一篇讲透全网最高频的Java NIO面试考点汇总。

Java面试那些事儿 Today

Reposted from Official Account 石杉的架构笔记，Author 胡宜宁

首先我们分别画图来看看，BIO、NIO、AIO，分别是什么?

BIO：传统的网络通讯模型，就是BIO，同步阻塞IO

它其实就是服务端创建一个ServerSocket， 然后就是客户端用一个Socket去连接服务端的那个ServerSocket， ServerSocket接收到了一个的连接请求就创建一个Socket和一个线程去跟那个Socket进行通讯。

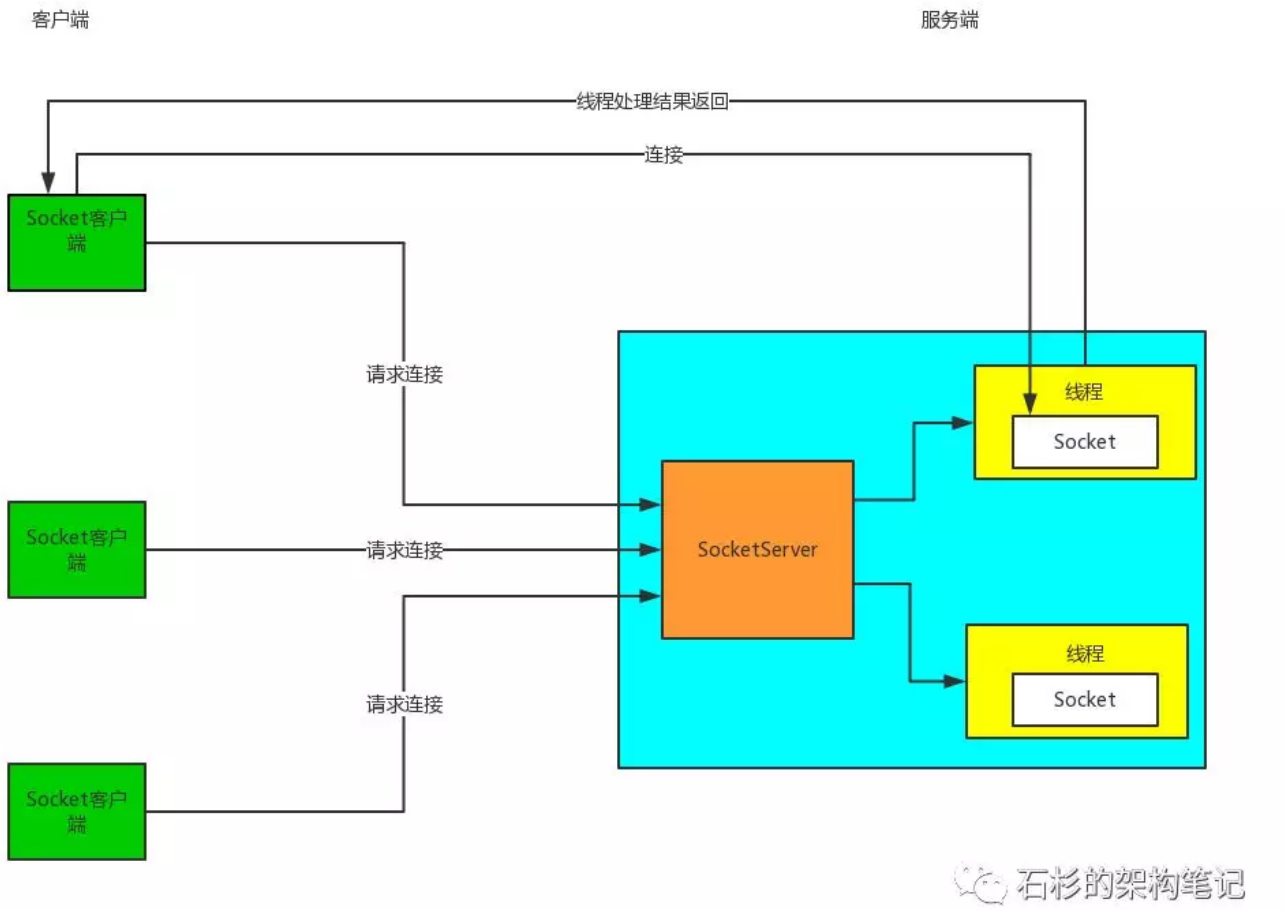
接着客户端和服务端就进行阻塞式的通信，客户端发送一个请求，服务端Socket进行处理后返回响应。

