是为什么通常会讲“一个线程只能对应一个socket”的原因。另外有个问题，如果一个线程中对应多个socket连接不行吗？语法上确实可以，但是实际上没有用，每一个socket都是阻塞的，所以在一个线程里只能处理一个socket，就算accept了多个也没用，前一个socket被阻塞了，后面的是无法被执行到的。

多线程并发模式，一个连接一个线程的缺点是：

缺点在于资源要求太高，系统中创建线程是需要比较高的系统资源的，如果连接数太高，系统无法承受，而且，线程的反复创建-销毁也需要代价。

模型特点

1. 采用阻塞 IO 模式获取输入的数据
2. 每个连接都需要独立的线程完成数据的输入，业务处理，数据返回

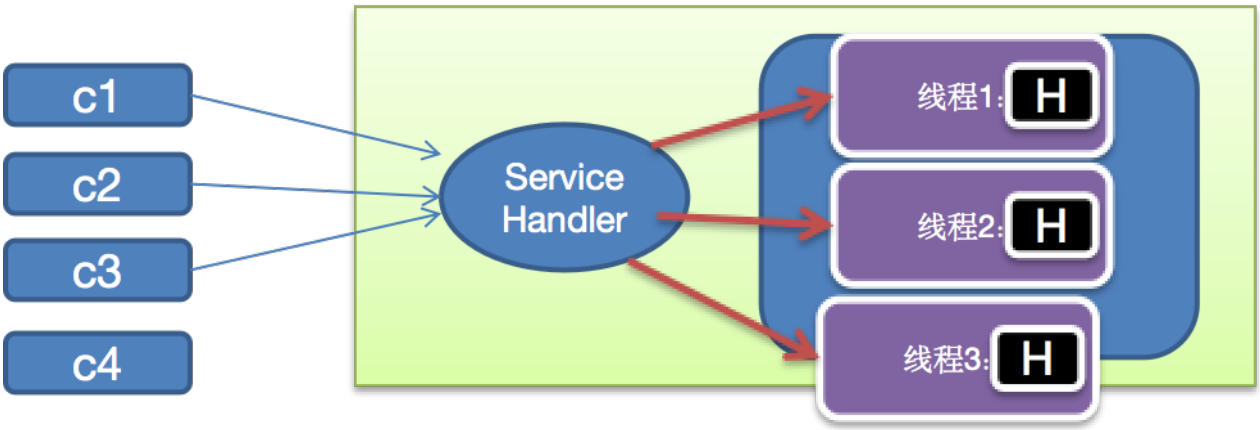
问题分析

1. 当并发数很大，就会创建大量的线程，占用很大系统资源
2. 连接创建后，如果当前线程暂时没有数据可读，该线程会阻塞在 read 操作，造成线程资源浪费

2.3 Reactor模式

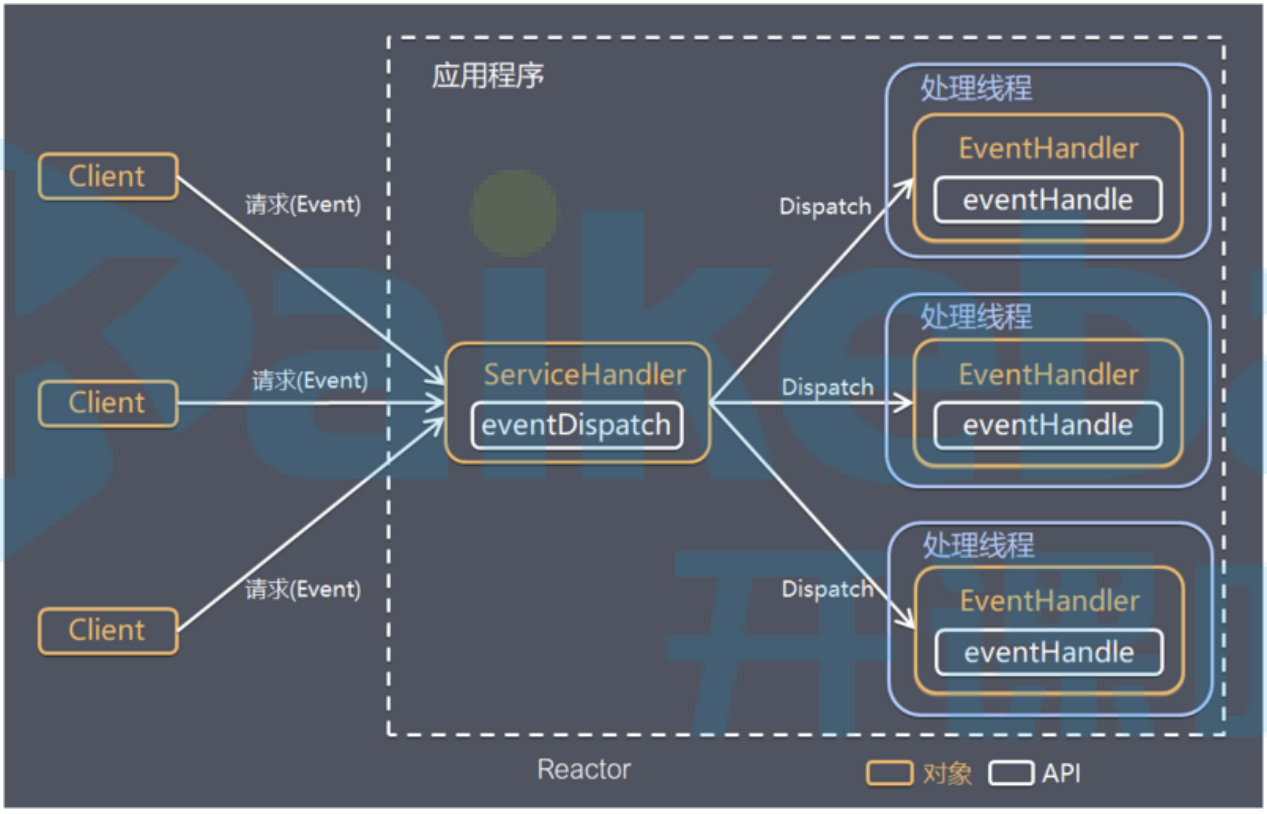
2.3.1 针对传统阻塞 I/O 服务模型的 2 个缺点，解决方案：

1. 基于 I/O 复用模型：多个连接共用一个阻塞对象，应用程序只需要在一个阻塞对象等待，无需阻塞等待所有连接。当某个连接有新的数据可以处理时，操作系统通知应用程序，线程从阻塞状态返回，开始进行业务处理 Reactor 对应的叫法：
    1. 反应器模式
    2. 分发者模式（Dispatcher）
    3. 通知者模式（notifier）
2. 基于线程池复用线程资源：不必再为每个连接创建线程，将连接完成后的业务处理任务分配给线程进行处理，一个线程可以处理多个连接的业务。



2.3.2 Reactor模型

IO多路复用 + 线程池 = Reactor模型



对上图说明：

1. Reactor 模式，通过一个或多个输入同时传递给服务处理器的模式（基于事件驱动）
2. 服务器端程序处理传入的多个请求,并将它们同步分派到相应的处理线程，因此 Reactor 模式也叫 Dispatcher 模式
3. Reactor 模式使用 IO 复用监听事件，收到事件后，分发给某个线程（进程），这点就是网络服务器高并发处理关键

2.3.3 Reactor模式中核心组成

1. **Reactor**：Reactor 在一个单独的线程中运行，负责监听和分发事件，分发给适当的处理程序来对 IO 事件做出反应。
2. **Handlers**：处理程序执行 I/O 事件要完成的实际事件，类似于客户想要与之交谈的公司中的实际官员。Reactor 通过调度适当的处理程序来响应 I/O 事件，处理程序执行非阻塞操作。

