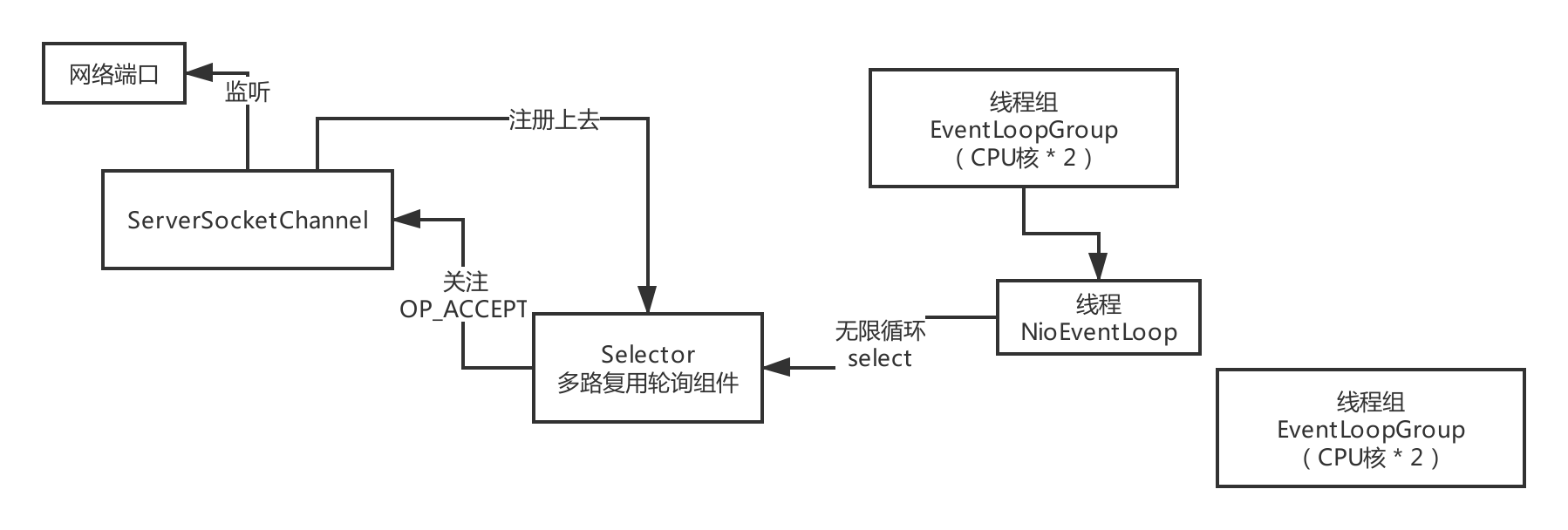
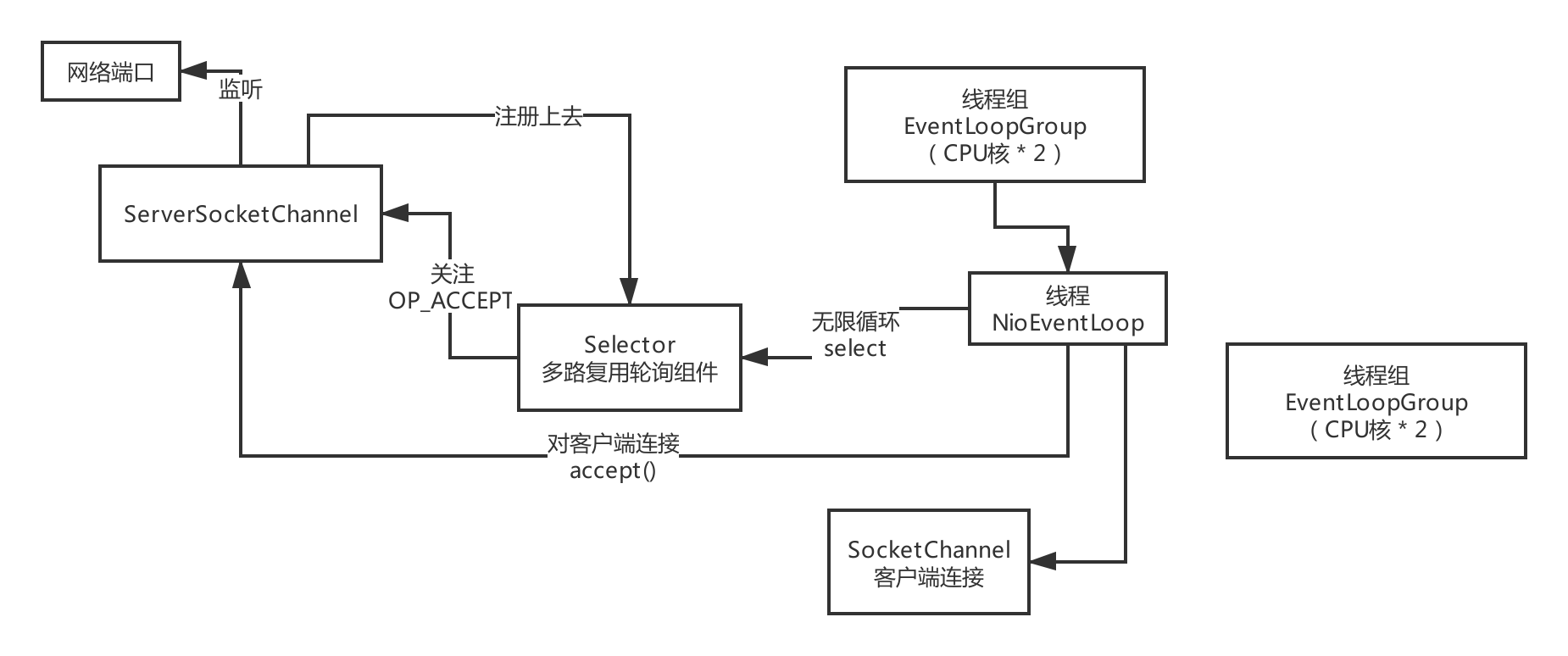
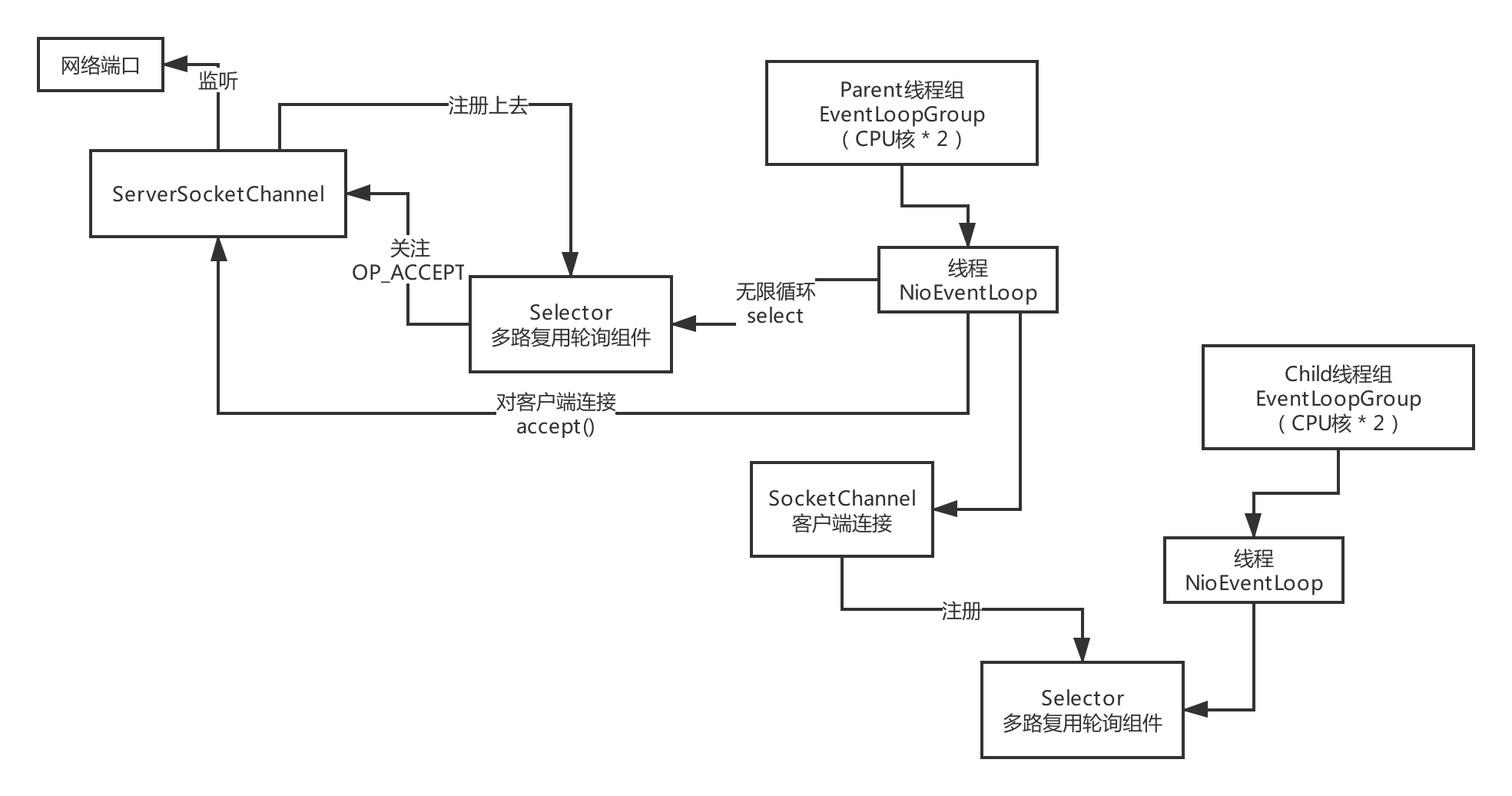
# **File Path: /Users/xiaotingting/Downloads/学习资料/12\_Netty核心功能精讲以及核心源码剖析/019~030资料/025\_Netty的线程是如何轮询ServerSocketChannel的网络连接事件的？/01\_Netty源码架构 (1).png**



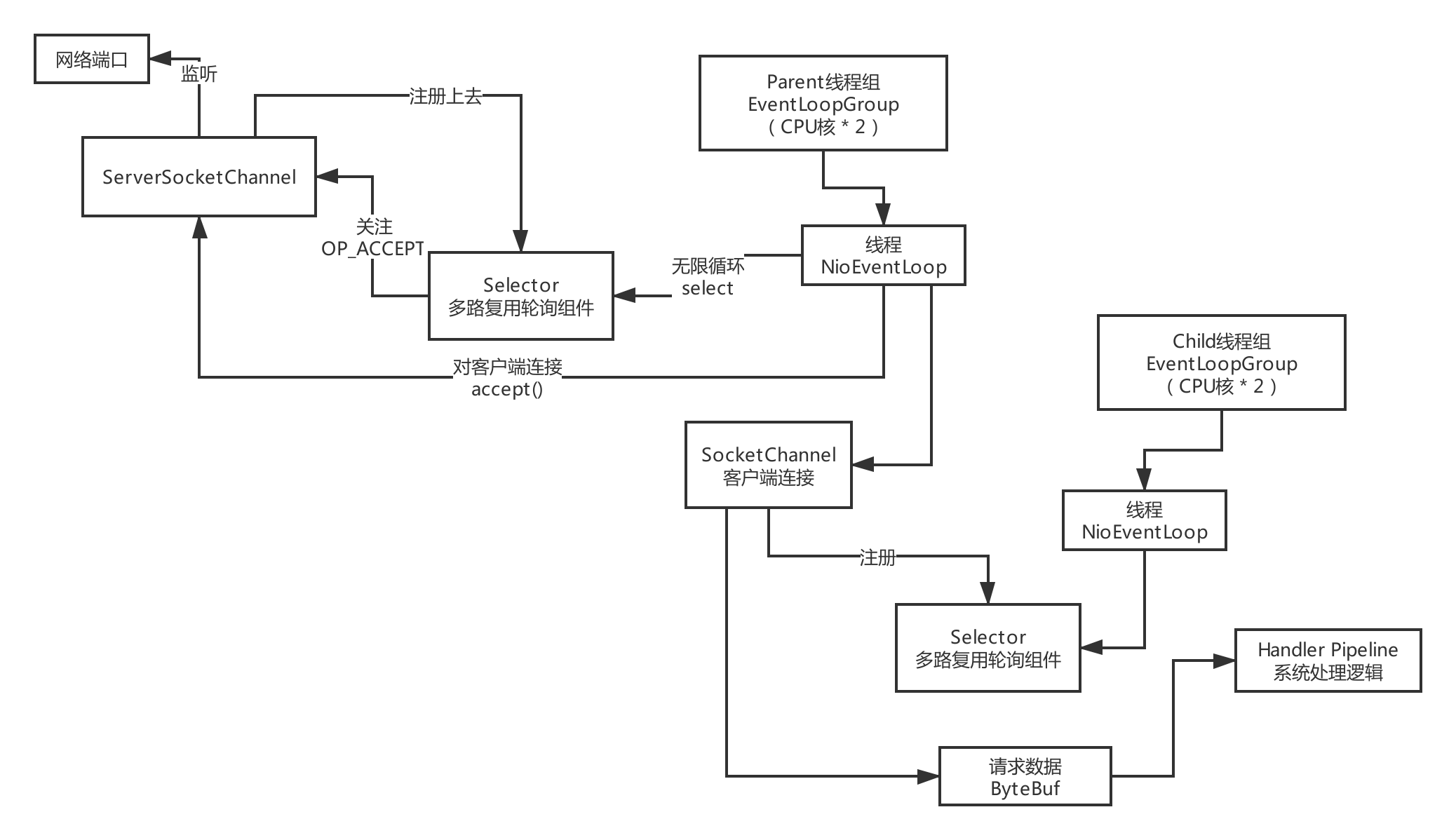
# **File Path: /Users/xiaotingting/Downloads/学习资料/12\_Netty核心功能精讲以及核心源码剖析/019~030资料/026\_如果发现了客户端发起的连接事件是如何进行处理的？/01\_Netty源码架构 (2).png**



