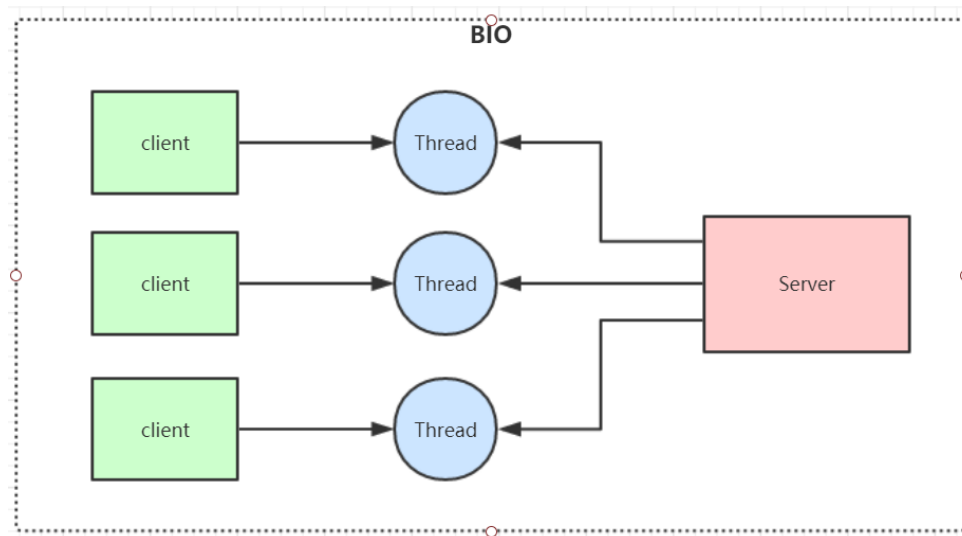


IO模型

IO模型就是说用什么样的通道进行数据的发送和接收，Java共支持3种网络编程IO模式：**BIO**，**NIO**，**AIO**

BIO(Blocking IO)

同步阻塞模型，一个客户端连接对应一个处理线程



BIO代码示例：

```
1 package com.tuling.bio;
2
3 import java.io.IOException;
4 import java.net.ServerSocket;
5 import java.net.Socket;
6
7 public class SocketServer {
8     public static void main(String[] args) throws IOException {
9         ServerSocket serverSocket = new ServerSocket(9000);
10        while (true) {
11            System.out.println("等待连接。。");
12            //阻塞方法
13            Socket clientSocket = serverSocket.accept();
14            System.out.println("有客户端连接了。。");
15            handler(clientSocket);
16        }
17        /*new Thread(new Runnable() {
18            @Override
19            public void run() {
20                try {
21                    handler(clientSocket);
22                } catch (IOException e) {
23                    e.printStackTrace();
24                }
25            }
26        }).start();*/
27    }
28 }
29
30 private static void handler(Socket clientSocket) throws IOException {
31     byte[] bytes = new byte[1024];
32     System.out.println("准备read。。");
33     //接收客户端的数据，阻塞方法，没有数据可读时就阻塞
34     int read = clientSocket.getInputStream().read(bytes);
35     System.out.println("read完毕。。");
36     if (read != -1) {
```

```

37 System.out.println("接收到客户端的数据: " + new String(bytes, 0, read));
38 }
39 clientSocket.getOutputStream().write("HelloClient".getBytes());
40 clientSocket.getOutputStream().flush();
41 }
42 }
43

```

```

1 //客户端代码
2 public class SocketClient {
3     public static void main(String[] args) throws IOException {
4         Socket socket = new Socket("localhost", 9000);
5         //向服务端发送数据
6         socket.getOutputStream().write("HelloServer".getBytes());
7         socket.getOutputStream().flush();
8         System.out.println("向服务端发送数据结束");
9         byte[] bytes = new byte[1024];
10        //接收服务端回传的数据
11        socket.getInputStream().read(bytes);
12        System.out.println("接收到服务端的数据: " + new String(bytes));
13        socket.close();
14    }
15 }

```

缺点:

- 1、IO代码里read操作是阻塞操作，如果连接不做数据读写操作会导致线程阻塞，浪费资源
- 2、如果线程很多，会导致服务器线程太多，压力太大，比如C10K问题

应用场景:

BIO 方式适用于连接数目比较小且固定的架构，这种方式对服务器资源要求比较高，但程序简单易理解。

NIO(Non Blocking IO)

同步非阻塞，服务器实现模式为一个线程可以处理多个请求(连接)，客户端发送的连接请求都会注册到多路复用器selector上，多路复用器轮询到连接有IO请求就进行处理，JDK1.4开始引入。

应用场景:

NIO方式适用于连接数目多且连接比较短（轻操作）的架构，比如聊天服务器，弹幕系统，服务器间通讯，编程比较复杂

NIO非阻塞代码示例:

```

1 package com.tuling.nio;
2
3 import java.io.IOException;
4 import java.net.InetSocketAddress;
5 import java.nio.ByteBuffer;
6 import java.nio.channels.ServerSocketChannel;
7 import java.nio.channels.SocketChannel;
8 import java.util.ArrayList;
9 import java.util.Iterator;
10 import java.util.List;
11
12 public class NioServer {
13
14     // 保存客户端连接
15     static List<SocketChannel> channelList = new ArrayList<>();
16
17     public static void main(String[] args) throws IOException, InterruptedException {
18
19         // 创建NIO ServerSocketChannel,与BIO的serverSocket类似
20         ServerSocketChannel serverSocket = ServerSocketChannel.open();
21         serverSocket.socket().bind(new InetSocketAddress(9000));
22         // 设置ServerSocketChannel为非阻塞

```

```

23 serverSocket.configureBlocking(false);
24 System.out.println("服务启动成功");
25
26 while (true) {
27     // 非阻塞模式accept方法不会阻塞，否则会阻塞
28     // NIO的非阻塞是由操作系统内部实现的，底层调用了linux内核的accept函数
29     SocketChannel socketChannel = serverSocket.accept();
30     if (socketChannel != null) { // 如果有客户端进行连接
31         System.out.println("连接成功");
32         // 设置SocketChannel为非阻塞
33         socketChannel.configureBlocking(false);
34         // 保存客户端连接在List中
35         channelList.add(socketChannel);
36     }
37     // 遍历连接进行数据读取
38     Iterator<SocketChannel> iterator = channelList.iterator();
39     while (iterator.hasNext()) {
40         SocketChannel sc = iterator.next();
41         ByteBuffer byteBuffer = ByteBuffer.allocate(128);
42         // 非阻塞模式read方法不会阻塞，否则会阻塞
43         int len = sc.read(byteBuffer);
44         // 如果有数据，把数据打印出来
45         if (len > 0) {
46             System.out.println("接收到消息: " + new String(byteBuffer.array()));
47         } else if (len == -1) { // 如果客户端断开，把socket从集合中去掉
48             iterator.remove();
49             System.out.println("客户端断开连接");
50         }
51     }
52 }
53 }
54 }

```

总结：如果连接数太多的话，会有大量的无效遍历，假如有10000个连接，其中只有1000个连接有写数据，但是由于其他9000个连接并没有断开，我们还是要每次轮询遍历一万次，其中有十分之九的遍历都是无效的，这显然不是一个让人很满意的状态。

NIO引入多路复用器代码示例：

```

1 package com.tuling.nio;
2
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;
11 import java.util.Set;
12
13 public class NioSelectorServer {
14
15     public static void main(String[] args) throws IOException, InterruptedException {
16
17         // 创建NIO ServerSocketChannel
18         ServerSocketChannel serverSocket = ServerSocketChannel.open();
19         serverSocket.socket().bind(new InetSocketAddress(9000));
20         // 设置ServerSocketChannel为非阻塞
21         serverSocket.configureBlocking(false);
22         // 打开Selector处理Channel，即创建epoll
23         Selector selector = Selector.open();
24         // 把ServerSocketChannel注册到selector上，并且selector对客户端accept连接操作感兴趣

