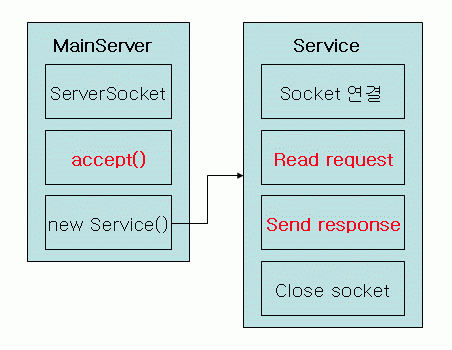
**주의: 본문서는 마이크로소프트웨어 2002~11월,12월호에 송지훈님이 기고한 글로써 본인의 동의없이 무단 배포하는 것을 금지함. 만약 글을 다른 곳에 포스팅하려 할 경우 반드시 강좌의 URL 링크를 사용해야함.   
  
  
J2SDK1.4에 추가된 nio로 비동기식 고가용성 서버 만들기**

***만든이: 송지훈***  
***소속:***[***JavaCafe***](http://www.javacafe.or.kr/)***부시샵***  
***email:***[***johnleen@hanmail.net***](mailto:johnleen@hanmail.net)

기존의 기술에 대응해 새로운 기술이 나온다는 것은 기존 기술에 어떤 문제점 있고 새로운 기술이 그런 문제점을 해결 또는 어느 정도 극복할 수 있다는 것을 의미한다. 이 글은 기존 io를 이용해 서버 프로그램을 할 경우 어떤 문제점들이 있는지를 알아보고 새로운 기술인 nio 에서 어떻게 기존의 문제점들을 극복하는지 간단한 채팅 서버를 만들어보면서 살펴볼 것이다. 자, 그럼 새로운 nio 의 세계로 떠나보자.   
  
자바는 그 태생 목적이 가전제품과 같은 곳에 들어가는 임베디드 시스템이었다. 앞으로는 자바의 태생 목적이 점차 되살아나서 각 가정의 PC가 서버화되고, 그 PC를 중심으로 한 가정이 네트워크화되어 여타 가전제품들을 제어하게 될 것이다. 즉, 흔히 말하는 홈 네트워킹이 일반화되는 것이다. 이 홈 네트워킹에서 가장 핵심적인 역할을 하는 것은 분명 각 가정의 개인 PC다. 우리는 이 홈 네트워크를 통해 인터넷, PDA, 핸드폰 등으로 집안의 가전제품들을 집 밖에서도 자유롭게 제어할 수 있게 된다. 집안 또는 집밖에서 특정 디바이스로 네트워크에 접속해서 이미 네트워크에 접속된 가전제품을 검색하고 제어할 수 있는 것이다. 물론 앞에서 말한 것은 RMI를 기반으로 하고 있는 지니(Jini)라는 자바기술로 구현되겠지만, 썬마이크로시스템즈에 라이선스를 지불해야한다는 점 때문에 다른 형태의 단순한 홈 네트워크를 지원하는 간단한 형태의 오픈 프레임워크가 나올 수도 있을 것 같다. 꼭 가전제품을 제어하는 것이 아니더라도 가정의 홈 서버를 이용해서 도시가스 사용량, 전기 사용량 등을 원격지의 회사 서버에서 조회하거나 혹은 자동으로 서버에 보내서 요금을 계산하도록 자동화하는 등의 다양한 부분으로 많은 응용 서비스들의 제공이 가능할 것이다. 앞으로는 지금의 채팅이나 게임 서버에 한정되어 사용되는 네트워크 프로그램 쪽의 관심과 수요가 좀 더 높아질 것이다. 그러나 기존의 io와 소켓은 이런 서버를 만들 때 많은 문제점을 갖고 있다. 바로 io와 관련된 작업이 C/C++에 비해 상대적으로 너무 느렸고 비동기식 통신이 지원되지 않았다는 점이다. nio는 이런 기존 io의 성능 문제 때문에 이미 C/C++ 에선 일반적으로 사용되는 비동기식 통신을 지원하기 위해 새롭게 추가된 패키지다. 그럼 기존 io에 단점이 어떤 것인지를 살펴보고 nio에선 어떻게 그 문제를 해결하는지 살펴보자.  
  
