小师妹学IO系列

java中最最让人激动的部分了IO和NIO。IO的全称是input output，是java程序跟外部世界交流的桥梁，IO指的是java.io包中的所有类，他们是从java1.0开始就存在的。NIO叫做new IO，是在java1.4中引入的新一代IO。

IO的本质是什么呢？它和NIO有什么区别呢？我们该怎么学习IO和NIO呢？

本系列将会借助小师妹的视角，详细讲述学习java IO的过程，希望大家能够喜欢。

小师妹何许人也？姓名不详，但是勤奋爱学，潜力无限，一起来看看吧。

本文的例子https://github.com/ddean2009/learn-java-io-nio

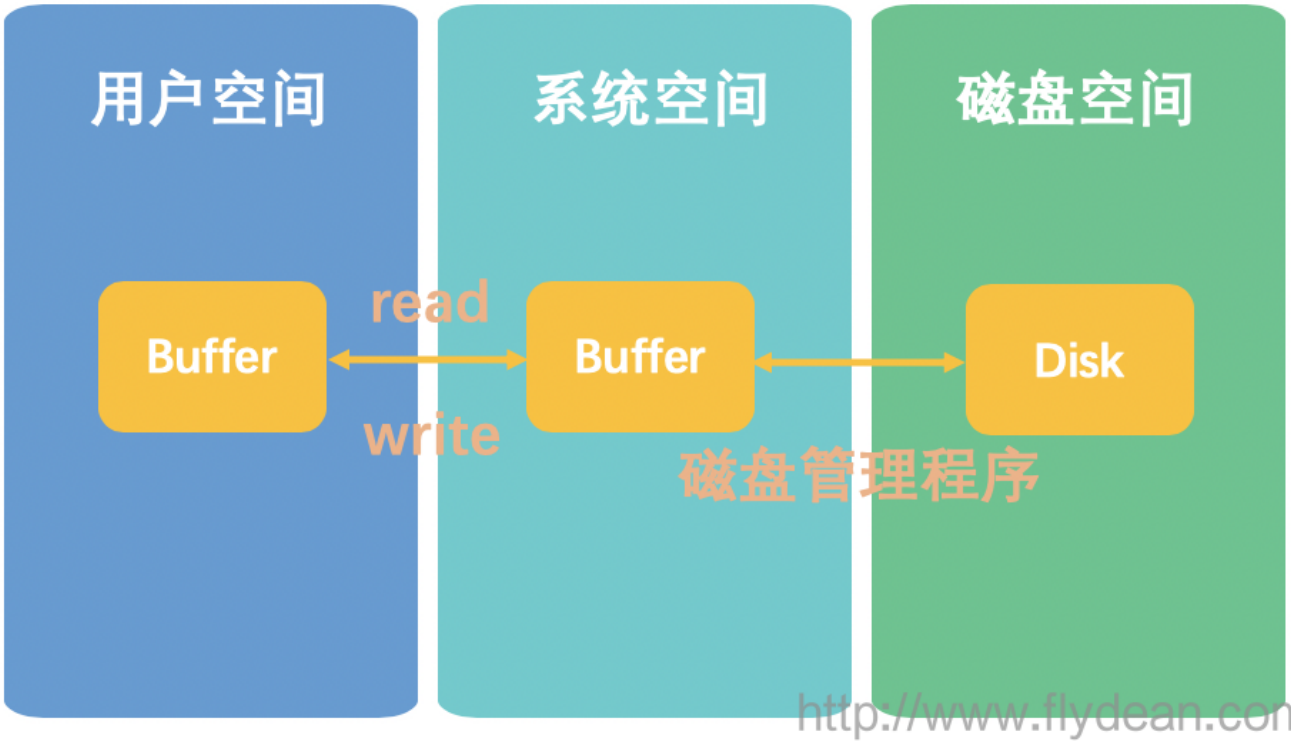
第一章 IO的本质

IO的本质

IO的作用就是从外部系统读取数据到java程序中，或者把java程序中输出的数据写回到外部系统。这里的外部系统可能是磁盘，网络流等等。

因为对所有的外部数据的处理都是由操作系统内核来实现的，对于java应用程序来说，只是调用操作系统中相应的接口方法，从而和外部数据进行交互。

所有IO的本质就是对Buffer的处理，我们把数据放入Buffer供系统写入外部数据，或者从系统Buffer中读取从外部系统中读取的数据。如下图所示：



用户空间也就是我们自己的java程序有一个Buffer，系统空间也有一个buffer。所以会出现系统空间缓存数据的情况，这种情况下系统空间将会直接返回Buffer中的数据，提升读取速度。

DMA和虚拟地址空间

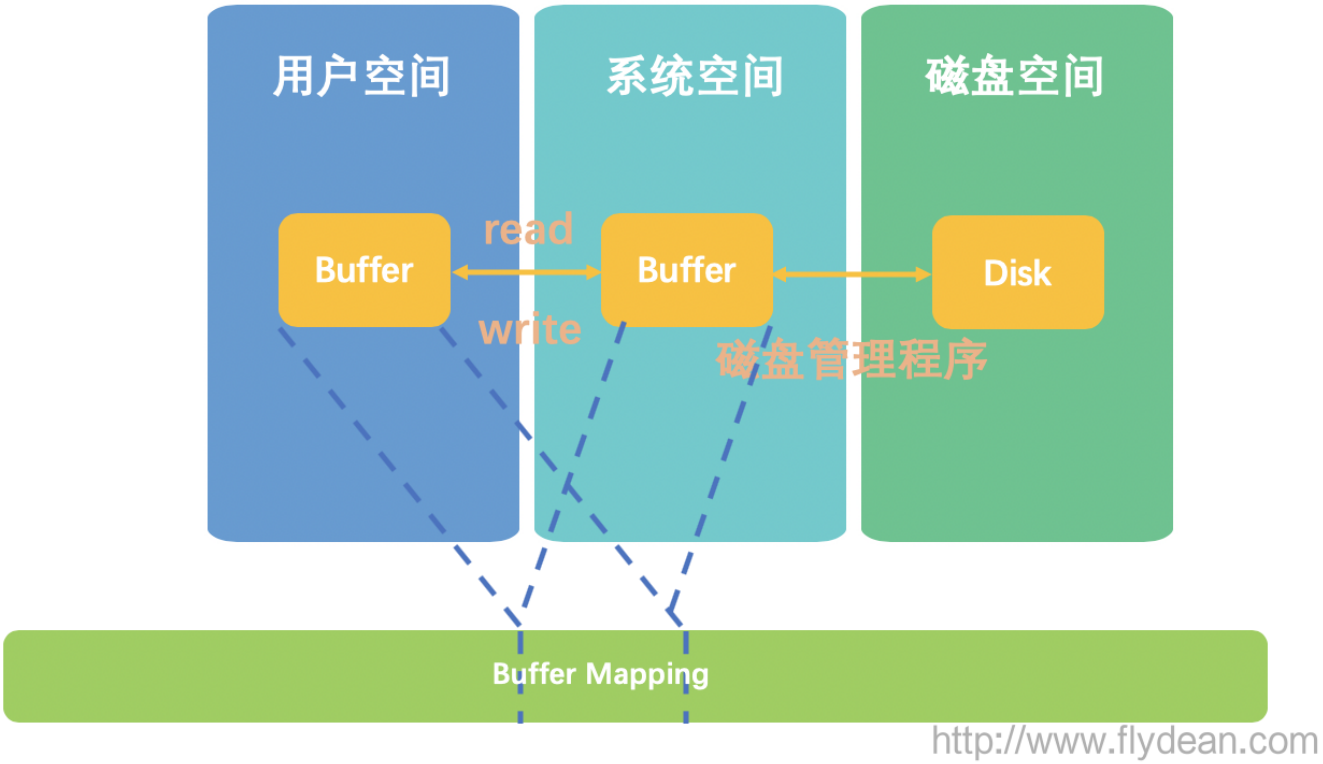
在继续讲解之前，我们先讲解两个操作系统中的基本概念，方便后面我们对IO的理解。

现代操作系统都有一个叫做DMA（Direct memory access）的组件。这个组件是做什么的呢?

一般来说对内存的读写都是要交给CPU来完成的，在没有DMA的情况下，如果程序进行IO操作，那么所有的CPU时间都会被占用，CPU没法去响应其他的任务，只能等待IO执行完成。这在现代应用程序中是无法想象的。

如果使用DMA，则CPU可以把IO操作转交给其他的操作系统组件，比如数据管理器来操作，只有当数据管理器操作完毕之后，才会通知CPU该IO操作完成。现代操作系统基本上都实现了DMA。

虚拟地址空间也叫做（Virtual address space），为了不同程序的互相隔离和保证程序中地址的确定性，现代计算机系统引入了虚拟地址空间的概念。简单点讲可以看做是跟实际物理地址的映射，通过使用分段或者分页的技术，将实际的物理地址映射到虚拟地址空间。



对于上面的IO的基本流程图中，我们可以将系统空间的buffer和用户空间的buffer同时映射到虚拟地址空间的同一个地方。这样就省略了从系统空间拷贝到用户空间的步骤。速度会更快。

同时为了解决虚拟空间比物理内存空间大的问题，现代计算机技术一般都是用了分页技术。

分页技术就是将虚拟空间分为很多个page，只有在需要用到的时候才为该page分配到物理内存的映射，这样物理内存实际上可以看做虚拟空间地址的缓存。

虚拟空间地址分页对IO的影响就在于，IO的操作也是基于page来的。

比较常用的page大小有：1,024, 2,048, 和 4,096 bytes。

IO的分类

IO可以分为File/Block IO和Stream I/O两类。

对于File/Block IO来说，数据是存储在disk中，而disk是由filesystem来进行管理的。我们可以通过filesystem来定义file的名字，路径，文件属性等内容。

filesystem通过把数据划分成为一个个的data blocks来进行管理。有些blocks存储着文件的元数据，有些block存储着真正的数据。

最后filesystem在处理数据的过程中，也进行了分页。filesystem的分页大小可以跟内存分页的大小一致，或者是它的倍数，比如 2,048 或者 8,192 bytes等。

并不是所有的数据都是以block的形式存在的，我们还有一类IO叫做stream IO。

stream IO就像是管道流，里面的数据是序列被消费的。

IO和NIO的区别

java1.0中的IO是流式IO，它只能一个字节一个字节的处理数据，所以IO也叫做Stream IO。

而NIO是为了提升IO的效率而生的，它是以Block的方式来读取数据的。

Stream IO中，input输入一个字节，output就输出一个字节，因为是Stream，所以可以加上过滤器或者过滤器链，可以想想一下web框架中的filter chain。在Stream IO中，数据只能处理一次，你不能在Stream中回退数据。

在Block IO中，数据是以block的形式来被处理的，因此其处理速度要比Stream IO快，同时可以回退处理数据。但是你需要自己处理buffer，所以复杂程度要比Stream IO高。

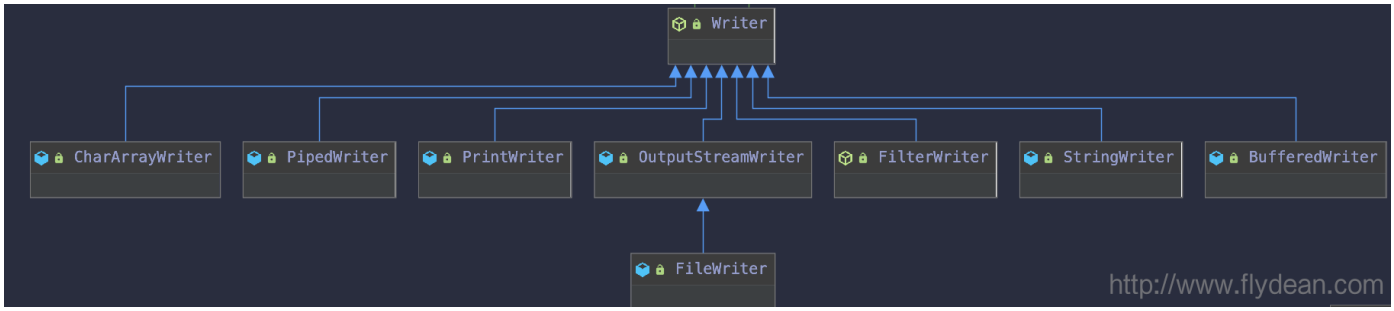
一般来说Stream IO是阻塞型IO，当线程进行读或者写操作的时候，线程会被阻塞。

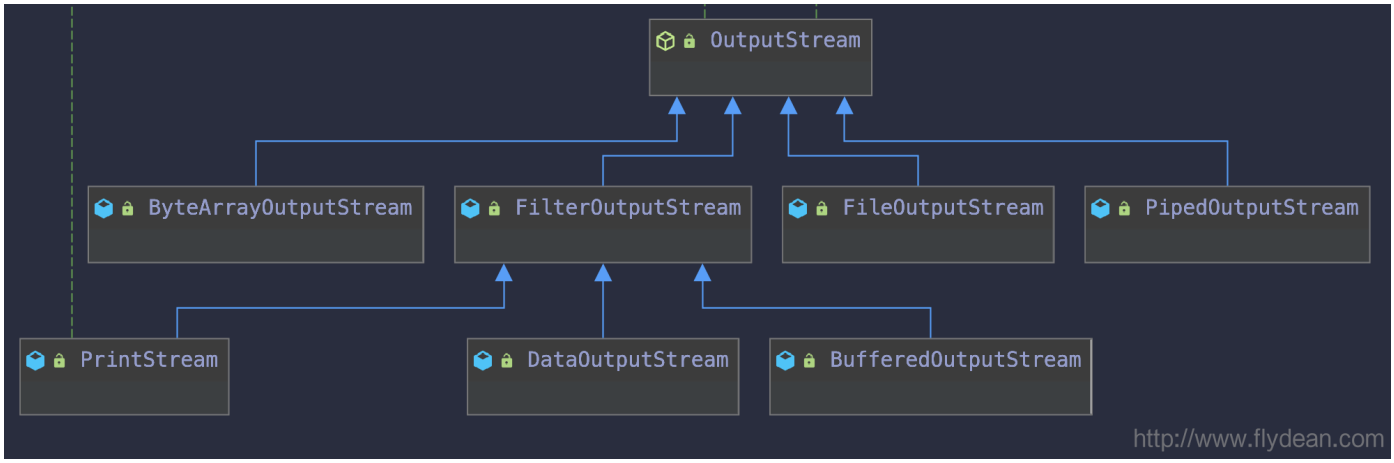
而NIO一般来说是非阻塞的，也就是说在进行读或者写的过程中可以去做其他的操作，而读或者写操作执行完毕之后会通知NIO操作的完成。

在IO中，主要分为DataOutPut和DataInput，分别对应IO的out和in。

DataOutPut有三大类，分别是Writer，OutputStream和ObjectOutput。

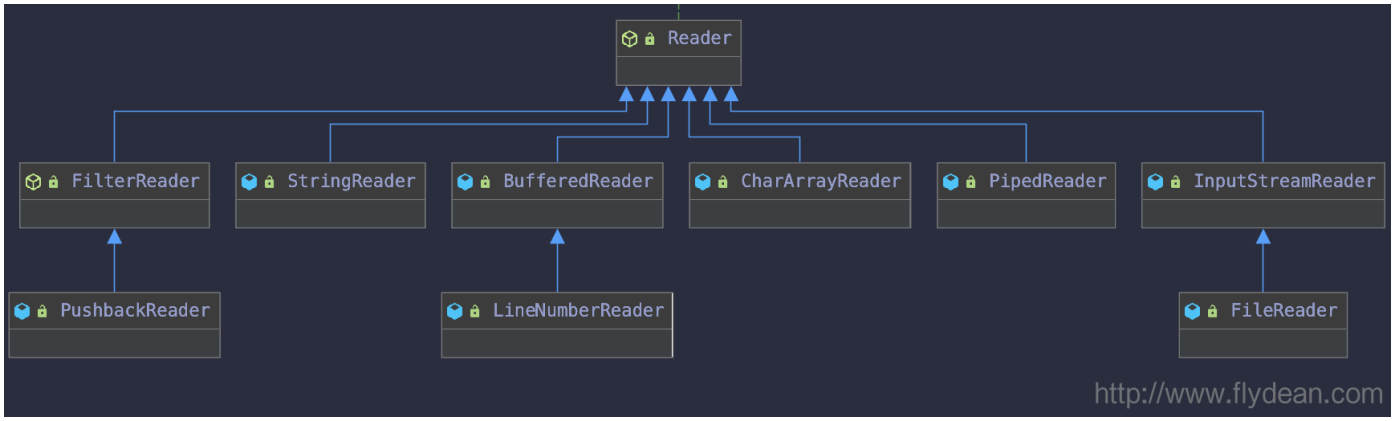
看下他们中的继承关系：

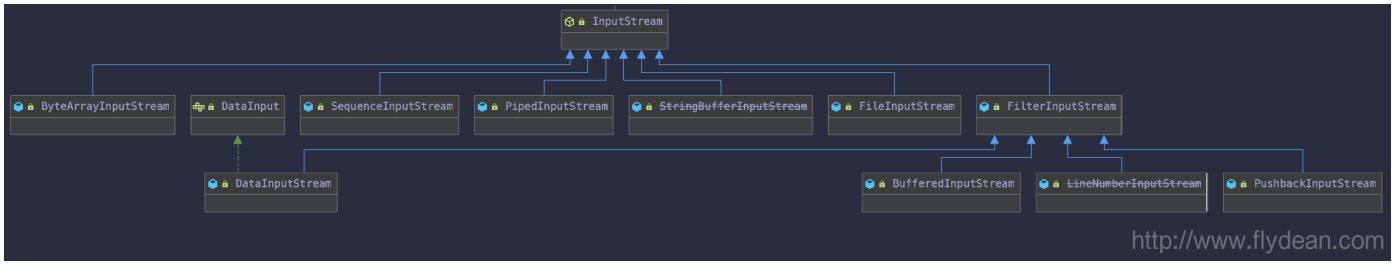




DataInput也有三大类，分别是ObjectInput，InputStream和Reader。

看看他们的继承关系：

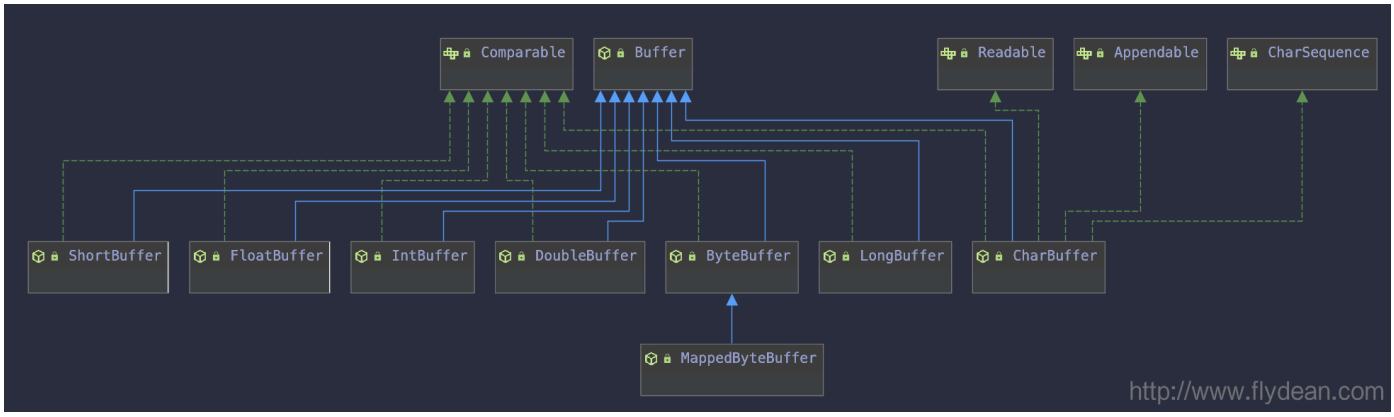




ObjectOutput和ObjectInput类比较少，这里就不列出来了。

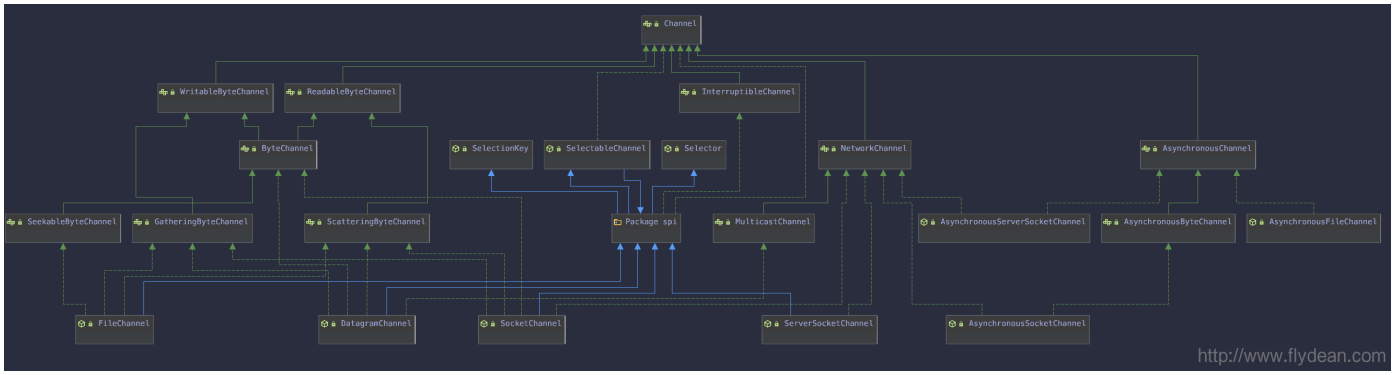
统计一下大概20个类左右，搞清楚这20个类的用处，恭喜你java IO你就懂了！

