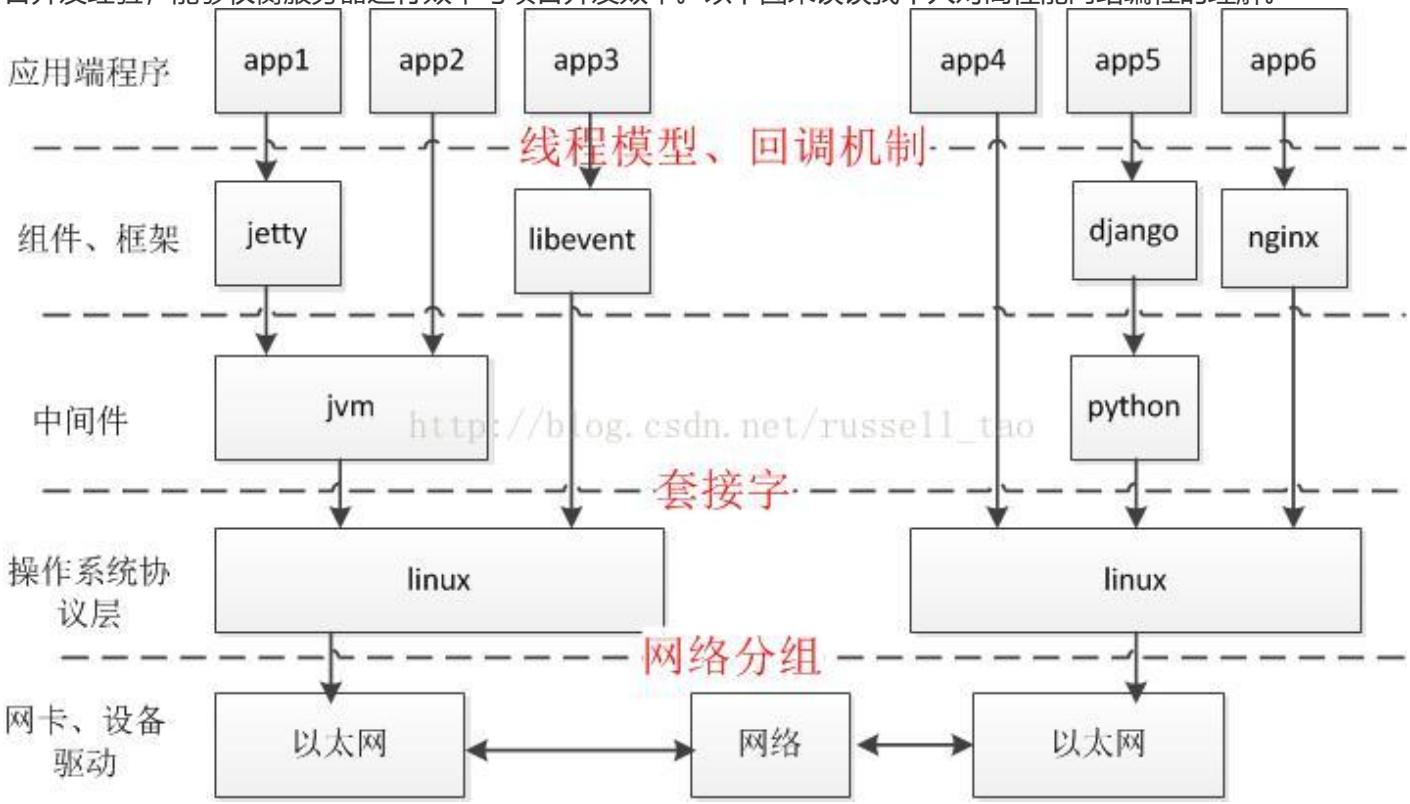
高性能网络编程（一）----accept建立连接

2013年06月24日 19:02:27 阅读数：44653

最近在部门内做了个高性能网络编程的培训，近日整理了下PPT，欲写成一系列文章从应用角度谈谈它。
编写服务器时，许多程序员习惯于使用高层次的组件、中间件（例如OO（面向对象）层层封装过的开源组件），相比于服务器的运行效率而言，他们更关注程序开发的效率，追求更快的完成项目功能点、希望应用代码完全不关心通讯细节。他们更喜欢在OO世界里，去实现某个接口、实现这个组件预定义的各种模式、设置组件参数来达到目的。学习复杂的通讯框架、底层细节，在习惯于使用OO语言的程序员眼里是绝对事倍功半的。以上做法无可厚非，但有一定的局限性，本文讲述的网络编程头前冠以“高性能”，它是指程序员设计编写的服务器需要处理很大的吞吐量，这与简单网络应用就有了质的不同。因为：1、高吞吐量下，容易触发到一些设计上的边界条件；2、偶然性的小概率事件，会在高吞吐量下变成必然性事件。3、IO是慢速的，高吞吐量通常意味着高并发，如同一时刻存在数以万计、十万计、百万计的TCP活动连接。所以，做高性能网络编程不能仅仅满足于学会开源组件、中间件是如何帮我实现期望功能的，对于企业级产品来说，需要了解更多的知识。

