11 | 答疑课堂：深入了解NIO的优化实现原理

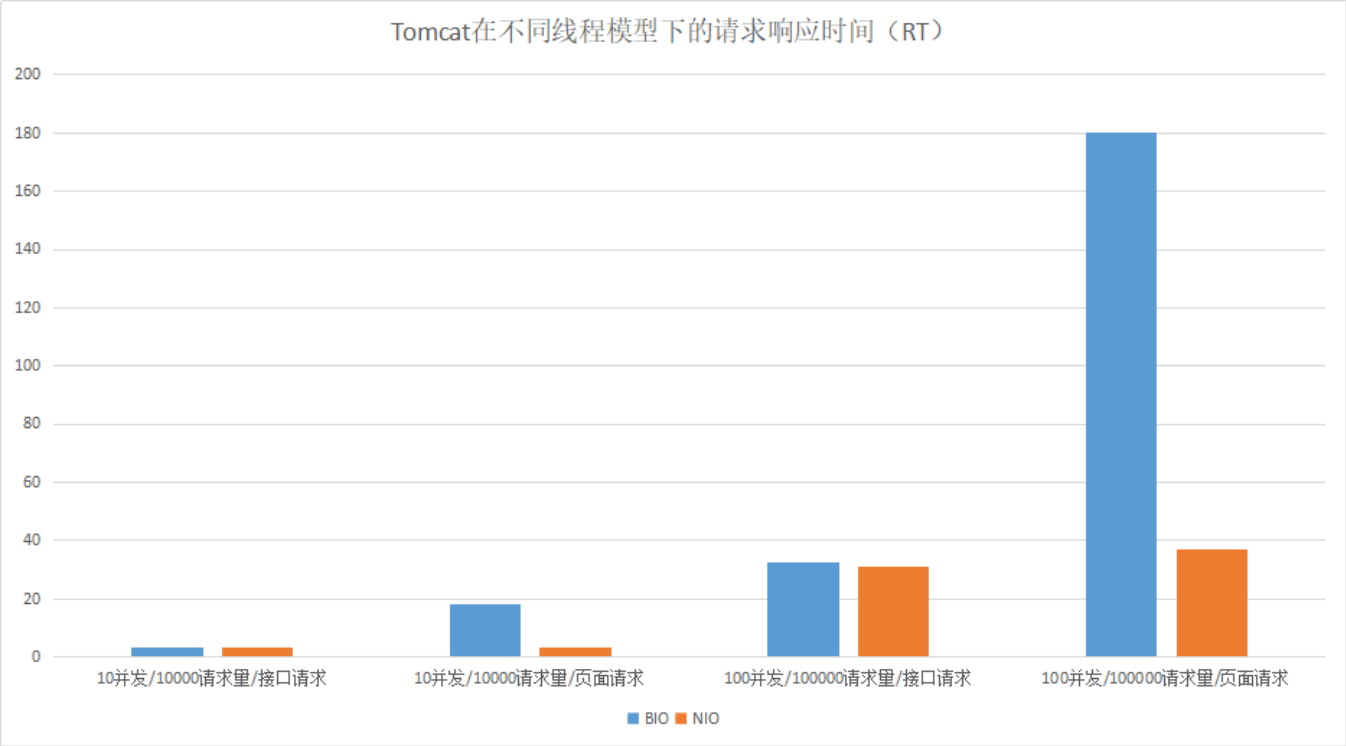
2019-06-13 刘超

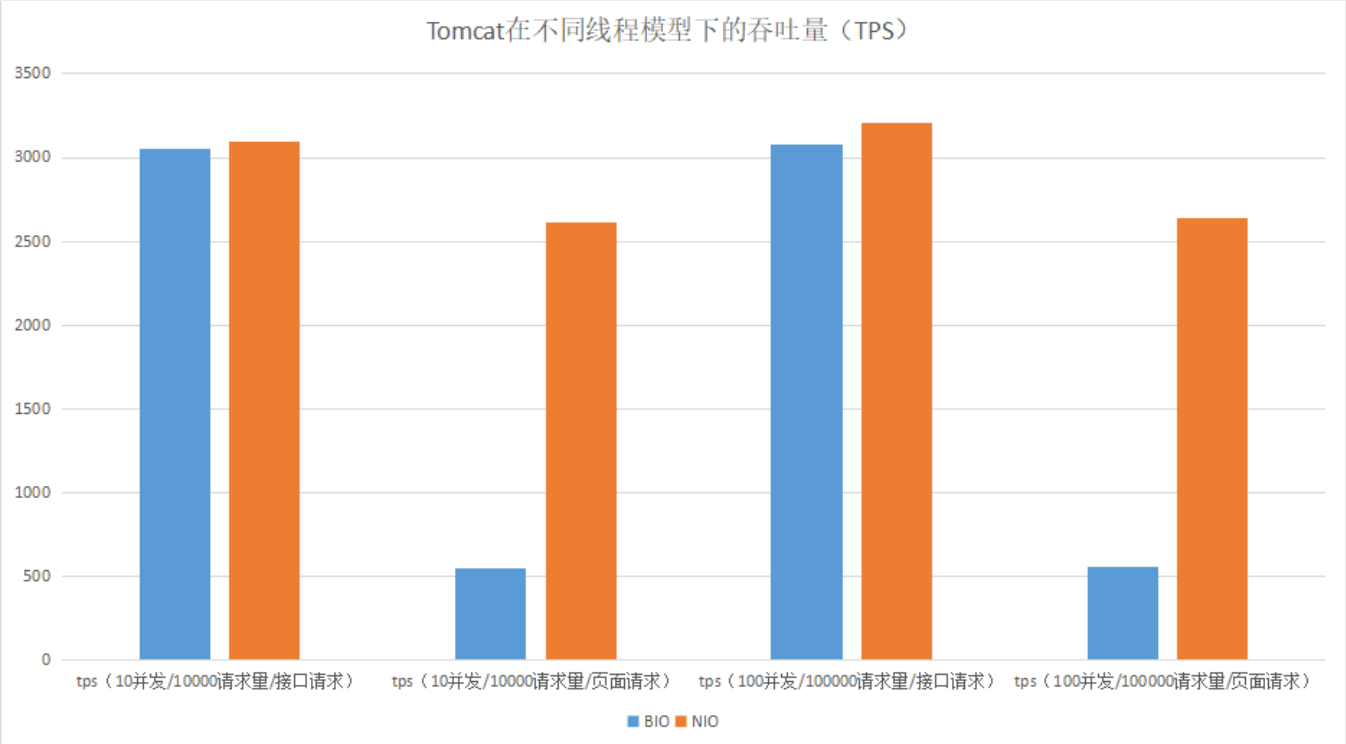
你好，我是刘超。专栏上线已经有20多天的时间了，首先要感谢各位同学的积极留言，交流的过程使我也收获良好。

综合查看完近期的留言以后，我的第一篇答疑课堂就顺势诞生了。我将继续讲解I/O优化，对大家在08讲中提到的内容做重点补充，并延伸一些有关I/O的知识点，更多结合实际场景进行分享。话不多说，我们马上切入正题。

Tomcat中经常被提到的一个调优就是修改线程的I/O模型。Tomcat 8.5版本之前，默认情况下使用的是BIO线程模型，如果在高负载、高并发的场景下，可以通过设置NIO线程模型，来提高系统的网络通信性能。

我们可以通过一个性能对比测试来看看在高负载或高并发的情况下，BIO和NIO通信性能（这里用页面请求模拟多I/O读写操作的请求）：





测试结果：Tomcat在I/O读写操作比较多的情况下，使用NIO线程模型有明显的优势。

Tomcat中看似一个简单的配置，其中却包含了大量的优化升级知识点。下面我们就从底层的网络I/O模型优化出发，再到内存拷贝优化和线程模型优化，深入分析下Tomcat、Netty等通信框架是如何通过优化I/O来提高系统性能的。