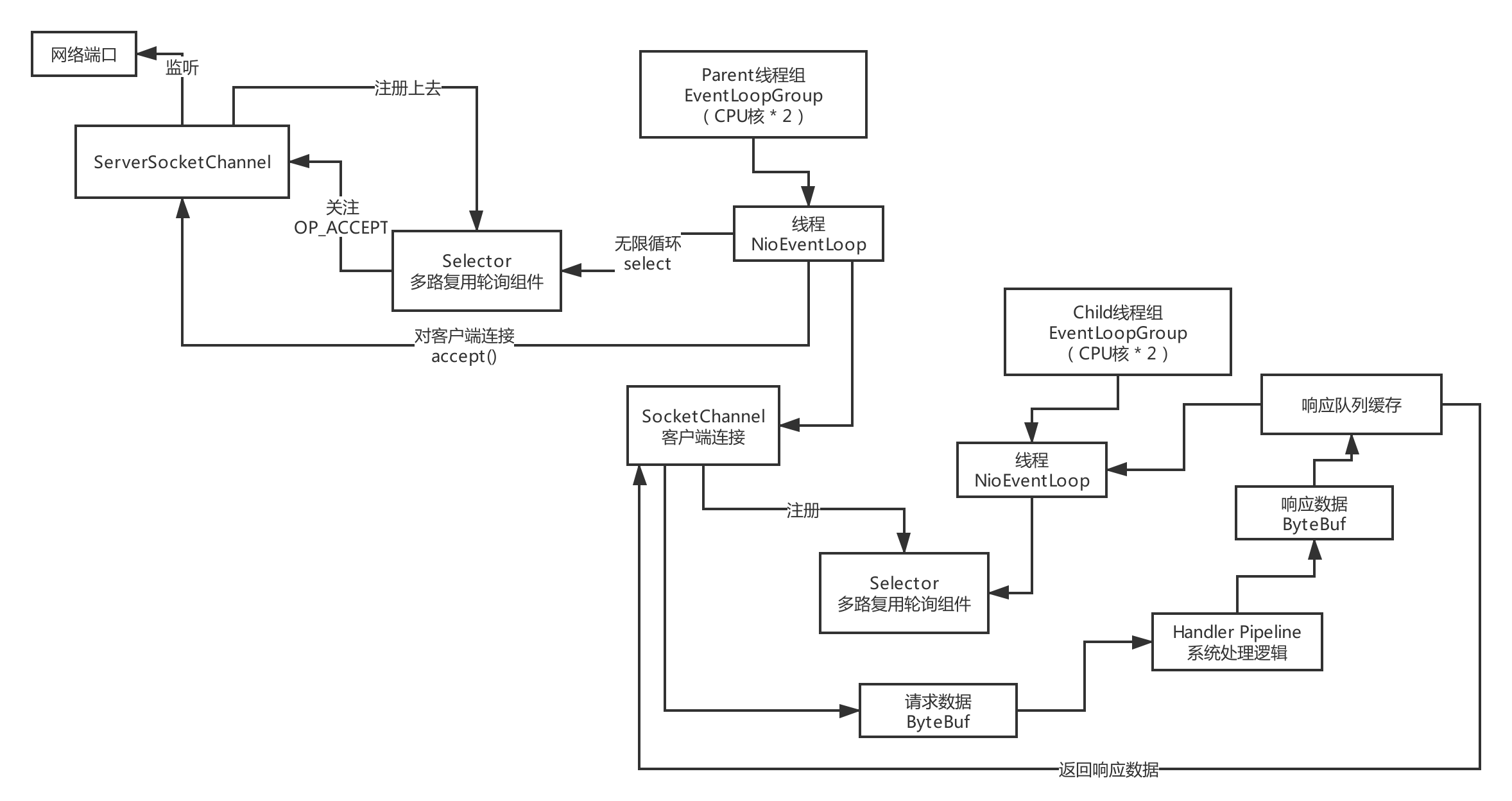
# **File Path: /Users/xiaotingting/Downloads/学习资料/12\_Netty核心功能精讲以及核心源码剖析/019~030资料/027\_Server端对建立好的客户端连接是交给谁来轮询网络请求的？/01\_Netty源码架构 (3).png**



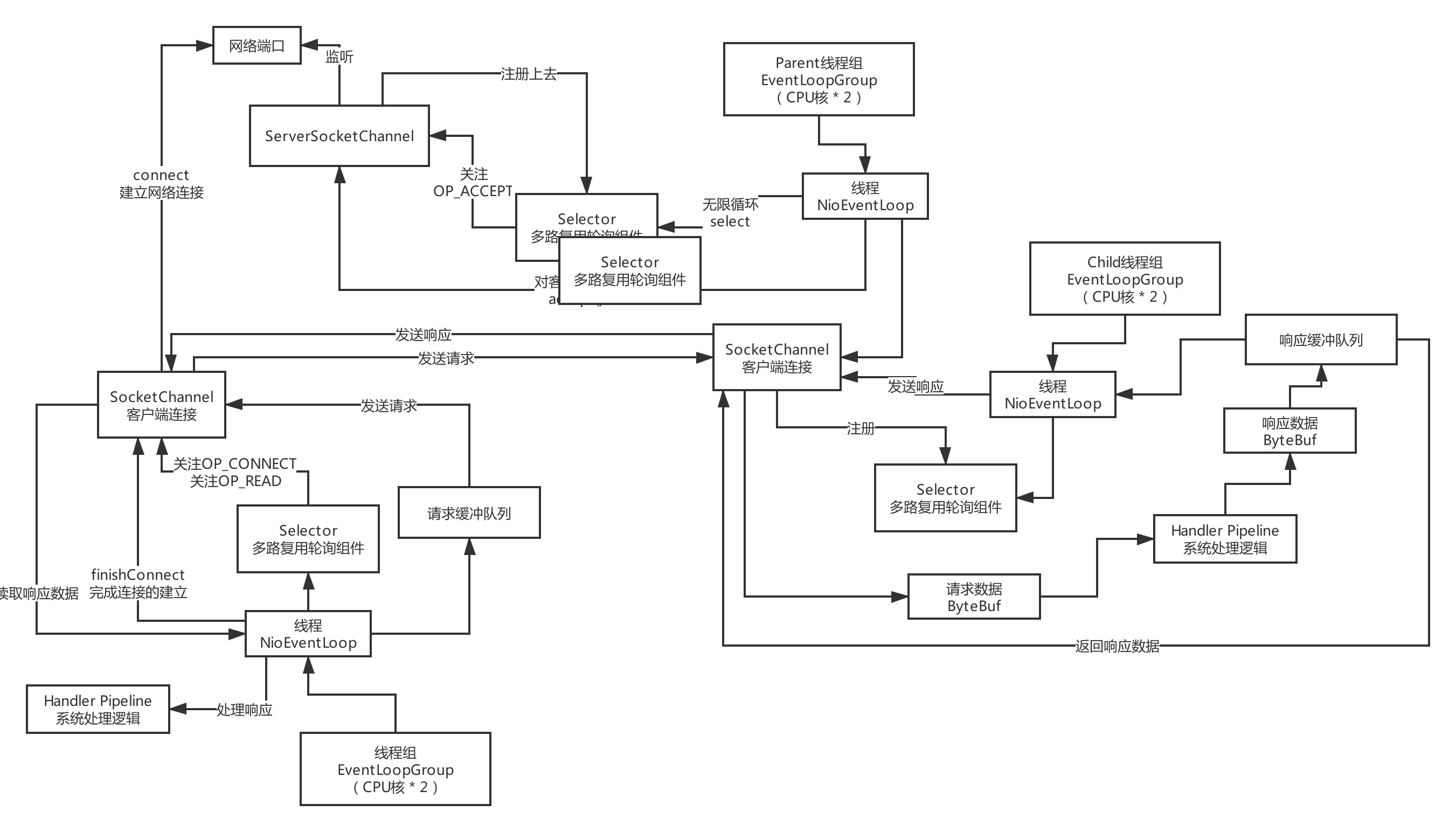
# **File Path: /Users/xiaotingting/Downloads/学习资料/12\_Netty核心功能精讲以及核心源码剖析/019~030资料/028\_对于客户端发送过来的消息是如何读取以及处理的？/01\_Netty源码架构 (4).png**



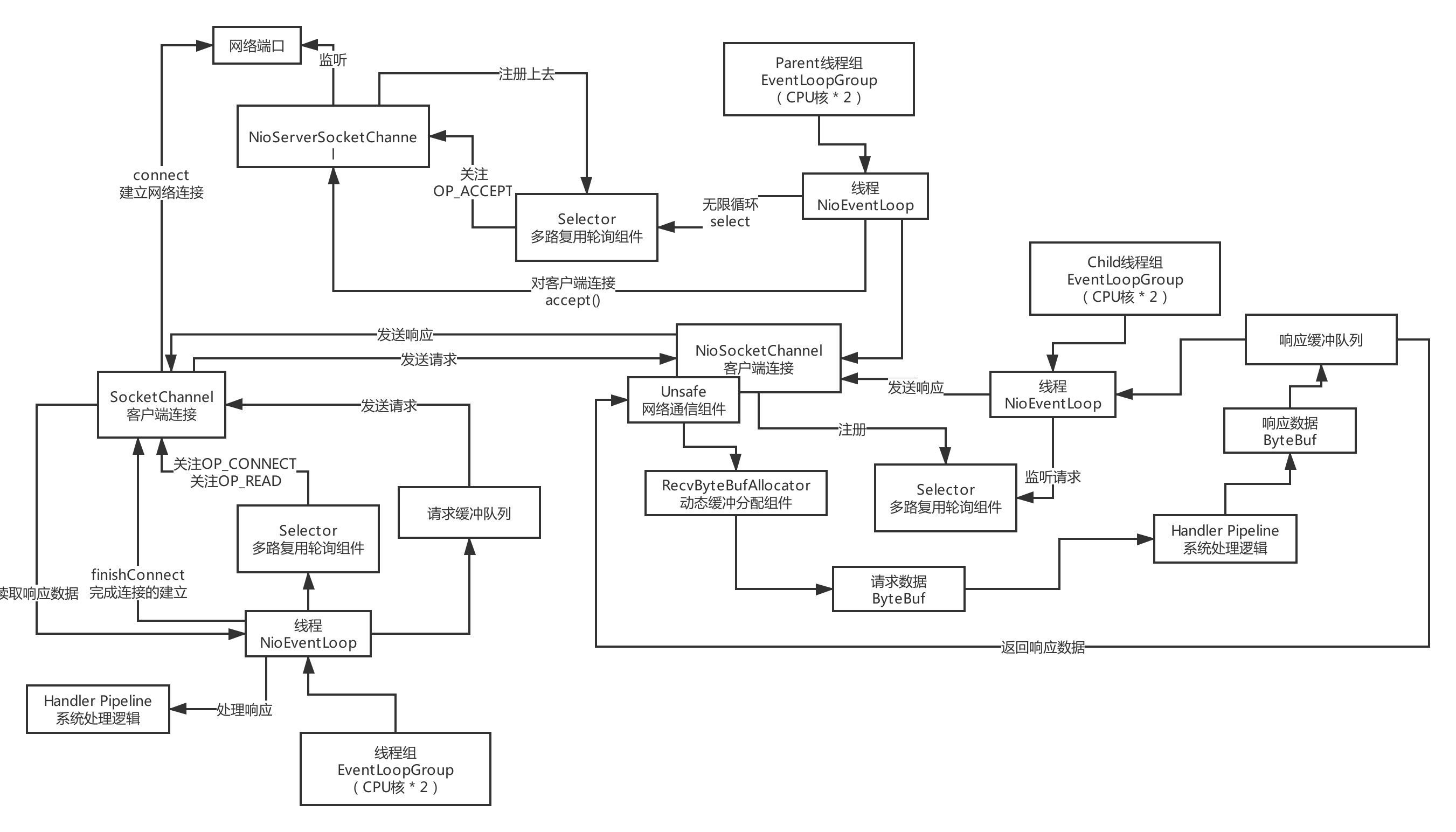
# **File Path: /Users/xiaotingting/Downloads/学习资料/12\_Netty核心功能精讲以及核心源码剖析/019~030资料/029\_系统处理完请求后又是如何将响应消息发送客户端的？/01\_Netty源码架构 (5).png**



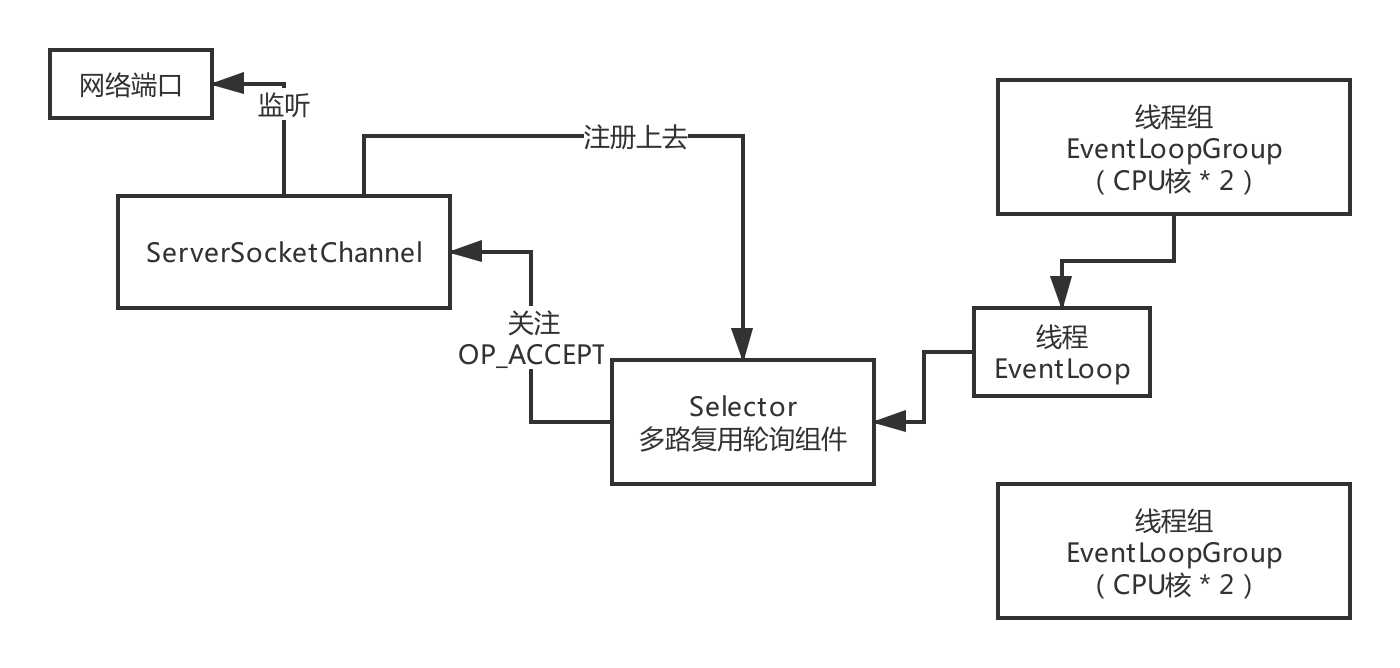
# **File Path: /Users/xiaotingting/Downloads/学习资料/12\_Netty核心功能精讲以及核心源码剖析/031~044资料/035\_从Netty Server端返回的响应是如何接收的？/01\_Netty源码架构 (6).png**



# **File Path: /Users/xiaotingting/Downloads/学习资料/12\_Netty核心功能精讲以及核心源码剖析/031~044资料/044\_Netty为请求处理提供的良好扩展：自定义业务逻辑链条/01\_Netty源码架构 (7).png**



# **File Path: /Users/xiaotingting/Downloads/学习资料/12\_Netty核心功能精讲以及核心源码剖析/019~030资料/023\_用一张图来展示出ServerSocketChannel初始化的过程/01\_Netty源码架构.png**



# **File Path: /Users/xiaotingting/Downloads/学习资料/12\_Netty核心功能精讲以及核心源码剖析/019~030资料/023\_用一张图来展示出ServerSocketChannel初始化的过程/源码分析.txt**

// 方法的名字，初始化ServerSocketChannel，注册到Selector多路复用轮询组件上去

final ChannelFuture initAndRegister() {

Channel channel = null;

try {

// 有一个默认的ChannelFactory

// 估计就是来创建ServerSocketChannel的，ServerSocketChannel创建好了

// 而且配置好了blocking是false

// 准备好了ServerSocketChannel将要关注的网络事件，OP\_ACCEPT

channel = channelFactory.newChannel();

// 接下来必然是让这个ServerSocketChannel去监听某个端口号，设置一些网络参数

// 然后再让这个ServerSocketChannel注册到Selector上去，关注他的OP\_ACCEPT网络事件，不停的轮询他

init(channel);

// 最最核心的，还是关注他核心的架构设计、底层技术的处理

} catch (Throwable t) {

if (channel != null) {

// channel can be null if newChannel crashed (eg SocketException("too many open files"))

channel.unsafe().closeForcibly();

// as the Channel is not registered yet we need to force the usage of the GlobalEventExecutor

return new DefaultChannelPromise(channel, GlobalEventExecutor.INSTANCE).setFailure(t);

}

// as the Channel is not registered yet we need to force the usage of the GlobalEventExecutor

return new DefaultChannelPromise(new FailedChannel(), GlobalEventExecutor.INSTANCE).setFailure(t);

}

// 拿出来了之前创建的一个EventLoopGroup，猜测，独立的线程，复杂用Selector轮询各种channel的网络事件

// 他把ServerSocketChannel注册到了一个EventLoopGroup上去

// 这里的意思，是不是就是说让EventLoopGroup中的独立线程采用一个Selector来注册channel，以及轮询事件

ChannelFuture regFuture = config().group().register(channel);

if (regFuture.cause() != null) {

if (channel.isRegistered()) {

channel.close();

} else {

channel.unsafe().closeForcibly();

}

}

// If we are here and the promise is not failed, it's one of the following cases:

// 1) If we attempted registration from the event loop, the registration has been completed at this point.