掌握高性能网络编程，涉及到对网络、操作系统协议栈、进程与线程、常见的网络组件等知识点，需要有丰富的项目开发经验，能够权衡服务器运行效率与项目开发效率。以下图来谈谈我个人对高性能网络编程的理解。



上面这张图中，由上至下有以下特点：

- •关注点，逐渐由特定业务向通用技术转移
- •使用场景上，由专业领域向通用领域转移
- •灵活性上要求越来越高
- •性能要求越来越高
- •对细节、原理的掌握，要求越来越高
- •对各种异常情况的处理，要求越来越高
- •稳定性越来越高，bug率越来越少

在做应用层的网络编程时，若服务器吞吐量大，则应该适度了解以上各层的关注点。

如上图红色文字所示，我认为编写高性能服务器的关注点有3个：
1、如果基于通用组件编程，关注点多是在组件如何封装套接字编程细节。为了使应用程序不感知套接字层，这些组件往往是通过各种回调机制来向应用层代码提供网络服务，通常，出于为应用层提供更高的开发效率，组件都大量使用了线程（Nginx等是个例外），当然，使用了线程后往往可以降低代码复杂度。但多线程引入的并发解决机

制还是需要重点关注的，特别是锁的使用。另外，使用多线程意味着把应用层的代码复杂度扔给了操作系统，大吞吐量时，需要关注多线程给操作系统内核带来的性能损耗。
基于通用组件编程，为了程序的高性能运行，需要清楚的了解组件的以下特性：怎么使用IO多路复用或者异步IO的？怎么实现并发性的？怎么组织线程模型的？怎么处理高吞吐量引发的异常情况的？

