高性能I/O设计模式概述

Yeolar 2012-12-15 22:26

这篇概述把网上的一些资料汇总到了一起，主要目的是解释清楚I/O模型的概念和Reactor、Proactor模式。

目录

服务器的几种实现方法

服务器一般需要支持高性能的I/O，大并发等。

第一种实现一个服务器的想当然的方法是，当有请求过来时，就 `fork` 一个子进程进行处理。这种方法易于理解，实现简单，如果并发量比较小，应该也能应付。但是如果是高并发的服务器就不理想了，因为 `fork` 大量的进程，并在不同的进程之间切换会消耗大量资源，造成服务器运行的效率下降。

第二种实现服务器的方法是所谓的preforking，也就是预先创建“足够的”进程，每个进程通过 `accept` 阻塞在被侦听的端口上。也就是所谓的线程池的方法。这种方法中，多个线程阻塞在同一个侦听端口上 `accept` ，存在所谓的“惊群”问题，因此 `select` （ `epoll` 、 `kqueue` 、 `iocp` ）就成了服务器编程的想当然的选择。

一些概念

同步和异步

同步和异步是针对应用程序和内核的交互而言的，同步指的是用户进程触发I/O操作并等待或者轮询的去查看I/O操作是否就绪，而异步是指用户进程触发I/O操作以后便开始做自己的事情，而当I/O操作已经完成的时候会得到I/O完成的通知。

阻塞和非阻塞

阻塞和非阻塞是针对于进程在访问数据的时候，根据I/O操作的就绪状态来采取的不同方式，说白了是一种读取或者写入操作函数的实现方式，阻塞方式下读取或者写入函数将一直等待，而非阻塞方式下，读取或者写入函数会立即返回一个状态值。

I/O模型

一般来说I/O模型可以分为：

同步阻塞I/O

在此种方式下，用户进程在发起一个I/O操作以后，必须等待I/O操作的完成，只有当真正完成了I/O操作以后，用户进程才能运行。Java传统的I/O模型属于此种方式。

同步非阻塞I/O
在此种方式下，用户进程发起一个I/O操作以后边可返回做其它事情，但是用户进程需要时不时的询问I/O操作是否就绪，这就要求用户进程不停的去询问，从而引入不必要的CPU资源浪费。目前Java的NIO就属于同步非阻塞I/O。

异步阻塞I/O
此种方式下是指应用发起一个I/O操作以后，不等待内核I/O操作的完成，等内核完成I/O操作以后会通知应用程序，这其实就是同步和异步最关键的区别，同步必须等待或者主动的去询问I/O是否完成，那么为什么说是阻塞的呢？因为此时是通过 `select` 系统调用来完成的，而 `select` 函数本身的实现方式是阻塞的，而采用 `select` 函数有个好处就是它可以同时监听多个文件句柄，从而提高系统的并发性。

异步非阻塞I/O
在此种模式下，用户进程只需要发起一个I/O操作然后立即返回，等I/O操作真正的完成以后，应用程序会得到I/O操作完成的通知，此时用户进程只需要对数据进行处理就好了，不需要进行实际的I/O读写操作，因为真正的I/O读取或者写入操作已经由内核完成了。目前Java中还没有支持此种I/O模型。

Reactor和Proactor模式

在高性能的I/O设计中，有两个比较著名的模式，Reactor和Proactor模式，其中Reactor模式用于同步I/O，而Proactor运用于异步I/O操作。

首先来看看Reactor模式，Reactor模式应用于同步I/O的场景。

Reactor设计模式中的要素如下：

- `Handles`： 也就是网络连接、文件句柄等，是事件源。
- `Synchronous Event Demultiplexer`： 同步事件的解复用（或者说派发），具体的比如 `select` 调用。比 `select` 更加高效的有Linux下的 `epoll` ， FreeBSD下的 `kqueue` 以及Windows下的 `iocp` （IO Completion port）。
- `Initiation Dispatcher`： 注册、移除和分派事件处理器。
- `Event Handler`： 事件处理器。

