> Java提供了哪些IO方式？ NIO如何实现多路复用？

Java IO和NIO

版本号：2018/9/19-1(14:00)

基本概念(15)

1、Java IO方式有哪些?

> 1. 传统java.io包：对文件进行了抽象、通过输入流输出流进行IO
> 2. java.net包: 网络通信同样是IO行为
> 3. java.nio包：Java1.4中引入了NIO框架
> 4. java7的NIO2：引入了异步非阻塞IO方式

2、按照阻塞方式分类

1. BIO: 同步、阻塞
2. NIO：同步、非阻塞
3. NIO2/AIO：异步、非阻塞

3、传统java.io包中的IO有什么特点?

1. 基于stream模型实现
2. 提供了常见的IO功能: File抽象、输入输出流
3. 交互方式：同步、阻塞的方式
4. 在读写动作完成前，线程会一直阻塞在那里，它们之间的调用是可靠的线性顺序。

4、Stream(流)到底是什么？有什么用?

1. Out：代表能产出数据的数据源对象
2. In：代表能接受数据的数据源对象
3. 作用：为数据源和目的地搭建一个传输通道

5、java.io包的好处和缺点

1. 优点：代码简单、直观
2. 缺点：IO效率、扩展性存在局限性，会成为应用性能的瓶颈

6、java.net下的网络通信的IO行为

1. java.net下的网络API：Socket、ServerSocket、HttpURLConnection
2. 这些也都属于同步阻塞IO类库

7、NIO框架是什么?

1. Java 1.4中引入
2. 位于java.nio 包
3. 提供了 Channel、Selector、Buffer 等新的抽象

8、NIO的特点?

1. 可以构建多路复用的、同步非阻塞 IO 程序
2. 同时提供了更接近操作系统底层的高性能数据操作方式。

9、NIO2或者AIO是什么?

1. NIO 2，又称为AIO（Asynchronous IO）。
2. 在 Java 7 中，对NIO进一步改进。
3. 引入了异步非阻塞 IO 方式，也
4. 异步 IO 操作基于事件和回调机制---应用操作直接返回，而不会阻塞，当后台处理完成后，操作系统会通知相应线程进行后续工作。

10、nio和io相比性能优势在于哪里?(@Deprecated的说法)

1. IO面向流，从Stream中逐步读取数据，并且没有缓冲区。
2. NIO面向面向缓冲区，数据整体操作更加高效。
3. IO是阻塞的，当前线程在没有数据可读时会出现阻塞。
4. NIO是非阻塞的，通过Selector选择器选择合适的Channel进行数据操作。当一个Channel没有数据时，会切换到有效的Channel处理其他io，更搞笑。

11、NIO的性能就一定比IO高?如果是带缓冲的IO和NIO相比呢?

1. 传统的IO理论上是没有NIO快的: 用IO进行一个字节一个字节的读取。
2. 但是如果合理使用，如带缓冲区的IO(BufferedInputStream、BufferedReader)时会很快。
3. 此外根据测试在进行文件拷贝等IO操作时，会发现 `NIO` 并没有比 `IO` 更快，甚至在个别场景还会出现 `NIO` 更慢的情况
4. IBM官方指明：JDK1.4时已经将 `java.io` 以 `nio` 为基础重新进行了实现，可以利用一些NIO的特性。因此处理方面的性能并不比NIO差。

12、NIO的真正优势并不是体现在速度上?

1. 随着JDK1.4对IO进行了重构。NIO在速度上的优势并不存在了。
2. NIO真正优势体现在:
    1. 分散和聚集: 利用 `Scatter/Gather` 委托操作系统完成数据分散和聚集的工作
    2. 文件锁定功能:
    3. 网络异步IO: 非阻塞IO、IO多路复用(解决服务端多线程时的线程占用问题)

同步和异步

13、同步和异步的区别?

1. 同步-synchronous
2. 异步-asynchronous
3. 同步是一种可靠的有序运行机制，同步操作时，后续的任务会等待当前调用的返回。
4. 异步中，其他任务不会等待当前调用返回，通常依靠事件、回调等级制来实现任务间次序关系

阻塞和非阻塞

14、阻塞和非阻塞的区别?

1. 阻塞-blocking
2. 非阻塞-non-blocking
3. 阻塞操作时，当前线程会处于阻塞状态，无法进行其他任务，只有当满足一定条件时，才继续执行
4. 非阻塞状态，不会去等待IO操作结束，会立即返回。相应操作会在后台处理

15、阻塞和同步就是低效的操作?

> 错误!
> 需要根据应用的实际场景。有些时候必须要进行阻塞和同步。

IO/BIO(32)

1、传统IO操作就是指对文件进行操作?

> 错误!
>
> 1. 文件操作是IO操作
> 2. 网络编程中，如Socket通信，都是典型的IO操作

2、IO流是什么吗？有什么用

> 1. Input流和Output流
> 2. 主要用于处理设备间的数据传输

3、IO流的两种分类方式

> 1. 字节流和字符流
> 2. 输入流和输出流

4、字节流的抽象基类?

> 1. InputStream
> 2. OutputStream

5、字符流的抽象基类

> 1. Reader
> 2. Writer

6、字符流中融合了编码表

> 系统默认的一般采用GBK

7、字符流与字节流的区别

> 1. 处理对象不同：
>   1. 字节流能处理所有类型的数据（如图片、多媒体等）
>   2. 字符流只能处理字符类型的文本数据。
> 2. 读写单位不同：
>   1. 字节流以字节byte为单位，1byte=8bit。
>   2. 字符流以字符为单位，1个字符=2个字节(java中采用unicode编码)。根据码表映射字符，一次可能读多个字节。
> 3. 有无缓冲区：

1. 字节流没有缓冲区，是直接输出的。字节流不调用colse()方法时，信息就已经输出了。
2. 字符流是输出到缓冲区的。只有在调用close()方法关闭缓冲区时，信息才输出。要想字符流在未关闭时输出信息，则需要手动调用flush()方法。

8、字符流让信息输出的两种办法

1. close()关闭缓冲区时，信息会输出。
2. 手动调用flush()来输出信息。

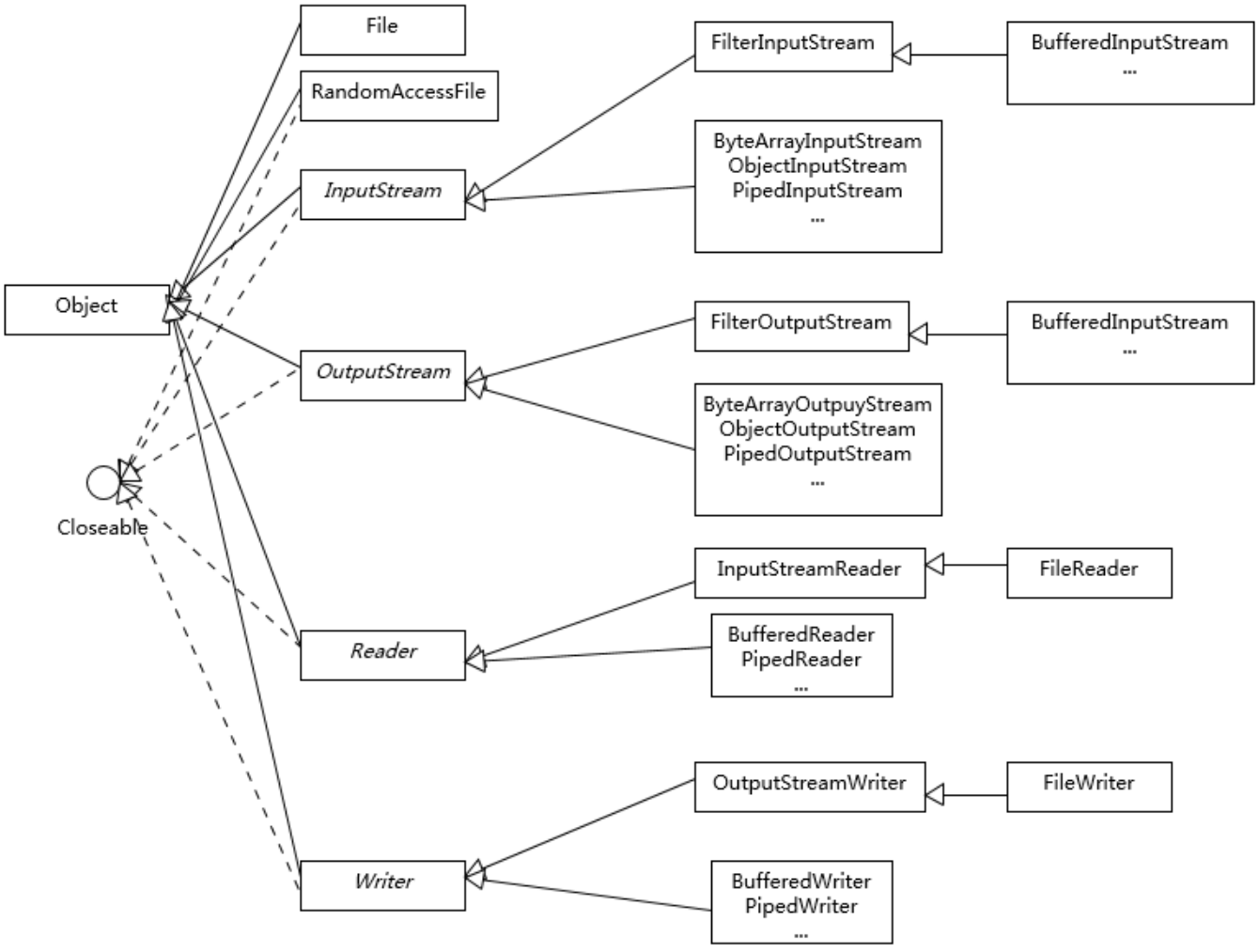
9、字符流和字节流如何选择?