网络I/O模型优化

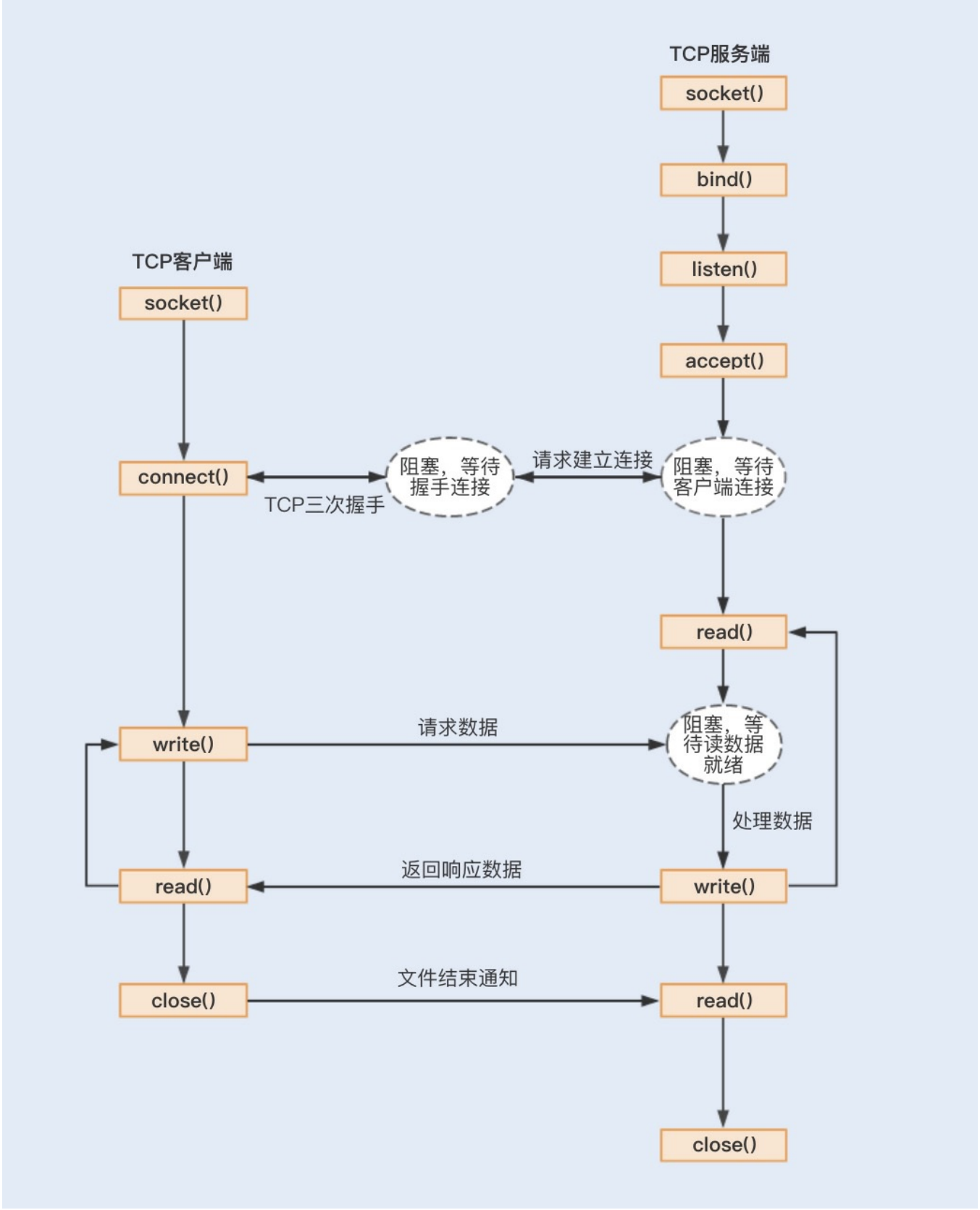
网络通信中，最底层的就是内核中的网络I/O模型了。随着技术的发展，操作系统内核的网络模型衍生出了五种I/O模型，《UNIX网络编程》一书将这五种I/O模型分为阻塞式I/O、非阻塞式I/O、I/O复用、信号驱动式I/O和异步I/O。每一种I/O模型的出现，都是基于前一种I/O模型的优化

升级。

最开始的阻塞式I/O，它在每一个连接创建时，都需要一个用户线程来处理，并且在I/O操作没有就绪或结束时，线程会被挂起，进入阻塞等待状态，阻塞式I/O就成为了导致性能瓶颈的根本原因。

那阻塞到底发生在套接字（socket）通信的哪些环节呢？

在《Unix网络编程》中，套接字通信可以分为流式套接字（TCP）和数据报套接字（UDP）。其中TCP连接是我们最常用的，一起来了解下TCP服务端的工作流程（由于TCP的数据传输比较复杂，存在拆包和装包的可能，这里我只假设一次最简单的TCP数据传输）：



- 首先，应用程序通过系统调用socket创建一个套接字，它是系统分配给应用程序的一个文件描述符；
- 其次，应用程序会通过系统调用bind，绑定地址和端口号，给套接字命名一个名称；
- 然后，系统会调用listen创建一个队列用于存放客户端进来的连接；
- 最后，应用服务会通过系统调用accept来监听客户端的连接请求。

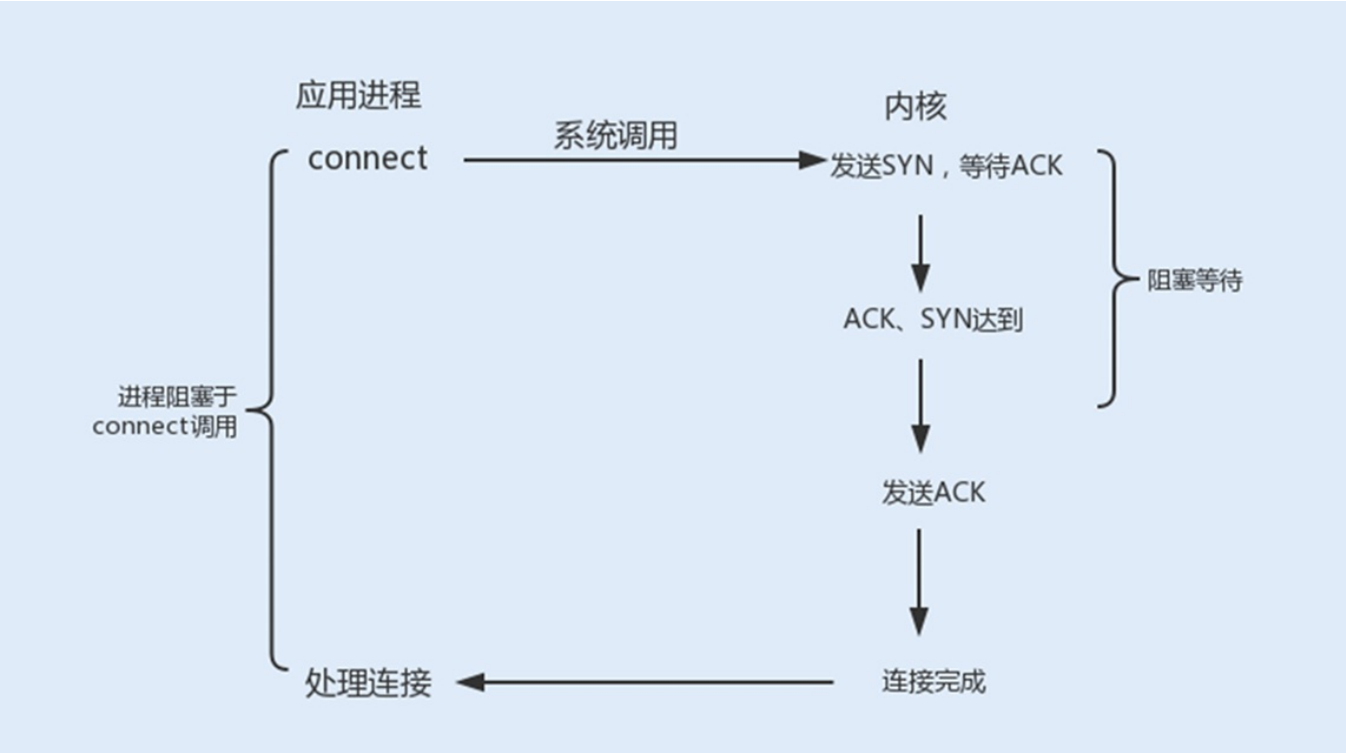
当有一个客户端连接到服务端之后，服务端就会调用fork创建一个子进程，通过系统调用read监