**빈번한 블러킹 발생과 쓰레드 과부하**우선 자바에서 기존 io와 소켓을 이용해 만든 서버를 블러킹(Blocking) 서버라고 한다. 그 이유는 곳곳에 너무 많은 블러킹이 존재했기 때문인데 여기서 블러킹이란 앞선 요청의 처리가 다 처리될 때까지 뒤의 요청이 기다려야 하는 것을 말한다. 기존의 일반적인 형태의 서버에서 어느 부분이 블러킹이 되는지 <그림 1>에서 살펴보자.  
  
  
<그림 1> 기존 서버에서 블러킹이 되는 부분  
  
<그림 1>에서 빨간색 글씨로 표시된 부분이 블러킹이 되는 부분이다. 블러킹은 자바의 synchronize 키워드와 같다고 생각하면 쉽게 이해될 것이다. 즉, 어떤 작업을 하기 위해 먼저 접근한 요청이 다 끝나기 전에는 뒤이어 접근한 요청들은 먼저 들어온 요청이 다 끝나기를 기다려야 하는 것이다. 만약 앞선 요청이 어떤 문제점에 의해 완전히 처리되지 않고 블럭된 상태로 있게 된다면 뒤이은 요청들은 영원히 앞선 요청이 끝나기만을 기다릴 것이다. 이에 반해 Non-Blocking 은 들어온 요청을 바로 처리하는 것이다. 즉, 앞선 요청이 다 끝나기를 기다리지 않아도 된다. 먼저 다음과 같이 서버에 접속해서 accept()를 호출하는 부분에서 블러킹이 발생한다.

ServerSocket ss = new ServerSocket(4567);   
while(true) {  
Socket s = ss.accept();  
...  
// 동시에 여러 클라이언트들의 요청을 수행하기 위해 별도의 쓰레드를 만들어서 처리함.   
Service service = new Service(s);  
service.start();  
...  
}

앞에서 accept()는 어떤 클라이언트가 접속할 때까지 블러킹이 된다. 만약 어떤 클라이언트가 accept()를 호출하면 뒤이어 이 서버 소켓으로 접속한 클라이언트는 먼저 접속한 클라이언트의 요청이 다 처리되기를 기다려야한다(앞의 코드에서는 while문 안의 accept() 이하의 부분이 다 처리될 때까지 기다림) 또한 동시에 여러 클라이언트들의 요청을 처리하기 위해서 별도의 쓰레드로 Service 클래스를 만들어서 처리하는데, 이것은 사용자가 늘어날 경우 클라이언트 한 명마다 하나의 쓰레드를 할당해주는 형태가 되므로 쓰레드 과부하를 가져오게 된다.   
  
◆ **Thread issue** : 쓰레드는 그 자체적으로도 생성하는데 시간이 걸리는 느린 작업이기도 하고 각각의 쓰레드들이 자신만의 고유한 스택(stack) 영역과 CPU를 점유해서 사용하기 때문에 앞의 코드 부분처럼 많은 쓰레드를 생성해야 하는 서버는 메모리와 CPU를 효율적으로 사용하지 못한다. 또한 클라이언트들의 동시 처리를 위해 생성된 Service 쓰레드들이 대부분의 처리 시간을 요청/응답의 블러킹 부분에서 소비한다는 것도 문제점이다. 그리고 결정적으로 하나의 JVM 은 몇 백개까지의 쓰레드를 생성해서 운영할 수 있지만 수 천개의 쓰레드를 생성할 수는 없다. 또한 시스템에 따라 그 시점은 다르지만 대개의 경우 특정 개수 이상의 쓰레드를 생성하면 급격한 성능 저하를 보이기도 하기 때문이다.   
  
다음 코드는 Service 클래스에서 클라이언트 요청을 읽고 응답을 보내는 부분에서 블러킹이 발생하는 것을 나타낸 것이다.