在响应返回前，客户端那边就阻塞等待，上门事情也做不了。

这种方式的缺点：每次一个客户端接入，都需要在服务端创建一个线程来服务这个客户端

这样大量客户端来的时候，就会造成服务端的线程数量可能达到了几千甚至几万，这样就可能会造成服务端过载过高，最后崩溃死掉。

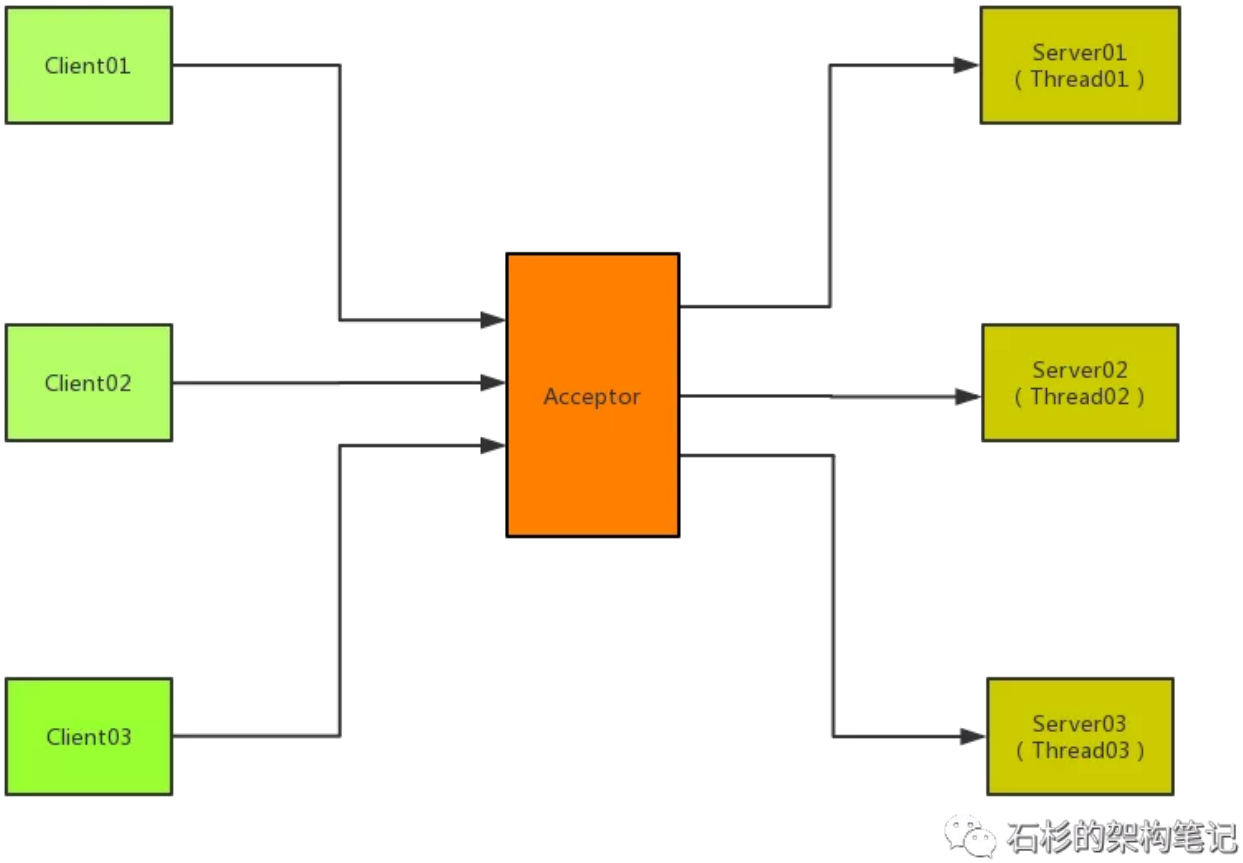
BIO模型图:



Acceptor:

传统的IO模型的网络服务的设计模式中有俩种比较经典的设计模式：一个是多线程， 一种是依靠线程池来进行处理。

如果是基于多线程的模式来的话，就是这样的模式，这种也是Acceptor线程模型。



NIO：

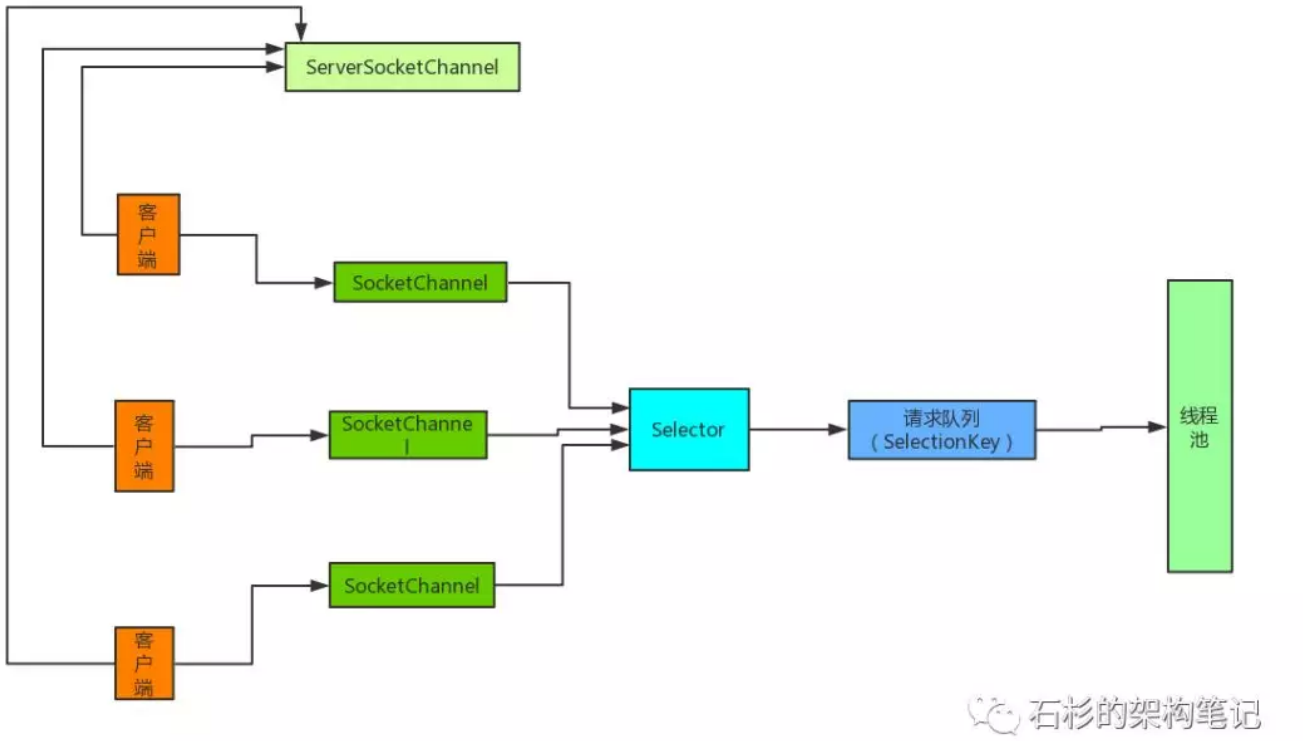
NIO是一种同步非阻塞IO，基于Reactor模型来实现的。

其实相当于就是一个线程处理大量的客户端的请求，通过一个线程轮询大量的channel，每次就获取一批有事件的channel，然后对每个请求启动一个线程处理即可。

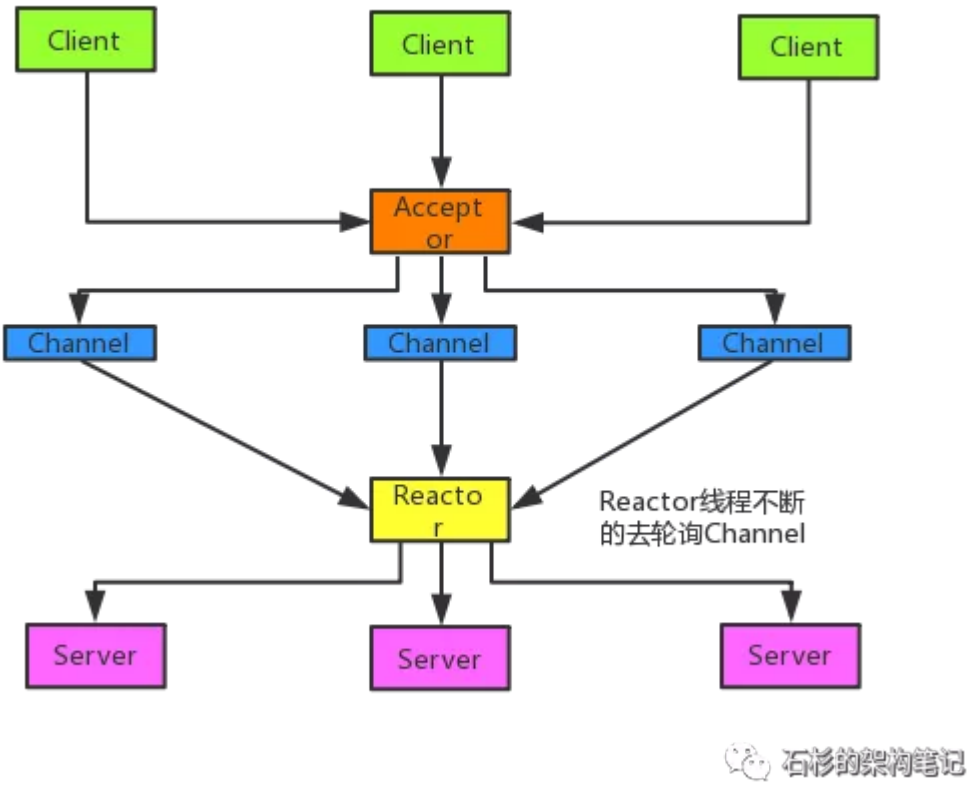
这里的核心就是非阻塞，就那个selector一个线程就可以不停轮询channel，所有客户端请求都不会阻塞，直接就会进来，大不了就是等待一下排着队而已。

这里面**优化BIO的核心**就是，一个客户端并不是时时刻刻都有数据进行交互，没有必要死耗着一个线程不放，所以客户端选择了让线程歇一歇，只有客户端有相应的操作的时候才发起通知，创建一个线程来处理请求。

NIO：模型图


