epoll 与 select

> 本节出处：
> https://www.zhihu.com/question/20122137/answer/146866418，糅杂了一

> 些个人思考

第一部分：select 和 epoll的任务

关键词：应用程序 文件句柄 用户态 内核态 监控者

要比较 epoll 相比较 select 高效在什么地方，就需要比较二者做相同事情的方法。

要完成对I/O流的复用需要完成如下几个事情：

1. 用户态怎么**将文件句柄传递到内核态**？
2. 内核态怎么**判断I/O流可读可写**？
3. 内核怎么通**知监控者有I/O流可读可写**？
4. 监控者如何找到可读可写的 **I/O 流并传递给用户态应用程序**？
5. 继续循环时监控者怎样重复上述步骤？

搞清楚上述的步骤也就能解开epoll高效的原因了。

select 的做法

步骤1的解法：select 函数创建 3 个文件描述符集，并将这些**文件描述符全部拷贝到内核中**，32 位机器上限制了文件句柄的最大的数量为1024，在等待 IO 时，select 会在对应文件描述符条件上陷入休眠。

步骤2的解法：当 IO 事件到达主机时，触发**IO中断**，CPU 根据操作系统预设置的中断向量表调用相应处理程序，例如陷入 TCP/IP 处理程序，随后内核针对读缓冲区和写缓冲区来判断是否可读可写，这个动作和 select 无关。

步骤3的解法：内核在检测到文件句柄可读/可写时就设置相应条件（条件变量），唤醒正在等待该套接字的线程，这里是监控者 select，select 被内核触发之后，返回可读写的文件句柄的总数。

步骤4的解法：select 会将之前传递给内核的**文件句柄再次从内核传到用户态**（全部拷贝），select 返回给用户态的只是可读可写的文件句柄总数，再使用 FD_ISSET 宏函数来检测哪些文件I/O可读可写，这是通过**遍历**解决的。

步骤5的解法：再次监控 IO 需要重复上述步骤，即重新将这些**文件描述符全部拷贝到内核中**。

epoll的做法

步骤1的解法：首先执行 epoll_create 在内核专属于 epoll 的高速 cache 区，并在该缓冲区建立**红黑树和就绪链表**，用户态传入的**文件句柄将被放到红黑树中**。

步骤2的解法：这一步与上述相同。

步骤3的解法：epoll_ctl 执行 add 动作时除了将文件句柄放到红黑树上之外，还**向内核注册了该文件句柄的回调函数，内核在检测到某句柄可读可写时则调用该回调函数，回调函数将文件句柄放到就绪链表，然后触发相应条件，唤醒等待队列中正在等待该条件的线程或进程，此处唤醒了 epoll。**

步骤4的解法：epoll_wait 只监控就绪链表就可以，如果就绪链表有文件句柄，则表示该文件句柄可读可写，并返回到用户态（**少量的拷贝**），注意这里仅仅**返回了文件描述符**，对应的文件数据（或IO数据）仍然在内核缓冲区中或文件中，采用共享内存或 mmap 文件内存映射加速。

步骤5的解法：由于原先的文件句柄没有被修改或者移动，因此无须重复步骤一，内核可以继续监控这些文件句柄，直到使用epoll_ctl删除文件句柄，否则不需要重新传入，**无须多次拷贝**。

简单说：epoll 是继承了select/poll 的I/O复用的思想，并在二者的基础上从监控IO流、查找I/O事件等角度来提高效率，具体地说就是内核句柄列表、红黑树、就绪链表来实现的。

> 文件句柄就是文件描述符。

第二部分：epoll详解

epoll 系统调用

先简单回顾下如何使用C库封装的3个epoll系统调用吧。

1. **int** epoll_create(**int** size);
2. **int** epoll_ctl(**int** epfd, **int** op, **int** fd, **struct** epoll_event *event);
3. **int** epoll_wait(**int** epfd, **struct** epoll_event *events, **int** maxevents, **int** timeout);

使用起来很清晰：

1. epoll_create 建立一个epoll对象，初始时红黑树和就绪链表都是空的。参数size是内核保证能够正确处理的最大句柄数，多于这个最大数时内核不保证效果。
2. epoll_ctl 可以操作上面建立的epoll，例如，将刚建立的 socket 加入到 epoll 中让其监控，或者把 epoll 正在监控的某个 socket 句柄移出 epoll，不再监控它等等(**也就是将I/O流放到内核**)。
3. epoll_wait在调用时，在给定的timeout时间内，当在监控的所有句柄中有事件发生时，就返回文件可读写的的总数给进程，同时设置这些事件到参数 events 中，**不需要用户去遍历查询**（**也就是在内核层面捕获可读写的I/O事件**）。

从上面的调用方式就可以看到 epoll 比 select/poll 的优越之处：

因为 select/poll 每次调用时都要传递你所要监控的所有 socket 给 select/poll 系统调用，这意味着需要将用户态的 socket 列表 copy 到内核态，如果以万计的句柄会导致每次都要 copy 几十几百KB的内存到内核态，非常低效。而我们调用 epoll_wait 时就相当于以往调用 select/poll，但是这时却不用传递socket句柄给内核，因为内核已经在epoll_ctl中拿到了要监控的句柄列表。

