1、redis 线程模型

单进程单线程

单进程单线程为何那么快?

1、基于内存操作

2、I/O多路复用

3、没有上下文切换的开销和锁的开销

2、Nginx线程模型

多进程单线程

一、进程模型 Nginx之所以为广大码农喜爱，除了其高性能外，还有其优雅的系统架构。与Memcached的经典多线程模型相比，Nginx是经典的多进程模型。Nginx启动后以daemon的方式在后台运行，后台进程包含一个master进程和多个worker进程，具体如下图：

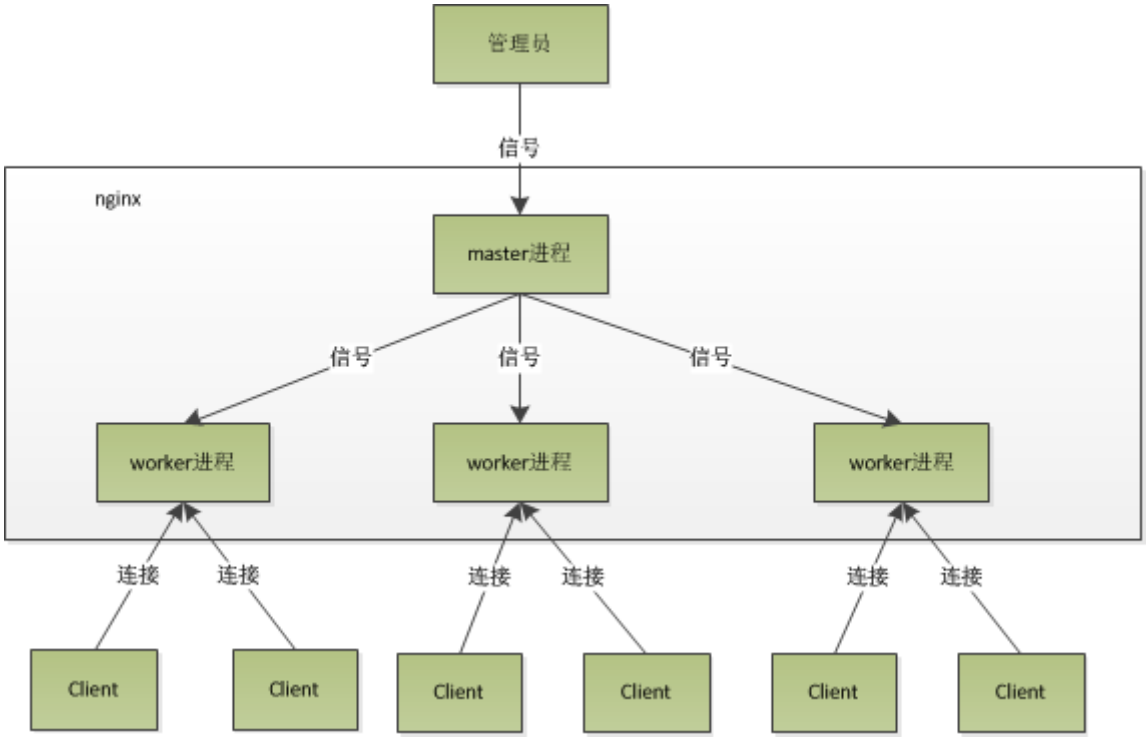


图1 Nginx多进程模型

master进程主要用来管理worker进程，具体包括如下4个主要功能： （1）接收来自外界的信号。（2）向各worker进程发送信号。 （3）监控woker进程的运行状态。 （4）当woker进程退出后（异常情况下），会自动重新启动新的woker进程。 woker进程主要用来处理网络事件，各个woker进程之间是对等且相互独立的，它们同等竞争来自客户端的请求，一个请求只可能在一个woker进程中处理，woker进程个数一般设置为机器CPU核数。

二、进程控制** 对Nginx进程的控制主要是通过master进程来做到的，主要有两种方式： （1）手动发送信号 从图1可以看出，master接收信号以管理众woker进程，那么，可以通过kill向master进程发送信号，比如kill -HUP pid用以通知Nginx从容重启。所谓从容重启就是不中断服务：master进程在接收到信号后，会先重新加载配置，然后再启动新进程开始接收新请求，并向所有老进程发送信号告知不

