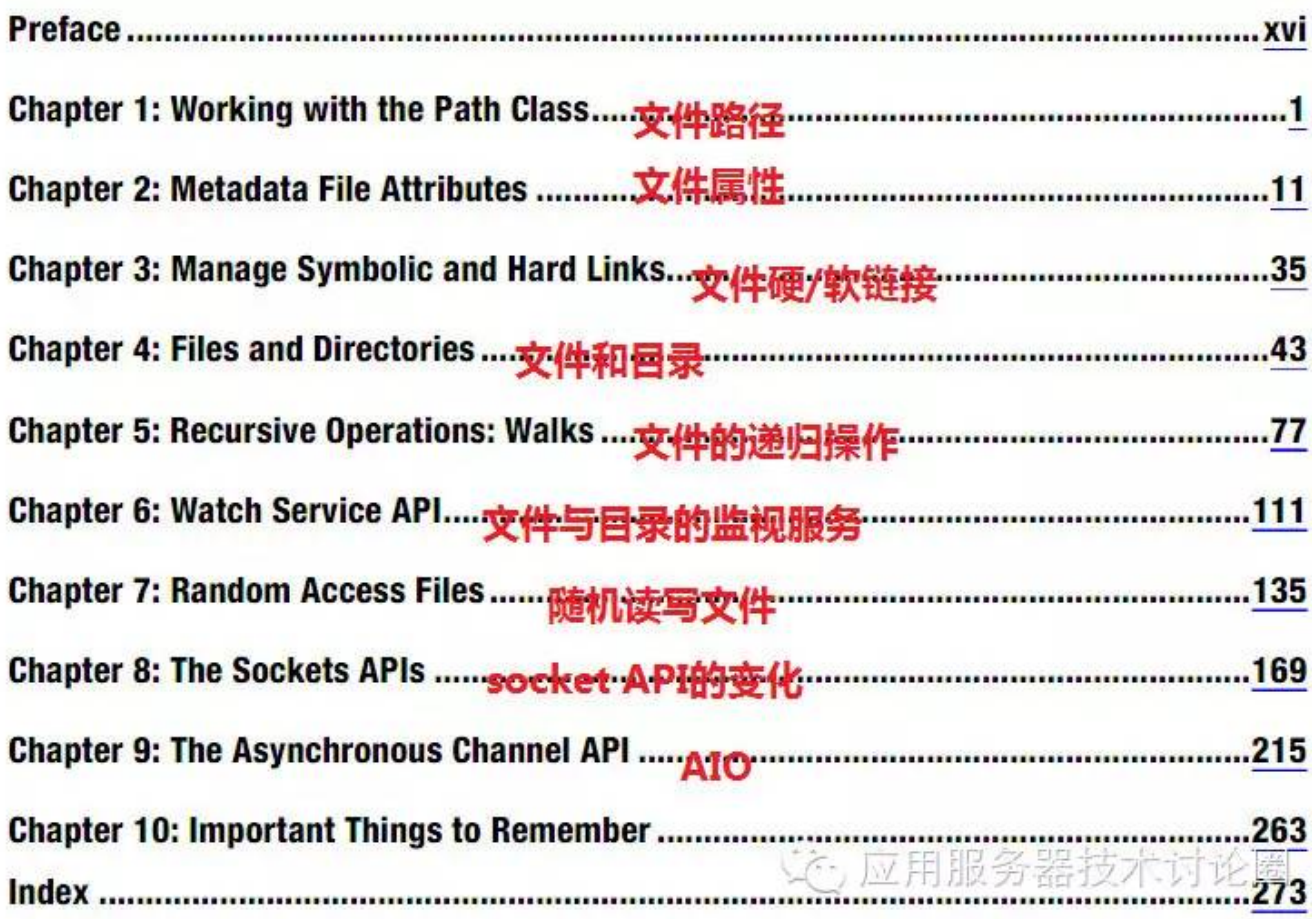
看看JDK7+的NIO2.0中的AIO例子

2016-02-06 14:12 feiying 0 阅读 99

NIO自从JDK5推出之后，从应用服务器前端的使用上，可以在java中使用上了操作系统的底层机制如select，poll等IO多路复用的技术，还有关于文件映射，DirectBuffer等；但是，却没有涉及操作系统的异步IO，这与在JDK发布的时候，异步IO对应的操作系统机制偏弱有关系，不过随着操作系统的发展，如FreeBSD中的异步IO系统调用效率已经非常高了，因此JDK7之后，AIO的API接口，随着JSR-203：More New APIs for Java Platform的推广并且实施，NIO2.0开始进入了实施。

在NIO2.0中，大多数是文件相关的内容，下面的截图是Pro JAVA NIO 2.0英文版的目录内容：



基本上NIO2.0都是文件相关的，最后两章一个是关于网络IO的socket接口的新变化，最后一章才是AIO，也就是IO模式的变化。

废话少说，我们上来先解析一段AIO的程序：


```
public class AIOServer {
```

```
public static void main(String args[]) throws InterruptedException,
ExecutionException, TimeoutException {
        //1.AIO的serversocketchannel打开
            AsynchronousServerSocketChannel server = AsynchronousServerSocketChannel.open()
.bind(new InetSocketAddress(7777));
System.out.println("Server listen on " + PORT);
            //5.最后handler注册完毕，服务器端AIO的通道开始监听，AIO服务器端设置完毕
server.accept(null,new CompletionHandler<AsynchronousSocketChannel, Object>() {//2.
注册AIO异步事件Handler
final ByteBuffer buffer = ByteBuffer.allocate(1024);
                //3.用户态IO就绪完成事件
public void completed(AsynchronousSocketChannel result,Object attachment) {
System.out.println(Thread.currentThread().getName());
System.out.println("start");
try {
                        //4.用户态进行IO操作，将用户态缓冲区的数据用于业务逻辑
    buffer.clear();
    result.read(buffer).get(100, TimeUnit.SECONDS);
    buffer.flip();
    System.out.println("received message: "+ new String(buffer.array()));
} catch (InterruptedException | ExecutionException e) {
    System.out.println(e.toString());
} catch (TimeoutException e) {
    e.printStackTrace();
} finally {
    try {
result.close();
server.accept(null, this);
    } catch (Exception e) {
System.out.println(e.toString());
    }
```

```
        }
        System.out.println("end");
    }
    public void failed(Throwable exc, Object attachment) {
       System.out.println("failed: " + exc);
    }
       });
    // 主线程继续自己的行为
    while (true) {
    System.out.println("main thread");
    Thread.sleep(1000);
    }
    }
}
```

总结一下，对于Java的AIO程序，API设计得非常好，非常简单：

通过异步的**AsynchronousServerSocketChannel打开**异步通道，然后打开socket端口，注册下handler事件，这样AIO服务器建立完毕了，