// i.e. It's safe to attempt bind() or connect() now because the channel has been registered.

// 2) If we attempted registration from the other thread, the registration request has been successfully

// added to the event loop's task queue for later execution.

// i.e. It's safe to attempt bind() or connect() now:

// because bind() or connect() will be executed \*after\* the scheduled registration task is executed

// because register(), bind(), and connect() are all bound to the same thread.

return regFuture;

}

@Override

public T newChannel() {

try {

// constructor，反射方式，NioServerSocketChannel.class

// 这个类是我们自己设置的，它是Netty提供的一个类

// 他一定是Netty自己把ServerSocketChannel给封装了一下，NioServerSocketChannel

// NettyServerSocketChannel

return constructor.newInstance();

} catch (Throwable t) {

throw new ChannelException("Unable to create Channel from class " + constructor.getDeclaringClass(), t);

}

}

private static ServerSocketChannel newSocket(SelectorProvider provider) {

try {

/\*\*

\* Use the {@link SelectorProvider} to open {@link SocketChannel} and so remove condition in

\* {@link SelectorProvider#provider()} which is called by each ServerSocketChannel.open() otherwise.

\*

\* See <a href="https://github.com/netty/netty/issues/2308">#2308</a>.

\*/

// 直接就是通过原生的NIO的API，创建了一个ServerSocketChannel

return provider.openServerSocketChannel();

} catch (IOException e) {

throw new ChannelException(

"Failed to open a server socket.", e);

}

}

@Override

void init(Channel channel) throws Exception {

// 对ServerSocketChannel进行网络参数的设置

final Map<ChannelOption<?>, Object> options = options0();

synchronized (options) {

setChannelOptions(channel, options, logger);

}

// 初始化了一些属性

final Map<AttributeKey<?>, Object> attrs = attrs0();

synchronized (attrs) {

for (Entry<AttributeKey<?>, Object> e: attrs.entrySet()) {

@SuppressWarnings("unchecked")

AttributeKey<Object> key = (AttributeKey<Object>) e.getKey();

channel.attr(key).set(e.getValue());

}

}

ChannelPipeline p = channel.pipeline();

final EventLoopGroup currentChildGroup = childGroup;

final ChannelHandler currentChildHandler = childHandler;

final Entry<ChannelOption<?>, Object>[] currentChildOptions;

final Entry<AttributeKey<?>, Object>[] currentChildAttrs;

synchronized (childOptions) {

currentChildOptions = childOptions.entrySet().toArray(newOptionArray(0));

}

synchronized (childAttrs) {

currentChildAttrs = childAttrs.entrySet().toArray(newAttrArray(0));

}

// 就是对网络请求处理链路中加入一个自己内置的一个处理逻辑

// 初始化了网络请求的处理链路

p.addLast(new ChannelInitializer<Channel>() {

@Override

public void initChannel(final Channel ch) throws Exception {

final ChannelPipeline pipeline = ch.pipeline();

ChannelHandler handler = config.handler();

if (handler != null) {

pipeline.addLast(handler);

}

ch.eventLoop().execute(new Runnable() {

@Override

public void run() {

pipeline.addLast(new ServerBootstrapAcceptor(

ch, currentChildGroup, currentChildHandler, currentChildOptions, currentChildAttrs));

}

});

}

});

}

@Override

public ChannelFuture register(Channel channel) {

// 所谓的EventLoopGroup就是可以不停的获取出来很多的线程

// 对应着一个线程池的，线程的数量是固定的，cpu核数 \* 2

// 调用next()方法轮询获取出来这个线程池里的下一个线程，推测就是这个意思

// 123456,6个线程，第一次Next()是线程1，第二次next()是线程2，第三次next()是线程3

// 他的next()方法，就是每次调用返回一个线程给你，让你通过这个线程去使用线程自己的Selector

// 注册对应的channel，不管是ServerSocketChannel，还是普通的SocketChannel

// 让这个线程去负责对应的这些channel的网络事件的轮询

// 不知道大家是否还记得，就是之前我们写Reactor模型的时候，也是类似的思路，我们搞了一组Processor线程

// 每次有新的网络连接进入，都是轮询各个Processor线程，保证每个线程都可以获取到一定的连接

return next().register(channel);

}

@Override

public EventExecutor next() {

return executors[Math.abs(idx.getAndIncrement() % executors.length)];

}

@Override

protected void doRegister() throws Exception {

boolean selected = false;

for (;;) {

try {

// javaChannel() -> ServerSocketChannel

// register(Selector, 0, this)

selectionKey = javaChannel().register(eventLoop().unwrappedSelector(), 0, this);

return;

} catch (CancelledKeyException e) {

if (!selected) {

// Force the Selector to select now as the "canceled" SelectionKey may still be

// cached and not removed because no Select.select(..) operation was called yet.

eventLoop().selectNow();

selected = true;

} else {

// We forced a select operation on the selector before but the SelectionKey is still cached

// for whatever reason. JDK bug ?

throw e;

}

}

}

}

# **File Path: /Users/xiaotingting/Downloads/学习资料/12\_Netty核心功能精讲以及核心源码剖析/019~030资料/020\_对照原生NIO代码看看Netty如何创建ServerSocketChannel/源码分析.txt**

// 方法的名字，初始化ServerSocketChannel，注册到Selector多路复用轮询组件上去

final ChannelFuture initAndRegister() {

Channel channel = null;

try {

// 有一个默认的ChannelFactory

// 估计就是来创建ServerSocketChannel的，ServerSocketChannel创建好了

// 而且配置好了blocking是false

// 准备好了ServerSocketChannel将要关注的网络事件，OP\_ACCEPT

channel = channelFactory.newChannel();

// 接下来必然是让这个ServerSocketChannel去监听某个端口号，设置一些网络参数

// 然后再让这个ServerSocketChannel注册到Selector上去，关注他的OP\_ACCEPT网络事件，不停的轮询他

init(channel);

} catch (Throwable t) {

if (channel != null) {

// channel can be null if newChannel crashed (eg SocketException("too many open files"))

channel.unsafe().closeForcibly();

// as the Channel is not registered yet we need to force the usage of the GlobalEventExecutor

return new DefaultChannelPromise(channel, GlobalEventExecutor.INSTANCE).setFailure(t);

}

// as the Channel is not registered yet we need to force the usage of the GlobalEventExecutor

return new DefaultChannelPromise(new FailedChannel(), GlobalEventExecutor.INSTANCE).setFailure(t);

}

ChannelFuture regFuture = config().group().register(channel);

if (regFuture.cause() != null) {

if (channel.isRegistered()) {

channel.close();

} else {

channel.unsafe().closeForcibly();

}

}

// If we are here and the promise is not failed, it's one of the following cases:

// 1) If we attempted registration from the event loop, the registration has been completed at this point.

// i.e. It's safe to attempt bind() or connect() now because the channel has been registered.

// 2) If we attempted registration from the other thread, the registration request has been successfully

// added to the event loop's task queue for later execution.

// i.e. It's safe to attempt bind() or connect() now:

// because bind() or connect() will be executed \*after\* the scheduled registration task is executed

// because register(), bind(), and connect() are all bound to the same thread.

return regFuture;

}

@Override

public T newChannel() {

try {

// constructor，反射方式，NioServerSocketChannel.class

// 这个类是我们自己设置的，它是Netty提供的一个类

// 他一定是Netty自己把ServerSocketChannel给封装了一下，NioServerSocketChannel

// NettyServerSocketChannel

return constructor.newInstance();

} catch (Throwable t) {

throw new ChannelException("Unable to create Channel from class " + constructor.getDeclaringClass(), t);

}

}

private static ServerSocketChannel newSocket(SelectorProvider provider) {

try {

/\*\*

\* Use the {@link SelectorProvider} to open {@link SocketChannel} and so remove condition in

\* {@link SelectorProvider#provider()} which is called by each ServerSocketChannel.open() otherwise.

\*

\* See <a href="https://github.com/netty/netty/issues/2308">#2308</a>.

\*/

// 直接就是通过原生的NIO的API，创建了一个ServerSocketChannel

return provider.openServerSocketChannel();

} catch (IOException e) {

throw new ChannelException(

"Failed to open a server socket.", e);

}

}

# **File Path: /Users/xiaotingting/Downloads/学习资料/12\_Netty核心功能精讲以及核心源码剖析/019~030资料/022\_如何将ServerSocketChannel注册到Selector由独立线程轮询？/源码分析.txt**

// 方法的名字，初始化ServerSocketChannel，注册到Selector多路复用轮询组件上去

final ChannelFuture initAndRegister() {

Channel channel = null;

try {

// 有一个默认的ChannelFactory

// 估计就是来创建ServerSocketChannel的，ServerSocketChannel创建好了

// 而且配置好了blocking是false

// 准备好了ServerSocketChannel将要关注的网络事件，OP\_ACCEPT

channel = channelFactory.newChannel();

// 接下来必然是让这个ServerSocketChannel去监听某个端口号，设置一些网络参数

// 然后再让这个ServerSocketChannel注册到Selector上去，关注他的OP\_ACCEPT网络事件，不停的轮询他

init(channel);

// 最最核心的，还是关注他核心的架构设计、底层技术的处理

} catch (Throwable t) {

if (channel != null) {

// channel can be null if newChannel crashed (eg SocketException("too many open files"))

channel.unsafe().closeForcibly();

// as the Channel is not registered yet we need to force the usage of the GlobalEventExecutor

return new DefaultChannelPromise(channel, GlobalEventExecutor.INSTANCE).setFailure(t);

}

// as the Channel is not registered yet we need to force the usage of the GlobalEventExecutor

return new DefaultChannelPromise(new FailedChannel(), GlobalEventExecutor.INSTANCE).setFailure(t);

}

// 拿出来了之前创建的一个EventLoopGroup，猜测，独立的线程，复杂用Selector轮询各种channel的网络事件

// 他把ServerSocketChannel注册到了一个EventLoopGroup上去

// 这里的意思，是不是就是说让EventLoopGroup中的独立线程采用一个Selector来注册channel，以及轮询事件

ChannelFuture regFuture = config().group().register(channel);

if (regFuture.cause() != null) {

if (channel.isRegistered()) {

channel.close();

} else {

channel.unsafe().closeForcibly();

}

}

// If we are here and the promise is not failed, it's one of the following cases:

// 1) If we attempted registration from the event loop, the registration has been completed at this point.

// i.e. It's safe to attempt bind() or connect() now because the channel has been registered.

// 2) If we attempted registration from the other thread, the registration request has been successfully

// added to the event loop's task queue for later execution.

// i.e. It's safe to attempt bind() or connect() now:

// because bind() or connect() will be executed \*after\* the scheduled registration task is executed

// because register(), bind(), and connect() are all bound to the same thread.

return regFuture;

}

@Override

public T newChannel() {

try {

// constructor，反射方式，NioServerSocketChannel.class

// 这个类是我们自己设置的，它是Netty提供的一个类

// 他一定是Netty自己把ServerSocketChannel给封装了一下，NioServerSocketChannel

// NettyServerSocketChannel

return constructor.newInstance();

} catch (Throwable t) {

throw new ChannelException("Unable to create Channel from class " + constructor.getDeclaringClass(), t);

}

}

private static ServerSocketChannel newSocket(SelectorProvider provider) {

try {

/\*\*

\* Use the {@link SelectorProvider} to open {@link SocketChannel} and so remove condition in

\* {@link SelectorProvider#provider()} which is called by each ServerSocketChannel.open() otherwise.

\*

\* See <a href="https://github.com/netty/netty/issues/2308">#2308</a>.