再接收新请求并在处理完所有未处理完的请求后自动退出。 （2）自动发送信号 可以通过带命令行参数启动新进程来发送信号给master进程，比如./nginx -s reload用以启动一个新的Nginx进程，而新进程在解析到reload参数后会向master进程发送信号（新进程会帮我们把手动发送信号中的动作自动完成）。当然也可以这样./nginx -s stop来停止Nginx。

三、网络事件 Nginx采用异步非阻塞的方式来处理网络事件，类似于Libevent，具体过程如下图：

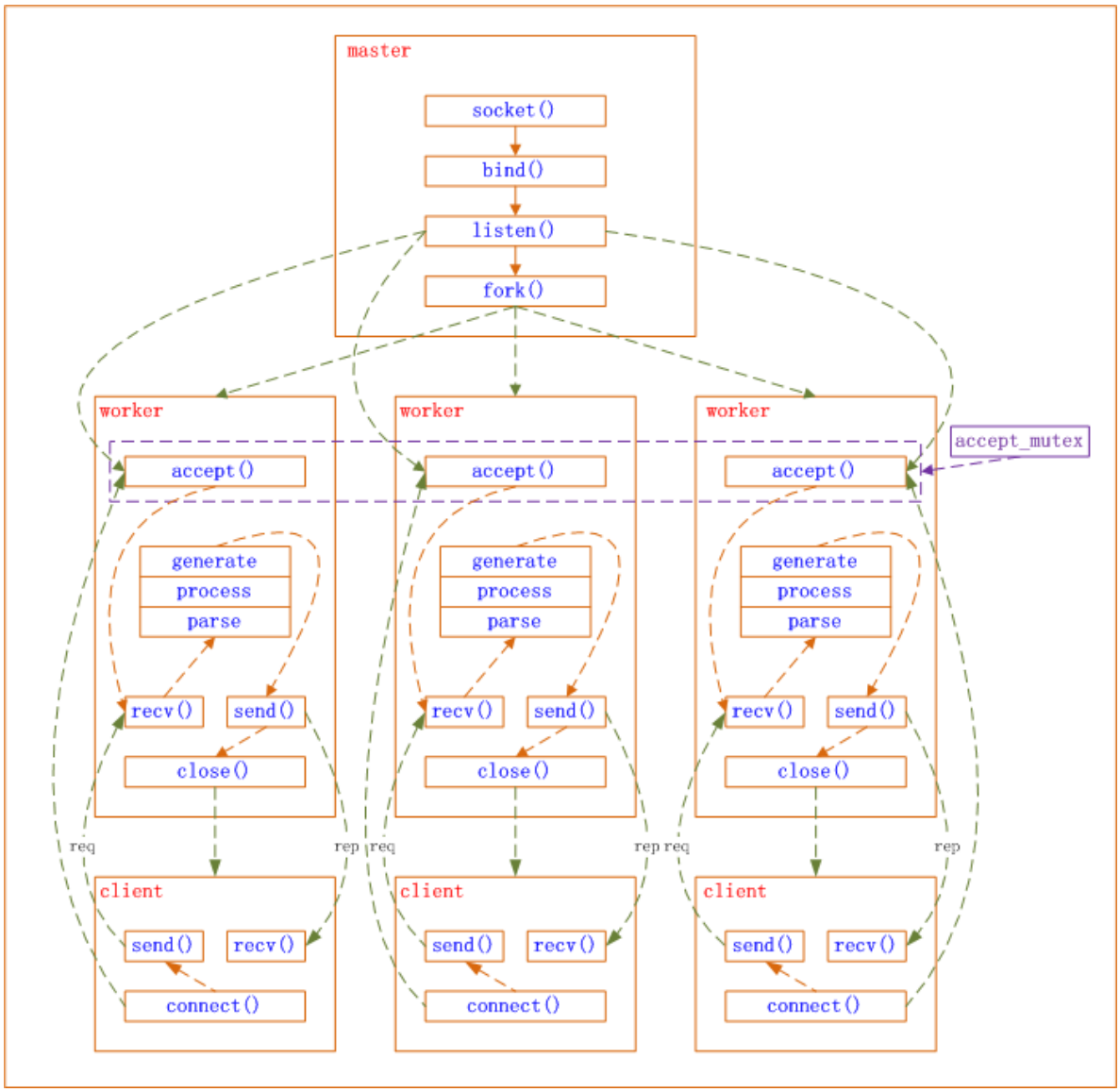


图2 Nginx网络事件

master进程先建好需要listen的socket后，然后再fork出多个woker进程，这样每个work进程都可以去accept这个socket。当一个client连接到来时，所有accept的work进程都会受到通知，但只有一个进程可以accept成功，其它的则会accept失败。Nginx提供了一把共享锁accept_mutex来保证同一时刻只有一个work进程在accept连接，从而解决惊群问题。当一个worker进程accept这个连接后，就开始读取请求，解析请求，处理请求，产生数据后，再返回给客户端，最后才断开连接，这样一个完成的请求就结束了。

Nginx 首先要明白，Nginx采用的是多进程（单线程）&多路IO复用模型。使用了I/O多路复用技术的Nginx，就成了”并发事件驱动“的服务器。

img

多进程的工作模式

1、Nginx在启动后，master进程fork()多个相互独立的worker进程。
2、接收来自外界的信号，然向各worker进程发送信号，每个进程都有可能来处理这个连接。
3、 master进程能监控worker进程的运行状态，当worker进程退出后(异常情况下)，会自动再fork()新worker进程。

注意worker进程数，一般会设置成机器cpu核数。因为更多的worker只会导致进程之间相互竞争cpu，从而带来不必要的上下文切换。

使用多进程模式，不仅能提高并发率，而且进程之间是相互独立的，一 个worker进程挂了不会影响到其他worker进程。

惊群现象 master进程首先通过socket()来创建一个监听描述符，然后fork()若干个worker，子进程将继承父进程的监听描述符，之后子进程在该监听描述符上accept()创建已连接描述符（connected descriptor），然后通过已连接描述符来与客户端通信。

