 **Netty4**

# 名词解释

## ServerBootstrap

统事件监听器（监听系

## EventLoopGroup

# 线程模型

## 基础

早期的线程模型都是阻塞式

while(true){

socket = accept();

handle(socket)

}

我们注册一个端口号，然后在该端口等待，来一条数据处理一条数据，处理数据的过程中，该scoket会被blocked，其他请求等待。优点是不需要考虑多线程，缺点也比较明显，效率极低。

后面由于计算机硬件发展，资源不是特别吃紧的时候，出现了下面这种模型

while(true){

socket = accept();

new thread(socket);

}

实际上早期的tomcat也是这样实现的，为每一次请求开启一个线程，相对之前的，处理其实也不是很麻烦。但是有另外个问题，就是请求量变大之后，系统要不停的开启线程，关闭线程，同时随着请求增多，资源占用越来越大（线程是很吃资源的）。

对于java世界来说，质的飞跃来自于NIO的诞生。Java自1.4以后，加入了新的IO处理方式

1. Channel

可以理解为一个通道，像流一样，又不是流，可以从里面读取数据，也可以向里面写入数据，但是不管读取和写入，都必须读写到Buffer中。Channel有很多种，我们重点关注的就是和服务器有关的ServerSocketChannel还有我们可以操作的FileChannel。

ServerSocketChannel也可以设置成同步的，不过意义不大

1. Selector

可以理解为选择器，它一直在检测他所管理的通道在干嘛，当有数据传输操作的时候，选择channel操作。实现了单一线程来处理数据的操作，当然，我们可以操作单一channel，也可以操作多个。

要和Selector一起用的话，channel必须使用异步方式，这样才能高效的处理所有数据请求，所以FileChannel是不可以通过selector处理的。

1. SelectionKey

准确的说，这个应该是连接channel的事件，当channel尝试注册到Selector的时候，当channel可以读取数据、写入数据的时候等等

1. SelectionKey.OP\_CONNECT
2. SelectionKey.OP\_ACCEPT
3. SelectionKey.OP\_READ
4. SelectionKey.OP\_WRITE

我们可以这样读取或者写入等等

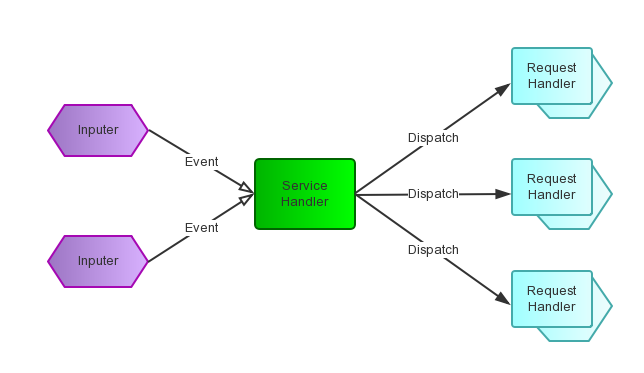
**this**.**selector**.selectedKeys().forEach((SelectionKey key)->{  
 **this**.**selector**.selectedKeys().remove(key);  
 **if**(key.isAcceptable()){  
 ServerSocketChannel channel = (ServerSocketChannel) key.channel();  
 **try** {  
 SocketChannel socketChannel = channel.accept();  
 socketChannel.configureBlocking(**false**);  
 socketChannel.write(ByteBuffer.*wrap*(**new** String(**"Hello Client We Have Connected With Each Other!"**).getBytes(**"UTF-8"**)));  
 socketChannel.register(**selector**,SelectionKey.***OP\_READ***);  
 } **catch** (IOException e) {  
 e.printStackTrace();  
 }  
 }**else if**(key.isReadable()){  
 SocketChannel socketChannel = (SocketChannel) key.channel();  
 ByteBuffer byteBuffer = ByteBuffer.*allocate*(1024);  
 **try** {  
 socketChannel.read(byteBuffer);  
 socketChannel.configureBlocking(**false**);  
 **byte** [] d = byteBuffer.array();  
 String r = **new** String(d);  
 System.***out***.println(**"it's coming to server: "** + r);  
 ByteBuffer out = ByteBuffer.*wrap*(**"Hello Client We Have Received Your Message"**.getBytes());  
 socketChannel.write(out);  
 } **catch** (IOException e) {  
 e.printStackTrace();  
 }  
 }  
});