\*/

// 直接就是通过原生的NIO的API，创建了一个ServerSocketChannel

return provider.openServerSocketChannel();

} catch (IOException e) {

throw new ChannelException(

"Failed to open a server socket.", e);

}

}

@Override

void init(Channel channel) throws Exception {

// 对ServerSocketChannel进行网络参数的设置

final Map<ChannelOption<?>, Object> options = options0();

synchronized (options) {

setChannelOptions(channel, options, logger);

}

// 初始化了一些属性

final Map<AttributeKey<?>, Object> attrs = attrs0();

synchronized (attrs) {

for (Entry<AttributeKey<?>, Object> e: attrs.entrySet()) {

@SuppressWarnings("unchecked")

AttributeKey<Object> key = (AttributeKey<Object>) e.getKey();

channel.attr(key).set(e.getValue());

}

}

ChannelPipeline p = channel.pipeline();

final EventLoopGroup currentChildGroup = childGroup;

final ChannelHandler currentChildHandler = childHandler;

final Entry<ChannelOption<?>, Object>[] currentChildOptions;

final Entry<AttributeKey<?>, Object>[] currentChildAttrs;

synchronized (childOptions) {

currentChildOptions = childOptions.entrySet().toArray(newOptionArray(0));

}

synchronized (childAttrs) {

currentChildAttrs = childAttrs.entrySet().toArray(newAttrArray(0));

}

// 就是对网络请求处理链路中加入一个自己内置的一个处理逻辑

// 初始化了网络请求的处理链路

p.addLast(new ChannelInitializer<Channel>() {

@Override

public void initChannel(final Channel ch) throws Exception {

final ChannelPipeline pipeline = ch.pipeline();

ChannelHandler handler = config.handler();

if (handler != null) {

pipeline.addLast(handler);

}

ch.eventLoop().execute(new Runnable() {

@Override

public void run() {

pipeline.addLast(new ServerBootstrapAcceptor(

ch, currentChildGroup, currentChildHandler, currentChildOptions, currentChildAttrs));

}

});

}

});

}

@Override

public ChannelFuture register(Channel channel) {

// 所谓的EventLoopGroup就是可以不停的获取出来很多的线程

// 对应着一个线程池的，线程的数量是固定的，cpu核数 \* 2

// 调用next()方法轮询获取出来这个线程池里的下一个线程，推测就是这个意思

// 123456,6个线程，第一次Next()是线程1，第二次next()是线程2，第三次next()是线程3

// 他的next()方法，就是每次调用返回一个线程给你，让你通过这个线程去使用线程自己的Selector

// 注册对应的channel，不管是ServerSocketChannel，还是普通的SocketChannel

// 让这个线程去负责对应的这些channel的网络事件的轮询

// 不知道大家是否还记得，就是之前我们写Reactor模型的时候，也是类似的思路，我们搞了一组Processor线程

// 每次有新的网络连接进入，都是轮询各个Processor线程，保证每个线程都可以获取到一定的连接

return next().register(channel);

}

@Override

public EventExecutor next() {

return executors[Math.abs(idx.getAndIncrement() % executors.length)];

}

@Override

protected void doRegister() throws Exception {

boolean selected = false;

for (;;) {

try {

// javaChannel() -> ServerSocketChannel

// register(Selector, 0, this)

selectionKey = javaChannel().register(eventLoop().unwrappedSelector(), 0, this);

return;

} catch (CancelledKeyException e) {

if (!selected) {

// Force the Selector to select now as the "canceled" SelectionKey may still be

// cached and not removed because no Select.select(..) operation was called yet.

eventLoop().selectNow();

selected = true;

} else {

// We forced a select operation on the selector before but the SelectionKey is still cached

// for whatever reason. JDK bug ?

throw e;

}

}

}

}

# **File Path: /Users/xiaotingting/Downloads/学习资料/12\_Netty核心功能精讲以及核心源码剖析/019~030资料/024\_补充一个细节：ServierSocketChannel如何绑定监听端口？/源码分析.txt**

// 方法的名字，初始化ServerSocketChannel，注册到Selector多路复用轮询组件上去

final ChannelFuture initAndRegister() {

Channel channel = null;

try {

// 有一个默认的ChannelFactory

// 估计就是来创建ServerSocketChannel的，ServerSocketChannel创建好了

// 而且配置好了blocking是false

// 准备好了ServerSocketChannel将要关注的网络事件，OP\_ACCEPT

channel = channelFactory.newChannel();

// 接下来必然是让这个ServerSocketChannel去监听某个端口号，设置一些网络参数

// 然后再让这个ServerSocketChannel注册到Selector上去，关注他的OP\_ACCEPT网络事件，不停的轮询他

init(channel);

// 最最核心的，还是关注他核心的架构设计、底层技术的处理

} catch (Throwable t) {

if (channel != null) {

// channel can be null if newChannel crashed (eg SocketException("too many open files"))

channel.unsafe().closeForcibly();

// as the Channel is not registered yet we need to force the usage of the GlobalEventExecutor

return new DefaultChannelPromise(channel, GlobalEventExecutor.INSTANCE).setFailure(t);

}

// as the Channel is not registered yet we need to force the usage of the GlobalEventExecutor

return new DefaultChannelPromise(new FailedChannel(), GlobalEventExecutor.INSTANCE).setFailure(t);

}

// 拿出来了之前创建的一个EventLoopGroup，猜测，独立的线程，复杂用Selector轮询各种channel的网络事件

// 他把ServerSocketChannel注册到了一个EventLoopGroup上去

// 这里的意思，是不是就是说让EventLoopGroup中的独立线程采用一个Selector来注册channel，以及轮询事件

ChannelFuture regFuture = config().group().register(channel);

if (regFuture.cause() != null) {

if (channel.isRegistered()) {

channel.close();

} else {

channel.unsafe().closeForcibly();

}

}

// If we are here and the promise is not failed, it's one of the following cases:

// 1) If we attempted registration from the event loop, the registration has been completed at this point.

// i.e. It's safe to attempt bind() or connect() now because the channel has been registered.

// 2) If we attempted registration from the other thread, the registration request has been successfully

// added to the event loop's task queue for later execution.

// i.e. It's safe to attempt bind() or connect() now:

// because bind() or connect() will be executed \*after\* the scheduled registration task is executed

// because register(), bind(), and connect() are all bound to the same thread.

return regFuture;

}

@Override

public T newChannel() {

try {

// constructor，反射方式，NioServerSocketChannel.class

// 这个类是我们自己设置的，它是Netty提供的一个类

// 他一定是Netty自己把ServerSocketChannel给封装了一下，NioServerSocketChannel

// NettyServerSocketChannel

return constructor.newInstance();

} catch (Throwable t) {

throw new ChannelException("Unable to create Channel from class " + constructor.getDeclaringClass(), t);

}

}

private static ServerSocketChannel newSocket(SelectorProvider provider) {

try {

/\*\*

\* Use the {@link SelectorProvider} to open {@link SocketChannel} and so remove condition in

\* {@link SelectorProvider#provider()} which is called by each ServerSocketChannel.open() otherwise.

\*

\* See <a href="https://github.com/netty/netty/issues/2308">#2308</a>.

\*/

// 直接就是通过原生的NIO的API，创建了一个ServerSocketChannel

return provider.openServerSocketChannel();

} catch (IOException e) {

throw new ChannelException(

"Failed to open a server socket.", e);

}

}

@Override

void init(Channel channel) throws Exception {

// 对ServerSocketChannel进行网络参数的设置

final Map<ChannelOption<?>, Object> options = options0();

synchronized (options) {

setChannelOptions(channel, options, logger);

}

// 初始化了一些属性

final Map<AttributeKey<?>, Object> attrs = attrs0();

synchronized (attrs) {

for (Entry<AttributeKey<?>, Object> e: attrs.entrySet()) {

@SuppressWarnings("unchecked")

AttributeKey<Object> key = (AttributeKey<Object>) e.getKey();

channel.attr(key).set(e.getValue());

}

}

ChannelPipeline p = channel.pipeline();

final EventLoopGroup currentChildGroup = childGroup;

final ChannelHandler currentChildHandler = childHandler;

final Entry<ChannelOption<?>, Object>[] currentChildOptions;

final Entry<AttributeKey<?>, Object>[] currentChildAttrs;

synchronized (childOptions) {

currentChildOptions = childOptions.entrySet().toArray(newOptionArray(0));

}

synchronized (childAttrs) {

currentChildAttrs = childAttrs.entrySet().toArray(newAttrArray(0));

}

// 就是对网络请求处理链路中加入一个自己内置的一个处理逻辑

// 初始化了网络请求的处理链路

p.addLast(new ChannelInitializer<Channel>() {

@Override

public void initChannel(final Channel ch) throws Exception {

final ChannelPipeline pipeline = ch.pipeline();

ChannelHandler handler = config.handler();

if (handler != null) {

pipeline.addLast(handler);

}

ch.eventLoop().execute(new Runnable() {

@Override

public void run() {

pipeline.addLast(new ServerBootstrapAcceptor(

ch, currentChildGroup, currentChildHandler, currentChildOptions, currentChildAttrs));

}

});

}

});

}

@Override

public ChannelFuture register(Channel channel) {

// 所谓的EventLoopGroup就是可以不停的获取出来很多的线程

// 对应着一个线程池的，线程的数量是固定的，cpu核数 \* 2

// 调用next()方法轮询获取出来这个线程池里的下一个线程，推测就是这个意思

// 123456,6个线程，第一次Next()是线程1，第二次next()是线程2，第三次next()是线程3

// 他的next()方法，就是每次调用返回一个线程给你，让你通过这个线程去使用线程自己的Selector

// 注册对应的channel，不管是ServerSocketChannel，还是普通的SocketChannel

// 让这个线程去负责对应的这些channel的网络事件的轮询

// 不知道大家是否还记得，就是之前我们写Reactor模型的时候，也是类似的思路，我们搞了一组Processor线程

// 每次有新的网络连接进入，都是轮询各个Processor线程，保证每个线程都可以获取到一定的连接

return next().register(channel);

}

@Override

public EventExecutor next() {

return executors[Math.abs(idx.getAndIncrement() % executors.length)];

}

@Override

protected void doRegister() throws Exception {

boolean selected = false;

for (;;) {

try {

// javaChannel() -> ServerSocketChannel

// register(Selector, 0, this)

selectionKey = javaChannel().register(eventLoop().unwrappedSelector(), 0, this);

return;

} catch (CancelledKeyException e) {

if (!selected) {

// Force the Selector to select now as the "canceled" SelectionKey may still be

// cached and not removed because no Select.select(..) operation was called yet.

eventLoop().selectNow();

selected = true;

} else {

// We forced a select operation on the selector before but the SelectionKey is still cached

// for whatever reason. JDK bug ?

throw e;

}

}

}

}

# **File Path: /Users/xiaotingting/Downloads/学习资料/12\_Netty核心功能精讲以及核心源码剖析/019~030资料/021\_Netty是如何对ServerSocketChannel进行初始化的？/源码分析.txt**

// 方法的名字，初始化ServerSocketChannel，注册到Selector多路复用轮询组件上去

final ChannelFuture initAndRegister() {

Channel channel = null;

try {

// 有一个默认的ChannelFactory

// 估计就是来创建ServerSocketChannel的，ServerSocketChannel创建好了

// 而且配置好了blocking是false

// 准备好了ServerSocketChannel将要关注的网络事件，OP\_ACCEPT

channel = channelFactory.newChannel();

// 接下来必然是让这个ServerSocketChannel去监听某个端口号，设置一些网络参数

// 然后再让这个ServerSocketChannel注册到Selector上去，关注他的OP\_ACCEPT网络事件，不停的轮询他

init(channel);

} catch (Throwable t) {

if (channel != null) {

// channel can be null if newChannel crashed (eg SocketException("too many open files"))

channel.unsafe().closeForcibly();

// as the Channel is not registered yet we need to force the usage of the GlobalEventExecutor

return new DefaultChannelPromise(channel, GlobalEventExecutor.INSTANCE).setFailure(t);

}

// as the Channel is not registered yet we need to force the usage of the GlobalEventExecutor

return new DefaultChannelPromise(new FailedChannel(), GlobalEventExecutor.INSTANCE).setFailure(t);

}

ChannelFuture regFuture = config().group().register(channel);

if (regFuture.cause() != null) {

if (channel.isRegistered()) {

channel.close();

} else {

channel.unsafe().closeForcibly();

}

}

// If we are here and the promise is not failed, it's one of the following cases:

// 1) If we attempted registration from the event loop, the registration has been completed at this point.

// i.e. It's safe to attempt bind() or connect() now because the channel has been registered.

// 2) If we attempted registration from the other thread, the registration request has been successfully

// added to the event loop's task queue for later execution.

// i.e. It's safe to attempt bind() or connect() now:

// because bind() or connect() will be executed \*after\* the scheduled registration task is executed

// because register(), bind(), and connect() are all bound to the same thread.

return regFuture;

}

@Override

public T newChannel() {

try {

// constructor，反射方式，NioServerSocketChannel.class

// 这个类是我们自己设置的，它是Netty提供的一个类

// 他一定是Netty自己把ServerSocketChannel给封装了一下，NioServerSocketChannel

// NettyServerSocketChannel

return constructor.newInstance();

} catch (Throwable t) {

throw new ChannelException("Unable to create Channel from class " + constructor.getDeclaringClass(), t);

}

}

private static ServerSocketChannel newSocket(SelectorProvider provider) {

try {

/\*\*

\* Use the {@link SelectorProvider} to open {@link SocketChannel} and so remove condition in

\* {@link SelectorProvider#provider()} which is called by each ServerSocketChannel.open() otherwise.

\*

\* See <a href="https://github.com/netty/netty/issues/2308">#2308</a>.

\*/

// 直接就是通过原生的NIO的API，创建了一个ServerSocketChannel

return provider.openServerSocketChannel();

} catch (IOException e) {

throw new ChannelException(

"Failed to open a server socket.", e);

}

}

@Override

void init(Channel channel) throws Exception {

// 对ServerSocketChannel进行网络参数的设置

final Map<ChannelOption<?>, Object> options = options0();

synchronized (options) {

setChannelOptions(channel, options, logger);

}

// 初始化了一些属性

final Map<AttributeKey<?>, Object> attrs = attrs0();

synchronized (attrs) {

for (Entry<AttributeKey<?>, Object> e: attrs.entrySet()) {

@SuppressWarnings("unchecked")

AttributeKey<Object> key = (AttributeKey<Object>) e.getKey();

channel.attr(key).set(e.getValue());

}

}

ChannelPipeline p = channel.pipeline();

final EventLoopGroup currentChildGroup = childGroup;

final ChannelHandler currentChildHandler = childHandler;

final Entry<ChannelOption<?>, Object>[] currentChildOptions;

final Entry<AttributeKey<?>, Object>[] currentChildAttrs;

synchronized (childOptions) {

currentChildOptions = childOptions.entrySet().toArray(newOptionArray(0));

}

synchronized (childAttrs) {

currentChildAttrs = childAttrs.entrySet().toArray(newAttrArray(0));

}

// 就是对网络请求处理链路中加入一个自己内置的一个处理逻辑

// 初始化了网络请求的处理链路

p.addLast(new ChannelInitializer<Channel>() {

@Override

public void initChannel(final Channel ch) throws Exception {

final ChannelPipeline pipeline = ch.pipeline();

ChannelHandler handler = config.handler();

if (handler != null) {

pipeline.addLast(handler);

}

ch.eventLoop().execute(new Runnable() {

@Override

public void run() {

pipeline.addLast(new ServerBootstrapAcceptor(

ch, currentChildGroup, currentChildHandler, currentChildOptions, currentChildAttrs));

}

});

}

});

}

# **File Path: /Users/xiaotingting/Downloads/学习资料/12\_Netty核心功能精讲以及核心源码剖析/019~030资料/019\_将Netty版本调整回4.x，避免使用废弃的5.x版本/源码分析.txt**

// 方法的名字，初始化ServerSocketChannel，注册到Selector多路复用轮询组件上去

final ChannelFuture initAndRegister() {

Channel channel = null;

try {

channel = channelFactory.newChannel();

init(channel);

} catch (Throwable t) {

if (channel != null) {

// channel can be null if newChannel crashed (eg SocketException("too many open files"))

channel.unsafe().closeForcibly();

// as the Channel is not registered yet we need to force the usage of the GlobalEventExecutor

return new DefaultChannelPromise(channel, GlobalEventExecutor.INSTANCE).setFailure(t);

}

// as the Channel is not registered yet we need to force the usage of the GlobalEventExecutor

return new DefaultChannelPromise(new FailedChannel(), GlobalEventExecutor.INSTANCE).setFailure(t);

}

ChannelFuture regFuture = config().group().register(channel);

if (regFuture.cause() != null) {

if (channel.isRegistered()) {

channel.close();

} else {

channel.unsafe().closeForcibly();

}

}

// If we are here and the promise is not failed, it's one of the following cases:

// 1) If we attempted registration from the event loop, the registration has been completed at this point.

