



Java性能调优实战
刘超
金山软件西山居技术经理

查看详情

5880 人已学习

高并发下I/O瓶颈?

- 09 | 网络通信优化之序列化：避免使用 Java序列化
- 10 | 网络通信优化之通信协议：如何优化RPC网络通信？
- 11 | 答疑课堂：深入了解NIO的优化实现原理

模块三 · 多线程性能调优 (10讲)

模块四 · JVM性能监测及调优 (3讲)

11 | 答疑课堂：深入了解NIO的优化实现原理

刘超 2019-06-13



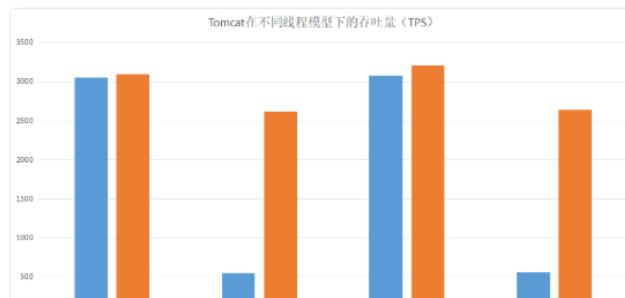
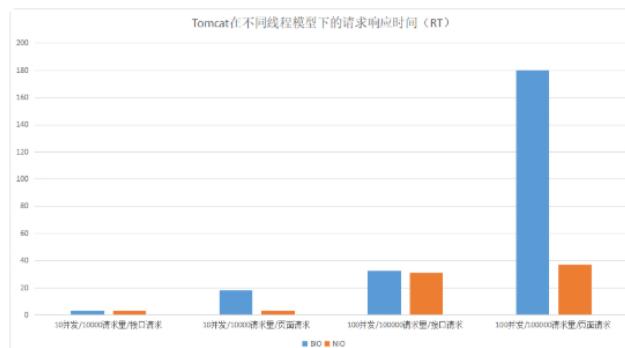
00:00 15:34
讲述：李良 大小：14.25M

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。

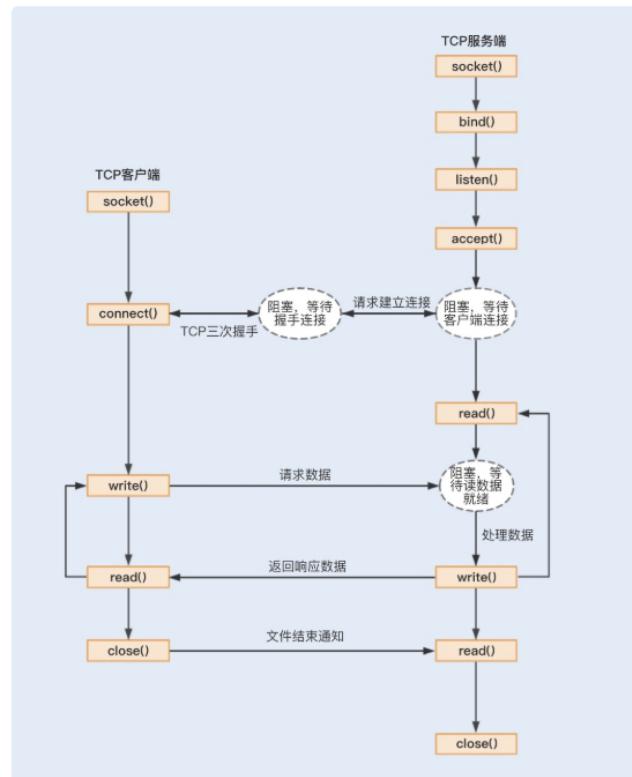
网络 I/O 模型优化

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）：

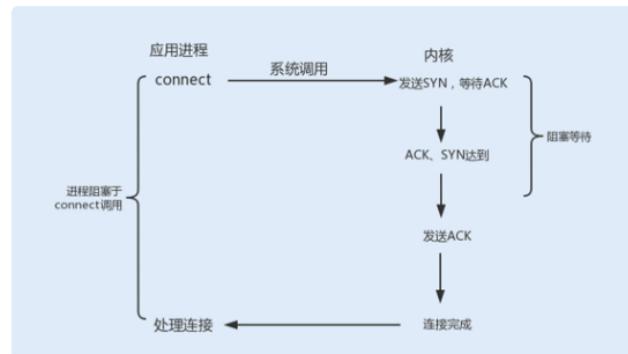


- 首先，应用程序通过系统调用 socket 创建一个套接字，它是系统分配给应用程序的一个文件描述符；
- 其次，应用程序会通过系统调用 bind，绑定地址和端口号，给套接字命名一个名称；
- 然后，系统会调用 listen 创建一个队列用于存放客户端进来的连接；
- 最后，应用服务会通过系统调用 accept 来监听客户端的连接请求。