这不是Selector，还需要注册选择器，一顿轮询之类的麻烦事，因为这是IO多路复用的必须的步骤，单从这一点上来看，

异步IO要简单的多的多，仅仅2个步骤就搞定。

客户端的代码更为简洁：

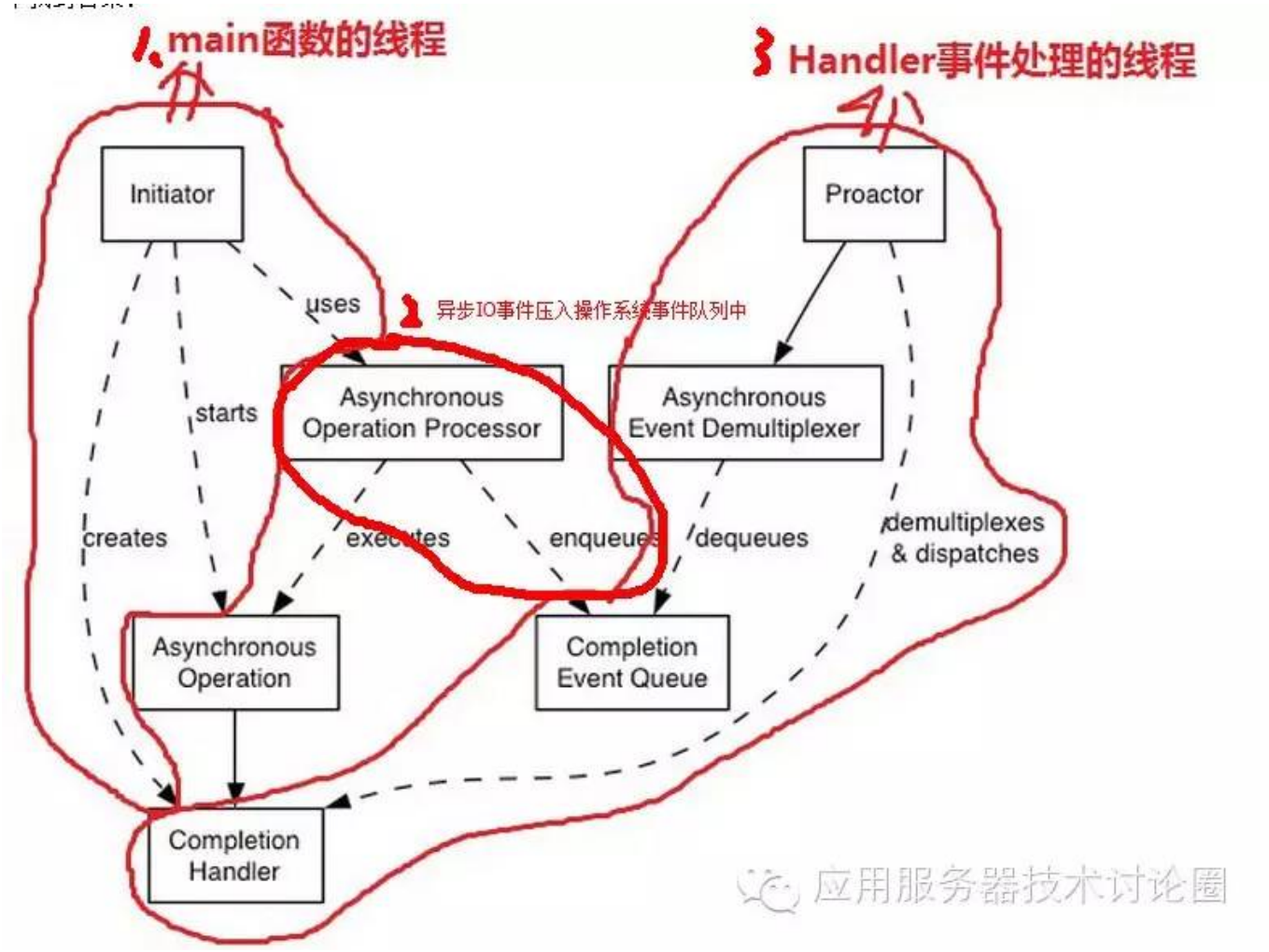
```
public class AIOClient {
    public static void main(String... args) throws Exception {
            //1.客户端socketchannel打开
    AsynchronousSocketChannel client = AsynchronousSocketChannel.open();
            //2. 连接服务器的serversocket，同样也可以客户端异步IO也可以注册xxxhandler进行回调
    client.connect(new InetSocketAddress("localhost", 9888)，attachment,xxxHandler);
            //3. 具体的客户端向服务器端发数据
    client.write(.....);
    }
```