对于NIO来说比较复杂一点，首先，为了处理block的信息，需要将数据读取到buffer中，所以在NIO中Buffer是一个非常中要的概念，我们看下NIO中的Buffer：



从上图我们可以看到NIO中为我们准备了各种各样的buffer类型使用。

另外一个非常重要的概念是channel，channel是NIO获取数据的通道：



NIO需要掌握的类的个数比IO要稍稍多一点，毕竟NIO要复杂一点。

就这么几十个类，我们就掌握了IO和NIO，想想都觉得兴奋。

总结

后面的文章中，我们会介绍小师妹给你们认识，刚好她也在学java IO，后面的学习就跟她一起进行吧，敬请期待。

第二章 try with和它的底层原理

简介

小师妹是个java初学者，最近正在学习使用java IO，作为大师兄的我自然要给她最给力的支持了。一起来看看她都遇到了什么问题和问题是怎么被解决的吧。

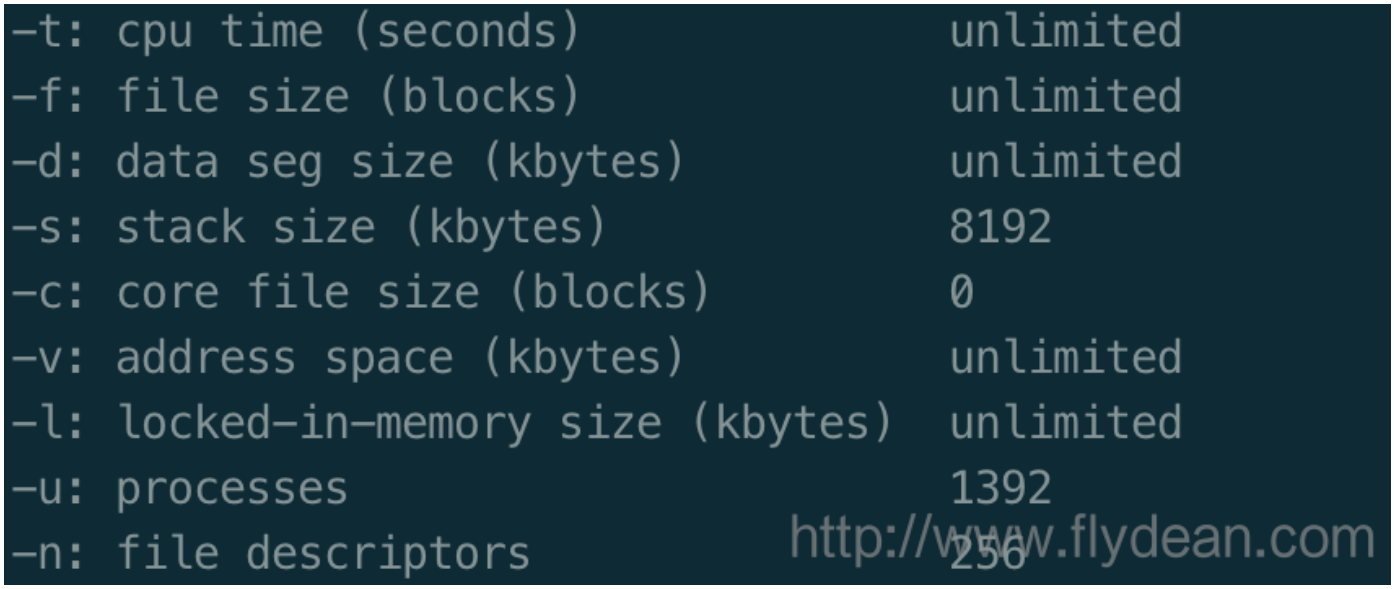
IO关闭的问题

这一天，小师妹一脸郁闷的问我：F师兄，我学Java IO也有好多天了，最近写了一个例子，读取一个文件没有问题，但是读取很多个文件就会告诉我：”Can't open so many files“，能帮我看看是什么问题吗?

> 更多内容请访问www.flydean.com

小师妹的要求当然不能拒绝，我立马响应：可能打开文件太多了吧，教你两个命令，查看最大文件打开限制。

一个命令是 ulimit -a



第二个命令是

```
ulimit -n
256
```

看起来是你的最大文件限制太小了，只有256个，调大一点就可以了。

小师妹却说：不对呀F师兄，我读文件都是一个一个读的，没有同时开这么多文件哟。

好吧，看下你写的代码吧：

```
BufferedReader bufferedReader = null;
        try {
            String line;
            bufferedReader = new BufferedReader(new
FileReader("trywith/src/main/resources/www.flydean.com"));
            while ((line = bufferedReader.readLine()) != null) {
                log.info(line);
            }
        } catch (IOException e) {
            log.error(e.getMessage(), e);
        }
```

看完代码，问题找到了，小师妹，你的IO没有关闭，应该在使用之后，在finally里面把你的reader关闭。

下面这段代码就行了：

```
BufferedReader bufferedReader = null;
        try {
            String line;
            bufferedReader = new BufferedReader(new
FileReader("trywith/src/main/resources/www.flydean.com"));
            while ((line = bufferedReader.readLine()) != null) {
                log.info(line);
            }
        } catch (IOException e) {
            log.error(e.getMessage(), e);
        } finally {
            try {
                if (bufferedReader != null){
                 bufferedReader.close();
                }
            } catch (IOException ex) {
                log.error(ex.getMessage(), ex);
            }
        }
```

小师妹道了一声谢，默默的去改代码了。

使用try with resource

过了半个小时，小师妹又来找我了，F师兄，现在每段代码都要手动添加finally，实在是太麻烦了，很多时候我又怕忘记关闭IO了，导致程序出现无法预料的异常。你也知道我这人从来就怕麻烦，有没有什么简单的办法，可以解决这个问题呢?

那么小师妹你用的JDK版本是多少?

小师妹不好意思的说：虽然最新的JDK已经到14了，我还是用的JDK8.

JDK8就够了，其实从JDK7开始，Java引入了try with resource的新功能，你把使用过后要关闭的resource放到try里面，JVM会帮你自动close的，是不是很方便，来看下面这段代码：

```
try (BufferedReader br = new BufferedReader(new
FileReader("trywith/src/main/resources/www.flydean.com")))
        {
            String sCurrentLine;
            while ((sCurrentLine = br.readLine()) != null)
            {
                log.info(sCurrentLine);
            }
        } catch (IOException e) {
            log.error(e.getMessage(), e);
        }
```

try with resource的原理

太棒了，小师妹非常开心，然后又开始问我了：F师兄，什么是resource呀？为什么放到try里面就可以不用自己close了？

resource就是资源，可以打开个关闭，我们可以把实现了java.lang.AutoCloseable接口的类都叫做resource。

先看下AutoCloseable的定义：

```
public interface AutoCloseable {
        void close() throws Exception;
}
```

AutoCloseable定义了一个close()方法，当我们在try with resource中打开了AutoCloseable的资源，那么当try block执行结束的时候，JVM会自动调用这个close（）方法来关闭资源。

我们看下上面的BufferedReader中close方法是怎么实现的：

```
public void close() throws IOException {
    synchronized (lock) {
        if (in == null)
            return;
        in.close();
        in = null;
        cb = null;
    }
}
```

自定义resource

小师妹恍然大悟：F师兄，那么我们是不是可以实现AutoCloseable来创建自己的resource呢？

当然可以了，我们举个例子，比如给你解答完这个问题，我就要去吃饭了，我们定义这样一个resource类：

```
public class CustResource implements AutoCloseable {

    public void helpSister(){
        log.info("帮助小师妹解决问题！");
    }

    @Override
    public void close() throws Exception {
        log.info("解决完问题，赶紧去吃饭！");
    }

    public static void main(String[] args) throws Exception {
       try( CustResource custResource= new CustResource()){
           custResource.helpSister();
       }
```

```
    }
}
```

运行输出结果：

```
[main] INFO com.flydean.CustResource - 帮助小师妹解决问题!
[main] INFO com.flydean.CustResource - 解决完问题，赶紧去吃饭!
```

总结

最后，小师妹的问题解决了，我也可以按时吃饭了。

第三章 File文件系统

简介

小师妹又遇到难题了，这次的问题是有关文件的创建，文件权限和文件系统相关的问题，还好这些问题的答案都在我的脑子里面，一起来看看吧。

文件权限和文件系统

早上刚到公司，小师妹就凑过来神神秘秘的问我：F师兄，我在服务器上面放了一些重要的文件，是非常非常重要的那种，有没有什么办法给它加个保护，还兼顾一点隐私？

> 更多内容请访问www.flydean.com

什么文件这么重要呀？不会是你的照片吧，放心没人会感兴趣的。

小师妹说：当然不是，我要把我的学习心得放上去，但是F师兄你知道的，我刚刚开始学习，很多想法都不太成熟，想先保个密，后面再公开。

看到小师妹这么有上进心，我老泪纵横，心里很是安慰。那就开始吧。

你知道，这个世界上操作系统分为两类，windows和linux（unix）系统。两个系统是有很大区别的，但两个系统都有一个文件的概念，当然linux中文件的范围更加广泛，几乎所有的资源都可以看做是文件。

有文件就有对应的文件系统，这些文件系统是由系统内核支持的，并不需要我们在java程序中重复造轮子，直接调用系统的内核接口就可以了。

小师妹：F师兄，这个我懂，我们不重复造轮子，我们只是轮子的搬运工。那么java是怎么调用系统内核来创建文件的呢？

创建文件最常用的方法就是调用File类中的createNewFile方法，我们看下这个方法的实现：

```
public boolean createNewFile() throws IOException {
        SecurityManager security = System.getSecurityManager();
        if (security != null) security.checkWrite(path);
        if (isInvalid()) {
            throw new IOException("Invalid file path");
        }
        return fs.createFileExclusively(path);
    }
```

方法内部先进行了安全性检测，如果通过了安全性检测就会调用FileSystem的createFileExclusively方法来创建文件。

在我的mac环境中，FileSystem的实现类是UnixFileSystem：

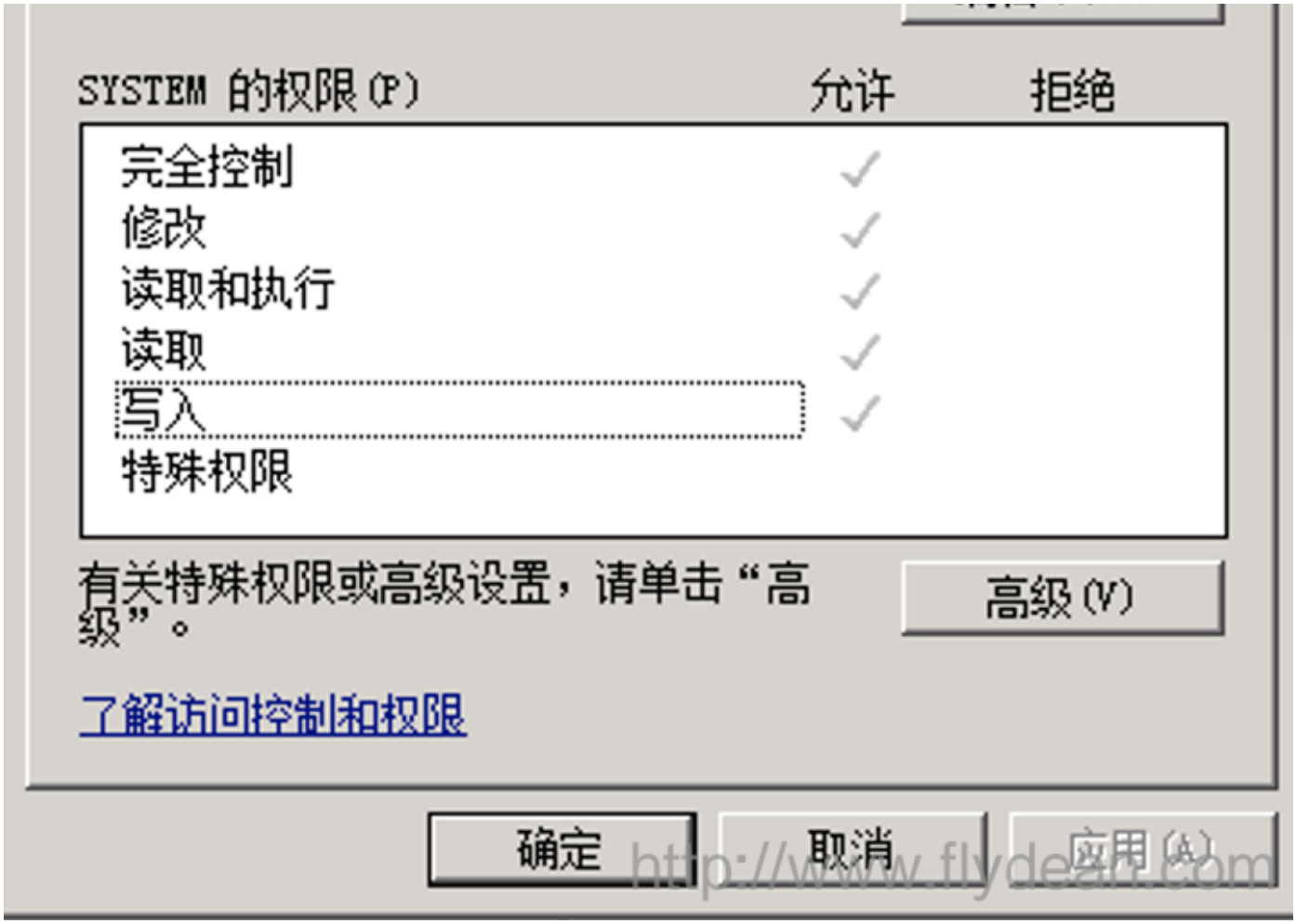
```
public native boolean createFileExclusively(String path)
        throws IOException;
```

看到了吗？UnixFileSystem中的createFileExclusively是一个native方法，它会去调用底层的系统接口。

小师妹：哇，文件创建好了，我们就可以给文件赋权限了，但是windows和linux的权限是一样的吗？

这个问题问得好，java代码是跨平台的，我们的代码需要同时在windows和linux上的JVM执行，所以必须找到他们权限的共同点。

我们先看一下windows文件的权限：



可以看到一个windows文件的权限可以有修改，读取和执行三种，特殊权限我们先不用考虑，因为我们需要找到windows和linux的共同点。

再看下linux文件的权限：

```
 ls -al www.flydean.com
-rw-r--r--  1 flydean  staff  15 May 14 15:43 www.flydean.com
```

上面我使用了一个ll命令列出了www.flydean.com这个文件的详细信息。 其中第一列就是文件的权限了。

linux的基本文件权限可以分为三部分，分别是owner，group，others，每部分和windows一样都有读，写和执行的权限，分别用rwx来表示。

三部分的权限连起来就成了rwxrwxrwx，对比上面我们的输出结果，我们可以看到www.flydean.com这个文件对owner自己是可读写的，对Group用户是只读的，对other用户也是只读的。

你要想把文件只对自己可读，那么可以执行下面的命令：

```
chmod 600 www.flydean.com
```

小师妹立马激动起来：F师兄，这个我懂，6用二进制表示就是110，600用二进制表示就是110000000，刚刚好对应rw-------。

对于小师妹的领悟能力，我感到非常满意。

文件的创建

虽然我们已经不是孔乙己时代了，不需要知道茴字的四种写法，但是多一条知识多一条路，做些充足的准备还是非常有必要的。

小师妹，那你知道在java中有哪几种文件的创建方法呢?

小师妹小声道：F师兄，我只知道一种new File的方法。

我满意的抚摸着我的胡子，显示一下自己高人的气场。

之前我们讲过了，IO有三大类，一种是Reader/Writer，一种是InputStream/OutputStream,最后一种是ObjectReader/ObjectWriter。

除了使用第一种new File之外，我们还可以使用OutputStream来实现，当然我们还要用到之前讲到try with resource特性，让代码更加简洁。

先看第一种方式：

```
public void createFileWithFile() throws IOException {
        File file = new File("file/src/main/resources/www.flydean.com");
        //Create the file
        if (file.createNewFile()){
            log.info("恭喜，文件创建成功");
        }else{
            log.info("不好意思，文件创建失败");
        }
        //Write Content
        try(FileWriter writer = new FileWriter(file)){
            writer.write("www.flydean.com");
        }
    }
```

再看第二种方式：

```
public  void createFileWithStream() throws IOException
    {
        String data = "www.flydean.com";
        try(FileOutputStream out = new
FileOutputStream("file/src/main/resources/www.flydean.com")){
            out.write(data.getBytes());
        }
    }
```

第二种方式看起来比第一种方式更加简介。

小师妹：慢着，F师兄，JDK7中NIO就已经出现了，能不能使用NIO来创建文件呢?

这个问题当然难不到我：

```
public void createFileWithNIO()  throws IOException
    {
        String data = "www.flydean.com";
        Files.write(Paths.get("file/src/main/resources/www.flydean.com"),
data.getBytes());

        List<String> lines = Arrays.asList("程序那些事", "www.flydean.com");
        Files.write(Paths.get("file/src/main/resources/www.flydean.com"),
                lines,
                StandardCharsets.UTF_8,
                StandardOpenOption.CREATE,
                StandardOpenOption.APPEND);
    }
```

NIO中提供了Files工具类来实现对文件的写操作，写的时候我们还可以带点参数，比如字符编码，是替换文件还是在append到文件后面等等。

代码中文件的权限

小师妹又有问题了：F师兄，讲了半天，还没有给我讲权限的事情啦。

别急，现在就讲权限：

```
public void fileWithPromission() throws IOException {
        File file = File.createTempFile("file/src/main/resources/www.flydean.com","");
        log.info("{}",file.exists());

        file.setExecutable(true);
        file.setReadable(true,true);
        file.setWritable(true);
        log.info("{}",file.canExecute());
        log.info("{}",file.canRead());
        log.info("{}",file.canWrite());

        Path path = Files.createTempFile("file/src/main/resources/www.flydean.com",
"");
        log.info("{}",Files.exists(path));
        log.info("{}",Files.isReadable(path));
        log.info("{}",Files.isWritable(path));
        log.info("{}",Files.isExecutable(path));
    }
```

上面我们讲过了，JVM为了通用，只能取windows和linux都有的功能，那就是说权限只有读写和执行权限，因为windows里面也可以区分本用户或者其他用户，所以是否是本用户的权限也保留了。

上面的例子我们使用了传统的File和NIO中的Files来更新文件的权限。

总结

好了，文件的权限就先讲到这里了。

第四章 文件读取那些事

简介

小师妹最新对java IO中的reader和stream产生了一点点困惑，不知道到底该用哪一个才对，怎么读取文件才是正确的姿势呢？今天F师兄现场为她解答。

字符和字节

小师妹最近很迷糊：F师兄，上次你讲到IO的读取分为两大类，分别是Reader，InputStream，这两大类有什么区别吗？为什么我看到有些类即是Reader又是Stream？比如：InputStreamReader?