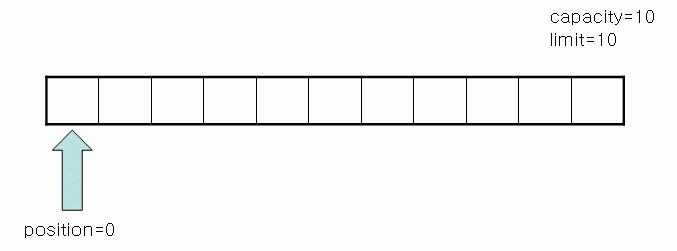
public class Service implements Thread {  
private Socket s;  
private DataInputStream in;  
private DataOutputStream out;  
public Service(Socket s) {  
this.s = s;  
// 이 부분에서 요청을 읽고 응답을 보낼 때 블러킹 됨.  
in = new DataInputStream(new BufferedInputStream(s.getInputStream()));  
out = new DataOutputStream(new BufferedOutputStream(s.getOutputStream()));  
}  
...  
}

요청, 응답 부분에서 블러킹이 되므로 아주 빈번하게 블러킹이 발생하게 된다. 그래서 아주 비효율적인(느린) 서버의 구성이 되는 것을 알 수 있을 것이다. 앞의 코드에서 Stream에서 Stream으로 데이터를 전달하는 과정에서 데이터 복사가 이뤄지고, 이 복사가 다 끝날 때까지 블러킹이 된다. 또한 효율을 높이기 위해 s.getInputStream()으로 읽어온 데이터를 BufferedInputStream 버퍼에 넣고 이것을 다시 DataInputStream으로 전달했지만, Stream에서 Stream으로 데이터를 복사해서 전달하므로 너무 많은 가비지(garbage, 소멸 대상 데이터)가 생성된다. Stream에서 전달되는 데이터는 대부분 특정 목적으로 한번 사용되고 곧바로 가비지 컬렉션(Garbage Collection) 대상이 된다. 예를 들어 채팅 서버에서 클라이언트가 보낸 대화 메시지(String)는 서버에서 대화방 안의 클라이언트들에게 브로드캐스트한 후 더 이상 쓸모없는 데이터가 되기 때문에 곧바로 가비지 컬렉션 대상이 되는 것이다. 따라서 Stream으로 데이터를 복사해서 전달한다는 것은 똑같은 데이터에 대해 한 개의 가비지가 더 생성되는 것이다. 가비지가 많이 생성된다는 것은 가비지 컬렉터가 그만큼 자주 호출된다는 것을 의미하고, 이것은 퍼포먼스에 악영향을 주게 되는 요인이 된다. 결론적으로 기존의 io에서는 효율을 위해 더 많은 가비지 생성을 감수하고 버퍼링을 하느냐 그렇게 하지 않느냐는 각각의 장단점을 갖는 양날의 칼과도 같았다. 하지만 대개의 경우 버퍼링을 사용한다. 그 이유는 서버 프로그래밍에서 최우선적으로 고려할 사항이 바로 효율(속도)이기 때문이다.  
  
◆ **가비지 컬렉션** : 프로그램은 프로그램을 진행하면서 데이터들을 저장하는 것이 필요하다. 데이터들은 모두 메모리에 저장이 되는데, 저장할 데이터가 있으면 메모리의 일정 공간을 할당받아서 사용하는 것이다. 그런데 더 이상 사용되지 않는 데이터에게 메모리를 계속 할당해 주는 것은 메모리를 낭비하는 것이므로, 그 데이터가 사용하던 메모리를 회수하는 것이 필요하다. 이렇게 사용되지 않는 메모리에 대한 회수를 가비지 컬렉션이라고 하고 이것을 수행하는 것을 가비지 컬렉터라고 한다. 그러나 어떤 데이터가 사용되지 않는다고 해서 곧바로 가비지 컬렉터가 실행되어 메모리를 회수하는 것은 아니다. 우선은 쓸모없는 데이터가 가비지 컬렉션의 대상으로 지정되고 프로그램이 진행되기 위해 메모리가 더 필요하게 되면, 그때 가서 메모리를 회수하는 것이다.  
  