// i.e. It's safe to attempt bind() or connect() now because the channel has been registered.

// 2) If we attempted registration from the other thread, the registration request has been successfully

// added to the event loop's task queue for later execution.

// i.e. It's safe to attempt bind() or connect() now:

// because bind() or connect() will be executed \*after\* the scheduled registration task is executed

// because register(), bind(), and connect() are all bound to the same thread.

return regFuture;

}

# **File Path: /Users/xiaotingting/Downloads/学习资料/12\_Netty核心功能精讲以及核心源码剖析/019~030资料/026\_如果发现了客户端发起的连接事件是如何进行处理的？/源码分析.txt**

@Override

protected int doReadMessages(List<Object> buf) throws Exception {

// 跟客户端建立连接，获取到对应的SocketChannel

SocketChannel ch = SocketUtils.accept(javaChannel());

try {

if (ch != null) {

// 把建立好的连接缓存在List里面了

// 大家知道为什么大名鼎鼎的Kafka，网络通信，为啥不用Netty呢？

// 如果你用Netty，开源框架，你必须精通他的源码，而且在适当的时候可以改造他的源码解决问题

// 对于Kafka这种中间件系统，对于底层的网络必须自己把控

// 如果网络框架一旦出了问题，必须自己可以从源码级别来解决，Kafka依赖Netty

// 一旦Netty有问题，源码层面Kafka作者没法来把控，就彻底Kafka也废掉了

// 处于这样的原因，Kafka是直接基于NIO自己研发网络通信组件的

// 分布式海量小文件存储系统，非常适合用NIO自己开发网络通信组件，哪怕是有一些问题，源码自己把控

buf.add(new NioSocketChannel(this, ch));

return 1;

}

} catch (Throwable t) {

logger.warn("Failed to create a new channel from an accepted socket.", t);

try {

ch.close();

} catch (Throwable t2) {

logger.warn("Failed to close a socket.", t2);

}

}

return 0;

}

# **File Path: /Users/xiaotingting/Downloads/学习资料/12\_Netty核心功能精讲以及核心源码剖析/001~005资料/003\_未来课程计划：三个基于Netty的真实大用户量项目实战/笔记.doc**

一周多的时间，课程设计，不比课程录制更简单，慢慢梳理，技术先讲，什么技术后讲，各种技术怎么串联起来，怎么去配合对应的项目

1、基于Netty以及自研DFS的仿钉钉云盘项目实战

互联网公司里不同的应用场景，仿钉钉的云盘项目，钉钉，但凡你是用办公通讯工具，有一个场景非常的场景，基于云盘的文件上传和分享。比如说你现在要上传一个文件，后续你根据自己的需求，要把文件分享给不同的同事

你跟一个同事在聊天，直接就把一个文件传输给他，文件会上传到云盘里去占据你的个人空加你，但是直接这个同事可以看到这个文件的分享链接/图标，然后他就可以去下载

基于Netty去开发客户端和服务端，文件从用户那里过来的一个上传，实际上这个文件最终会进入到分布式海量小文件存储系统里去，人家下载文件，就从分布式海量小文件存储系统里去读取

大量的客户端，百万客户端，都连接到服务器，进行文件的上传，天然和真实的一个Netty使用场景，维护一个连接，做耗时的网络通信和数据传输

2、基于Netty的百万日活APP即时通信系统实战

互联网公司里非常常见的一个应用，体育运动APP，在线教育APP，社交APP，在线问诊APP，他都会有APP内部的一个即时通讯系统，APP内部，你可以给不同的用户发送消息，去聊天 ，一群人可以在一起聊天

点对点，一个人跟一个人去聊天

非常适合用Netty来开发，大量的客户端跟服务器维持长连接，如果有一个人要跟另外一个人说话，发送一条消息过去，直接会通过长连接推送到服务器，服务器通过长连接推送到别人的客户端APP上去

3、基于Netty的支撑1亿客户端的消息推送系统实战

手机经常会收到一些APP的push消息，突然会弹出来一些框，在手机顶部，就是各种APP推送的消息，你的服务器需要向上亿客户端push消息，也得基于Netty来开发

很多同学如果学电商的系统实战，但是你自己手头做的可能就是一些很low的内部系统，OA，工厂管理，财务系统，电商？另外一个，做大量的基础系统，分布式海量小文件系统，分布式微服务注册中心，仿钉钉的云盘系统，即时通信系统，消息PUSH系统，直接可以说自己是基础架构系统的架构师，或者资深工程师，都可以

# **File Path: /Users/xiaotingting/Downloads/学习资料/12\_Netty核心功能精讲以及核心源码剖析/001~005资料/002\_未来课程计划：Netty核心功能精讲以及核心源码剖析/笔记.doc**

集合、并发、IO、网络、NIO，都搞定了

Netty -> 基于NIO的网络通信的框架，高性能、高并发，很多时候都会基于Netty去研发一些底层的基础系统

Netty核心功能精讲，做几个demo，剖析他的核心源码，分析他的架构设计之道，高性能、高可靠、高并发

基于Netty开发很多互联网公司里的应对海量用户、大量客户端的一些基础系统的项目

# **File Path: /Users/xiaotingting/Downloads/学习资料/12\_Netty核心功能精讲以及核心源码剖析/001~005资料/005\_未来课程计划：完成两个工业级的中间件项目的架构重构/笔记.doc**

netty的三个基础系统，dubbo rpc框架源码，两个中间件项目给搞完，50%的微服务注册中心项目，架构做大量的重构，补充进去基于netty的网络通信框架；80%的分布式海量小文件存储系统，精讲zookeeper技术，引入这个系统，做元数据节点改造成分布式和高可用的，两个项目完成

底层技术极为扎实，并发源码，netty源码，大量的基础系统、中间件系统的项目实战，积累大量的底层技术的实战经验

技术功底极为深厚，基础架构经验，2020年开春，或者2019年年底出去找工作，今年的一大堆的基础架构的项目，都可以派上用场了，哪怕你做的项目很low，但是你可以用已经讲解过的亿级流量的缓存架构，分布式架构，基础架构，拿出去说事儿

中小公司的架构师，就没问题了，年薪五六十万，去面大公司的资深工程师，妥妥的，阿里P6+，年薪五六十万

明年，高并发、高性能、高可用的业务系统架构，对应的源码都搞完，1年多的时间

3年多的课，100%吃透 -> 有我指导跳槽两次左右，经历一些知名点的公司 ->

中小公司的首席架构，独角兽（估值几十亿美金的公司），知名创业公司（估值几亿美金），高级架构，资深架构，七八十万起步；大公司，P7，学历、履历，七八十万起步的

# **File Path: /Users/xiaotingting/Downloads/学习资料/12\_Netty核心功能精讲以及核心源码剖析/001~005资料/001\_课程计划说明、未来年薪展望以及职业发展规划/笔记.doc**

电商系统 -> 分布式架构 -> 分布式事务、分布式锁、分布式服务框架

集合、并发、IO、网络 -> 两个自研的中间件项目，分布式微服务注册中心（50%），分布式海量小文件存储系统（80%）

Netty -> 带出来一大堆的项目

---> 你有能力做基础架构的架构师，任何你公司里需要的各种基础架构的系统，或者是底层的系统，或者是需要自研的中间件系统，你都可以带团队，来做架构设计，基于底层技术来进行研发

---> 你有足够的底层技术和功底，来研究后续技术的源码，Kafka（大数据架构课）、RocketMQ、MyCat、Tomcat

很多很多的系统，中国没有那么多的高并发系统，大部分的系统，微服务+分布式，一般来说就够了，微服务架构，配置中心、日志中心、监控中心、服务治理、API治理

缓存、消息、搜索、海量数据，技术上的难度没有那么的高

----> 小公司的架构师的水平 + 大公司的资深工程师的水平 -> 50万~70万左右

----> 小公司的首席架构/CTO的水平 + 大公司的P7+的水平（P8）-> 七八十万 ~ 100万左右

Kafka的源码怎么看？并发、集合、IO、网络

RocketMQ的源码怎么看？并发、集合、IO、网络

亿级流量的电商详情页的缓存系统 -> Redis的使用 + 常规性的缓存架构的设计方案

# **File Path: /Users/xiaotingting/Downloads/学习资料/12\_Netty核心功能精讲以及核心源码剖析/001~005资料/004\_未来课程计划：从Dubbo源码剖析到Netty在RPC框架中的实战/笔记.doc**

带大家手写RPC框架，类似Dubbo，之前Dubbo非常的不活跃了，很多公司其实都会封装自己的版本，但是现在阿里重启Dubbo了，手写RPC框架就没意义了，Netty除了中间件系统、基础系统的研发，用在RPC框架里，是分布式系统通信框架，网络通信，RPC框架都是用的Netty来做网络通信

直接来读Dubbo的源码，然后剖析类似Netty的网络通信框架在RPC框架里的运用和实战

# **File Path: /Users/xiaotingting/Downloads/学习资料/12\_Netty核心功能精讲以及核心源码剖析/006~018资料/018\_作业：对比NIO程序自己去探索Netty Server的启动过程/笔记.doc**

为了让大家开始真正的学会“渔”的能力，不要总是只是知道跟着我来学，每周都给大家留一些作业，让大家课后自己去做，有问题可以QQ找我来提问，也可以在学员群里跟大家去交流，都可以

对比一下NIO Server程序启动的过程，你可以自己顺着Netty的源码先去预习，先尝试自己去分析，Netty Server启动的过程，是涉及到了哪些东西和环节，用processon画画图，自己先预习和思考

下周我会把Netty Server启动初始化的源码，Netty Client初始化的源码，互相发送请求，处理请求，返回响应的这个过程，源码，给大家快速的梳理一遍

# **File Path: /Users/xiaotingting/Downloads/学习资料/12\_Netty核心功能精讲以及核心源码剖析/006~018资料/013\_运行我们的Netty第一个入门程序调试一下看看效果/笔记.doc**

# **File Path: /Users/xiaotingting/Downloads/学习资料/12\_Netty核心功能精讲以及核心源码剖析/006~018资料/016\_Netty的EventLoopGroup线程池是如何进行初始化的？/笔记.doc**

每个线程就对应了一个NioEventLoop

很多的线程，每个线程叫做NioEventLoop，每个线程都会负责一部分的客户端连接的SocketChannel，对这些SocketChannel都会注册在线程自己的Selector中，每个线程通过自己的Selector去轮询（Loop）他负责的这一批客户端连接的网络请求事件

NioEventLoop，负责轮询Nio事件的线程，轮询多个客户端连接的Nio事件

线程池的初始化，NioEventLoopGroup，cpu核 \* 2 = 线程数量，每个线程就对应一个NioEventLoop，有一个自己的Selector，每个线程就通过Selector负责一批SocketChannel（客户端连接）的Nio网络事件的轮询

# **File Path: /Users/xiaotingting/Downloads/学习资料/12\_Netty核心功能精讲以及核心源码剖析/006~018资料/006\_使用Java原生NIO进行网络编程有哪些缺陷？/笔记.doc**

NIO网络编程技术，大量的进行了项目实战，自己封装和开发了复杂的网络通信的程序

连接异常、网络闪断、半包读写、网络拥塞、异常码流，一大堆生产问题需要自己进行大量的编程来实现

如果要直接使用原生NIO进行开发，比如大名鼎鼎的Kafka就是自行基于NIO封装的网络通信框架，那么需要对上面的问题进行处理，比如我们之前的项目，就必须考虑到网络中断，连接异常，粘包拆包，请求排队，等等，一大堆的问题

Kafka可以这么做，我们可以这么做，但是不见得所有普通人都可以这么做，别看我轻松把代码写了出来，但是其实对大部分人来说，这是很困难的一件事情

如果你技术很牛，能hold住，其实可以自己基于Java NIO来封装和定制开发；但是如果说对网络通信这块不是很了解的话，或者说没有那么的精通，建议采用NIO框架来进行编程，框架就是把常见的各种生产问题都考虑到了，都做了优化

# **File Path: /Users/xiaotingting/Downloads/学习资料/12\_Netty核心功能精讲以及核心源码剖析/006~018资料/007\_大名鼎鼎的Netty到底有哪些优点让我们去使用它？/笔记.doc**

NIO框架：Netty，封装了底层很多复杂的网络通信细节，让你开发程序非常的简单，它提供了很多的高阶的功能，可以让你基于他开发出来非常复杂的网络通信程序

NIO API，有点复杂，Selector、Channel、SelectionKey、Buffer

Netty简化了网络编程的API

数据传输，直接基于Buffer封装成二进制字节流的数据格式，网络通信的时候，需要支持不同的协议，而且可以对自定义的数据结构进行编码和解码，Netty都支持了

Netty还提供了很灵活的扩展的功能

Netty：高性能、高并发/高吞吐、高可靠

Netty：大量的商业项目都使用了Netty，所以经过了复杂生产环境的验证，基本上来说作为一个开源项目非常的成熟

# **File Path: /Users/xiaotingting/Downloads/学习资料/12\_Netty核心功能精讲以及核心源码剖析/006~018资料/011\_一步一步动手开发Netty入门程序的客户端/笔记.doc**

# **File Path: /Users/xiaotingting/Downloads/学习资料/12\_Netty核心功能精讲以及核心源码剖析/006~018资料/012\_一步一步动手开发Netty入门程序的客户端（2）/笔记.doc**

# **File Path: /Users/xiaotingting/Downloads/学习资料/12\_Netty核心功能精讲以及核心源码剖析/006~018资料/008\_搭建一个Netty入门程序的工程以及配置好其Maven依赖/笔记.doc**

NIO，必须先写一些Hello World程序

# **File Path: /Users/xiaotingting/Downloads/学习资料/12\_Netty核心功能精讲以及核心源码剖析/006~018资料/010\_一步一步动手开发Netty入门程序的服务端（2）/笔记.doc**

初次学习Netty的新手，学习Netty的陡峭程度可能比Java NIO还要高一些

但是Java NIO是原生的，其实你会发现，Java NIO的那套模型因为省略了很多复杂的概念，所以去芜存菁，简洁，学习的难度曲线很平顺，很好理解，很好上手

Netty学习陡峭曲线是很陡峭的，学习挺难理解的，好用，容易用，功能强大，开发生产级的功能和程序会简答很多，理解他比较难，理解他需要结合他的源码去理解

直接配合源码去理解Netty他的架构原理，底层细节，彻底搞透Netty内部到底是怎么回事，接下来再来基于Netty去开发一些基础系统，做一些项目

# **File Path: /Users/xiaotingting/Downloads/学习资料/12\_Netty核心功能精讲以及核心源码剖析/006~018资料/014\_把Netty的源码包下载追加到Eclipse中方便我们调试源码/笔记.doc**