那么，由于所有子进程都继承了父进程的 sockfd，那么当连接进来时，所有子进程都将收到通知并“争着”与它建立连接，这就叫“惊群现象”。大量的进程被激活又挂起，只有一个进程可以accept() 到这个连接，这当然会消耗系统资源。

Nginx对惊群现象的处理： Nginx提供了一个accept_mutex这个东西，这是一个加在accept上的一把互斥锁。即每个worker进程在执行accept()之前都需要先获取锁，accept()成功之后再解锁。有了这把锁，同一时刻，只会有一个进程执行accpet()，这样就不会有惊群问题了。accept_mutex是一个可控选项，我们可以显示地关掉，默认是打开的。

worker进程工作流程 当一个 worker 进程在 accept() 这个连接之后，就开始读取请求，解析请求，处理请求，产生数据后，再返回给客户端，最后才断开连接，一个完整的请求。一个请求，完全由worker进程来处理，而且只会在一个worker进程中处理。

这样做带来的好处：

1、节省锁带来的开销。每个worker进程都彼此独立地工作，不共享任何资源，因此不需要锁。同时在编程以及问题排查上时，也会方便很多。

2、独立进程，减少风险。采用独立的进程，可以让互相之间不会影响，一个进程退出后，其它进程还在工作，服务不会中断，master进程则很快重新启动新的worker进程。当然，worker进程自己也能发生意外退出。

核心：Nginx采用的 IO多路复用模型epoll 多路复用，允许我们只在事件发生时才将控制返回给程序，而其他时候内核都挂起进程，随时待命。

epoll通过在Linux内核中申请一个简易的文件系统（文件系统一般用B+树数据结构来实现），其工作流程分为三部分：

1、调用 int epoll_create(int size)建立一个epoll对象，内核会创建一个eventpoll结构体，用于存放通过epoll_ctl()向epoll对象中添加进来的事件，这些事件都会挂载在红黑树中。
2、调用 int epoll_ctl(int epfd, int op, int fd, struct epoll_event *event) 在 epoll 对象中为 fd 注册事件，所有添加到epoll中的事件都会与设备驱动程序建立回调关系，也就是说，当相应的事件发生时会调用这个sockfd的回调方法，将sockfd添加到eventpoll 中的双链表。
3、调用 int epoll_wait(int epfd, struct epoll_event * events, int maxevents, int timeout) 来等待事件的发生，timeout 为 -1 时，该调用会阻塞知道有事件发生