自java解决了线程问题后，我们可以做一些改动，比如同一个消息请求，整体来讲，我们的网络框架要做的就是这几件事连接🡪读取🡪处理🡪写入。拆分后将该事件分成，事件粒度更小，后面的处理线程，是不影响主线程的，这样提高吞吐量。比如Reactor模式

## Reactor模式

### 什么是Reactor模式

要回答这个问题，首先当然是求助Google或Wikipedia，其中Wikipedia上说：“The reactor design pattern is an event handling pattern for handling service requests delivered concurrently by one or more inputs. The service handler then demultiplexes the incoming requests and dispatches them synchronously to associated request handlers.”。从这个描述中，我们知道Reactor模式首先是事件驱动的，有一个或多个并发输入源，有一个Service Handler，有多个Request Handlers；这个Service Handler会同步的将输入的请求（Event）多路复用的分发给相应的Request Handler。如果用图来表达：

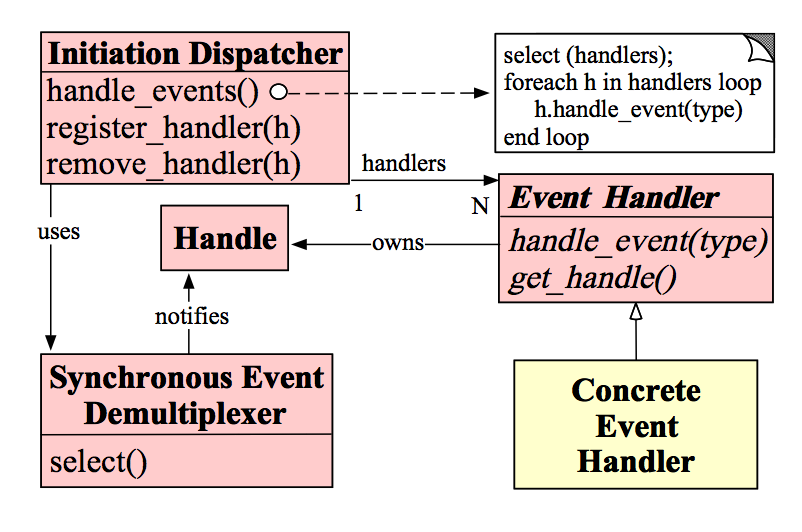


中央控制器，类似于Selector，inputer类似于注册的channel，输入事件触发后，ServiceHanlder处理事件，将请求分发给RequestHanlder。

从结构上，这有点类似生产者消费者模式，即有一个或多个生产者将事件放入一个Queue中，而一个或多个消费者主动的从这个Queue中Poll事件来处理；而Reactor模式则并没有Queue来做缓冲，每当一个Event输入到Service Handler之后，该Service Handler会主动的根据不同的Event类型将其分发给对应的Request Handler来处理。  
  
更学术的，这篇文章（[Reactor An Object Behavioral Pattern for Demultiplexing and Dispatching Handles for Synchronous Events](http://www.dre.vanderbilt.edu/~schmidt/PDF/reactor-siemens.pdf)）上说：“The Reactor design pattern handles service requests that are delivered concurrently to an application by one or more clients. Each service in an application may consistent of several methods and is represented by a separate event handler that is responsible for dispatching service-specific requests. Dispatching of event handlers is performed by an initiation dispatcher, which manages the registered event handlers. Demultiplexing of service requests is performed by a synchronous event demultiplexer. Also known as Dispatcher, Notifier”。这段描述和Wikipedia上的描述类似，有多个输入源，有多个不同的EventHandler（RequestHandler）来处理不同的请求，Initiation Dispatcher用于管理EventHander，EventHandler首先要注册到Initiation Dispatcher中，然后Initiation Dispatcher根据输入的Event分发给注册的EventHandler；然而Initiation Dispatcher并不监听Event的到来，这个工作交给Synchronous Event Demultiplexer来处理。