ServerSocketChannel
客户端
SocketChannel
客户端
SocketChannel
Selector
请求队列
(SelectionKey)
线程
池
SocketChannel
客户端
石杉的架构笔记

Reactor模型:


Client
Client
Client
Accept
or
Channel
Channel
Channel
Reacto
r
Reactor线程不断
的去轮询Channel
Server
Server
Server
石杉的架构笔记

AIO

AIO：异步非阻塞IO，基于Proactor模型实现。

每个连接发送过来的请求，都会绑定一个Buffer，然后通知操作系统去完成异步的读，这个时间你就可以去做其他的事情

等到操作系统完成读之后，就会调用你的接口，给你操作系统异步读完的数据。这个时候你就可以拿到数据进行处理，将数据往回写

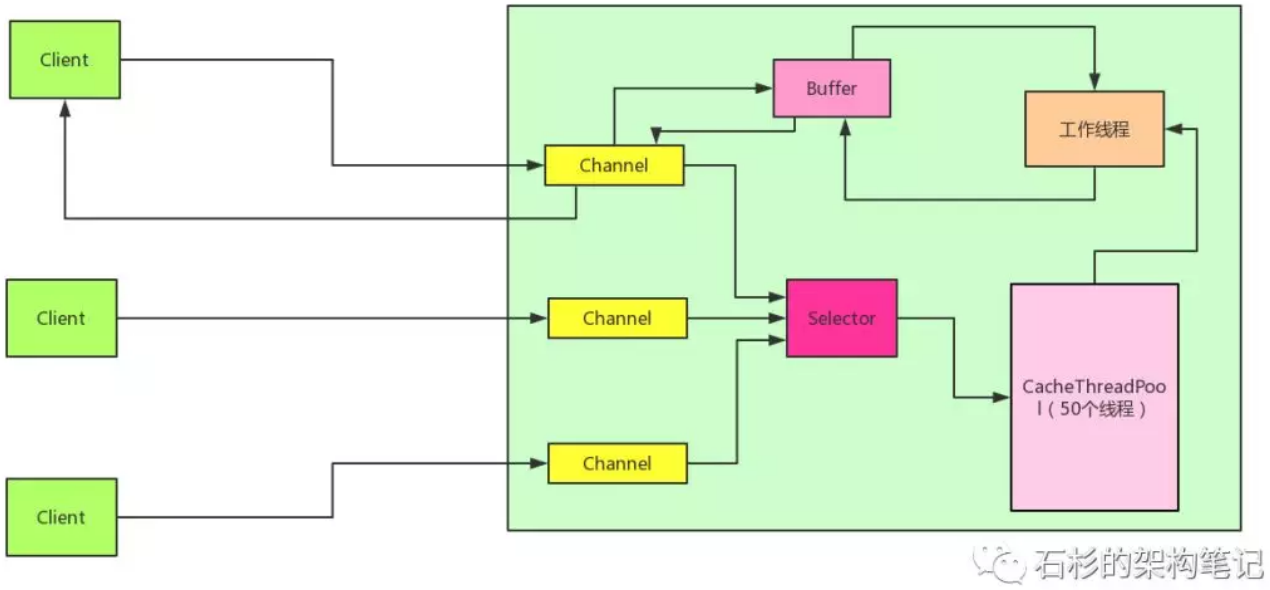
在往回写的过程，同样是给操作系统一个Buffer，让操作系统去完成写，写完了来通知你。