```

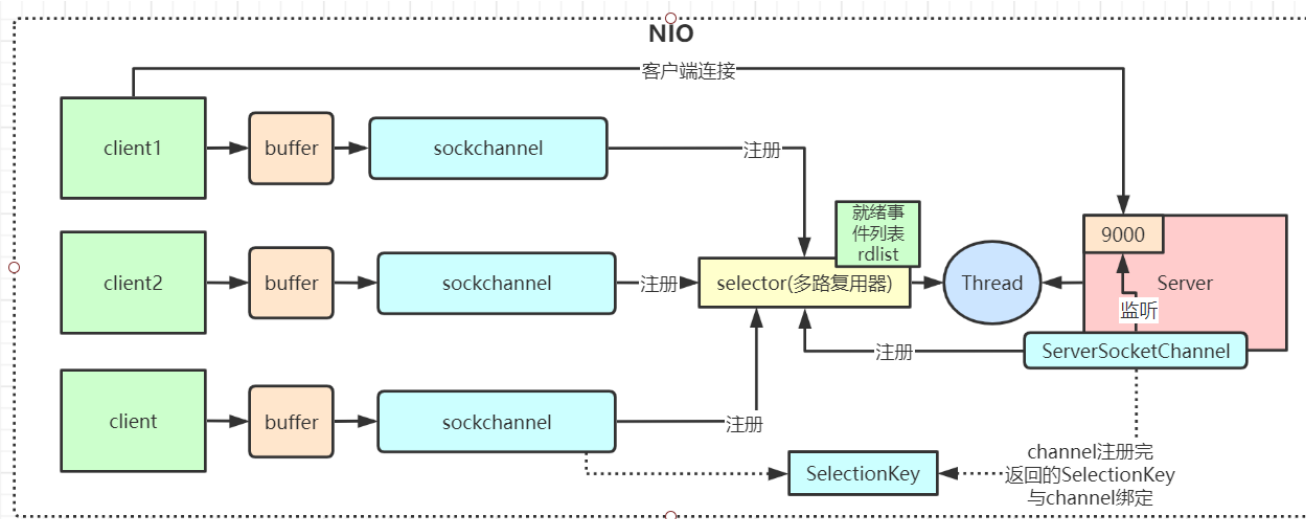
```

25 serverSocket.register(selector, SelectionKey.OP_ACCEPT);
26 System.out.println("服务启动成功");
27
28 while (true) {
29     // 阻塞等待需要处理的事件发生
30     selector.select();
31
32     // 获取selector中注册的全部事件的 SelectionKey 实例
33     Set<SelectionKey> selectionKeys = selector.selectedKeys();
34     Iterator<SelectionKey> iterator = selectionKeys.iterator();
35
36     // 遍历SelectionKey对事件进行处理
37     while (iterator.hasNext()) {
38         SelectionKey key = iterator.next();
39         // 如果是OP_ACCEPT事件，则进行连接获取和事件注册
40         if (key.isAcceptable()) {
41             ServerSocketChannel server = (ServerSocketChannel) key.channel();
42             SocketChannel socketChannel = server.accept();
43             socketChannel.configureBlocking(false);
44             // 这里只注册了读事件，如果需要给客户端发送数据可以注册写事件
45             socketChannel.register(selector, SelectionKey.OP_READ);
46             System.out.println("客户端连接成功");
47         } else if (key.isReadable()) { // 如果是OP_READ事件，则进行读取和打印
48             SocketChannel socketChannel = (SocketChannel) key.channel();
49             ByteBuffer byteBuffer = ByteBuffer.allocate(128);
50             int len = socketChannel.read(byteBuffer);
51             // 如果有数据，把数据打印出来
52             if (len > 0) {
53                 System.out.println("接收到消息: " + new String(byteBuffer.array()));
54             } else if (len == -1) { // 如果客户端断开连接，关闭Socket
55                 System.out.println("客户端断开连接");
56                 socketChannel.close();
57             }
58         }
59         // 从事件集合里删除本次处理的key，防止下次select重复处理
60         iterator.remove();
61     }
62 }
63 }
64 }