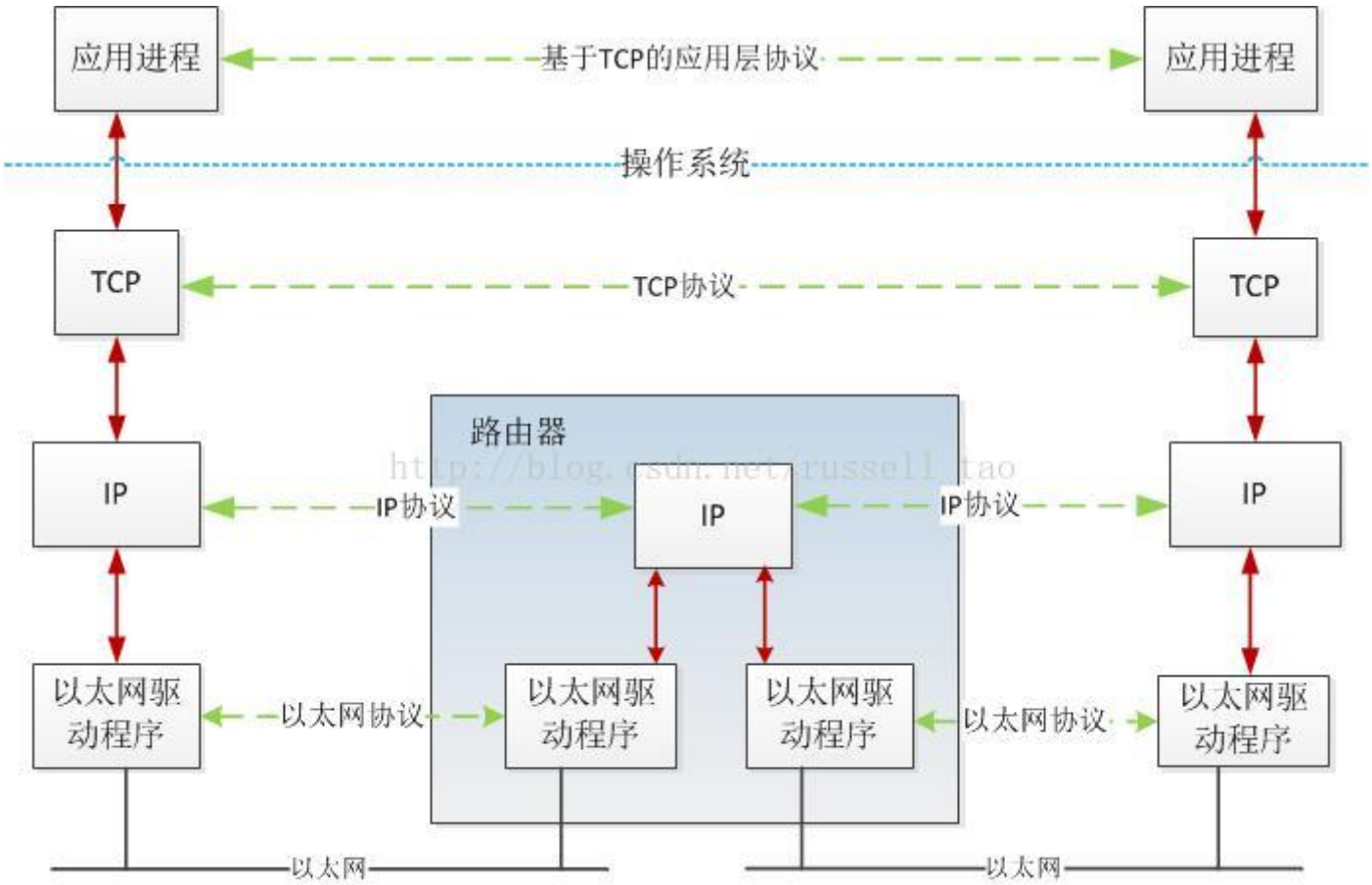
2、通用组件只是在封装套接字，操作系统是通过提供套接字来为进程提供网络通讯能力的。所以，不了解套接字编程，往往对组件的性能就没有原理上的认识。学习套接字层的编程是有必要的，或许很少会自己从头去写，但操作系统的API提供方式经久不变，一经学会，受用终身，同时在项目的架构设计时，选用何种网络组件就非常准确了。
学习套接字编程，关注点主要在：套接字的编程方法有哪些？阻塞套接字的各方法是如何阻塞住当前代码段的？非阻塞套接字上的方法如何不阻塞当前代码段的？IO多路复用机制是怎样与套接字结合的？异步IO是如何实现的？网络协议的各种异常情况、操作系统的各种异常情况是怎么通过套接字传递给应用性程序的？

3、网络的复杂性会影响到服务器的吞吐量，而且，高吞吐量场景下，多种临界条件会导致应用程序的不正常，特别是组件中有bug或考虑不周或没有配置正确时。了解网络分组可以定位出这些问题，可以正确的配置系统、组件，可以正确的理解系统的瓶颈。
这里的关注点主要在：TCP、UDP、IP协议的特点？linux等操作系统如何处理这些协议的？使用tcpdump等抓包工具分析各网络分组。