1. 只要是处理纯文本数据，就优先考虑使用字符流
2. 除此之外都使用字节流

10、Closeable 接口

1. 很多 IO 工具类都实现了 Closeable 接口，因为需要进行资源的释放。
2. 需要利用try-with-resources、 try-finally 等机制保证 资源被释放
3. Cleaner 或 finalize 机制作为资源释放的最后把关，也是必要的。

11、Java传统IO相关的类图



12、java.io包中六大类和接口

> 1. File、RandomAccessFile、InputStream、OutputStream、Reader、Writer
> 2. Serializable

13、InputStream/OutputStream 和 Reader/Writer 的关系和区别。

> 1. 都实现了Closeable接口，用于资源的释放。
> 2. 字节流: InputStream/OutputStream
> 3. 字符流: Reader/Writer

14、Java I/O 主要的三个部分

> 1. 流式部分-IO主体部分
> 2. 非流式部分-一些辅助流式部分的类：File、RandomAccessFile、FileDescriptor
> 3. 其他类-文件读取部分、安全相关的类

File

15、File类

> 1. 采用File文件作为类名并不准确.
> 2. 本质上是文件路径，使用FilePath会更准确。

16、创建的新文件，为什么只有很少的内容，也会占据几KB?

> 操作系统有最小分配空间

17、不同文件的开头会包含该文件类型相关信息

18、文件的创建

```
//创建文件
File file = new File("d:\\a.txt");
if(file.exists() == false)
{
    file.createNewFile();
}
```

19、文件夹的创建

```
//创建文件夹
File fileFolder = new File("d:\\New Folder");
if(fileFolder.isDirectory() == false)
{
    fileFolder.mkdir(); //创建folders
}
```

20、列出文件夹内所有文件

```
//列出所有文件
File fileFolder = new File("d:\\New Folder");
if(fileFolder.isDirectory() == false)
{
    File []files = fileFolder.listFiles();
    for (File file : files) {
        file.getName();
    }
}
```

RandomAccessFile

21、RandomAccessFile是什么?

> 1. 随机文件操作
> 2. 一个独立的类，直接继承至Object.
> 3. 功能丰富，可以从文件的任意位置进行存取（输入输出）操作。

InputStream

22、输入流/输出流的作用?

> 1. InputStream/OutputStream
> 2. 是用于读取或写入字节的，例如操作图片文件。

23、FileInputStream的注意点

> 1. 打开 FileInputStream，会获取相应的文件描述符（FileDescriptor）
> 2. 需要利用try-with-resources、 try-finally 等机制保证 FileInputStream 被明确关闭，进而相应文件
>
>   描述符也会失效，否则将导致资源无法被释放。

FileInputStream

24、FileInputStream读取文件中数据

```
// 1、创建文件
File file = new File("d:\\a.txt");
// 2、创建输入流(字节流)
FileInputStream fileInputStream = new FileInputStream(file);
byte[] bytes = new byte[1024];
int n;
// 3、从输入流中读取数据，存放到byte数组中
while((n = fileInputStream.read(bytes)) != -1)
{
    // 4、创建String并且显示
    String s = new String(bytes, 0, n);
    System.out.println(s + "\r\n"); //换行
}
// 5、关闭输入流
fileInputStream.close();
```

OutputStream

FilterOutputStream

BufferedOutputStream

25、BufferedOutputStream的作用?

> 1. BufferedOutputStream 等带缓冲区的实现，
> 2. 可以避免频繁的磁盘读写，进而提高 IO 处理效率。
> 3. 这种设计利用了缓冲区，将批量数据进行一次操作，
> 4. 使用中一定要 `flush` 。

Reader

26、Reader/Writer的作用?

> 1. Reader/Writer 则是用于操作字符
> 2. 增加了字符编解码等功能
> 3. 适用于从文件中读取或者写入文本信息等操作。
> 4. 本质上计算机操作的都是字节(不管是网络通信还是文件读取)，Reader/Writer 相当于构建了应用逻辑和原始数据之间的桥梁。

InputStreamReader

FileReader

27、FileReader读取文件中数据

```
// 1、创建文件
File file = new File("d:\\a.txt");
// 2、创建输入流(字符流)
FileReader fileReader = new FileReader(file);
char[] chars = new char[1024];
int n;
// 3、从输入流中读取数据，存放到byte数组中
while((n = fileReader.read(chars)) != -1)
{
    // 4、创建String并且显示
    String s = new String(chars, 0, n);
    System.out.println(s + "\r\n"); //换行
}
// 5、关闭输入字符流
fileReader.close();
```

BufferedReader

28、BufferedReader的作用

> 1. 包装Reader的子类
> 2. 增加缓存区的功能

29、BufferedReader的使用

```
// 1、创建FileReader
FileReader fileReader = new FileReader(new File("d:\\a.txt"));
// 2、创建BufferedReader，利用缓存区增强性能，并且提供readline()功能
BufferedReader bufferedReader = new BufferedReader(fileReader);
// 3、从输入流中读取数据，存放到byte数组中
String str;
while((str = bufferedReader.readLine()) != null)
{
    System.out.println(str); //换行
}
// 4、关闭BufferedReader
bufferedReader.close();
```

Writer

PrintWriter

30、PrintWriter的作用

> 1. 向文本输出流, 以格式化的形式, 打印数据。
> 2.

31、PrintWriter的使用

```
// 1、创建PrintWriter
PrintWriter printWriter = new PrintWriter(new File("d:\\a.txt"));
// 2、向文件中写入数据。原来的所有数据会先删除。然后依次写入数据
printWriter.append("Hello");
printWriter.write("World!");
printWriter.print("Godebye");
// 3、刷新缓存区
printWriter.flush();
printWriter.close();
```

32、write、print、append之间的区别?

> 1. 效果上没有区别，都是写入数据。
> 2. 返回值上会有区别，append()会返回 `printWriter` ，可以进行链式调用。
> 3. write和print都没有返回值。
> 4. print()参数为(String)null，会打印出null
> 5. write()参数为null，会有空指针异常。

NIO(62)

1、NIO的主要组成部分

> 1. Buffer
> 2. Channel
> 3. Selector
> 4. ChartSet

Channel

2、Channel的作用?