### Reactor模式结构

在解决了什么是Reactor模式后，我们来看看Reactor模式是由什么模块构成。图是一种比较简洁形象的表现方式，因而先上一张图来表达各个模块的名称和他们之间的关系：



1. **Handle**

即操作系统中的句柄，是对资源在操作系统层面上的一种抽象，它可以是打开的文件、一个连接(Socket)、Timer等。由于Reactor模式一般使用在网络编程中，因而这里一般指Socket Handle，即一个网络连接（Connection，在Java NIO中的Channel）。这个Channel注册到Synchronous Event Demultiplexer中，以监听Handle中发生的事件，对ServerSocketChannnel可以是CONNECT事件，对SocketChannel可以是READ、WRITE、CLOSE事件等。

1. **Synchronous Event Demultiplexer**

阻塞等待一系列的Handle中的事件到来，如果阻塞等待返回，即表示在返回的Handle中可以不阻塞的执行返回的事件类型。这个模块一般使用操作系统的select来实现。在Java NIO中用Selector来封装，当Selector.select()返回时，可以调用Selector的selectedKeys()方法获取Set<SelectionKey>，一个SelectionKey表达一个有事件发生的Channel以及该Channel上的事件类型。上图的“Synchronous Event Demultiplexer ---notifies--> Handle”的流程如果是对的，那内部实现应该是select()方法在事件到来后会先设置Handle的状态，然后返回。不了解内部实现机制，因而保留原图。

1. **Initiation Dispatcher**

用于管理Event Handler，即EventHandler的容器，用以注册、移除EventHandler等；另外，它还作为Reactor模式的入口调用Synchronous Event Demultiplexer的select方法以阻塞等待事件返回，当阻塞等待返回时，根据事件发生的Handle将其分发给对应的Event Handler处理，即回调EventHandler中的handle\_event()方法。

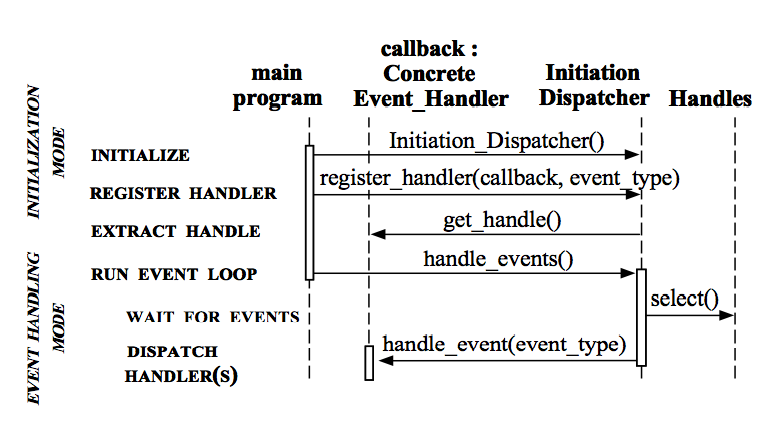
1. **Event Handler**

定义事件处理方法：handle\_event()，以供InitiationDispatcher回调使用。

1. **Concrete Event Handler**

事件EventHandler接口，实现特定事件处理逻辑。

### Reactor模式模块间交互



1. 初始化InitiationDispatcher，并初始化一个Handle到EventHandler的Map。
2. 注册EventHandler到InitiationDispatcher中，每个EventHandler包含对相应Handle的引用，从而建立Handle到EventHandler的映射（Map）。
3. 调用InitiationDispatcher的handle\_events()方法以启动Event Loop。在Event Loop中，调用select()方法（Synchronous Event Demultiplexer）阻塞等待Event发生。
4. 当某个或某些Handle的Event发生后，select()方法返回，InitiationDispatcher根据返回的Handle找到注册的EventHandler，并回调该EventHandler的handle\_events()方法。
5. 在EventHandler的handle\_events()方法中还可以向InitiationDispatcher中注册新的Eventhandler，比如对AcceptorEventHandler来，当有新的client连接时，它会产生新的EventHandler以处理新的连接，并注册到InitiationDispatcher中。