```

NIO 有三大核心组件：**Channel(通道)**，**Buffer(缓冲区)**，**Selector(多路复用器)**

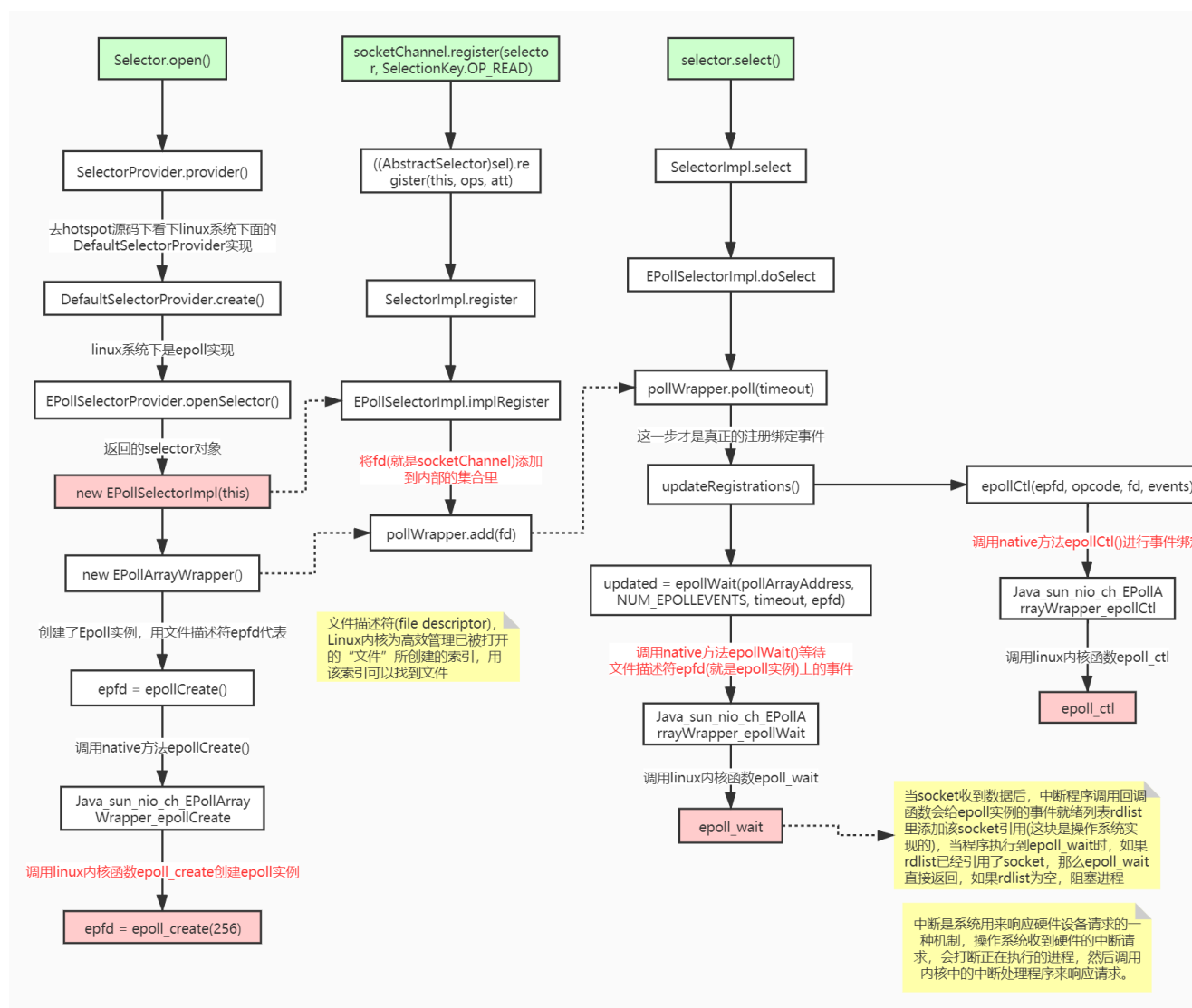
- 1、channel 类似于流，每个 channel 对应一个 buffer 缓冲区，buffer 底层就是个数组
- 2、channel 会注册到 selector 上，由 selector 根据 channel 读写事件的发生将其交由某个空闲的线程处理
- 3、NIO 的 Buffer 和 channel 都是既可以读也可以写



NIO底层在JDK1.4版本是用linux的内核函数select()或poll()来实现，跟上面的NioServer代码类似，selector每次都会轮询所有的sockchannel看下哪个channel有读写事件，有的话就处理，没有就继续遍历，JDK1.5开始引入了epoll基于事件响应机制来优化NIO。

NioSelectorServer 代码里如下几个方法非常重要，我们从Hotspot与Linux内核函数级别来理解下

```
1 Selector.open() //创建多路复用器
2 socketChannel.register() //将channel注册到多路复用器上
3 selector.select() //阻塞等待需要处理的事件发生
```