听客户端发来的消息，再通过write向客户端返回信息。

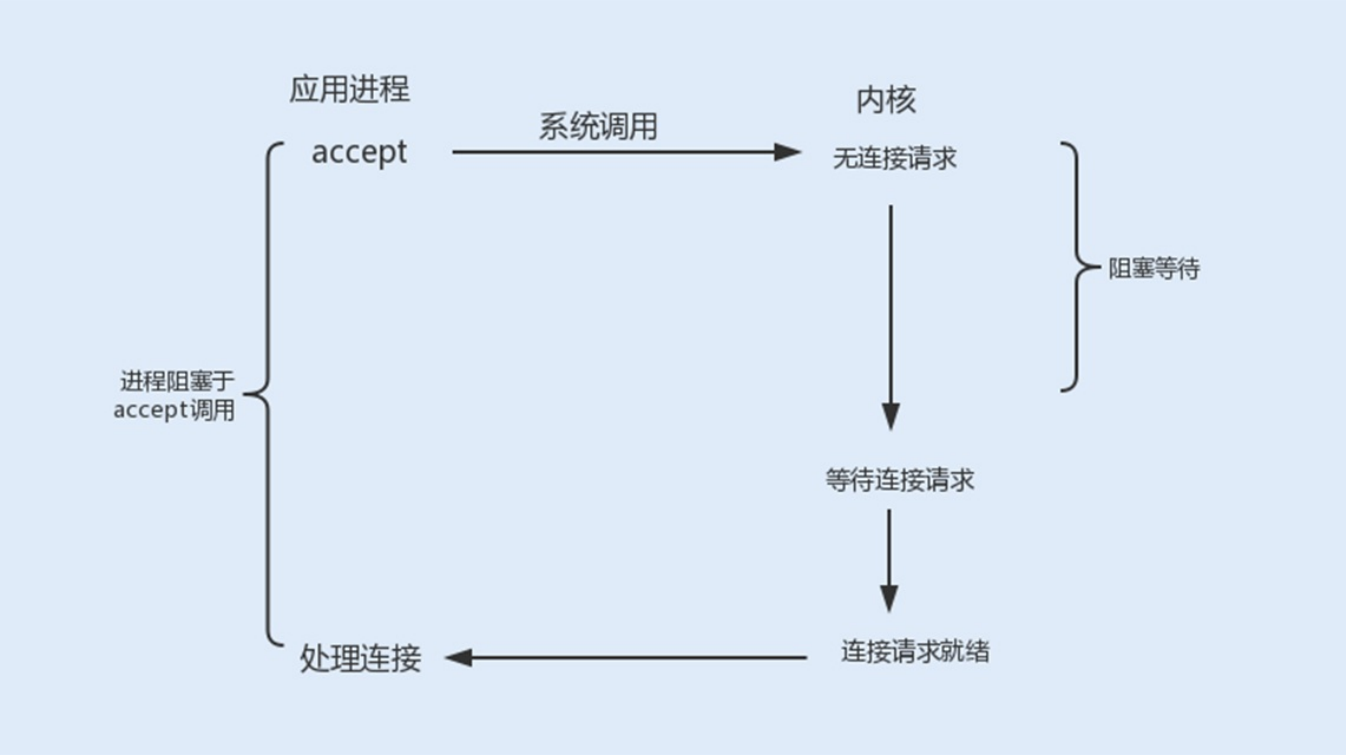
1.阻塞式I/O

在整个socket通信工作流程中，socket的默认状态是阻塞的。也就是说，当发出一个不能立即完成的套接字调用时，其进程将被阻塞，被系统挂起，进入睡眠状态，一直等待相应的操作响应。从上图中，我们可以发现，可能存在的阻塞主要包括以下三种。

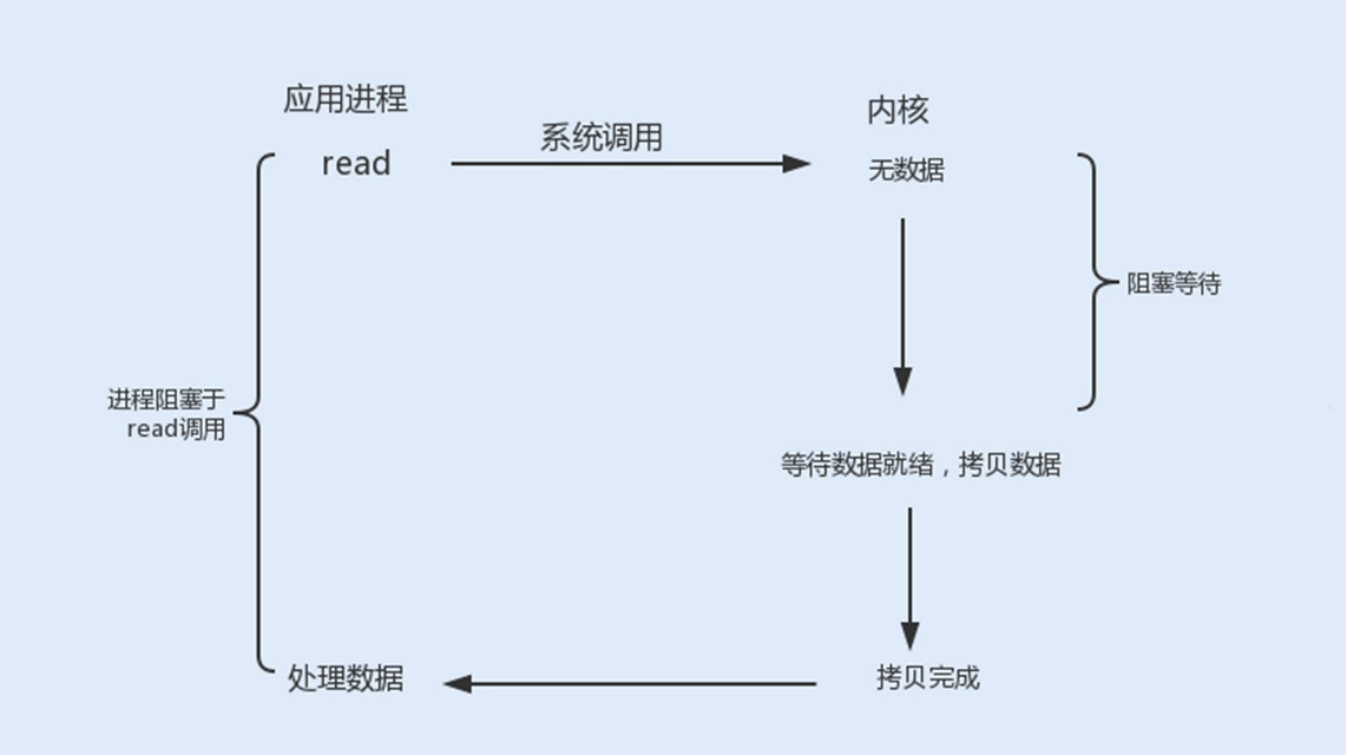
connect阻塞：当客户端发起TCP连接请求，通过系统调用connect函数，TCP连接的建立需要完成三次握手过程，客户端需要等待服务端发送回来的ACK以及SYN信号，同样服务端也需要阻塞等待客户端确认连接的ACK信号，这就意味着TCP的每个connect都会阻塞等待，直到确认连接。



accept阻塞：一个阻塞的socket通信的服务端接收外来连接，会调用accept函数，如果没有新的连接到达，调用进程将被挂起，进入阻塞状态。



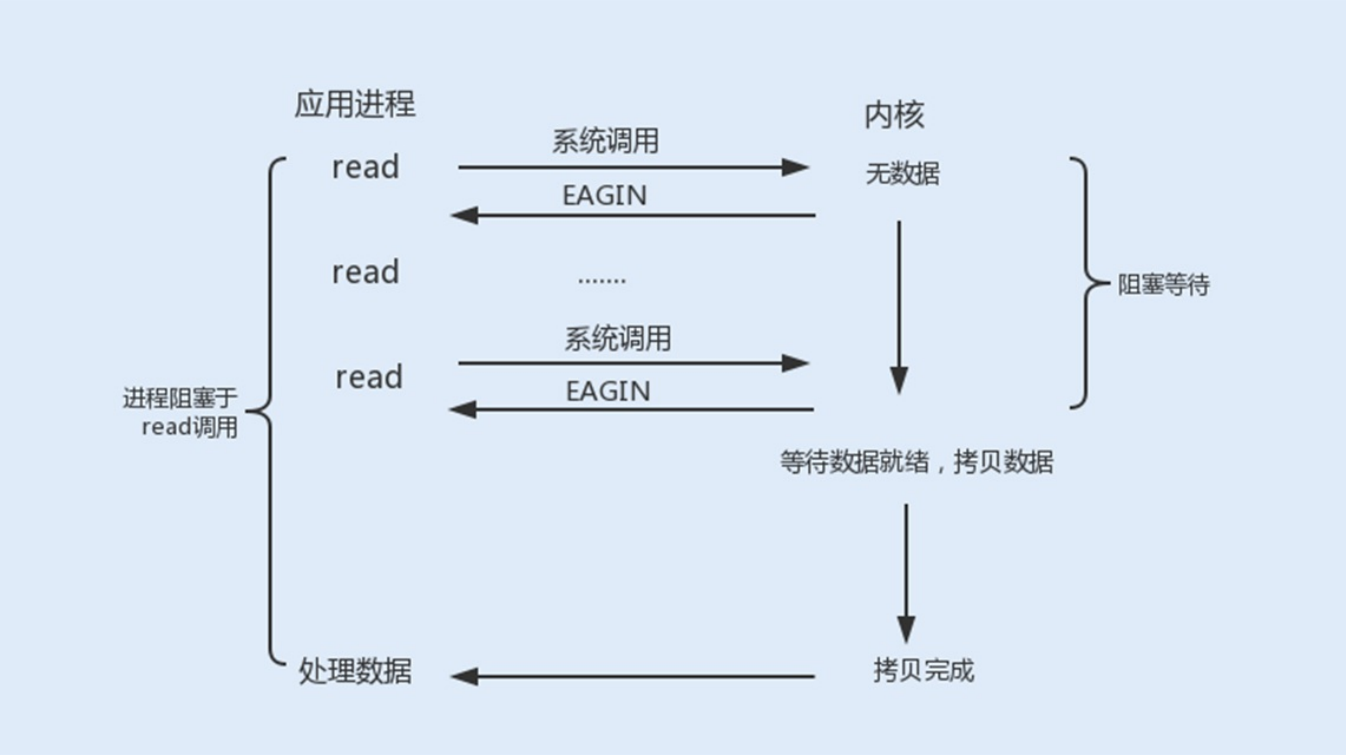
read、write阻塞：当一个socket连接创建成功之后，服务端用fork函数创建一个子进程， 调用read函数等待客户端的数据写入，如果没有数据写入，调用子进程将被挂起，进入阻塞状态。



2.非阻塞式I/O

使用fcntl可以把以上三种操作都设置为非阻塞操作。如果没有数据返回，就会直接返回一个EWOULDBLOCK或EAGAIN错误，此时进程就不会一直被阻塞。

当我们把以上操作设置为了非阻塞状态，我们需要设置一个线程对该操作进行轮询检查，这也是最传统的非阻塞I/O模型。



3. I/O复用

如果使用用户线程轮询查看一个I/O操作的状态，在大量请求的情况下，这对于CPU的使用率无疑是种灾难。 那么除了这种方式，还有其它方式可以实现非阻塞I/O套接字吗？