不解释，直接开始硬调源码

# **File Path: /Users/xiaotingting/Downloads/学习资料/12\_Netty核心功能精讲以及核心源码剖析/006~018资料/009\_一步一步动手开发Netty入门程序的服务端/笔记.doc**

# **File Path: /Users/xiaotingting/Downloads/学习资料/12\_Netty核心功能精讲以及核心源码剖析/006~018资料/017\_那么ServerBootstrap又是个什么东西呢？/笔记.doc**

# **File Path: /Users/xiaotingting/Downloads/学习资料/12\_Netty核心功能精讲以及核心源码剖析/006~018资料/015\_先来看看让人懵逼的EventLoopGroup是什么？/笔记.doc**

EventLoopGroup，本质在底层就是一个线程池的这么一个东西，可以让你从里面获取新的线程，以及他会负责管理这些线程的生命周期

他的线程池里的默认的线程数量，实际上就是你的机器可用的cpu核的数量 \* 2，比如说你是4核8G的机器，那么默认线程池的线程数量就是8个，最小最小也得是1，起码线程池里得有1个线程

ThreadFactory是null，肯定是会用默认的机制去创建新的线程出来

ThreadPerTaskExecutor + 默认的ThreadFactory

你有多少个线程，就会对应一个EventExecutor数组，每个线程就对应一个EventExecutor

# **File Path: /Users/xiaotingting/Downloads/学习资料/12\_Netty核心功能精讲以及核心源码剖析/019~030资料/025\_Netty的线程是如何轮询ServerSocketChannel的网络连接事件的？/笔记.doc**

select方法，大体上可以认为在一个无限循环的方法里，不停的去等待是否有新的网络事件的发生，如果有就返回

# **File Path: /Users/xiaotingting/Downloads/学习资料/12\_Netty核心功能精讲以及核心源码剖析/019~030资料/023\_用一张图来展示出ServerSocketChannel初始化的过程/笔记.doc**

netty官方已经废弃掉了最新的5.x版本，一直在更新的还是4.x版本

event loop，轮询网络事件，线程

这里面的东西肯定是负责通过一个Selector去轮询多个网络连接的事件

ServerSocketChannel -> 端口号，然后通过一个独立的线程（Acceptor）去使用Selector去轮询这个ServerSocketChannel是否有连接的事件接入，如果有的话，跟客户端建立连接，然后将客户端的连接分发给Processor线程

每个Processor线程应该是负责处理一部分客户端的连接，使用自己的Selector不断的轮询各个客户端连接的网络事件，收到了请求解析出来

请求应该是交给Handler线程去进行处理，处理请求，设置响应

Processor把响应发送给对应的客户端就可以了

# **File Path: /Users/xiaotingting/Downloads/学习资料/12\_Netty核心功能精讲以及核心源码剖析/019~030资料/027\_Server端对建立好的客户端连接是交给谁来轮询网络请求的？/笔记.doc**

就是ServerSocketChannel的pipeline有一个初始化的过程，他在里面加入了一个自己的Handler，这个Handler就是专门负责处理连接的，他就会在这里来进行处理

# **File Path: /Users/xiaotingting/Downloads/学习资料/12\_Netty核心功能精讲以及核心源码剖析/019~030资料/020\_对照原生NIO代码看看Netty如何创建ServerSocketChannel/笔记.doc**

netty官方已经废弃掉了最新的5.x版本，一直在更新的还是4.x版本

event loop，轮询网络事件，线程

这里面的东西肯定是负责通过一个Selector去轮询多个网络连接的事件

ServerSocketChannel -> 端口号，然后通过一个独立的线程（Acceptor）去使用Selector去轮询这个ServerSocketChannel是否有连接的事件接入，如果有的话，跟客户端建立连接，然后将客户端的连接分发给Processor线程

每个Processor线程应该是负责处理一部分客户端的连接，使用自己的Selector不断的轮询各个客户端连接的网络事件，收到了请求解析出来

请求应该是交给Handler线程去进行处理，处理请求，设置响应

Processor把响应发送给对应的客户端就可以了

# **File Path: /Users/xiaotingting/Downloads/学习资料/12\_Netty核心功能精讲以及核心源码剖析/019~030资料/022\_如何将ServerSocketChannel注册到Selector由独立线程轮询？/笔记.doc**

netty官方已经废弃掉了最新的5.x版本，一直在更新的还是4.x版本

event loop，轮询网络事件，线程

这里面的东西肯定是负责通过一个Selector去轮询多个网络连接的事件

ServerSocketChannel -> 端口号，然后通过一个独立的线程（Acceptor）去使用Selector去轮询这个ServerSocketChannel是否有连接的事件接入，如果有的话，跟客户端建立连接，然后将客户端的连接分发给Processor线程

每个Processor线程应该是负责处理一部分客户端的连接，使用自己的Selector不断的轮询各个客户端连接的网络事件，收到了请求解析出来

请求应该是交给Handler线程去进行处理，处理请求，设置响应

Processor把响应发送给对应的客户端就可以了

# **File Path: /Users/xiaotingting/Downloads/学习资料/12\_Netty核心功能精讲以及核心源码剖析/019~030资料/024\_补充一个细节：ServierSocketChannel如何绑定监听端口？/笔记.doc**

netty官方已经废弃掉了最新的5.x版本，一直在更新的还是4.x版本

event loop，轮询网络事件，线程

这里面的东西肯定是负责通过一个Selector去轮询多个网络连接的事件

ServerSocketChannel -> 端口号，然后通过一个独立的线程（Acceptor）去使用Selector去轮询这个ServerSocketChannel是否有连接的事件接入，如果有的话，跟客户端建立连接，然后将客户端的连接分发给Processor线程

每个Processor线程应该是负责处理一部分客户端的连接，使用自己的Selector不断的轮询各个客户端连接的网络事件，收到了请求解析出来

请求应该是交给Handler线程去进行处理，处理请求，设置响应

Processor把响应发送给对应的客户端就可以了

Broker这块的代码，写的过于复杂了，而且设计的不太好，scala来写服务端代码，没看出来任何优势，玩儿玩儿scala的语法糖，client -> Producer + 网络通信，代码写的是相当的棒，非常的有章法

Netty有点恶心，核心架构、核心机制、底层设计，还是不错的，高并发、高可靠、高性能，做的是不错的，代码写的一点都不好

# **File Path: /Users/xiaotingting/Downloads/学习资料/12\_Netty核心功能精讲以及核心源码剖析/019~030资料/021\_Netty是如何对ServerSocketChannel进行初始化的？/笔记.doc**

netty官方已经废弃掉了最新的5.x版本，一直在更新的还是4.x版本

event loop，轮询网络事件，线程

这里面的东西肯定是负责通过一个Selector去轮询多个网络连接的事件

ServerSocketChannel -> 端口号，然后通过一个独立的线程（Acceptor）去使用Selector去轮询这个ServerSocketChannel是否有连接的事件接入，如果有的话，跟客户端建立连接，然后将客户端的连接分发给Processor线程

每个Processor线程应该是负责处理一部分客户端的连接，使用自己的Selector不断的轮询各个客户端连接的网络事件，收到了请求解析出来

请求应该是交给Handler线程去进行处理，处理请求，设置响应

Processor把响应发送给对应的客户端就可以了

# **File Path: /Users/xiaotingting/Downloads/学习资料/12\_Netty核心功能精讲以及核心源码剖析/019~030资料/019\_将Netty版本调整回4.x，避免使用废弃的5.x版本/笔记.doc**

netty官方已经废弃掉了最新的5.x版本，一直在更新的还是4.x版本

event loop，轮询网络事件，线程

这里面的东西肯定是负责通过一个Selector去轮询多个网络连接的事件

ServerSocketChannel -> 端口号，然后通过一个独立的线程（Acceptor）去使用Selector去轮询这个ServerSocketChannel是否有连接的事件接入，如果有的话，跟客户端建立连接，然后将客户端的连接分发给Processor线程

每个Processor线程应该是负责处理一部分客户端的连接，使用自己的Selector不断的轮询各个客户端连接的网络事件，收到了请求解析出来

请求应该是交给Handler线程去进行处理，处理请求，设置响应

Processor把响应发送给对应的客户端就可以了

# **File Path: /Users/xiaotingting/Downloads/学习资料/12\_Netty核心功能精讲以及核心源码剖析/019~030资料/028\_对于客户端发送过来的消息是如何读取以及处理的？/笔记.doc**

# **File Path: /Users/xiaotingting/Downloads/学习资料/12\_Netty核心功能精讲以及核心源码剖析/019~030资料/026\_如果发现了客户端发起的连接事件是如何进行处理的？/笔记.doc**

用一个字来形容：烂

netty优点，很多地方人家确实都考虑到了，功能、性能、并发，都做的不错

掩盖代码的烂

# **File Path: /Users/xiaotingting/Downloads/学习资料/12\_Netty核心功能精讲以及核心源码剖析/031~044资料/040\_再看Netty Server端读取请求的网络IO核心源码流程/笔记.doc**

# **File Path: /Users/xiaotingting/Downloads/学习资料/12\_Netty核心功能精讲以及核心源码剖析/031~044资料/037\_线程模型的进化：单个Acceptor线程 + 多个Processor线程/笔记.doc**

# **File Path: /Users/xiaotingting/Downloads/学习资料/12\_Netty核心功能精讲以及核心源码剖析/031~044资料/039\_Netty中的两个线程池就足够了吗？不，还不够！/笔记.doc**

# **File Path: /Users/xiaotingting/Downloads/学习资料/12\_Netty核心功能精讲以及核心源码剖析/031~044资料/031\_Netty Client是如何尝试跟Server建立连接的？/笔记.doc**

netty的源码，远远不如kafka网络通信这块的源码，远远不如我们写的分布式海量小文件存储系统中的自己封装的nio网络通信框架的源码

代码写的很差，可读性很差

# **File Path: /Users/xiaotingting/Downloads/学习资料/12\_Netty核心功能精讲以及核心源码剖析/031~044资料/033\_Netty Client的线程是如何轮询建立好连接的网络事件的？/笔记.doc**

# **File Path: /Users/xiaotingting/Downloads/学习资料/12\_Netty核心功能精讲以及核心源码剖析/031~044资料/044\_Netty为请求处理提供的良好扩展：自定义业务逻辑链条/笔记.doc**

可以加入自己的业务处理链条，一个请求过来，同步处理，同步返回响应；异步处理，提交请求数据到一个异步线程池，线程池是你自己做的，异步读写磁盘，干一些网络通信的事情，直接返回响应即可

同步+异步一起干，有些东西是需要同步的，有些东西可以是异步的

# **File Path: /Users/xiaotingting/Downloads/学习资料/12\_Netty核心功能精讲以及核心源码剖析/031~044资料/032\_具体看看Netty Client对Server发起connect请求的源码/笔记.doc**

# **File Path: /Users/xiaotingting/Downloads/学习资料/12\_Netty核心功能精讲以及核心源码剖析/031~044资料/035\_从Netty Server端返回的响应是如何接收的？/笔记.doc**

# **File Path: /Users/xiaotingting/Downloads/学习资料/12\_Netty核心功能精讲以及核心源码剖析/031~044资料/034\_Netty Client是如何将我们准备好的请求发送出去的？/笔记.doc**

应该说这个方法，是在我们建立好连接之前就调用了，所以建立好连接之后，其实打了断点，也没走到这个地方来

# **File Path: /Users/xiaotingting/Downloads/学习资料/12\_Netty核心功能精讲以及核心源码剖析/031~044资料/041\_仔细阅读Unsafe内部网络IO的数据缓冲机制源码/笔记.doc**

说一下他的这个组件的RecvByteBufAllocator，动态的根据你上一次请求获取到的数据大小，动态的预估这次请求的数据大小大致是会有多少，根据预估的结果创建出来一个比较符合预估大小的一个缓冲区出来

他应该是负责去分配ByteBuf数据缓冲区的一个组件

他每次到底是分配多大的一个ByteBuf数据缓冲区呢？分布式海量小文件存储系统的时候，自定义了一套协议，kafka，每次请求过来，请求头都必修带着本次请求数据的大小，我们是根据请求头来分配ByteBuffer的

netty而言，他不能指望每次请求都有一个请求头，通用框架，根据每次请求的大小动态的预估下一次请求的大小，动态根据预估的大小创建对应的ByteBuf

# **File Path: /Users/xiaotingting/Downloads/学习资料/12\_Netty核心功能精讲以及核心源码剖析/031~044资料/042\_细看一次网络请求数据到底是如何读取出来的？/笔记.doc**

根据预估分配一个ByteBuf

根据预估分配的ByteBuf的大小创建一个原生的NIO ByteBuffer，从原生SocketChannel中读取数据放入原生ByteBuffer里，再把数据放入Netty ByteBuf里，完成一次请求数据的读取，就搞定了

# **File Path: /Users/xiaotingting/Downloads/学习资料/12\_Netty核心功能精讲以及核心源码剖析/031~044资料/038\_支撑百万连接的线程模型：Acceptor线程池/笔记.doc**

现在的连接都是最最简单的TCP三次握手的这种连接而已

很多生产环境的连接会更加的复杂，Acceptor线程跟每个客户端发起的连接请求不光是TCP三次握手，还要基于授权认证的协议跟连接进行复杂的权限确认，确保说这个客户单是有权限接入进来的

百万连接，单个高配置物理机，32核128G的高配机器，单机支撑百万客户端连接的话，单个Acceptor线程肯定是来不及处理这么多的连接请求的

Acceptor线程进行池化，由很多个Acceptor角色的线程负责处理跟客户端连接的请求，每个Acceptor线程可以负责跟一批客户端建立连接，建立好的连接再转交给 Processor线程池，100个Acceptor线程

百万 连接，每个Acceptor线程负责处理1万个客户端的连接，那这样的话，压力就会小很多

# **File Path: /Users/xiaotingting/Downloads/学习资料/12\_Netty核心功能精讲以及核心源码剖析/031~044资料/043\_原生Java NIO中的ByteBuffer使用时有哪些缺陷？/笔记.doc**

1、原生ByteBuffer是固定长度的，一旦创建无法动态的调整大小，创建大了，会导致内存浪费；创建小了，就不过放的

2、ByteBuffer非常的不好用，因为他里面需要read，channel读取数据放入ByteBuffer中，此时要使用这个ByteBuffer里的数据，还得记得调用flip，重置position，然后才可以从ByteBuffer里读取数据

Netty为什么要封装自己的ByteBuf呢？第一，他允许ByteBuf大小动态调整，第二，他封装的读写API更加的简单，不需要去care flip

# **File Path: /Users/xiaotingting/Downloads/学习资料/12\_Netty核心功能精讲以及核心源码剖析/031~044资料/036\_再看Netty线程模型：如果Server端单线程会有什么问题？/笔记.doc**

Server端纯用单线程来处理，可以不可以？

但是性能实在是太差了，单线程，就算是把CPU给跑满，也难以应对大量的客户端高并发的连接和请求，CPU打满，大量的客户端发送过来的请求来不及处理，直接进行超时处理，还需要进行请求重试