小师妹，你知道哲学家的终极三问吗？你是谁？从哪里来？到哪里去？

F师兄，你是不是迷糊了，我在问你java，你扯什么哲学。

小师妹，其实吧，哲学是一切学问的基础，你知道科学原理的英文怎么翻译吗？the philosophy of science，科学的原理就是哲学。

你看计算机中代码的本质是什么？代码的本质就是0和1组成的一串长长的二进制数，这么多二进制数组合起来就成了计算机中的代码，也就是JVM可以识别可以运行的二进制代码。

> 更多内容请访问www.flydean.com

小师妹一脸崇拜：F师兄说的好像很有道理，但是这和Reader，InputStream有什么关系呢？

别急，冥冥中自有定数，先问你一个问题，java中存储的最小单位是什么？

小师妹：容我想想，java中最小的应该是boolean，true和false正好和二进制1，0对应。

对了一半，虽然boolean也是java中存储的最小单位，但是它需要占用一个字节Byte的空间。java中最小的存储单位其实是字节Byte。不信的话可以用之前我介绍的JOL工具来验证一下：

```
[main] INFO com.flydean.JolUsage - java.lang.Boolean object internals:
 OFFSET  SIZE      TYPE DESCRIPTION                               VALUE
      0    12           (object header)                           N/A
     12     1   boolean Boolean.value                              N/A
     13     3           (loss due to the next object alignment)
Instance size: 16 bytes
Space losses: 0 bytes internal + 3 bytes external = 3 bytes total
```

上面是装箱过后的Boolean，可以看到虽然Boolean最后占用16bytes，但是里面的boolean只有1byte。

byte翻译成中文就是字节，字节是java中存储的基本单位。

有了字节，我们就可以解释字符了，字符就是由字节组成的，根据编码方式的不同，字符可以有1个，2个或者多个字节组成。我们人类可以肉眼识别的汉字呀，英文什么的都可以看做是字符。

而Reader就是按照一定编码格式读取的字符，而InputStream就是直接读取的更加底层的字节。

小师妹：我懂了，如果是文本文件我们就可以用Reader，非文本文件我们就可以用InputStream。

孺子可教，小师妹进步的很快。

按字符读取的方式

小师妹，接下来F师兄给你讲下按字符读取文件的几种方式，第一种就是使用FileReader来读取File，但是FileReader本身并没有提供任何读取数据的方法，想要真正的读取数据，我们还是要用到BufferedReader来连接FileReader，BufferedReader提供了读取的缓存，可以一次读取一行：

```
public void withFileReader() throws IOException {
        File file = new File("src/main/resources/www.flydean.com");

        try (FileReader fr = new FileReader(file); BufferedReader br = new
BufferedReader(fr)) {
            String line;
            while ((line = br.readLine()) != null) {
                if (line.contains("www.flydean.com")) {
                    log.info(line);
                }
            }
        }
    }
```

每次读取一行，可以把这些行连起来就组成了stream，通过Files.lines，我们获取到了一个stream，在stream中我们就可以使用lambda表达式来读取文件了，这是谓第二种方式：

```
public void withStream() throws IOException {
        Path filePath = Paths.get("src/main/resources", "www.flydean.com");
        try (Stream<String> lines = Files.lines(filePath))
        {
            List<String> filteredLines = lines.filter(s ->
s.contains("www.flydean.com"))
                    .collect(Collectors.toList());
            filteredLines.forEach(log::info);
        }
    }
```

第三种其实并不常用，但是师兄也想教给你。这一种方式就是用工具类中的Scanner。通过Scanner可以通过换行符来分割文件，用起来也不错：

```
public void withScanner() throws FileNotFoundException {
        FileInputStream fin = new FileInputStream(new
File("src/main/resources/www.flydean.com"));
        Scanner scanner = new Scanner(fin,"UTF-8").useDelimiter("\n");
        String theString = scanner.hasNext() ? scanner.next() : "";
        log.info(theString);
        scanner.close();
    }
```

按字节读取的方式

小师妹听得很满足，连忙催促我：F师兄，字符读取方式我都懂了，快将字节读取吧。

我点了点头，小师妹，哲学的本质还记得吗？字节就是java存储的本质。掌握到本质才能勘破一切虚伪。

还记得之前讲过的Files工具类吗？这个工具类提供了很多文件操作相关的方法，其中就有读取所有bytes的方法，小师妹要注意了，这里是一次性读取所有的字节！一定要慎用，只可用于文件较少的场景，切记切记。

```
public void readBytes() throws IOException {
        Path path = Paths.get("src/main/resources/www.flydean.com");
        byte[] data = Files.readAllBytes(path);
        log.info("{}",data);
    }
```

如果是比较大的文件，那么可以使用FileInputStream来一次读取一定数量的bytes：

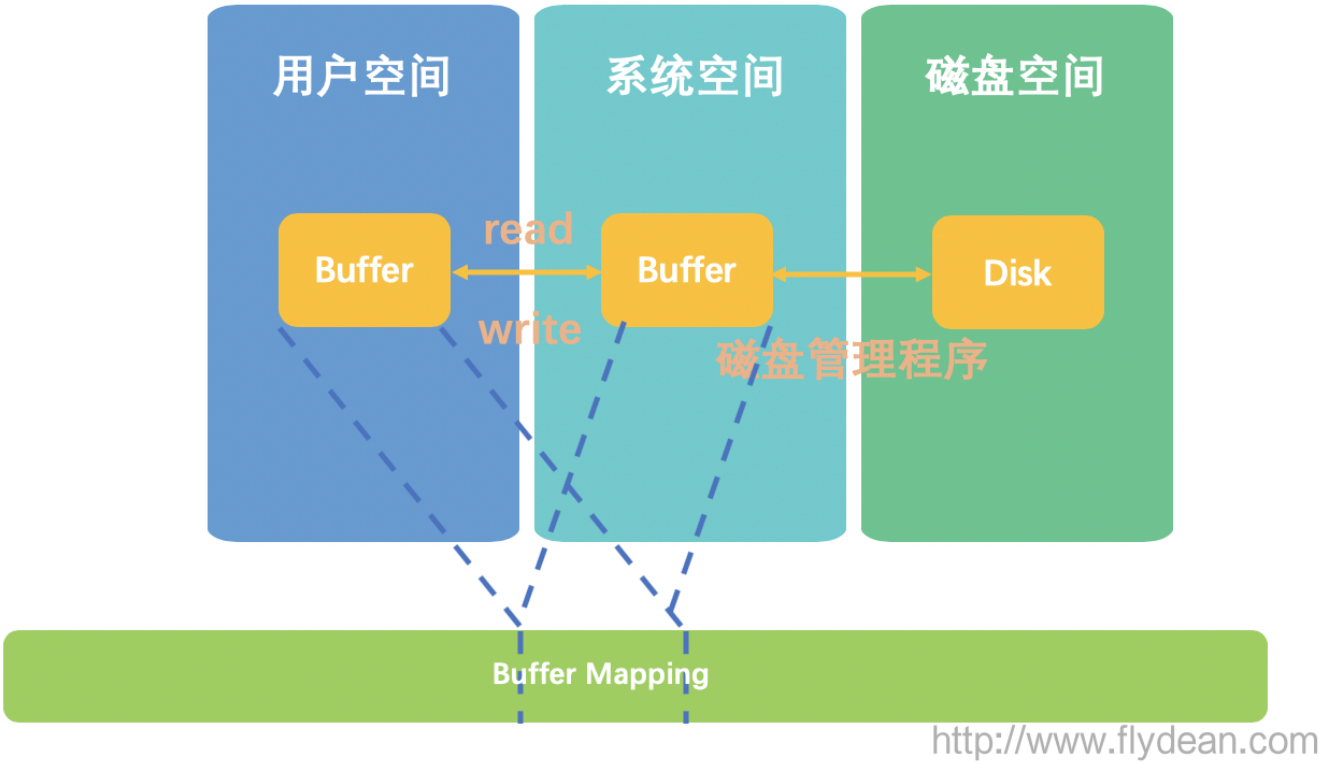
```
public void readWithStream() throws IOException {
        File file = new File("src/main/resources/www.flydean.com");
        byte[] bFile = new byte[(int) file.length()];
        try(FileInputStream fileInputStream  = new FileInputStream(file))
        {
            fileInputStream.read(bFile);
            for (int i = 0; i < bFile.length; i++) {
                log.info("{}",bFile[i]);
            }
        }
    }
```

Stream读取都是一个字节一个字节来读的，这样做会比较慢，我们使用NIO中的FileChannel和ByteBuffer来加快一些读取速度：

```
public void readWithBlock() throws IOException {
        try (RandomAccessFile aFile = new
RandomAccessFile("src/main/resources/www.flydean.com", "r");
             FileChannel inChannel = aFile.getChannel();) {
            ByteBuffer buffer = ByteBuffer.allocate(1024);
            while (inChannel.read(buffer) > 0) {
                buffer.flip();
                for (int i = 0; i < buffer.limit(); i++) {
                    log.info("{}", buffer.get());
                }
                buffer.clear();
            }
        }
    }
```

小师妹：如果是非常非常大的文件的读取，有没有更快的方法呢？

当然有，记得上次我们讲过的虚拟地址空间的映射吧：



我们可以直接将用户的地址空间和系统的地址空间同时map到同一个虚拟地址内存中，这样就免除了拷贝带来的性能开销：

```
public void copyWithMap() throws IOException{
        try (RandomAccessFile aFile = new
RandomAccessFile("src/main/resources/www.flydean.com", "r");
             FileChannel inChannel = aFile.getChannel()) {
             MappedByteBuffer buffer = inChannel.map(FileChannel.MapMode.READ_ONLY, 0,
inChannel.size());
             buffer.load();
            for (int i = 0; i < buffer.limit(); i++)
            {
                log.info("{}", buffer.get());
            }
            buffer.clear();
        }
    }
```

寻找出错的行数

小师妹：好赞！F师兄你讲得真好，小师妹我还有一个问题：最近在做文件解析，有些文件格式不规范，解析到一半就解析失败了，但是也没有个错误提示到底错在哪一行，很难定位问题呀，有没有什么好的解决办法?

看看天色已经不早了，师兄就再教你一个方法，java中有一个类叫做LineNumberReader，使用它来读取文件可以打印出行号，是不是就满足了你的需求：

```
public void useLineNumberReader() throws IOException {
```

```
        try(LineNumberReader lineNumberReader = new LineNumberReader(new
FileReader("src/main/resources/www.flydean.com")))
        {
            //输出初始行数
            log.info("Line {}" , lineNumberReader.getLineNumber());
            //重置行数
            lineNumberReader.setLineNumber(2);
            //获取现有行数
            log.info("Line {} ", lineNumberReader.getLineNumber());
            //读取所有文件内容
            String line = null;
            while ((line = lineNumberReader.readLine()) != null)
            {
                log.info("Line {} is : {}" , lineNumberReader.getLineNumber() , line);
            }
        }
    }
```

总结

今天给小师妹讲解了字符流和字节流，还讲解了文件读取的基本方法，不虚此行。

第五章 文件写入那些事

简介

小师妹又对F师兄提了一大堆奇奇怪怪的需求，要格式化输出，要特定的编码输出，要自己定位输出，什么？还要阅后即焚？大家看F师兄怎么一一接招吧。

字符输出和字节输出

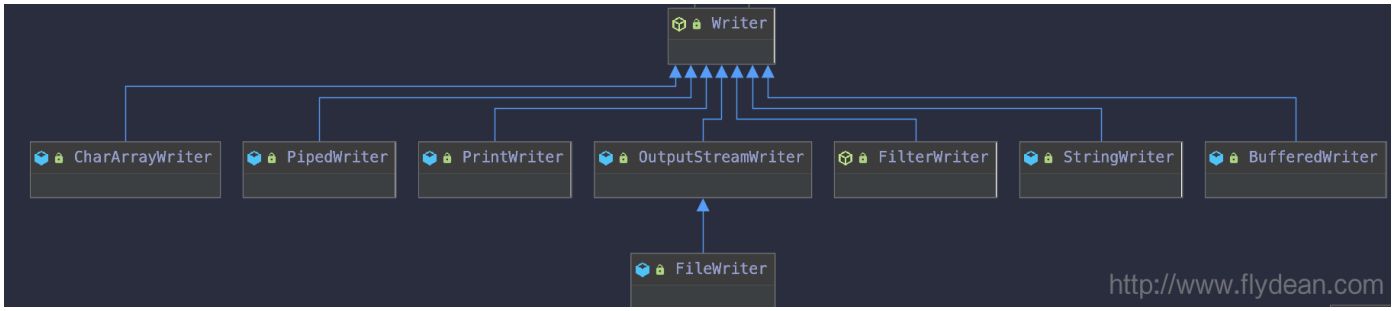
小师妹：F师兄，上次你的IO讲到了一半，文件读取是基本上讲完了，但是文件的写入还没有讲，什么时候给小师妹我再科普科普？

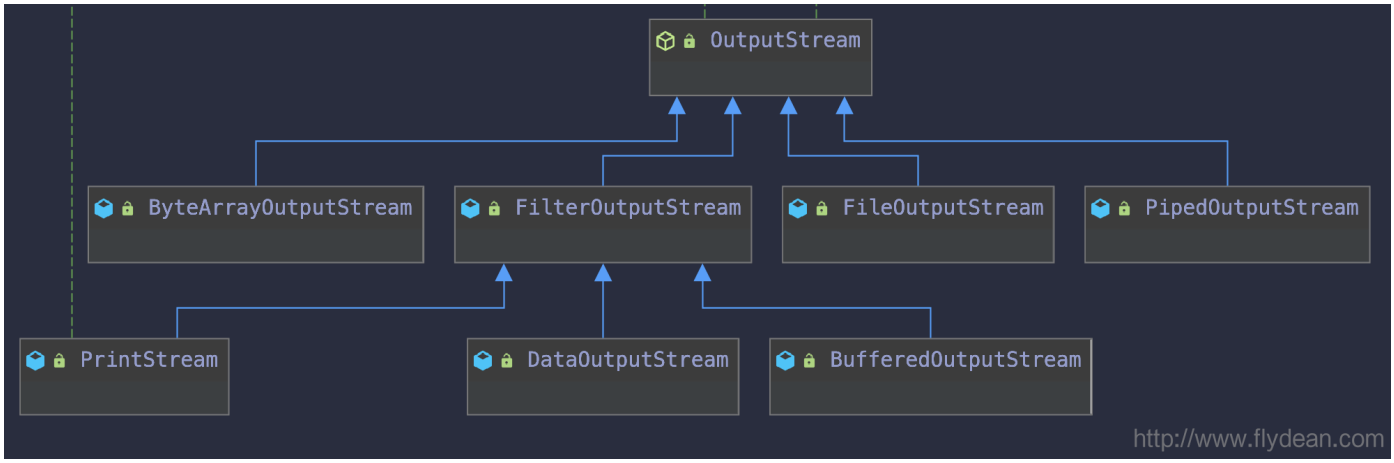
小师妹：F师兄，你知道我这个人一直以来都是勤奋好学的典范，是老师们眼中的好学生,同学们心中的好榜样,父母身边乖巧的好孩子。在我永攀科学高峰的时候，居然发现还有一半的知识没有获取，真是让我扼腕叹息，F师兄，快快把知识传给我吧。

小师妹你的请求，师兄我自当尽力办到，但是我怎么记得上次讲IO文件读取已经过了好几天了，怎么今天你才来找我。

小师妹红着脸：F师兄，这不是使用的时候遇到了点问题，才想找你把知识再复习一遍。

那先把输出类的结构再过一遍：





上面就是输出的两大系统了：Writer和OutputStream。

Writer主要针对于字符，而Stream主要针对Bytes。

Writer中最最常用的就是FileWriter和BufferedWriter,我们看下一个最基本写入的例子：

```
public void useBufferedWriter() throws IOException {
        String content = "www.flydean.com";
        File file = new File("src/main/resources/www.flydean.com");

        FileWriter fw = new FileWriter(file);
        try(BufferedWriter bw = new BufferedWriter(fw)){
            bw.write(content);
        }
    }
```

BufferedWriter是对FileWriter的封装，它提供了一定的buffer机制，可以提高写入的效率。

其实BufferedWriter提供了三种写入的方式：

```
public void write(int c)
public void write(char cbuf[], int off, int len)
public void write(String s, int off, int len)
```

第一个方法传入一个int，第二个方法传入字符数组和开始读取的位置和长度，第三个方法传入字符串和开始读取的位置和长度。是不是很简单，完全可以理解?

小师妹：不对呀，F师兄，后面两个方法的参数，不管是char和String都是字符我可以理解，第一个方法传入int是什么鬼?

小师妹，之前跟你讲的道理是不是都忘记的差不多了，int的底层存储是bytes,char和String的底层存储也是bytes，我们把int和char做个强制转换就行了。我们看下是怎么转换的：

```
public void write(int c) throws IOException {
        synchronized (lock) {
            ensureOpen();
            if (nextChar >= nChars)
                flushBuffer();
            cb[nextChar++] = (char) c;
        }
    }
```

还记得int需要占用多少个字节吗？4个，char需要占用2个字节。这样强制从int转换到char会有精度丢失的问题，只会保留低位的2个字节的数据，高位的两个字节的数据会被丢弃，这个需要在使用中注意。

看完Writer，我们再来看看Stream：

```
public void useFileOutputStream() throws IOException {
        String str = "www.flydean.com";
        try(FileOutputStream outputStream = new
FileOutputStream("src/main/resources/www.flydean.com");
            BufferedOutputStream bufferedOutputStream= new
BufferedOutputStream(outputStream)){
            byte[] strToBytes = str.getBytes();
            bufferedOutputStream.write(strToBytes);
        }
    }
```

跟Writer一样，BufferedOutputStream也是对FileOutputStream的封装，我们看下BufferedOutputStream中提供的write方法：

```
public synchronized void write(int b)
public synchronized void write(byte b[], int off, int len)
```

比较一下和Writer的区别，BufferedOutputStream的方法是synchronized的，并且BufferedOutputStream是直接对byte进行操作的。

第一个write方法传入int参数也是需要进行截取的，不过这次是从int转换成byte。

格式化输出

小师妹：F师兄，我们经常用的System.out.println可以直接向标准输出中输出格式化过后的字符串，文件的写入是不是也有类似的功能呢？

肯定有，PrintWriter就是做格式化输出用的：

```
public void usePrintWriter() throws IOException {
        FileWriter fileWriter = new FileWriter("src/main/resources/www.flydean.com");
        try(PrintWriter printWriter = new PrintWriter(fileWriter)){
            printWriter.print("www.flydean.com");
            printWriter.printf("程序那些事 %s ", "非常棒");
        }
    }
```

输出其他对象

小师妹：F师兄，我们看到可以输出String，char还有Byte，那可不可以输出Integer,Long等基础类型呢?

可以的，使用DataOutputStream就可以做到：

```
public void useDataOutPutStream()
            throws IOException {
        String value = "www.flydean.com";
        try(FileOutputStream fos = new
FileOutputStream("src/main/resources/www.flydean.com")){
            DataOutputStream outStream = new DataOutputStream(new
BufferedOutputStream(fos));
            outStream.writeUTF(value);
        }
    }
```

DataOutputStream提供了writeLong,writeDouble,writeFloat等等方法，还可以writeUTF!

在特定的位置写入

小师妹：F师兄，有时候我们不需要每次都从头开始写入到文件，能不能自定义在什么位置写入呢?

使用RandomAccessFile就可以了：

```
public void useRandomAccess() throws IOException {
        try(RandomAccessFile writer = new
RandomAccessFile("src/main/resources/www.flydean.com", "rw")){
            writer.seek(100);
            writer.writeInt(50);
        }
    }
```

RandomAccessFile可以通过seek来定位，然后通过write方法从指定的位置写入。

给文件加锁

小师妹：F师兄，最后还有一个问题，怎么保证我在进行文件写的时候别人不会覆盖我写的内容，不会产生冲突呢?

FileChannel可以调用tryLock方法来获得一个FileLock锁，通过这个锁，我们可以控制文件的访问。

```
public void useFileLock()
            throws IOException {
        try(RandomAccessFile stream = new
RandomAccessFile("src/main/resources/www.flydean.com", "rw");
        FileChannel channel = stream.getChannel()){
            FileLock lock = null;
            try {
                lock = channel.tryLock();
            } catch (final OverlappingFileLockException e) {
                stream.close();
                channel.close();
            }
            stream.writeChars("www.flydean.com");
            lock.release();
        }
    }
```

总结

今天给小师妹将了好多种文件的写的方法，够她学习一阵子了。

第六章 目录还是文件

简介

目录和文件傻傻分不清楚，目录和文件的本质到底是什么？在java中怎么操纵目录，怎么遍历目录。本文F师兄会为大家一一讲述。

linux中的文件和目录

小师妹：F师兄，我最近有一个疑惑，java代码中好像只有文件没有目录呀，是不是当初发明java的大神，一步小心走了神?

F师兄:小师妹真勇气可嘉呀，敢于质疑权威是从小工到专家的最重要的一步。想想F师兄我，从小没人提点，老师讲什么我就信什么，专家说什么我就听什么:股市必上一万点，房子是给人住的不是给人炒的,原油宝当然是小白理财必备产品....然后，就没有然后了。

> 更多内容请访问www.flydean.com

虽然java中没有目录的概念只有File文件，而File其实是可以表示目录的：

```
public boolean isDirectory()
```

File中有个isDirectory方法，可以判断该File是否是目录。

File和目录傻傻分不清楚，小师妹，有没有联想到点什么?