> select监控的句柄列表在用户态，每次调用都需要从用户态将句柄列表拷贝到内核态，但是epoll中句柄就是建立在内核中的，这样就减少了内核和用户态的拷贝，高效的原因之一。

所以，实际上在你调用epoll_create后，内核就已经在内核态开始准备帮你存储要监控的句柄了，每次调用epoll_ctl只是在往内核的**数据结构**里塞入新的socket句柄。

在内核里，一切皆文件。所以，epoll向内核注册了一个文件系统，用于存储上述的被监控socket。当你调用epoll_create时，就会在这个虚拟的epoll文件系统里创建一个file结点。当然这个file不是普通文件，它只服务于epoll。

epoll在被内核初始化时（**操作系统启动**），**同时会开辟出epoll自己的内核高速cache区，用于安置每一个我们想监控的socket，这些socket会以红黑树的形式保存在内核cache里，以支持快速的查找、插入、删除**。这个内核高速cache区，就是建立连续的物理内存页，然后在之上建立slab层，简单的说，**就是物理上提前分配好你想要的size的内存对象，每次使用时都是使用空闲的已分配好的对象。**

> lab是Linux操作系统的一种内存分配机制。其工作是针对一些经常分配并释放的对象，如进程描述符等，这些对象的大小一般比较小，如果直接采用伙伴系统来进行分配和释放，不仅会造成大量的内存碎片，而且处理速度也太慢。而slab分配器是基于对象进行管理的，相同类型的对象归为一类(如进程描述符就是一类)，每当要申请这样一个对象，slab分配器就从一个slab列表中分配一个这样大小的单元出去，而当要释放时，将其重新保存在该列表中，而不是直接返回给伙伴系统，从而避免这些内碎片。slab分配器并不丢弃已分配的对象，而是释放并把它们保存在内存中。当以后又要请求新的对象时，就可以从内存直接获取而不用重复初始化。

epoll高效的原因

这是由于**我们在调用epoll_create时，内核除了帮我们在epoll文件系统里建了个file结点，在内核cache里建了个红黑树用于存储以后epoll_ctl传来的socket外，还会再建立一个list链表，用于存储准备就绪的事件.**

当**epoll_wait调用时，仅仅观察这个list链表里有没有数据即可。有数据就返回，没有数据就sleep，等到timeout时间到后即使链表没数据也返回。所以，epoll_wait 非常高效**。而且，通常情况下即使我们要监控百万计的句柄，**大多一次也只返回很少量的准备就绪句柄而**已，所以，epoll_wait 仅需要**从内核态copy少量的句柄到用户态而已。**

那么，这个准备就绪list链表是怎么维护的呢？

当我们执行epoll_ctl时，除了把socket放到epoll文件系统里file对象对应的红黑树上之外，还会给内核中断处理程序注册一个回调函数，告诉内核，如果这个句柄的中断到了，就把它放到准备就绪list链表里。所以，当一个socket上有数据到了，内核在把网卡上的数据copy到内核中后就来把socket插入到准备就绪链表里了，然后 epoll 被唤醒，返回一个可读写的文件总数。

epoll综合的执行过程

如此，一棵红黑树，一张准备就绪句柄链表，少量的内核cache，就帮我们解决了大并发下的socket处理问题。执行epoll_create时，创建了红黑树和就绪链表，执行epoll_ctl时，如果增加socket句柄，则检查在红黑树中是否存在，存在立即返回，不存在则添加到树干上，然后向内核注册回调函数，用于当中断事件来临时向准备就绪链表中插入数据。执行epoll_wait时立刻返回准备就绪链表里的数据即可。

epoll 水平触发（LT）和边缘触发（ET）的实现

当一个socket句柄上有事件时，内核会把该句柄插入上面所说的准备就绪list链表，这时我们调用epoll_wait，会把准备就绪的socket拷贝到用户态内存，然后清空准备就绪list链表， 最后，epoll_wait干了件事，就是检查这些socket，如果不是ET模式（就是LT模式的句柄了），并且这些socket上确实有未处理的事件时，又把该句柄放回到刚刚清空的准备就绪链表了，所以，非ET的句柄，只要它上面还有事件，epoll_wait每次都会返回。而ET模式的句柄，除非有新中断到，即使socket上的事件没有处理完，也是不会次次从epoll_wait返回的。

LT 就是只要文件描述符关联的读内核缓冲区非空，有数据可以读取，就一直发出可读信号进行通知，当文件描述符关联的内核写缓冲区不满，有空间可以写入，就一直发出可写信号进行通知。LT模式支持阻塞和非阻塞两种方式。epoll默认的模式是LT。