> 1. 类似在Linux操作系统上的文件描述符
> 2. 一种操作系统底层的抽象
> 3. 用来支持批量式IO操作

3、Channel 是操作系统底层的一种抽象。

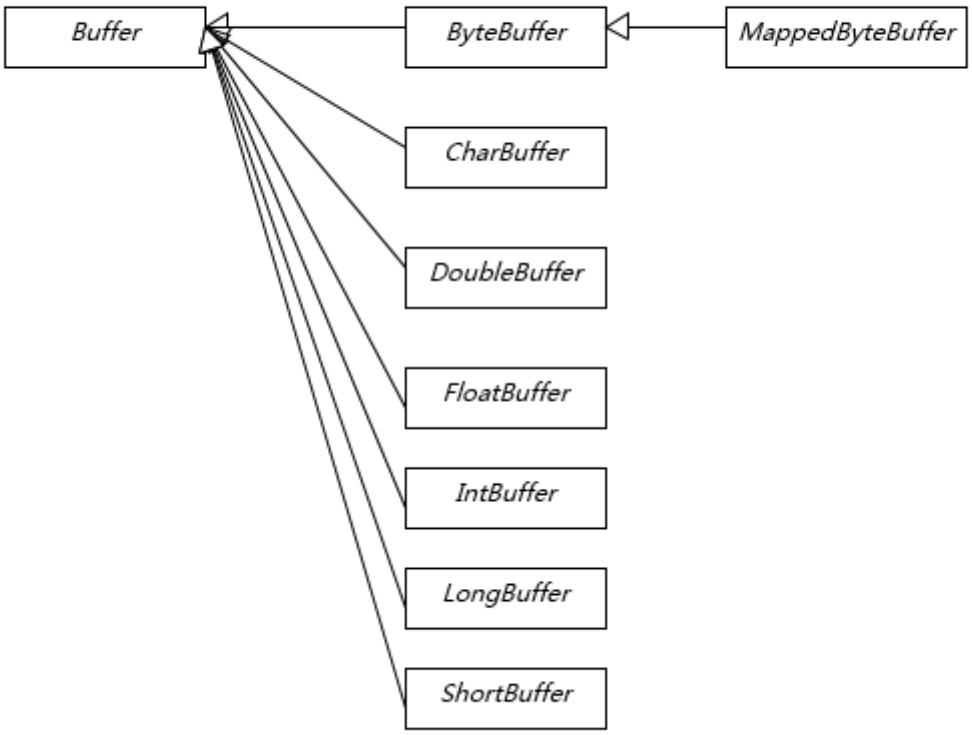
> 1. File 或者 Socket，通常被认为是比较高层次的抽象
> 2. Channel 是更加操作系统底层的一种抽象
> 3. 使得 NIO 得以充分利用现代操作系统底层机制，获得特定场景的性能优化，
> 4. 例如，DMA（Direct Memory Access）等。
> 5. 不同层次的抽象是相互关联的，Socket和Channel之间能相互获取。

Buffer

4、Buffer的作用?

> 1. Buffer是NIO操作数据的基本工具
> 2. Java为每种原始数据类型都提供了Buffer(布尔类型除外)
> 3. 高效的数据容器

5、Buffer的分类(7种)



6、Buffer的基本属性

基本属性	作用
capcity	Buffer的大小，也就是数组的长度
position	对数据进行操作的起始位置
limit	操作的限额。读取操作，limit就是所能容纳数据的上线；写入操作，limit就是设置为容量或者容量以下的可写额度
mark	上一次position的位置，默认-1

7、Buffer的创建

```
ByteBuffer byteBuffer = ByteBuffer.allocate(10);
```

> 1. capcity = 10(容量)

2. position = 0，会从第一个数据才是操作。
3. limit = 10， 操作的数据不能超过容量。
4. mark = -1(默认值)

8、读取数据到Buffer中

```
socketChannel.read(byteBuffer);
```

1. position会随着read操作而不断增大，但不会超过limit

9、buffer.flip: 反转操作，用于读取之前写入的数据

```
byteBuffer.flip();
```

1. 如翻转前：position = 10， limit = 20
2. flip后：limit = 10(position的旧值), position = 0(复位到0)

10、从Buffer中读取数据

```
socketChannel.write(buffer);
```

1. 和read类似
2. 随着操作，buffer中的position会逐渐增加，接近limit(但不会超过)

11、buffer.rewind：重读数据

```
buffer.rewind();
```

1. limit保持不变
2. position = 0.

ByteBuffer

12、ByteBuffer是什么?

1. NIO中使用的Byte Buffer
2. 包含两个实现方法：
    1. HeapByteBuffer: 基于Java堆的实现
    2. DirectByteBuffer: 堆外的实现方法，采用了 `unsafe API`

13、从Channel中读取数据到ByteBuffer中

```
byteBuffer = ByteBuffer.allocate(N);
//读取数据，写入byteBuffer
socketChannel.read(byteBuffer);

// 翻转，才能打印出来
byteBuffer.flip();
System.out.println("receive msg from client: "+Charset.defaultCharset().decode(byteBuffer.asRea
```

14、向Channel中写入数据

```
socketChannel.write(Charset.defaultCharset().encode("Hello World!"));
```

flip

15、ByteBuffer.flip()

> 1. 进行翻转。将limit设置到position，然后将position复位到0。
> 2. 从Channel中read后，立即用于写数据。

```
ByteBuffer byteBuffer = ByteBuffer.allocate(1024);
// 1、读取到数据
socketChannel.read(byteBuffer);
// 2、翻转
byteBuffer.flip();
// 3、发送给Client
socketChannel.write(byteBuffer);
```

Direct、Heap

16、Buffer.isDirect()有什么用?

```
public abstract boolean isDirect();
```

> 1. 抽象方法
> 2. 用于判断具体Buffer是属于 堆内`Buffer(Heap)` 还是 堆外`Buffer(Direct)`

17、如何创建堆外的Buffer?

> 1. 使用allocateDirect()方法创建
> 2. 只有 `ByteBuffer` 可以调用 `allocateDirect()` 创建 堆外 Buffer

```
public static ByteBuffer allocateDirect(int capacity) {
    return new DirectByteBuffer(capacity);
}
```

18、如何创建堆内的Buffer?

1. allocate()方法创建
2. 所有Buffer都可以创建在堆内

```
// ByteBuffer.java
public static ByteBuffer allocate(int capacity) {
    if (capacity < 0)
        throw new IllegalArgumentException();
    return new HeapByteBuffer(capacity, capacity);
}


// IntBuffer.java
public static IntBuffer allocate(int capacity) {
    if (capacity < 0)
        throw new IllegalArgumentException();
    return new HeapIntBuffer(capacity, capacity);
}


// LongBuffer.java
public static LongBuffer allocate(int capacity) {
    if (capacity < 0)
        throw new IllegalArgumentException();
    return new HeapLongBuffer(capacity, capacity);
}
```

19、所有的Buffer都可以选择创建在堆内或者堆外?

不准确!

1. 只有ByteBuffer可以调用 `allocateDirect()` 创建 `Direct Buffer(堆外)`
2. 其他类型Buffer没有该方法，但是也有相关类 `DirectCharBufferS` 等等。

20、堆内Buffer底层是如何实现的?

1. 共七种Buffer: ByteBuffer、IntBuffer、LongBuffer等等
2. 内部就是对应的数组；byte数组、int数组、long数组等等，
3. position、limit、mark、capacity的赋值，都是在基类Buffer的构造方法中执行。

21、使用堆内Buffer操作相关的api，jdk会额外进行Direct Buffer缓存

1. 使用堆内Buffer操作相关的api, jdk会将其复制为Direct buffer，并且在线程内部进行缓存。
2. 早期jdk对Direct Buffer的缓存大小没有限制，但是容易出现OOM，后续jdk进行了限制。
3. 因此建议：尽量不要使用堆内的ByteBuffer操作Channel类api。

MappedByteBuffer

22、DirectByteBuffer是什么?