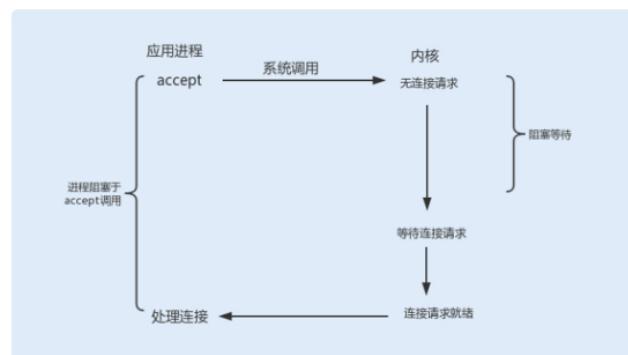
当有一个客户端连接到服务端之后，服务端就会调用 fork 创建一个子进程，通过系统调用 read 监听客户端发来的消息，再通过 write 向客户端返回信息。

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。

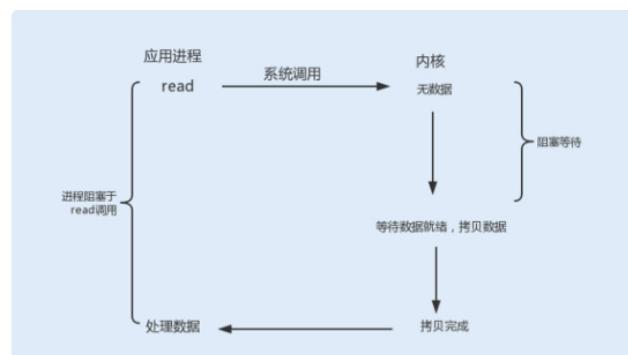
connect 阻塞：当客户端发起 TCP 连接请求，通过系统调用 connect 函数，TCP 连接的建立需要完成三次握手过程，客户端需要等待服务端发送回来的 ACK 以及 SYN 信号，同样服务端也需要阻塞等待客户端确认连接的 ACK 信号，这就意味着 TCP 的每个 connect 都会阻塞等待，直到确认连接。



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。



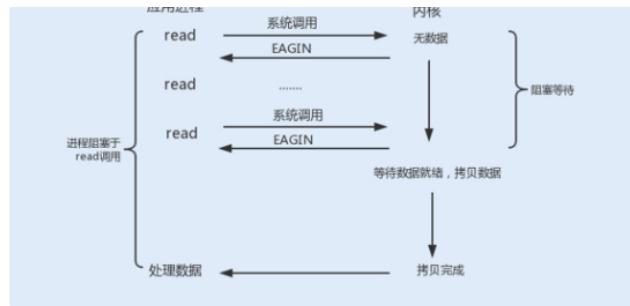
read、write 阻塞：当一个 socket 连接创建成功之后，服务端用 fork 函数创建一个子进程，调用 read 函数等待客户端的数据写入，如果没有数据写入，调用子进程将被挂起，进入阻塞状态。



2. 非阻塞 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。

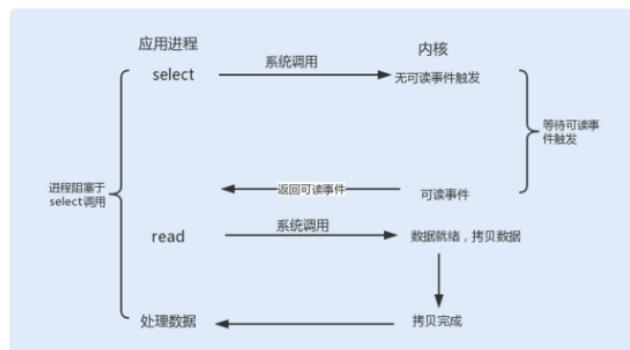
select() 函数：它的用途是，在超时时间内，监听用户感兴趣的文件描述符上的可读可写和异常事件的发生。Linux 操作系统的内核将所有外部设备都看做一个文件来操作，对一个文件的读写操作会调用内核提供的系统命令，返回一个文件描述符 (fd)。

```
1 int select(int maxfdp1, fd_set *readset, fd_set *writeset, fd_set *exceptset, const
```