ET 就是当文件描述符关联的读内核缓冲区由空转化为非空的时候，则发出可读信号进行通知；当文件描述符关联的内核写缓冲区由满转化为不满的时候，则发出可写信号进行通知。

两者的区别在哪里呢？水平触发是只要读缓冲区有数据，就会一直触发可读信号，而边缘触发仅仅在**空变为非空**的时候通知一次，

LT(level triggered)是缺省的工作方式，并且同时支持block和no-block socket.在这种做法中，内核告诉你一个文件描述符是否就绪了，然后你可以对这个就绪的fd进行IO操作。**如果你不作任何操作，内核还是会继续通知你的，所以，这种模式编程出错误可能性要小一点。**传统的select/poll都是这种模型的代表。

> 当你必须要处理某个 IO 事件时，使用 LT 更加安全，但相对而言更耗资源。
>
> 当你会主动去查缓冲区是否有数据时，换句话说，当你收到一次通知后，你自己就会变得主动起来，而无需内核次次通知。

> 举个生活中的例子，如果你有 2000 元想去买东西，例如你想买 10 个价值 200 元的礼物，而你父母就是监控你的人：
>
> LT 的做法是：每买一次东西就向你父母汇报，一共汇报十次。
>
> ET 的做法是：第一次买的时候汇报一下，剩余 9 次就不再通知你父母了，当你第一次汇报时，你的父母就应该要知道你可能会接着买东西。什么时候再次向你父母汇报呢？直到钱花光了，再次向你父母要钱，重新开始买东西，此时又会向你父母汇报一次。

第三部分：epoll高效的本质

1. 通过红黑树维护各个文件句柄，维护句柄十分高效。

2. 通过双向链表维护就绪文件句柄。

3. 事件发生，不用采用轮询机制依次查询，直接采用回调机制。

4. 每次事件完成，用户不需要再次把文件句柄拷贝到内核。epoll只需第一次拷贝。后面就不需要了。每次新加入的文件句柄都会通过红黑树插入进去。

5. 当要检测的文件句柄数量很大的时候。轮询机制耗时太严重，而epoll就高效的回调效率就体现出来了

NIO

NIO（Non-blocking I/O，在Java领域，也称为New I/O），是一种同步非阻塞的I/O模型，也是I/O多路复用的基础，已经被越来越多地应用到大型应用服务器，成为解决高并发与大量连接、I/O处理问题的有效方式。

那么NIO的本质是什么样的呢？它是怎样与事件模型结合来解放线程、提高系统吞吐的呢？

本文会从传统的阻塞I/O和线程池模型面临的问题讲起，然后对比几种常见I/O模型，一步步分析NIO怎么利用事件模型处理I/O，解决线程池瓶颈处理海量连接，包括利用面向事件的方式编写服务端/客户端程序。最后延展到一些高级主题，如Reactor与Proactor模型的对比、Selector的唤醒、Buffer的选择等。

注：本文的代码都是伪代码，主要是为了示意，不可用于生产环境。

传统BIO模型分析

> 本节出处Java NIO浅析 - 知乎 (zhihu.com)

让我们先回忆一下传统的服务器端同步阻塞I/O处理（也就是BIO，Blocking I/O）的经典编程模型：

```
{
 ExecutorService executor = Excutors.newFixedThreadPollExecutor(100);//线
程池

 ServerSocket serverSocket = new ServerSocket();
 serverSocket.bind(8088);
 while(!Thread.currentThread.isInturrupted()){//主线程死循环等待新连接到来
     Socket socket = serverSocket.accept();
     executor.submit(new ConnectIOnHandler(socket));//为新的连接创建新的线程
 }

class ConnectIOnHandler extends Thread{
    private Socket socket;
    public ConnectIOnHandler(Socket socket){
       this.socket = socket;
    }
    public void run(){
      while(!Thread.currentThread.isInturrupted()&&!socket.isClosed()){死
循环处理读写事件
          String someThing = socket.read()....//读取数据
          if(someThing!=null){
             ......//处理数据
             socket.write()....//写数据
          }

      }
    }
}
```

这是一个经典的每连接每线程的模型，之所以使用多线程，主要原因在于socket.accept()、socket.read()、socket.write()三个主要函数都是同步阻塞的，当一个连接在处理I/O的时候，系统是阻塞的，如果是单线程的话必然就挂死在那里；但CPU是被释放出来的，开启多线程，就可以让CPU去处理更多的事情。其实这也是所有使用多线程的本质：

1. 利用多核。
2. 当I/O阻塞系统，但CPU空闲的时候，可以利用多线程使用CPU资源。

现在的多线程一般都使用线程池，可以让线程的创建和回收成本相对较低。在活动连接数不是特别高（小于单机1000）的情况下，这种模型是比较不错的，可以让每一个连接专注于自己的I/O并且编程模型简单，也不用过多考虑系统的过载、限流等问题。线程池本身就是一个天然的漏斗，可以缓冲一些系统处理不了的连接或请求。