这样，注册好事件之后，只要有fd上事件发生，epoll_wait()就能检测到并返回给用户，用户执行阻塞函数时就不会发生阻塞了。

epoll()在中内核维护一个链表，epoll_wait直接检查链表是不是空就知道是否有文件描述符准备好了。顺便提一提，epoll与select、poll相比最大的优点是不会随着sockfd数目增长而降低效率，使用select()时，内核采用轮训的方法来查看是否有fd准备好，其中的保存sockfd的是类似数组的数据结构fd_set，key 为 fd，value为0或者1（发生时间）。

能达到这种效果，是因为在内核实现中epoll是根据每 sockfd 上面的与设备驱动程序建立起来的回调函数实现的。那么，某个sockfd上的事件发生时，与它对应的回调函数就会被调用，将这个sockfd加入链表，其他处于“空闲的”状态的则不会。在这点上，epoll 实现了一个"伪"AIO。

可以看出，因为一个进程里只有一个线程，所以一个进程同时只能做一件事，但是可以通过不断地切换来“同时”处理多个请求。

例子：Nginx 会注册一个事件：“如果来自一个新客户端的连接请求到来了，再通知我”，此后只有连接请求到来，服务器才会执行 accept() 来接收请求。又比如向上游服务器（比如 PHP-FPM）转发请求，并等待请求返回时，这个处理的 worker 不会在这阻塞，它会在发送完请求后，注册一个事件：“如果缓冲区接收到数据了，告诉我一声，我再将它读进来”，于是进程就空闲下来等待事件发生。

这样，基于 多进程+epoll， Nginx 便能实现高并发。

使用 epoll 处理事件的一个框架，代码转自：http://www.cnblogs.com/fnlingnzb-learner/p/5835573.html

```
for( ; ; )  //  无限循环
      {
          nfds = epoll_wait(epfd,events,20,500);  //  最长阻塞 500s
          for(i=0;i<nfds;++i)
          {
              if(events[i].data.fd==listenfd) //有新的连接
              {
                  connfd = accept(listenfd,(sockaddr *)&clientaddr, &clilen);
//accept这个连接
                  ev.data.fd=connfd;
                  ev.events=EPOLLIN|EPOLLET;
                  epoll_ctl(epfd,EPOLL_CTL_ADD,connfd,&ev); //将新的fd添加到epoll的
监听队列中
              }
              else if( events[i].events&EPOLLIN ) //接收到数据，读socket
              {
                  n = read(sockfd, line, MAXLINE)) < 0    //读
                  ev.data.ptr = md;     //md为自定义类型，添加数据
                  ev.events=EPOLLOUT|EPOLLET;
                  epoll_ctl(epfd,EPOLL_CTL_MOD,sockfd,&ev);//修改标识符，等待下一个循
环时发送数据，异步处理的精髓
              }
              else if(events[i].events&EPOLLOUT) //有数据待发送，写socket
              {
                  struct myepoll_data* md = (myepoll_data*)events[i].data.ptr;
 //取数据
                  sockfd = md->fd;
                  send( sockfd, md->ptr, strlen((char*)md->ptr), 0 );        //发
送数据
                  ev.data.fd=sockfd;
                  ev.events=EPOLLIN|EPOLLET;
                  epoll_ctl(epfd,EPOLL_CTL_MOD,sockfd,&ev); //修改标识符，等待下一个
循环时接收数据
              }
              else
              {
```