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：

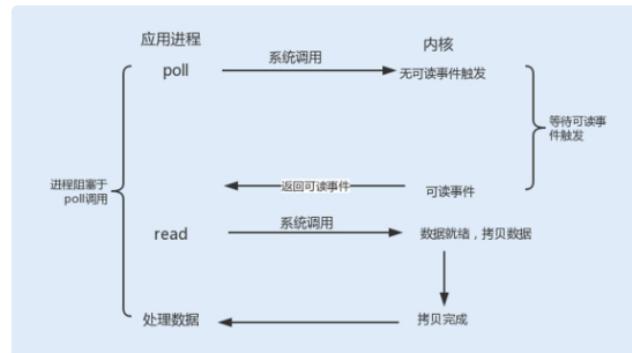
```
1 void FD_ZERO(fd_set *fdset);           // 清空集合
2 void FD_SET(int fd, fd_set *fdset);    // 将一个给定的文件描述符加入集合之中
3 void FD_CLR(int fd, fd_set *fdset);    // 将一个给定的文件描述符从集合中删除
4 int FD_ISSET(int fd, fd_set *fdset);   // 检查集合中指定的文件描述符是否可
```



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。

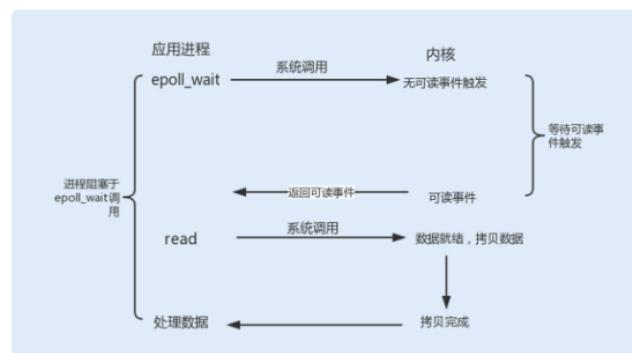
```
1 int epoll_ctl(int epfd, int op, int fd, struct epoll_event event)
2
3
```

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。

复制代码

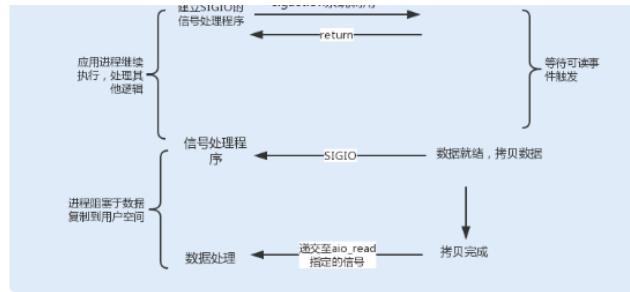
```
1 int epoll_wait(int epfd, struct epoll_event events,int maxevents,int timeout)
```



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。

应用进程 内核 网络编程



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。

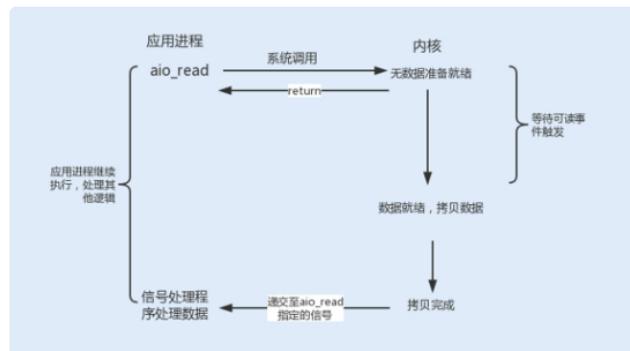
而由于 TCP 来说，信号驱动式 I/O 几乎没有被使用，这是因为 SIGIO 信号是一种 Unix 信号，信号没有附加信息，如果一个信号源有多种产生信号的原因，信号接收者就无法确定究竟发生了什么。而 TCP socket 生产的信号事件有七种之多，这样应用程序收到 SIGIO，根本无从区分处理。

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

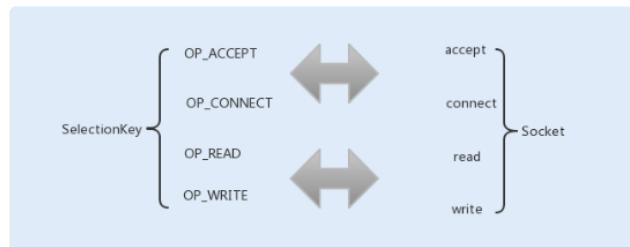
信号驱动式 I/O 虽然在等待数据就绪时，没有阻塞进程，但在被通知后进行的 I/O 操作还是阻塞的，进程会等待数据从内核空间复制到用户空间中。而异步 I/O 则是实现了真正的非阻塞 I/O。