一般掌握以上3点，就可以挥洒自如的实现高性能网络服务器了。

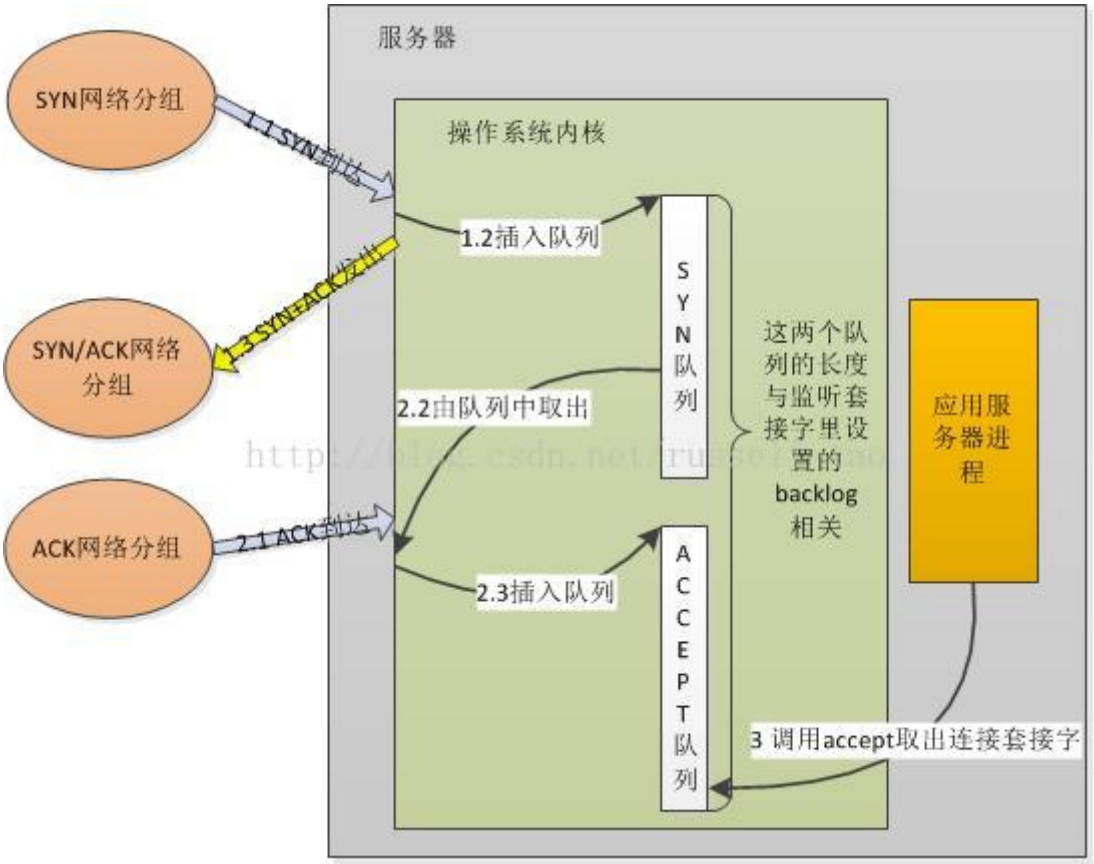
下面具体谈谈如何做到高性能网络编程。
众所周知，IO是计算机上最慢的部分，先不看磁盘IO，针对网络编程，自然是针对网络IO。网络协议对网络IO影响很大，当下，TCP/IP协议是毫无疑问的主流协议，本文就主要以TCP协议为例来说明网络IO。
网络IO中应用服务器往往聚焦于以下几个由网络IO组成的功能中：A）与客户端建立起TCP连接。B）读取客户端的请求流。C）向客户端发送响应流。D）关闭TCP连接。E）向其他服务器发起TCP连接。
要掌握住这5个功能，不仅仅需要熟悉一些API的使用，更要理解底层网络如何与上层API之间互相发生影响。同时，还需要对不同的场景下，如何权衡开发效率、进程、线程与这些API的组合使用。下面依次来说说这些网络IO。

1、与客户端建立起TCP连接
谈这个功能前，先来看看网络、协议、应用服务器间的关系：



上图中可知：
为简化不同场景下的编程，TCP/IP协议族划分了应用层、TCP传输层、IP网络层、链路层等，每一层只专注于少量功能。
例如，IP层只专注于每一个网络分组如何到达目的主机，而不管目的主机如何处理。
传输层最基本的功能是专注于端到端，也就是一台主机上的进程发出的包，如何到达目的主机上的某个进程。当然，TCP层为了可靠性，还额外需要解决3个大问题：丢包（网络分组在传输中存在的丢失）、重复（协议层异常引发的多个相同网络分组）、延迟（很久后网络分组才到达目的地）。
链路层则只关心以太网或其他二层网络内网络包的传输。

回到应用层，往往只需要调用类似于accept的API就可以建立TCP连接。建立连接的流程大家都了解--三次握手，它如何与accept交互呢？下面以一个不太精确却通俗易懂的图来说明之：