}

客户端的流程和服务器端几乎类似，服务器端的异步IO仅仅多了一个绑定自己机器端口的这一个环节，而客户端的异步IO是连接对应的端口，而不至于像IO多路复用机制中的Selector中，服务器一顿轮询，监听xxx事件，客户端和服务器端代码差距较大；

对于上述的代码，以客户端为例，client.connect的第二个参数attachment，是客户端传给执行回调的handler的参数，例如一些配置信息，而需要值得注意的一点是，异步IO的handler的线程和当前main函数的主线程是不同的线程。

关于这点可以从专门描述异步IO的Proactor模式中找到答案:



从左侧来看，Initiator是上述程序中的main函数的主线程，它的作用是注册handler，开启异步IO，与异步处理器交互，将关注的事件压入到操作系统的事件队列中，仅此而已。而可以分析得出，对于Handler的线程最初的触发是由操作系统的底层异步IO机制中触发的，事件队列在操作系统内核中感知到，将事件传到异步IO多路分离器中，再传到JVM中的Proactor实现，进行分离，最终触发CompleleHandler的，严格意义上来说，Handler的线程应该仅仅就是JVM中的

上图的Proactor实现到Handler这一小块才是Handler的java线程关注的焦点。

如果按照上述的理解，**Handler线程按照默认的角度来将，都是在上图的Proactor实现位置进行new Thread，然后Handler结束之后，再将Thread进行释放掉。**

但是，需要注意的是，实现AIO的代码，通常都是应用服务器的前端，代码执行频繁，因此频繁的这种创建线程是否就是合理的呢？

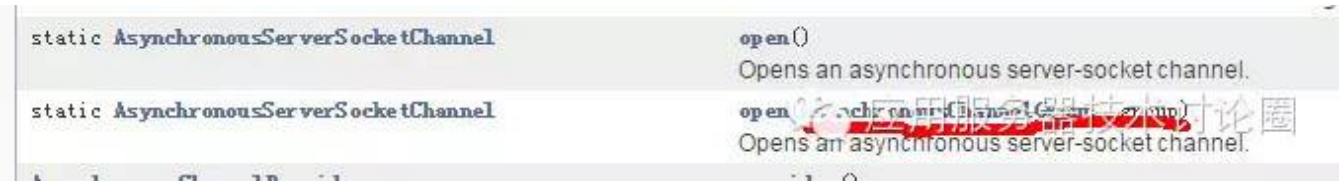
在JAVA AIO中，引入了一个AsynchronousChannelGroup：

异步channel的分组管理，目的是为了资源共享。

一个AsynchronousChannelGroup绑定一个线程池，这个线程池执行两个任务：处理IO事件和派发CompletionHandler。

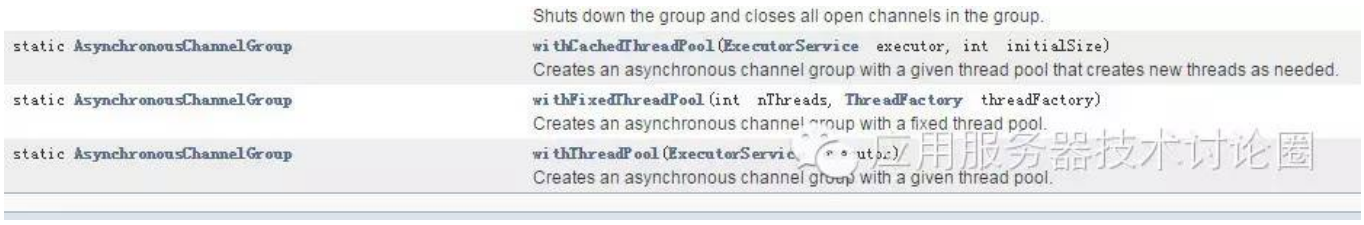
这两个任务，其实就是我们上面看到的Proactor模式中两部分的线程，一部分是main线程的，另一部分是Handler线程的。