1. 继承自抽象类 `MappedByteBuffer`

2. 实现了 `DirectBuffer` 接口
3. Direct-堆外相关的类都无法在JDK之外去引用
4. 底层利用了unsafe_allocatememory

23、MappedByteBuffer的内部原理

1. 将文件按照指定大小，直接映射为 内存区域
2. 程序访问该内存区域时，可以直接操作文件的数据。
3. 直接将数据拷贝到了用户空间, 从而省去了将数据从 内核空间 向 用户空间 传输的损耗。
4. 本质上就是一种 `Direct Buffer`

24、MappedByteBuffer会影响NIO性能?

1. 错误观点: NIO如果使用不当，速度会比传统IO慢几十倍。比如用 `MappedByteBuffer` 将文件映射到内存时。
2. 因为本质是节省了上下文切换的性能开销，性能应该更好。

25、内存映射的效率比系统调用 `read、write` 还要高?

1. read、write作为系统调用，将数据拷贝到内存中，需要经过两次拷贝：磁盘文件到内核缓存区，内核缓存区到用户空间缓存区。
2. 内存映射，直接将文件数据从硬盘拷贝到了 用户空间 ，只有一次数据拷贝。
3. NIO中直接内存映射到Buffer相当于直接在内存中存取数据，不需要经过内核态的缓冲区，性能更高。

26、MappedByteBuffer的创建

> 1-实例

```
// 只读、文件的起始位置10、map的size为30-最大位置是40
fileChannel.map(READ_ONLY, 10, 30);
```

> 2-map()方法

```
public abstract MappedByteBuffer map(MapMode mode, // 模式
                                     long position, //从文件的position位置开始进行map映射
                                     long size) //map映射的最大尺寸
```

> 3-Mode有三种: 只读、可读可写、写时复制

```
 // read-only：只读
public static final MapMode READ_ONLY;

// read/write：可读可写
public static final MapMode READ_WRITE;

// private (copy-on-write)：私有模式，写时进行拷贝。
public static final MapMode PRIVATE;
```

27、copy-on-write是什么意思?(COW)

> 1. 写时拷贝技术-COW

28、DirectBuffer接口有什么用?

> 1-内部定义了三个方法

```
public interface DirectBuffer {
    long address();
    Object attachment();
    Cleaner cleaner();
}
```

性能开销

29、DirectBuffer的性能优势？为什么会有性能优势?

> 1. 实际使用中，Java会尽量对Direct Buffer只做 本地IO操作
> 2. 对于很多大数据量的 IO密集操作 ，性能会比较高
> 3. Direct Buffer 生命周期内内存地址都不会再发生改变，因此内核可以安全的进行访问， IO操作会很高效 。(本质就是寻址简单，跟锁没关系)
> 4. 减少了Heap堆内对象存储时的维护工作， 访问效率会提高 。

30、为什么Direct Buffer 生命周期内内存地址都不会再发生改变，因此IO操作会很高效?是否是因为没有锁竞争?

> 1. 本质就是寻址简单
> 2. 跟锁完全没有关系

31、DirectBuffer的性能缺点？什么场景下才适合使用DirectBuffer?

> 1. DirectBuffer 在创建和销毁中，相比 Heap Buffer 会有额外的开销
> 2. 适合长期使用、数据较大的场景。

32、Direct Buffer(堆外)比Heap Buffer更高效，所以应该尽可能都用Direct Buffer?

> 错误!
> 短期使用、数据量较少时还是 堆内Heap Buffer 更好一些。

垃圾回收

33、Direct Buffer对内存和JVM参数的影响

> 1. 不在堆上，Xmx之类的参数，不能影响Direct Buffer等堆外成员所使用的内存额度
> 2. 堆外Buffer设置内存额度的JVM参数: -xx:MaxDirectMemorySize=512M
> 3. 计算Java可以使用的内存大小，除了需要考虑堆内，还需要考虑DirectBuffer等堆外元素

34、Direct Buffer什么时候会被垃圾回收?

> 1. 实际无法预测
> 2. 依赖于cleanner
> 3. 一般是延迟到 `full GC` 时期，快满时会被 `System.gc()` 触发ref处理。

35、Java可使用的内存大小只和堆有关?

> 错误!
>
> 1. 不仅需要考虑Heap
> 2. 还需要考虑堆外元素

36、如果出现内存不足，可能有哪些原因?

> 1. 需要考虑堆外内存占用

37、垃圾回收是否会回收Direct Buffer?Direct Buffer是如何回收的?

> 1. 绝大部分GC都不会主动收集Direct Buffer
> 2. Direct Buffer的垃圾回收是基于 `Cleaner` 机制和 `PhantomReference`-虚引用、幻想引用
> 3. Direct Buffer本身不是public类型，内部具有一个 `Deallocator` 负责销毁逻辑
> 4. 其销毁主要是在 `full GC` 的时候，使用不当会 `OOM`

38、Direct Buffer垃圾回收上的注意点

> 1. 应用程序中，要显式地调用 `System.gc()` 来强制触发GC
> 2. 在大量使用Direct Buffer的时候，主动去调用释放方法。(可以参考Netty框架，实现在PlatformDependent中)
> 3. 重复使用Direct Buffer

39、Direct Memory一定不会引起Full GC, 只有在Full GC和调用System.gc()时才会去回收?

> 1. 不是，还是利用 `sun.misc.Cleaner`
> 2. 但是具体实现有瑕疵，进场需要依赖 `System.gc()`
> 3. 后续的jdk版本有改进

40、如何跟踪和诊断Direct Buffer的内存占用?

> 1. 通常的垃圾回收日志不会包含Direct Buffer的信息

> 2. JDK8之后，可以使用 `Native Memory Tracking(NMT)` 特性进行诊断
> 3. NMT的参数: `-XX:NativeMemoryTracking=(summary | detail)`
> 4. 激活NMT通常会导致 `JVM` 出现 `5%~10%` 的性能下降

41、NMT可以在运行时采用命令进行交互式对比

> 1-打印NMT信息

```
// 打印NMT信息
jcmd <pid> VM.native_memory detail
```

> 2-进行baseline，并且对比分配内存变化

```
// 进行baseline
jcmd <pid> VM.native_memory baseline

// 对比分配内存变化
jcmd <pid> VM.native_memory detail.diff
```

> 3-输出结果的Internal部分会包含Direct Buffer内存使用情况

```
-Internal (reserved=679KB +4KB, committed=679KB +4KB)
          (malloc=615KB +4KB #1571 +4)
          (mmap: reserved=64KB, committed=64KB)
```

> 4-底层利用了unsafe_allocatememory, 但是本质并不是JVM内部使用的内存，在JDK11之后，将其分类在other部分

Selector

42、Selector的作用

> 1. 是 NIO 实现多路复用的基础，
> 2. 它提供了一种高效的机制，可以检测到注册在Selector 上的多个 Channel 中，是否有 Channel 处于就绪状态，进而实现了单线程对多Channel 的高效管理。
> 3. Selector也是基于底层操作系统机制的，不同模式、不同版本都存在区别。
> 4. Linux 上依赖于epoll
> 5. Windows 上 NIO2（AIO）模式则是依赖于iocp

43、Linux中的epoll是什么?

44、Windows中的iocp是什么?

SelectionKey

45、SelectionKey是什么?

> 1. 表示 `SelectableChannel` 在Selector中注册的句柄/标记
> 2. `serverSocketChannel.register(selector, SelectionKey.OP_ACCEPT);` 会返回注册事件的句柄。

46、一个Selector对象包含三种类型的SelectionKey集合

all-keys	当前所有向Selector注册的Channel的句柄(SelectionKey)的集合	selector.keys()
selected-keys	相关事件已经被Selector捕获的SelectionKey的集合	selector.selectedKeys()
cancelled-keys	已经被取消的SelectionKey的集合	无API

47、SelectionKey何时被新建？何时会被加入到Selector的all-keys集合中?

> 1. Channel注册到Selector中时, 会新建一个SelectionKey，然后加入到all-keys集合中。
> 2. serverSocketChannel.register(selector, xxx)

48、SelectionKey对象何时会被遗弃(加入到cancelled-keys集合中)?