我们分别以读操作和写操作为例来看看Reactor中的具体步骤：

读取操作：

1. 应用程序注册读取事件和相关联的事件处理器。
2. 事件分离器等待事件的发生。
3. 当发生读取事件的时候，事件分离器调用第一步注册的事件处理器。
4. 事件处理器首先执行实际的读取操作，然后根据读取到的内容进行进一步的处理。

写入操作类似于读取操作，只不过第一步注册的是写就绪事件。

下面我们来看看Proactor模式中读取操作和写入操作的过程：

读取操作：

1. 应用程序初始化一个异步读取操作，然后注册相应的事件处理器，此时事件处理器不关注读取就绪事

件，而是关注读取完成事件，这是区别于Reactor的关键。
2. 事件分离器等待读取操作完成事件。
3. 在事件分离器等待读取操作完成的时候，操作系统调用内核线程完成读取操作，并将读取的内容放入用户传递过来的缓存区中。这也是区别于Reactor的一点，Proactor中，应用程序需要传递缓存区。
4. 事件分离器捕获到读取完成事件后，激活应用程序注册的事件处理器，事件处理器直接从缓存区读取数据，而不需要进行实际的读取操作。

Proactor中写入操作和读取操作，只不过感兴趣的事件是写入完成事件。

从上面可以看出，Reactor和Proactor模式的主要区别就是真正的读取和写入操作是有谁来完成的，Reactor中需要应用程序自己读取或者写入数据，而Proactor模式中，应用程序不需要进行实际的读写过程，它只需要从缓存区读取或者写入即可，操作系统会读取缓存区或者写入缓存区到真正的I/O设备。

Reactor通过某种变形，可以将其改装为Proactor，在某些不支持异步I/O的系统上，也可以隐藏底层的实现，利于编写跨平台代码。我们只需要在dispatch（也就是Demultiplexer）中封装同步I/O操作的代码，在上层，用户提交自己的缓冲区到这一层，这一层检查到设备可操作时，不像原来立即回调Handler，而是开始I/O操作，然后将操作结果放到用户缓冲区（读），然后再回调Handler。这样，对于上层Handler而言，就像是proactor一样。详细技法参见 Comparing Two High-Performance I/O Design Patterns (/note/2012/12/15/io-design-patterns/) 。

综上所述，同步和异步是相对于应用和内核的交互方式而言的，同步需要主动去询问，而异步的时候内核在I/O事件发生的时候通知应用程序，而阻塞和非阻塞仅仅是系统在调用系统调用的时候函数的实现方式而已。

半同步/半异步和领导者/追随者设计模式

在《Pattern-Oriented Software Architecture, Volume 2: Patterns for Concurrent and Networked Objects》一书中，关于这两个模式有两个很形象的比喻：

半同步/半异步（half-sync/half-async）
许多餐厅使用 半同步/半异步 模式的变体。例如，餐厅常常雇佣一个领班负责迎接顾客，并在餐厅繁忙时留意给顾客安排桌位，为等待就餐的顾客按序排队是必要的。领班由所有顾客“共享”，不能被任何特定顾客占用太多时间。当顾客在一张桌子入坐后，有一个侍应生专门为这张桌子服务。

领导者/追随者（Leader/Followers）
在日常生活中，领导者/追随者模式用于管理许多飞机场出租车候车台。在该用例中，出租车扮演“线程”角色，排在第一辆的出租车成为领导者，剩下的出租车成为追随者。同样，到达出租车候车台的乘客构成了必须被多路分解给出租车的事件，一般以先进先出排序。一般来说，如果任何出租车可以为任何顾客服务，该场景就主要相当于非绑定句柄/线程关联。然而，如果仅仅是某些出租车可以为某些乘客服务，该场景就相当于绑定句柄/线程关联。