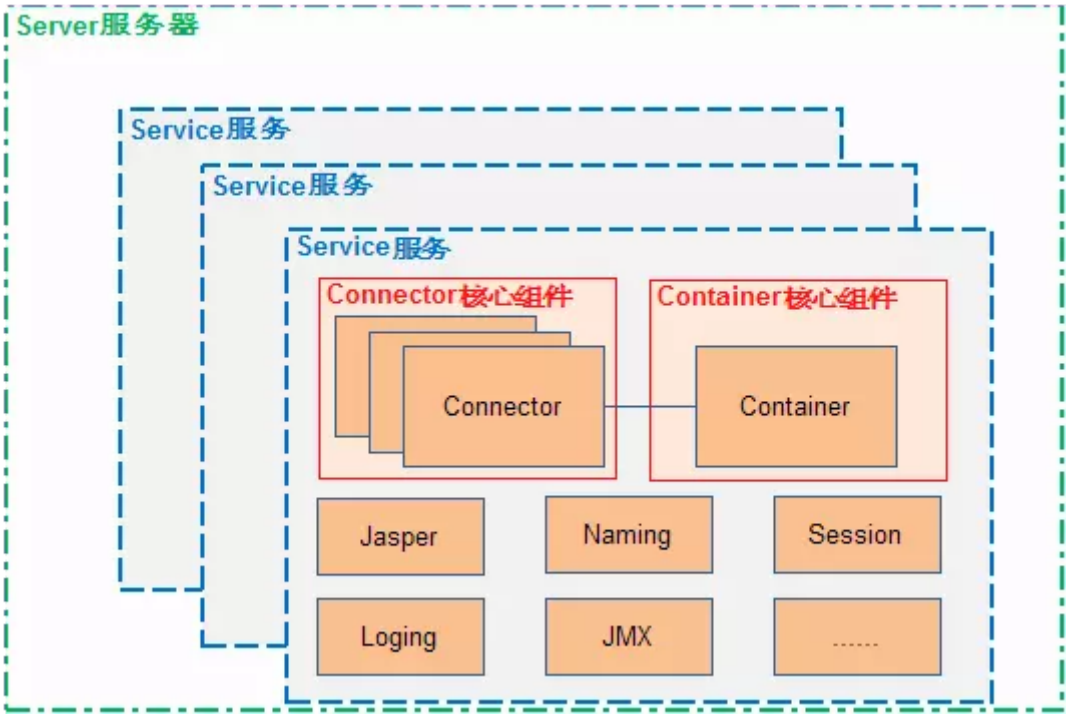
```
                    //其他的处理
                }
            }
        }
```

Nginx 与 多进程模式 Apache 的比较： 事件驱动适合于I/O密集型服务，多进程或线程适合于CPU密集型服务： 1、Nginx能够大量作为反向代理使用。 2、事件驱动服务器，最适合做的就是这种I/O密集型工作，如反向代理，它在客户端与WEB服务器之间起一个数据中转作用，纯粹是I/O操作，自身并不涉及到复杂计算。因为进程在一个地方进行计算时，那么这个进程就不能处理其他事件了。 3、Nginx只需要少量进程配合事件驱动，起几个进程跑libevent，不像 Apache传统多进程模型那样动辄数百的进程数。 4、Nginx处理静态文件效果也很好，那是因为读写文件和网络通信其实都是I/O操作，处理过程是一样的。