> 1. SelectionKey相关的Channel被关闭
> 2. 调用了 `SelectionKey.cancel()` 方法

CharSet

49、Chartset的作用

> 1. 提供 Unicode 字符串定义，
> 2. NIO 也提供了相应的编解码器等，
> 3. 例如，通过下面的方式将字符串转换到 ByteBuffer：

```
Charset.defaultCharset().encode("Hello world!"));
```

50、ByteBuffer转换为String

```
Charset charset = Charset.defaultCharset();
// asReadOnlyBuffer将Buffer复制一份出来。
CharBuffer charBuffer = charset.decode(byteBuffer.asReadOnlyBuffer());
String string = charBuffer.toString();
```

多路复用

51、为什么需要多路复用?

1. 传统IO是一个线程处理一个链接
2. 采用多路复用可以一个线程处理多个链接

52、NIO多路复用的局限性

1. 当有IO请求在数据拷贝阶段。
2. 由于资源类型过于庞大，会导致线程长期阻塞
3. 造成性能瓶颈

Scatter/Gather

53、NIO的Scatter/Gather机制是什么?

1. 作为一个强大的工具，将数据的分散和聚集的任务委托给操作系统来完成。
2. Scatter: 将读取到的数据分开放置到多个存储桶中(Bucket)
3. Gather: 将不同的数据区块合并成一个整体

54、如何在channel读取时，将不同片段写入到对应的Buffer中(类似二进制消息拆分为消息头、消息体)？可以采用NIO的什么机制?

1. 可以采用NIO分散-scatter机制来写入不同Buffer。
2. 但是需要请求头的长度固定：

```
// 消息头
ByteBuffer header = ByteBuffer.allocate(128);
// 消息体
ByteBuffer body = ByteBuffer.allocate(1024);

// 从channel中读取不同片段
ByteBuffer[] bufferArray = {header, body}; channel.read(bufferArray);
```

NIO2

55、NIO2

1. Java 7引入了NIO 2
2. 提供了一种额外的异步IO模式
3. 利用事件和回调，处理 `Accept`、`Read` 等操作。

56、NIO 2 也不仅仅是异步

57、Future

58、CompletionHandler

59、Reactor和Proactor模式需要和Netty主题一起

60、NIO和NIO2的类似处

> 1. AsynchronousServerSocketChannel对应ServerSocketChannel
> 2. AsynchronousSocketChannel对应SocketChannel

61、NIO2的局限性

AsynchronousServerSocketChannel

62、AsynchronousServerSocketChannel进行网络请求

```
// 1、创建AsynchronousServerSocketChannel
AsynchronousServerSocketChannel serverSocketChannel = AsynchronousServerSocketChannel.open()){
// 2、绑定IP和端口
serverSocketChannel.bind(new InetSocketAddress(InetAddress.getLocalHost(), 8888));

// 3、为异步操作，指定CompletionHandler回调。
serverSocketChannel.accept(serverSocketChannel, new CompletionHandler<AsynchronousSocketChannel, Asynchrono
        @Override
        public void completed(AsynchronousSocketChannel result, AsynchronousServerSocketChannel attachment)
               serverSocketChannel.accept(serverSocketChannel, this);
            handleRequest(result);
        }

        @Override
        public void failed(Throwable exc, AsynchronousServerSocketChannel attachment) {

        }});
```

网络IO(13)

IO/BIO

1、Socket简单实现客户端和服务端通信

> 1-服务端：建立ServerSocket，等待客户端连接，然后处理数据。

```
public class DemoSocketServer extends Thread{
    private ServerSocket serverSocket;
    public int getPort(){
        return serverSocket.getLocalPort();
    }
    @Override
    public void run() {
        try {
            // 1、服务端启动ServerSocket，端口=0，表示自动绑定一个空闲端口
            serverSocket = new ServerSocket(0);
            while (true){
                // 2、阻塞等待一客户端的连接
                Socket socket = serverSocket.accept();
                // 3、处理客户端(新建一个线程)
                RequestHandler requestHandler = new RequestHandler(socket);
                requestHandler.start();
            }
        } catch (IOException e) {
            e.printStackTrace();
        }finally {
            // 任何情况下都要保障Socket资源关闭。
            if(serverSocket != null){
                try {
                    serverSocket.close();
                } catch (IOException e) {
                    e.printStackTrace();
                }
            }
        }
    }
    // 客户端请求的Handler
    public static class RequestHandler extends Thread{
        private Socket mSocket;
        RequestHandler(Socket socket){
            mSocket = socket;
        }
        @Override
        public void run() {
            try {
                // 1、Socket的输出流来创建printWriter
                PrintWriter printWriter = new PrintWriter(mSocket.getOutputStream());
                // 2、写入数据
                printWriter.println("Hello World!");
                printWriter.flush();
            } catch (IOException e) {
                e.printStackTrace();
            }
        }
    }
}
```

> 2-客户端(简单的打印数据)：借助try-with-resources，用Reader去读取数据。

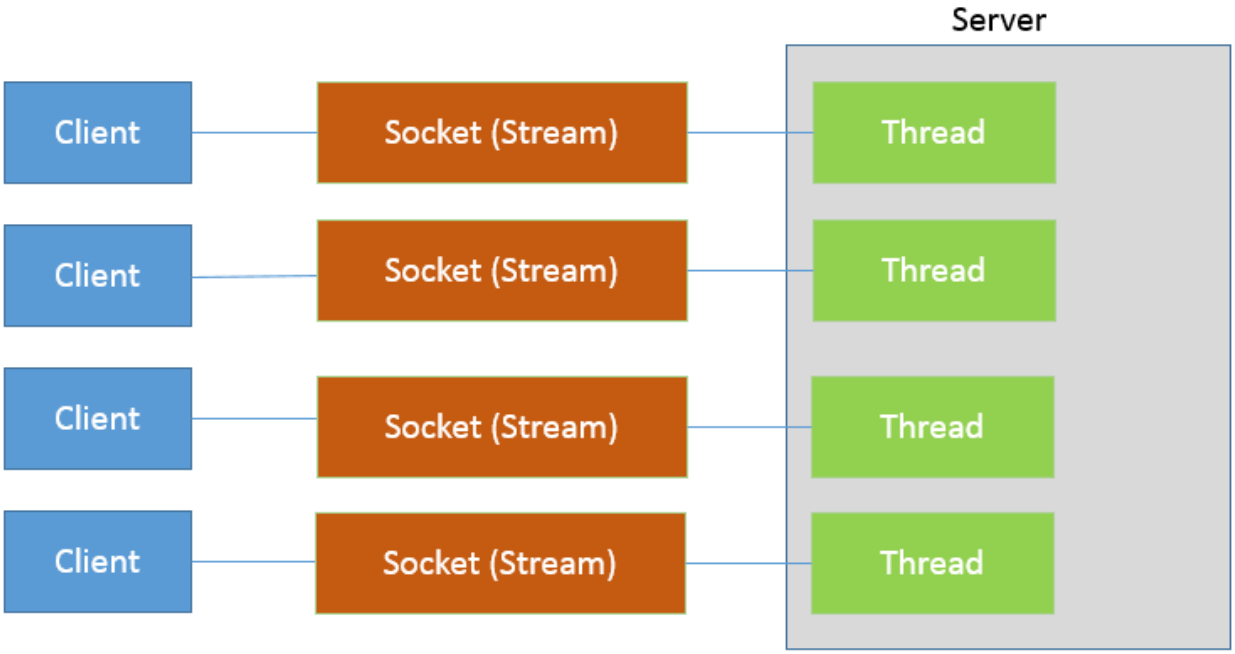
```
// 客户端
public class Main {
    public static void main(String[] args) throws IOException {
        DemoSocketServer server = new DemoSocketServer();
        server.start();
        // 1、Socket客户端，绑定Server端Host地址，和Server端的端口。(这边是本机)
        try (Socket client = new Socket(InetAddress.getLocalHost(), server.getPort())) {
            // 2、通过客户端的inpustream，创建Reader
            BufferedReader bufferedReader = new BufferedReader(new InputStreamReader(client.getInputStream(
            // 3、从Reader中读取到数据，并且打印。
        }
    }
}
```

2、线程池改进服务端

> 1. 需要减少线程频繁创建和销毁的开销

```
// 线程池
private Executor mExecutor;
@Override
public void run() {
    try {
        serverSocket = new ServerSocket(0);
        // 1、创建线程池：只有核心线程数，没有非核心线程数。任务队列无限。空闲线程会立即停止
        mExecutor = Executors.newFixedThreadPool(8);
        while (true){
            Socket socket = serverSocket.accept();
            RequestHandler requestHandler = new RequestHandler(socket);
            // 2、线程池进行处理
            mExecutor.execute(requestHandler);
        }
    } catch (IOException e) {
        e.printStackTrace();
    }finally {
        // 任何情况下都要保障Socket资源关闭。
        if(serverSocket != null){
            try {
                serverSocket.close();
            } catch (IOException e) {
                e.printStackTrace();
            }
        }
    }
}
```

3、服务端采用线程池来提供服务的典型工作模式图



4、服务端采用线程池处理客户端连接的缺点?