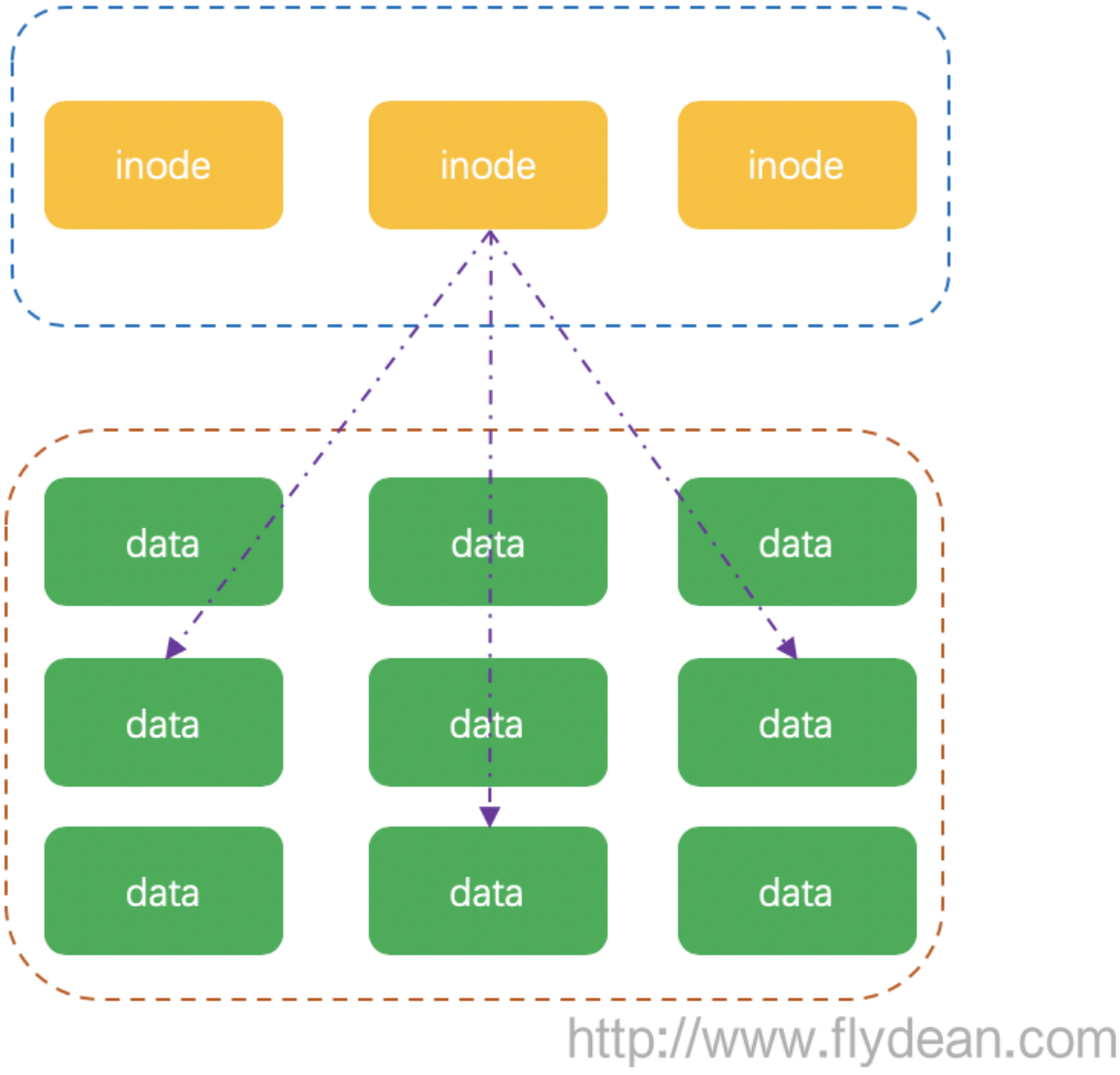
小师妹：F师兄，我记得你上次讲到Linux下面所有的资源都可以看做是文件，在linux下面文件和目录的本质是不是一样的?

对的，在linux下面文件是一等公民，所有的资源都是以文件的形式来区分的。

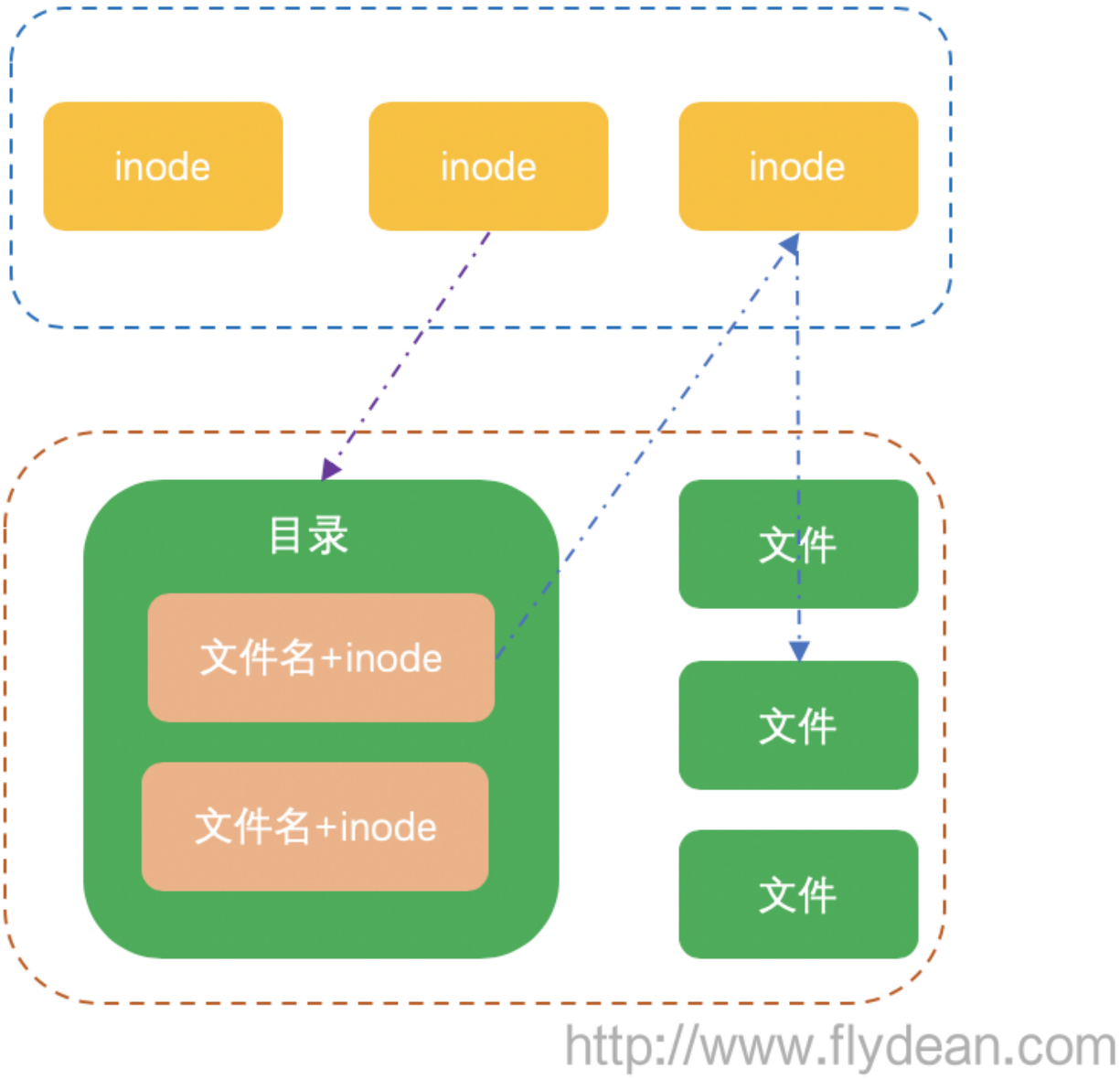
什么扇区，逻辑块，页之类的底层结构我们就不讲了。我们先考虑一下一个文件到底应该包含哪些内容。除了文件本身的数据之外，还有很多元数据的东西，比如文件权限，所有者，group，创建时间等信息。

在linux系统中，这两个部分是分开存储的。存放数据本身的叫做block，存放元数据的叫做inode。

inode中存储了block的地址，可以通过inode找到文件实际数据存储的block地址，从而进行文件访问。考虑一下大文件可能占用很多个block，所以一个inode中可以存储多个block的地址，而一个文件通常来说使用一个inode就够了。



为了显示层级关系和方便文件的管理，目录的数据文件中存放的是该目录下的文件和文件的inode地址，从而形成了一种一环套一环，圆环套圆环的链式关系。



上图列出了一个通过目录查找其下文件的环中环布局。

我想java中目录没有单独列出来一个类的原因可能是参考了linux底层的文件布局吧。

目录的基本操作

因为在java中目录和文件是公用File这个类的，所以File的基本操作目录它全都会。

基本上，目录和文件相比要多注意下面三类方法：

```
public boolean isDirectory()
public File[] listFiles()
public boolean mkdir()
```

为什么说是三类呢？因为还有几个和他们比较接近的方法，这里就不一一列举了。

isDirectory判断该文件是不是目录。listFiles列出该目录下面的所有文件。mkdir创建一个文件目录。

小师妹:F师兄，之前我们还以目录的遍历要耗费比较长的时间，经过你一讲解目录的数据结构，感觉listFiles并不是一个耗时操作呀，所有的数据都已经准备好了，直接读取出来就行。

对，看问题不要看表面，要看到隐藏在表面的本质内涵。你看师兄我平时不显山露水，其实是真正的中流砥柱，堪称公司优秀员工模范。

小师妹:F师兄，那平时也没看上头表彰你啥的？哦，我懂了，一定是老板怕表彰了你引起别人的嫉妒，会让你的好好大师兄的形象崩塌吧，看来老板真的懂你呀。

目录的进阶操作

好了小师妹，你懂了就行，下面F师兄给你讲一下目录的进阶操作，比如我们怎么拷贝一个目录呀?

小师妹，拷贝目录简单的F师兄，上次你就教我了：

```
cp -rf
```

一个命令的事情不就解决了吗？难道里面还隐藏了点秘密?

咳咳咳，秘密倒是没有，小师妹，我记得你上次说要对java从一而终的，今天师兄给你介绍一个在java中拷贝文件目录的方法。

其实Files工具类里已经为我们提供了一个拷贝文件的优秀方法：

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

使用这个方法，我们就可以进行文件的拷贝了。

如果想要拷贝目录，就遍历目录中的文件，循环调用这个copy方法就够了。

小师妹：且慢，F师兄，如果目录下面还有目录的，目录下还套目录的情况该怎么处理?

这就是圈套呀，看我用个递归的方法解决它：

```
public void useCopyFolder() throws IOException {
        File sourceFolder = new File("src/main/resources/flydean-source");
        File destinationFolder = new File("src/main/resources/flydean-dest");
        copyFolder(sourceFolder, destinationFolder);
    }

    private static void copyFolder(File sourceFolder, File destinationFolder) throws
IOException
    {
        //如果是dir则递归遍历创建dir，如果是文件则直接拷贝
        if (sourceFolder.isDirectory())
        {
            //查看目标dir是否存在
            if (!destinationFolder.exists())
            {
                destinationFolder.mkdir();
                log.info("目标dir已经创建：{}",destinationFolder);
```

```
            }
            for (String file : sourceFolder.list())
            {
                File srcFile = new File(sourceFolder, file);
                File destFile = new File(destinationFolder, file);
                copyFolder(srcFile, destFile);
            }
        }
        else
        {
            //使用Files.copy来拷贝具体的文件
            Files.copy(sourceFolder.toPath(), destinationFolder.toPath(),
StandardCopyOption.REPLACE_EXISTING);
            log.info("拷贝目标文件：{}",destinationFolder);
        }
    }
```

基本思想就是遇到目录我就遍历，遇到文件我就拷贝。

目录的腰疼操作

小师妹：F师兄，假如我想删除一个目录中的文件，或者我们想统计一下这个目录下面到底有多少个文件该怎么做呢?

虽然这些操作有点腰疼，还是可以解决的，Files工具类中有个方法叫做walk，返回一个Stream对象，我们可以使用Stream的API来对文件进行处理。

删除文件：

```
public void useFileWalkToDelete() throws IOException {
        Path dir = Paths.get("src/main/resources/flydean");
        Files.walk(dir)
                .sorted(Comparator.reverseOrder())
                .map(Path::toFile)
                .forEach(File::delete);
    }
```

统计文件：

```
 public void useFileWalkToSumSize() throws IOException {

        Path folder = Paths.get("src/test/resources");
        long size = Files.walk(folder)
                .filter(p -> p.toFile().isFile())
                .mapToLong(p -> p.toFile().length())
                .sum();
        log.info("dir size is: {}",size);
    }
```

总结

本文介绍了目录的一些非常常见和有用的操作。

第七章 文件系统和WatchService

简介

小师妹这次遇到了监控文件变化的问题，F师兄给小师妹介绍了JDK7 nio中引入的WatchService，没想到又顺道普及了一下文件系统的概念，万万没想到。

监控的痛点

小师妹：F师兄最近你有没有感觉到呼吸有点困难，后领有点凉飕飕的，说话有点不顺畅的那种?

没有啊小师妹，你是不是秋衣穿反了?

小师妹：不是的F师兄，我讲的是心里的感觉，那种莫须有的压力，还有一丝悸动缠绕在心。

别绕弯子了小师妹，是不是又遇到问题了。

> 更多内容请访问www.flydean.com

小师妹：还是F师兄懂我，这不上次的Properties文件用得非常上手，每次修改Properties文件都要重启java应用程序，真的是很痛苦。有没有什么其他的办法呢?

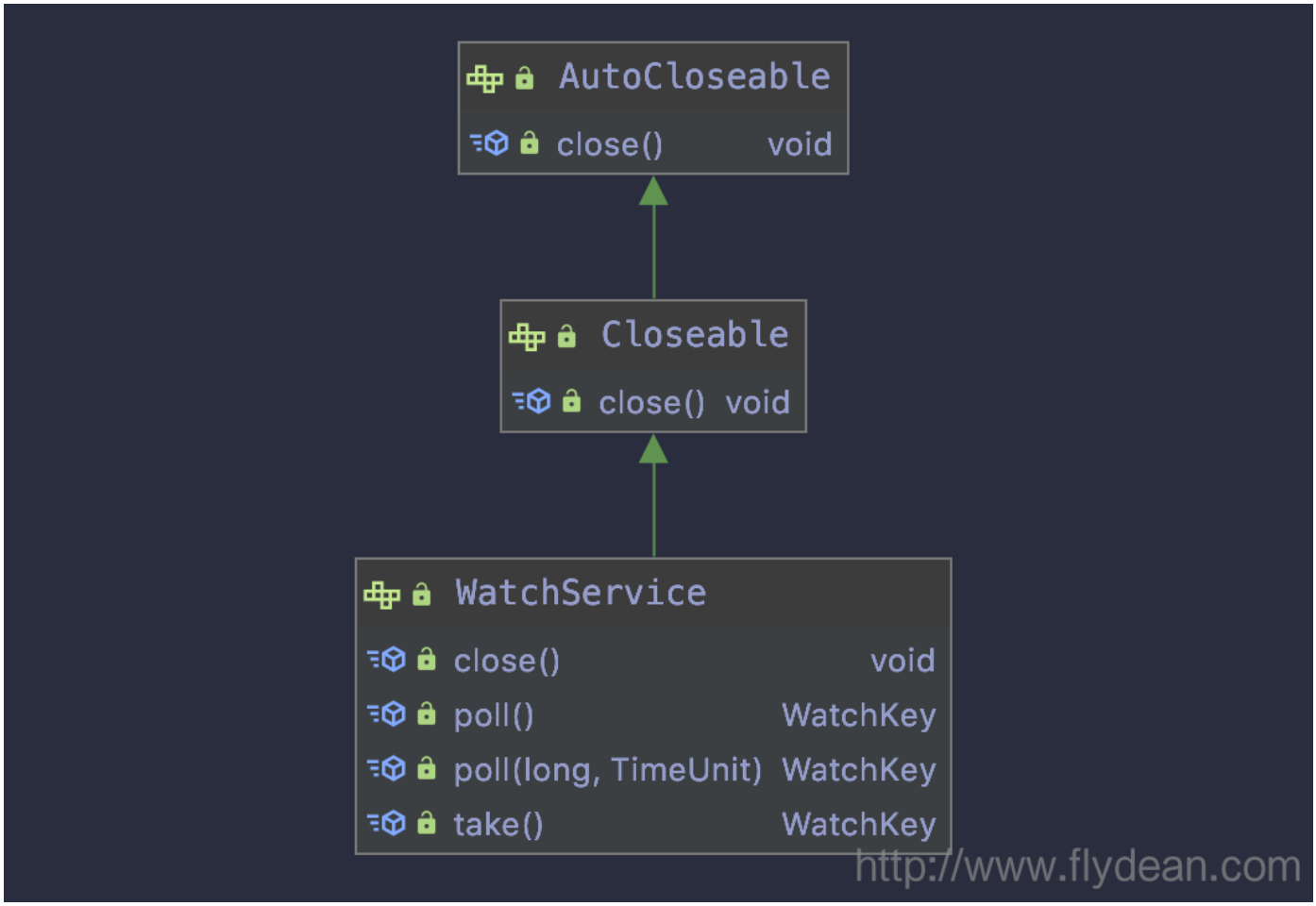
办法当然有，最基础的办法就是开一个线程定时去监控属性文件的最后修改时间，如果修改了就重新加载，这样不就行了。

小师妹：写线程啊，这么麻烦，有没有什么更简单的办法呢?

就知道你要这样问，还好我准备的比较充分，今天给你介绍一个JDK7在nio中引入的类WatchService。

WatchService和文件系统

WatchService是JDK7在nio中引入的接口：



监控的服务叫做WatchService，被监控的对象叫做Watchable：

```
WatchKey register(WatchService watcher,
                  WatchEvent.Kind<?>[] events,
                  WatchEvent.Modifier... modifiers)
        throws IOException;
WatchKey register(WatchService watcher, WatchEvent.Kind<?>... events)
        throws IOException;
```

Watchable通过register将该对象的WatchEvent注册到WatchService上。从此只要有WatchEvent发生在Watchable对象上，就会通知WatchService。

WatchEvent有四种类型：

1. ENTRY_CREATE 目标被创建
2. ENTRY_DELETE 目标被删除
3. ENTRY_MODIFY 目标被修改
4. OVERFLOW 一个特殊的Event，表示Event被放弃或者丢失

register返回的WatchKey就是监听到的WatchEvent的集合。

现在来看WatchService的4个方法：

1. close 关闭watchService
2. poll 获取下一个watchKey，如果没有则返回null
3. 带时间参数的poll 在等待的一定时间内获取下一个watchKey

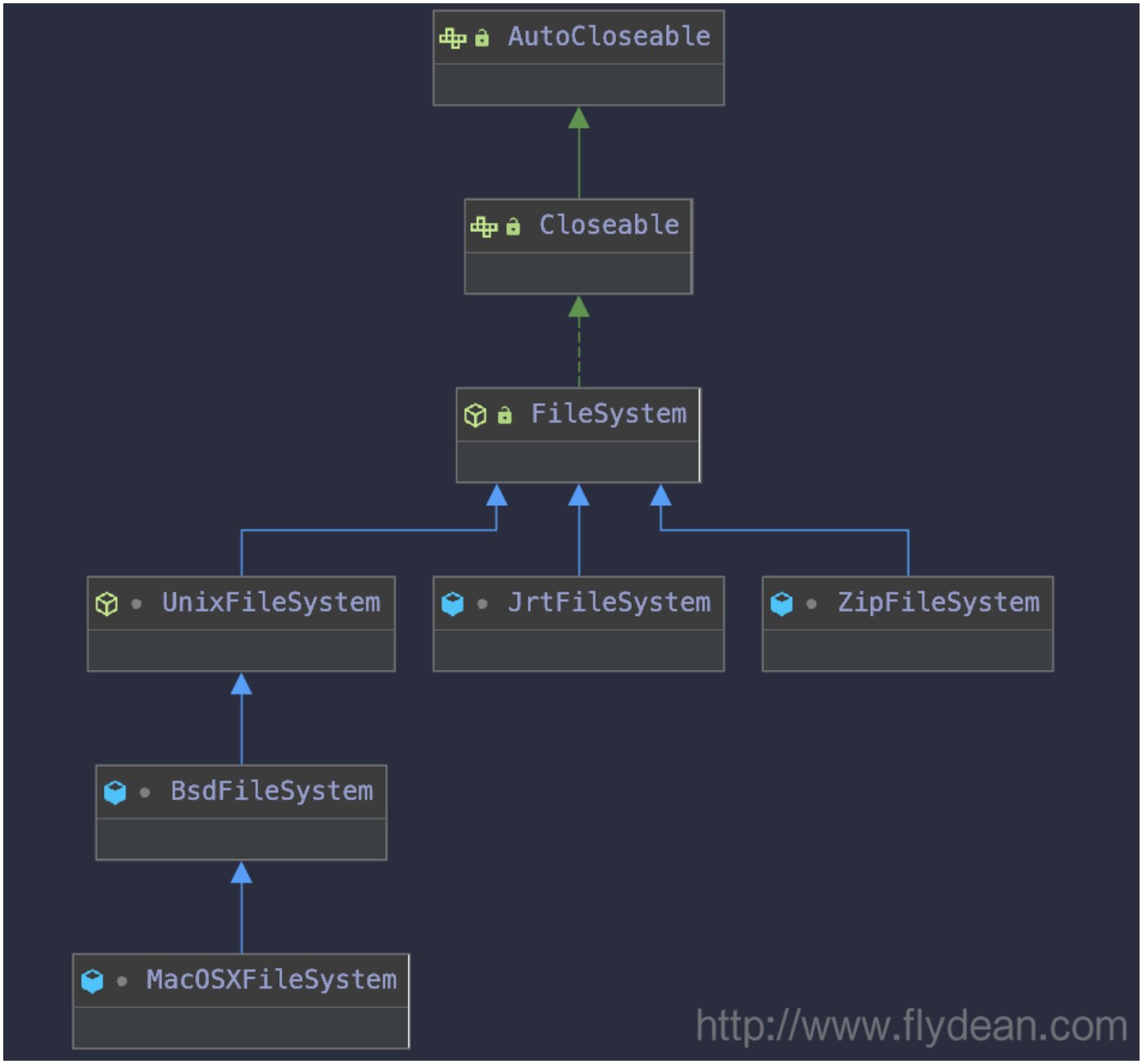
4. take 获取下一个watchKey，如果没有则一直等待

小师妹：F师兄，那怎么才能构建一个WatchService呢?