不过，这个模型最本质的问题在于，严重依赖于线程。但线程是很"贵"的资源，主要表现在：

1. 线程的创建和销毁成本很高，在Linux这样的操作系统中，线程本质上就是一个进程。创建和销毁都是重量级的系统函数。
2. 线程本身占用较大内存，像Java的线程栈，一般至少分配512K～1M的空间，如果系统中的线程数过千，恐怕整个JVM的内存都会被吃掉一半。
3. 线程的切换成本是很高的。操作系统发生线程切换的时候，需要保留线程的上下文，然后执行系统调用。如果线程数过高，可能执行线程切换的时间甚至会大于线程执行的时间，这时候带来的表现往往是系统load偏高、CPU sy使用率特别高（超过20%以上)，导致系统几乎陷入不可用的状态。
4. 容易造成锯齿状的系统负载。因为系统负载是用活动线程数或CPU核心数，一旦线程数量高但外部网络环境不是很稳定，就很容易造成大量请求的结果同时返回，激活大量阻塞线程从而使系统负载压力过大。

所以，当面对十万甚至百万级连接的时候，传统的BIO模型是无能为力的。随着移动端应用的兴起和各种网络游戏的盛行，百万级长连接日趋普遍，此时，必然需要一种更高效的I/O处理模型。

NIO是怎么工作的

很多刚接触NIO的人，第一眼看到的就是Java相对晦涩的API，比如：Channel，Selector，Socket什么的；然后就是一坨上百行的代码来演示NIO的服务端Demo……瞬间头大有没有？

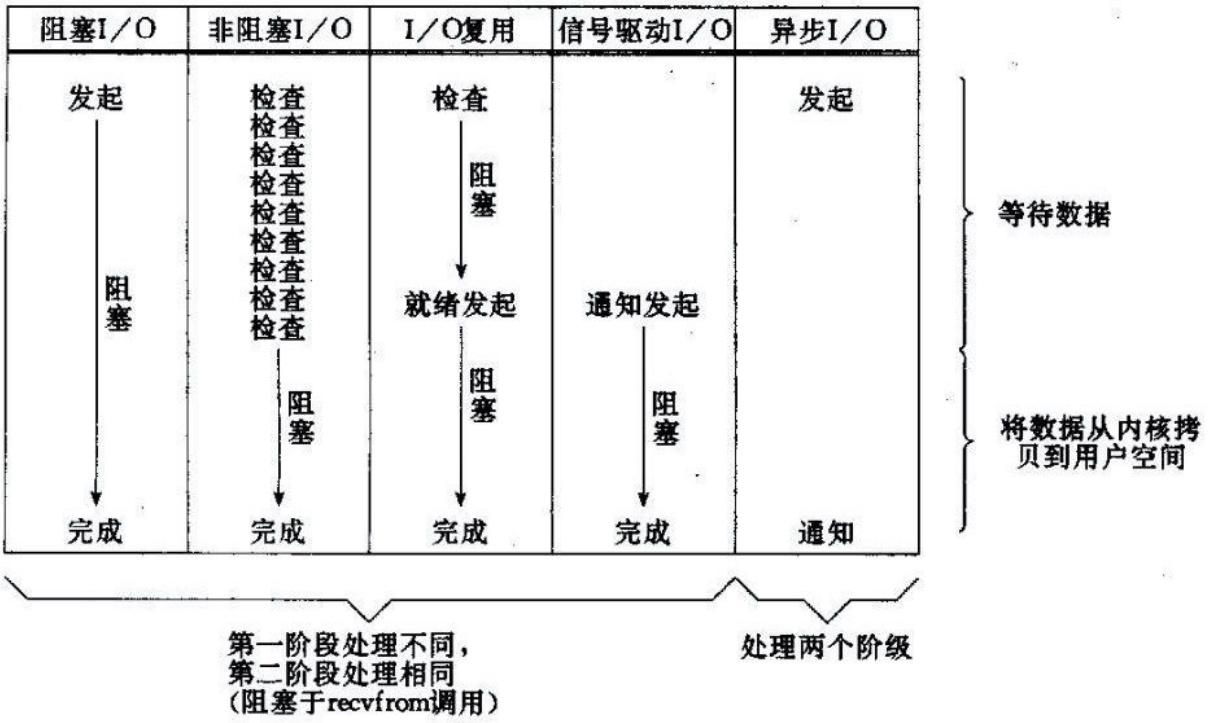
我们不管这些，抛开现象看本质，先分析下NIO是怎么工作的。

常见I/O模型对比

所有的系统I/O都分为两个阶段：等待就绪和操作。举例来说，读函数，分为等待系统可读和真正的读；同理，写函数分为等待网卡可以写和真正的写。

需要说明的是等待就绪的阻塞是不使用CPU的，是在“空等”；而真正的读写操作的阻塞是使用CPU的，真正在"干活"，而且这个过程非常快，属于memory copy，带宽通常在1GB/s级别以上，可以理解为基本不耗时。