Linux提供了I/O复用函数select/poll/epoll，进程将一个或多个读操作通过系统调用函数，阻塞在函数操作上。这样，系统内核就可以帮我们侦测多个读操作是否处于就绪状态。

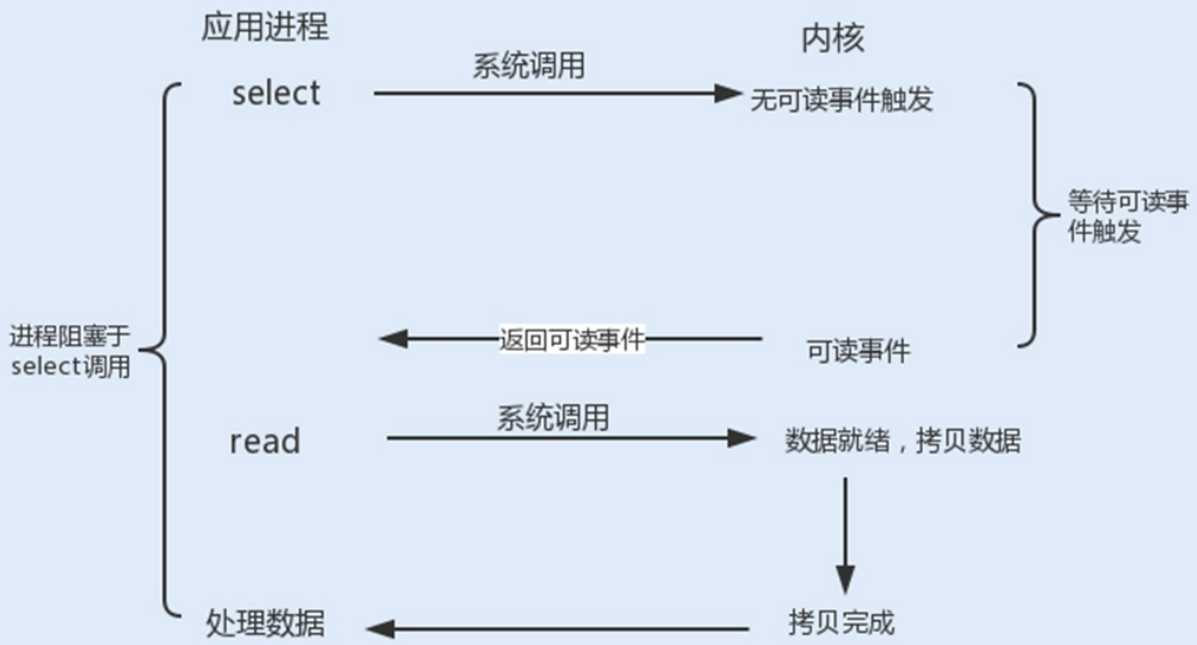
select()函数：它的用途是，在超时时间内，监听用户感兴趣的文件描述符上的可读可写和异常事件的发生。Linux 操作系统的内核将所有外部设备都看做一个文件来操作，对一个文件的读写操作会调用内核提供的系统命令，返回一个文件描述符（fd）。

```
int select(int maxfdp1,fd_set *readset,fd_set *writeset,fd_set *exceptset,const struct timeval *timeout)
```

查看以上代码，select() 函数监视的文件描述符分3类，分别是writefds（写文件描述符）、readfds（读文件描述符）以及exceptfds（异常事件文件描述符）。

调用后select() 函数会阻塞，直到有描述符就绪或者超时，函数返回。当select函数返回后，可以通过函数FD_ISSET遍历fdset，来找到就绪的描述符。fd_set可以理解为一个集合，这个集合中存放的是文件描述符，可通过以下四个宏进行设置：

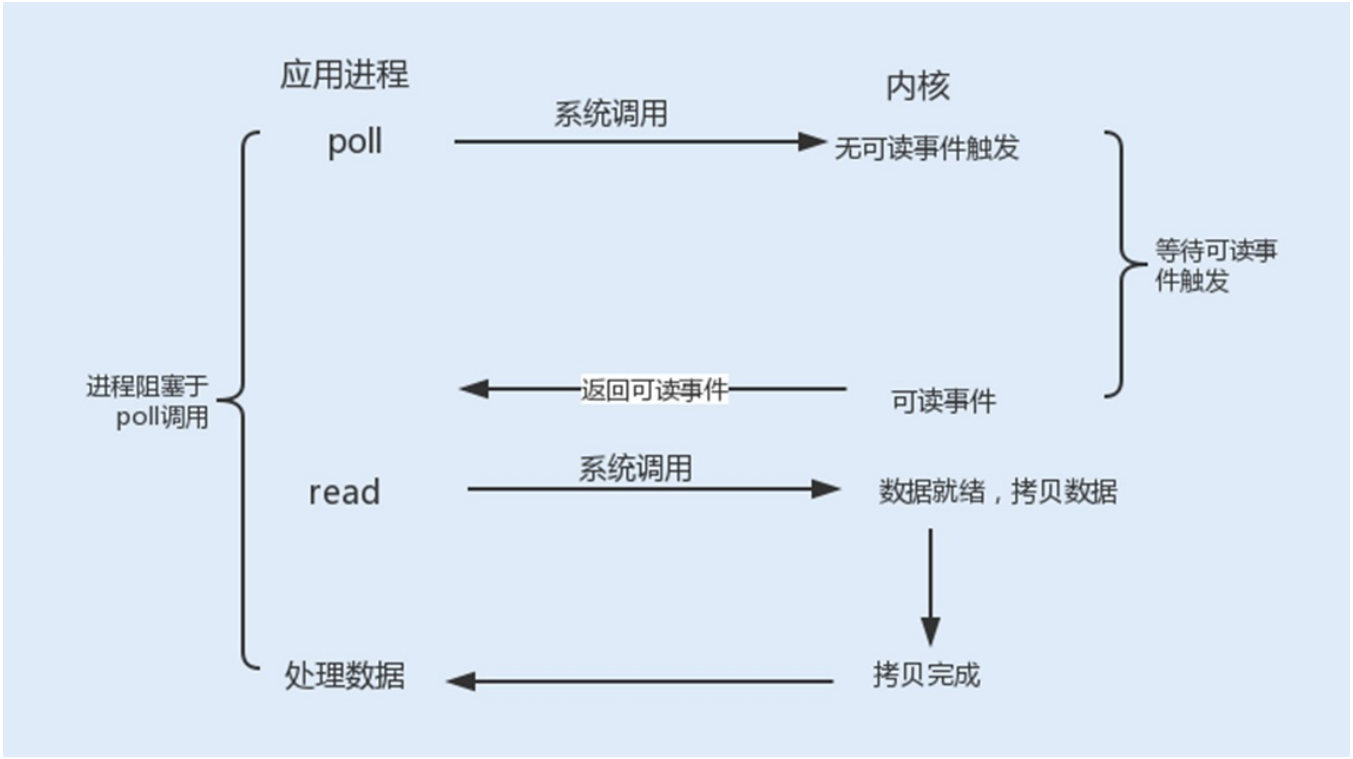
```
void FD_ZERO(fd_set *fdset);            //清空集合
void FD_SET(int fd, fd_set *fdset);   //将一个给定的文件描述符加入集合之中
void FD_CLR(int fd, fd_set *fdset);   //将一个给定的文件描述符从集合中删除
int FD_ISSET(int fd, fd_set *fdset);   // 检查集合中指定的文件描述符是否可以读写
```



poll()函数：在每次调用select()函数之前，系统需要把一个fd从用户态拷贝到内核态，这样就给系统带来了一定的性能开销。再有单个进程监视的fd数量默认是1024，我们可以通过修改宏定义甚至重新编译内核的方式打破这一限制。但由于fd_set是基于数组实现的，在新增和删除fd时，数量过大会导致效率降低。

poll() 的机制与 select() 类似，二者在本质上差别不大。poll() 管理多个描述符也是通过轮询，根据描述符的状态进行处理，但 poll() 没有最大文件描述符数量的限制。

poll() 和 select() 存在一个相同的缺点，那就是包含大量文件描述符的数组被整体复制到用户态和内核的地址空间之间，而无论这些文件描述符是否就绪，他们的开销都会随着文件描述符数量的增加而线性增大。