上次文章中说的文件系统，小师妹还记得吧，FileSystem中就有一个获取WatchService的方法：

```
public abstract WatchService newWatchService() throws IOException;
```

我们看下FileSystem的结构图：



在我的mac系统上，FileSystem可以分为三大类，UnixFileSystem，JrtFileSystem和ZipFileSystem。我猜在windows上面应该还有对应的windows相关的文件系统。小师妹你要是有兴趣可以去看一下。

小师妹：UnixFileSystem用来处理Unix下面的文件，ZipFileSystem用来处理zip文件。那JrtFileSystem是用来做什么的?

哎呀，这就又要扯远了，为什么每次问问题都要扯到天边....

从前当JDK还是9的时候，做了一个非常大的改动叫做模块化JPMS（Java Platform Module System），这个Jrt就是为了给模块化系统用的，我们来举个例子：

```
public void useJRTFileSystem(){
        String resource = "java/lang/Object.class";
        URL url = ClassLoader.getSystemResource(resource);
        log.info("{}",url);
    }
```

上面一段代码我们获取到了Object这个class的url，我们看下如果是在JDK8中，输出是什么：

```
jar:file:/Library/Java/JavaVirtualMachines/jdk1.8.0_171.jdk/Contents/Home/jre/lib/rt.ja
r!/java/lang/Object.class
```

输出结果是jar:file表示这个Object class是放在jar文件中的，后面是jar文件的路径。

如果是在JDK9之后：

```
jrt:/java.base/java/lang/Object.class
```

结果是jrt开头的，java.base是模块的名字，后面是Object的路径。看起来是不是比传统的jar路径更加简洁明了。

有了文件系统，我们就可以在获取系统默认的文件系统的同时，获取到相应的WatchService：

```
WatchService watchService = FileSystems.getDefault().newWatchService();
```

WatchSerice的使用和实现本质

小师妹：F师兄，WatchSerice是咋实现的呀？这么神奇，为我们省了这么多工作。

其实JDK提供了这么多类的目的就是为了不让我们重复造轮子，之前跟你讲监控文件的最简单办法就是开一个独立的线程来监控文件变化吗？其实.....WatchService就是这样做的！

```
PollingWatchService() {
        // TBD: Make the number of threads configurable
        scheduledExecutor = Executors
            .newSingleThreadScheduledExecutor(new ThreadFactory() {
                 @Override
                 public Thread newThread(Runnable r) {
                     Thread t = new Thread(null, r, "FileSystemWatcher", 0, false);
                     t.setDaemon(true);
                     return t;
                 }});
    }
```

上面的方法就是生成WatchService的方法，小师妹看到没有，它的本质就是开启了一个daemon的线程，用来接收监控任务。

下面看下怎么把一个文件注册到WatchService上面：

```
private void startWatcher(String dirPath, String file) throws IOException {
        WatchService watchService = FileSystems.getDefault().newWatchService();
        Path path = Paths.get(dirPath);
        path.register(watchService, ENTRY_MODIFY);

        Runtime.getRuntime().addShutdownHook(new Thread(() -> {
            try {
                watchService.close();
            } catch (IOException e) {
                log.error(e.getMessage());
            }
        }));

        WatchKey key = null;
        while (true) {
            try {
                key = watchService.take();
                for (WatchEvent<?> event : key.pollEvents()) {
                    if (event.context().toString().equals(fileName)) {
                        loadConfig(dirPath + file);
                    }
                }
                boolean reset = key.reset();
                if (!reset) {
                    log.info("该文件无法重置");
                    break;
                }
            } catch (Exception e) {
                log.error(e.getMessage());
            }
        }
    }
```

上面的关键方法就是path.register，其中Path是一个Watchable对象。

然后使用watchService.take来获取生成的WatchEvent，最后根据WatchEvent来处理文件。

总结

道生一，一生二，二生三，三生万物。一个简简单单的功能其实背后隐藏着...道德经，哦，不对，背后隐藏着道的哲学。

第八章 文件File和路径Path

简介

文件和路径有什么关系？文件和路径又隐藏了什么秘密？在文件系统的管理下，创建路径的方式又有哪些？今天F师兄带小师妹再给大家来一场精彩的表演。

文件和路径

小师妹：F师兄我有一个问题，java中的文件File是一个类可以理解，因为文件里面包含了很多其他的信息，但是路径Path为什么也要单独一个类出来？只用一个String表示不是更简单？

> 更多内容请访问www.flydean.com

万物皆有因，没有无缘无故的爱，也没有无缘无故的恨。一切真的是妙不可言啊。

我们来看下File和path的定义：

```
public class File
   implements Serializable, Comparable<File>
```

```
public interface Path
    extends Comparable<Path>, Iterable<Path>, Watchable
```

首先，File是一个类，它表示的是所有的文件系统都拥有的属性和功能，不管你是windows还是linux，他们中的File对象都应该是一样的。

File中包含了Path，小师妹你且看，Path是一个interface,为什么是一个interface呢？因为Path根据不同的情况可以分为JrtPath，UnixPath和ZipPath。三个Path所对应的FileSystem我们在上一篇文章中已经讨论过了。所以Path的实现是不同的，但是包含Path的File是相同的。

小师妹：F师兄，这个怎么这么拗口，给我来一个直白通俗的解释吧。

既然这样，且听我解释：爱国版的，或许我们属于不同的民族，但是我们都是中国人。通俗版的，大家都是文化人儿，为啥就你这么拽。文化版的，同九年，汝何秀？

再看两者的实现接口，File实现了Serializable表示可以被序列化，实现了Comparable，表示可以被排序。

Path继承Comparable，表示可以被排序。继承Iterable表示可以被遍历，可以被遍历是因为Path可以表示目录。继承Watchable，表示可以被注册到WatchService中，进行监控。

文件中的不同路径

小师妹：F师兄，File中有好几个关于Path的get方法，能讲一下他们的不同之处吗？

直接上代码：

```
public void getFilePath() throws IOException {
        File file= new File("../../www.flydean.com.txt");
        log.info("name is : {}",file.getName());

        log.info("path is : {}",file.getPath());
        log.info("absolutePath is : {}",file.getAbsolutePath());
        log.info("canonicalPath is : {}",file.getCanonicalPath());
    }
```

File中有三个跟Path有关的方法，分别是getPath，getAbsolutePath和getCanonicalPath。

getPath返回的结果就是new File的时候传入的路径，输入什么返回什么。

getAbsolutePath返回的是绝对路径，就是在getPath前面加上了当前的路径。

getCanonicalPath返回的是精简后的AbsolutePath，就是去掉了.或者..之类的指代符号。

看下输出结果：

```
 INFO com.flydean.FilePathUsage - name is : www.flydean.com.txt
 INFO com.flydean.FilePathUsage - path is : ../../www.flydean.com.txt
 INFO com.flydean.FilePathUsage - absolutePath is : /Users/flydean/learn-java-io-
nio/file-path/../../www.flydean.com.txt
 INFO com.flydean.FilePathUsage - canonicalPath is : /Users/flydean/www.flydean.com.txt
```

构建不同的Path

小师妹：F师兄，我记得路径有相对路径，绝对路径等，是不是也有相应的创建Path的方法呢?

当然有的，先看下绝对路径的创建：

```
public void getAbsolutePath(){
        Path absolutePath = Paths.get("/data/flydean/learn-java-io-nio/file-path",
"src/resource","www.flydean.com.txt");
        log.info("absolutePath {}",absolutePath );
    }
```

我们可以使用Paths.get方法传入绝对路径的地址来构建绝对路径。

同样使用Paths.get方法，传入非绝对路径可以构建相对路径。

```
public void getRelativePath(){
        Path RelativePath = Paths.get("src", "resource","www.flydean.com.txt");
        log.info("absolutePath {}",RelativePath.toAbsolutePath() );
    }
```

我们还可以从URI中构建Path：

```
public void getPathfromURI(){
        URI uri = URI.create("file:///data/flydean/learn-java-io-nio/file-
path/src/resource/www.flydean.com.txt");
        log.info("schema {}",uri.getScheme());
        log.info("default provider absolutePath
{}",FileSystems.getDefault().provider().getPath(uri).toAbsolutePath().toString());
    }
```

也可以从FileSystem构建Path：

```
public void getPathWithFileSystem(){
            Path path1 =
FileSystems.getDefault().getPath(System.getProperty("user.home"), "flydean",
"flydean.txt");
           log.info(path1.toAbsolutePath().toString());

            Path path2 = FileSystems.getDefault().getPath("/Users", "flydean",
"flydean.txt");
            log.info(path2.toAbsolutePath().toString());

        }
```

总结

好多好多Path的创建方法，总有一款适合你。快来挑选吧。

第九章 Buffer和Buff

简介

小师妹在学习NIO的路上越走越远，唯一能够帮到她的就是在她需要的时候给她以全力的支持。什么都不说了，今天介绍的是NIO的基础Buffer。老铁给我上个Buff。

Buffer是什么

小师妹：F师兄，这个Buffer是我们纵横王者峡谷中那句：老铁给我加个Buff的意思吗?

当然不是了，此Buffer非彼Buff，Buffer是NIO的基础，没有Buffer就没有NIO，没有Buffer就没有今天的java。

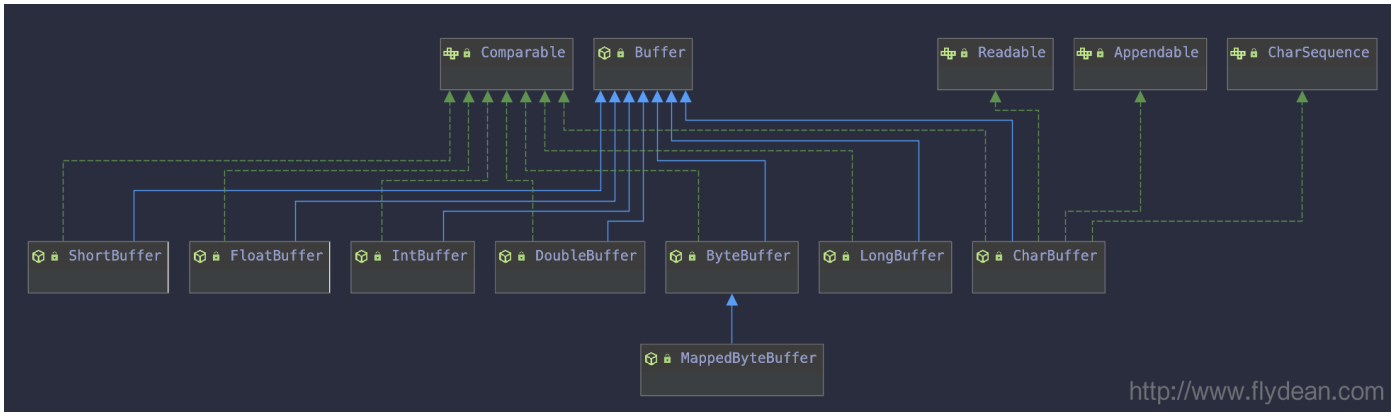
因为NIO是按Block来读取数据的，这个一个Block就可以看做是一个Buffer。我们在Buffer中存储要读取的数据和要写入的数据，通过Buffer来提高读取和写入的效率。

> 更多内容请访问www.flydean.com

还记得java对象的底层存储单位是什么吗?

小师妹：这个我知道，java对象的底层存储单位是字节Byte。

对，我们看下Buffer的继承图：



Buffer是一个接口，它下面有诸多实现，包括最基本的ByteBuffer和其他的基本类型封装的其他Buffer。

小师妹：F师兄，有ByteBuffer不就够了吗？还要其他的类型Buffer做什么？

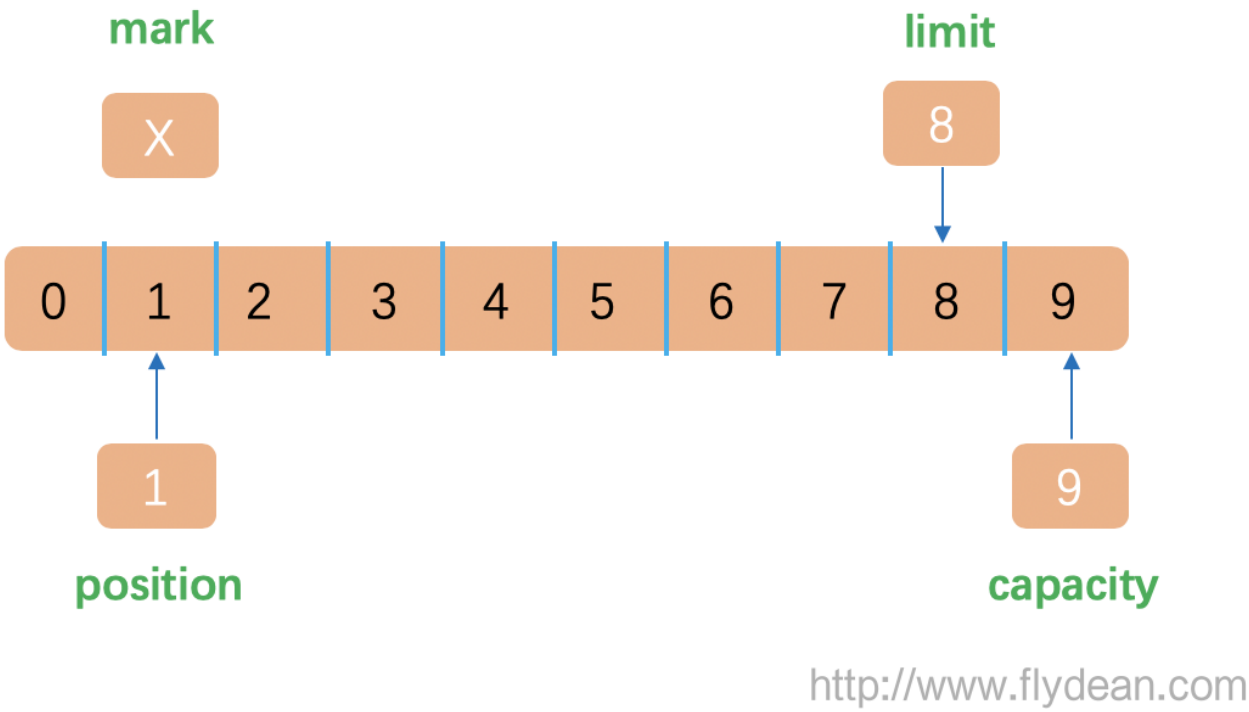
小师妹，山珍再好，也有吃腻的时候，偶尔也要换个萝卜白菜啥的，你以为乾隆下江南都干了些啥？

ByteBuffer虽然好用，但是它毕竟是最小的单位，在它之上我们还有Char，int，Double，Short等等基础类型，为了简单起见，我们也给他们都搞一套Buffer。

Buffer进阶

小师妹：F师兄，既然Buffer是这些基础类型的集合，为什么不直接用结合来表示呢？给他们封装成一个对象，好像有点多余。

我们既然在面向对象的世界，从表面来看自然是使用Object比较合乎情理，从底层的本质上看，这些封装的Buffer包含了一些额外的元数据信息，并且还提供了一些意想不到的功能。



上图列出了Buffer中的几个关键的概念，分别是Capacity，Limit，Position和Mark。Buffer底层的本质是数组，我们以ByteBuffer为例，它的底层是：

```
final byte[] hb;
```

- Capacity表示的是该Buffer能够承载元素的最大数目，这个是在Buffer创建初期就设置的，不可以被改变。
- Limit表示的Buffer中可以被访问的元素个数，也就是说Buffer中存活的元素个数。
- Position表示的是下一个可以被访问元素的index，可以通过put和get方法进行自动更新。
- Mark表示的是历史index，当我们调用mark方法的时候，会把设置Mark为当前的position，通过调用reset方法把Mark的值恢复到position中。

创建Buffer

小师妹：F师兄呀，这么多Buffer创建起来是不是很麻烦？有没有什么快捷的使用办法？

一般来说创建Buffer有两种方法，一种叫做allocate，一种叫做wrap。

```
public void createBuffer(){
        IntBuffer intBuffer= IntBuffer.allocate(10);
        log.info("{}",intBuffer);
        log.info("{}",intBuffer.hasArray());
        int[] intArray=new int[10];
        IntBuffer intBuffer2= IntBuffer.wrap(intArray);
        log.info("{}",intBuffer2);
        IntBuffer intBuffer3= IntBuffer.wrap(intArray,2,5);
        log.info("{}",intBuffer3);
        intBuffer3.clear();
        log.info("{}",intBuffer3);
        log.info("{}",intBuffer3.hasArray());
    }
```

allocate可以为Buffer分配一个空间，wrap同样为Buffer分配一个空间，不同的是这个空间背后的数组是自定义的，wrap还支持三个参数的方法，后面两个参数分别是offset和length。

```
INFO com.flydean.BufferUsage - java.nio.HeapIntBuffer[pos=0 lim=10 cap=10]
INFO com.flydean.BufferUsage - true
INFO com.flydean.BufferUsage - java.nio.HeapIntBuffer[pos=0 lim=10 cap=10]
INFO com.flydean.BufferUsage - java.nio.HeapIntBuffer[pos=2 lim=7 cap=10]
INFO com.flydean.BufferUsage - java.nio.HeapIntBuffer[pos=0 lim=10 cap=10]
INFO com.flydean.BufferUsage - true
```

hasArray用来判断该Buffer的底层是不是数组实现的，可以看到，不管是wrap还是allocate，其底层都是数组。

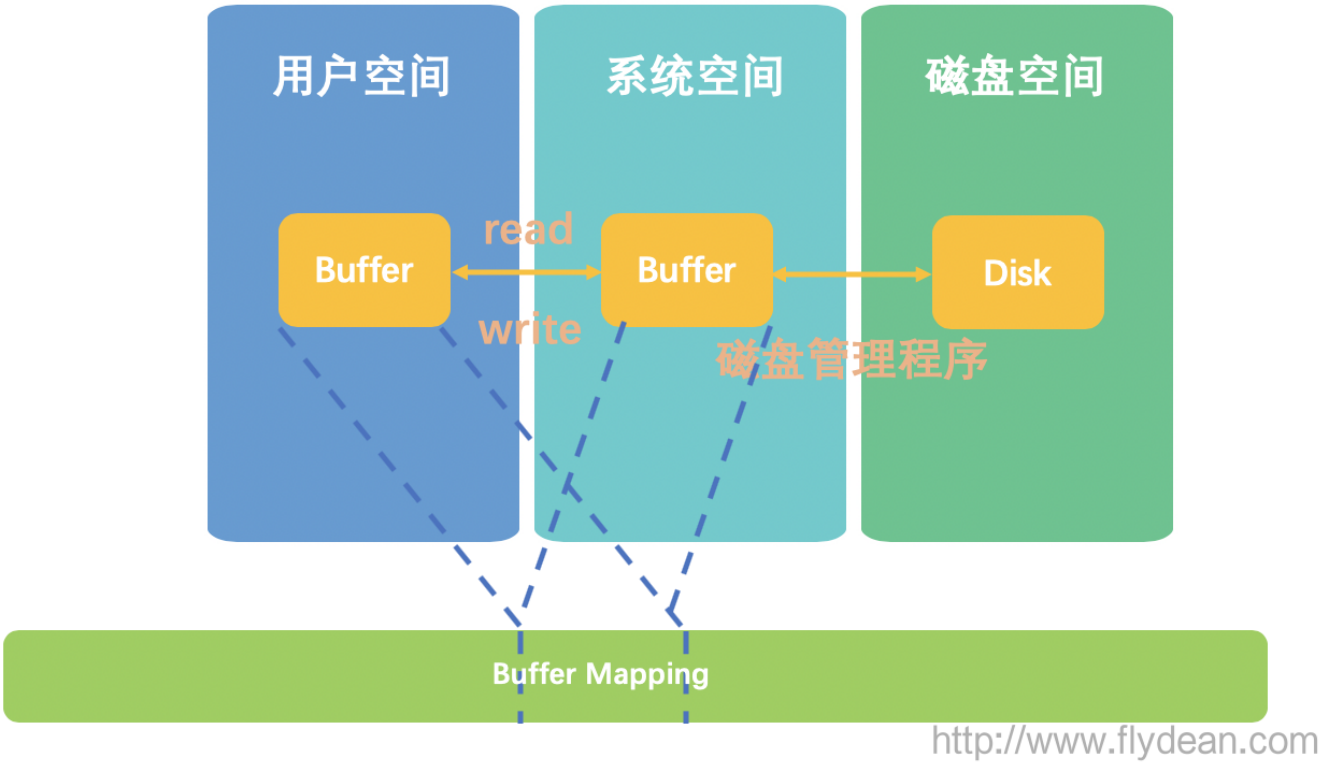
> 需要注意的一点，最后，我们调用了clear方法，clear方法调用之后，我们发现Buffer的position和limit都被重置了。这说明wrap的三个参数方法设定的只是初始值，可以被重置。

Direct VS non-Direct

小师妹：F师兄，你说了两种创建Buffer的方法，但是两种Buffer的后台都是数组，难道还有非数组的Buffer吗?

自然是有的,但是只有ByteBuffer有。ByteBuffer有一个allocateDirect方法，可以分配Direct Buffer。

小师妹：Direct和非Direct有什么区别呢?