下图是几种常见I/O模型的对比：



以socket.read()为例子：

传统的BIO里面socket.read()，如果TCP RecvBuffer里没有数据，函数会一直阻塞，直到收到数据，返回读到的数据。

对于NIO，如果TCP RecvBuffer有数据，就把数据从网卡读到内存，并且返回给用户；反之则直接返回0，永远不会阻塞。

最新的AIO(Async I/O)里面会更进一步：不但等待就绪是非阻塞的，就连数据从网卡到内存的过程也是异步的。

换句话说，BIO里用户最关心“我要读”，NIO里用户最关心"我可以读了"，在AIO模型里用户更需要关注的是“读完了”。

NIO一个重要的特点是：socket主要的读、写、注册和接收函数，在等待就绪阶段都是非阻塞的，真正的I/O操作是同步阻塞的（消耗CPU但性能非常高）。

如何结合事件模型使用NIO同步非阻塞特性

回忆BIO模型，之所以需要多线程，是因为在进行I/O操作的时候，一是没有办法知道到底能不能写、能不能读，只能"傻等"，即使通过各种估算，算出来操作系统没有能力进行读写，也没法在socket.read()和socket.write()函数中返回，这两个函数无法进行有效的中断。所以除了多开线程另起炉灶，没有好的办法利用CPU。

NIO的读写函数可以立刻返回，这就给了我们不开线程利用CPU的最好机会：如果一个连接不能读写（socket.read() 返回0或者socket.write()返回0），我们可以把这件事记下来，**记录的方式通常是在Selector上注册标记位，然后切换到其它就绪的连接（channel）继续进行读写。**

下面具体看下如何利用事件模型单线程处理所有I/O请求：

NIO的主要事件有几个：读就绪、写就绪、有新连接到来。

我们首先需要注册当这几个事件到来的时候所对应的处理器。然后在合适的时机告诉事件选择器：我对这个事件感兴趣。对于写操作，就是写不出去的时候对写事件感兴趣；对于读操作，就是完成连接和系统没有办法承载新读入的数据的时；对于accept，一般是服务器刚启动的时候；而对于connect，一般是connect失败需要重连或者直接异步调用connect的时候。

其次，用一个死循环选择就绪的事件，会执行系统调用（Linux 2.6之前是select、poll，2.6之后是epoll，Windows是IOCP），还会阻塞的等待新事件的到来。新事件到来的时候，会在selector上注册标记位，标示可读、可写或者有连接到来。

注意，select是阻塞的，无论是通过操作系统的通知（epoll）还是不停的轮询(select，poll)，这个函数是阻塞的。所以你可以放心大胆地在一个while(true)里面调用这个函数而不用担心CPU空转。

所以我们的程序大概的模样是：

```
   interface ChannelHandler{
      void channelReadable(Channel channel);
      void channelWritable(Channel channel);
   }
   class Channel{
     Socket socket;
     Event event;//读，写或者连接
   }

   //IO线程主循环:
   class IoThread extends Thread{
   public void run(){
   Channel channel;
   while(channel=Selector.select()){//选择就绪的事件和对应的连接
      if(channel.event==accept){
         registerNewChannelHandler(channel);//如果是新连接，则注册一个新的读写处理器
      }
      if(channel.event==write){
         getChannelHandler(channel).channelWritable(channel);//如果可以写，则执行写事件
      }
      if(channel.event==read){
          getChannelHandler(channel).channelReadable(channel);//如果可以读，则执行读事件
      }
    }
   }
   Map<Channel，ChannelHandler> handlerMap;//所有channel的对应事件处理器
  }
```

这个程序很简短，也是最简单的 Reactor 模式：**注册所有感兴趣的事件处理器，单线程轮询选择就绪事件，执行事件处理器。**

优化线程模型

由上面的示例我们大概可以总结出NIO是怎么解决掉线程的瓶颈并处理海量连接的：

NIO由原来的阻塞读写（占用线程）变成了单线程轮询事件，找到可以进行读写的网络描述符进行读写。除了事件的轮询是阻塞的（没有可干的事情必须要阻塞），剩余的I/O操作都是纯CPU操作，没有必要开启多线程。