AsynchronousServerSocketChannel创建的时候可以传入一个 AsynchronousChannelGroup，如下重构第二个参数：

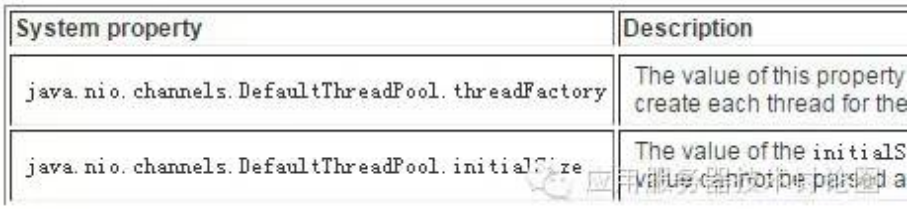


那么通过AsynchronousServerSocketChannel创建的 AsynchronousSocketChannel将同属于一个组，共享资源。这也就是意味着，上述的Proactor模式中的第2部分，第3部分都由这个AsynchronousChannelGroup来进行接管。

AsynchronousChannelGroup一般会绑定3种JDK自身的线程池：



如果没有明确绑定或者自己想DIY的话，AsynchronousChannelGroup会在初始化的时候，自动读取系统的环境变量，找



System property	Description
java.nio.channels.DefaultThreadPool.threadFactory	The value of this property create each thread for the
java.nio.channels.DefaultThreadPool.initialSize	The value of the initialS value cannot be parsed a

这两个配置，进行对应线程池自定义的初始化。

最后值得说的一点就是，上述的Handler注册仅仅是一种做法，因为Future的机制也在JDK7中引入，而Future也完全是为了异步而设计的，与异步IO也非常的契合，因此，同样异步IO可以和Future联动起来：

```
//1.AIO的server端绑定
server = AsynchronousServerSocketChannel.open().bind(new InetSocketAddress(PORT));
System.out.println("Server listen on " + PORT);
//2.AIO的server的启动，回调内容在Future中，这一步是异步的
Future<AsynchronousSocketChannel> future = server.accept();
//3.但是这个future.get是需要等到完成事件OK了，才返回，所以这一步是阻塞的
//当然，这一步一般会放到一个线程中来执行
AsynchronousSocketChannel socket = future.get();
//4.最后进行自己的业务逻辑处理
ByteBuffer readBuf = ByteBuffer.allocate(1024);
readBuf.clear();
socket.read(readBuf).get(100, TimeUnit.SECONDS);
readBuf.flip();
System.out.printf("received message:" + new String(readBuf.array()));
System.out.println(Thread.currentThread().getName());
```

如上述所说，Future作为任务的载体，最好不要立刻就future.get，否则即失去了AIO完全异步的意义了。

总结：

Java的AIO确确实实就是操作系统的异步IO机制，并且API中对于线程池绑定，

简化Handler注册等机制做了很多的心思，可以一试！

应用服务器技术讨论圈 更多文章

东方通加码大数据业务 拟募资8亿收购微智信业 (/html/308/201504/206211355/1.html)

玩转Netty – 从Netty3升级到Netty4 (/html/308/201504/206233287/1.html)

金蝶中间件2015招聘来吧！Come on！ (/html/308/201505/206307460/1.html)

GlassFish 4.1 发布，J2EE 应用服务器 (/html/308/201505/206323120/1.html)

Tomcat对keep-alive的实现逻辑 (/html/308/201505/206357679/1.html)

猜您喜欢

带你看世界 | 10个巧用地图元素的优秀网页设计 (/html/458/201506/245311983/1.html)

浅谈PHP自动化代码审计技术 (/html/312/201504/208522032/1.html)

k歌自助安全系统 (/html/223/201608/2651232000/1.html)

倒计时5天 | 国际体验设计大会优秀作品展抢先看 (/html/215/201407/200566937/1.html)

临时说一些吧 (/html/377/201404/200394170/1.html)