这俩个过程都有buffer存在，数据都是通过buffer来完成读写。

这里面的主要的区别在于将数据写入的缓冲区后，就不去管它，剩下的去交给操作系统去完成。

操作系统写回数据也是一样，写到Buffer里面，写完后通知客户端来进行读取数据。

AIO：模型图



聊完了BIO，NIO，AIO的区别之后，现在我们再结合这三个模型来说下同步和阻塞的一些问题。

同步阻塞

为什么说BIO是同步阻塞的呢？

其实这里说的不是针对网络通讯模型而言，而是针对磁盘文件读写IO操作来说的。

因为用BIO的流读写文件，例如FileInputStrem，是说你发起个IO请求直接hang死，卡在那里，必须等着搞完了这次IO才能返回。

同步非阻塞：

为什么说NIO为啥是同步非阻塞？

因为无论多少客户端都可以接入服务端，客户端接入并不会耗费一个线程，只会创建一个连接然后注册到selector上去，这样你就可以去干其他你想干的其他事情了

一个selector线程不断的轮询所有的socket连接，发现有事件了就通知你，然后你就启动一个线程处理一个请求即可，这个过程的话就是非阻塞的。

但是这个处理的过程中，你还是要先读取数据，处理，再返回的，这是个同步的过程。

异步非阻塞

为什么说AIO是异步非阻塞？

通过AIO发起个文件IO操作之后，你立马就返回可以干别的事儿了，接下来你也不用管了，操作系统自己干完了IO之后，告诉你说ok了

当你基于AIO的api去读写文件时， 当你发起一个请求之后，剩下的事情就是交给了操作系统

当读写完成后， 操作系统会来回调你的接口， 告诉你操作完成

在这期间不需要等待， 也不需要去轮询判断操作系统完成的状态，你可以去干其他的事情。

同步就是自己还得主动去轮询操作系统，异步就是操作系统反过来通知你。所以来说， AIO就是异步非阻塞的。

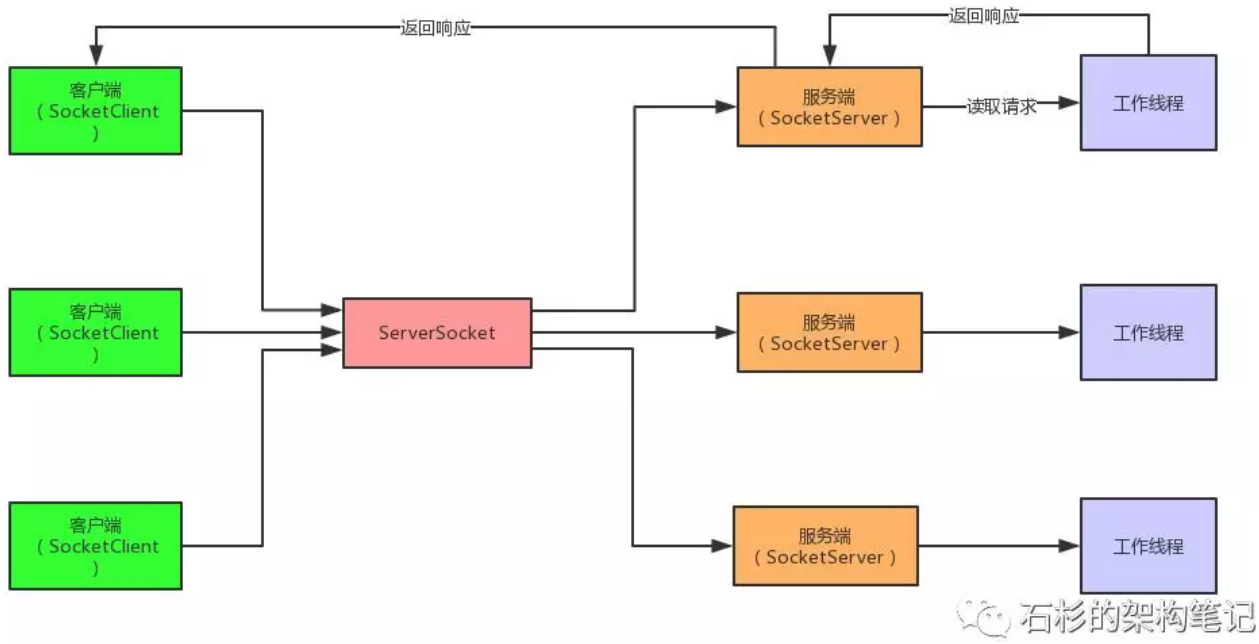
NIO核心组件详细讲解

学习NIO先来搞清楚一些相关的概念，NIO通讯有哪些相关组件，对应的作用都是什么，之间有哪些联系？

多路复用机制实现Selector

首先我们来了解下传统的Socket网络通讯模型。

传统Socket通讯原理图



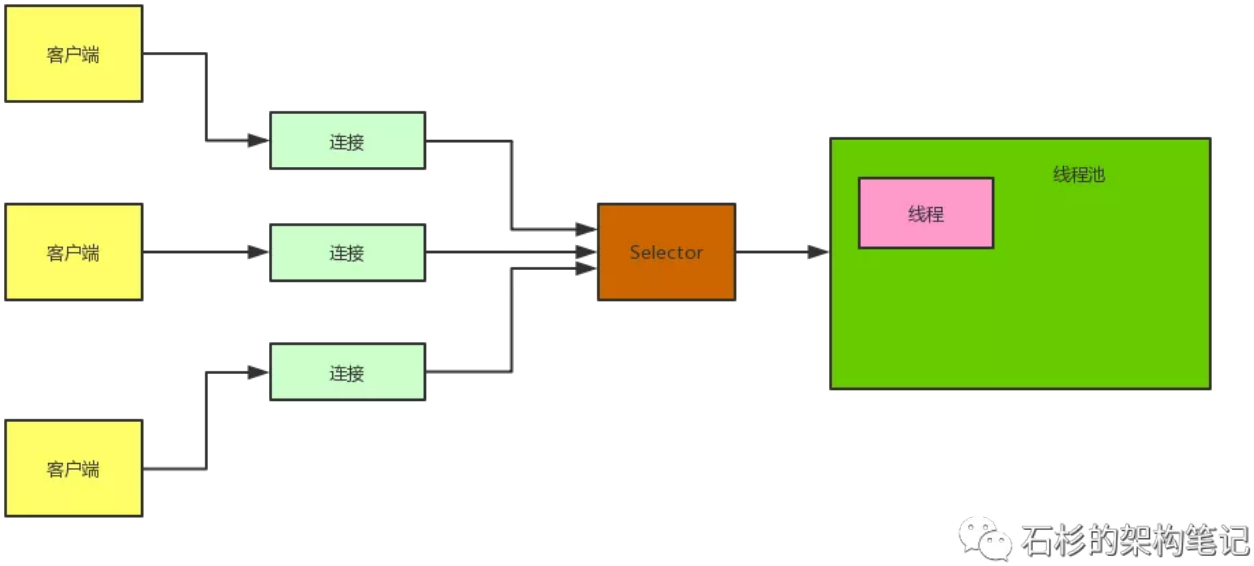
为什么传统的socket不支持海量连接？

每次一个客户端接入，都是要在服务端创建一个线程来服务这个客户端的

这会导致大量的客户端的时候，服务端的线程数量可能达到几千甚至几万，几十万，这会导致服务器端程序负载过高，不堪重负，最终系统崩溃死掉。

接着来看下NIO是如何基于Selector实现多路复用机制支持的海量连接。

NIO原理图



多路复用机制是如何支持海量连接?

NIO的线程模型对Socket发起的连接不需要每个都创建一个线程，完全可以使用一个Selector来多路复用监听N多个Channel是否有请求，该请求是对应的连接请求，还是发送数据的请求

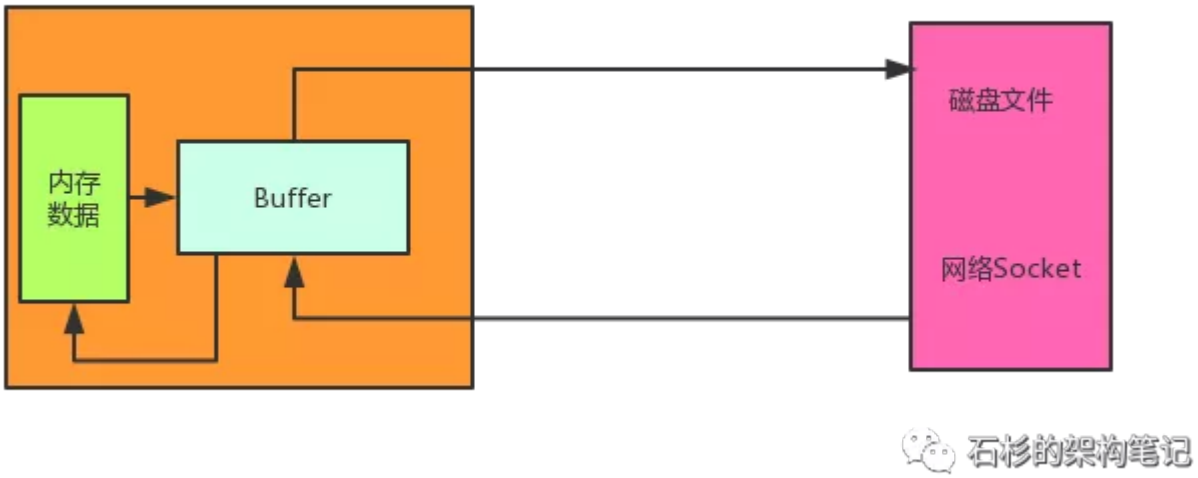
这里面是基于操作系统底层的Select通知机制的，一个Selector不断的轮询多个Channel，这样避免了创建多个线程