并且由于线程的节约，连接数大的时候因为线程切换带来的问题也随之解决，进而为处理海量连接提供了可能。

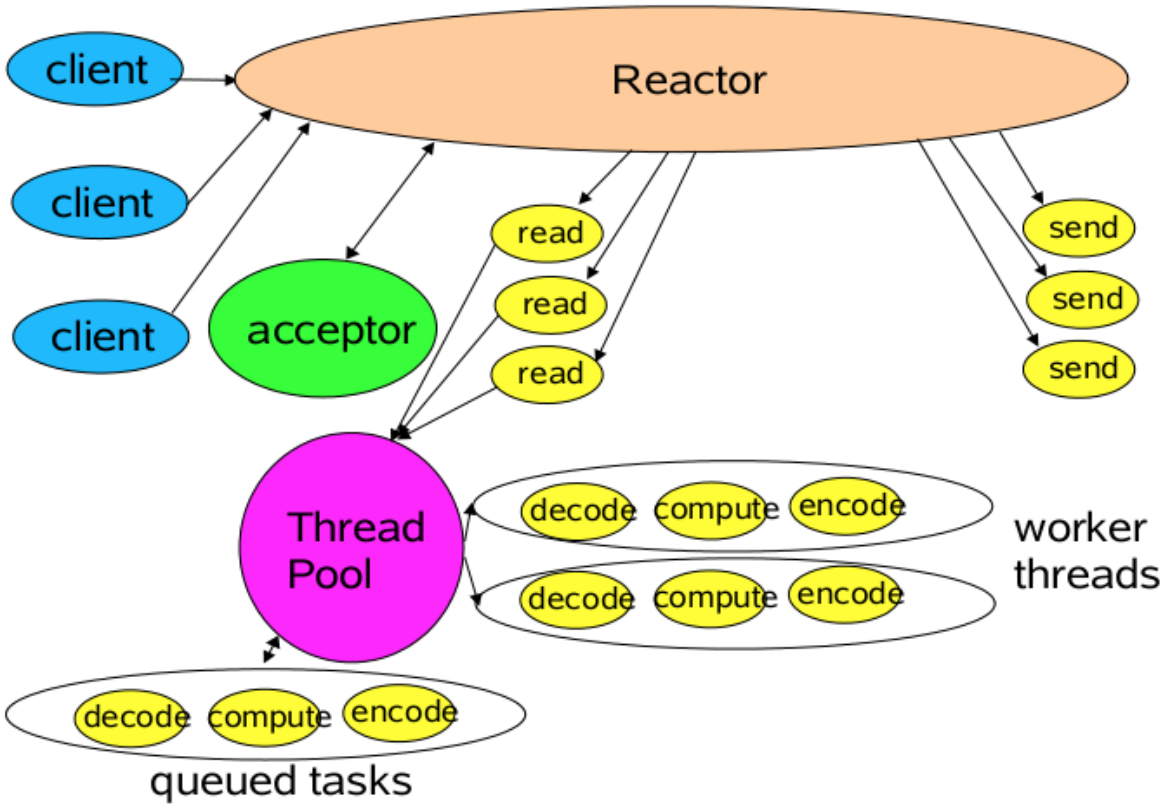
单线程处理I/O的效率确实非常高，没有线程切换，只是拼命的读、写、选择事件。但现在的服务器，一般都是多核处理器，如果能够利用多核心进行I/O，无疑对效率会有更大的提高。

仔细分析一下我们需要的线程，其实主要包括以下几种：

1. 事件分发器，单线程选择就绪的事件。
2. I/O处理器，包括connect、read、write等，这种纯CPU操作，一般开启CPU核心个线程就可以。
3. 业务线程，在处理完I/O后，业务一般还会有自己的业务逻辑，有的还会有其他的阻塞I/O，如DB操作，RPC等。只要有阻塞，就需要单独的线程。

Java的Selector对于Linux系统来说，有一个致命限制：**同一个channel的select不能被并发的调用**。因此，如果有多个I/O线程，必须保证：一个socket只能属于一个IoThread，而一个IoThread可以管理多个socket。

另外连接的处理和读写的处理通常可以选择分开，这样对于海量连接的注册和读写就可以分发。虽然read()和write()是比较高效无阻塞的函数，但毕竟会占用CPU，如果面对更高的并发则无能为力。



NIO在客户端的魔力

通过上面的分析，可以看出NIO在服务端对于解放线程，优化I/O和处理海量连接方面，确实有自己的用武之地。那么在客户端上，NIO又有什么使用场景呢?

常见的客户端BIO+连接池模型，可以建立n个连接，然后当某一个连接被I/O占用的时候，可以使用其他连接来提高性能。

但多线程的模型面临和服务端相同的问题：如果指望增加连接数来提高性能，则连接数又受制于线程数、线程很贵、无法建立很多线程，则性能遇到瓶颈。

每连接顺序请求的Redis

对于Redis来说，由于**服务端是全局串行的**，能够保证同一连接的所有请求与返回顺序一致。这样可以使用单线程＋队列，把请求数据缓冲。然后pipeline发送，返回future，然后channel可读时，直接在队列中把future取回来，done()就可以了。

伪代码如下：

```
class RedisClient Implements ChannelHandler{
 private BlockingQueue CmdQueue;
 private EventLoop eventLoop;
 private Channel channel;
 class Cmd{
  String cmd;
  Future result;
```

```
 }
 public Future get(String key){
   Cmd cmd= new Cmd(key);
   queue.offer(cmd);
   eventLoop.submit(new Runnable(){
        List list = new ArrayList();
        queue.drainTo(list);
        if(channel.isWritable()){
         channel.writeAndFlush(list);
        }
   });
}
 public void ChannelReadFinish(Channel channel，Buffer Buffer){
    List result = handleBuffer();//处理数据
    //从cmdQueue取出future，并设值，future.done();
}
 public void ChannelWritable(Channel channel){
   channel.flush();
}
}
```

这样做，能够充分的利用pipeline来提高I/O能力，同时获取异步处理能力。

多连接短连接的HttpClient

类似于竞对抓取的项目，往往需要建立无数的HTTP短连接，然后抓取，然后销毁，当需要单机抓取上千网站线程数又受制的时候，怎么保证性能呢?