很有可能会导致大量的连接频繁的发送请求超时，最后连接断开，非常的不稳定

单线程是来不及处理的，会导致性能很差，连接非常的不稳定

# **File Path: /Users/xiaotingting/Downloads/学习资料/12\_Netty核心功能精讲以及核心源码剖析/045~062资料/045~062资料/046\_Netty底层基于NIO的非阻塞模型为什么可以支撑高并发/笔记.doc**

高并发

大量的客户端，成千上万，十万级，百万级；大量的请求过来，每秒几千几万，每秒十万

# **File Path: /Users/xiaotingting/Downloads/学习资料/12\_Netty核心功能精讲以及核心源码剖析/045~062资料/045~062资料/057\_链路不可用的自动监测：Netty的空闲检测机制/笔记.doc**

# **File Path: /Users/xiaotingting/Downloads/学习资料/12\_Netty核心功能精讲以及核心源码剖析/045~062资料/045~062资料/052\_网络通信中的序列化环节应该如何优化性能？/笔记.doc**

比如说你有一个自定义的对象，序列化成二进制字节流，把字节流发送到服务端；服务端反序列化，二进制字节流还原成一个对象，处理，再次把响应序列化成二进制字节流，发送到客户端去，客户端再次反序列化成响应对象

序列化，就是把你的数据搞成二进制字节流，或者二进制数组；反序列化，就是一个反过程

Java序列化机制，会把序列化以后的字节流搞的太大了，而且序列化的过程也较慢，所以不是太好，Netty默认支持Protobuf，序列化里性能几乎是最好的，而且序列化以后的字节流也不大

所以序列化这块性能更好

Protobuf，序列化成二进制字节流，小，速度快，反序列化速度也会比较快

# **File Path: /Users/xiaotingting/Downloads/学习资料/12\_Netty核心功能精讲以及核心源码剖析/045~062资料/045~062资料/062\_作业：对于提到的架构设计的细节自己在源码里找出来/笔记.doc**

光看视频，只能达到50%的效果，认认真真完成我布置的作业，有问题可以来问我，达到剩余50%的效果，100%的效果

高并发架构设计：两层线程模型、NIO多路复用非阻塞、无锁串行化、并发优化

高性能架构设计：Protobuf序列化协议、direct buffer、bytebuf内存池、网络参数优化

高可靠架构设计：网络连接断开、网络链路探查、NioEventLoop线程容错、JDK epoll bug处理、内存块自动释放

可扩展架构设计：handler链条你可以自己扩展、序列化协议、定制网络通信协议

# **File Path: /Users/xiaotingting/Downloads/学习资料/12\_Netty核心功能精讲以及核心源码剖析/045~062资料/045~062资料/048\_JDK 1.5之后对NIO优化为采用epoll模式为何提高了性能/笔记.doc**

select/poll模式下，如果线程被唤醒通知有某个文件是就绪状态，此时线程需要重新遍历所有的文件去看到底是谁就绪了

Selector注册了100个客户端，其中只有一个客户端有网络事件，这个时候Selector必须重新遍历一遍100个客户端，收集出来一个客户端的网络事件交给你的线程来进行处理，性能有问题

epoll不一样，如果有某个文件有就绪状态，直接回调epoll回调函数，把就绪的文件放入epoll的一个数据结果中，然后epoll直接就知道哪些文件是就绪的，不需要有一个重新遍历的过程，所以效率更高

# **File Path: /Users/xiaotingting/Downloads/学习资料/12\_Netty核心功能精讲以及核心源码剖析/045~062资料/045~062资料/051\_Netty是如何尽一切可能去优化内部的多线程并发的/笔记.doc**

高并发

对Netty框架内部而言，主要的性能优化就是在于并发的优化，尽量避免使用synchronized等重量级锁，而是采用volatile、CAS、并发安全集合、读写锁，来尽最大可能优化多线程并发的锁争用问题

NioEventLoop，内部就对很多成员变量都用了volatile，尽量对一个线程写，其他线程读的场景，采用volatile保证可见性即可，避免使用重量级锁

ChannelOutputBuffer中，很多统计数据，都是采用Atomic类实现的，多线程并发读写，都没问题

NioEventLoop，newTaskQueue，ConcurrentLinkedQueue，之前给大家分析过这些源码，其实他们底层大量基于volatile、CAS，降低锁粒度，各种方式来写的，优化了并发性能

HashedWheelTimer，里面用了读写锁来优化

大量的高级并发技术的运用，大幅度优化了Netty内部并发锁冲突的性能问题，大家以后注意，在并发编程的时候尽量要提升性能

作业：希望大家可以把netty的源码导入intellij idea中去，在那个里面可以去搜索搜索，netty中框架内部，为了优化多线程并发能力，减少锁争用，volatile、CAS、并发集合类、读写锁，在哪里使用了，读一读细节

# **File Path: /Users/xiaotingting/Downloads/学习资料/12\_Netty核心功能精讲以及核心源码剖析/045~062资料/045~062资料/059\_NIO epoll空轮询bug：Netty自动识别与处理/笔记.doc**

NIO底层epoll，JDK版本的bug，轮询遍历所有的网络连接，epoll空轮询，明明没有任何的事件，他不停的疯狂的轮询底层的所有的网络连接，导致CPU 100%，NIO，Netty在底层直接解决掉了

直接把注册在这个Selector上面的SocketChannel注册到新创建的Selector上面去，把当前这个有问题的Selector给关闭，释放资源，避免他空轮询导致cpu 100%

# **File Path: /Users/xiaotingting/Downloads/学习资料/12\_Netty核心功能精讲以及核心源码剖析/045~062资料/045~062资料/060\_自动对缓冲池中的缓冲块进行释放避免内存泄露/笔记.doc**

# **File Path: /Users/xiaotingting/Downloads/学习资料/12\_Netty核心功能精讲以及核心源码剖析/045~062资料/045~062资料/053\_通过Direct Buffer如何大幅度提升网络读写性能？/笔记.doc**

ByteBuffer，有一种特殊的Buffer，Direct模式的Buffer，创建ByteBuffer的时候可以指定是创建Direct模式的Buffer

不用Direct模式，性能比较差

ByteBuffer，direct模式来创建的

# **File Path: /Users/xiaotingting/Downloads/学习资料/12\_Netty核心功能精讲以及核心源码剖析/045~062资料/045~062资料/045\_先回顾一下传统BIO网络通信为什么性能那么差/笔记.doc**

Netty架构设计，做一个总结和收尾，高并发、高性能、高可靠

BIO，一个请求一个线程，高并发下线程资源不够，机器cpu负载100%，机器容易宕机，或者系统性能急剧下降

你呢基本上来说，是来一个请求用一个线程来处理，或者一个客户端就用一个线程来进行处理，几千个客户端要来发起请求，难道你用几千个线程来抗吗？来一个请求创建一个线程进行处理

线程资源是很宝贵的

# **File Path: /Users/xiaotingting/Downloads/学习资料/12\_Netty核心功能精讲以及核心源码剖析/045~062资料/045~062资料/049\_如何优化Netty的线程模型支撑百万连接接入？/笔记.doc**

一般采用多个线程监听ServerSocketChannel的连接请求，然后多个线程负责IO读写，这是常见的模式，在高配置物理机上，甚至可以单机支撑百万连接，比如64核128G的高配物理机，Acceptor线程开启100个，负责百万客户端的接入，然后IO线程开启1000个甚至更多，每个IO线程仅仅负责1000左右的客户端读写请求

这就可以单机实现百万连接了

# **File Path: /Users/xiaotingting/Downloads/学习资料/12\_Netty核心功能精讲以及核心源码剖析/045~062资料/045~062资料/058\_在NioEventLoop线程中处理IO异常避免线程中断/笔记.doc**

# **File Path: /Users/xiaotingting/Downloads/学习资料/12\_Netty核心功能精讲以及核心源码剖析/045~062资料/045~062资料/061\_Netty架构设计总结：高并发、高性能、高可靠、易于扩展/笔记.doc**

高并发架构设计：两层线程模型、NIO多路复用非阻塞、无锁串行化、并发优化

高性能架构设计：Protobuf序列化协议、direct buffer、bytebuf内存池、网络参数优化

高可靠架构设计：网络连接断开、网络链路探查、NioEventLoop线程容错、JDK epoll bug处理、内存块自动释放

可扩展架构设计：handler链条你可以自己扩展、序列化协议、定制网络通信协议

# **File Path: /Users/xiaotingting/Downloads/学习资料/12\_Netty核心功能精讲以及核心源码剖析/045~062资料/045~062资料/056\_Netty的高可靠性：自动识别连接断开和释放资源/笔记.doc**

CONNECT\_TIMEOUT\_MILLIS，可以设置连接超时时间

# **File Path: /Users/xiaotingting/Downloads/学习资料/12\_Netty核心功能精讲以及核心源码剖析/045~062资料/045~062资料/055\_对核心的网络参数如何优化可以保证高性能？/笔记.doc**

SO\_RCVBUF和SO\_SNDBUF，128kb或者256kb

SO\_TCPNODELAY，关闭这个算法，避免自动打包发送，避免高延时

# **File Path: /Users/xiaotingting/Downloads/学习资料/12\_Netty核心功能精讲以及核心源码剖析/045~062资料/045~062资料/047\_JDK 1.4采用select、poll模式实现的NIO是什么原理/笔记.doc**

线程遍历所有的文件（linux里一切皆文件，比如说网络连接也可以理解为文件），如果某个网络连接没有就绪（没有网络事件发生），就对那个网络连接插入一个wait\_queue节点，然后继续遍历别的文件

如果有某个文件有就绪状态（比如有网络数据包到达），就把这些文件的就绪状态复制给用户进程，如果没有一个文件是就绪状态

那么就阻塞等待唤醒，如果某个网络连接有事件（比如网络数据包到达），就遍历自己的wait\_queue等待队列，然后回调函数，唤醒在阻塞等待的线程

线程被唤醒之后，再次遍历所有文件的就绪状态，如果有就绪就返回给用户进程

实现单个线程的多路复用，select/poll模式几乎是一致的

# **File Path: /Users/xiaotingting/Downloads/学习资料/12\_Netty核心功能精讲以及核心源码剖析/045~062资料/045~062资料/050\_Netty的无锁串行化设计思想是如何保证高性能的/笔记.doc**

一个IO线程负责监听多个客户端的请求，并且处理请求，然后发送响应

这就是所谓的无锁串行化设计思想，假如说一个IO线程接收了多个请求分发给多个不同的Worker线程去处理，其实就是之前我们做的那种思路，可能会导致线程与线程之间进行共享资源的争抢

一旦发生了锁的问题，那么会导致并发能力更差

所以Netty默认就是一个IO线程无锁串行化处理各个客户端的请求，处理完一个返回了响应，再处理下一个请求

如果我们要是认为再加一层Handler线程池，争抢处理请求，而且不会导致资源争用，那么其实自己再加一层线程池也没什么

高并发架构设计里去，你自己开启很多线程，这些线程处理共享资源的时候频繁加锁，争用锁，导致性能的问题

# **File Path: /Users/xiaotingting/Downloads/学习资料/12\_Netty核心功能精讲以及核心源码剖析/045~062资料/045~062资料/054\_如何通过内存池实现ByteBuf分配与回收的性能优化/笔记.doc**

内存块的申请和销毁都是比较耗时的，对性能也有影响

内存池，ByteBuf池子的概念，就可以在池子里维护一定的ByteBuf，需要的时候直接从池子里获取ByteBuf使用即可，使用完毕之后就重新放回到池子里去

# **File Path: /Users/xiaotingting/Downloads/学习资料/12\_Netty核心功能精讲以及核心源码剖析/17 数据存储技术选型/001~010资料/008\_面向高并发场景的NoSQL数据库：KV存储（3）/笔记.docx**

《08\_面向高并发场景的NoSQL数据库：KV存储（3）》

互联网公司里，几乎是绕不过去一个话题，NoSQL数据存储中的一种，分布式KV存储

核心业务系统，数据库 -> 分库分表 -> 分布式数据库，如果有搜索场景，用ES作为一种数据存储的补充技术，互联网遇到这样的一些场景，核心业务系统，他们也不需要事务，也不需要复杂的库表数据模型，也不需要执行复杂的SQL语句

数据模型很简单的，key-value，kv数据格式，简单的kv读写的操作，互联网公司核心业务场景里会遇到这类场景，读写并发很高，比较高并发的写和比较高并发的读，当同时满足这些业务场景和情况的时候，一般来说，就可以引入NoSQL数据库里的分布式kv存储

社交系统，数据模型比较简单，不需要大量的表和复杂的字段，简单的读写操作，并发量很高，直接面向C端用户的，社交场景的系统，不适合使用MySQL / TiDB这种关系型的数据库来支持的

数据模型简单（key-value） + 读写操作（kv读写）+ 读写并发很高，挂很多的从库，分库分表，TiDB，浪费！高并发的支持就不是特别好，8核16GB的机器，2000 QPS，2万QPS，此时你需要10台机器来支撑

NoSQL数据库

Redis（开启持久化，保证数据可能会丢失一点点，但是不会丢失太多），kv，kv读写，数据结构和基于数据结构的复杂操作，读写并发能力极为强悍，主要是基于内存来实现读写操作，轻松可以支持几万QPS，甚至10万QPS

数据存储主要是基于内存，虽然会基于磁盘做持久化，避免数据丢失，全量数据都是放在内存里的，他不适合海量数据的场景，kv存储，热点数据缓存

中大型互联网公司，自研分布式kv存储，淘宝开源的Tair，支持存储海量数据，读写自动内存热点数据+磁盘存储海量全量数据，性能很高，QPS也是几万甚至十万，都会自己研发类似的分布式kv存储

数据库 + 分库分表 / TiDB

Elasticsearch

分布式KV存储：自研，HBase

HBase，大数据架构那边的课程，会对他的原理有一个很深入的理解

面向列的存储，核心就是kv存储，半结构化，

不停的灌入大量的简单的kv模型的数据，分布式存储，扩容、运维都很方便，写入kv数据，对他来说，并发能力是超高的，不支持事务，直接写入内存buffer缓冲就可以了，WAL预写日志

只要你是根据rowkey去进行读取，有一定的概率会命中内存里的缓存，直接从内存里加载数据出来，布隆过滤器、LSM索引树，大幅度的提升和优化了rowkey的读取，性能和并发也是不错的

海量数据，模型简单，操作简单，读写并发能力和性能也还是不错的

HBase的NoSQL数据库也好，分布式KV存储也好，其实可以支持模型简单、操作简单、数据量很大、并发很高，HBase来支撑类似的场景也是比较多的，在电商卖家日志系统里，去做一个项目实战

3616118202

# **File Path: /Users/xiaotingting/Downloads/学习资料/12\_Netty核心功能精讲以及核心源码剖析/17 数据存储技术选型/001~010资料/005\_分布式搜索引擎：关系型数据库的一种补充/笔记.docx**