只有当莫个Channel有对应的请求的时候才会创建线程，可能说1000个请求， 只有100个请求是有数据交互的

这个时候可能server端就提供10个线程就能够处理这些请求。这样的话就可以避免了创建大量的线程。

NIO如何通过Buffer来缓冲数据的

NIO中的Buffer是个什么东西 ？

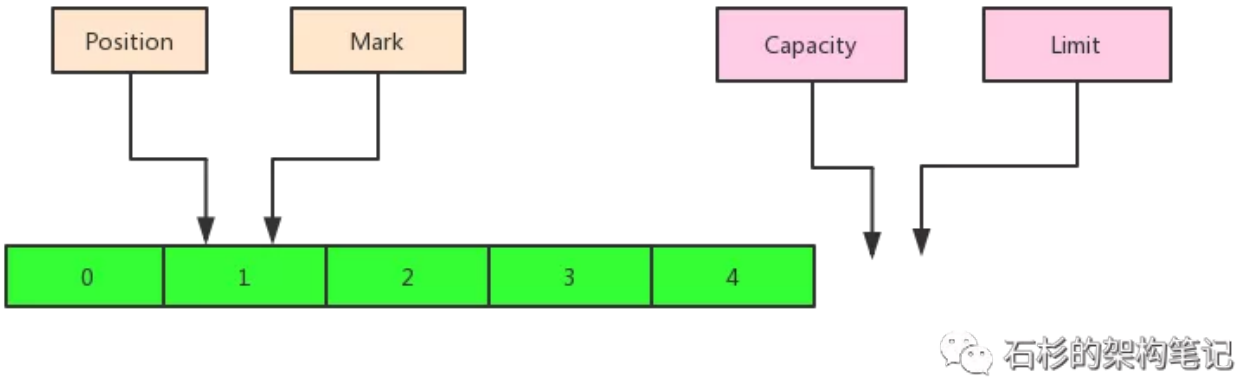


学习NIO，首当其冲就是要了解所谓的Buffer缓冲区，这个东西是NIO里比较核心的一个部分

一般来说，如果你要通过NIO写数据到文件或者网络，或者是从文件和网络读取数据出来此时就需要通过Buffer缓冲区来进行。Buffer的使用一般有如下几个步骤：

写入数据到Buffer，调用flip()方法，从Buffer中读取数据，调用clear()方法或者compact()方法。

Buffer中对应的Position， Mark， Capacity，Limit都啥？



- **capacity**：缓冲区容量的大小，就是里面包含的数据大小。
- **limit**：对buffer缓冲区使用的一个限制，从这个index开始就不能读取数据了。

- **position**：代表着数组中可以开始读写的index， 不能大于limit。
- **mark**：是类似路标的东西，在某个position的时候，设置一下mark，此时就可以设置一个标记

  后续调用reset()方法可以把position复位到当时设置的那个mark上。去把position或limit调整为小于mark的值时，就丢弃这个mark

  如果使用的是Direct模式创建的Buffer的话，就会减少中间缓冲直接使用DirectorBuffer来进行数据的存储。

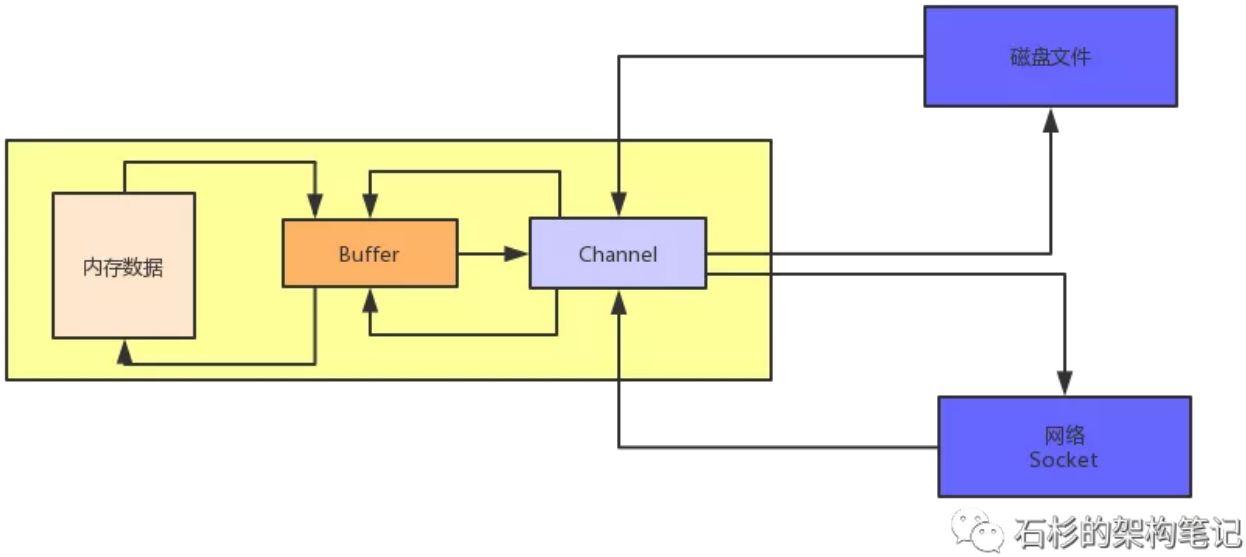
如何通过Channel和FileChannel读取Buffer数据写入磁盘的

NIO中，Channel是什么?

Channel是NIO中的数据通道，类似流，但是又有些不同

Channel既可从中读取数据，又可以从写数据到通道中，但是流的读写通常是单向的。

Channel可以异步的读写。Channel中的数据总是要先读到一个Buffer中，或者从缓冲区中将数据写到通道中。



FileChannel的作用是什么?