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。



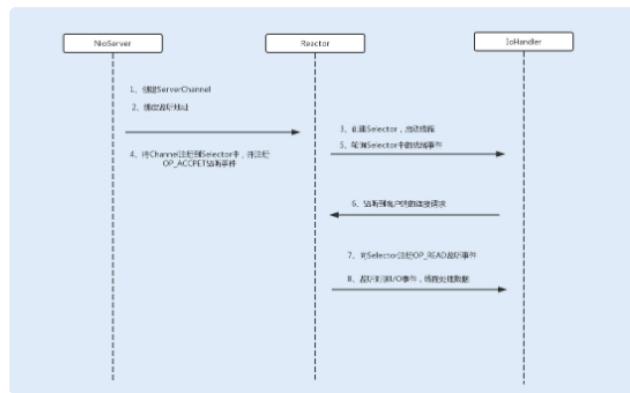
在 08 讲中，我讲到了 NIO 使用 I/O 复用器 Selector 实现非阻塞 I/O，[Selector 就是使用了这五种类型中的 I/O 复用模型](#)。Java 中的 Selector 其实就是 select/poll/epoll 的外包类。

我们在上面的 TCP 通信流程中讲到，Socket 通信中的 connect、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，开将 Channel 注册到 Selector，程序会通过 Selector 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：

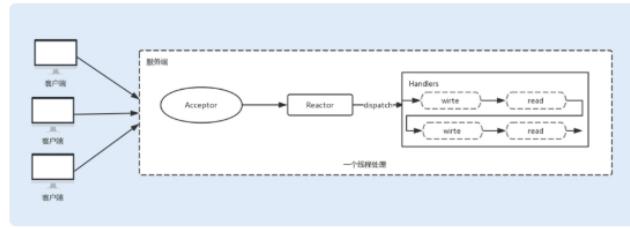
- 事件接收器 Acceptor：主要负责接收请求连接；
- 事件分离器 Reactor：接收请求后，会将建立的连接注册到分离器中，依赖于循环监听多路复用器 Selector，一旦监听到事件，就会将事件 dispatch 到事件处理器；
- 事件处理器 Handlers：事件处理器主要是完成相关的事件处理，比如读写 I/O 操作。

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

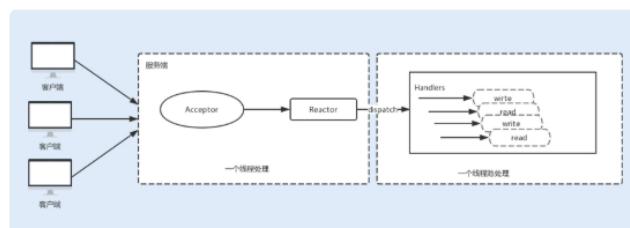
状态，这种方式在高负载、高并发的场景下会存在性能瓶颈，一个 NIO 线程如果同时处理上万连接的 I/O 操作，系统是无法支撑这种量级的请求的。



2. 多线程 Reactor 线程模型

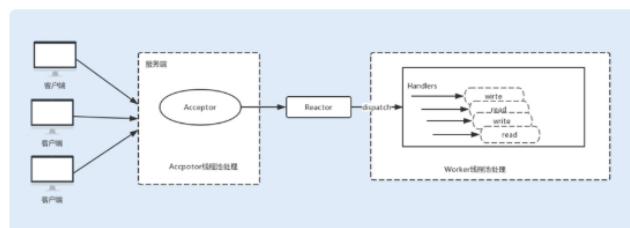
为了解决这种单线程的 NIO 在高负载、高并发场景下的性能瓶颈，后来使用了线程池。

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。



© 版权归极客邦科技所有，未经许可不得传播售卖。页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

胡晓

由作者筛选后的优质留言将会公开显示，欢迎踊跃留言。

Command + Enter 发表

0/2000字

提交留言

精选留言(16)

QQ怪

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

8

-W.LI-

老师好对Reactor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

作者回复: 主从模式则是, Reactor主线程主要处理监听连接事件, 而Reactor从线程主要监听I/O事件。这里是多线程处理accept事件, 而不是创建多个ServerSocketChannel。

2019-06-14

···

4

52

 -W.LI-

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),

epoll()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I/O多路复用进一步优化, selector也非阻塞了。但是signl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段, 有理解错的请老师指正谢谢。后半段没看完。