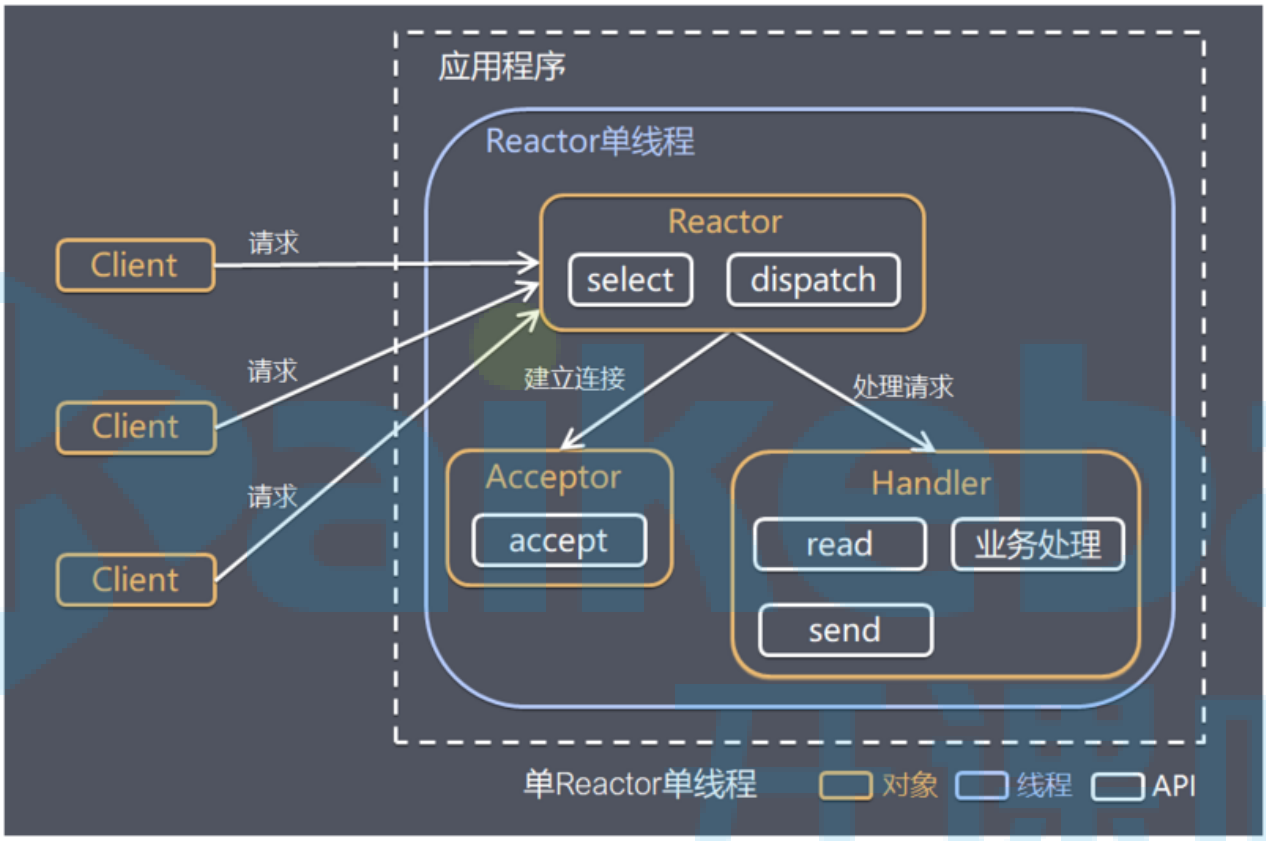
2.3.4 Reactor模式分类

根据 Reactor 的数量和处理资源池线程的数量不同，有 3 种典型的实现

1. 单 `Reactor` 单线程
2. 单 `Reactor` 多线程
3. 主从 Reactor 多线程

2.4 单Reactor单线程

2.4.1 原理图



1. Select 是前面 I/O 复用模型介绍的标准网络编程 API，可以实现应用程序通过一个阻塞对象监听多路连接请求
2. Reactor 对象通过 Select 监控客户端请求事件，收到事件后通过 Dispatch 进行分发
3. 如果是建立连接请求事件，则由 `Acceptor` 通过 `Accept` 处理连接请求，然后创建一个 `Handler` 对象处理连接完成后的后续业务处理
4. 如果不是建立连接事件，则 `Reactor` 会分发调用连接对应的 `Handler` 来响应
5. Handler 会完成 Read → 业务处理 → Send 的完整业务流程

服务器端用一个线程通过多路复用搞定所有的 IO 操作（包括连接，读、写等），编码简单，清晰明了，但是如果客户端连接数量较多，将无法支撑，前面的 NIO 案例就属于这种模型。

java的NIO模式的Selector网络通讯，其实就是一个简单的Reactor模型。可以说是Reactor模型的朴素原型。

实际上的Reactor模式，是基于java nio的，在他的基础上，抽象出来两个组件——Reactor和Handler两个组件：

1) Reactor:负责响应IO事件，当检测到一个新的事件，将其发送给相应的Handler去处理；新的事件包含连接建立就绪、读就绪、写就绪等。

2)Handler:将自身（handler）与事件绑定，负责事件的处理，完成channel的读入，完成处理业务逻辑后，负责将结果写出channel。

2.4.2 方案优缺点分析

1. 优点：模型简单，没有多线程、进程通信、竞争的问题，全部都在一个线程中完成
2. 缺点：性能问题，只有一个线程，无法完全发挥多核 `CPU` 的性能。`Handler`在处理某个连接上的业务时，整个进程无法处理其他连接事件，很容易导致性能瓶颈
3. 缺点：可靠性问题，线程意外终止，或者进入死循环，会导致整个系统通信模块不可用，不能接收和处理外部消息，造成节点故障
4. 使用场景：客户端的数量有限，业务处理非常快速，比如 Redis 在业务处理的时间复杂度 O(1) 的情况

> Redis内部就是这种模型

2.4.3 单线程Reactor的参考代码

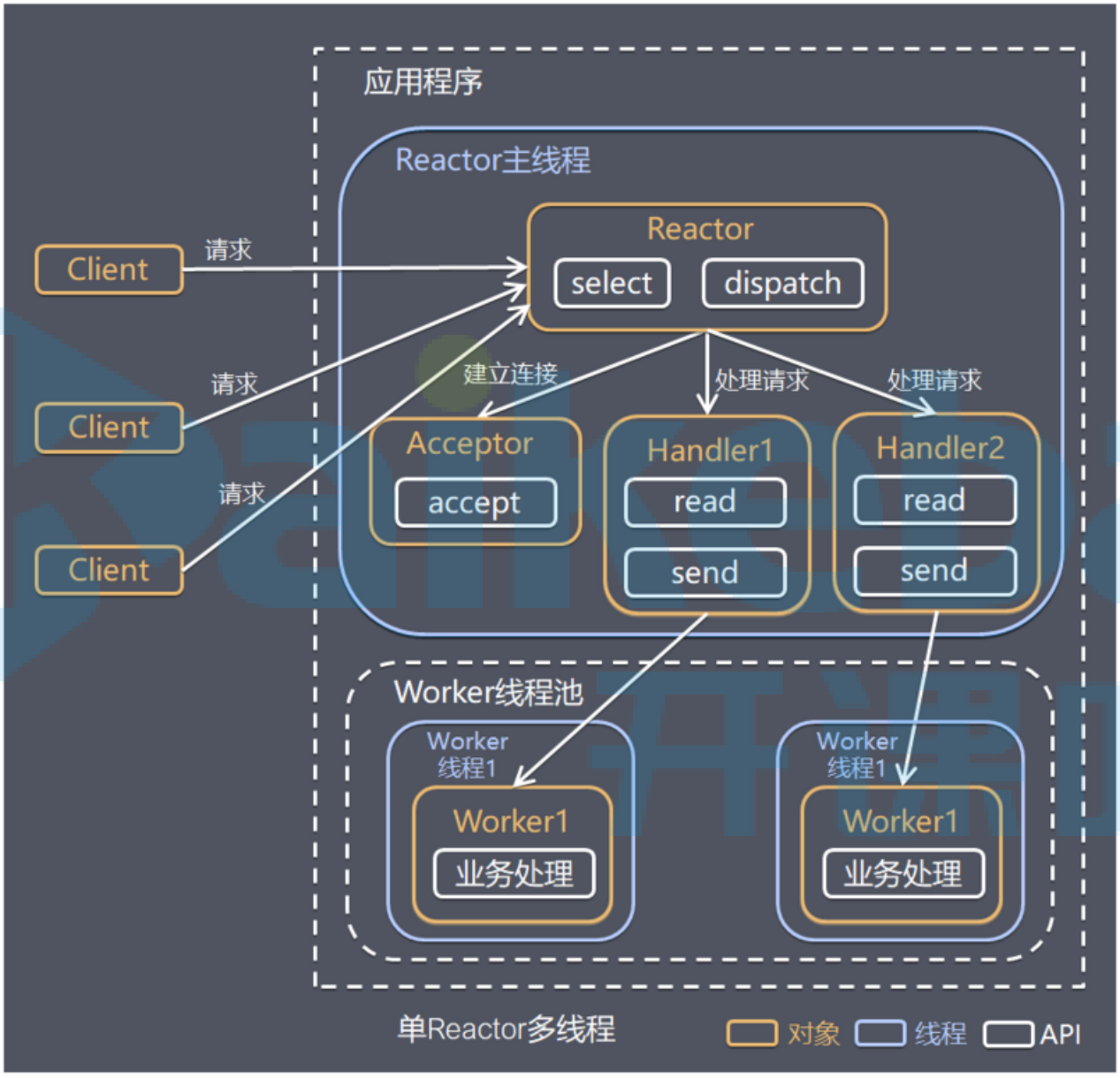
```
import java.io.IOException;
import java.net.InetSocketAddress;
import java.nio.ByteBuffer;
import java.nio.channels.SelectionKey;
import java.nio.channels.Selector;
import java.nio.channels.ServerSocketChannel;
import java.nio.channels.SocketChannel;
import java.util.Iterator;
import java.util.Set;

public class SimpleReactor implements Runnable {
    final Selector selector;
    final ServerSocketChannel serverSocket;

    SimpleReactor(int port) throws IOException { //Reactor初始化
        selector = Selector.open();
        serverSocket = ServerSocketChannel.open();
        serverSocket.socket().bind(new InetSocketAddress(port));
        //非阻塞
        serverSocket.configureBlocking(false);

        //分步处理,第一步,接收accept事件
        SelectionKey sk =
                serverSocket.register(selector, SelectionKey.OP_ACCEPT);
```

```
        //attach callback object, Acceptor
        sk.attach(new Acceptor());
    }

    public void run() {
        try
        {
            while (!Thread.interrupted())
            {
                selector.select();
                Set selected = selector.selectedKeys();
                Iterator it = selected.iterator();
                while (it.hasNext()) {
                    //Reactor负责dispatch收到的事件
                    dispatch((SelectionKey) (it.next()));
                }
                selected.clear();
            }
        } catch (IOException ex)
        { /* ... */ }
    }

    void dispatch(SelectionKey k) {
        Runnable r = (Runnable) (k.attachment());
        //调用之前注册的callback对象
        if (r != null)
        {
            r.run();
        }
    }

    // inner class
    class Acceptor implements Runnable {
        public void run()
        {
            try
            {
                SocketChannel channel = serverSocket.accept();
                if (channel != null)
                    new SimpleHandler(selector, channel);
            } catch (IOException ex)
            { /* ... */ }
        }
    }

    public static void main(String[] args) throws IOException {
        SimpleReactor simpleReactor = new SimpleReactor(7002);
        new Thread(simpleReactor).start();
    }
```

```
}
```