Direct Buffer就是说，不需要在用户空间再复制拷贝一份数据，直接在虚拟地址映射空间中进行操作。这叫Direct。这样做的好处就是快。缺点就是在分配和销毁的时候会占用更多的资源，并且因为Direct Buffer不在用户空间之内，所以也不受垃圾回收机制的管辖。

所以通常来说只有在数据量比较大，生命周期比较长的数据来使用Direct Buffer。

看下代码：

```
public void createByteBuffer() throws IOException {
        ByteBuffer byteBuffer= ByteBuffer.allocateDirect(10);
        log.info("{}",byteBuffer);
        log.info("{}",byteBuffer.hasArray());
        log.info("{}",byteBuffer.isDirect());

        try (RandomAccessFile aFile = new
RandomAccessFile("src/main/resources/www.flydean.com", "r");
             FileChannel inChannel = aFile.getChannel()) {
            MappedByteBuffer buffer = inChannel.map(FileChannel.MapMode.READ_ONLY, 0,
inChannel.size());
            log.info("{}",buffer);
            log.info("{}",buffer.hasArray());
            log.info("{}",buffer.isDirect());
        }
    }
```

除了allocateDirect,使用FileChannel的map方法也可以得到一个Direct的MappedByteBuffer。

上面的例子输出结果：

```
INFO com.flydean.BufferUsage - java.nio.DirectByteBuffer[pos=0 lim=10 cap=10]
INFO com.flydean.BufferUsage - false
INFO com.flydean.BufferUsage - true
INFO com.flydean.BufferUsage - java.nio.DirectByteBufferR[pos=0 lim=0 cap=0]
INFO com.flydean.BufferUsage - false
INFO com.flydean.BufferUsage - true
```

Buffer的日常操作

小师妹:F师兄，看起来Buffer确实有那么一点复杂，那么Buffer都有哪些操作呢?

Buffer的操作有很多，下面我们一一来讲解。

向Buffer写数据

向Buffer写数据可以调用Buffer的put方法:

```
public void putBuffer(){
        IntBuffer intBuffer= IntBuffer.allocate(10);
        intBuffer.put(1).put(2).put(3);
        log.info("{}",intBuffer.array());
        intBuffer.put(0,4);
        log.info("{}",intBuffer.array());
    }
```

因为put方法返回的还是一个IntBuffer类，所以Buffer的put方法可以像Stream那样连写。

同时，我们还可以指定put在什么位置。上面的代码输出:

```
INFO com.flydean.BufferUsage - [1, 2, 3, 0, 0, 0, 0, 0, 0, 0]
INFO com.flydean.BufferUsage - [4, 2, 3, 0, 0, 0, 0, 0, 0, 0]
```

从Buffer读数据

读数据使用get方法，但是在get方法之前我们需要调用flip方法。

flip方法是做什么用的呢？上面讲到Buffer有个position和limit字段，position会随着get或者put的方法自动指向后面一个元素，而limit表示的是该Buffer中有多少可用元素。

如果我们要读取Buffer的值则会从positon开始到limit结束:

```
public void getBuffer(){
        IntBuffer intBuffer= IntBuffer.allocate(10);
        intBuffer.put(1).put(2).put(3);
        intBuffer.flip();
        while (intBuffer.hasRemaining()) {
            log.info("{}",intBuffer.get());
        }
        intBuffer.clear();
    }
```

可以通过hasRemaining来判断是否还有下一个元素。通过调用clear来清除Buffer，以供下次使用。

rewind Buffer

rewind和flip很类似，不同之处在于rewind不会改变limit的值，只会将position重置为0。

```
public void rewindBuffer(){
        IntBuffer intBuffer= IntBuffer.allocate(10);
        intBuffer.put(1).put(2).put(3);
        log.info("{}",intBuffer);
        intBuffer.rewind();
        log.info("{}",intBuffer);
    }
```

上面的结果输出：

```
INFO com.flydean.BufferUsage - java.nio.HeapIntBuffer[pos=3 lim=10 cap=10]
INFO com.flydean.BufferUsage - java.nio.HeapIntBuffer[pos=0 lim=10 cap=10]
```

Compact Buffer

Buffer还有一个compact方法，顾名思义compact就是压缩的意思，就是把Buffer从当前position到limit的值赋值到position为0的位置：

```
public void useCompact(){
        IntBuffer intBuffer= IntBuffer.allocate(10);
        intBuffer.put(1).put(2).put(3);
        intBuffer.flip();
        log.info("{}",intBuffer);
        intBuffer.get();
        intBuffer.compact();
        log.info("{}",intBuffer);
        log.info("{}",intBuffer.array());
    }
```

上面代码输出：

```
INFO com.flydean.BufferUsage - java.nio.HeapIntBuffer[pos=0 lim=3 cap=10]
INFO com.flydean.BufferUsage - java.nio.HeapIntBuffer[pos=2 lim=10 cap=10]
INFO com.flydean.BufferUsage - [2, 3, 3, 0, 0, 0, 0, 0, 0, 0]
```

duplicate Buffer

最后我们讲一下复制Buffer，有三种方法，duplicate，asReadOnlyBuffer，和slice。

duplicate就是拷贝原Buffer的position，limit和mark，它和原Buffer是共享原始数据的。所以修改了duplicate之后的Buffer也会同时修改原Buffer。

如果用asReadOnlyBuffer就不允许拷贝之后的Buffer进行修改。

slice也是readOnly的，不过它拷贝的是从原Buffer的position到limit-position之间的部分。

```
public void duplicateBuffer(){
        IntBuffer intBuffer= IntBuffer.allocate(10);
        intBuffer.put(1).put(2).put(3);
        log.info("{}",intBuffer);
        IntBuffer duplicateBuffer=intBuffer.duplicate();
        log.info("{}",duplicateBuffer);
        IntBuffer readOnlyBuffer=intBuffer.asReadOnlyBuffer();
        log.info("{}",readOnlyBuffer);
        IntBuffer sliceBuffer=intBuffer.slice();
        log.info("{}",sliceBuffer);
    }
```

输出结果：

```
INFO com.flydean.BufferUsage - java.nio.HeapIntBuffer[pos=3 lim=10 cap=10]
INFO com.flydean.BufferUsage - java.nio.HeapIntBuffer[pos=3 lim=10 cap=10]
INFO com.flydean.BufferUsage - java.nio.HeapIntBufferR[pos=3 lim=10 cap=10]
INFO com.flydean.BufferUsage - java.nio.HeapIntBuffer[pos=0 lim=7 cap=7]
```

总结

今天给小师妹介绍了Buffer的原理和基本操作。

第十章 File copy和File filter

简介

一个linux命令的事情，小师妹非要让我教她怎么用java来实现，哎，摊上个这么杠精的小师妹，我也是深感无力，做一个师兄真的好难。

使用java拷贝文件

今天小师妹找到我了：F师兄，能告诉怎么拷贝文件吗？

拷贝文件？不是很简单的事情吗？如果你有了文件的读权限，只需要这样就可以了。

```
cp www.flydean.com www.flydean.com.back
```

当然，如果是目录的话还可以加两个参数遍历和强制拷贝：

```
cp -rf srcDir distDir
```

这么简单的linux命令，不要告诉我你不会。

小师妹笑了：F师兄，我不要用linux命令，我就想用java来实现，我不正在学java吗？学一门当然要找准机会来练习啦，快快教教我吧。

既然这样，那我就开讲了。java中文件的拷贝其实也有三种方法，可以使用传统的文件读写的方法，也可以使用最新的NIO中提供的拷贝方法。

使用传统方法当然没有NIO快，也没有NIO简洁，我们先来看看怎么使用传统的文件读写的方法来拷贝文件：

```
    public  void copyWithFileStreams() throws IOException
    {
        File fileToCopy = new File("src/main/resources/www.flydean.com");
        File newFile = new File("src/main/resources/www.flydean.com.back");
        newFile.createNewFile();
        try(FileOutputStream output = new FileOutputStream(newFile);FileInputStream
input = new FileInputStream(fileToCopy)){
            byte[] buf = new byte[1024];
            int bytesRead;
            while ((bytesRead = input.read(buf)) > 0)
            {
                output.write(buf, 0, bytesRead);
            }
        }
    }
```

上面的例子中，我们首先定义了两个文件，然后从两个文件中生成了OutputStream和InputStream，最后以字节流的形式从input中读出数据到outputStream中，最终完成了文件的拷贝。

传统的File IO拷贝比较繁琐，速度也比较慢。我们接下来看看怎么使用NIO来完成这个过程：

```
public  void copyWithNIOChannel() throws IOException
    {
        File fileToCopy = new File("src/main/resources/www.flydean.com");
        File newFile = new File("src/main/resources/www.flydean.com.back");

        try(FileInputStream inputStream = new
FileInputStream(fileToCopy);FileOutputStream outputStream = new
FileOutputStream(newFile)){
            FileChannel inChannel = inputStream.getChannel();
            FileChannel outChannel = outputStream.getChannel();
            inChannel.transferTo(0, fileToCopy.length(), outChannel);
        }
    }
```

之前我们讲到NIO中一个非常重要的概念就是channel,通过构建源文件和目标文件的channel通道，可以直接在channel层面进行拷贝，如上面的例子所示，我们调用了inChannel.transferTo完成了拷贝。

最后，还有一个更简单的NIO文件拷贝的方法：

```
public  void copyWithNIOFiles() throws IOException
    {
        Path source = Paths.get("src/main/resources/www.flydean.com");
        Path destination = Paths.get("src/main/resources/www.flydean.com.back");
        Files.copy(source, destination, StandardCopyOption.REPLACE_EXISTING);
    }
```

直接使用工具类Files提供的copy方法即可。

使用File filter

太棒了，小师妹一脸崇拜：F师兄，我还有一个需求，就是想删除某个目录里面的以.log结尾的日志文件，这个需求是不是很常见？F师兄一般是怎么操作的?

一般这种操作我都是一个linux命令就搞定了，如果搞不定那就用两个：

```
rm -rf *.log
```

当然，如果需要，我们也是可以用java来实现的。

java中提供了两个Filter都可以用来实现这个功能。

这两个Filter是java.io.FilenameFilter和java.io.FileFilter:

```
@FunctionalInterface
public interface FilenameFilter {
    boolean accept(File dir, String name);
}
```

```
@FunctionalInterface
public interface FileFilter {
    boolean accept(File pathname);
}
```

这两个接口都是函数式接口，所以他们的实现可以直接用lambda表达式来代替。

两者的区别在于，FilenameFilter进行过滤的是文件名和文件所在的目录。而FileFilter进行过滤的直接就是目标文件。

在java中是没有目录的概念的，一个目录也是用File的表示的。

上面的两个使用起来非常类似，我们就以FilenameFilter为例，看下怎么删除.log文件：

```
public void useFileNameFilter()
    {
        String targetDirectory = "src/main/resources/";
        File directory = new File(targetDirectory);

        //Filter out all log files
        String[] logFiles = directory.list( (dir, fileName)->
fileName.endsWith(".log"));

        //If no log file found; no need to go further
        if (logFiles.length == 0)
            return;

        //This code will delete all log files one by one
        for (String logfile : logFiles)
        {
            String tempLogFile = targetDirectory + File.separator + logfile;
            File fileDelete = new File(tempLogFile);
            boolean isdeleted = fileDelete.delete();
            log.info("file : {} is deleted : {} ", tempLogFile , isdeleted);
        }
    }
```

上面的例子中，我们通过directory.list方法，传入lambda表达式创建的Filter，实现了过滤的效果。

最后，我们将过滤之后的文件删除。实现了目标。

总结

小师妹的两个问题解决了，希望今天可以不要再见到她。

第十一章 NIO中Channel的妙用

简介

小师妹，你还记得我们使用IO和NIO的初心吗？

小师妹：F师兄，使用IO和NIO不就是为了让生活更美好，世界充满爱吗？让我等程序员可以优雅的将数据从一个地方搬运到另外一个地方。利其器，善其事，才有更多的时间去享受生活呀。

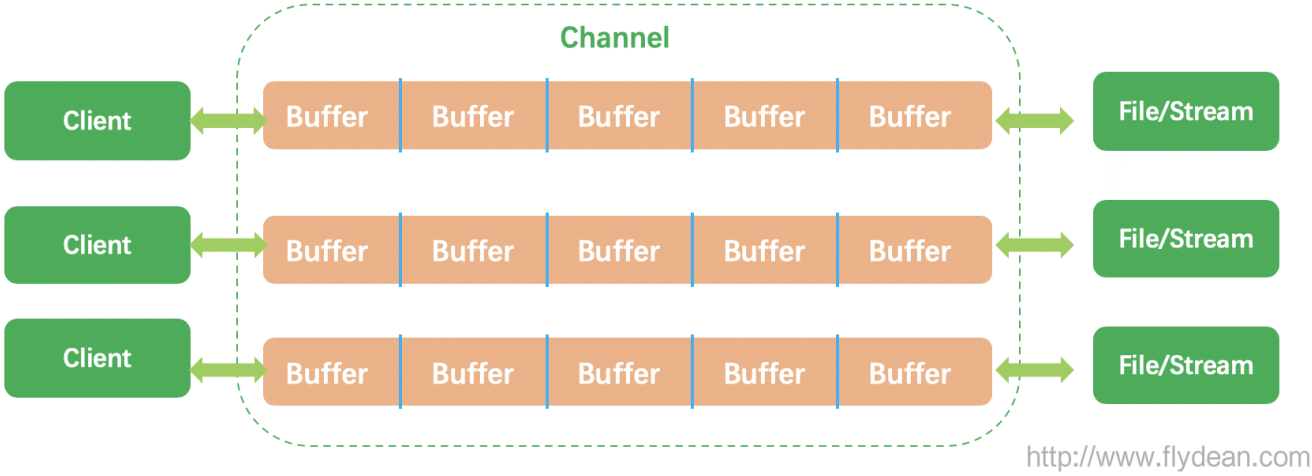
善，如果将数据比做人，IO，NIO的目的就是把人运到美国。

小师妹：F师兄，为什么要运到美国呀，美国现在新冠太严重了，还是待在中国吧。中国是世界上最安全的国家！

好吧，为了保险起见，我们要把人运到上海。人就是数据，怎么运过去呢？可以坐飞机，坐汽车，坐火车，这些什么飞机，汽车，火车就可以看做是一个一个的Buffer。

最后飞机的航线，汽车的公路和火车的轨道就可以看做是一个个的channel。

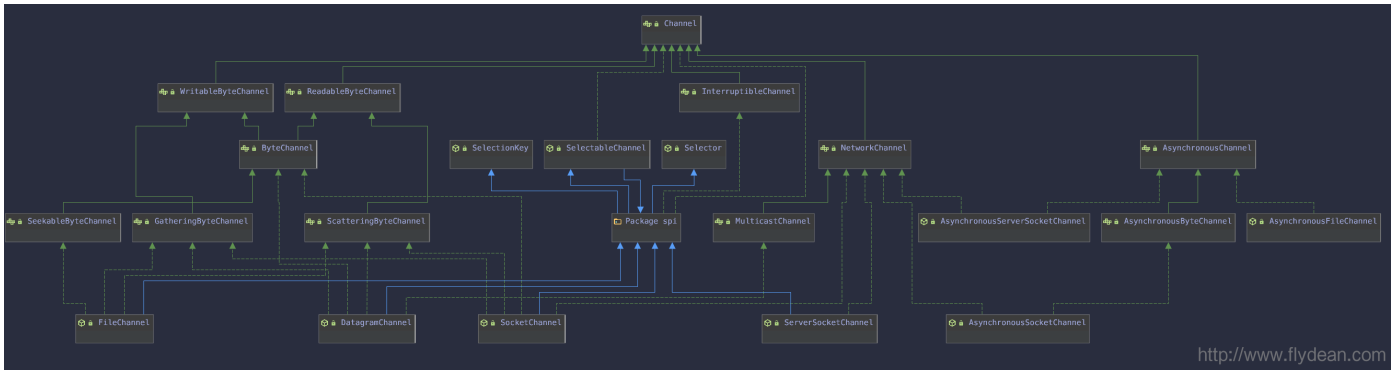
简单点讲，channel就是负责运送Buffer的通道。



IO按源头来分，可以分为两种，从文件来的File IO，从Stream来的Stream IO。不管哪种IO，都可以通过channel来运送数据。

Channel的分类

虽然数据的来源只有两种，但是JDK中Channel的分类可不少，如下图所示：



先来看看最基本的，也是最顶层的接口Channel：

```
public interface Channel extends Closeable {
    public boolean isOpen();
    public void close() throws IOException;

}
```

最顶层的Channel很简单，继承了Closeable接口，需要实现两个方法isOpen和close。

一个用来判断channel是否打开，一个用来关闭channel。

小师妹：F师兄，顶层的Channel怎么这么简单，完全不符合Channel很复杂的人设啊。

别急，JDK这么做其实也是有道理的，因为是顶层的接口，必须要更加抽象更加通用，结果，一通用就发现还真的就只有这么两个方法是通用的。

所以为了应对这个问题，Channel中定义了很多种不同的类型。

最最底层的Channel有5大类型，分别是：

FileChannel

这5大channel中，和文件File有关的就是这个FileChannel了。

FileChannel可以从RandomAccessFile, FileInputStream或者FileOutputStream中通过调用getChannel()来得到。

也可以直接调用FileChannel中的open方法传入Path创建。

```
public abstract class FileChannel
    extends AbstractInterruptibleChannel
    implements SeekableByteChannel, GatheringByteChannel, ScatteringByteChannel
```

我们看下FileChannel继承或者实现的接口和类。

AbstractInterruptibleChannel实现了InterruptibleChannel接口，interrupt大家都知道吧，用来中断线程执行的利器。来看一下下面一段非常玄妙的代码：

```
protected final void begin() {
        if (interruptor == null) {
            interruptor = new Interruptible() {
                    public void interrupt(Thread target) {
                        synchronized (closeLock) {
                            if (closed)
                                return;
                            closed = true;
                            interrupted = target;
                            try {
                                AbstractInterruptibleChannel.this.implCloseChannel();
                            } catch (IOException x) { }
                        }
                    }};
        }
```

```
        blockedOn(interruptor);
        Thread me = Thread.currentThread();
        if (me.isInterrupted())
            interruptor.interrupt(me);
    }
```

上面这段代码就是AbstractInterruptibleChannel的核心所在。

首先定义了一个Interruptible的实例，这个实例中有一个interrupt方法，用来关闭Channel。

然后获得当前线程的实例，判断当前线程是否Interrupted，如果是的话，就调用Interruptible的interrupt方法将当前channel关闭。

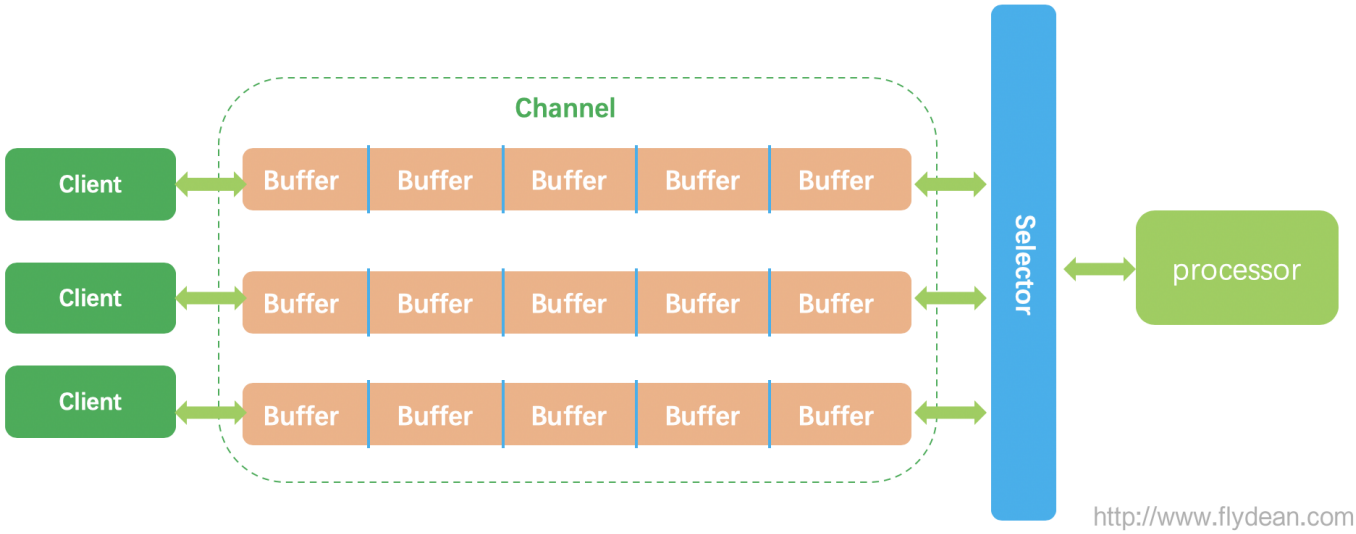
SeekableByteChannel用来连接Entry或者File。它有一个独特的属性叫做position，表示当前读取的位置。可以被修改。

GatheringByteChannel和ScatteringByteChannel表示可以一次读写一个Buffer序列结合（Buffer Array）：

```
public long write(ByteBuffer[] srcs, int offset, int length)
        throws IOException;
public long read(ByteBuffer[] dsts, int offset, int length)
        throws IOException;
```

Selector和Channel

在讲其他几个Channel之前，我们看一个和下面几个channel相关的Selector：



这里要介绍一个新的Channel类型叫做SelectableChannel，之前的FileChannel的连接是一对一的，也就是说一个channel要对应一个处理的线程。而SelectableChannel则是一对多的，也就是说一个处理线程可以通过Selector来对应处理多个channel。

SelectableChannel通过注册不同的SelectionKey，实现对多个Channel的监听。后面我们会具体的讲解Selector的使用，敬请期待。

DatagramChannel

DatagramChannel是用来处理UDP的Channel。它自带了Open方法来创建实例。

来看看DatagramChannel的定义：

```
public abstract class DatagramChannel
    extends AbstractSelectableChannel
    implements ByteChannel, ScatteringByteChannel, GatheringByteChannel,
MulticastChannel
```

ByteChannel表示它同时是ReadableByteChannel也是WritableByteChannel，可以同时写入和读取。

MulticastChannel代表的是一种多播协议。正好和UDP对应。

SocketChannel

SocketChannel是用来处理TCP的channel。它也是通过Open方法来创建的。

```
public abstract class SocketChannel
    extends AbstractSelectableChannel
    implements ByteChannel, ScatteringByteChannel, GatheringByteChannel, NetworkChannel
```

SocketChannel跟DatagramChannel的唯一不同之处就是实现的是NetworkChannel借口。

NetworkChannel提供了一些network socket的操作，比如绑定地址等。

ServerSocketChannel

ServerSocketChannel也是一个NetworkChannel，它主要用在服务器端的监听。

```
public abstract class ServerSocketChannel
    extends AbstractSelectableChannel
    implements NetworkChannel
```

AsynchronousSocketChannel

最后AsynchronousSocketChannel是一种异步的Channel：

```
public abstract class AsynchronousSocketChannel
    implements AsynchronousByteChannel, NetworkChannel
```

为什么是异步呢？我们看一个方法：

```
public abstract Future<Integer> read(ByteBuffer dst);
```

可以看到返回值是一个Future，所以read方法可以立刻返回，只在我们需要的时候从Future中取值即可。

使用Channel

小师妹：F师兄，讲了这么多种类的Channel，看得我眼花缭乱，能不能讲一个Channel的具体例子呢?