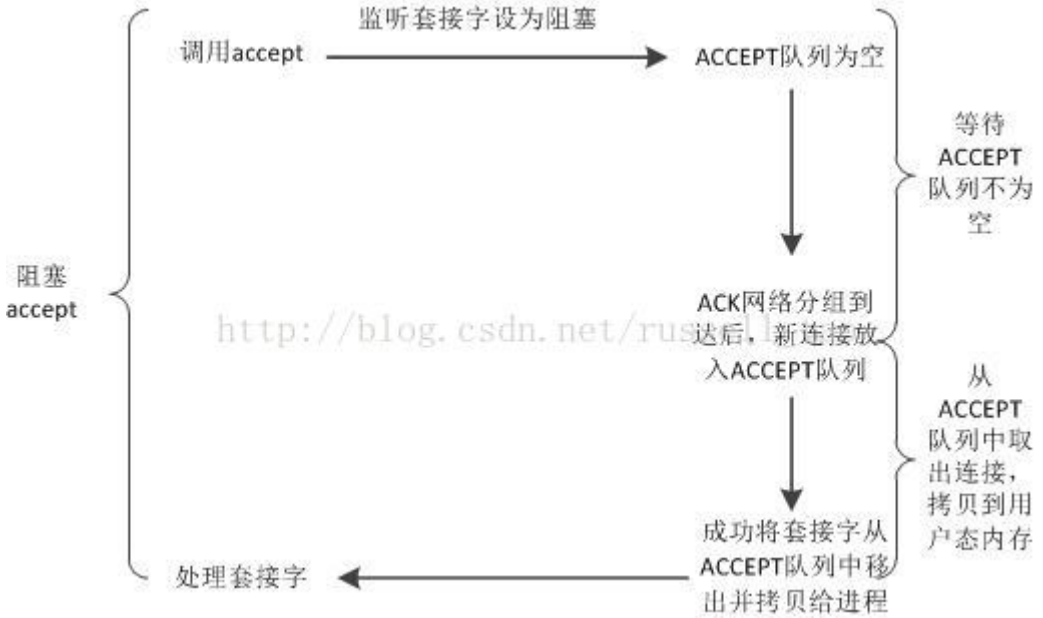
研究过backlog含义的朋友都很容易理解上图。这两个队列是内核实现的，当服务器绑定、监听了某个端口后，这个端口的SYN队列和ACCEPT队列就建立好了。客户端使用connect向服务器发起TCP连接，当图中1.1步骤客户端的SYN包到达了服务器后，内核会把这一信息放到SYN队列（即未完成握手队列）中，同时回一个SYN+ACK包给客户端。一段时间后，在较中2.1步骤中客户端再次发来了针对服务器SYN包的ACK网络分组时，内核会把连接从SYN队列中取出，再把这个连接放到ACCEPT队列（即已完成握手队列）中。而服务器在第3步调用accept时，其实就是直接从ACCEPT队列中取出已经建立成功的连接套接字而已。

现有我们可以来讨论应用层组件：为何有的应用服务器进程中，会单独使用1个线程，只调用accept方法来建立连接，例如tomcat；有的应用服务器进程中，却用1个线程做所有的事，包括accept获取新连接。

原因在于：首先，SYN队列和ACCEPT队列都不是无限长度的，它们的长度限制与调用listen监听某个地址端口时传递的backlog参数有关。既然队列长度是一个值，那么，队列会满吗？当然会，如果上图中第1步执行的速度大于第2步执行的速度，SYN队列就会不断增大直到队列满；如果第2步执行的速度远大于第3步执行的速度，ACCEPT队列同样会达到上限。第1、2步不是应用程序可控的，但第3步却是应用程序的行为，假设进程中调用accept获取新连接的代码段长期得不到执行，例如获取不到锁、IO阻塞等。

那么，这两个队列满了后，新的请求到达了又将发生什么？
若SYN队列满，则会直接丢弃请求，即新的SYN网络分组会被丢弃；如果ACCEPT队列满，则不会导致放弃连接，也不会把连接从SYN列队中移出，这会加剧SYN队列的增长。所以，对应用服务器来说，如果ACCEPT队列中有已经建立好的TCP连接，却没有及时的把它取出来，这样，一旦导致两个队列满了后，就会使客户端不能再建立新连接，引发严重问题。
所以，如TOMCAT等服务器会使用独立的线程，只做accept获取连接这一件事，以防止不能及时的去accept获取连接。

那么，为什么如Nginx等一些服务器，在一个线程内做accept的同时，还会做其他IO等操作呢？
这里就带出阻塞和非阻塞的概念。应用程序可以把listen时设置的套接字设为非阻塞模式（默认为阻塞模式），这两种模式会导致accept方法有不同的行为。对阻塞套接字，accept行为如下图：



这幅图中可以看到，阻塞套接字上使用accept，第一个阶段是等待ACCEPT队列不为空的阶段，它耗时不定，由客户端是否向自己发起了TCP请求而定，可能会耗时很长。
对非阻塞套接字，accept会有两种返回，如下图：

非阻塞套接字上的accept，不存在等待ACCEPT队列不为空的阶段，它要么返回成功并拿到建立好的连接，要么返回失败。

所以，企业级的服务器进程中，若某一线程既使用accept获取新连接，又继续在这个连接上读、写字符流，那么，这个连接对应的套接字通常要设为非阻塞。原因如上图，调用accept时不会长期占用所属线程的CPU时间片，使得线程能够及时的做其他工作。