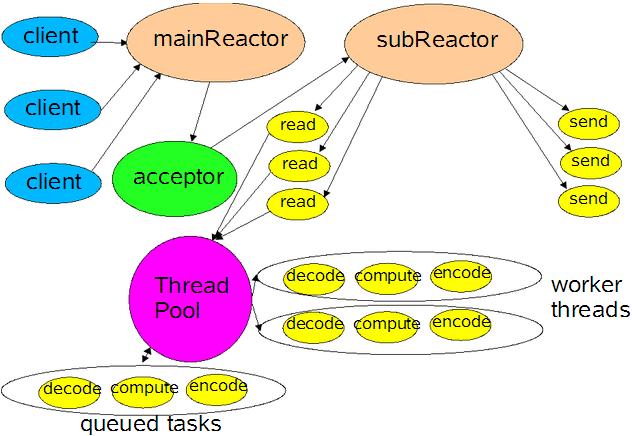
流程大概就是这样，没什么好说的。

### Reactor模式的缺点

Reactor模式的缺点貌似也是显而易见的：

1. 相比传统的简单模型，Reactor增加了一定的复杂性，因而有一定的门槛，并且不易于调试
2. Reactor模式需要底层的Synchronous Event Demultiplexer支持，比如Java中的Selector支持，操作系统的select系统调用支持，如果要自己实现Synchronous Event Demultiplexer可能不会有那么高效。
3. Reactor模式在IO读写数据时还是在同一个线程中实现的，即使使用多个Reactor机制的情况下，那些共享一个Reactor的Channel如果出现一个长时间的数据读写，会影响这个Reactor中其他Channel的相应时间，比如在大文件传输时，IO操作就会影响其他Client的相应时间，因而对这种操作，使用传统的Thread-Per-Connection或许是一个更好的选择，或则此时使用Proactor模式

对于多个CPU的机器，为充分利用系统资源，将Reactor拆分为两部分



这也是为什么我们在使用netty的时候，会有两个group的原因。

# Netty

以下代码摘自官方示例：

EventLoopGroup bossGroup = **new** NioEventLoopGroup(1);  
EventLoopGroup workerGroup = **new** NioEventLoopGroup();  
**try** {  
 ServerBootstrap b = **new** ServerBootstrap();  
 b.group(bossGroup, workerGroup)  
 .channel(NioServerSocketChannel.**class**)  
 .handler(**new** LoggingHandler(LogLevel.***INFO***))  
 .childHandler(**new** ChannelInitializer<SocketChannel>() {  
 @Override  
 **public void** initChannel(SocketChannel ch) {  
 ChannelPipeline p = ch.pipeline();  
 **if** (sslCtx != **null**) {  
 p.addLast(sslCtx.newHandler(ch.alloc()));  
 }  
 p.addLast(**new** DiscardServerHandler());  
 }  
 });  
  
 *// Bind and start to accept incoming connections.* ChannelFuture f = b.bind(***PORT***).sync();  
  
 *// Wait until the server socket is closed.  
 // In this example, this does not happen, but you can do that to gracefully  
 // shut down your server.* f.channel().closeFuture().sync();  
} **finally** {  
 workerGroup.shutdownGracefully();  
 bossGroup.shutdownGracefully();  
}

服务端启动的时候，创建了两个NioEventLoopGroup，它们实际是两个独立的Reactor线程池。一个用于接收客户端的TCP连接，另一个用于处理I/O相关的读写操作，或者执行系统Task、定时任务Task等。