> 1. 连接数几百时，这种模式没有问题。
> 2. 但是在高并发，客户端及其多的情况下，就会出现问题。
> 3. 线程上下文切换的开销会在高并发时非常明显。
> 4. 这就是同步阻塞方式的低扩展性的体现

NIO

5、NIO优化服务端连接问题的实例

```
public class NIOServer extends Thread{
    @Override
    public void run() {
        try (// 1、创建Selector。调度员的角色。
             Selector selector = Selector.open();
             /**-------------------------
              * 2、创建Channel。并进行配置。
              *-------------------------*/
             ServerSocketChannel serverSocketChannel = ServerSocketChannel.open()){
            // 1. 绑定IP和端口
            serverSocketChannel.bind(new InetSocketAddress(InetAddress.getLocalHost(), 8888));
            // 2. 非阻塞模式。因为阻塞模式下是不允许注册的。
            serverSocketChannel.configureBlocking(false);
            /**-------------------------
             * 3、向Selector进行注册。通过OP_ACCEPT，表明关注的是新的连接请求
             *-------------------------*/
            serverSocketChannel.register(selector, SelectionKey.OP_ACCEPT);
            while(true){
                // 4、Selector调度员，阻塞在select操作。当有Channel有接入请求时，会被唤醒
                selector.select();
                // 5、被唤醒，获取到事件已经被捕获的SelectionKey的集合
                Set<SelectionKey> selectionKeys = selector.selectedKeys();
                Iterator<SelectionKey> iterator = selectionKeys.iterator();
                while (iterator.hasNext()){
                    SelectionKey selectionKey = iterator.next();
                    // 6、从SelectionKey中获取到对应的Channel
                    handleRequest((ServerSocketChannel) selectionKey.channel());
                    iterator.remove();
                }
            }

        } catch (IOException e) {
            e.printStackTrace();
        }
    }

    // 处理客户端的请求
    private void handleRequest(ServerSocketChannel socketChannel){

        // 1、获取到连接到该Channel Socket的连接
        try(SocketChannel client = socketChannel.accept()) {
            ByteBuffer byteBuffer = ByteBuffer.allocate(1024);

            // 2、从客户端读取数据
            client.read(byteBuffer);

            // 3、翻转，用于读取显示和发送给服务器
            byteBuffer.flip();
            System.out.println("receive msg from client: "
                    +Charset.defaultCharset().decode(byteBuffer.asReadOnlyBuffer()).toString());

            // 4、向客户端中写数据
            client.write(byteBuffer);
        } catch (IOException e) {
```

```
            e.printStackTrace();
        }
    }
}
```

> 测试

```
public class Main {
    public static void main(String[] args) throws IOException {
        // 开启服务器
        NIOServer server = new NIOServer();
        server.start();

        // 1、Socket客户端，绑定Server端Host地址，和Server端的端口。(这边是本机)
        try (Socket client = new Socket(InetAddress.getLocalHost(), 8888)) {

            // 2、写入数据
            BufferedWriter bufferedWriter = new BufferedWriter(new OutputStreamWriter(client.getOutputStrea
            bufferedWriter.write("I'm Client!");
            bufferedWriter.flush();

            // 3、读取数据
            BufferedReader bufferedReader = new BufferedReader(new InputStreamReader(client.getInputStream(
            System.out.println("receive msg from server: "+bufferedReader.readLine());
        }
    }
}
```

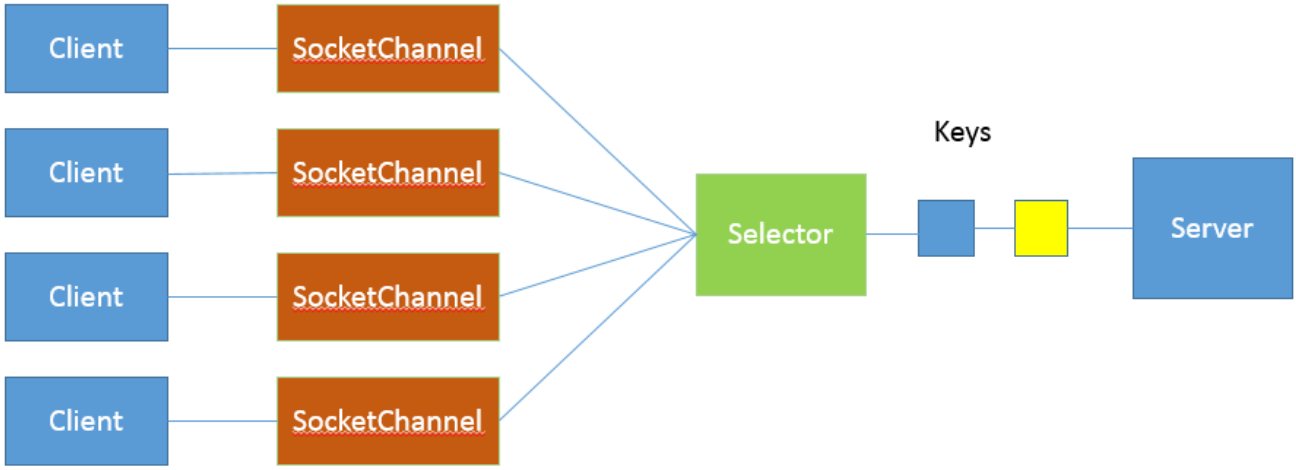
6、NIO在Socket编程上的优势

1. 使用非阻塞的IO方式
2. 支持IO多路复用
3. 这些特性改变了传统网络编程中一个线程只能管理一个链接的情况，现在可以采用一个线程管理多个链接。

7、NIO为什么比IO同步阻塞模式要更好?

1. 同步阻塞模式需要多线程来处理多任务。
2. NIO利用了单线程轮询事件的机制，高效定位就绪的Channel。
3. 仅仅是 `select` 阶段是阻塞的，可以避免大量客户端连接时，频繁切换线程带来的问题。

8、NIO实现网络通信的工作模式图



9、NIO能解决什么问题?

> 1. 服务端多线程并发处理任务，即使使用线程池，高并发处理依然会因为上下文切换，导致性能问题。
> 2. NIO是利用单线程轮询事件的机制，高效的去选择来请求连接的Channel仅提供服务。

10、NIO的请求接收和处理都是在一个线程处理，如果有多个请求的处理顺序是什么?

> 1. 多个请求会按照顺序处理
> 2. 如果一个处理具有耗时操作，会阻塞后续操作。
> 3.

11、NIO是否应该在服务端开启多线程进行处理?

> 1. 我觉得是可以的

12、NIO遇到大量耗时操作该怎么办?

> 1. 如果有大量耗时操作，那么整个 `NIO模型` 就不适用于这种场景。？？感觉可以开多线程。
> 2. 过多的耗时操作，可以采用传统的IO方式。

13、selector在单线程下的处理监听任务会成为性能瓶颈?

> 1. 是的。单线程中需要依次处理监听。会导致性能问题。
> 2. 在并发数数万、数十万的情况下，会导致性能问题。
> 3. Doug Lea推荐使用多个 `selector` ，在多个线程中并发监听Socket事件

文件拷贝(22)

1、Java有几种文件拷贝方式?哪一种效率最高?