Buffer有不同的类型，同样Channel也有好几个类型。

- FileChannel
- DatagramChannel
- SocketChannel
- ServerSocketChannel

这些通道涵盖了UDP 和 TCP 网络IO，以及文件IO。而FileChannel就是文件IO对应的管道， 在读取文件的时候会用到这个管道。

下面给一个简单的NIO实现读取文件的Demo代码

```
public class FileChannelDemo1 {

  public static void main(String[] args) throws Exception {
    // 构造一个传统的文件输出流
    FileOutputStream out = new FileOutputStream(
        "F:\development\tmp\hello.txt");
    // 通过文件输出流获取到对应的FileChannel，以NIO的方式来写文件
    FileChannel channel = out.getChannel();
     // 将数据写入到Buffer中
```

```
    ByteBuffer buffer = ByteBuffer.wrap("hello world".getBytes());
    // 通过FileChannel管道将Buffer中的数据写到输出流中去，持久化到磁盘中去
    channel.write(buffer);

    channel.close();
    out.close();
  }
  }
```

NIOServer端和Client端代码案例

最后，给大家一个NIO客户端和服务端示例代码，简单感受下NIO通讯的方式。

- NIO通讯Client端

```
import java.io.IOException;
import java.net.InetSocketAddress;
import java.nio.ByteBuffer;
import java.nio.channels.SelectionKey;
import java.nio.channels.Selector;
import java.nio.channels.SocketChannel;
import java.util.Iterator;

public class NIOClient {

    public static void main(String[] args) {
        for(int i = 0; i < 10; i++){
            new Worker().start();
        }
    }

    static class Worker extends Thread {

     @Override
     public void run() {
       SocketChannel channel = null;
       Selector selector = null;
       try {
           // SocketChannel，一看底层就是封装了一个Socket
         channel = SocketChannel.open();          // SocketChannel是连接到底层的Socket网
         // 数据通道就是负责基于网络读写数据的
         channel.configureBlocking(false);
         channel.connect(new InetSocketAddress("localhost", 9000));
```

```
    // 后台一定是tcp三次握手建立网络连接

selector = Selector.open();
//  监听Connect这个行为
channel.register(selector, SelectionKey.OP_CONNECT);
while(true){
    // selector多路复用机制的实现  循环去遍历各个注册的Channel
  selector.select();
  Iterator<SelectionKey> keysIterator = selector.selectedKeys().iterator();
  while(keysIterator.hasNext()){
    SelectionKey key = (SelectionKey) keysIterator.next();
    keysIterator.remove();
    // 如果发现返回的时候一个可连接的消息 走到下面去接受数据
    if(key.isConnectable()){                channel = (SocketChannel) key.c
      if(channel.isConnectionPending()){
        channel.finishConnect();
        // 接下来对这个SocketChannel感兴趣的就是人家server给你发送过来的数据了
        // READ事件，就是可以读数据的事件
        // 一旦建立连接成功了以后，此时就可以给server发送一个请求了
        ByteBuffer buffer = ByteBuffer.allocate(1024);
        buffer.put("你好".getBytes());
        buffer.flip();
        channel.write(buffer);
      }

      channel.register(selector, SelectionKey.OP_READ);
    }
    // 这里的话就时候名服务器端返回了一条数据可以读了
    else if(key.isReadable()){              channel = (SocketChannel) key.c

      // 构建一个缓冲区
      ByteBuffer buffer = ByteBuffer.allocate(1024);
      // 把数据写入buffer，position推进到读取的字节数数字
      int len = channel.read(buffer);
      if(len > 0) {
        System.out.println("[" + Thread.currentThread().getName()
            + "]收到响应: " + new String(buffer.array(), 0, len));
        Thread.sleep(5000);
        channel.register(selector, SelectionKey.OP_WRITE);
      }
    } else if(key.isWritable()) {
          ByteBuffer buffer = ByteBuffer.allocate(1024);
          buffer.put("你好".getBytes());
          buffer.flip();

          channel = (SocketChannel) key.channel();
          channel.write(buffer);
          channel.register(selector, SelectionKey.OP_READ);
        }
  }
```

```
        }
      } catch (Exception e) {
        e.printStackTrace();
      } finally{
        if(channel != null){
          try {
            channel.close();
          } catch (IOException e) {
            e.printStackTrace();
          }
        }

        if(selector != null){
          try {
            selector.close();
          } catch (IOException e) {
            e.printStackTrace();
          }
        }
      }
    }

  }

}
```

