http://wiki.jikexueyuan.com/project/java-nio-zh/java-nio-asynchronousfilechannel.html

Java7中新增了AsynchronousFileChannel作为nio的一部分。AsynchronousFileChannel使得数据可以进行异步读写。下面将介绍一下AsynchronousFileChannel的使用。

创建AsynchronousFileChannel

AsynchronousFileChannel的创建可以通过open()静态方法：

```
Path path = Paths.get("data/test.xml");
AsynchronousFileChannel fileChannel
    = AsynchronousFileChannel.open(path,
StandardOpenOption.READ);
```

open()的第一个参数是一个Path实体，指向我们需要操作的文件。 第二个参数是操作类型。上述示例中我们用的是StandardOpenOption.READ，表示以读的形式操作文件。

读取数据（Reading Data）

读取AsynchronousFileChannel的数据有两种方式。每种方法都会调用AsynchronousFileChannel的一个read()接口。下面分别看一下这两种写法。

通过Future读取数据（Reading Data Via a Future）

第一种方式是调用返回值为Future的read()方法：

```
Future<Integer> operation = fileChannel.read(buffer, 0);
```

这种方式中，read()接受一个ByteBuffer座位第一个参数，数据会被读取到ByteBuffer中。

第二个参数是开始读取数据的位置。

read()方法会立刻返回，即使读操作没有完成。我们可以通过isDone()方法检查操作是否完成。

下面是一个略长的示例：

```
 AsynchronousFileChannel fileChannel
    = AsynchronousFileChannel.open(path,
StandardOpenOption.READ);

ByteBuffer buffer = ByteBuffer.allocate(1024);
long position = 0;

Future<Integer> operation = fileChannel.read(buffer,
position);
```

```
while(!operation.isDone());

buffer.flip();
byte[] data = new byte[buffer.limit()];
buffer.get(data);
System.out.println(new String(data));
buffer.clear();
```

在这个例子中我们创建了一个AsynchronousFileChannel，然后创建一个ByteBuffer作为参数传给read。接着我们创建了一个循环来检查是否读取完毕isDone()。这里的循环操作比较低效，它的意思是我们需要等待读取动作完成。一旦读取完成后，我们就可以把数据写入ByteBuffer，然后输出。

通过CompletionHandler读取数据（Reading Data Via a CompletionHandler）

另一种方式是调用接收CompletionHandler作为参数的read()方法。下面是具体的使用：

```
fileChannel.read(buffer, position, buffer,
  new CompletionHandler<Integer, ByteBuffer>() {
    @Override
    public void completed(Integer result, ByteBuffer
attachment) {
        System.out.println("result = " + result);

        attachment.flip();
        byte[] data = new byte[attachment.limit()];
        attachment.get(data);
        System.out.println(new String(data));
        attachment.clear();
    }

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

    }
});
```

这里，一旦读取完成，将会触发CompletionHandler的completed()方法，并传入一个Integer和ByteBuffer。前面的整型表示的是读取到的字节数大小。第二个ByteBuffer也可以换成其他合适的对象方便数据写入。 如果读取操作失败了，那么会触发failed()方法。

写数据（Writing Data）

和读数据类似某些数据也有两种方式，调动不同的的write()方法，下面分别看介绍这两种方法。

通过Future写数据（Writing Data Via a Future）

通过AsynchronousFileChannel我们可以一步写数据

```
Path path = Paths.get("data/test-write.txt");
AsynchronousFileChannel fileChannel
    = AsynchronousFileChannel.open(path,
StandardOpenOption.WRITE);

ByteBuffer buffer = ByteBuffer.allocate(1024);
long position = 0;
buffer.put("test data".getBytes());
buffer.flip();

Future<Integer> operation = fileChannel.write(buffer,
position);
buffer.clear();

while(!operation.isDone());

System.out.println("Write done");
```

首先把文件已写方式打开，接着创建一个ByteBuffer座位写入数据的目的地。再把数据进入ByteBuffer。最后检查一下是否写入完成。 需要注意的是，这里的文件必须是已经存在的，否者在尝试write数据是会抛出一个java.nio.file.NoSuchFileException.

检查一个文件是否存在可以通过下面的方法：

```
if(!Files.exists(path)){
    Files.createFile(path);
}
```

通过CompletionHandler写数据（Writing Data Via a CompletionHandler）

我们也可以通过CompletionHandler来写数据：

```
Path path = Paths.get("data/test-write.txt");
if(!Files.exists(path)){
    Files.createFile(path);
}
AsynchronousFileChannel fileChannel
    = AsynchronousFileChannel.open(path,
StandardOpenOption.WRITE);
```

```
ByteBuffer buffer = ByteBuffer.allocate(1024);
long position = 0;

buffer.put("test data".getBytes());
buffer.flip();

fileChannel.write(buffer, position, buffer,
  new CompletionHandler<Integer, ByteBuffer>() {
    @Override
    public void completed(Integer result, ByteBuffer
attachment) {
        System.out.println("bytes written: " + result);
    }

    @Override
    public void failed(Throwable exc, ByteBuffer attachment) {
        System.out.println("Write failed");
        exc.printStackTrace();
    }
});
```

同样当数据吸入完成后completed()会被调用，如果失败了那么failed()会被调用。