1. java.io类库: 为源文件构建FileInputStream，为目标文件构建FileOutputStream，从input中读取，写入到output中
2. java.nio类库: transferTo或者transferFrom
3. java.nio.file.Files: Files.copy()进行拷贝

2、不同的文件复制方式，底层机制有什么不同?

3、用户态空间是什么?

1. 用户态空间-User Space
2. 用户态空间是普通应用和服务所使用的

4、内核态空间是什么?

1. 内核态空间-Kernel Space
2. 操作系统内核、硬件驱动等都运行在内核态空间

5、读写操作时的上下文切换是什么?

1. 就是内核态和用户态之间的切换。
2. 使用输入流、输出流进行读写操作时，就在进行用户态和内核态之间的上下文切换。
3. 当读取数据时，会切换至 内核态 将数据从磁盘读取到内核缓存。然后再切换到 用户态 ，将数据从内核缓存读取到用户缓存。
4. 写入操作类似, 仅仅是方向相反。
   !上下文切换
   (https://static001.geekbang.org/resource/image/6d/85/6d2368424431f1b0d2b935386324b585.png)

6、读写操作时的性能问题是什么?如何去解决?

1. 读写操作时，进行用户态和内核态的上下文切换，会带来额外的开销，从而降低IO效率。
2. 基于NIO transferTo的实现方式，在Linux和Unix上，会用到零拷贝技术。
3. 数据传输不再需要用户态参与，节省了上下文切换的开销和不必要的内存拷贝，进而可能提高应用拷贝性能。

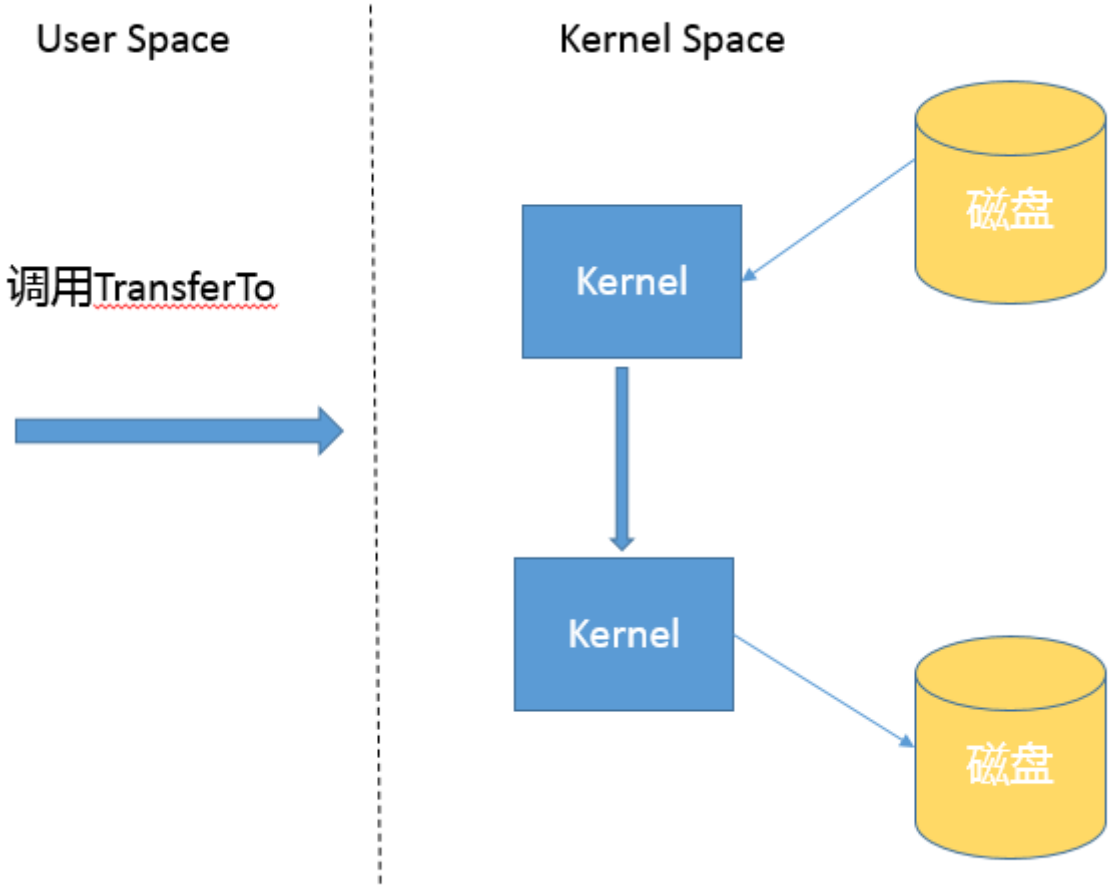
7、为什么零拷贝(zero-copy)可能有性能优势?

1. 不再需要用户态和内核态的切换
2. 减少了从 内核缓存 拷贝到 用户缓存 的这些不必要的内存拷贝。
   1. 原来是4次拷贝: 磁盘->内核缓存，内核缓存->用户缓存，用户缓存->内核缓存，内核缓存->磁盘
   1. 零拷贝只有2次拷贝: 磁盘->内核缓存，内核缓存->磁盘

8、NIO transferTo的实现方式

1. 会采用零拷贝技术

2. `transferTo` 不仅仅用在文件拷贝中，也能用于 读取磁盘文件 、然后Socket进行发送。同样性能有很大优势。



9、为什么copy要设计成需要进行上下文切换的方式？为什么不和nio的transfer一样设计为不需要用户态切换的开销？

1. 大部分工作需要用户态。为什么？
2. transfer是特定场景而不是通用长焦

BIO

10、利用javaio的InputStream和OutputStream进行文件拷贝

1-实现文件拷贝

```
    /**=======================================
     * java.io: 实现文件复制
     * @param src 源文件
     * @param dst 目标文件
     *=======================================*/
    public static void copyFileByIO(File src, File dst){
        try(InputStream inputStream = new FileInputStream(src);
            OutputStream outputStream = new FileOutputStream(dst)){

            byte[] buffer = new byte[1024];
            int length;
            // 读取数据到byte数组中，然后输出到OutStream中
            while((length = inputStream.read(buffer)) > 0){
                outputStream.write(buffer, 0, length);
            }
        } catch (FileNotFoundException e) {
            e.printStackTrace();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
```

> 2-测试程序

```
File src = new File("D:\\src.txt");
File dst = new File("D:\\dst.txt");
copyFileByIO(src, dst);
```

NIO

11、利用NIO实现文件的拷贝

```
/**==================================
 * java.nio: 实现文件复制
 * @param src 源文件
 * @param dst 目标文件
 *================================*/
public static void copyFileByChannel(File src, File dst){
    // 1、获取到源文件和目标文件的FileChannel
    try(FileChannel srcFileChannel  = new FileInputStream(src).getChannel();
        FileChannel dstFileChannel = new FileOutputStream(dst).getChannel()){
        // 2、当前FileChannel的大小
        long count = srcFileChannel.size();
        while(count > 0){
            /**============================================================
             *  3、从源文件的FileChannel中将bytes写入到目标FileChannel中
             *     1. srcFileChannel.position(): 源文件中开始的位置，不能为负
             *     2. count: 转移的最大字节数，不能为负
             *     3. dstFileChannel: 目标文件
             *============================================================*/
            long transferred = srcFileChannel.transferTo(srcFileChannel.position(),
                    count, dstFileChannel);
            // 4、传输完成后，更改源文件的position到新位置
            srcFileChannel.position(srcFileChannel.position() + transferred);
            // 5、计算出剩下多少byte需要传输
            count -= transferred;
        }

    } catch (FileNotFoundException e) {
        e.printStackTrace();
    } catch (IOException e) {
        e.printStackTrace();
    }
}
```

> 测试

```
File src = new File("D:\\src.txt");
File dst = new File("D:\\dst.txt");
// nio拷贝
copyFileByChannel(src, dst);
```

12、Nio的transferTo一定会快于BIO吗?