epoll()函数：select/poll是顺序扫描fd是否就绪，而且支持的fd数量不宜过大，因此它的使用受到了一些制约。

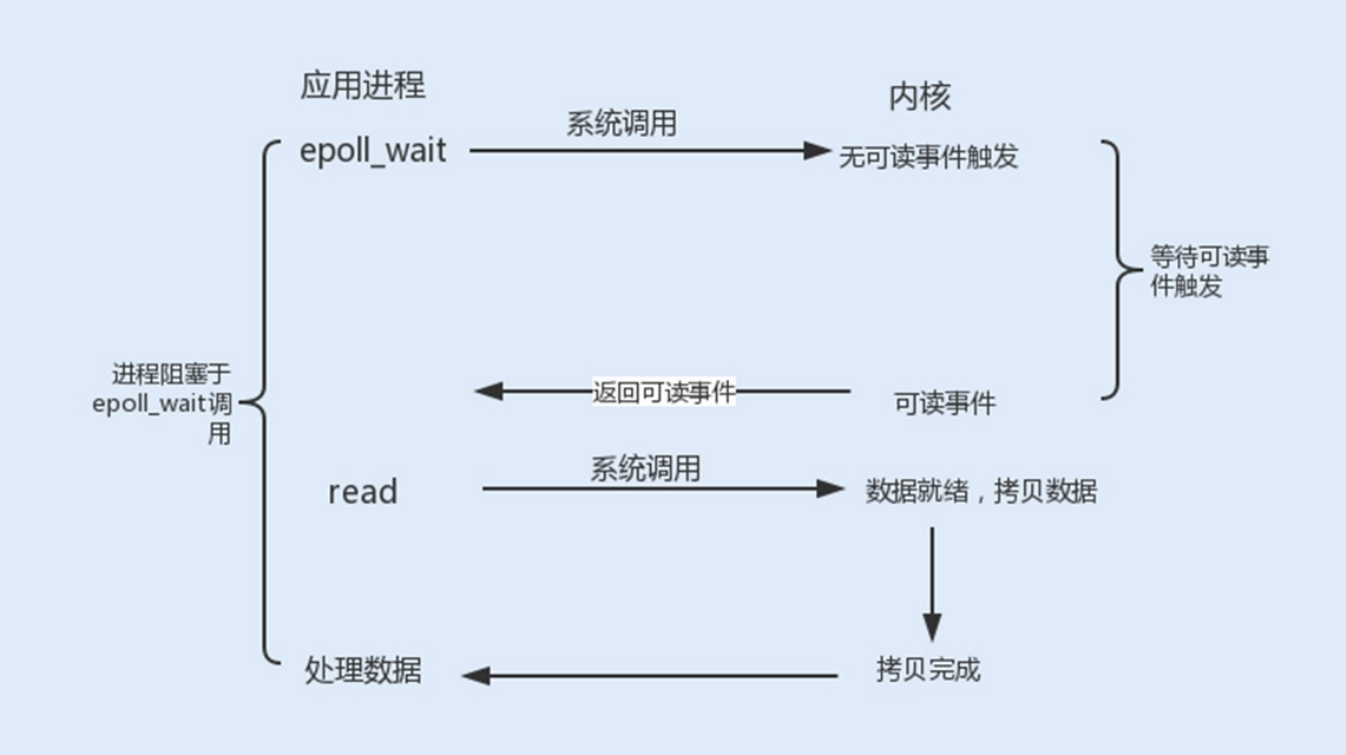
Linux在2.6内核版本中提供了一个epoll调用，epoll使用事件驱动的方式代替轮询扫描fd。epoll事先通过epoll_ctl()来注册一个文件描述符，将文件描述符存放到内核的一个事件表中，这个事件表是基于红黑树实现的，所以在大量I/O请求的场景下，插入和删除的性能比select/poll的数组fd_set要好，因此epoll的性能更胜一筹，而且不会受到fd数量的限制。

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

通过以上代码，我们可以看到：epoll_ctl()函数中的epfd是由 epoll_create()函数生成的一个epoll专用文件描述符。op代表操作事件类型，fd表示关联文件描述符，event表示指定监听的事件类型。

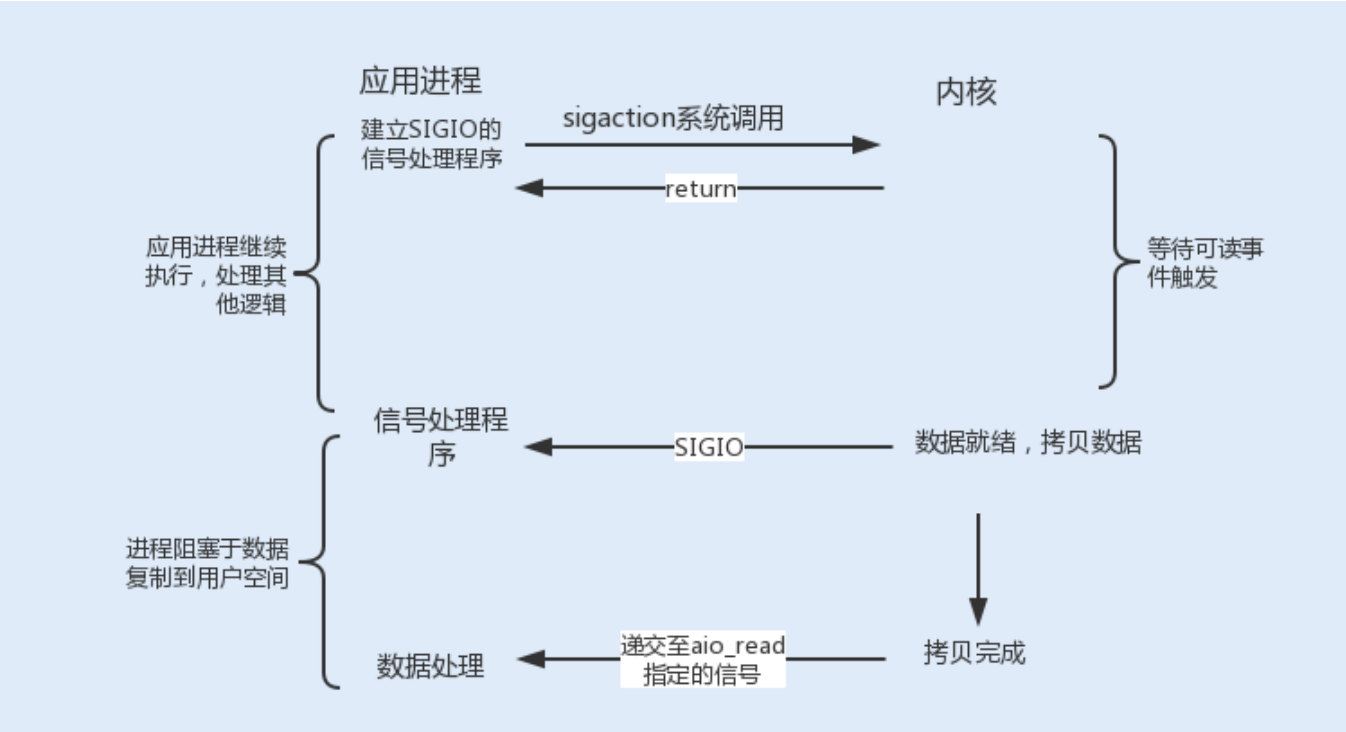
一旦某个文件描述符就绪时，内核会采用类似callback的回调机制，迅速激活这个文件描述符，当进程调用epoll_wait()时便得到通知，之后进程将完成相关I/O操作。

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



4.信号驱动式I/O

信号驱动式I/O类似观察者模式，内核就是一个观察者，信号回调则是通知。用户进程发起一个I/O请求操作，会通过系统调用sigaction函数，给对应的套接字注册一个信号回调，此时不阻塞用户进程，进程会继续工作。当内核数据就绪时，内核就为该进程生成一个SIGIO信号，通过信号回调通知进程进行相关I/O操作。



信号驱动式I/O相比于前三种I/O模式，实现了在等待数据就绪时，进程不被阻塞，主循环可以继续工作，所以性能更佳。

而由于TCP来说，信号驱动式I/O几乎没有被使用，这是因为SIGIO信号是一种Unix信号，信号没

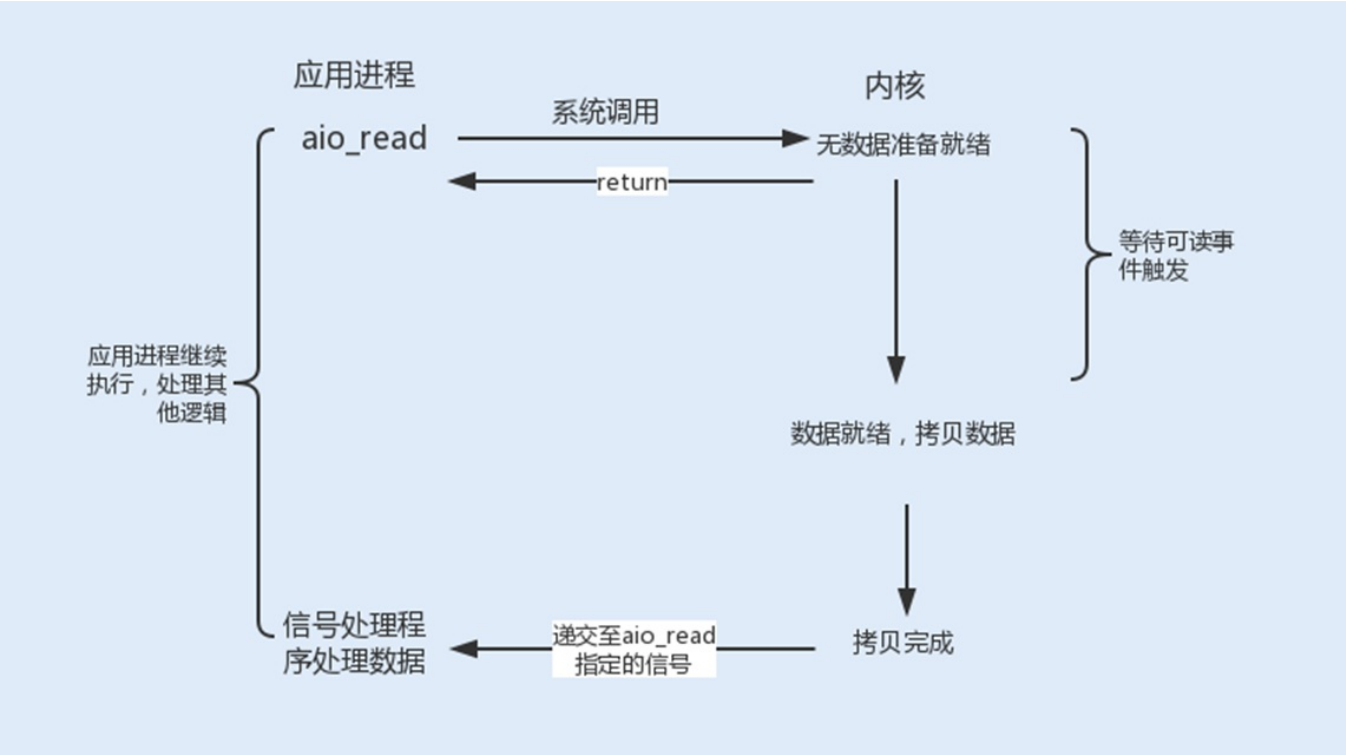
有附加信息，如果一个信号源有多种产生信号的原因，信号接收者就无法确定究竟发生了什么。而 TCP socket生产的信号事件有七种之多，这样应用程序收到 SIGIO，根本无从区分处理。