Netty用于接收客户端请求的线程池职责如下。

（1）接收客户端TCP连接，初始化Channel参数；

（2）将链路状态变更事件通知给ChannelPipeline。

Netty处理I/O操作的Reactor线程池职责如下。

（1）异步读取通信对端的数据报，发送读事件到ChannelPipeline；

（2）异步发送消息到通信对端，调用ChannelPipeline的消息发送接口；

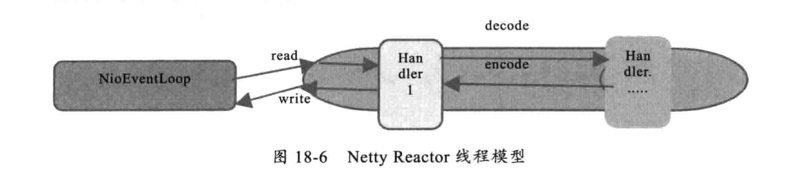
（3）执行系统调用Task；

（4）执行定时任务Task，例如链路空闲状态监测定时任务。

通过调整线程池的线程个数、是否共享线程池等方式，Netty的Reactor线程模型可以在单线程、多线程和主从多线程间切换，这种灵活的配置方式可以最大程度地满足不同用户的个性化定制。

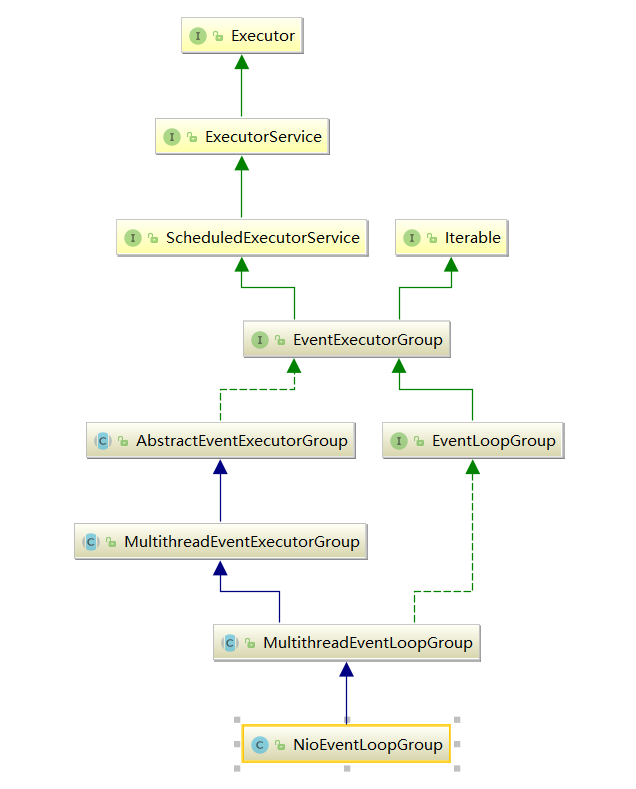
为了尽可能地提升性能，Netty在很多地方进行了无锁化的设计，例如在I/O线程内部进行串行操作，避免多线程竞争导致的性能下降问题。表面上看，串行化设计似乎CPU利用率不高，并发程度不够。但是，通过调整NIO线程池的线程参数，可以同时启动多个串行化的线程并行运行，这种局部无锁化的串行线程设计相比一个队列—多个工作线程的模型性能更优。

它的设计原理如图：



Netty的NioEventLoop读取到消息之后，直接调用ChannelPipeline的fireChannelRead (Object msg)。只要用户不主动切换线程，一直都是由NioEventLoop调用用户的Handler，期间不进行线程切换。这种串行化处理方式避免了多线程操作导致的锁的竞争，从性能角度看是最优的。

## NioEventLoopGroup



### NioEventLoopGroup

1. Executor

JDK标准接口，启动一个线程，只有一个函数execute，接收一个参数（Runnable）

1. ExecutorService

实现接口

1. shutdown

停掉所有线程，包括所有submitted。

1. isShutdown

当前实现是否停掉了。（if this executor has been shut down）

1. isTerminated

除非调用shutdown，否则永远是false的

1. awaitTermination

shutdown之后，等待线程执行完再关掉当前executor，和shutdown函数对比

1. submit(Callable task)

block当前线程，等待task执行，接收task返回的结果，返回一个Future，future.get获取task返回的结果

1. submit(Runnable task)

block当前线程，执行task线程，注意，这个返回null。

1. submit(Runnable task,T result)