这里仅将Nginx与传统多进程工作模式下的Apache做比较，Apache也有事件驱动的工作模式。

为什么Netty也是一个线程一个EventLoop。也是异步事件驱动的。起到一个数据中转的作用。所以也是（网络）I/O密集型服务。所以很像。

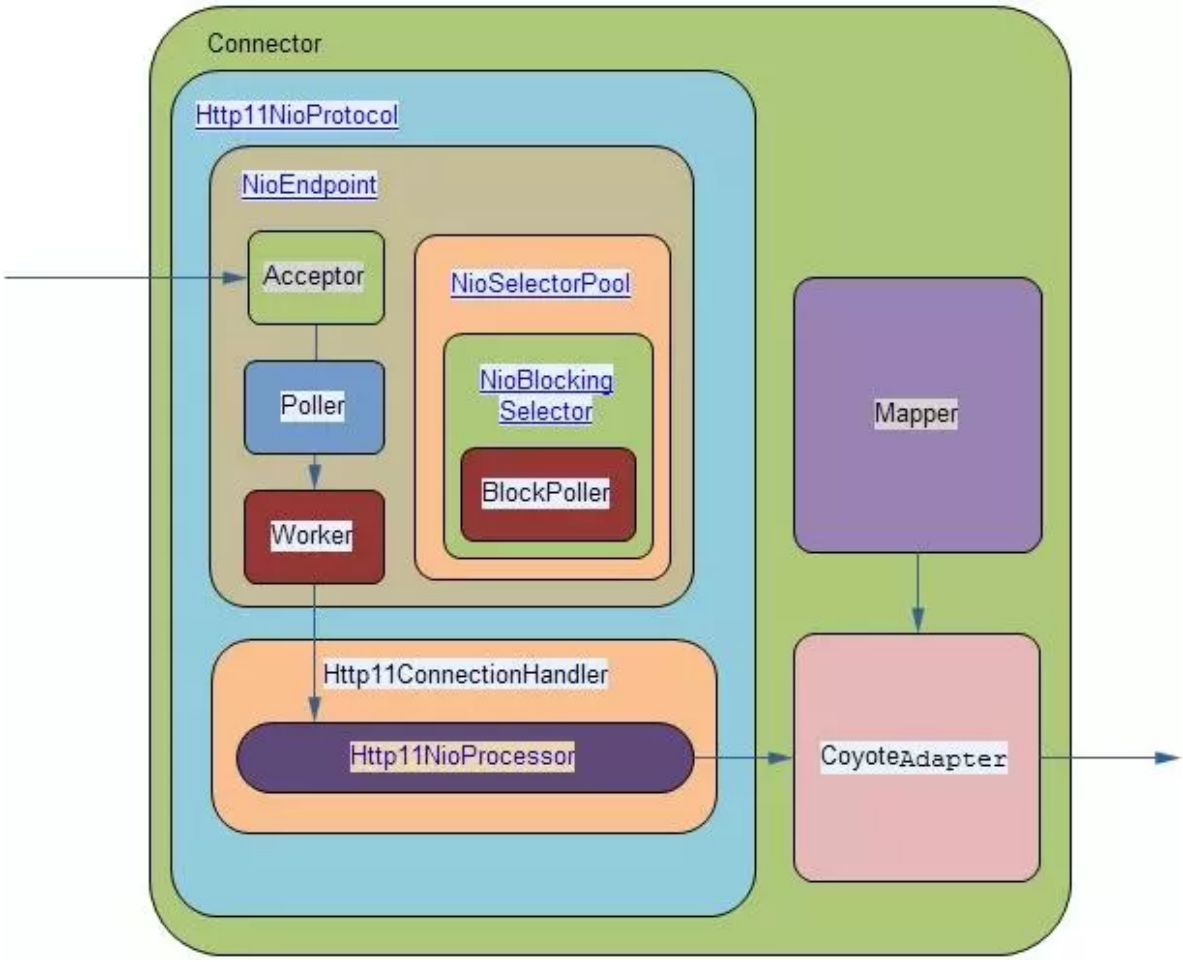
3、Tomcat总体架构



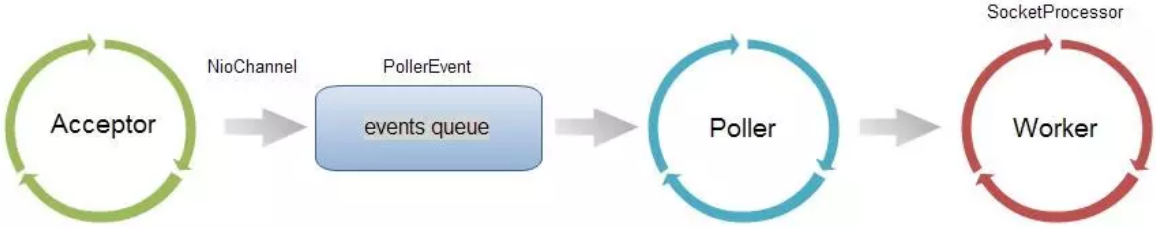
Tomcat有Connector和Container两大核心组件，Connector组件负责网络请求接入，Connector目前支持BIO、NIO、APR三种模式，后续文章会再重点对比下NIO和APR，Tomcat5以后的版本开始支持NIO了；Container组件实现了对servlet的容器管理功能；service服务将Connector和Container又包了一层，包装成外部可以获取的服务；多有service都运行在Tomcat这个大Server服务上，Server有所有service的实例，并实现了LifeCycle接口可以控制所有service的生命周期。