好的小师妹，我们现在讲一个使用Channel进行文件拷贝的例子，虽然Channel提供了transferTo的方法可以非常简单的进行拷贝，但是为了能够看清楚Channel的通用使用，我们选择一个更加常规的例子：

```
public void useChannelCopy() throws IOException {
        FileInputStream input = new FileInputStream
("src/main/resources/www.flydean.com");
        FileOutputStream output = new FileOutputStream
("src/main/resources/www.flydean.com.txt");
        try(ReadableByteChannel source = input.getChannel(); WritableByteChannel dest =
output.getChannel()){
            ByteBuffer buffer = ByteBuffer.allocateDirect(1024);
            while (source.read(buffer) != -1)
            {
                // flip buffer,准备写入
                buffer.flip();
                // 查看是否有更多的内容
                while (buffer.hasRemaining())
                {
                    dest.write(buffer);
                }
                // clear buffer，供下一次使用
                buffer.clear();
            }
        }
    }
```

上面的例子中我们从InputStream中读取Buffer，然后写入到FileOutputStream。

总结

今天讲解了Channel的具体分类，和一个简单的例子，后面我们会再体验一下Channel的其他例子，敬请期待。

第十二章 MappedByteBuffer多大的文件我都装得下

简介

大大大，我要大！小师妹要读取的文件越来越大，该怎么帮帮她，让程序在性能和速度上面得到平衡呢？快来跟F师兄一起看看吧。

虚拟地址空间

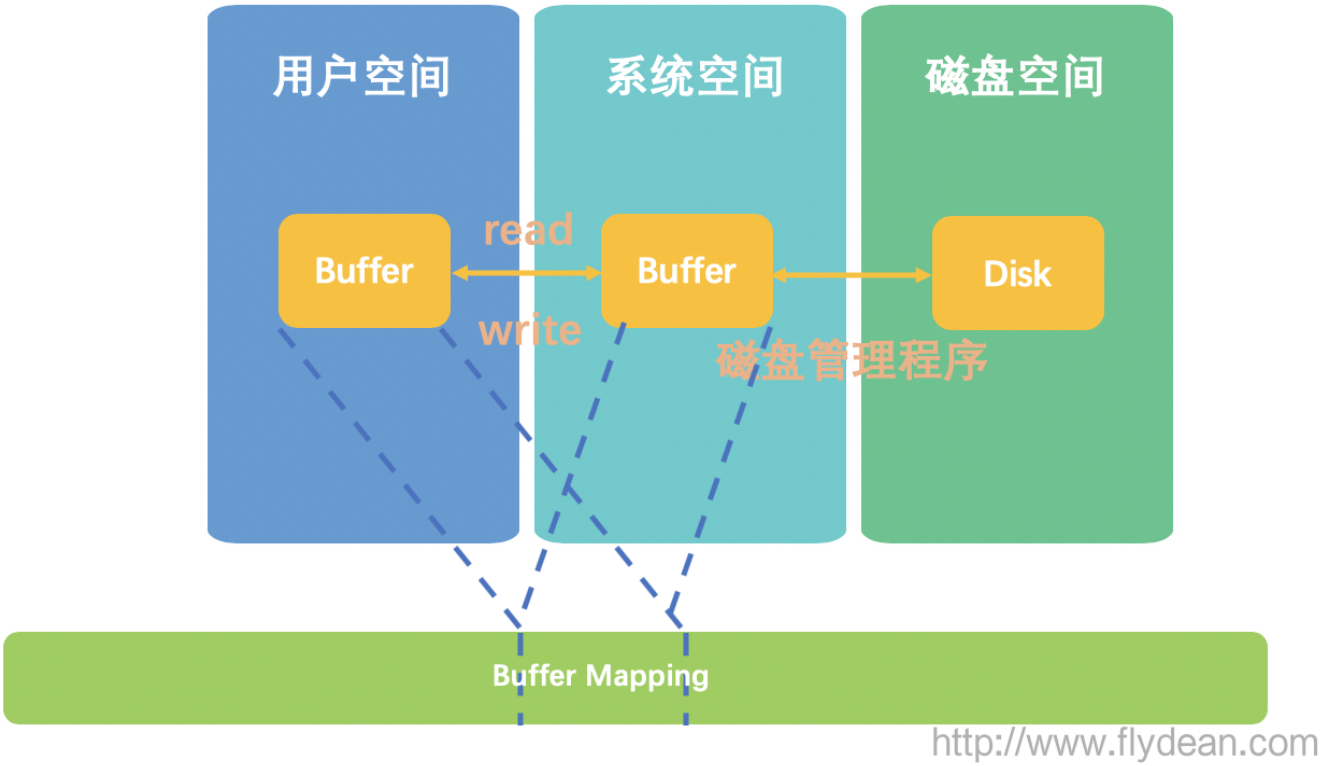
小师妹：F师兄，你有没有发现，最近硬盘的价格真的是好便宜好便宜，1T的硬盘大概要500块，平均1M五毛钱。现在下个电影都1G起步，这是不是意味着我们买入了大数据时代?

没错，小师妹，硬件技术的进步也带来了软件技术的进步，两者相辅相成，缺一不可。

小师妹：F师兄，如果要是去读取G级的文件，有没有什么快捷简单的方法?

还记得上次我们讲的虚拟地址空间吗?

再把上次讲的图搬过来：



通常来说我们的应用程序调用系统的接口从磁盘空间获取Buffer数据，我们把自己的应用程序称之为用户空间，把系统的底层称之为系统空间。

传统的IO操作，是操作系统讲磁盘中的文件读入到系统空间里面，然后再拷贝到用户空间中，供用户使用。

这中间多了一个Buffer拷贝的过程，如果这个量够大的话，其实还是挺浪费时间的。

于是有人在想了，拷贝太麻烦太耗时了，我们单独划出一块内存区域，让系统空间和用户空间同时映射到同一块地址不就省略了拷贝的步骤吗?

这个被划出来的单独的内存区域叫做虚拟地址空间，而不同空间到虚拟地址的映射就叫做Buffer Map。 Java中是有一个专门的MappedByteBuffer来代表这种操作。

小师妹：F师兄，那这个虚拟地址空间和内存有什么区别呢？有了内存还要啥虚拟地址空间?

虚拟地址空间有两个好处。

第一个好处就是虚拟地址空间对于应用程序本身而言是独立的，从而保证了程序的互相隔离和程序中地址的确定性。比如说一个程序如果运行在虚拟地址空间中，那么它的空间地址是固定的，不管他运行多少次。如果直接使用内存地址，那么可能这次运行的时候内存地址可用，下次运行的时候内存地址不可用，就会导致潜在的程序出错。

第二个好处就是虚拟空间地址可以比真实的内存地址大，这个大其实是对内存的使用做了优化，比如说会把很少使用的内存写如磁盘，从而释放出更多的内存来做更有意义的事情，而之前存储到磁盘的数据，当真正需要的时候，再从磁盘中加载到内存中。

这样物理内存实际上可以看做虚拟空间地址的缓存。

详解MappedByteBuffer

小师妹：MappedByteBuffer听起来好神奇，怎么使用它呢?

我们先来看看MappedByteBuffer的定义：

```
public abstract class MappedByteBuffer
    extends ByteBuffer
```

它实际上是一个抽象类，具体的实现有两个：

```
class DirectByteBuffer extends MappedByteBuffer implements DirectBuffer
```

```
class DirectByteBufferR extends DirectByteBuffer
implements DirectBuffer
```

分别是DirectByteBuffer和DirectByteBufferR。

小师妹：F师兄，这两个ByteBuffer有什么区别呢？这个R是什么意思?

R代表的是ReadOnly的意思，可能是因为本身是个类的名字就够长了，所以搞了个缩写。但是也不写个注解，让人看起来十分费解....

我们可以从RandomAccessFile的FilChannel中调用map方法获得它的实例。

我们看下map方法的定义：

```
 public abstract MappedByteBuffer map(MapMode mode, long position, long size)
        throws IOException;
```

MapMode代表的是映射的模式，position表示是map开始的地址，size表示是ByteBuffer的大小。

MapMode

小师妹：F师兄，文件有只读，读写两种模式，是不是MapMode也包含这两类?

对的，其实NIO中的MapMode除了这两个之外，还有一些其他很有趣的用法。

- FileChannel.MapMode.READ_ONLY 表示只读模式
- FileChannel.MapMode.READ_WRITE 表示读写模式
- FileChannel.MapMode.PRIVATE 表示copy-on-write模式，这个模式和READ_ONLY有点相似，它的操作是先对原数据进行拷贝，然后可以在拷贝之后的Buffer中进行读写。但是这个写入并不会影响原数据。可以看做是数据的本地拷贝，所以叫做Private。

基本的MapMode就这三种了，其实除了基础的MapMode，还有两种扩展的MapMode：

- ExtendedMapMode.READ_ONLY_SYNC 同步的读
- ExtendedMapMode.READ_WRITE_SYNC 同步的读写

MappedByteBuffer的最大值

小师妹：F师兄，既然可以映射到虚拟内存空间，那么这个MappedByteBuffer是不是可以无限大?

当然不是了，首先虚拟地址空间的大小是有限制的，如果是32位的CPU，那么一个指针占用的地址就是4个字节，那么能够表示的最大值是0xFFFFFFFF，也就是4G。

另外我们看下map方法中size的类型是long，在java中long能够表示的最大值是0x7fffffff，也就是2147483647字节，换算一下大概是2G。也就是说MappedByteBuffer的最大值是2G，一次最多只能map 2G的数据。

MappedByteBuffer的使用

小师妹，F师兄我们来举两个使用MappedByteBuffer读写的例子吧。

善!

先看一下怎么使用MappedByteBuffer来读数据：

```
public void readWithMap() throws IOException {
        try (RandomAccessFile file = new RandomAccessFile(new
File("src/main/resources/big.www.flydean.com"), "r"))
        {
            //get Channel
            FileChannel fileChannel = file.getChannel();
            //get mappedByteBuffer from fileChannel
            MappedByteBuffer buffer = fileChannel.map(FileChannel.MapMode.READ_ONLY, 0,
fileChannel.size());
            // check buffer
            log.info("is Loaded in physical memory: {}",buffer.isLoaded());  //只是一个提
醒而不是guarantee
            log.info("capacity {}",buffer.capacity());
            //read the buffer
            for (int i = 0; i < buffer.limit(); i++)
            {
                log.info("get {}", buffer.get());
            }
        }
    }
```

然后再看一个使用MappedByteBuffer来写数据的例子：

```
public void writeWithMap() throws IOException {
        try (RandomAccessFile file = new RandomAccessFile(new
File("src/main/resources/big.www.flydean.com"), "rw"))
        {
            //get Channel
            FileChannel fileChannel = file.getChannel();
            //get mappedByteBuffer from fileChannel
            MappedByteBuffer buffer = fileChannel.map(FileChannel.MapMode.READ_WRITE,
0, 4096 * 8 );
            // check buffer
```

```
            log.info("is Loaded in physical memory: {}",buffer.isLoaded());  //只是一个提醒而不是guarantee
            log.info("capacity {}",buffer.capacity());
            //write the content
            buffer.put("www.flydean.com".getBytes());
        }
    }
```

MappedByteBuffer要注意的事项

小师妹：F师兄，MappedByteBuffer因为使用了内存映射，所以读写的速度都会有所提升。那么我们在使用中应该注意哪些问题呢?

MappedByteBuffer是没有close方法的，即使它的FileChannel被close了，MappedByteBuffer仍然处于打开状态，只有JVM进行垃圾回收的时候才会被关闭。而这个时间是不确定的。

总结

本文再次介绍了虚拟地址空间和MappedByteBuffer的使用。

第十三章 NIO中那些奇怪的Buffer

简介

妖魔鬼怪快快显形，今天F师兄帮助小师妹来斩妖除魔啦，什么BufferB，BufferL，BufferRB，BufferRL，BufferS，BufferU，BufferRS，BufferRU统统给你剖析个清清楚楚明明白白。

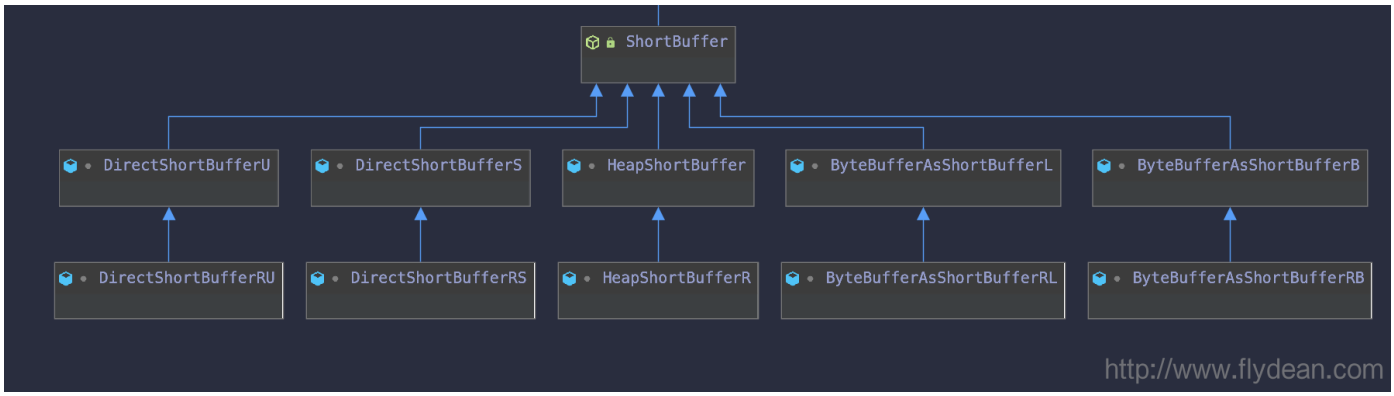
Buffer的分类

小师妹：F师兄不都说JDK源码是最好的java老师吗？为程不识源码，就称牛人也枉然。但是我最近在学习NIO的时候竟然发现有些Buffer类居然没有注释，就那么突兀的写在哪里，让人好生心烦。

> 更多内容请访问www.flydean.com

居然还有这样的事情？快带F师兄去看看。

小师妹：F师兄你看，以ShortBuffer为例，它的子类怎么后面都带一些奇奇怪怪的字符：



什么什么BufferB，BufferL，BufferRB，BufferRL，BufferS，BufferU，BufferRS，BufferRU都来了，点进去看他们的源码也没有说明这些类到底是做什么的。

还真有这种事情，给我一个小时，让我仔细研究研究。

一个小时后，小师妹，经过我一个小时的辛苦勘察，结果发现，确实没有官方文档介绍这几个类到底是什么含义，但是师兄我掐指一算，好像发现了这些类之间的小秘密，且听为兄娓娓道来。

之前的文章，我们讲到Buffer根据类型可以分为ShortBuffer，LongBuffer，DoubleBuffer等等。

但是根据本质和使用习惯，我们又可以分为三类，分别是：ByteBufferAsXXXBuffer，DirectXXXBuffer和HeapXXXBuffer。

ByteBufferAsXXXBuffer主要将ByteBuffer转换成为特定类型的Buffer，比如CharBuffer，IntBuffer等等。

而DirectXXXBuffer则是和虚拟内存映射打交道的Buffer。

最后HeapXXXBuffer是在堆空间上面创建的Buffer。

Big Endian 和 Little Endian

小师妹，F师兄，你刚刚讲的都不重要，我就想知道类后面的B，L，R，S，U是做什么的。

好吧，在给你讲解这些内容之前，师兄我给你讲一个故事。

话说在明末浙江才女吴绛雪写过一首诗：《春 景 诗》

> 莺啼岸柳弄春晴，
> 柳弄春晴夜月明。
> 明月夜晴春弄柳，
> 晴春弄柳岸啼莺。

小师妹，可有看出什么特异之处？最好是多读几遍，读出声来。

小师妹：哇，F师兄，这首诗从头到尾和从尾到头读起来是一样的呀，又对称又有意境！

不错，这就是中文的魅力啦，根据读的方式不同，得出的结果也不同，其实在计算机世界也存在这样的问题。

我们知道在java中底层的最小存储单元是Byte，一个Byte是8bits，用16进制表示就是Ox00-OxFF。

java中除了byte，boolean是占一个字节以外，好像其他的类型都会占用多个字节。

如果以int来举例，int占用4个字节，其范围是从Ox00000000-OxFFFFFFFF,假如我们有一个int=Ox12345678，存到内存地址里面就有这样两种方式。

第一种Big Endian将高位的字节存储在起始地址

第二种Little Endian将地位的字节存储在起始地址

其实Big Endian更加符合人类的读写习惯，而Little Endian更加符合机器的读写习惯。

目前主流的两大CPU阵营中，PowerPC系列采用big endian方式存储数据，而x86系列则采用little endian方式存储数据。

如果不同的CPU架构直接进行通信，就由可能因为读取顺序的不同而产生问题。

java的设计初衷就是一次编写处处运行，所以自然也做了设计。

所以BufferB表示的是Big Endian的buffer，BufferL表示的是Little endian的Buffer。

而BufferRB，BufferRL表示的是两种只读Buffer。

aligned内存对齐

小师妹：F师兄，那这几个又是做什么用的呢？ BufferS，BufferU，BufferRS，BufferRU。

在讲解这几个类之前，我们先要回顾一下JVM中对象的存储方式。

还记得我们是怎么使用JOL来分析JVM的信息的吗？代码非常非常简单：

```
log.info("{}", VM.current().details());
```

输出结果：

```
## Running 64-bit HotSpot VM.
## Using compressed oop with 3-bit shift.
## Using compressed klass with 3-bit shift.
## WARNING | Compressed references base/shifts are guessed by the experiment!
## WARNING | Therefore, computed addresses are just guesses, and ARE NOT RELIABLE.
## WARNING | Make sure to attach Serviceability Agent to get the reliable addresses.
## Objects are 8 bytes aligned.
## Field sizes by type: 4, 1, 1, 2, 2, 4, 4, 8, 8 [bytes]
## Array element sizes: 4, 1, 1, 2, 2, 4, 4, 8, 8 [bytes]
```

上面的输出中，我们可以看到：Objects are 8 bytes aligned，这意味着所有的对象分配的字节都是8的整数倍。

再注意上面输出的一个关键字aligned，确认过眼神，是对的那个人。

aligned对齐的意思，表示JVM中的对象都是以8字节对齐的，如果对象本身占用的空间不足8字节或者不是8字节的倍数，则补齐。

还是用JOL来分析String对象：

```
[main] INFO com.flydean.JolUsage - java.lang.String object internals:
 OFFSET  SIZE      TYPE DESCRIPTION                               VALUE
      0    12           (object header)                           N/A
     12     4    byte[] String.value                               N/A
     16     4       int String.hash                                N/A
     20     1      byte String.coder                               N/A
     21     1   boolean String.hashIsZero                          N/A
     22     2           (loss due to the next object alignment)
Instance size: 24 bytes
Space losses: 0 bytes internal + 2 bytes external = 2 bytes total
```

可以看到一个String对象占用24字节，但是真正有意义的是22字节，有两个2字节是补齐用的。

对齐的好处显而易见，就是CPU在读取数据的时候更加方便和快捷，因为CPU设定是一次读取多少字节来的，如果你存储是没有对齐的，则CPU读取起来效率会比较低。

现在可以回答部分问题：BufferU表示是unaligned，BufferRU表示是只读的unaligned。

小师妹：那BufferS和BufferRS呢?