block当前线程，task执行完后，返回一个result放在Future里了，get获取result

1. invokeAll

block当前线程，执行集合里的所有task，然后返回一个集合，里面全是Future，可以添加timeout参数

1. invokeAny

调用集合里的task，谁先完成就返回那个结果到Future里

1. ScheduledExecutorService

继承自ExecutorService，主要是添加了定时调用和间隔调用

1. **EventExecutorGroup**

Netty自定义的ExecutorService，主要是

1. next

Netty将ExecutorService组成一组，可以获取下一个

1. AbstractEventExecutorGroup

使用next()获取下一个ExecutorService，实现之前的所有接口

1. MultithreadEventExecutorGroup
   1. children

EventExecutor数组，EventExecutor接口继承自ExentExecutorGroup

* + 1. next()

获取下一个EventExecutor，相当于一个链表操作，每个都可以指向下一个

* + 1. inEventLoop

当前线程（currentThread）是否在当前event loop中执行

* + 1. inEventLoop(Thread thread)

同上

* + 1. newPromise

返回一个Promise

* + 1. newProgressivePromise

返回一个ProgressivePromise

* + 1. newSuccessedFuture(V result)

直接返回一个成功的Future，result写入到future中

* + 1. newFailedFuture

同上

* 1. readOnlyChildren
  2. terminatedChildren
  3. terminationFuture

返回一个默认的Promise，使用GlobalEventExecutor

* 1. chooser

EventExecutorChooserFactory.EventExecutorChooser

* 1. **构造函数**

返回executor参数为空时，设置默认参数，会为每个task创建一个新的线程。理论上之前说的那种tomcat就是这种实现方式，这样的话，长连接，长事物的情况，比如上传下载这些动作，可以用这个。

实例化children里的EventExecutor。这里会调用newChild生成，留给实现类去做了。

实例化新的chooser（DefaultEventExecutorChooserFactory），里面的两种实际上是一样的，不过位与运算要快于%运算。确实很蛋疼，不过也说明，项目研发很抠细节。