2.Tomcat NIO相关类

Tomcat的NIO实现主要是在Connector组件内，Connector 组件是 Tomcat 中两个核心组件之一，它的主要任务是负责接收浏览器的发过来的 tcp 连接请求，创建一个 Request 和 Response 对象分别用于和请求端交换数据，然后会产生一个线程来处理这个请求并把产生的 Request 和 Response 对象传给处理这个请求的线程，处理这个请求的线程就是 Container 组件要做的事了。



整个Connector组件包含三部分：Http11NioProtocol、Mapper、CoyoteAdapter。
Http11NioProtocol包含NioEndpoint和Http11ConnectionHandler，NioEndpoint是Http11NioProtocol中负责接收处理socket的主要模块；Http11ConnectionHandler是连接处理器。
NioEndpoint主要是实现了socket请求监听线程Acceptor、socket NIO poller线程、以及请求处理线程池。 NioEndpoint的内部处理流程为：



Acceptor 接收socket线程，这里虽然是基于NIO的connector，但是在接收socket方面还是传统的serverSocket.accept()方式，获得SocketChannel对象，然后封装在一个tomcat的实现类org.apache.tomcat.util.net.NioChannel对象中。然后将NioChannel对象封装在一个PollerEvent对象中，并将PollerEvent对象压入events queue里。这里是个典型的生产者-消费者模式，Acceptor与Poller线程之间通过queue通信，Acceptor是events queue的生产者，Poller是events queue的消费者。 **Poller** Poller线程中维护了一个Selector对象，NIO就是基于Selector来完成逻辑的。在connector中并不止一个Selector，在socket的读写数据时，为了控制timeout也有一个Selector，在后面的BlockSelector中介绍。可以先把Poller线程中维护的这个Selector标为主Selector。 Poller是NIO实现的主要线程。首先作为events queue的消费者，从queue中取出PollerEvent对象，然后将此对象中的channel以OP_READ事件注册到主Selector中，然后主Selector执行select操作，遍历出可以读数据的socket，并从Worker线程池中拿到可用的Worker线程，然后将socket传递给Worker。整个过程是典型的NIO实现。 **Worker** Worker线程拿到Poller传过来的socket后，将socket封装在SocketProcessor对象中。然后从Http11ConnectionHandler中取出Http11NioProcessor对象，从Http11NioProcessor中调用CoyoteAdapter的逻辑，跟BIO实现一样。在Worker线程中，会完成从socket中读取http request，解析成HttpServletRequest对象，分派到相应的servlet并完成逻辑，然后将response通过socket发回client。在从socket中读数据和往socket中写数据的过程，并没有像典型的非阻塞的NIO的那样，注册OP_READ或OP_WRITE事件到主Selector，而是直接通过socket完成读写，这时是阻塞完成的，但是在timeout控制上，使用了NIO的Selector机制，但是这个Selector并不是Poller线程维护的主Selector，而是BlockPoller线程中维护的Selector，称之为辅Selector。 **NioSelectorPool** NioEndpoint对象中维护了一个NioSelecPool对象，这个NioSelectorPool中又维护了一个BlockPoller线程，这个线程就是基于辅Selector进行NIO的逻辑。以执行servlet后，得到response，往socket中写数据为例，最终写的过程调用NioBlockingSelector的write方法。

3.请求处理流程

上面介绍了Tomcat的总体架构和涉及到NIO的相关工作类，下面从一个网络请求到Tomcat处理的过程直到业务servlet处理的过程，整体上说下一个网络请求的处理流程，下面借用网上的一张流程图，如果图片作者看到觉得侵权请下面留言，马上删掉：）