作者回复: 理解正确, 赞一个

2019-06-13

···

4

52

 每天晒白牙

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

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。

2019-06-15

···

2

52

 余冲

老师能对reactor的几种模型, 给一个简单版的代码例子看看吗。感觉通过代码应该能更好的理解理论。

作者回复: 好的, 后面补上

2019-06-18

···

1

52

 行者

感谢老师分享, 联想到Redis的单线程模式, Redis使用同一个线程来做selector, 以及处理handler, 这样的优点是减少上下文切换, 不需要考虑并发问题; 但是缺点也很明显, 在IO数据量大的情况下, 会导致QPS下降; 这是由Redis选择IO模型决定的。

作者回复: 对的, redis本身是操作内存, 所以读取数据的效率会高很多。

2019-07-14



kaixiao7

老师, 有两个疑惑还望您解答, 谢谢

1. ulimit -n 显示单个进程的文件句柄数为1024, 但是启动一个socket服务(bio实现)后, cat /proc/<pid>/limits 中显示的open files为4096, 实际测试当socket连接数达到4000左右时就无法再连接了. 还请老师解答一下4096怎么来的, 为什么1024不生效呢?

2. 您在文中提到epoll不受fd的限制. 但是我用NIO实现的服务端也是在连接到4000左右时无法再接收新的连接, 环境为Centos7(虚拟机, 内核3.10, 除了系统之外, 没有跑其他程序), jdk1.8, ulimit -n 结果为1024, cat /proc/sys/fs/file-max 结果为382293. 按理说socket连接数最大可以达到38000左右, 代码如下:

```
public static void main(String[] args) throws IOException {  
    ServerSocketChannel channel = ServerSocketChannel.open();  
    Selector selector = Selector.open();
```

```
    channel.configureBlocking(false);  
    channel.socket().bind(new InetSocketAddress(10301));  
    channel.register(selector, SelectionKey.OP_ACCEPT);
```

```
    int size = 0;
```

```
    while (true) {  
        if (selector.select() == 0) {  
            continue;  
        }
```

```
        Iterator<SelectionKey> iterator = selector.selectedKeys().iterator();  
        while (iterator.hasNext()) {  
            SelectionKey key = iterator.next();  
            iterator.remove();
```

```
            if (key.isAcceptable()) {  
                size++;  
                ServerSocketChannel server = (ServerSocketChannel) key.channel();  
                SocketChannel client = server.accept();
```

```
                System.out.println("当前客户端连接数: " + size + ", " + client.getRemoteAddress());  
            }
```

```
}
```

```
}
```

2019-07-11



Geek_ebda96

老师, 请教一个问题, maxthreads这个参数在tomcat中是指只是单独处理I/O的读写线程数, 还是读取完数据后, 本身的业务层处理也是在这个线程池里处理

2019-06-25



吾爱有三

文章多次提到挂起会进入阻塞状态, 然到挂起等价阻塞? 不是吧

作者回复: 这里的挂起是一个动作, 阻塞是一种状态。

2019-06-18



赵衍

I/O多路复用其实就相当于用了一个专门的线程来监听多个注册的事件, 而之前的I/O模型中, 每一个事件都需要一个线程来监听, 不知道我这样理解的是否正确? 老师我还有一个问题, 就是当select监听到一个事件到来时, 它是另起一个线程把数据从内核态拷贝到用户态, 还是自己就把这个事儿给干了?

作者回复: 理解正确。select监听到事件之后就用当前线程把数据从内核态拷贝到用户态。

2019-06-17

···

+

z.l

老师,隔壁李号双老师的《深入拆解Tomcat & Jetty》中关于DirectByteBuffer的解释和您不一样,他的文章中DirectByteBuffer的作用是: DirectByteBuffer 避免了 JVM 堆与本地内存直接的拷贝,而并没有避免内存从内核空间到用户空间的拷贝。而sendfile 特性才是避免了内核与应用之间的内存拷贝。请问哪种才是对的?

作者回复: 这里的本地内存应该指的是物理内存,避免堆内存和物理内存的拷贝,其实就是避免内核空间和用户空间的拷贝。

2019-06-16

···

+

-W.LI-

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?

作者回复: epoll是使用了wait方法阻塞等待事件,所以是阻塞的。

2019-06-15

···

+

西茲茲

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?

作者回复: 不是的,这个acceptCount是Acceptor的线程数量,也就是Reactor主线程数量。

2019-06-14

···

+

DemonLee

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

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

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？

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directbuffer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分配比较大，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。

2019-06-13