但信号驱动式I/O现在被用在了UDP通信上，我们从10讲中的UDP通信流程图中可以发现，UDP只有一个数据请求事件，这也就意味着在正常情况下UDP进程只要捕获SIGIO信号，就调用recvfrom读取到达的数据报。如果出现异常，就返回一个异常错误。比如，NTP服务器就应用了这种模型。

5.异步I/O

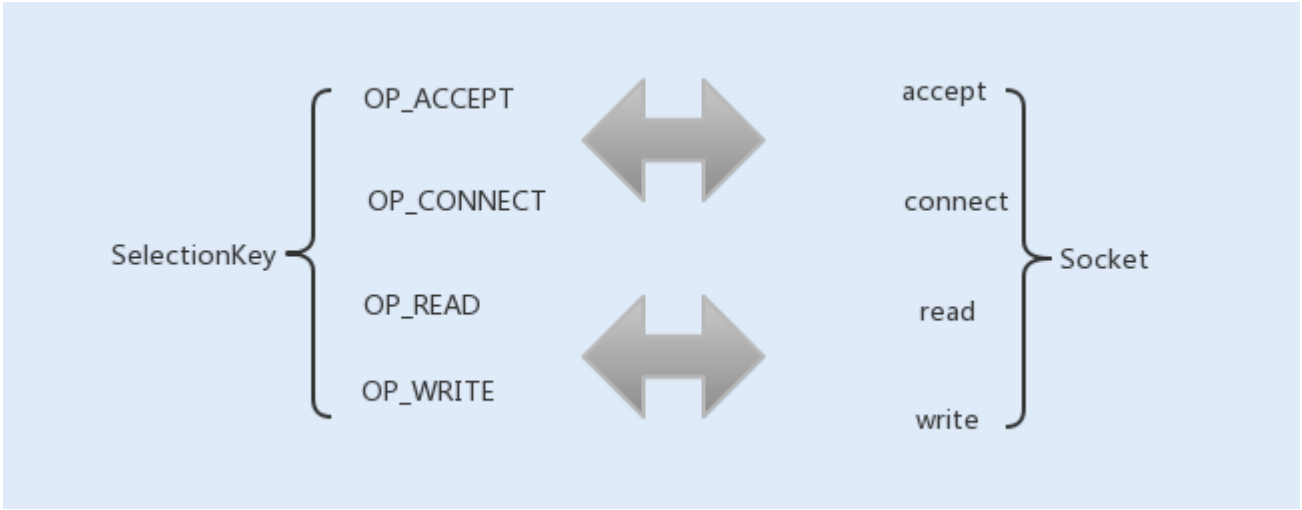
信号驱动式I/O虽然在等待数据就绪时，没有阻塞进程，但在被通知后进行的I/O操作还是阻塞的，进程会等待数据从内核空间复制到用户空间中。而异步I/O则是实现了真正的非阻塞I/O。

当用户进程发起一个I/O请求操作，系统会告知内核启动某个操作，并让内核在整个操作完成后通知进程。这个操作包括等待数据就绪和数据从内核复制到用户空间。由于程序的代码复杂度高，调试难度大，且支持异步I/O的操作系统比较少见（目前Linux暂不支持，而Windows已经实现了异步I/O），所以在实际生产环境中很少用到异步I/O模型。

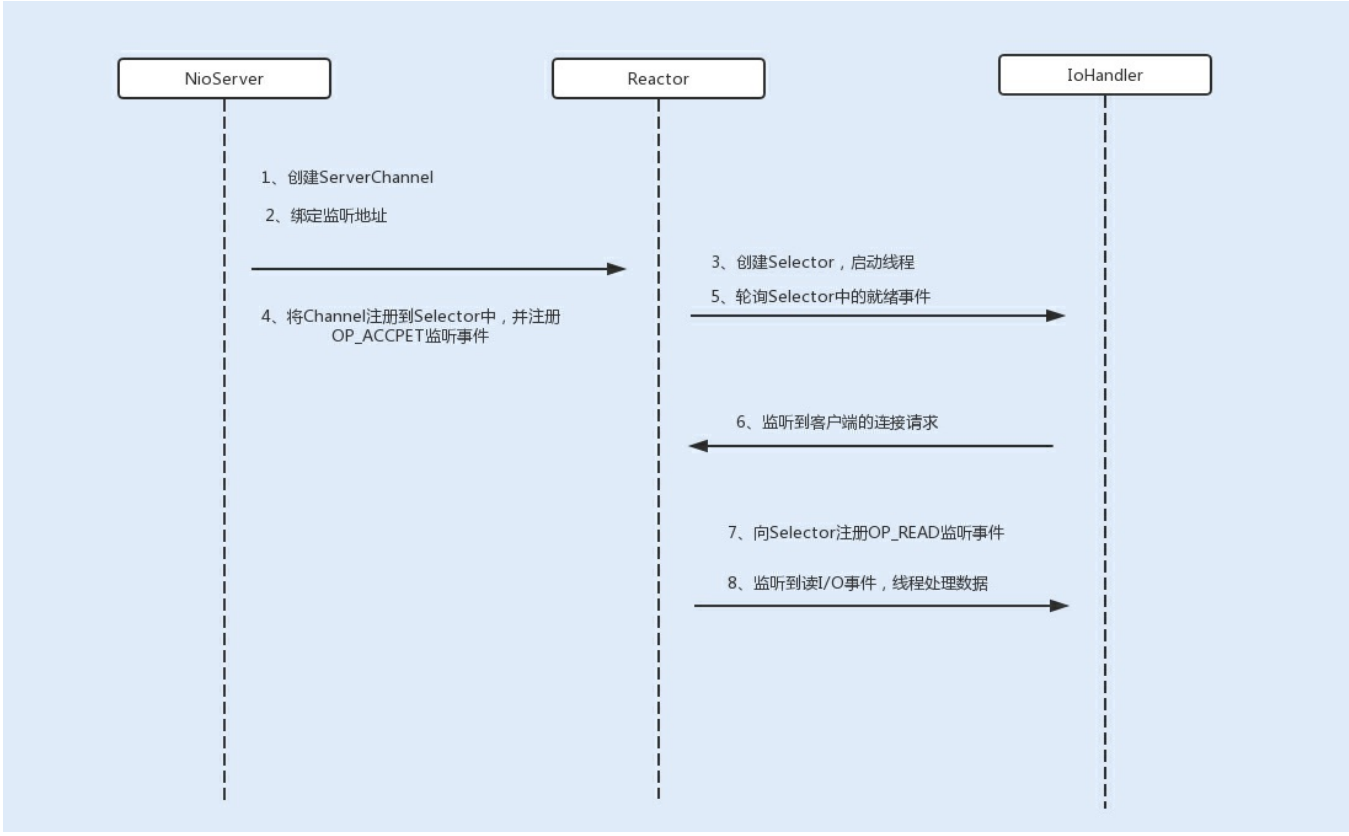


在08讲中，我讲到了NIO使用I/O复用器Selector实现非阻塞I/O，Selector就是使用了这五种类型中的I/O复用模型。Java中的Selector其实就是select/poll/epoll的外包类。

我们在上面的TCP通信流程中讲到，Socket通信中的conect、accept、read以及write为阻塞操作，在Selector中分别对应SelectionKey的四个监听事件OP_ACCEPT、OP_CONNECT、OP_READ以及OP_WRITE。



在NIO服务端通信编程中，首先会创建一个Channel，用于监听客户端连接；接着，创建多路复用器Selector，并将Channel注册到Selector，程序会通过Selector来轮询注册在其上的Channel，当发现一个或多个Channel处于就绪状态时，返回就绪的监听事件，最后程序匹配到监听事件，进行相关的I/O操作。



在创建Selector时，程序会根据操作系统版本选择使用哪种I/O复用函数。在JDK1.5版本中，如果程序运行在Linux操作系统，且内核版本在2.6以上，NIO中会选择epoll来替代传统的select/poll，这也极大地提升了NIO通信的性能。