 1563204859539

对于Acceptor监听到的Socket请求，经过NioEndpoint内部的NIO 线程模型处理后，会转变为SocketProcessor在Executor中运行，其在Run过程中会交给Http11ConnectionHandler处理，Http11ConnectionHandler会从ConcurrentHashMap<NioChannel,Http11NioProcessor>缓存中获取相应的Http11NioProcessor来继续处理，Http11NioProcessor主要是负责解析socket请求Header，解析完成后，会将Request、Response（这里的请求、响应在tomcat中看成是coyote的请求、响应，意思是还需要CoyoteAdaper处理）交给CoyoteAdaper继续处理，CoyoteAdaper这里的工作主要将socket解析的Request、Response转化为HttpServletRequest、HttpServletResponse，而这里的请求响应就是最后交给Container去处理。 同时我们可以看到Acceptor线程会将接受到的SocketChannel（一个socket请求）封装为PollerEvent放到Poller线程中的ConcurrentLinkedQueue缓存中，注意到这里的缓存是ConcurrentLinkedQueue是支持并发的，那么在Poller线程的内部，它只需要从这个缓存中不停地获取PollerEvent然后处理就可以了。最后Poller线程处理完成后会封装成SocketProcessor交给NioEndpoint内的线程池Executor去处理。线程池中的Work thread线程在处理SocketProcessor过程中，会调用Http11ConnectionHandler处理，而Http11ConnectionHandler则从ConcurrentHashMap<NioChannel,Http11NioProcessor>缓存中获取相应的Http11NioProcessor来继续处理，这里要注意的ConcurrentHashMap也是支持并发的。

4.NIO相关参数

一个或多个Acceptor线程，每个线程都有自己的Selector，Acceptor只负责accept新的连接，一旦连接建立之后就将连接注册到其他Worker线程中

多个Worker线程，有时候也叫IO线程，就是专门负责IO读写的。一种实现方式就是像Netty一样，每个Worker线程都有自己的Selector，可以负责多个连接的IO读写事件，每个连接归属于某个线程。另一种方式实现方式就是有专门的线程负责IO事件监听，这些线程有自己的Selector，一旦监听到有IO读写事件，并不是像第一种实现方式那样（自己去执行IO操作），而是将IO操作封装成一个Runnable交给Worker线程池来执行，这种情况每个连接可能会被多个线程同时操作，相比第一种并发性提高了，但是也可能引来多线程问题，在处理上要更加谨慎些。tomcat的NIO模型就是第二种。

所以一般参数就是Acceptor线程个数，Worker线程个数。 参考官方文档 https://tomcat.apache.org/tomcat-8.5-doc/config/http.html?spm=5176.100239.blogcont39093.5.Vomyf0 参数主要有以下几个： 1）acceptCount 连接在被ServerSocketChannel accept之前就暂存在这个队列中，acceptCount就是这个队列的最大长度。ServerSocketChannel accept就是从这个队列中不断取出已经建立连接的的请求。所以当ServerSocketChannel accept取出不及时就有可能造成该队列积压，一旦满了连接就被拒绝了； 2）acceptorThreadCount Acceptor线程只负责从上述队列中取出已经建立连接的请求。在启动的时候使用一个ServerSocketChannel监听一个连接端口如8080，可以有多个Acceptor线程并发不断调用上述ServerSocketChannel的accept方法来获取新的连接。参数acceptorThreadCount其实使用的Acceptor线程的个数;

1. maxConnections 这里就是tomcat对于连接数的一个控制，即最大连接数限制。一旦发现当前连接数已经超过了一定的数量（NIO默认是10000），上述的Acceptor线程就被阻塞了，即不再执行ServerSocketChannel的accept方法从队列中获取已经建立的连接。但是它并不阻止新的连接的建立，新的连接的建立过程不是Acceptor控制的，Acceptor仅仅是从队列中获取新建立的连接。所以当连接数已经超过maxConnections后，仍然是可以建立新的连接的，存放在上述acceptCount大小的队列中，这个队列里面的连接没有被Acceptor获取，就处于连接建立了但是不被处理的状态。当连接数低于maxConnections之后，Acceptor线程就不再阻塞，继续调用ServerSocketChannel的accept方法从acceptCount大小的队列中继续获取新的连接，之后就开始处理这些新的连接的IO事件了;
2. maxThread 专门处理IO的Worker数，默认是200;