> 1. 需要看实际场景，比如普通的笔记本电脑。？

Files.copy

13、利用Files.copy()进行文件拷贝

```
Path srcPath = Paths.get("D:\\src.txt");
Path dstPath = Paths.get("D:\\dst.txt");
try {
  // 进行拷贝，CopyOption参数可以没有
    Files.copy(srcPath, dstPath);
} catch (IOException e) {
    e.printStackTrace();
}
```

14、Files.copy()有哪4种方法?

> 1-文件间进行copy

```
public static Path copy(Path source, Path target, CopyOption... options);
```

> 2-从输入流中copy到文件中

```
public static long copy(InputStream in, Path target, CopyOption... options);
```

> 3-从文件中copy到输出流

```
public static long copy(Path source, OutputStream out);
```

> 4-从输入流copy到输出流中

```
private static long copy(InputStream source, OutputStream sink);
```

15、Path copy(Path, Path, CopyOption...)源码分析

```
public static Path copy(Path source, Path target, CopyOption... options)
{
    FileSystemProvider provider = provider(source);
    if (provider(target) == provider) {
        // same provider-同种文件系统拷贝
        provider.copy(source, target, options);
    } else {
        // different providers-不同文件系统拷贝
        CopyMoveHelper.copyToForeignTarget(source, target, options);
    }
    return target;
}
```

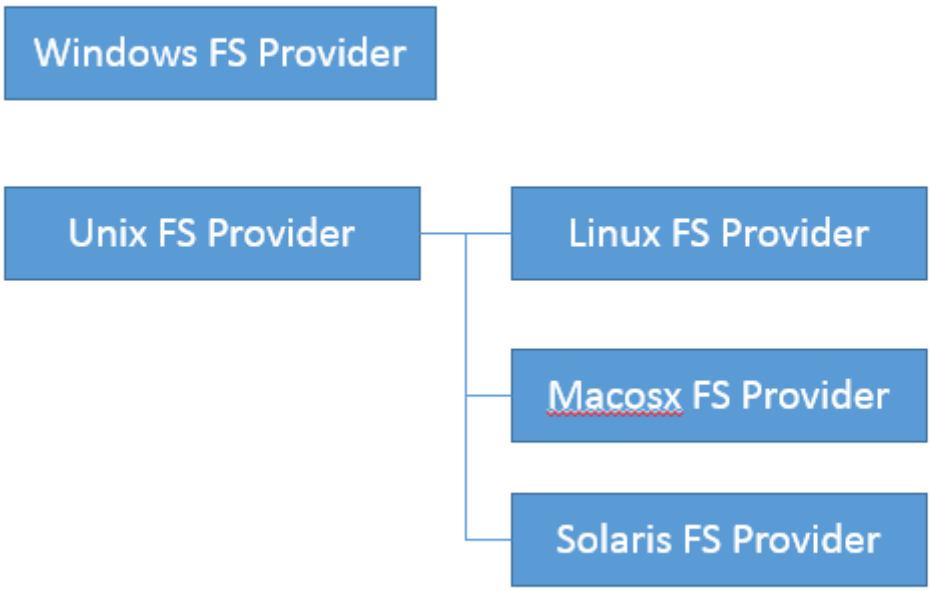
> 1. 从文件系统中最终会定位到 `UnixCopyFile.c`
> 2. 根据内部实现的说明，这只是简单的用户态空间拷贝。
> 3. 因此这个 `copy(path, path)` 并不是利用 `transferTo` ， 而是本地技术实现的 用户态拷贝

16、FileSystemProvider是什么？如何提供文件系统的?

1. 文件系统的服务的提供者
2. 是一个抽象类
3. 内部通过 ServiceLoader机制 加载一系列 文件系统 ， 然后提供服务。
4. 文件系统的实际逻辑是在JDK内部实现的，可以在JDK源码中搜索 FileSystemProvider、nio ， 可以定位到 sun/nio/fs ， 因为NIO底层和操作系统紧密相连，所以每个平台都有自己部分特有的文件系统。
5. Unix平台上会定位到 UnixFileSystemProvider -> UnixCopyFile.Transfer -> UnixCopyFile.c

```
// FileSystemProvider.java
private static List<FileSystemProvider> loadInstalledProviders() {
    //加载所有已经安装的文件系统服务的提供者
    List<FileSystemProvider> list = new ArrayList<FileSystemProvider>();
    ServiceLoader<FileSystemProvider> sl = ServiceLoader
        .load(FileSystemProvider.class, ClassLoader.getSystemClassLoader());
    //xxx
}
```

17、平台特有的文件系统服务提供者: FileSystemProvider



18、copy(InputStream, OutputStream)源码分析

1. 直接进行 inoputstream和outputstream 的read和write操作，本质和一般IO操作是一样的。
2. 也就是用户态的读写

```
private static long copy(InputStream source, OutputStream sink)
{
    long nread = 0L;
    byte[] buf = new byte[BUFFER_SIZE];
    int n;
    while ((n = source.read(buf)) > 0) {
        sink.write(buf, 0, n);
        nread += n;
    }
    return nread;
}
```

19、copy(InputStream, Path, CopyOption...)源码分析

> 本质调用的copy(InputStream, OutputStream)方法

```
public static long copy(InputStream in, Path target, CopyOption... options)
{
    // xx省略代码xx
    Objects.requireNonNull(in);
    OutputStream ostream;

    // 1、通过target的Path获取到OutputStream
    ostream = newOutputStream(target, StandardOpenOption.CREATE_NEW,
                                        StandardOpenOption.WRITE);

    // 2、然后还是直接调用copy(InputStream, OutpuStream)的方法进行数据拷贝。
    try (OutputStream out = ostream) {
        return copy(in, out);
    }
}
```

20、copy(Path, OutputStream)源码分析

> 本质调用的copy(InputStream, OutputStream)方法

```
public static long copy(Path source, OutputStream out) throws IOException {
    //1、输出流不为Null
    Objects.requireNonNull(out);
    //2、本质调用的copy(InputStream, OutputStream)方法
    try (InputStream in = newInputStream(source)) {
        return copy(in, out);
    }
}
```

21、JDK10中Files.copy()实现的轻微改变:InputStream.transferTo

> 1. copy(Path, Path, CopyOption...)内部机制没有变化
> 2. 剩余copy()方法，将输入流、输出流的读写封装到了方法中： `InputStream.transferTo()` ，也就是处于用户态的读写

```
public long transferTo(OutputStream out) throws IOException {
        Objects.requireNonNull(out, "out");
        long transferred = 0;
        byte[] buffer = new byte[DEFAULT_BUFFER_SIZE];
        int read;
        //IO读写操作
        while ((read = this.read(buffer, 0, DEFAULT_BUFFER_SIZE)) >= 0) {
            out.write(buffer, 0, read);
            transferred += read;
        }
        return transferred;
}
```

拷贝性能

22、如何提升类似拷贝等IO操作的性能?

> 1. 合理使用缓存等机制，合理减少IO次数
> 2. 使用transferTo等机制，减少上下文切换和额外的IO操作
> 3. 减少不必要的转换过程, 如：
>   1. 编解码
>   1. 对象序列化和反序列化
>   1. 操作文本文件或者网络通信，如果不是要使用文本信息，可以直接传输二进制信息。而不是传输字符串。

知识扩展

1、NIO 提供的高性能数据操作方式是基于什么原理，如何使用?

2、从开发者的角度来看，NIO 自身实现存在哪些问题？有什么改进的想法吗?

> 1. NIO的多路复用存在系能瓶颈

3、开启一个线程需要多少内存消耗？(32位和64位)

问题汇总

参考资料

1. 极客时间-Java提供了哪些IO方式？ NIO如何实现多路复用?
2. 极客时间-Java有几种文件拷贝方式？哪一种最高效?
3. Java NIO 英文博客详解
4. 【Java.NIO】Selector，及SelectionKey

5. 图解ByteBuffer
6. Java NIO Files 操作文件
7. Java NIO系列教程（十 五）Java NIO Path
8. MappedByteBuffer以及ByteBufer的底层原理
9. NIO文件锁实现进程独占