2.5 单Reactor多线程

2.5.1 原理图



图中多线程体现在两个部分：

- Reactor主线程

Reactor通过select监听客户请求，如果是连接请求事件，则由Acceptor处理连接，如果是其他请求，则由dispatch找到对应的Handler，这里的**Handler只负责响应事件，读取和响应**，会将具体的业务处理交由Worker线程池处理。

- Worker线程池

Worker线程池会分配独立线程完成真正的业务，并将结果返回给Handler，Handler收到响应后，通过send将结果返回给客户端。

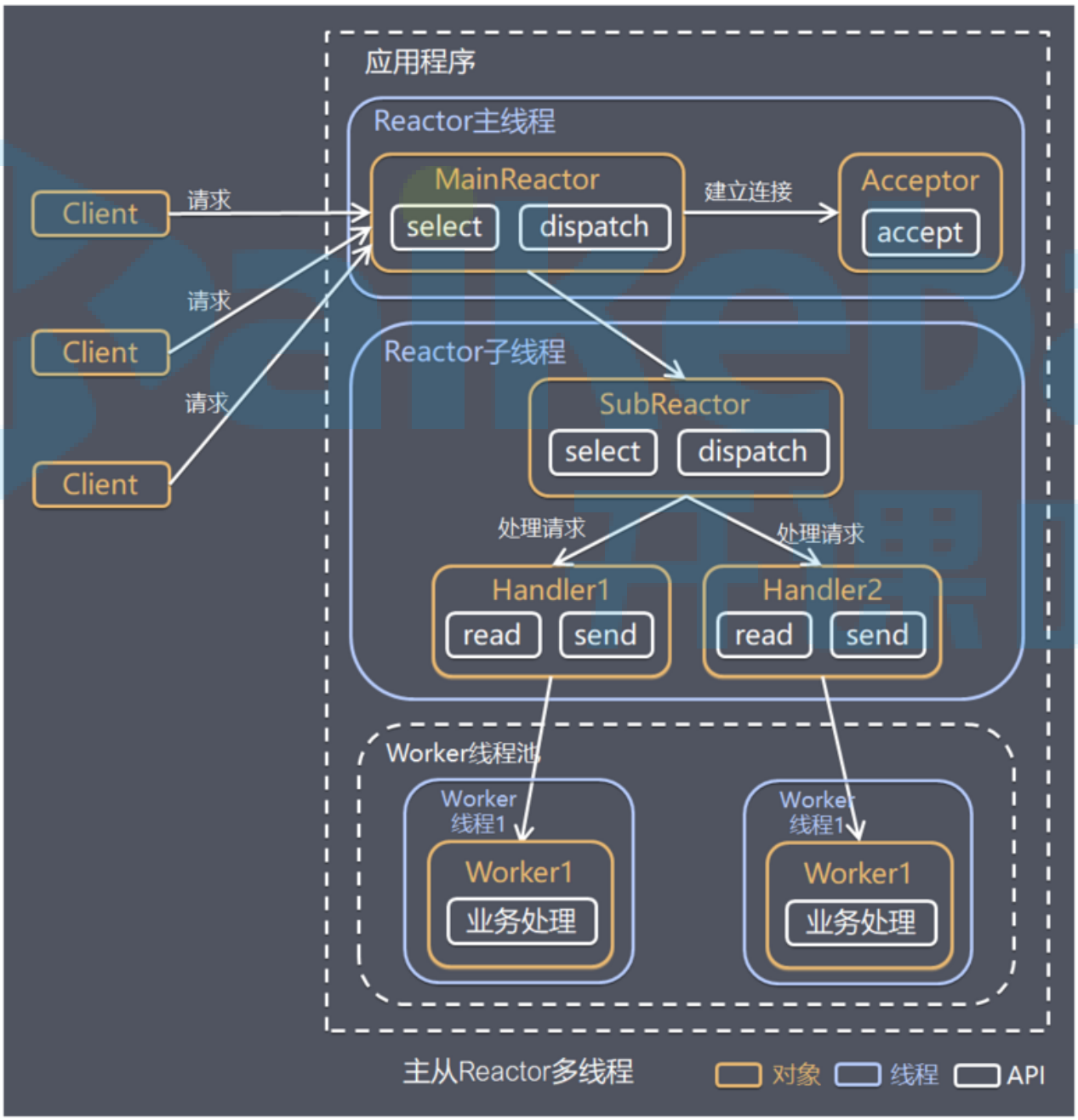
这里Reactor处理所有的事件监听和响应，高并发情景下容易出现性能瓶颈。

2.5.2 方案优缺点分析

1. 优点:可以充分的利用多核 cpu 的处理能力
2. 缺点:多线程数据共享和访问比较复杂，**Reactor 处理所有的事件的监听和响应，在单线程运行，在高并发场景容易出现性能瓶颈**。