这个问题其实还是很难回答的，但是经过师兄我的不断研究和探索，终于找到了答案：

先看下DirectShortBufferRU和DirectShortBufferRS的区别，两者的区别在两个地方，先看第一个Order：

```
DirectShortBufferRU:

public ByteOrder order() {
        return ((ByteOrder.nativeOrder() != ByteOrder.BIG_ENDIAN)
                ? ByteOrder.LITTLE_ENDIAN : ByteOrder.BIG_ENDIAN);
    }
```

```
DirectShortBufferRS:

public ByteOrder order() {
        return ((ByteOrder.nativeOrder() == ByteOrder.BIG_ENDIAN)
                ? ByteOrder.LITTLE_ENDIAN : ByteOrder.BIG_ENDIAN);
    }
```

可以看到DirectShortBufferRU的Order是跟nativeOrder是一致的。而DirectShortBufferRS的Order跟nativeOrder是相反的。

为什么相反？再看两者get方法的不同：

```
DirectShortBufferU:

public short get() {
        try {
            checkSegment();
            return ((UNSAFE.getShort(ix(nextGetIndex()))));
        } finally {
            Reference.reachabilityFence(this);
        }
    }
```

```
DirectShortBufferS:

public short get() {
        try {
            checkSegment();
            return (Bits.swap(UNSAFE.getShort(ix(nextGetIndex()))));
        } finally {
            Reference.reachabilityFence(this);
        }
    }
```

区别出来了，DirectShortBufferS在返回的时候做了一个bits的swap操作。

所以BufferS表示的是swap过后的Buffer，和BufferRS表示的是只读的swap过后的Buffer。

总结

不写注释实在是害死人啊！尤其是JDK自己也不写注释的情况下！

第十四章 用Selector来说再见

简介

NIO有三宝:Buffer,Channel，Selector少不了。本文将会介绍NIO三件套中的最后一套Selector，并在理解Selector的基础上，协助小师妹发一张好人卡。我们开始吧。

Selector介绍

小师妹：F师兄，最近我的桃花有点旺，好几个师兄莫名其妙的跟我打招呼，可是我一心向着工作，不想谈论这些事情。毕竟先有事业才有家嘛。我又不好直接拒绝，有没有什么比较隐晦的方法来让他们放弃这个想法?

> 更多内容请访问www.flydean.com

这个问题，我沉思了大约0.001秒，于是给出了答案：给他们发张好人卡吧，应该就不会再来纠缠你了。

小师妹：F师兄，如果给他们发完好人卡还没有用呢?

那就只能切断跟他们的联系了，来个一刀两断。哈哈。

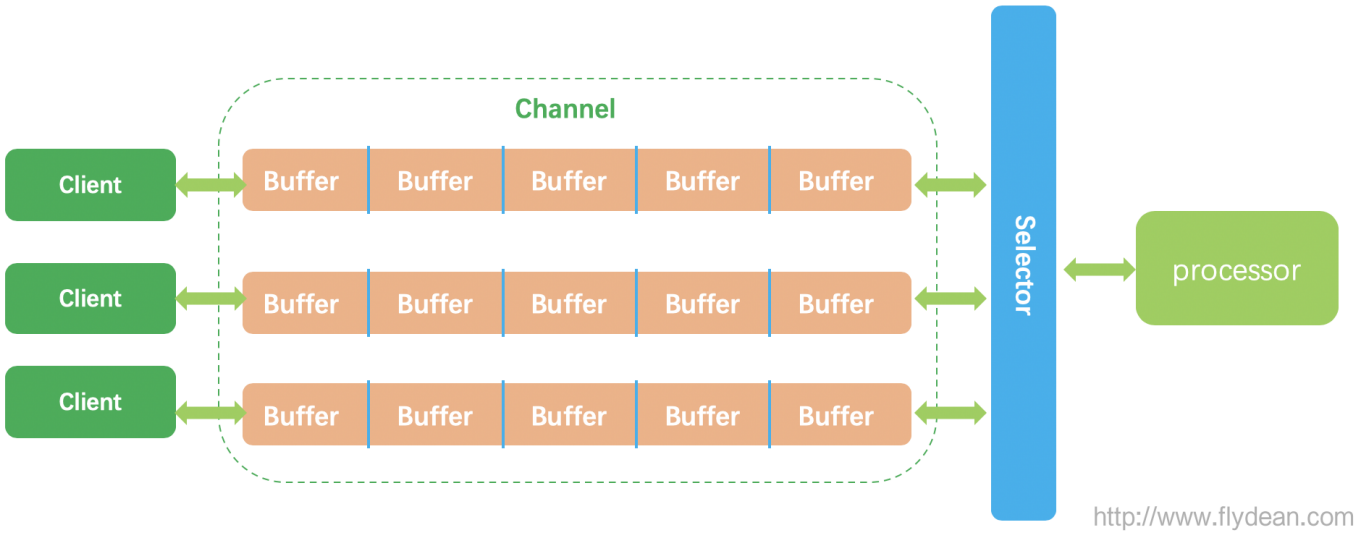
这样吧，小师妹你最近不是在学NIO吗？刚好我们可以用Selector来模拟一下发好人卡的过程。

假如你的志伟师兄和子丹师兄想跟你建立联系，每个人都想跟你建立一个沟通通道，那么你就需要创建两个channel。

两个channel其实还好，如果有多个人都想同时跟你建立联系通道，那么要维持这些通道就需要保持连接，从而浪费了资源。

但是建立的这些连接并不是时时刻刻都有消息在传输，所以其实大多数时间这些建立联系的通道其实是浪费的。

如果使用Selector就可以只启用一个线程来监听通道的消息变动，这就是Selector。



从上面的图可以看出，Selector监听三个不同的channel，然后交给一个processor来处理，从而节约了资源。

创建Selector

先看下selector的定义：

```
public abstract class Selector implements Closeable
```

Selector是一个abstract类，并且实现了Closeable，表示Selector是可以被关闭的。

虽然Selector是一个abstract类，但是可以通过open来简单的创建：

```
Selector selector = Selector.open();
```

如果细看open的实现可以发现一个很有趣的现象：

```
public static Selector open() throws IOException {
        return SelectorProvider.provider().openSelector();
    }
```

open方法调用的是SelectorProvider中的openSelector方法。

再看下provider的实现：

```
 public SelectorProvider run() {
   if (loadProviderFromProperty())
        return provider;
    if (loadProviderAsService())
        return provider;
      provider = sun.nio.ch.DefaultSelectorProvider.create();
      return provider;
    }
 });
```

有三种情况可以加载一个SelectorProvider，如果系统属性指定了java.nio.channels.spi.SelectorProvider，那么从指定的属性加载。

如果没有直接指定属性，则从ServiceLoader来加载。

最后如果都找不到的情况下，使用默认的DefaultSelectorProvider。

关于ServiceLoader的用法，我们后面会有专门的文章来讲述。这里先不做多的解释。

注册Selector到Channel中

```
ServerSocketChannel serverSocketChannel = ServerSocketChannel.open();
        serverSocketChannel.bind(new InetSocketAddress("localhost", 9527));
        serverSocketChannel.configureBlocking(false);
        serverSocketChannel.register(selector, SelectionKey.OP_ACCEPT);
```

如果是在服务器端，我们需要先创建一个ServerSocketChannel，绑定Server的地址和端口，然后将Blocking设置为false。因为我们使用了Selector，它实际上是一个非阻塞的IO。

> 注意FileChannels是不能使用Selector的，因为它是一个阻塞型IO。

小师妹：F师兄，为啥FileChannel是阻塞型的呀？做成非阻塞型的不是更快？

小师妹，我们使用FileChannel的目的是什么？就是为了读文件呀，读取文件肯定是一直读一直读，没有可能读一会这个channel再读另外一个channel吧，因为对于每个channel自己来讲，在文件没读取完之前，都是繁忙状态，没有必要在channel中切换。

最后我们将创建好的Selector注册到channel中去。

SelectionKey

SelectionKey表示的是我们希望监听到的事件。

总的来说，有4种Event：

- SelectionKey.OP_READ 表示服务器准备好，可以从channel中读取数据。
- SelectionKey.OP_WRITE 表示服务器准备好，可以向channel中写入数据。
- SelectionKey.OP_CONNECT 表示客户端尝试去连接服务端
- SelectionKey.OP_ACCEPT 表示服务器accept一个客户端的请求

```
public static final int OP_READ = 1 << 0;
public static final int OP_WRITE = 1 << 2;
public static final int OP_CONNECT = 1 << 3;
public static final int OP_ACCEPT = 1 << 4;
```

我们可以看到上面的4个Event是用位运算来定义的，如果将这个四个event使用或运算合并起来，就得到了SelectionKey中的interestOps。

和interestOps类似，SelectionKey还有一个readyOps。

一个表示感兴趣的操作，一个表示ready的操作。

最后，SelectionKey在注册的时候，还可以attach一个Object，比如我们可以在这个对象中保存这个channel的id：

```
SelectionKey key = channel.register(
  selector, SelectionKey.OP_ACCEPT, object);
key.attach(Object);
Object object = key.attachment();
```

object可以在register的时候传入，也可以调用attach方法。

最后，我们可以通过key的attachment方法，获得该对象。

selector 和 SelectionKey

我们通过selector.select()这个一个blocking操作，来获取一个ready的channel。

然后我们通过调用selector.selectedKeys()来获取到SelectionKey对象。

在SelectionKey对象中，我们通过判断ready的event来处理相应的消息。

总的例子

接下来，我们把之前将的串联起来，先建立一个小师妹的ChatServer：

```
public class ChatServer {

    private static String BYE_BYE="再见";

    public static void main(String[] args) throws IOException, InterruptedException {
        Selector selector = Selector.open();
        ServerSocketChannel serverSocketChannel = ServerSocketChannel.open();
        serverSocketChannel.bind(new InetSocketAddress("localhost", 9527));
        serverSocketChannel.configureBlocking(false);
        serverSocketChannel.register(selector, SelectionKey.OP_ACCEPT);
        ByteBuffer byteBuffer = ByteBuffer.allocate(512);

        while (true) {
            selector.select();
            Set<SelectionKey> selectedKeys = selector.selectedKeys();
            Iterator<SelectionKey> iter = selectedKeys.iterator();
            while (iter.hasNext()) {
                SelectionKey selectionKey = iter.next();
                if (selectionKey.isAcceptable()) {
                    register(selector, serverSocketChannel);
                }
                if (selectionKey.isReadable()) {
                    serverResonse(byteBuffer, selectionKey);
                }
                iter.remove();
            }
            Thread.sleep(1000);
        }
    }

    private static void serverResonse(ByteBuffer byteBuffer, SelectionKey selectionKey)
            throws IOException {
        SocketChannel socketChannel = (SocketChannel) selectionKey.channel();
        socketChannel.read(byteBuffer);
        byteBuffer.flip();
        byte[] bytes= new byte[byteBuffer.limit()];
        byteBuffer.get(bytes);
        log.info(new String(bytes).trim());
        if(new String(bytes).trim().equals(BYE_BYE)){
            log.info("说再见不如不见！");
            socketChannel.write(ByteBuffer.wrap("再见".getBytes()));
            socketChannel.close();
        }else {
            socketChannel.write(ByteBuffer.wrap("你是个好人".getBytes()));
        }
        byteBuffer.clear();
    }
```

```
    private static void register(Selector selector, ServerSocketChannel
serverSocketChannel)
            throws IOException {
        SocketChannel socketChannel = serverSocketChannel.accept();
        socketChannel.configureBlocking(false);
        socketChannel.register(selector, SelectionKey.OP_READ);
    }
}
```

上面例子有两点需要注意，我们在循环遍历中，当selectionKey.isAcceptable时，表示服务器收到了一个新的客户端连接，这个时候我们需要调用register方法，再注册一个OP_READ事件到这个新的SocketChannel中，然后继续遍历。

第二，我们定义了一个stop word，当收到这个stop word的时候，会直接关闭这个client channel。

再看看客户端的代码：

```
public class ChatClient {

    private static SocketChannel socketChannel;
    private static ByteBuffer byteBuffer;

    public static void main(String[] args) throws IOException {

        ChatClient chatClient = new ChatClient();
        String response = chatClient.sendMessage("hello 小师妹! ");
        log.info("response is {}", response);
        response = chatClient.sendMessage("能不能? ");
        log.info("response is {}", response);
        chatClient.stop();

    }

    public void stop() throws IOException {
        socketChannel.close();
        byteBuffer = null;
    }

    public ChatClient() throws IOException {
        socketChannel = SocketChannel.open(new InetSocketAddress("localhost", 9527));
        byteBuffer = ByteBuffer.allocate(512);
    }

    public String sendMessage(String msg) throws IOException {
        byteBuffer = ByteBuffer.wrap(msg.getBytes());
        String response = null;
        socketChannel.write(byteBuffer);
        byteBuffer.clear();
        socketChannel.read(byteBuffer);
```

```
            byteBuffer.flip();
            byte[] bytes= new byte[byteBuffer.limit()];
            byteBuffer.get(bytes);
            response =new String(bytes).trim();
            byteBuffer.clear();
            return response;

        }
    }
```

客户端代码没什么特别的，需要注意的是Buffer的读取。

最后输出结果：

```
server收到：  INFO com.flydean.ChatServer - hello 小师妹!
client收到：INFO com.flydean.ChatClient - response is 你是个好人
server收到：  INFO com.flydean.ChatServer - 能不能?
client收到：  INFO com.flydean.ChatClient - response is 再见
```

解释一下整个流程：志伟跟小师妹建立了一个连接，志伟向小师妹打了一个招呼，小师妹给志伟发了一张好人卡。志伟不死心，想继续纠缠，小师妹回复再见，然后自己关闭了通道。

总结

本文介绍了Selector和channel在发好人卡的过程中的作用。

第十五章 文件编码和字符集Unicode

简介

小师妹一时兴起，使用了一项从来都没用过的新技能，没想却出现了一个无法解决的问题。把大象装进冰箱到底有几步？乱码的问题又是怎么解决的？快来跟F师兄一起看看吧。

使用Properties读取文件

这天，小师妹心情很愉悦，吹着口哨唱着歌，标准的45度俯视让人好不自在。

小师妹呀，什么事情这么高兴，说出来让师兄也沾点喜庆?

小师妹：F师兄，最新我发现了一种新型的读取文件的方法，很好用的，就跟map一样：

```
public void usePropertiesFile() throws IOException {
        Properties configProp = new Properties();
        InputStream in =
this.getClass().getClassLoader().getResourceAsStream("www.flydean.com.properties");
        configProp.load(in);
        log.info(configProp.getProperty("name"));
        configProp.setProperty("name", "www.flydean.com");
        log.info(configProp.getProperty("name"));
    }
```

F师兄你看，我使用了Properties来读取文件，文件里面的内容是key=value形式的，在做配置文件使用的时候非常恰当。我是从Spring项目中的properties配置文件中得到的灵感，才发现原来java还有一个专门读取属性文件的类Properties。

小师妹现在都会抢答了，果然青出于蓝。

乱码初现

小师妹你做得非常好，就这样触类旁通，很快java就要尽归你手了，后面的什么scala，go，JS等估计也统统不在话下。再过几年你就可以升任架构师，公司技术在你的带领之下一定会蒸蒸日上。

做为师兄，最大的责任就是给小师妹以鼓励和信心，给她描绘美好的未来，什么出任CEO，赢取高富帅等全都不在话下。听说有个专业的词汇来描述这个过程叫做：画饼。

小师妹有点心虚：可是F师兄，我还有点小小的问题没有解决，有点中文的小小乱码....

我深有体会的点点头：马赛克是阻碍人类进步的绊脚石...哦，不是马赛克，是文件乱码，要想弄清楚这个问题，还要从那个字符集和文件编码讲起。

字符集和文件编码

在很久很久以前，师兄我都还没有出生的时候，西方世界出现了一种叫做计算机的高科技产品。

初代计算机只能做些简单的算数运算，还要使用人工打孔的程序才能运行，不过随着时间的推移，计算机的体积越来越小，计算能力越来越强，打孔已经不存在了，变成了人工编写的计算机语言。

一切都在变化，唯有一件事情没有变化。这件事件就是计算机和编程语言只流传在西方。而西方日常交流使用26个字母加有限的标点符号就够了。

最初的计算机存储可以是非常昂贵的，我们用一个字节也就是8bit来存储所有能够用到的字符，除了最开始的1bit不用以外，总共有128中选择，装26个小写+26个大写字母和其他的一些标点符号之类的完全够用了。

这就是最初的ASCII编码，也叫做美国信息交换标准代码（American Standard Code for Information Interchange）。

后面计算机传到了全球，人们才发现好像之前的ASCII编码不够用了，比如中文中常用的汉字就有4千多个，怎么办呢?

没关系，将ASCII编码本地化，叫做ANSI编码。1个字节不够用就用2个字节嘛，路是人走出来的，编码也是为人来服务的。于是产生了各种如GB2312, BIG5, JIS等各自的编码标准。这些编码虽然与ASCII编码兼容，但是相互之间却并不兼容。

这严重的影响了国际化的进程，这样还怎么去实现同一个地球，同一片家园的梦想?

于是国际组织出手了，制定了UNICODE字符集，为所有语言的所有字符都定义了一个唯一的编码，unicode的字符集是从U+0000到U+10FFFF这么多个编码。

小师妹：F师兄，那么unicode和我平时听说的UTF-8，UTF-16，UTF-32有什么关系呢?

我笑着问小师妹：小师妹，把大象装进冰箱有几步?

小师妹：F师兄，脑筋急转弯的故事，已经不适合我了，大象装进冰箱有三步，第一打开冰箱，第二把大象装进去，第三关上冰箱，完事了。

小师妹呀，作为一个有文化的中国人，要真正的承担起民族复兴，科技进步的大任，你的想法是很错误的，不能光想口号，要有实际的可操作性的方案才行，要不然我们什么时候才能够打造秦芯，唐芯和明芯呢?

师兄说的对，可是这跟unicode有什么关系呢?

unicode字符集最后是要存储到文件或者内存里面的，那怎么存呢？使用固定的1个字节，2个字节还是用变长的字节呢？根据编码方式的不同，可以分为UTF-8，UTF-16，UTF-32等多种编码方式。

其中UTF-8是一种变长的编码方案，它使用1-4个字节来存储。UTF-16使用2个或者4个字节来存储，JDK9之后的String的底层编码方式变成了两种：LATIN1和UTF16。

而UTF-32是使用4个字节来存储。这三种编码方式中，只有UTF-8是兼容ASCII的，这也是为什么国际上UTF-8编码方式比较通用的原因（毕竟计算机技术都是西方人搞出来的）。

解决Properties中的乱码

小师妹，要解决你Properties中的乱码问题很简单，Reader基本上都有一个Charsets的参数，通过这个参数可以传入要读取的编码方式，我们把UTF-8传进去就行了：

```
public void usePropertiesWithUTF8() throws IOException{
        Properties configProp = new Properties();
        InputStream in =
this.getClass().getClassLoader().getResourceAsStream("www.flydean.com.properties");
        InputStreamReader inputStreamReader= new InputStreamReader(in,
StandardCharsets.UTF_8);
        configProp.load(inputStreamReader);
        log.info(configProp.getProperty("name"));
        configProp.setProperty("name", "www.flydean.com");
        log.info(configProp.getProperty("name"));
    }
```

上面的代码中，我们使用InputStreamReader封装了InputStream，最终解决了中文乱码的问题。

真.终极解决办法

小师妹又有问题了：F师兄，这样做是因为我们知道文件的编码方式是UTF-8，如果不知道该怎么办呢？是选UTF-8，UTF-16还是UTF-32呢?

小师妹问的问题越来越刁钻了，还好这个问题我也有准备。

接下来介绍我们的终极解决办法，我们将各种编码的字符最后都转换成unicode字符集存到properties文件中，再读取的时候是不是就没有编码的问题了？

转换需要用到JDK自带的工具：

```
 native2ascii -encoding utf-8 file/src/main/resources/www.flydean.com.properties.utf8
file/src/main/resources/www.flydean.com.properties.cn
```

上面的命令将utf-8的编码转成了unicode。

转换前：

```
site=www.flydean.com
name=程序那些事
```

转换后：

```
site=www.flydean.com
name=\u7a0b\u5e8f\u90a3\u4e9b\u4e8b
```

再运行下测试代码：

```
public void usePropertiesFileWithTransfer() throws IOException {
        Properties configProp = new Properties();
        InputStream in =
this.getClass().getClassLoader().getResourceAsStream("www.flydean.com.properties.cn");
        configProp.load(in);
        log.info(configProp.getProperty("name"));
        configProp.setProperty("name", "www.flydean.com");
        log.info(configProp.getProperty("name"));
    }
```

输出正确的结果。

如果要做国际化支持，也是这样做的。

总结

千辛万苦终于解决了小师妹的问题，F师兄要休息一下。

本文的例子https://github.com/ddean2009/learn-java-io-nio

> 本文作者：flydean程序那些事
>
> 本文链接：www.flydean.com
>
> 本文来源：flydean的博客
>
> 欢迎关注我的公众号:程序那些事，更多精彩等着您！