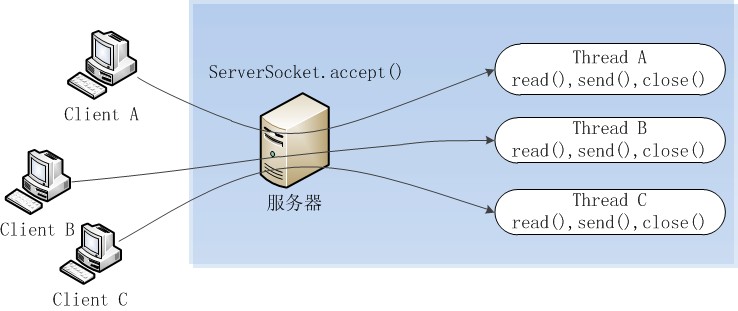
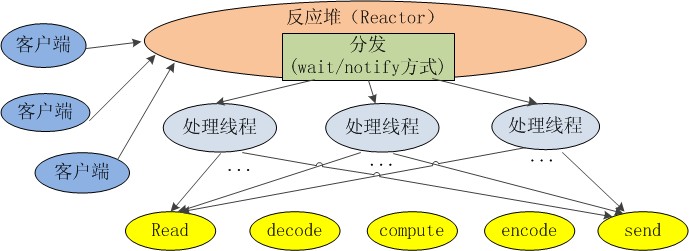
**Java NIO原理图文分析及代码实现**   
**前言:**  
  
最近在分析hadoop的RPC(Remote Procedure Call Protocol ，远程过程调用协议，它是一种通过网络从远程计算机程序上请求服务，而不需要了解底层网络技术的协议。可以参考：<http://baike.baidu.com/view/32726.htm> ）机制时，发现hadoop的RPC机制的实现主要用到了两个技术：动态代理（动态代理可以参考博客：<http://weixiaolu.iteye.com/blog/1477774> ）和java NIO。为了能够正确地分析hadoop的RPC源码，我觉得很有必要先研究一下java NIO的原理和具体实现。  
  
这篇博客我主要从两个方向来分析java NIO  
  
目录：  
一．java NIO 和阻塞I/O的区别  
     1. 阻塞I/O通信模型  
     2. java NIO原理及通信模型  
二．java NIO服务端和客户端代码实现   
  
**具体分析：**  
  
**一．java NIO 和阻塞I/O的区别**   
  
**1. 阻塞I/O通信模型**   
  
假如现在你对阻塞I/O已有了一定了解，我们知道阻塞I/O在调用InputStream.read()方法时是阻塞的，它会一直等到数据到来时（或超时）才会返回；同样，在调用ServerSocket.accept()方法时，也会一直阻塞到有客户端连接才会返回，每个客户端连接过来后，服务端都会启动一个线程去处理该客户端的请求。阻塞I/O的通信模型示意图如下：



如果你细细分析，一定会发现阻塞I/O存在一些缺点。根据阻塞I/O通信模型，我总结了它的两点缺点：  
1. 当客户端多时，会创建大量的处理线程。且每个线程都要占用栈空间和一些CPU时间  
  
2. 阻塞可能带来频繁的上下文切换，且大部分上下文切换可能是无意义的。  
  
在这种情况下非阻塞式I/O就有了它的应用前景。  
  
**2. java NIO原理及通信模型**  
  
Java NIO是在jdk1.4开始使用的，它既可以说成“新I/O”，也可以说成非阻塞式I/O。下面是java NIO的工作原理：  
  
1. 由一个专门的线程来处理所有的 IO 事件，并负责分发。   
2. 事件驱动机制：事件到的时候触发，而不是同步的去监视事件。   
3. 线程通讯：线程之间通过 wait,notify 等方式通讯。保证每次上下文切换都是有意义的。减少无谓的线程切换。   
  
阅读过一些资料之后，下面贴出我理解的java NIO的工作原理图：

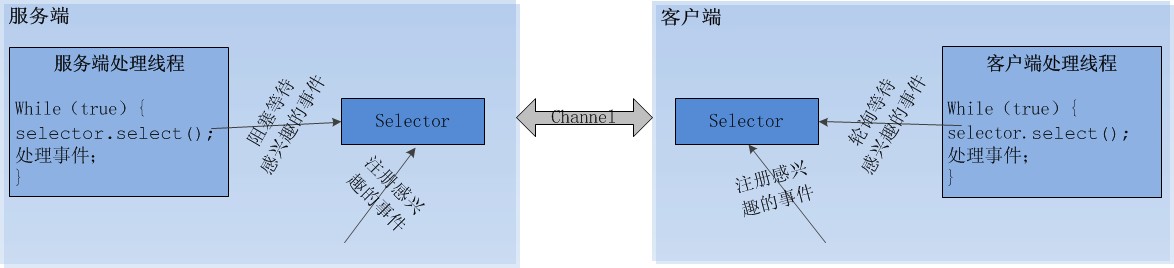


（注：每个线程的处理流程大概都是读取数据、解码、计算处理、编码、发送响应。）  
  
Java NIO的服务端只需启动一个专门的线程来处理所有的 IO 事件，这种通信模型是怎么实现的呢？呵呵，我们一起来探究它的奥秘吧。java NIO采用了双向通道（channel）进行数据传输，而不是单向的流（stream），在通道上可以注册我们感兴趣的事件。一共有以下四种事件：