2.6 主从Reactor多线程

2.6.1 原理图



这种模式是对单Reactor的改进，由原来单Reactor改成了Reactor主线程与Reactor子线程。

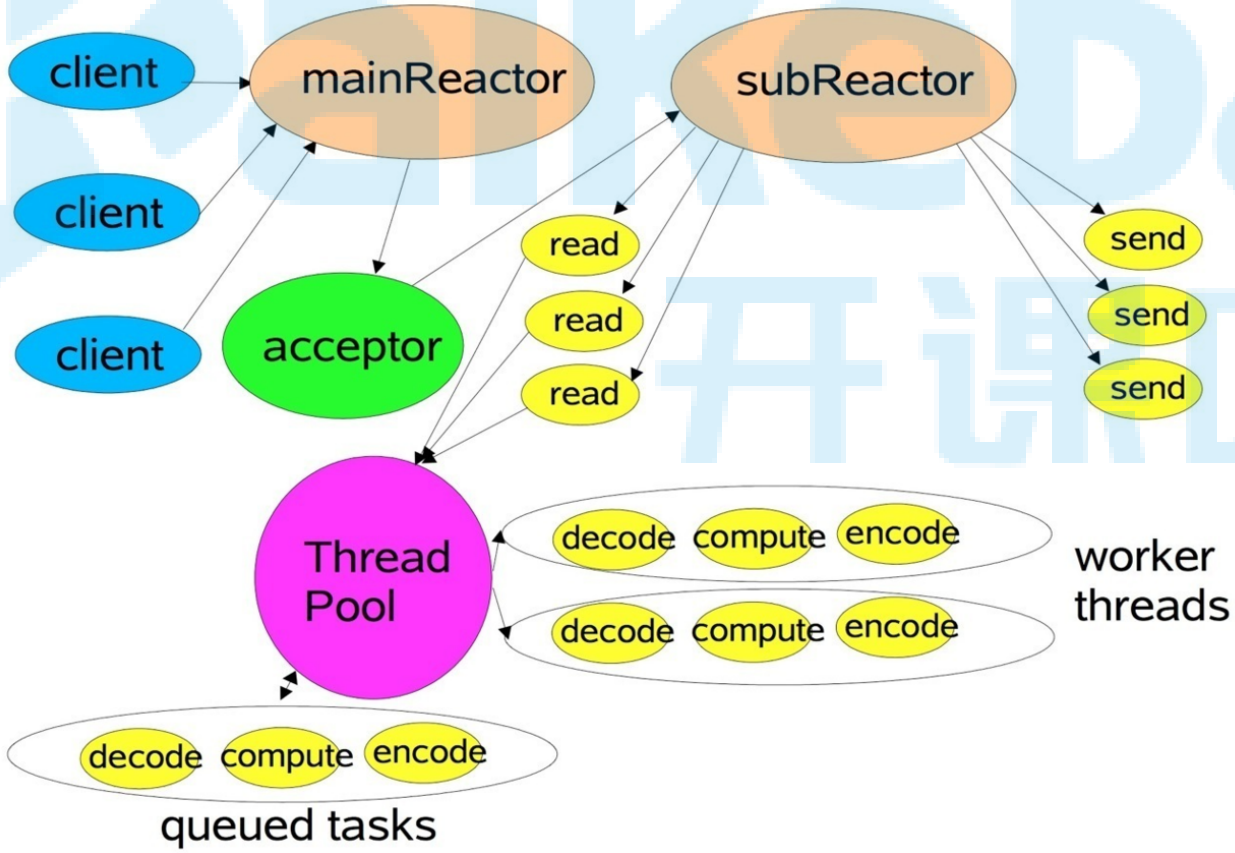
1. Reactor 主线程 MainReactor 对象通过 select 监听**连接事件**，收到事件后，通过 Acceptor 处理

连接事件
2. 当 Acceptor 处理**连接事件**后，MainReactor 将连接分配给 SubReactor
3. `subreactor` 将连接加入到连接队列进行监听，并创建 `handler` 进行各种事件处理
4. 当有**新事件read、send发生时**，`subreactor` 就会调用对应的 `handler` 处理
5. `handler` 通过 `read` 读取数据，分发给后面的 `worker` 线程处理
6. `worker` 线程池分配独立的 `worker` 线程进行业务处理，并返回结果
7. `handler` 收到响应的结果后，再通过 `send` 将结果返回给 `client`
8. Reactor 主线程可以对应多个 Reactor 子线程，即 MainRecator 可以关联多个 SubReactor

2.6.2 方案优缺点说明

1. 优点：父线程与子线程的数据交互简单职责明确，**父线程只需要接收新连接，子线程完成后续的业务处理**。
2. 优点：父线程与子线程的数据交互简单，`Reactor` 主线程只需要把新连接传给子线程，子线程无需返回数据。
3. 缺点：编程复杂度较高
4. 结合实例：这种模型在许多项目中广泛使用，包括 Nginx 主从 Reactor 多进程模型，Memcached 主从多线程，Netty 主从多线程模型的支持

2.6.3 Scalable IO in Java 对 Multiple Reactors 的原理图解：



2.7 Reactor 模式小结

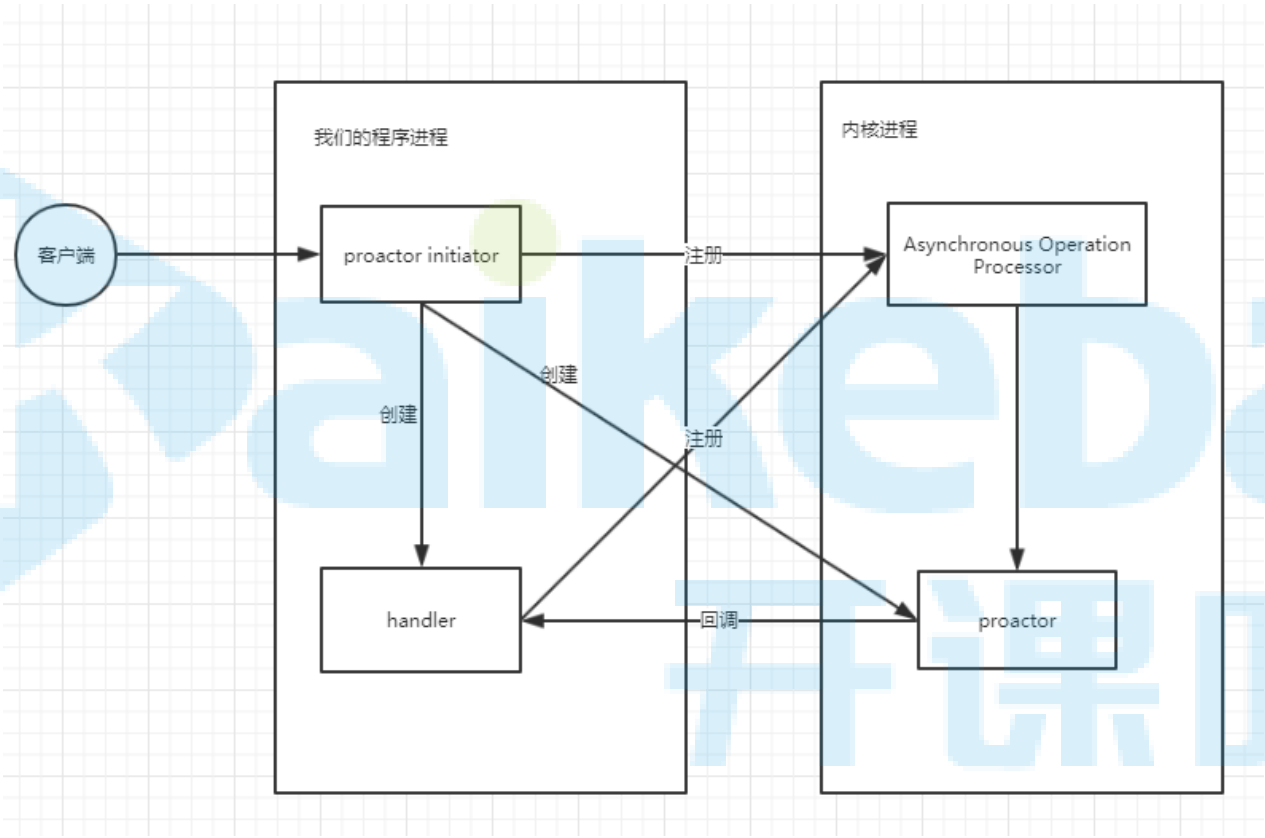
Reactor 模式具有如下的优点

1. 响应快，不必为单个同步时间所阻塞，虽然 Reactor 本身依然是同步的
2. 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
3. 扩展性好，可以方便的通过增加 Reactor 实例个数来充分利用 CPU 资源
4. 复用性好，Reactor 模型本身与具体事件处理逻辑无关，具有很高的复用性

2.8 Proactor模型

Proactor用于异步IO，而Reactor用于同步IO

也就是我们不必等待I/O数据准备好也就是内核缓存已经读数据到用户空间。这一切都有内核来帮我们搞定，数据准备好了之后就通知Proactor，然后Proactor就调用相应的Handler进行业务处理。相对于Reactor省去了遍历事件通知队列selector 的代价。