由于信号驱动式I/O对TCP通信的不支持，以及异步I/O在Linux操作系统内核中的应用还不大成熟，大部分框架都还是基于I/O复用模型实现的网络通信。

零拷贝

在I/O复用模型中，执行读写I/O操作依然是阻塞的，在执行读写I/O操作时，存在着多次内存拷贝和上下文切换，给系统增加了性能开销。

零拷贝是一种避免多次内存复制的技术，用来优化读写I/O操作。

在网络编程中，通常由read、write来完成一次I/O读写操作。每一次I/O读写操作都需要完成四次内存拷贝，路径是I/O设备->内核空间->用户空间->内核空间->其它I/O设备。

Linux内核中的mmap函数可以代替read、write的I/O读写操作，实现用户空间和内核空间共享一个缓存数据。mmap将用户空间的一块地址和内核空间的一块地址同时映射到相同的一块物理内存地址，不管是用户空间还是内核空间都是虚拟地址，最终要通过地址映射映射到物理内存地址。这种方式避免了内核空间与用户空间的数据交换。I/O复用中的epoll函数中就是使用了mmap减少了内存拷贝。

在Java的NIO编程中，则是使用到了Direct Buffer来实现内存的零拷贝。Java直接在JVM内存空间之外开辟了一个物理内存空间，这样内核和用户进程都能共享一份缓存数据。这是在08讲中已经详细讲解过的内容，你可以再去回顾下。

线程模型优化

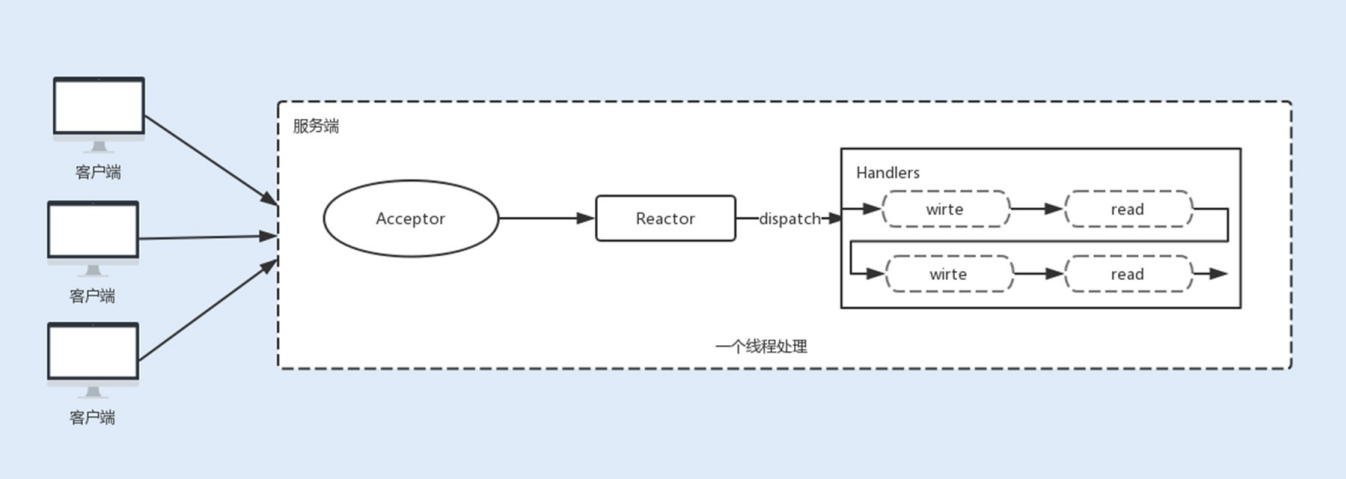
除了内核对网络I/O模型的优化，NIO在用户层也做了优化升级。NIO是基于事件驱动模型来实现的I/O操作。Reactor模型是同步I/O事件处理的一种常见模型，其核心思想是将I/O事件注册到多路复用器上，一旦有I/O事件触发，多路复用器就会将事件分发到事件处理器中，执行就绪的I/O事件操作。**该模型有以下三个主要组件：**

- 事件接收器Acceptor：主要负责接收请求连接；
- 事件分离器Reactor：接收请求后，会将建立的连接注册到分离器中，依赖于循环监听多路复用器Selector，一旦监听到事件，就会将事件dispatch到事件处理器；
- 事件处理器Handlers：事件处理器主要是完成相关的事件处理，比如读写I/O操作。

1.单线程Reactor线程模型

最开始NIO是基于单线程实现的，所有的I/O操作都是在一个NIO线程上完成。由于NIO是非阻塞I/O，理论上一个线程可以完成所有的I/O操作。

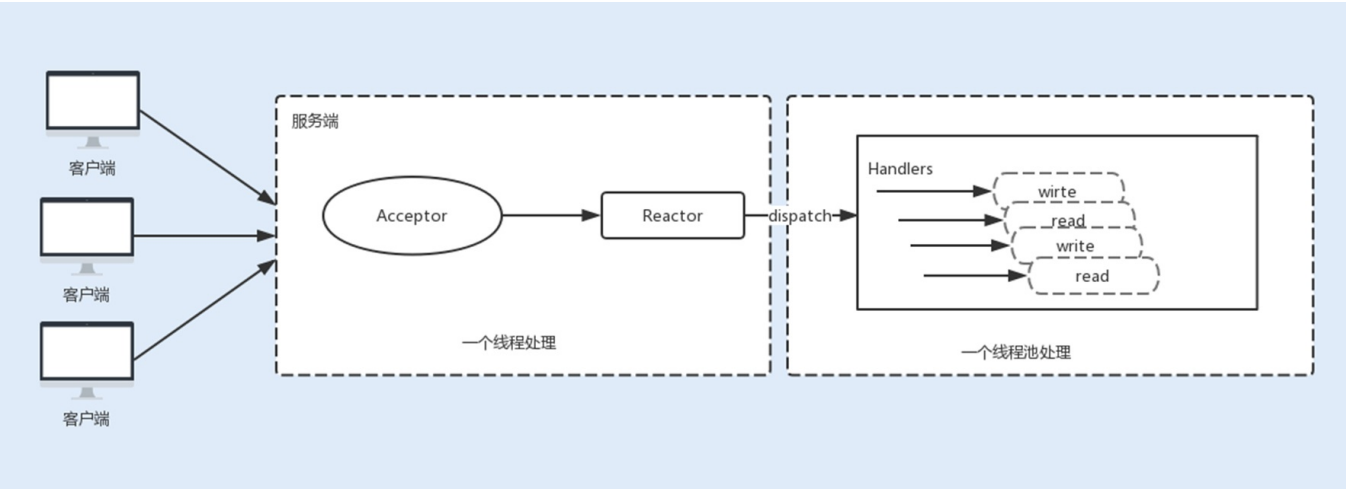
但NIO其实还不算真正地实现了非阻塞I/O操作，因为读写I/O操作时用户进程还是处于阻塞状态，这种方式在高负载、高并发的场景下会存在性能瓶颈，一个NIO线程如果同时处理上万连接的I/O操作，系统是无法支撑这种量级的请求的。



2.多线程Reactor线程模型

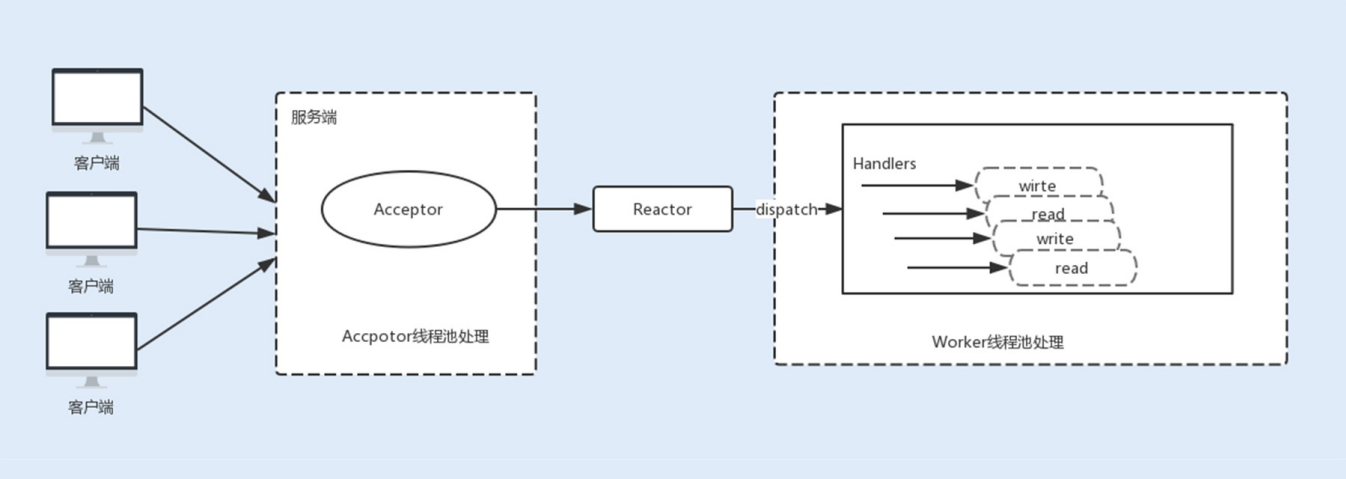
为了解决这种单线程的NIO在高负载、高并发场景下的性能瓶颈，后来使用了线程池。

在Tomcat和Netty中都使用了一个Acceptor线程来监听连接请求事件，当连接成功之后，会将建立的连接注册到多路复用器中，一旦监听到事件，将交给Worker线程池来负责处理。大多数情况下，这种线程模型可以满足性能要求，但如果连接的客户端再上一个量级，一个Acceptor线程可能会存在性能瓶颈。