|  |  |
| --- | --- |
| 事件名 | 对应值 |
| 服务端接收客户端连接事件 | SelectionKey.OP\_ACCEPT(16) |
| 客户端连接服务端事件 | SelectionKey.OP\_CONNECT(8) |
| 读事件 | SelectionKey.OP\_READ(1) |
| 写事件 | SelectionKey.OP\_WRITE(4) |

|  |  |
| --- | --- |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |

服务端和客户端各自维护一个管理通道的对象，我们称之为selector，该对象能检测一个或多个通道 (channel) 上的事件。我们以服务端为例，如果服务端的selector上注册了读事件，某时刻客户端给服务端发送了一些数据，阻塞I/O这时会调用read()方法阻塞地读取数据，而NIO的服务端会在selector中添加一个读事件。服务端的处理线程会轮询地访问selector，如果访问selector时发现有感兴趣的事件到达，则处理这些事件，如果没有感兴趣的事件到达，则处理线程会一直阻塞直到感兴趣的事件到达为止。下面是我理解的java NIO的通信模型示意图：



**二．java NIO服务端和客户端代码实现**   
  
为了更好地理解java NIO,下面贴出服务端和客户端的简单代码实现。  
  
**服务端：**

**Java代码  [收藏代码](javascript:void())**

1. **package** cn.nio;
3. **import** java.io.IOException;
4. **import** java.net.InetSocketAddress;
5. **import** java.nio.ByteBuffer;
6. **import** java.nio.channels.SelectionKey;
7. **import** java.nio.channels.Selector;
8. **import** java.nio.channels.ServerSocketChannel;
9. **import** java.nio.channels.SocketChannel;
10. **import** java.util.Iterator;
12. /\*\*
13. \* NIO服务端
14. \* @author 小路
15. \*/
16. **public** **class** NIOServer {
17. //通道管理器
18. **private** Selector selector;
20. /\*\*
21. \* 获得一个ServerSocket通道，并对该通道做一些初始化的工作
22. \* @param port  绑定的端口号
23. \* @throws IOException
24. \*/
25. **public** **void** initServer(**int** port) **throws** IOException {
26. // 获得一个ServerSocket通道
27. ServerSocketChannel serverChannel = ServerSocketChannel.open();
28. // 设置通道为非阻塞
29. serverChannel.configureBlocking(**false**);
30. // 将该通道对应的ServerSocket绑定到port端口
31. serverChannel.socket().bind(**new** InetSocketAddress(port));
32. // 获得一个通道管理器
33. **this**.selector = Selector.open();
34. //将通道管理器和该通道绑定，并为该通道注册SelectionKey.OP\_ACCEPT事件,注册该事件后，
35. //当该事件到达时，selector.select()会返回，如果该事件没到达selector.select()会一直阻塞。
36. serverChannel.register(selector, SelectionKey.OP\_ACCEPT);
37. }
39. /\*\*
40. \* 采用轮询的方式监听selector上是否有需要处理的事件，如果有，则进行处理
41. \* @throws IOException
42. \*/
43. @SuppressWarnings("unchecked")
44. **public** **void** listen() **throws** IOException {
45. System.out.println("服务端启动成功！");
46. // 轮询访问selector
47. **while** (**true**) {
48. //当注册的事件到达时，方法返回；否则,该方法会一直阻塞
49. selector.select();
50. // 获得selector中选中的项的迭代器，选中的项为注册的事件
51. Iterator ite = **this**.selector.selectedKeys().iterator();
52. **while** (ite.hasNext()) {
53. SelectionKey key = (SelectionKey) ite.next();
54. // 删除已选的key,以防重复处理
55. ite.remove();
56. // 客户端请求连接事件
57. **if** (key.isAcceptable()) {
58. ServerSocketChannel server = (ServerSocketChannel) key
59. .channel();
60. // 获得和客户端连接的通道
61. SocketChannel channel = server.accept();
62. // 设置成非阻塞
63. channel.configureBlocking(**false**);
65. //在这里可以给客户端发送信息哦
66. channel.write(ByteBuffer.wrap(**new** String("向客户端发送了一条信息").getBytes()));
67. //在和客户端连接成功之后，为了可以接收到客户端的信息，需要给通道设置读的权限。
68. channel.register(**this**.selector, SelectionKey.OP\_READ);
70. // 获得了可读的事件
71. } **else** **if** (key.isReadable()) {
72. read(key);
73. }
75. }
77. }
78. }
79. /\*\*
80. \* 处理读取客户端发来的信息 的事件
81. \* @param key
82. \* @throws IOException
83. \*/
84. **public** **void** read(SelectionKey key) **throws** IOException{
85. // 服务器可读取消息:得到事件发生的Socket通道
86. SocketChannel channel = (SocketChannel) key.channel();
87. // 创建读取的缓冲区
88. ByteBuffer buffer = ByteBuffer.allocate(10);
89. channel.read(buffer);
90. **byte**[] data = buffer.array();
91. String msg = **new** String(data).trim();
92. System.out.println("服务端收到信息："+msg);
93. ByteBuffer outBuffer = ByteBuffer.wrap(msg.getBytes());
94. channel.write(outBuffer);// 将消息回送给客户端
95. }
97. /\*\*
98. \* 启动服务端测试
99. \* @throws IOException
100. \*/
101. **public** **static** **void** main(String[] args) **throws** IOException {
102. NIOServer server = **new** NIOServer();
103. server.initServer(8000);
104. server.listen();
105. }
107. }