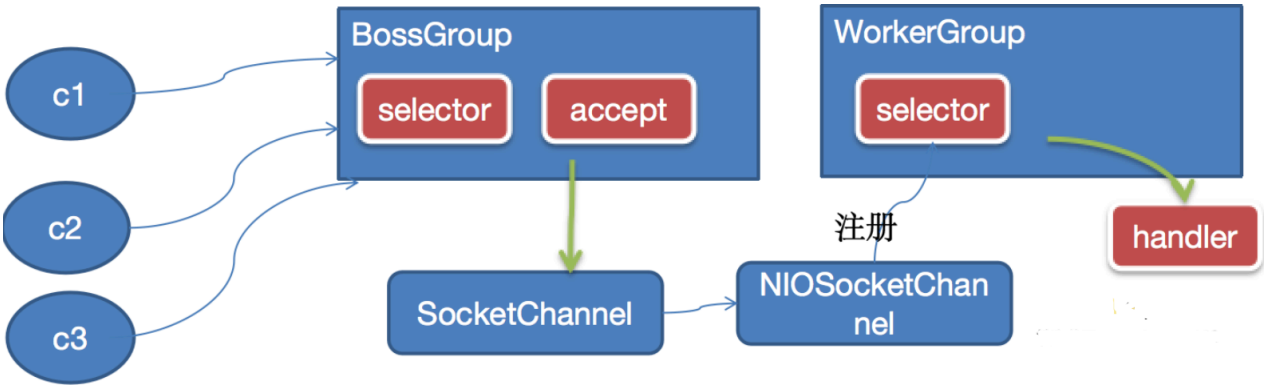


Proactor模型在理论上是比Reactor模型性能更好，但是因为依赖于操作系统的非阻塞异步模型，而linux的非阻塞异步模型还不完善，并没有真正的实现Proactor模型，而是epoll模拟出Proactor模型，所以还是以Reactor为主。

2.9 Netty模型

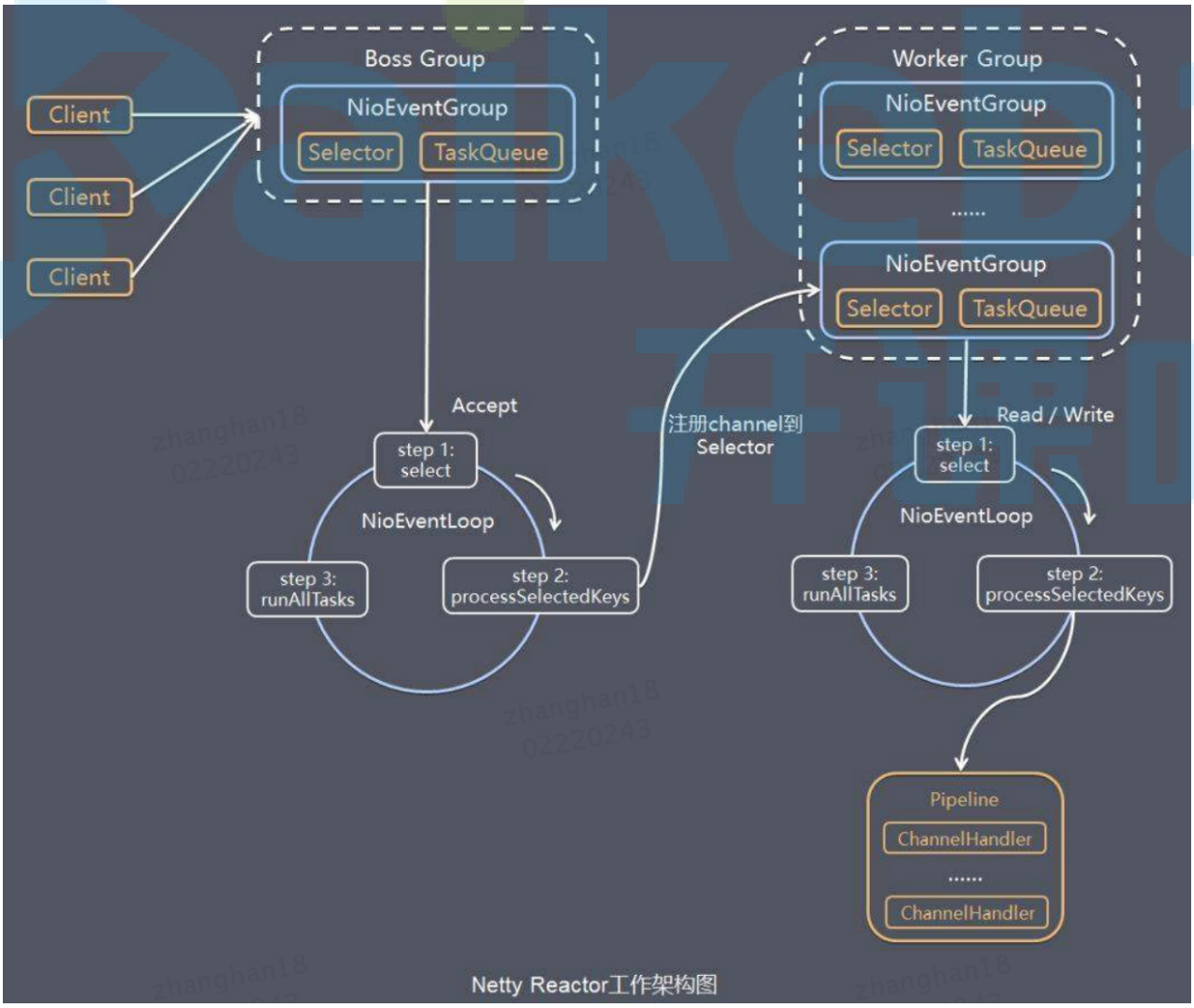
2.9.1 工作原理-简版

Netty 主要基于主从 Reactor多线程模型做了一定的改进，其中主从 Reactor 多线程模型有多个 Reactor



1. BossGroup 线程维护 Selector，只关注 Accecpt
2. 当接收到 Accept 事件，获取到对应的 SocketChannel，封装成 NIOScoketChannel 并注册到 Worker 线程（事件循环），并进行维护
3. 当 Worker 线程监听到 Selector 中通道发生自己感兴趣的事件后，就进行处理（由 handler），注意 handler 已经加入到通道

2.9.2 工作原理-详版



1. Netty 抽象出两组线程池 **BossGroup 专门负责接收客户端的连接，WorkerGroup 专门负责网络的读写**
2. BossGroup 和 WorkerGroup 类型都是 NioEventLoopGroup(threads)
3. NioEventLoopGroup 相当于一个事件循环组，这个组中含有多个事件循环，每一个事件循环是 NioEventLoop(thread)
4. NioEventLoop 表示一个不断循环的执行处理任务的线程，每个 NioEventLoop 都有一个 Selector，用于监听绑定在其上的 socket 的网络通讯，NioEventLoop 内部采用串行化设计，从消息的读取->解码->处理->编码->发送，始终由 IO 线程 NioEventLoop 负责。
5. NioEventLoopGroup 可以有多个线程，即可以含有多个 NioEventLoop
6. 每个 BossNioEventLoop 循环执行的步骤有 3 步
    - 轮询 accept 事件
    - 处理 `accept` 事件，与 `client` 建立连接，生成 `NioScocketChannel`，并将其注册到某个 `worker` `NIOEventLoop` 上的 `Selector`
    - 处理任务队列的任务，即 runAllTasks
7. 每个 Worker NIOEventLoop 循环执行的步骤
    - 轮询 `read`，`write` 事件
    - 处理 `I/O` 事件，即 `read`，`write` 事件，在对应 `NioScocketChannel` 处理
    - 处理任务队列的任务，即 runAllTasks
8. 每个 Worker NIOEventLoop 处理业务时，会使用 pipeline（管道），pipeline 中包含了 channel，即通过 pipeline 可以获取到对应通道，管道中维护了很多的处理器

> NioEventLoopGroup 下包含多个 NioEventLoop
>
> - 每个 NioEventLoop 中包含有一个 Selector，一个 taskQueue
> - 每个 NioEventLoop 的 Selector 上可以注册监听多个 NioChannel
> - 每个 `NioChannel` 只会绑定在唯一的 `NioEventLoop` 上
> - 每个 NioChannel 都绑定有一个自己的 ChannelPipeline

2.9.3 NioEventLoop

NioEventLoop 介绍

NioEventLoop 是 Netty 的 Reactor 线程，它的职责如下：

1. 作为服务端 Acceptor 线程，负责处理客户端的请求接入；
2. 作为客户端 Connecor 线程，负责注册监听连接操作位，用于判断异步连接结果；
3. 作为 IO 线程，监听网络读操作位，负责从 SocketChannel 中读取报文；
4. 作为 IO 线程，负责向 SocketChannel 写入报文发送给对方，如果发生写半包，会自动注册监听写事件，用于后续继续发送半包数据，直到数据全部发送完成；
5. 作为定时任务线程，可以执行定时任务，例如链路空闲检测和发送心跳消息等；
6. 作为线程执行器可以执行普通的任务线程（Runnable）。