何不尝试NIO，单线程进行连接、写、读操作？如果连接、读、写操作系统没有能力处理，简单的注册一个事件，等待下次循环就好了。

如何存储不同的请求/响应呢？由于http是无状态没有版本的协议，又没有办法使用队列，好像办法不多。比较笨的办法是对于不同的socket，直接存储socket的引用作为map的key。

常见的RPC框架，如Thrift，Dubbo

这种框架内部一般维护了请求的协议和请求号，可以维护一个以请求号为key，结果的result为future的map，结合NIO+长连接，获取非常不错的性能。

NIO高级主题

Proactor与Reactor

一般情况下，I/O 复用机制需要事件分发器（event dispatcher）。 事件分发器的作用，即将那些读写事件源分发给各读写事件的处理者，就像送快递的在楼下喊: 谁谁谁的快递到了， 快来拿吧！开发人员在开始的时候需要在分发器那里注册感兴趣的事件，并提

供相应的处理者（event handler)，或者是回调函数；事件分发器在适当的时候，会将请求的事件分发给这些handler或者回调函数。

涉及到事件分发器的两种模式称为：Reactor和Proactor。 **Reactor模式是基于同步I/O的，而Proactor模式是和异步I/O相关的**。在Reactor模式中，事件分发器等待某个事件或者可应用或个操作的状态发生（比如文件描述符可读写，或者是socket可读写），事件分发器就把这个事件传给事先注册的事件处理函数或者回调函数，由后者来做实际的读写操作。

而在Proactor模式中，事件处理者（或者代由事件分发器发起）直接发起一个异步读写操作（相当于请求），而实际的读写工作是由操作系统来完成的。发起时，需要提供的参数包括用于存放读到数据的缓存区、读的数据大小或用于存放外发数据的缓存区，以及这个请求完后的回调函数等信息。事件分发器得知了这个请求，它默默等待这个请求的完成，然后转发完成事件给相应的事件处理者或者回调。举例来说，在Windows上事件处理者投递了一个异步IO操作（称为overlapped技术），事件分发器等IO Complete事件完成。这种异步模式的典型实现是基于操作系统底层异步API的，所以我们可称之为“系统级别”的或者“真正意义上”的异步，因为具体的读写是由操作系统代劳的。

> 同步 IO 就是你自己下楼拿外卖，而异步 IO 就是外卖小哥送到你家，可以直接吃了！

举个例子，将有助于理解Reactor与Proactor二者的差异，以读操作为例（写操作类似）。

在Reactor中实现读

- 注册读就绪事件和相应的事件处理器。
- 事件分发器等待事件。
- 事件到来，激活分发器，分发器调用事件对应的处理器。
- **事件处理器完成实际的读操作**，处理读到的数据，注册新的事件，然后返还控制权。

在Proactor中实现读

- 处理器发起异步读操作（注意：操作系统必须支持异步IO）。在这种情况下，处理器无视IO就绪事件，它关注的是完成事件。
- 事件分发器等待操作完成事件。
- 在分发器等待过程中，操作系统利用并行的内核线程执行实际的读操作，并将结果数据存入用户自定义缓冲区，最后通知事件分发器读操作完成。
- 事件分发器呼唤处理器。
- 事件处理器处理用户自定义缓冲区中的数据，然后启动一个新的异步操作，并将控制权返回事件分发器。

可以看出，两个模式的相同点，都是对某个I/O事件的事件通知（即告诉某个模块，这个I/O操作可以进行或已经完成)。在结构上，两者也有相同点：事件分发器负责提交IO操作（异步)、查询设备是否可操作（同步)，然后当条件满足时，就回调handler；不同点在

于，异步情况下（Proactor)，当回调handler时，表示I/O操作已经完成；同步情况下（Reactor)，回调handler时，表示I/O设备可以进行某个操作（can read 或 can write)。

下面，我们将尝试应对为Proactor和Reactor模式建立可移植框架的挑战。在改进方案中，我们将Reactor原来位于事件处理器内的Read/Write操作移至分发器（不妨将这个思路称为“模拟异步”），以此寻求将Reactor多路同步I/O转化为模拟异步I/O。以读操作为例子，改进过程如下：