3.主从Reactor线程模型

现在主流通信框架中的NIO通信框架都是基于主从Reactor线程模型来实现的。在这个模型中，Acceptor不再是一个单独的NIO线程，而是一个线程池。Acceptor接收到客户端的TCP连接请求，建立连接之后，后续的I/O操作将交给Worker I/O线程。

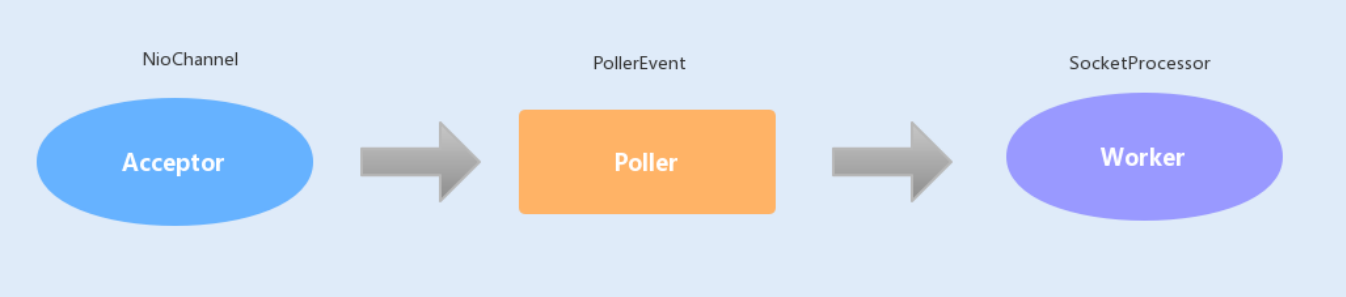


基于线程模型的Tomcat参数调优

Tomcat中，BIO、NIO是基于主从Reactor线程模型实现的。

在BIO中，Tomcat中的Acceptor只负责监听新的连接，一旦连接建立监听到I/O操作，将会交给Worker线程中，Worker线程专门负责I/O读写操作。

在NIO中，Tomcat新增了一个Poller线程池，Acceptor监听到连接后，不是直接使用Worker中的线程处理请求，而是先将请求发送给了Poller缓冲队列。在Poller中，维护了一个Selector对象，当Poller从队列中取出连接后，注册到该Selector中；然后通过遍历Selector，找出其中就绪的I/O操作，并使用Worker中的线程处理相应的请求。



你可以通过以下几个参数来设置Acceptor线程池和Worker线程池的配置项。

acceptorThreadCount：该参数代表Acceptor的线程数量，在请求客户端的数据量非常巨大的情况下，可以适当地调大该线程数量来提高处理请求连接的能力，默认值为1。

maxThreads：专门处理I/O操作的Worker线程数量，默认是200，可以根据实际的环境来调整该参数，但不一定越大越好。

acceptCount：Tomcat的Acceptor线程是负责从accept队列中取出该connection，然后交给工作线程去执行相关操作，这里的acceptCount指的是accept队列的大小。

当Http关闭keep alive，在并发量比较大时，可以适当地调大这个值。而在Http开启keep alive时，因为Worker线程数量有限，Worker线程就可能因长时间被占用，而连接在accept队列中等待超时。如果accept队列过大，就容易浪费连接。

maxConnections：表示有多少个socket连接到Tomcat上。在BIO模式中，一个线程只能处理一个连接，一般maxConnections与maxThreads的值大小相同；在NIO模式中，一个线程同时处理多个连接，maxConnections应该设置得比maxThreads要大的多，默认是10000。

今天的内容比较多，看到这里不知道你消化得如何？如果还有疑问，请在留言区中提出，我们共同探讨。最后欢迎你点击“请朋友读”，把今天的内容分享给身边的朋友，邀请他加入讨论。

精选留言

QQ怪 5

老师这篇可以配合隔壁专栏tomcat的13，14章一起看，会更加有味道。

2019-06-13

-W.LI- 2

老师好对Reacktor的三种模式还是理解不太好。帮忙看看哪里有问题
单线程模型:一个selector同时监听accept,事件和read事件。检测到就在一个线程处理。
多线程模型:一个线程监听accept事件，创建channel注册到selector上，检听到Read等事件从线程池中获取线程处理。
主从模式:没看懂:-(，一个端口只能被一个serverSocketChannel监听，第二个好像会报错?这边的主从怎么理解啊

2019-06-14

-W.LI- 1

老师好!万分感觉，写的非常非常好谢谢。不过开心的同时，好多没看懂:-(先讲下我的理解吧。
阻塞IO:调用read()线程阻塞了
非阻塞IO:调用read()马上拿到一个数据未就绪，或者就绪。
I/O多路复用:selector线程阻塞，channel非阻塞，用阻塞一个selector线程换了多个channel了非阻塞。select()函数基于数组，fd个数限制1024，poll()函数也是基于数组但是fd数目无限制。都会负责所有的fd(未就绪的开销浪了)，
epll()基于红黑数实现，fd无大小限制，平衡二叉数插入删除效率高。
信号驱动模式IO:对IO多路复用进一步优化，selector也非阻塞了。但是sign信号无法区分多信号源。所以socket未使用这种，只有在单一信号模型上才能应用。
异步IO模型:真正的非阻塞IO，其实前面的四种IO都不是真正的非阻塞IO，他们的非阻塞只是，从网络或者内存磁盘到内核空间的非阻塞，调用read()后还需要从内核拷贝到用户空间。异步IO基于回调，这一步也非阻塞了，从内核拷贝到用户空间后才通知用户进程。
能我是这么理解的前半断，有理解错的请老师指正谢谢。后半断没看完。

2019-06-13

作者回复

理解正确，赞一个

2019-06-14

每天晒白牙 0

老师您在介绍Reactor线程模型的时候，关于多线程Reactor线程模型和主从Reactor线程模型，我有不同的理解。您画的多线程模型，其中读写交给了线程池，我在看Doug Lea的《Scalable in java》中画的图和代码示例，读写事件还是由Reactor线程处理，只把业务处理交给了线程池。主从模型也是同样的，Reactor主线程处理连接，Reactor从线程池处理读写事件，业务交给单独的线程池处理。
还望老师指点

2019-06-15

作者回复

你好，Reactor是一个模型，每个框架或者每个开发人员在处理I/O事件可能不一样，根据自己业务场景来处理。

Netty是基于Reactor主线程去监听连接， Reactor从线程池监听读写事件，同时如果监听到事件后直接在该从线程中操作读写I/O，将业务交给单独的业务线程池，也可以不交给单独的线程池处理，直接在从线程池处理。不交给业务线程池的好处是，减少上下文切换，坏处是会造成线程阻塞。

所以根据自己的业务的特性，如果你的数据特别大，I/O读写操作放到handler线程池，，Reactor从线程数量有限，如果开大了，由于开多个多路复用器也会带来性能消耗。所以这种处理也是一种提高系统吞吐量的优化。

2019-06-16

-W.LI- 0

老师好!又看了一遍总结了下

epoll()方式的优点如下
1.无需用户空间到内核空间的fd拷贝过程。
2.通过事件表，只返回就绪事件无需轮训遍历
3.基于红黑树增删快。
4.事件发生后内核主动回调，用户进程wait状态(此时算阻塞还是非阻塞啊?)
内核也像观察者，(事件驱动的都像观察者)
还有别的优点么?

2019-06-15

西兹兹 0

刘老师，请问poller线程池 poller队列和文末提的acceptCount队列是不是一个队列？

2019-06-14

DemonLee 0

晕了，如果能结合点生活中的例子就更好了，我先去看看其他资料，再回来提问题。

2019-06-13

kim118000 0

acceptorThreadCount：该参数代表 Acceptor 的线程数量，在请求客户端的数据量非常巨大的情况下，可以适当地调大该线程数量来提高处理请求连接的能力，默认值为 1。

老师，我今天看了源码注释
doesn't seem to work that well with mutiple acceptor threads

我之前的理解是Java还做不到多个接受连接来提高请求连接的处理能力，目前普遍的做法是通过fork多个子进程来达到同时监听同一个socket fd，但这样有惊群，所以利用mutex多个监听者只有一个能处理本次连接操作

目前还是一主多从方案，但这已经够用，可以通过多台机器提高并发。

2019-06-13

陆离 0

我所使用的Tomcat版本是9，默认的就是NIO，是不是版本不同默认模型也不同？
directbuffer如果满了会阻塞还是会报错？这一块的大小设置是不是也可以优化？
因为Linux的aio这一块不成熟所以nio现在是主流？还是有其他原因？

2019-06-13

作者回复

是的，在Tomcat9版本改成了默认NIO。

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；

这个跟堆内存溢出是类似的道理，如果物理内存被分配完了就会出现溢出错误。NIO中的direct buffer是用来分配内存读取或写入数据操作，如果数据比较大，而directbuffer分配比较下，则会分多次去读写，如果数据比较大的情况下可以适当调大提高效率。

2019-06-14