NioEventLoop 继承 SingleThreadEventExecutor，这就意味着它实际上是一个线程个数为 1 的线程池。

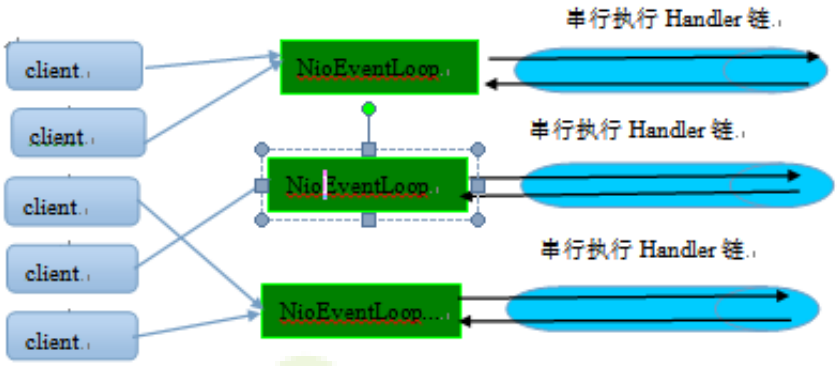
对于用户而言，直接调用 NioEventLoop 的 execute(Runnable task) 方法即可执行自定义的 Task。

NioEventLoop(thread) 设计原理

我们知道当系统在运行过程中，如果频繁的进行线程上下文切换，会带来额外的性能损耗。多线程并发执行某个业务流程，业务开发者还需要时刻对线程安全保持警惕，哪些数据可能会被并发修改，如何保护？这不仅降低了开发效率，也会带来额外的性能损耗。

串行执行 Handler 链

为了解决上述问题，**Netty 采用了串行化设计理念**，从消息的读取、编码以及后续 Handler 的执行，始终都**由 IO 线程 NioEventLoop 负责，这就意外着整个流程不会进行线程上下文的切换，数据也不会面临被并发修改的风险，**对于用户而言，甚至不需要了解 Netty 的线程细节，这确实是个非常好的设计理念，它的工作原理图如下：



NioEventLoop 串行执行 ChannelHandler

一个 NioEventLoop 聚合了一个多路复用器 Selector，因此可以处理成百上千的客户端连接，Netty 的处理策略是**每当有一个新的客户端接入，则从 NioEventLoopGroup线程组中顺序获取一个可用的 NioEventLoop**，当到达数组上限之后，重新返回到 0，通过这种方式，可以基本保证各个 NioEventLoop 的负载均衡。**一个客户端连接只注册到一个 NioEventLoop 上，这样就避免了多个 IO 线程去并发操作它。**

Netty 通过串行化设计理念降低了用户的开发难度，提升了处理性能。利用线程组实现了多个串行化线程水平并行执行，**线程之间并没有交集，这样既可以充分利用多核提升并行处理能力，同时避免了线程上下文的切换和并发保护带来的额外性能损耗。**

2.9.4 快速入门实例

```
package com.kkb.demo.netty.example.simple;

import io.netty.bootstrap.ServerBootstrap;
import io.netty.channel.*;
import io.netty.channel.nio.NioEventLoopGroup;
import io.netty.channel.socket.SocketChannel;
import io.netty.channel.socket.nio.NioServerSocketChannel;


/**
 * 1. `Netty` 服务器在 `6668` 端口监听，客户端能发送消息给服务器"hello,服务器~"
 * 2. 服务器可以回复消息给客户端"hello"
 * 3. 目的：对 `Netty` 线程模型有一个初步认识，便于理解 `Netty` 模型理论
 */
```

```
public class NettyServer {
    public static void main(String[] args) {
        //创建BossGroup 和 WorkerGroup
        //说明
        //1. 创建两个线程组 bossGroup 和 workerGroup
        //2. bossGroup 只是处理连接请求 , 真正的和客户端业务处理，会交给 workerGroup完成
        //3. 两个都是无限循环
        //4. bossGroup 和 workerGroup 含有的子线程(NioEventLoop)的个数
        //   默认实际 cpu核数 * 2
        EventLoopGroup bossGroup = new NioEventLoopGroup(1);
        EventLoopGroup workerGroup = new NioEventLoopGroup(); //16

        try {
            //创建服务器端的启动对象，配置参数
            ServerBootstrap bootstrap = new ServerBootstrap();
            //使用链式编程来进行设置
            bootstrap.group(bossGroup, workerGroup) //设置两个线程组
                    .channel(NioServerSocketChannel.class) //使用NioSocketChannel 作为服务器的通道实现
                    .option(ChannelOption.SO_BACKLOG, 128) // 设置线程队列得到连接个数
                    .childOption(ChannelOption.SO_KEEPALIVE, true) //设置保持活动连接状态
                    //         .handler(null) // 该 handler对应 bossGroup , childHandler 对应 workerGroup
                    .childHandler(new ChannelInitializer<SocketChannel>() {//创建一个通道初始化对象(匿名对象)
                        //给pipeline 设置处理器
                        @Override
                        protected void initChannel(SocketChannel ch) throws Exception {
                            System.out.println("客户socketchannel hashcode=" + ch.hashCode()); //可以使用一个集合管理 SocketChannel， 再推送消息时，可以将业务加入到各个channel 对应的 NIOEventLoop 的 taskQueue 或者 scheduleTaskQueue
                            ch.pipeline().addLast(new NettyServerHandler());
                        }
                    }); // 给我们的workerGroup 的 EventLoop 对应的管道设置处理器

            System.out.println(".....服务器 is ready...");


            //绑定一个端口并且同步, 生成了一个 ChannelFuture 对象
            //启动服务器(并绑定端口)
            ChannelFuture cf = bootstrap.bind(6668).sync();

            //给cf 注册监听器，监控我们关心的事件

            cf.addListener(new ChannelFutureListener() {
```

```
                @Override
                public void operationComplete(ChannelFuture future) throws
Exception {
                    if (cf.isSuccess()) {
                        System.out.println("监听端口 6668 成功");
                    } else {
                        System.out.println("监听端口 6668 失败");
                    }
                }
            });

        //对关闭通道进行监听

            cf.channel().closeFuture().sync();
        } catch (InterruptedException e) {
            e.printStackTrace();
        } finally {
            bossGroup.shutdownGracefully();
            workerGroup.shutdownGracefully();
        }

    }
}


import io.netty.buffer.ByteBuf;
import io.netty.buffer.Unpooled;
import io.netty.channel.Channel;
import io.netty.channel.ChannelHandlerContext;
import io.netty.channel.ChannelInboundHandlerAdapter;
import io.netty.channel.ChannelPipeline;
import io.netty.util.CharsetUtil;

import java.util.concurrent.TimeUnit;

public class NettyServerHandler extends ChannelInboundHandlerAdapter {


    //读取数据实际(这里我们可以读取客户端发送的消息)

    /**
     * 1. ChannelHandlerContext ctx:上下文对象, 含有 管道pipeline , 通道channel,
地址
     * 2. Object msg: 就是客户端发送的数据 默认Object
     */
    @Override
    public void channelRead(ChannelHandlerContext ctx, Object msg) throws
Exception {
```

```
        // 比如这里我们有一个非常耗时长的业务-> 异步执行 -> 提交该channel 对应的
        // NIOEventLoop 的 taskQueue中,

        // 解决方案1 用户程序自定义的普通任务

        ctx.channel().eventLoop().execute(new Runnable() {
            @Override
            public void run() {

                try {
                    Thread.sleep(5 * 1000);
                    ctx.writeAndFlush(Unpooled.copiedBuffer("hello, Client No
2", CharsetUtil.UTF_8));
                    System.out.println("channel code=" +
ctx.channel().hashCode());
                } catch (Exception ex) {
                    System.out.println("发生异常" + ex.getMessage());
                }
            }
        });

        ctx.channel().eventLoop().execute(new Runnable() {
            @Override
            public void run() {

                try {
                    Thread.sleep(5 * 1000);
                    ctx.writeAndFlush(Unpooled.copiedBuffer("hello, Client No
3", CharsetUtil.UTF_8));
                    System.out.println("channel code=" +
ctx.channel().hashCode());
                } catch (Exception ex) {
                    System.out.println("发生异常" + ex.getMessage());
                }
            }
        });

        //解决方案2 : 用户自定义定时任务 -》 该任务是提交到 scheduleTaskQueue中

        ctx.channel().eventLoop().schedule(new Runnable() {
            @Override
            public void run() {

                try {
                    Thread.sleep(5 * 1000);
                    ctx.writeAndFlush(Unpooled.copiedBuffer("hello, Client No
4", CharsetUtil.UTF_8));
                    System.out.println("channel code=" +
ctx.channel().hashCode());
```

```
                } catch (Exception ex) {
                    System.out.println("发生异常" + ex.getMessage());
                }
            }
        }, 5, TimeUnit.SECONDS);

        System.out.println("go on ...");

//        System.out.println("服务器读取线程 " + Thread.currentThread().getName()
+ " channle =" + ctx.channel());
//        System.out.println("server ctx =" + ctx);
//        System.out.println("看看channel 和 pipeline的关系");
//        Channel channel = ctx.channel();
//        ChannelPipeline pipeline = ctx.pipeline(); //本质是一个双向链接, 出站入站
//
//        //将 msg 转成一个 ByteBuf
//        //ByteBuf 是 Netty 提供的, 不是 NIO 的 ByteBuffer.
//        ByteBuf buf = (ByteBuf) msg;
//        System.out.println("客户端发送消息是:" +
buf.toString(CharsetUtil.UTF_8));
//        System.out.println("客户端地址:" + channel.remoteAddress());
    }


    public void channelRead2(ChannelHandlerContext ctx, Object msg) throws
Exception {
        System.out.println("服务器读取线程 " + Thread.currentThread().getName() +
" channle =" + ctx.channel());
        System.out.println("server ctx =" + ctx);
        System.out.println("看看channel 和 pipeline的关系");
        Channel channel = ctx.channel();
        ChannelPipeline pipeline = ctx.pipeline(); //本质是一个双向链接, 出站入站

        //将 msg 转成一个 ByteBuf
        //ByteBuf 是 Netty 提供的, 不是 NIO 的 ByteBuffer.
        ByteBuf buf = (ByteBuf) msg;
        System.out.println("客户端发送消息是:" +
buf.toString(CharsetUtil.UTF_8));
        System.out.println("客户端地址:" + channel.remoteAddress());
    }

    //数据读取完毕
    @Override
    public void channelReadComplete(ChannelHandlerContext ctx) throws
Exception {
        //writeAndFlush 是 write + flush
        //将数据写入到缓存, 并刷新
        //一般讲, 我们对这个发送的数据进行编码
```

```
        ctx.writeAndFlush(Unpooled.copiedBuffer("hello, Client No 1",
CharsetUtil.UTF_8));
    }

    //处理异常, 一般是需要关闭通道
    @Override
    public void exceptionCaught(ChannelHandlerContext ctx, Throwable cause)
throws Exception {
        ctx.close();
    }
}
```