《05\_分布式搜索引擎：关系型数据库的一种补充》

普通数据库架构 -> 读写分离 + 分库分表 + 数据库中间件（MyCat / Sharding-Sphere）- > 最近这一年，出现了一个趋势，涌现出了一个非常成功和比较好的分布式数据库的技术，TiDB

TiDB，分布式关系型数据库，SQL语法支持、事务支持、各种功能支持，都是要有的，适合的也是那种核心业务系统，商家系统，商品系统，订单系统

要做一些搜索类的业务，要做一些简单的语义分析和基于语义的搜索

1 我喜欢吃鸡腿

2 我除了鸡腿，还是吃拉面

3 鸡腿蛋炒饭的一个做法

select \* from xxx where id=1

搜索“鸡腿”，只要跟鸡腿有关的文本，都需要给我查询出来

MySQL，关系型数据库，也可以通过建立全文索引来支持类似这种基于文本和语义的搜索，他们对这块的支持是非常的不好的

Elasticseach，就是用来做搜索的，对你的各种文本建立倒排索引，然后可以基于文本和语义去进行搜索，在数据存储架构里，不是必须要有的，一般来说是你出现搜索业务场景的时候，会作为补充引入ES技术

3616118202

# **File Path: /Users/xiaotingting/Downloads/学习资料/12\_Netty核心功能精讲以及核心源码剖析/17 数据存储技术选型/001~010资料/001\_学习海量数据架构设计实战之前，先夯实理论基础/笔记.docx**

3616118202

《01\_学习海量数据架构设计实战之前，先夯实理论基础》

官网公布的2020年新版本大纲里，在业务架构之后，海量数据架构，符合演进的过程，大部分的系统，刚开始主要都是做业务，DDD业务建模，软件工程理论和流程，项目管理的流程，推动一个业务系统的开发，基础框架+分布式架构+微服务架构+持续集成/持续交付，数据库的表设计，基本上一个系统已经可以出来了

高并发、高性能、高可用、稳定性的问题

国内就只有少数公司是真的有高并发的场景和问题的，大多数公司的并发都是很低的，哪怕是一些互联网公司，并发也没那么高；高性能，系统的数据量就很少，就压根儿就没有性能问题；比较简单的话，也谈不上高可用；稳定性

海量数据问题，数据量可能会比较大一些，海量数据存储架构如何设计，技术选型

MySQL、分库分表、数据库中间件、TiDB、MongoDB、HBase、Elasticsearch、Redis，各自的特点是什么，各自的优劣势是什么，分别适合在哪些场景下使用，了解清楚了，再针对不同的业务场景，学习不同场景下的数据存储架构设计实战，结合具体的项目来做

不同的数据存储技术，考虑哪些因素：

运维成本：监控告警、备份恢复、升级和迁移、社区活跃度、性能优化、troubleshooting

稳定性：数据多副本、高可用、多写多活

性能：延迟、QPS，等等

扩展性：横向扩容和纵向扩容，等等

安全：权限，认证，审计，等等

# **File Path: /Users/xiaotingting/Downloads/学习资料/12\_Netty核心功能精讲以及核心源码剖析/17 数据存储技术选型/001~010资料/006\_面向高并发场景的NoSQL数据库：KV存储（1）/笔记.docx**

《06\_面向高并发场景的NoSQL数据库：KV存储》

互联网公司里，几乎是绕不过去一个话题，NoSQL数据存储中的一种，分布式KV存储

核心业务系统，数据库 -> 分库分表 -> 分布式数据库，如果有搜索场景，用ES作为一种数据存储的补充技术，互联网遇到这样的一些场景，核心业务系统，他们也不需要事务，也不需要复杂的库表数据模型，也不需要执行复杂的SQL语句

数据模型很简单的，key-value，kv数据格式，简单的kv读写的操作，互联网公司核心业务场景里会遇到这类场景，读写并发很高，比较高并发的写和比较高并发的读，当同时满足这些业务场景和情况的时候，一般来说，就可以引入NoSQL数据库里的分布式kv存储

数据模型简单（key-value） + 读写操作（kv读写）+ 读写并发很高，挂很多的从库，分库分表，TiDB，浪费！高并发的支持就不是特别好，8核16GB的机器，2000 QPS，2万QPS，此时你需要10台机器来支撑

3616118202

# **File Path: /Users/xiaotingting/Downloads/学习资料/12\_Netty核心功能精讲以及核心源码剖析/17 数据存储技术选型/001~010资料/009\_MongoDB这种NoSQL数据存储一般适合什么场景？/笔记.docx**

3616118202

《09\_MongoDB这种NoSQL数据存储一般适合什么场景？》

数据库 + 分库分表 / TiDB，适合的场景是核心业务系统

Elasticsearch，适合的场景是搜索

分布式KV存储：自研，HBase，简单的KV模型+简单的KV读写+海量数据+高并发，有一些特殊的场景，电商卖家日志系统

NoSQL：MongoDB，适合的场景类似于社交系统，电商买家社交系统

不太好说把MongoDB划分到分布式KV存储里去，数据模型有点类似于ES，是面向文档（Document）的数据模型，他支持的数据模型相对来说会复杂一些，也可以建立索引，数据存储经过了大量的优化，稍微有一点点复杂度的数据模型，支持稍微复杂一点的功能，事务，海量数据+高并发，分布式架构

# **File Path: /Users/xiaotingting/Downloads/学习资料/12\_Netty核心功能精讲以及核心源码剖析/17 数据存储技术选型/001~010资料/004\_分库分表架构的未来演进趋势：分布式数据库/笔记.docx**

3616118202

《04\_分库分表架构的未来演进趋势：分布式数据库》

数据库普通开发架构 -> 读写分离 + 分库分表 + 数据库中间件（MyCat / Sharding-Sphere）- > 最近这一年，出现了一个趋势，涌现出了一个非常成功和比较好的分布式数据库的技术，TiDB

分布式数据库，彻底替代比较复杂的、有很多痛点的数据库架构：读写分离 + 分库分表 + 数据库中间件（MyCat / Sharding-Sphere）

后续做海量数据存储架构实战，分库分表那块，我们会用真实的复杂订单系统来做实战，包括如何设计方案，如何做数据迁移，如何无缝接入系统，如何基于中间件支持全部订单功能，如何基于ES做订单搜索，这是一个完整的实战，但是大家也必须意识到，这种分库分表架构终将成为一个过渡方案，因为无论如何，这个SQL语法支持是不完善的，同时运维起来是极为麻烦的，所以会进入分布式数据库的阶段

TiDB，分布式数据库，架构上就是完全基于多台服务器形成的一个分布式的架构，灌入数据，内部自动把数据均匀分散在多台服务器上面，本质上就跟分库分表是一个概念，运维，他内部给你透明的完成了，不需要你自己去关注这个东西，扩容，加几台服务器给TiDB集群，内部可以自动的数据迁移和扩容，分布式事务，支持的也会更好，复杂SQL语法的支持，也会更好

所以未来的一个趋势就是：分布式数据库，以TiDB为首，完全分布式，扩容、运维，都极为的方便，支持更加完善的复杂的SQL语法，同样包含事务支持，让研发人员使用起来，不用去care所谓的分多少库和表，扩容的时候也不用care这些细节

过去一年多里，很多互联网公司都开始在引入TiDB这个技术，部分开始替代原来的分库分表的架构，坑，踩坑，解决他自己的一些问题，还是比自己去玩儿原始的分库分表要好多了，TiDB会成为替代分库分表的主流

接着我们会给大家讲解，如何用分布式数据库去重构订单系统数据存储架构

# **File Path: /Users/xiaotingting/Downloads/学习资料/12\_Netty核心功能精讲以及核心源码剖析/17 数据存储技术选型/001~010资料/003\_基于MySQL的分库分表架构：痛并快乐着！/笔记.docx**

3616118202

《03\_基于MySQL的分库分表架构：痛并快乐着！》

备份恢复+高可用，都是mysql比较常规的功能和架构，网上一搜，多的是

一般中小型公司，如果没有百亿级数据、每秒上万TPS写入压力的话，其实就做一个MySQL数据库的备份恢复方案+高可用自动故障转移方案，就足够了，但是如果是有点规模的互联网公司，数据量几十亿到百亿级，写入TPS轻松破万，那么互联网公司玩读写分离+分库分表基本就是个标配

读写分离，主库 + 几个从库，主库 -> 从库会同步数据，半同步，数据会有一定的延迟，几十ms，写入就往主库里写，查询可以从主库或者是其他的从库里查，多挂几个从库，就可以分摊你的查询QPS

数据量很大，上百亿数据量，一台数据库服务器磁盘空间都放不下，磁盘都快满了；写入TPS压力极大，每秒钟写入压力要达到上万，分库分表

其实分库分表往简单了说也并不难理解，就是搞好几台数据库服务器，创建一堆库，每个表搞一堆表，分散在各个数据库服务器上，然后这样一个表的数据就分散在多台服务器上了，每秒上万的写入TPS就分散在多台服务器上了

写入和查询，如果是简单的那条根据id主键的单条数据，很简单，做一个路由，路由到某个库某个表，写入到里面，以后也从这个库这个表里查就行了

但是，真的这么简单吗？万一你要搞一些很复杂很恶心的查询呢？跨表？跨库？这怎么办，要不然基于mycat、sharding-sphere之类的中间件执行，他们支持的复杂SQL语法还有限制，而且会基于内存来做，效果也不是太好，要不然就是用ES之类的额外数据存储，全量数据打进去，然后重建数据模型来做复杂查询

另外，运维成本呢？相当高，修改表字段，建表之类的，一搞就是多个库多个表，弄的很麻烦不是吗？还有最恶心的就是你扩容的时候，怎么扩？做数据迁移？那是相当的麻烦，所以是痛并快乐着

分布式事务，SQL，跨多个数据库和多个表去执行的，事务如何保证？

# **File Path: /Users/xiaotingting/Downloads/学习资料/12\_Netty核心功能精讲以及核心源码剖析/17 数据存储技术选型/001~010资料/007\_面向高并发场景的NoSQL数据库：KV存储（2）/笔记.docx**

《07\_面向高并发场景的NoSQL数据库：KV存储（2）》

互联网公司里，几乎是绕不过去一个话题，NoSQL数据存储中的一种，分布式KV存储

核心业务系统，数据库 -> 分库分表 -> 分布式数据库，如果有搜索场景，用ES作为一种数据存储的补充技术，互联网遇到这样的一些场景，核心业务系统，他们也不需要事务，也不需要复杂的库表数据模型，也不需要执行复杂的SQL语句

数据模型很简单的，key-value，kv数据格式，简单的kv读写的操作，互联网公司核心业务场景里会遇到这类场景，读写并发很高，比较高并发的写和比较高并发的读，当同时满足这些业务场景和情况的时候，一般来说，就可以引入NoSQL数据库里的分布式kv存储

社交系统，数据模型比较简单，不需要大量的表和复杂的字段，简单的读写操作，并发量很高，直接面向C端用户的，社交场景的系统，不适合使用MySQL / TiDB这种关系型的数据库来支持的

数据模型简单（key-value） + 读写操作（kv读写）+ 读写并发很高，挂很多的从库，分库分表，TiDB，浪费！高并发的支持就不是特别好，8核16GB的机器，2000 QPS，2万QPS，此时你需要10台机器来支撑

NoSQL数据库

Redis（开启持久化，保证数据可能会丢失一点点，但是不会丢失太多），kv，kv读写，数据结构和基于数据结构的复杂操作，读写并发能力极为强悍，主要是基于内存来实现读写操作，轻松可以支持几万QPS，甚至10万QPS

数据存储主要是基于内存，虽然会基于磁盘做持久化，避免数据丢失，全量数据都是放在内存里的，他不适合海量数据的场景，kv存储，热点数据缓存

中大型互联网公司，自研分布式kv存储，淘宝开源的Tair，支持存储海量数据，读写自动内存热点数据+磁盘存储海量全量数据，性能很高，QPS也是几万甚至十万，都会自己研发类似的分布式kv存储

3616118202

# **File Path: /Users/xiaotingting/Downloads/学习资料/12\_Netty核心功能精讲以及核心源码剖析/17 数据存储技术选型/001~010资料/002\_国内互联网公司标配：MySQL数据库/笔记.docx**

《02\_国内互联网公司标配：MySQL数据库》

国内互联网公司标配，传统软件公司，Oracle、SQL Server + .NET

MySQL的特点是，完善的关系型数据库的模型，支持完善的SQL语法，而且有事务支持，性能还不错，而且数据库必备的一些功能通常也都有，非常适合互联网公司的核心系统的开发，免费，开源

但是缺点也比较明显，因为他支持复杂SQL以及事务，儒猿技术窝，《从0开始带你成为MySQL实战高手》，面向磁盘，用很多内存来读写数据，支持锁，事务也要做很多的事情，所以TPS是比较低的，一般8核16G的机器，大概也就支持两三千的TPS，机器负载就很高了，有点扛不住了

MySQL，天生就不是用来抗高并发的

所以一般在互联网公司里用MySQL，一定会做这么几块架构：

（1）数据备份和恢复，一般是日备份、周备份，会设定几个级别

（2）高可用架构，也就是做master-slave主从架构，数据实时同步，读写分离开来，一定程度上扩展数据库的读QPS，同时可以做故障自动切换，当然也可以采用MHA之类的其他高可用架构

（3）读写分离：就是上面说的，如果读压力比较大，就多挂几个从库扩容读QPS

（4）分库分表：搞多台数据库服务器组成一个分库分表的集群，然后基于mycat、sharding-sphere之类的中间件去支持，这样可以对数据存储容量进行扩容，同时可以扩展数据库写入TPS

至于说业务场景，说实话，一般的系统，只要没特殊要求，刚开始就都用MySQL就行了，如果后续感觉有点瓶颈了，再用别的数据存储就行3616118202

# **File Path: /Users/xiaotingting/Downloads/学习资料/12\_Netty核心功能精讲以及核心源码剖析/17 数据存储技术选型/001~010资料/010\_海量数据场景下的难题：数据迁移和数据同步/笔记.docx**

《10\_海量数据场景下的难题：数据迁移和数据同步》

数据库 + 分库分表 / TiDB，适合的场景是核心业务系统，订单系统

Elasticsearch，适合的场景是搜索，订单系统

分布式KV存储：自研，HBase，简单的KV模型+简单的KV读写+海量数据+高并发，有一些特殊的场景，电商卖家日志系统

NoSQL：MongoDB，适合的场景类似于社交系统，电商买家社交系统

不太好说把MongoDB划分到分布式KV存储里去，数据模型有点类似于ES，是面向文档（Document）的数据模型，他支持的数据模型相对来说会复杂一些，也可以建立索引，数据存储经过了大量的优化，稍微有一点点复杂度的数据模型，支持稍微复杂一点的功能，事务，海量数据+高并发，分布式架构

数据迁移，要从一个数据存储里把一大堆的数据迁移到另外一个数据存储里去，其实里面的技术方案和不同的场景，案例 -> 方案；数据同步，你要做异构数据存储架构，要在多种数据存储之间做同步，案例 -> 方案

SQL、NoSQL、NewSQL

可能会把数据存储技术选型和架构设计，理论知识铺垫，也可能没有

3616118202