在《POSA2》书中列举的例子都比较复杂，并且书上没有列出完整的代码。但是这两个模式其实都可以在《unix网络编程》一书中找到对应的完整的代码和相关的讨论。

在半同步/半异步模式中，需要由模式实现者显示构造一个队列，以便同步层和异步层可以通信。

在《unix网络编程》一书的“27.12 TCP预先创建线程服务器程序，主线程统一 accept ”的例子中，如果只是从处理 accept 这个事件上看，可以认为这是一个使用了半同步/半异步模式的例子。但是从具体的业务处理（即 web_child 的处理上），仍然可以认为是一个ThreadPerConnection模型，因为在 thread_main 中直接读取请求和发送响应。在这个例子中，就有一个队列：

```
//这就是一个典型的循环队列的定义，iget 是队列头，iput 是队列尾
int clifd[MAXNCLI], iget, iput;

int main( int argc, char * argv[] )
{
  ......
  int listenfd = Tcp_listen( NULL, argv[ 1 ], &addrlen );
  ......

  iget = iput = 0;

  for( int i = 0; i < nthreads; i++ ) {
    pthread_create( &tptr[i].thread_tid, NULL, &thread_main, (void*)i );

  for( ; ; ) {
    connfd = accept( listenfd, cliaddr,, &clilen );
    clifd[ iput ] = connfd;     // 接受到的连接句柄放入队列
    if( ++iput == MAXNCLI ) iput = 0;
  }
}

void * thread_main( void * arg )
{
  for( ; ; ) {
    while( iget == iput ) pthread_cond_wait( ..... );
    connfd = clifd[ iget ];     // 从队列中获得连接句柄
    if( ++iget == MAXNCLI ) iget = 0;
    ......
    web_child( connfd );
    close( connfd );
  }
}
```

而在领导者/追随者模式中，同样是有一个队列的，不过不需要模式实现者显示构造，而是直接使用了操作系统底层的队列。

在《unix网络编程》一书的 “27.11 TCP预先创建服务器线程，每个线程各自 accept ”的例子中，就是直接使用了操作系统中关于 accept 的队列。这个例子可以认为是领导者/追随者模式的一个例子。

```
int listenfd;

int main( int argc, char * argv[] )
{
  ......
  listenfd = Tcp_listen( NULL, argv[ 1 ], &addrlen );
  ......
  for( int i = 0; i < nthreads; i++ ){
    pthread_create( &tptr[i].thread_tid, NULL, &thread_main, (void*)i );
  }
  ......
}

void * thread_main( void * arg )
{
  for( ; ; ){
    ......
    // 多个线程同时阻塞在这个 accept 调用上，依靠操作系统的队列
    connfd = accept( listenfd, cliaddr, &clilen );
    ......
    web_child( connfd );
    close( connfd );
    ......
  }
}
```

当然，这里提到的操作系统的队列，在半同步/半异步模式中虽然没有明显地指出来，但只要是通过操作系统来做 `accept` ，那么在半同步/半异步模式中仍然会隐式地用到。

在《POSA2》中，作者评价：因为半同步/半异步设计在web服务器虚拟内存而不是操作系统内核内排队请求，所以它更具伸缩性。

参考资料

- http://xmuzyq.iteye.com/blog/783218 (http://xmuzyq.iteye.com/blog/783218)
- http://blog.sina.com.cn/s/blog_6a4c492f0100o6lr.html (http://blog.sina.com.cn/s/blog_6a4c492f0100o6lr.html)
- http://www.cppblog.com/kevinlynx/archive/2008/06/06/52356.html (http://www.cppblog.com/kevinlynx/archive/2008/06/06/52356.html)
- http://www.iteye.com/topic/60414 (http://www.iteye.com/topic/60414)

http://www.yeolar.com/note/2012/12/15/high-performance-io-design-patterns/