客户端

```
import io.netty.bootstrap.Bootstrap;
import io.netty.channel.ChannelFuture;
import io.netty.channel.ChannelInitializer;
import io.netty.channel.EventLoopGroup;
import io.netty.channel.nio.NioEventLoopGroup;
import io.netty.channel.socket.SocketChannel;
import io.netty.channel.socket.nio.NioSocketChannel;

public class NettyClient {
    public static void main(String[] args) {
        //客户端需要一个事件循环组
        EventLoopGroup group = new NioEventLoopGroup();

        //创建客户端启动对象
        //注意客户端使用的不是 ServerBootstrap 而是 Bootstrap
        Bootstrap bootstrap = new Bootstrap();

        try{
            //设置相关参数
            bootstrap.group(group) //设置线程组
                    .channel(NioSocketChannel.class) // 设置客户端通道的实现类(反
射)
                    .handler(new ChannelInitializer<SocketChannel>() {
                        @Override
                        protected void initChannel(SocketChannel ch) throws
Exception {
                            ch.pipeline().addLast(new NettyClientHandler());
//加入自己的处理器
                        }
                    });

            System.out.println("客户端 ok..");

            //启动客户端去连接服务器端
```

```
            //关于 ChannelFuture 要分析，涉及到netty的异步模型
            ChannelFuture channelFuture = null;
            try {
                channelFuture = bootstrap.connect("127.0.0.1", 6668).sync();
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            //给关闭通道进行监听
            channelFuture.channel().closeFuture().sync();
        } catch (InterruptedException e) {
            e.printStackTrace();
        } finally {
            group.shutdownGracefully();
        }

    }
}

import io.netty.buffer.ByteBuf;
import io.netty.buffer.Unpooled;
import io.netty.channel.ChannelHandlerContext;
import io.netty.channel.ChannelInboundHandlerAdapter;
import io.netty.util.CharsetUtil;

public class NettyClientHandler extends ChannelInboundHandlerAdapter {
    //当通道就绪就会触发该方法
    @Override
    public void channelActive(ChannelHandlerContext ctx) throws Exception {
        System.out.println("client " + ctx);
        ctx.writeAndFlush(Unpooled.copiedBuffer("hello, server: (>^ω^<)喵",
CharsetUtil.UTF_8));
    }

    //当通道有读取事件时，会触发
    @Override
    public void channelRead(ChannelHandlerContext ctx, Object msg) throws
Exception {
        ByteBuf buf = (ByteBuf) msg;
        System.out.println("服务器回复的消息:" +
buf.toString(CharsetUtil.UTF_8));
        System.out.println("服务器的地址： " + ctx.channel().remoteAddress());
    }

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

```
}
```

2.10 异步模型

2.10.1 基本介绍

1 . 异步操作概念 :

调用者调用一个异步操作后 , 并不能马上知道该操作的返回值 , 该操作也不会马上执行完成 , 该操作完成后 , 会通过回调机制 , 如 通知 , 注册的回调函数等机制通知调用者 ;

2 . Netty 中的异步操作与 ChannelFuture 返回值 :

① **异步操作 :** Netty 模型中凡是关于 IO 的操作 , 如绑定端口 (Bind) , 远程连接 (Connect) , 读取数据 (Read) , 写出数据 (Write) 等操作都是异步操作 ;

② **异步操作返回值 :** 上述 IO 操作返回值都是 ChannelFuture 类型实例 , ChannelFuture 是异步 IO 操作的返回结果 ;

③ **在服务器端绑定端口号时** , 调用 Bootstrap 的 bind 方法 , 会返回 ChannelFuture 对象 ;

④ **在客户端调用 Bootstrap 的 connect 方法** , 也会返回 ChannelFuture 对象 ;

3 . Netty 中的异步操作机制 :

① **Future-Listener 机制 :** Future 表示当前不知道结果 , 在未来的某个时刻才知道结果 , Listener 表示监听操作 , 监听返回的结果 ;

② **Netty 异步模型的两个基础 :** Future (ChannelFuture 未来知道结果) , Callback (监听回调) ;

4 . 以客户端写出数据到服务器端为例 :

客户端写出数据 : 客户端调用写出数据方法 ChannelFuture writeAndFlush(Object msg) , 向服务器写出数据 ;

操作耗时 : 假设在服务器中接收到该数据后 , 要执行一个非常耗时的操作才能返回结果 , 就是操作非常耗时 ;

客户端不等待 : 客户端这里写出了数据 , 肯定不能阻塞等待写出操作的结果 , 需要立刻执行下面的操作 , 因此该方法是异步的 ;

客户端监听 : writeAndFlush 方法返回一个 ChannelFuture 对象 , 如果客户端需要该操作的返回结果 , 那么通过 ChannelFuture 可以监听该写出方法是否成功 ;

5 . 异步操作返回结果 :

① **返回结果 :** Future 表示异步 IO 操作执行结果 , 通过该 Future 提供的 检索 , 计算 等方法检查异步操作是否执行完成 ;

② **常用接口 :** ChannelFuture 继承了 Future , 也是一个接口 , 可以为该接口对象注册监听器 , 当异步任务完成后会回调该监听器方法 ;

6 . Future 链式操作 : 这里以读取数据 , 处理后返回结果为例 ;

- 数据读取操作 ;
- 对读取的数据进行解码处理 ;

- 执行业务逻辑
- 将数据编码 ;
- 将编码后的数据写出 ;

上述5 个步骤 , 每个数据处理操作 , 都有与之对应的 Handler 处理器 ;

2.10.2 Future-Listener 机制

① **Future 返回值 :** 在 Netty 中执行 IO 操作 , 如 bind , read , write , connect 等方法 , 会立刻返回 ChannelFuture 对象 ;

② **ChannelFuture 返回时状态 :** 调用 IO 方法后 , 立刻返回 ChannelFuture 对象 , 此时该操作未完成 ;

③ **注册监听器 :** ChannelFuture 可以设置 ChannelFutureListener 监听器 , 监听该 IO 操作完成状态 , 如果 IO 操作完成 , 那么会回调其 public void operationComplete(ChannelFuture future) throws Exception 接口实现方法 ;

④ **IO 操作执行状态判定 :** 在 operationComplete 方法中通过 调用 ChannelFuture future 参数的如下方法 , 判定当前 IO 操作完成状态 ;

- **future.isDone() :** IO 操作是否完成 ;
- **future.isSuccess() :** IO 操作是否成功 ; **(常用)**
- **future.isCancelled() :** IO 操作是否被取消 ;
- **future.cause() :** IO 操作的失败原因 ;

2 . 核心代码示例 :