总结：NIO整个调用流程就是Java调用了操作系统的内核函数来创建Socket，获取到Socket的文件描述符，再创建一个Selector对象，对应操作系统的Epoll描述符，将获取到的Socket连接的文件描述符的事件绑定到Selector对应的Epoll文件描述符上，进行事件的异步通知，这样就实现了使用一条线程，并且不需要太多的无效的遍历，将事件处理交给了操作系统内核(操作系统中断程序实现)，大大提高了效率。

Epoll函数详解

```
1 int epoll_create(int size);
```

创建一个epoll实例，并返回一个非负数作为文件描述符，用于对epoll接口的所有后续调用。参数size代表可能会容纳size个描述符，但size不是一个最大值，只是提示操作系统它的数量级，现在这个参数基本上已经弃用了。

```
1 int epoll_ctl(int epfd, int op, int fd, struct epoll_event *event);
```

使用文件描述符epfd引用的epoll实例，对目标文件描述符fd执行op操作。

参数epfd表示epoll对应的文件描述符，参数fd表示socket对应的文件描述符。

参数op有以下几个值：

EPOLL_CTL_ADD：注册新的fd到epfd中，并关联事件event；

EPOLL_CTL_MOD：修改已经注册的fd的监听事件；

EPOLL_CTL_DEL：从epfd中移除fd，并且忽略掉绑定的event，这时event可以为null；

参数event是一个结构体

```
1 struct epoll_event {
2     __uint32_t events; /* Epoll events */
3     epoll_data_t data; /* User data variable */
4 };
5
6 typedef union epoll_data {
7     void *ptr;
8     int fd;
9     __uint32_t u32;
10    __uint64_t u64;
11 } epoll_data_t;
```

events有很多可选值，这里只举例最常见的几个：

EPOLLIN：表示对应的文件描述符是可读的；

EPOLLOUT：表示对应的文件描述符是可写的；

EPOLLERR：表示对应的文件描述符发生了错误；

成功则返回0，失败返回-1

```
1 int epoll_wait(int epfd, struct epoll_event *events, int maxevents, int timeout);
```

等待文件描述符epfd上的事件。

epfd是Epoll对应的文件描述符，events表示调用者所有可用事件的集合，maxevents表示最多等到多少个事件就返回，timeout是超时时间。

I/O多路复用底层主要用的Linux 内核函数（select，poll，epoll）来实现，windows不支持epoll实现，windows底层是基于winsock2的select函数实现的(不开源)

	select	poll	epoll(jdk 1.5及以上)
操作方式	遍历	遍历	回调
底层实现	数组	链表	哈希表
IO效率	每次调用都进行线性遍历，时间复杂度为O(n)	每次调用都进行线性遍历，时间复杂度为O(n)	事件通知方式，每当有IO事件就绪，系统注册的回调函数就会被调用，时间复杂度O(1)
最大连接	有上限	无上限	无上限

Redis线程模型

Redis就是典型的基于epoll的NIO线程模型(nginx也是), epoll实例收集所有事件(连接与读写事件), 由一个服务端线程连续处理所有事件命令。

Redis底层关于epoll的源码实现在redis的src源码目录的ae_epoll.c文件里, 感兴趣可以自行研究。

AIO(NIO 2.0)

异步非阻塞, 由操作系统完成后回调通知服务端程序启动线程去处理, 一般适用于连接数较多且连接时间较长的应用

应用场景:

AIO方式适用于连接数目多且连接比较长(重操作)的架构, JDK7 开始支持

AIO代码示例:

```
1 package com.tuling.aio;
2
3 import java.io.IOException;
4 import java.net.InetSocketAddress;
5 import java.nio.ByteBuffer;
6 import java.nio.channels.AsynchronousServerSocketChannel;
7 import java.nio.channels.AsynchronousSocketChannel;
8 import java.nio.channels.CompletionHandler;
9
10 public class AIOServer {
11
12     public static void main(String[] args) throws Exception {
13         final AsynchronousServerSocketChannel serverChannel =
14             AsynchronousServerSocketChannel.open().bind(new InetSocketAddress(9000));
15
16         serverChannel.accept(null, new CompletionHandler<AsynchronousSocketChannel, Object>() {
17             @Override
18             public void completed(AsynchronousSocketChannel socketChannel, Object attachment) {
19                 try {
20                     System.out.println("2--"+Thread.currentThread().getName());
21                     // 再此接收客户端连接, 如果不写这行代码后面的客户端连接不上服务端
22                     serverChannel.accept(attachment, this);
23                     System.out.println(socketChannel.getRemoteAddress());
24                     ByteBuffer buffer = ByteBuffer.allocate(1024);
25                     socketChannel.read(buffer, buffer, new CompletionHandler<Integer, ByteBuffer>() {
26                         @Override
27                         public void completed(Integer result, ByteBuffer buffer) {
28                             System.out.println("3--"+Thread.currentThread().getName());
29                             buffer.flip();
30                             System.out.println(new String(buffer.array(), 0, result));
31                             socketChannel.write(ByteBuffer.wrap("HelloClient".getBytes()));
32                         }
33                     });
34                     @Override
35                     public void failed(Throwable exc, ByteBuffer buffer) {
36                         exc.printStackTrace();
37                     }
38                 });
39             } catch (IOException e) {
40                 e.printStackTrace();
41             }
42         }
43
44         @Override
45         public void failed(Throwable exc, Object attachment) {
46             exc.printStackTrace();
47         }
48     });
49
50     System.out.println("1--"+Thread.currentThread().getName());
```

```

51 Thread.sleep(Integer.MAX_VALUE);
52 }
53 }

```

```

1 package com.tuling.aio;
2
3 import java.net.InetSocketAddress;
4 import java.nio.ByteBuffer;
5 import java.nio.channels.AsynchronousSocketChannel;
6
7 public class AIOClient {
8
9     public static void main(String... args) throws Exception {
10         AsynchronousSocketChannel socketChannel = AsynchronousSocketChannel.open();
11         socketChannel.connect(new InetSocketAddress("127.0.0.1", 9000)).get();
12         socketChannel.write(ByteBuffer.wrap("HelloServer".getBytes()));
13         ByteBuffer buffer = ByteBuffer.allocate(512);
14         Integer len = socketChannel.read(buffer).get();
15         if (len != -1) {
16             System.out.println("客户端收到信息: " + new String(buffer.array(), 0, len));
17         }
18     }
19 }

```

BIO、NIO、AIO 对比:

	BIO	NIO	AIO
IO 模型	同步阻塞	同步非阻塞（多路复用）	异步非阻塞
编程难度	简单	复杂	复杂
可靠性	差	好	好
吞吐量	低	高	高

为什么Netty使用NIO而不是AIO?

在Linux系统上，AIO的底层实现仍使用Epoll，没有很好实现AIO，因此在性能上没有明显的优势，而且被JDK封装了一层不容易深度优化，Linux上AIO还不够成熟。Netty是**异步非阻塞**框架，Netty在NIO上做了很多异步的封装。

同步异步与阻塞非阻塞(段子)

老张爱喝茶，废话不说，煮开水。

出场人物：老张，水壶两把（普通水壶，简称水壶；会响的水壶，简称响水壶）。

1 老张把水壶放到火上，立等水开。（**同步阻塞**）

老张觉得自己有点傻

2 老张把水壶放到火上，去客厅看电视，时不时去厨房看看水开没有。（**同步非阻塞**）

老张还是觉得自己有点傻，于是变高端了，买了把会响笛的那种水壶。水开之后，能大声发出嘀~~~~的噪音。

3 老张把响水壶放到火上，立等水开。（**异步阻塞**）

老张觉得这样傻等意义不大

4 老张把响水壶放到火上，去客厅看电视，水壶响之前不再去看它了，响了再去拿壶。（**异步非阻塞**）

老张觉得自己聪明了。

所谓同步异步，只是对于水壶而言。

普通水壶，同步；响水壶，异步。

虽然都能干活，但响水壶可以在自己完工之后，提示老张水开了。这是普通水壶所不能及的。

同步只能让调用者去轮询自己（情况2中），造成老张效率的低下。

所谓阻塞非阻塞，仅仅对于老张而言。

立等的老张，阻塞；看电视的老张，非阻塞。

1 文档: [01-VIP-深入Hotspot源码与Linux内核理解NIO与Epoll](#)

2 链接: <http://note.youdao.com/noteshare?id=916f44987d1fe0e35ec935bf5391d762&sub=4A9134F6DB4F424EB5F1CC2AF939B11B>