- NIO通讯Server端

```
import java.io.IOException;
import java.net.InetSocketAddress;
import java.nio.ByteBuffer;
import java.nio.channels.ClosedChannelException;
import java.nio.channels.SelectionKey;
import java.nio.channels.Selector;
import java.nio.channels.ServerSocketChannel;
import java.nio.channels.SocketChannel;
import java.util.Iterator;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.LinkedBlockingQueue;

public class NIOServer {

    private static Selector selector;
    private static LinkedBlockingQueue<SelectionKey> requestQueue;
    private static ExecutorService threadPool;

    public static void main(String[] args) {
```

```
        init();
        listen();
    }

    private static void init(){
        ServerSocketChannel serverSocketChannel = null;

        try {
            selector = Selector.open();

            serverSocketChannel = ServerSocketChannel.open();
            // 将Channel设置为非阻塞的 NIO就是支持非阻塞的
            serverSocketChannel.configureBlocking(false);          serverSocketChannel
             // ServerSocket，就是负责去跟各个客户端连接连接请求的
              serverSocketChannel.register(selector, SelectionKey.OP_ACCEPT);
              // 就是仅仅关注这个ServerSocketChannel接收到的TCP连接的请求
        } catch (IOException e) {
            e.printStackTrace();
        }

        requestQueue = new LinkedBlockingQueue<SelectionKey>(500);

        threadPool = Executors.newFixedThreadPool(10);
        for(int i = 0; i < 10; i++) {
            threadPool.submit(new Worker());
        }
    }

    private static void listen() {
        while(true){
            try{
                selector.select();
                Iterator<SelectionKey> keysIterator = selector.selectedKeys().itera

                while(keysIterator.hasNext()){
                    SelectionKey key = (SelectionKey) keysIterator.next();
                    // 可以认为一个SelectionKey是代表了一个请求
                    keysIterator.remove();
                    handleRequest(key);
                }
            }
            catch(Throwable t){
                t.printStackTrace();
            }
        }
    }

    private static void handleRequest(SelectionKey key)
            throws IOException, ClosedChannelException {
        // 后台的线程池中的线程处理下面的代码逻辑
        SocketChannel channel = null;
```

```
        try{
            // 如果说这个Key是一个acceptable,也就是一个连接请求
            if(key.isAcceptable()){
                ServerSocketChannel serverSocketChannel = (ServerSocketChannel) key
                // 调用accept这个方法 就可以进行TCP三次握手了
                channel = serverSocketChannel.accept();
                // 握手成功的话就可以获取到一个TCP连接好的SocketChannel
                channel.configureBlocking(false);
                channel.register(selector, SelectionKey.OP_READ);
                // 仅仅关注这个READ请求，就是人家发送数据过来的请求
            }
            // 如果说这个key是readable，是个发送了数据过来的话，此时需要读取客户端发送过来的数据
            else if(key.isReadable()){
                channel = (SocketChannel) key.channel();
                ByteBuffer buffer = ByteBuffer.allocate(1024);
                int count = channel.read(buffer);
                 // 通过底层的socket读取数据，写buffer中，position可能就会变成21之类的
                // 你读取到了多少个字节，此时buffer的position就会变成多少

                if(count > 0){
                 // 准备读取刚写入的数据，就是将limit设置为当前position，将position设置为0，
                  // position = 0，limit = 21，仅仅读取buffer中，0~21这段刚刚写入进去的数据
                  buffer.flip();
                            System.out.println("服务端接收请求: " + new String(buffer
                    channel.register(selector, SelectionKey.OP_WRITE);
                }
            } else if(key.isWritable()) {
              ByteBuffer buffer = ByteBuffer.allocate(1024);
              buffer.put("收到".getBytes());
              buffer.flip();

              channel = (SocketChannel) key.channel();
              channel.write(buffer);
              channel.register(selector, SelectionKey.OP_READ);
            }
        }
        catch(Throwable t){
            t.printStackTrace();
            if(channel != null){
                channel.close();
            }
        }
    }

// 创建一个线程任务来执行
static class Worker implements Runnable {

    @Override
    public void run() {
      while(true) {
```

```
        try {
          SelectionKey key = requestQueue.take();
          handleRequest(key);
        } catch (Exception e) {
          e.printStackTrace();
        }
      }
    }

    private void handleRequest(SelectionKey key)
              throws IOException, ClosedChannelException {
        // 假设想象一下，后台有个线程池获取到了请求
        // 下面的代码，都是在后台线程池的工作线程里在处理和执行
          SocketChannel channel = null;

          try{
              // 如果说这个key是个acceptable，是个连接请求的话
              if(key.isAcceptable()){                    System.out.println("[" + Threa
                  ServerSocketChannel serverSocketChannel = (ServerSocketChannel) k
                  // 调用accept方法 和客户端进行三次握手
                  channel = serverSocketChannel.accept();                    System.
                  // 如果三次握手成功了之后，就可以获取到一个建立好TCP连接的SocketChannel
                  // 这个SocketChannel大概可以理解为，底层有一个Socket，是跟客户端进行连接的
                  // 你的SocketChannel就是联通到那个Socket上去，负责进行网络数据的读写的
                  // 设置为非阻塞的
                  channel.configureBlocking(false);
                  // 关注的是Reade请求
                  channel.register(selector, SelectionKey.OP_READ);                }

                   // 如果说这个key是readable，是个发送了数据过来的话，此时需要读取客户端发送过
              else if(key.isReadable()){
                  channel = (SocketChannel) key.channel();
                  ByteBuffer buffer = ByteBuffer.allocate(1024);
                  int count = channel.read(buffer);
                   // 通过底层的socket读取数据，写入buffer中，position可能就会变成21之类的
                  // 你读取到了多少个字节，此时buffer的position就会变成多少
                  System.out.println("[" + Thread.currentThread().getName() + "]接收

                  if(count > 0){
                    buffer.flip();   // position = 0, limit = 21, 仅仅读取buffer中, 0~
                      System.out.println("服务端接收请求: " + new String(buffer.array(
                      channel.register(selector, SelectionKey.OP_WRITE);
                  }
              } else if(key.isWritable()) {
                ByteBuffer buffer = ByteBuffer.allocate(1024);
                buffer.put("收到".getBytes());
                buffer.flip();

                channel = (SocketChannel) key.channel();
                channel.write(buffer);
                channel.register(selector, SelectionKey.OP_READ);
```

```
                }
            }
            catch(Throwable t){
                t.printStackTrace();
                if(channel != null){
                    channel.close();
                }
            }
        }

    }

}
```

总结:

通过本篇文章，主要是分析了常见的NIO的一些问题：

- BIO， NIO， AIO各自的特点
- 什么同步阻塞，同步非阻塞，异步非阻塞
- 为什么NIO能够应对支持海量的请求
- NIO相关组件的原理
- NIO通讯的简单案例

本文仅仅是介绍了一下网络通讯的一些原理，应对面试来讲解

NIO通讯其实有很多的的东西，在中间件的研发过程中使用的频率还是非常高的，后续有机会再和大家分享交流。

热文推荐

译文：初级，中级和高级开发人员之间的差异?

同事问我，为什么阿里P3C不建议返回值用枚举，而参数可以呢?

老王：Netty到底是个什么鬼？有没有简单的理解方式?

同时，分享一份Java面试资料给大家，覆盖了算法题目、常见面试题、JVM、锁、高并发、反射、Spring原理、微服务、Zookeeper、数据库、数据结构等等。

获取方式：点“**在看**”，关注公众号并回复 面试 领取。