添加终止监听器。后面childrenset等等，不叙述了。

1. MultithreadEventLoopGroup
   1. DEFAULT\_EVENT\_LOOP\_THREADS

NettyRuntime.availableProcessors() \* 2 获取CPU等信息吧

* 1. **EventLoopGroup**

继承自EventExecutorGroup，主要添加了register函数，将channel注册到eventLoop中

* 1. EventLoop

继承EventLoopGroup和OrderedEventExecutor。真正的操作事件集合

* 1. register函数

使用next()获取EventLoop，将channel注册到EventLoop中

1. NioEventLoopGroup
   1. DefaultThreadFactory

Netty中ThreadFactory的默认实现。

* 1. SelectorProvider

Java中的标准实现，一般不用改

* 1. DefaultSelectStrategy

Netty的默认选择策略，后续会用到

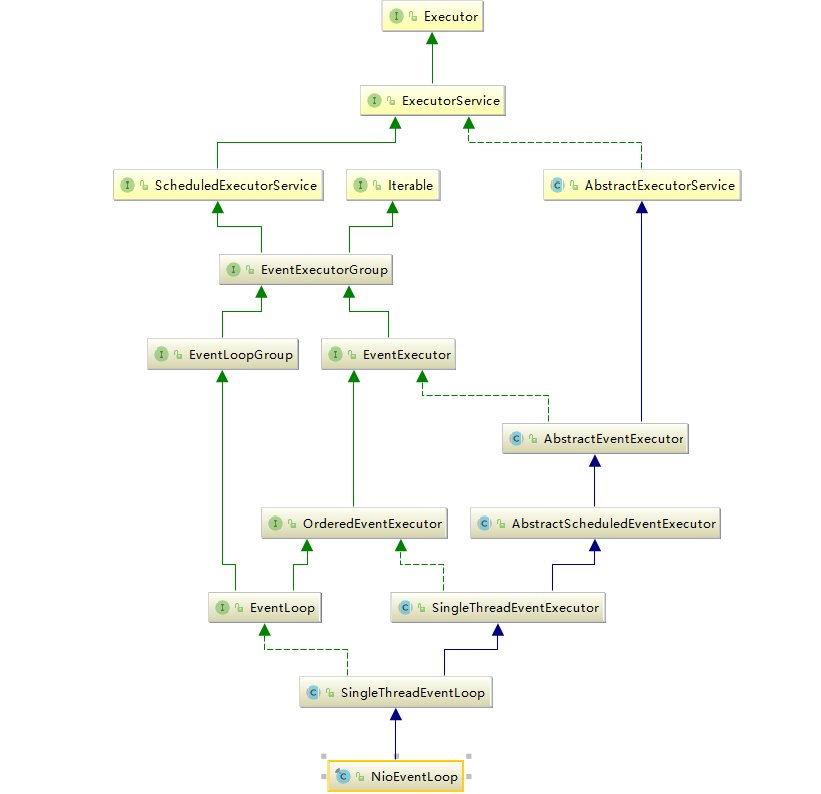
* 1. RejectedExecutionHandlers

也是后面说到

### NioEventLoop

*\* {****@link*** *SingleThreadEventLoop} implementation which register the {****@link*** *Channel}'s to a  
\* {****@link*** *Selector} and so does the multi-plexing of these in the event loop.*

注册channel到Selector上。



1. AbstractEventExecutor

主要是提供了AbstractExecutorService的默认实现

1. AbstractScheduledEventExecutor

添加了定时处理，Netty自己定义了一个队列Queen的设置，挺不错，其他地方也可以参照DefaultPriorityQueue

1. SingleThreadEventExecutor

所有的task事件全都在同一个线程里操作，节省了很多线程资源

1. DEFAULT\_MAX\_PENDING\_EXECUTOR\_TASKS

默认的pendingtask数量，默认16个，可以使用系统参数配置（**io.netty.eventexecutor.maxPendingTasks**）

1. **private static final** AtomicIntegerFieldUpdater<SingleThreadEventExecutor> ***STATE\_UPDATER*** =  
    AtomicIntegerFieldUpdater.*newUpdater*(SingleThreadEventExecutor.**class**, **"state"**);  
   **private static final** AtomicReferenceFieldUpdater<SingleThreadEventExecutor, ThreadProperties> ***PROPERTIES\_UPDATER*** =  
    AtomicReferenceFieldUpdater.*newUpdater*(  
    SingleThreadEventExecutor.**class**, ThreadProperties.**class**, **"threadProperties"**);

动态将当前Bean的某些属性变成原子变量，不管是基本变量还是reference，

1. taskQueue

实例化任务序列

1. SingleThreadEventLoop

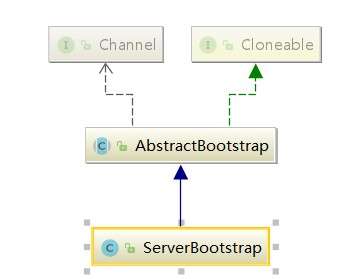
继承自SingleThreadEventExcutor，并且实现了EventLoop，EventLoop里的register等等一些。

### 总结

大概就是这样的，主要的逻辑是：

一切的起点都是Executor，ExecutorService用于调用新线程，后面ScheduledExecutorService是添加了一个定时任务实现，EventExecutorGroup实现了循环调用线程，EventLoopGroup实现将Channel注册到EventLoopGroup，MultiEventLoopGroup实现了多线程处理每一个Channel事件，基于CPU盒数算出的数量，当然也可以自己设定。比如设定多线程为10个，将EventLoopGroup（每一个EventLoopGroup都是继承自EventExecutor的）设定进待选择列表中，最终NioEventLoopGroup实现

