# IO 多路复用是什么意思？

<https://mp.weixin.qq.com/s/2KK9DcuYJyp0yOb4PHaS1A>

假设你是一个机场的空管， 你需要管理到你机场的所有的航线， 包括进港，出港， 有些航班需要放到停机坪等待，有些航班需要去登机口接乘客。

你会怎么做?

最简单的做法，就是你去招一大批空管员，然后每人盯一架飞机， 从进港，接客，排位，出港，航线监控，直至交接给下一个空港，全程监控。

那么问题就来了：

很快你就发现空管塔里面聚集起来一大票的空管员，交通稍微繁忙一点，新的空管员就已经挤不进来了。

空管员之间需要协调，屋子里面就1, 2个人的时候还好，几十号人以后 ，基本上就成菜市场了。

空管员经常需要更新一些公用的东西，比如起飞显示屏，比如下一个小时后的出港排期，最后你会很惊奇的发现，每个人的时间最后都花在了抢这些资源上。

现实上我们的空管同时管几十架飞机稀松平常的事情， 他们怎么做的呢？他们用这个东西：



这个东西叫 flight progress strip. 每一个块代表一个航班，不同的槽代表不同的状态，然后一个空管员可以管理一组这样的块（一组航班），而他的工作，就是在航班信息有新的更新的时候，把对应的块放到不同的槽子里面。

这个东西现在还没有淘汰哦，只是变成电子的了而已。

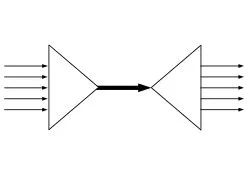
是不是觉得一下子效率高了很多，一个空管塔里可以调度的航线可以是前一种方法的几倍到几十倍。

如果你把每一个航线当成一个Sock(I/O 流), 空管当成你的服务端Sock管理代码的话.

第一种方法就是最传统的多进程并发模型 (每进来一个新的I/O流会分配一个新的进程管理。)

第二种方法就是I/O多路复用 (单个线程，通过记录跟踪每个I/O流(sock)的状态，来同时管理多个I/O流 。)

其实“I/O 多路复用”这个坑爹翻译可能是这个概念在中文里面如此难理解的原因。所谓的I/O多路复用在英文中其实叫 I/O multiplexing. 如果你搜索multiplexing啥意思，基本上都会出这个图：



于是大部分人都直接联想到"一根网线，多个sock复用" 这个概念，包括上面的几个回答， 其实不管你用多进程还是I/O多路复用， 网线都只有一根好伐。多个Sock复用一根网线这个功能是在内核＋驱动层实现的。

重要的事情再说一遍：**I/O multiplexing 这里面的 multiplexing 指的其实是在单个线程通过记录跟踪每一个Sock(I/O流)的状态(对应空管塔里面的Fight progress strip槽)来同时管理多个I/O流.**

发明它的原因，是尽量多的提高服务器的吞吐能力。

是不是听起来好拗口，看个图就懂了：



在同一个线程里面， 通过拨开关的方式，来同时传输多个I/O流， (学过EE的人现在可以站出来义正严辞说这个叫“时分复用”了）。

什么，你还没有搞懂“一个请求到来了，nginx使用epoll接收请求的过程是怎样的”， 多看看这个图就了解了。提醒下，**ngnix会有很多连接进来， epoll会把他们都监视起来，然后像拨开关一样，谁有数据就拨向谁，然后调用相应的代码处理。**

了解这个基本的概念以后，其他的就很好解释了。

select, poll, epoll 都是I/O多路复用的具体的实现，之所以有这三个鬼存在，其实是他们出现是有先后顺序的。

I/O多路复用这个概念被提出来以后， select是第一个实现 (1983 左右在BSD里面实现的)。

select 被实现以后，很快就暴露出了很多问题。

* select 会修改传入的参数数组，这个对于一个需要调用很多次的函数，是非常不友好的。
* select 如果任何一个sock(I/O stream)出现了数据，select 仅仅会返回，但是并不会告诉你是那个sock上有数据，于是你只能自己一个一个的找，10几个sock可能还好，要是几万的sock每次都找一遍，这个无谓的开销就颇有海天盛筵的豪气了。
* select 只能监视1024个链接， 这个跟草榴没啥关系哦，linux 定义在头文件中的，参见FD\_SETSIZE。
* select 不是线程安全的，如果你把一个sock加入到select, 然后突然另外一个线程发现，尼玛，这个sock不用，要收回。对不起，这个select 不支持的，如果你丧心病狂的竟然关掉这个sock, select的标准行为是。。呃。。不可预测的， 这个可是写在文档中的哦.