**客户端：**

**Java代码  [收藏代码](javascript:void())**

1. **package** cn.nio;
3. **import** java.io.IOException;
4. **import** java.net.InetSocketAddress;
5. **import** java.nio.ByteBuffer;
6. **import** java.nio.channels.SelectionKey;
7. **import** java.nio.channels.Selector;
8. **import** java.nio.channels.SocketChannel;
9. **import** java.util.Iterator;
11. /\*\*
12. \* NIO客户端
13. \* @author 小路
14. \*/
15. **public** **class** NIOClient {
16. //通道管理器
17. **private** Selector selector;
19. /\*\*
20. \* 获得一个Socket通道，并对该通道做一些初始化的工作
21. \* @param ip 连接的服务器的ip
22. \* @param port  连接的服务器的端口号
23. \* @throws IOException
24. \*/
25. **public** **void** initClient(String ip,**int** port) **throws** IOException {
26. // 获得一个Socket通道
27. SocketChannel channel = SocketChannel.open();
28. // 设置通道为非阻塞
29. channel.configureBlocking(**false**);
30. // 获得一个通道管理器
31. **this**.selector = Selector.open();
33. // 客户端连接服务器,其实方法执行并没有实现连接，需要在listen（）方法中调
34. //用channel.finishConnect();才能完成连接
35. channel.connect(**new** InetSocketAddress(ip,port));
36. //将通道管理器和该通道绑定，并为该通道注册SelectionKey.OP\_CONNECT事件。
37. channel.register(selector, SelectionKey.OP\_CONNECT);
38. }
40. /\*\*
41. \* 采用轮询的方式监听selector上是否有需要处理的事件，如果有，则进行处理
42. \* @throws IOException
43. \*/
44. @SuppressWarnings("unchecked")
45. **public** **void** listen() **throws** IOException {
46. // 轮询访问selector
47. **while** (**true**) {
48. selector.select();
49. // 获得selector中选中的项的迭代器
50. Iterator ite = **this**.selector.selectedKeys().iterator();
51. **while** (ite.hasNext()) {
52. SelectionKey key = (SelectionKey) ite.next();
53. // 删除已选的key,以防重复处理
54. ite.remove();
55. // 连接事件发生
56. **if** (key.isConnectable()) {
57. SocketChannel channel = (SocketChannel) key
58. .channel();
59. // 如果正在连接，则完成连接
60. **if**(channel.isConnectionPending()){
61. channel.finishConnect();
63. }
64. // 设置成非阻塞
65. channel.configureBlocking(**false**);
67. //在这里可以给服务端发送信息哦
68. channel.write(ByteBuffer.wrap(**new** String("向服务端发送了一条信息").getBytes()));
69. //在和服务端连接成功之后，为了可以接收到服务端的信息，需要给通道设置读的权限。
70. channel.register(**this**.selector, SelectionKey.OP\_READ);
72. // 获得了可读的事件
73. } **else** **if** (key.isReadable()) {
74. read(key);
75. }
77. }
79. }
80. }
81. /\*\*
82. \* 处理读取服务端发来的信息 的事件
83. \* @param key
84. \* @throws IOException
85. \*/
86. **public** **void** read(SelectionKey key) **throws** IOException{
87. //和服务端的read方法一样
88. }

91. /\*\*
92. \* 启动客户端测试
93. \* @throws IOException
94. \*/
95. **public** **static** **void** main(String[] args) **throws** IOException {
96. NIOClient client = **new** NIOClient();
97. client.initClient("localhost",8000);
98. client.listen();
99. }
101. }