## ServerBootstrap



基础设定，启动基点。

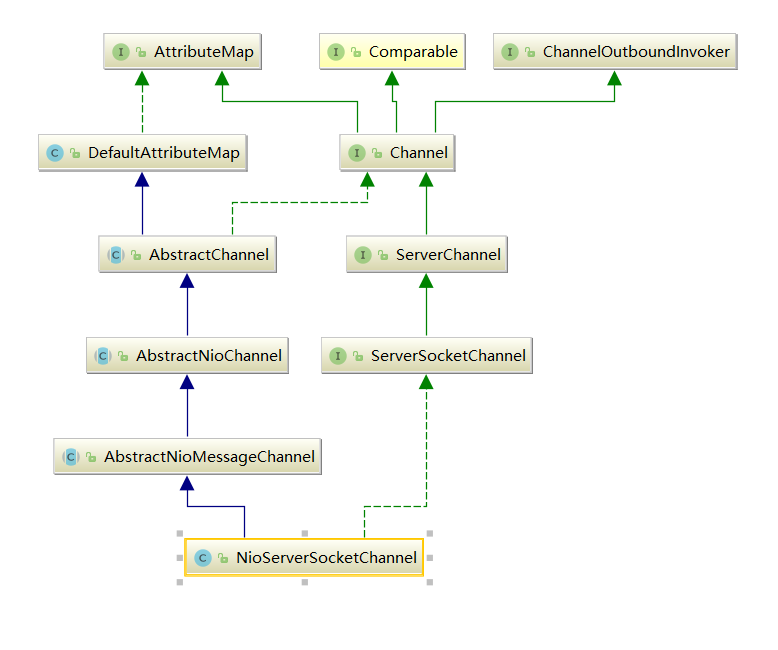
ParentGroup

childGroup

ReflectiveChannelFactory

默认的ChannelFactory，使用构造函数构建Channel

## NioServerSocketChannel



1. DefaultAttributeMap

Netty实现的默认的基于链表的数据存储，主要添加了AtomicReference等的操作，保证原子性

1. AbstractChannel

设定基础的Channel操作，比如操作自定义的PipeLine等，注册啊，什么的，动作的外围处理。当真实调用的时候，会调用AbstractNioMessageChannel等其他实现中的成员函数。

* + - 1. parent
      2. id

序列

* + - 1. unsafe

unsafe是netty系统内部流转，用来处理消息数据等的。应该是接收数据，zero-copy用的

* + - 1. pipeline

生成一个DefaultChannelPipeline，将当前channel注册到pipeline中，初始化HeadContext和TailContext，互相指引对方，设定一个链式上下文。

构造函数

调用java SelectorProvider生成ServerSocketChannel

调用AbstractChannel构造函数，parent为空

根据规则生成一个id

## 初始化

ServerBootStrap🡪bind

可以使用初始化的时候，我们自己做一个ServerBootStrap，然后复制给new的另一个，这样可以完成一部分初始化信息的录入

🡪 initAndRegister

初始化Channel，并将其注册到EventLoopGroup中

🡪init

添加一个ChannelInitializer

🡪 PipeLine.addLast

添加channelHandler到PipeLine中

🡪checkMultiplicity（校验当前的ChannelHandler是否开启了Sharable，如果没有开启的话，不可以注册到多个pipeline上）

🡪newContext（初始化Context--DefaultChannelHandlerContext）

🡪addLast0（改个链式操作，把这个新的加进去，没啥）

🡪callHandlerCallbackLater（添加一个callback的Task）

🡪config().group().register(channel);

NioEventLoopGroup.MultithreadEventLoopGroup.register

NioEventLoop.SingleThreadEventLoop.register

AbstractChannel.register

初始化注册的时候，将NioEventLoopGroup放到了EventExecutor池子中，这里选出来一个用来注册。

SingleThreadEventLoop调用注册的时候，调用channel中的unsafe去注册，最终还是调用AbstractChannel去注册。

🡪register0

🡪doRegister（AbstractChannel中注册，使用原生的javaChannel去向java的Selector中注册—AbstractNioChannel.doRegister。）

注册成功之后的回调函数，生成相应的调用链，这里还是独立线程

ChannelInitializer.handlerAdded（调用回调）

🡪initChannel（调用初始化channel，调用ServerBootstrap的那个初始化的Initializer，初始化BossGroup）

init初始化结束，添加handler到pipeline中，这里添加的ServerBootstrapAcceptor使用了异步的方式，不需要同步。

调用ChannelHandlerContext中的invokeChannelRegistered

Channel还没有连接，这里直接过了。