“If a file descriptor being monitored by select() is closed in another thread, the result is unspecified”

霸不霸气？

于是 14 年以后(1997 年）一帮人又实现了 poll, poll 修复了 select 的很多问题，比如

* poll 去掉了1024个链接的限制，于是要多少链接呢， 主人你开心就好。
* poll 从设计上来说，不再修改传入数组，不过这个要看你的平台了，所以行走江湖，还是小心为妙。

其实拖 14 年那么久也不是效率问题， 而是那个时代的硬件实在太弱，一台服务器处理1千多个链接简直就是神一样的存在了，select很长段时间已经满足需求。

但是poll仍然不是线程安全的， 这就意味着，不管服务器有多强悍，你也只能在一个线程里面处理一组 I/O 流。你当然可以拿多进程来配合了，不过然后你就有了多进程的各种问题。

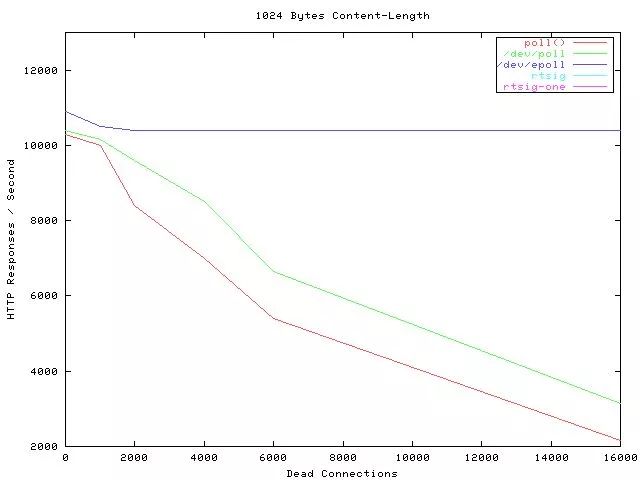
于是5年以后, 在2002, 大神 Davide Libenzi 实现了epoll，epoll 可以说是 I/O 多路复用最新的一个实现，epoll 修复了poll 和select绝大部分问题, 比如：

* epoll 现在是线程安全的。
* epoll 现在不仅告诉你sock组里面数据，还会告诉你具体哪个sock有数据，你不用自己去找了。

epoll 当年的 patch，现在还在，下面链接可以看得到：/dev/epoll Home Page

链接：http://www.xmailserver.org/linux-patches/nio-improve.html

贴一张霸气的图，看看当年神一样的性能（测试代码都是死链了， 如果有人可以刨坟找出来，可以研究下细节怎么测的).



横轴 Dead connections 就是链接数的意思，叫这个名字只是它的测试工具叫deadcon. 纵轴是每秒处理请求的数量，你可以看到，epoll每秒处理请求的数量基本不会随着链接变多而下降的。poll 和/dev/poll 就很惨了。

**可是epoll 有个致命的缺点。只有linux支持。比如BSD上面对应的实现是kqueue。**

其实有些国内知名厂商把epoll从安卓里面裁掉这种脑残的事情我会主动告诉你嘛。什么，你说没人用安卓做服务器，尼玛你是看不起p2p软件了啦。

而ngnix 的设计原则里面， 它会使用目标平台上面最高效的I/O多路复用模型，所以才会有这个设置。一般情况下，如果可能的话，尽量都用epoll/kqueue吧。

详细的在这里: Connection processing methods

链接：<http://nginx.org/en/docs/events.html>

PS: 上面所有这些比较分析，都建立在大并发下面，如果你的并发数太少，用哪个，其实都没有区别。如果像是在欧朋数据中心里面的转码服务器那种动不动就是几万几十万的并发，不用epoll我可以直接去撞墙了。