- 注册读就绪事件和相应的事件处理器。并为分发器提供数据缓冲区地址，需要读取数据量等信息。
- 分发器等待事件（如在select()上等待）。
- 事件到来，激活分发器。分发器执行一个非阻塞读操作（它有完成这个操作所需的全部信息），最后调用对应处理器。
- 事件处理器处理用户自定义缓冲区的数据，注册新的事件（当然同样要给出数据缓冲区地址，需要读取的数据量等信息），最后将控制权返还分发器。 如我们所见，通过对多路I/O模式功能结构的改造，可将Reactor转化为Proactor模式。改造前后，模型实际完成的工作量没有增加，只不过参与者间对工作职责稍加调换。没有工作量的改变，自然不会造成性能的削弱。对如下各步骤的比较，可以证明工作量的恒定：

标准/典型的Reactor

- 步骤1：等待事件到来（Reactor负责）。
- 步骤2：将读就绪事件分发给用户定义的处理器（Reactor负责）。
- 步骤3：读数据（用户处理器负责）。
- 步骤4：处理数据（用户处理器负责）。

改进实现的模拟Proactor

- 步骤1：等待事件到来（Proactor负责）。
- 步骤2：得到读就绪事件，执行读数据（现在由Proactor负责）。
- 步骤3：将读完成事件分发给用户处理器（Proactor负责）。
- 步骤4：处理数据（用户处理器负责）。

  对于不提供异步I/O API的操作系统来说，这种办法可以隐藏Socket API的交互细节，从而对外暴露一个完整的异步接口。借此，我们就可以进一步构建完全可移植的，平台无关的，有通用对外接口的解决方案。

代码示例如下：

```
interface ChannelHandler{
      void channelReadComplate(Channel channel，byte[] data);
      void channelWritable(Channel channel);
```

```
    }
    class Channel{
      Socket socket;
      Event event;//读，写或者连接
    }

    //IO线程主循环：
    class IoThread extends Thread{
    public void run(){
    Channel channel;
    while(channel=Selector.select()){//选择就绪的事件和对应的连接
       if(channel.event==accept){
          registerNewChannelHandler(channel);//如果是新连接，则注册一个新的读
写处理器
          Selector.interested(read);
       }
       if(channel.event==write){
          getChannelHandler(channel).channelWritable(channel);//如果可以
写，则执行写事件
       }
       if(channel.event==read){
           byte[] data = channel.read();
           if(channel.read()==0)//没有读到数据，表示本次数据读完了
           {
                 getChannelHandler(channel).channelReadComplate(channel，
data;//处理读完成事件
           }
           if(过载保护){
           Selector.interested(read);
           }

       }
      }
     }
    Map<Channel，ChannelHandler> handlerMap;//所有channel的对应事件处理器
    }
```

Selector.wakeup()

主要作用

解除阻塞在Selector.select()/select(long)上的线程，立即返回。

两次成功的select之间多次调用wakeup等价于一次调用。

如果当前没有阻塞在select上，则本次wakeup调用将作用于下一次select——“记忆”作用。

为什么要唤醒？

- 注册了新的channel或者事件。

- channel关闭，取消注册。
- 优先级更高的事件触发（如定时器事件），希望及时处理。

原理

Linux上利用pipe调用创建一个管道，Windows上则是一个loopback的tcp连接。这是因为win32的管道无法加入select的fd set，将管道或者TCP连接加入select fd set。

wakeup往管道或者连接写入一个字节，阻塞的select因为有I/O事件就绪，立即返回。可见，wakeup的调用开销不可忽视。

Buffer的选择

通常情况下，操作系统的一次写操作分为两步：

1. 将数据从用户空间拷贝到系统空间。
2. **从系统空间往网卡写（同步 IO 与异步 IO 的区别发送在这里**。

同理，读操作也分为两步：

1. **将数据从网卡拷贝到系统空间（同步 IO 与异步 IO 的区别发送在这里）。**
2. 将数据从系统空间拷贝到用户空间。

对于NIO来说，缓存的使用可以使用 DirectByteBuffer 和 HeapByteBuffer。如果使用了DirectByteBuffer，一般来说可以减少一次系统空间到用户空间的拷贝。但Buffer创建和销毁的成本更高，更不宜维护，通常会用内存池来提高性能。

如果数据量比较小的中小应用情况下，可以考虑使用heapBuffer；反之可以用directBuffer。

NIO存在的问题

使用NIO != 高性能，当连接数<1000，并发程度不高或者局域网环境下NIO并没有显著的性能优势。

NIO并没有完全屏蔽平台差异，它仍然是基于各个操作系统的I/O系统实现的，差异仍然存在。使用NIO做网络编程构建事件驱动模型并不容易，陷阱重重。

推荐大家使用成熟的NIO框架，如Netty，MINA等。解决了很多NIO的陷阱，并屏蔽了操作系统的差异，有较好的性能和编程模型。

总结

最后总结一下到底NIO给我们带来了些什么：

- 事件驱动模型
- 避免多线程
- 单线程处理多任务
- 非阻塞I/O，I/O读写不再阻塞，而是返回0
- 基于block的传输，通常比基于流的传输更高效
- 更高级的IO函数，zero-copy
- IO多路复用大大提高了Java网络应用的可伸缩性和实用性