```
// 监听绑定操作的结果
// 添加 ChannelFutureListener 监听器, 监听 bind 操作的结果
channelFuture.addListener(new ChannelFutureListener() {
    @Override
    public void operationComplete(ChannelFuture future) throws Exception {
        if(future.isDone()){
            System.out.println("绑定端口完成");
        }
        if(future.isSuccess()){
            System.out.println("绑定端口成功");
        }else{
            System.out.println("绑定端口失败");
        }
        if(future.isCancelled()){
            System.out.println("绑定端口取消");
        }
        System.out.println("失败原因 : " + future.cause());
    }
});
```

2.10.3 Future-Listener 机制代码示例(了解)

```
import io.netty.bootstrap.ServerBootstrap;
import io.netty.channel.*;
import io.netty.channel.nio.NioEventLoopGroup;
import io.netty.channel.socket.SocketChannel;
import io.netty.channel.socket.nio.NioServerSocketChannel;

/**
 * Netty 案例服务器端
 */
public class AsyncServer {
    public static void main(String[] args) {

        // 1. 创建 BossGroup 线程池 和 WorkerGroup 线程池, 其中维护 NioEventLoop 线
程
        //     NioEventLoop 线程中执行无限循环操作

        // BossGroup 线程池 : 负责客户端的连接
        // 指定线程个数 : 客户端个数很少, 不用很多线程维护, 这里指定线程池中线程个数为 1
        EventLoopGroup bossGroup = new NioEventLoopGroup(1);
        // WorkerGroup 线程池 : 负责客户端连接的数据读写
        EventLoopGroup workerGroup = new NioEventLoopGroup();

        // 2. 服务器启动对象, 需要为该对象配置各种参数
        ServerBootstrap bootstrap = new ServerBootstrap();
        bootstrap.group(bossGroup, workerGroup) // 设置 主从 线程组 , 分别对应 主
Reactor 和 从 Reactor
                .channel(NioServerSocketChannel.class)  // 设置 NIO 网络套接字通
道类型
                .option(ChannelOption.SO_BACKLOG, 128)  // 设置线程队列维护的连接
个数
                .childOption(ChannelOption.SO_KEEPALIVE, true)  // 设置连接状态行
为, 保持连接状态
                .childHandler(  // 为 WorkerGroup 线程池对应的 NioEventLoop 设置对
应的事件 处理器 Handler
                        new ChannelInitializer<SocketChannel>() {// 创建通道初始
化对象
                            @Override
                            protected void initChannel(SocketChannel ch)
throws Exception {
                                // 该方法在服务器与客户端连接建立成功后会回调
                                // 为 管道 Pipeline 设置处理器 Hanedler
                                // 这里暂时设置为 null , 执行不会失败 , 服务器绑定端
口会成功
                                ch.pipeline().addLast(null);
                            }
                        }
                );
        System.out.println("服务器准备完毕 ...");
```

```
        ChannelFuture channelFuture = null;
        try {
            // 绑定本地端口, 进行同步操作 , 并返回 ChannelFuture
            channelFuture = bootstrap.bind(8888).sync();
            System.out.println("服务器开始监听 8888 端口 ...");

            // ( 本次示例核心代码 ) ----------------------------------------------
-------------
            // 监听绑定操作的结果 ( 本次示例核心代码 )
            // 添加 ChannelFutureListener 监听器, 监听 bind 操作的结果
            channelFuture.addListener(new ChannelFutureListener() {
                @Override
                public void operationComplete(ChannelFuture future) throws
Exception {
                    if(future.isDone()){
                        System.out.println("绑定端口完成");
                    }

                    if(future.isSuccess()){
                        System.out.println("绑定端口成功");
                    }else{
                        System.out.println("绑定端口失败");
                    }

                    if(future.isCancelled()){
                        System.out.println("绑定端口取消");
                    }

                    System.out.println("失败原因 : " + future.cause());
                }
            });
            // ( 本次示例核心代码 ) ----------------------------------------------
-------------


            // 关闭通道 , 开始监听操作
            channelFuture.channel().closeFuture().sync();
        } catch (InterruptedException e) {
            e.printStackTrace();
        } finally {
            // 出现异常后, 优雅的关闭
            bossGroup.shutdownGracefully();
            workerGroup.shutdownGracefully();
        }

    }
}
```

2.11 入门实例-HTTP服务

```
import io.netty.bootstrap.ServerBootstrap;
import io.netty.channel.ChannelFuture;
import io.netty.channel.EventLoopGroup;
import io.netty.channel.nio.NioEventLoopGroup;
import io.netty.channel.socket.nio.NioServerSocketChannel;

/**
 * 1. `Netty` 服务器在 `6868` 端口监听，浏览器发出请求 `http://localhost:6868/`
 * 2. 服务器可以回复消息给客户端"Hello",并对特定请求资源进行过滤。
 * 3. 目的：`Netty` 可以做 `Http` 服务开发，并且理解 `Handler` 实例和客户端及其请求的关系。
 */
public class TestHttpServer {

    public static void main(String[] args) throws Exception {

        EventLoopGroup bossGroup = new NioEventLoopGroup(1);
        EventLoopGroup workerGroup = new NioEventLoopGroup();

        try {
            ServerBootstrap serverBootstrap = new ServerBootstrap();

            serverBootstrap.group(bossGroup, workerGroup).channel(NioServerSocketChannel.class).childHandler(new TestServerInitializer());

            ChannelFuture channelFuture = serverBootstrap.bind(6868).sync();

            channelFuture.channel().closeFuture().sync();

        } finally {
            bossGroup.shutdownGracefully();
            workerGroup.shutdownGracefully();
        }
    }
}
```

```
import io.netty.channel.ChannelInitializer;
import io.netty.channel.ChannelPipeline;
import io.netty.channel.socket.SocketChannel;
import io.netty.handler.codec.http.HttpServerCodec;

public class TestServerInitializer extends ChannelInitializer<SocketChannel> {
    @Override
```

```
    protected void initChannel(SocketChannel ch) throws Exception {
        //向管道加入处理器

        //得到管道
        ChannelPipeline pipeline = ch.pipeline();

        //加入一个netty 提供的httpServerCodec codec =>[coder - decoder]
        //HttpServerCodec 说明
        //1. HttpServerCodec 是netty 提供的处理http的 编-解码器
        pipeline.addLast("MyHttpServerCodec", new HttpServerCodec());
        //2. 增加一个自定义的handler
        pipeline.addLast("MyTestHttpServerHandler", new
TestHttpServerHandler());

        System.out.println("ok~~~~");
    }
}
```

```
import io.netty.buffer.ByteBuf;
import io.netty.buffer.Unpooled;
import io.netty.channel.ChannelHandlerContext;
import io.netty.channel.SimpleChannelInboundHandler;
import io.netty.handler.codec.http.*;
import io.netty.util.CharsetUtil;

import java.net.URI;

public class TestHttpServerHandler extends
SimpleChannelInboundHandler<HttpObject> {
    @Override
    protected void channelRead0(ChannelHandlerContext ctx, HttpObject msg)
throws Exception {
        System.out.println("对应的channel=" + ctx.channel() + " pipeline=" +
ctx
                .pipeline() + " 通过pipeline获取channel" +
ctx.pipeline().channel());

        System.out.println("当前ctx的handler=" + ctx.handler());

        //判断 msg 是不是 httprequest请求
        if (msg instanceof HttpRequest) {

            System.out.println("ctx 类型=" + ctx.getClass());

            System.out.println("pipeline hashcode" + ctx.pipeline().hashCode()
+ " TestHttpServerHandler hash=" + this.hashCode());
```

```
            System.out.println("msg 类型=" + msg.getClass());
            System.out.println("客户端地址" + ctx.channel().remoteAddress());

            //获取到
            HttpRequest httpRequest = (HttpRequest) msg;
            //获取uri, 过滤指定的资源
            URI uri = new URI(httpRequest.uri());
            if ("/favicon.ico".equals(uri.getPath())) {
                System.out.println("请求了 favicon.ico, 不做响应");
                return;
            }
            //回复信息给浏览器 [http协议]

            ByteBuf content = Unpooled.copiedBuffer("hello,my name is shuai",
CharsetUtil.UTF_8);

            //构造一个http的相应，即 httpresponse
            FullHttpResponse response = new
DefaultFullHttpResponse(HttpVersion.HTTP_1_1, HttpResponseStatus.OK, content);

            response.headers().set(HttpHeaderNames.CONTENT_TYPE,
"text/plain");
            response.headers().set(HttpHeaderNames.CONTENT_LENGTH,
content.readableBytes());

            //将构建好 response返回
            ctx.writeAndFlush(response);
        }
    }
}
```