◆ **가비지 컬렉터 이슈(Garbage Collector issue)** : 자바에서는 메모리를 JVM이 알아서 수거한다. 하지만 그렇게 메모리를 수거하기 위해서 가비지 컬렉터를 이용하는데 이것은 상당히 속도가 느린 작업이다. 그런데 앞의 코드처럼 Stream간 빈번하게 데이터를 복사하고 사용 후 바로 소멸하는 것으로 인해 아주 많은 양의 데이터가 가비지 컬렉션의 대상이 된다. 이것은 곧 가비지 컬렉터가 빈번하게 호출될 수 있다는 것을 의미하고 이것은 결국 성능 저하를 가져오게 된다. 가비지를 전혀 만들지 않는 서버를 만드는 것이 가장 이상적이겠지만 그건 어디까지나 이상일 뿐이고 우리는 최대한 가비지가 적게 만들어지도록 하는 것이 고성능 서버를 만드는 최선의 방법이다. 즉, 자바 서버 프로그래밍에서 고효율의 서버를 만들기 위해 가장 세심한 주의를 기울여야하는 부분이 가비지 관리인 것이다.  
  
앞에서 살펴본 바와 같이 자바에서 기존의 io와 소켓은 너무 많은 블러킹이 존재하고, 구조적으로 많은 가비지가 생성될 수밖에 없다. 이것은 효율이 최우선시 되는 서버 프로그램에서는 그리 반갑지 못한 조건이다.   
  
**새로운 대안, 비동기식 서버의 등장**앞서 설명한 동기식 서버의 문제점들을 해결하기 위해 J2SDK1.4에서 nio가 나왔다. nio에서는 accept()와 클라이언트의 요청/응답에 대해 블러킹이 없다. 이것을 가능하게 한 것은 채널(Channel) 인터페이스를 구현하는 SelectableChannel이라는 새로운 클래스를 non-blocking으로 설정함으로써 accept()에 대한 블러킹을 피할 수 있도록 했고, Buffer라는 새로운 클래스의 도입으로 입출력 작업에서 블러킹을 피하고 기존 io에서의 Stream간의 데이터 복사에 의한 가비지 생성을 예방함으로써 효율적인 버퍼링이 가능해졌다. 또한 채널과 버퍼(direct buffer)는 네이티브 접근을 함으로써 기존의 동기식 서버보다 훨씬 나은 성능을 갖출 수 있다. 그럼 nio의 핵심 구성요소들을 살펴보자.  
  
ps. 좀 더 정확하게 말하자면 nio 는 완벽한 비동기식이 아니다. 그것보다는 난블러킹(non-blocking)이라는 표현이 더 적절할 것이다. 완벽한 비동기식 소켓 통신의 지원은 1.5 타이거에서 나온다는 얘기가 있으니 기대해 보자.  
  
**효율적인 io 작업을 위해 탄생한 Buffer**자바에서 고성능 서버를 만들기 위해 반드시 고려해야할 것 중 하나가 바로 가비지 컬렉션이다. 즉, 가비지를 최대한 적게 만들어서 최대한 가비지 컬렉터 호출을 최소화시켜야 한다. 그러나 기존 io에서는 너무 많은 가비지가 생성되는 구조적 문제점이 있었다. 이런 문제점의 해결방안으로 nio에서는 Buffer라는 새로운 클래스가 추가됐다. 그럼 이제부터 Buffer에 대해 자세하게 살펴보겠다.   
  
Buffer 는 그 클래스 타입에 따라 하나의 데이터 타입만을 저장할 수 있는 선형의 순차적 dataset이다. nio 버퍼 군의 최상위 abstract 클래스인 Buffer를 중심으로 아래 표와 같은 다양한 종류의 클래스들이 존재한다.   
  
<표 1> java.nio의 Buffer 클래스들

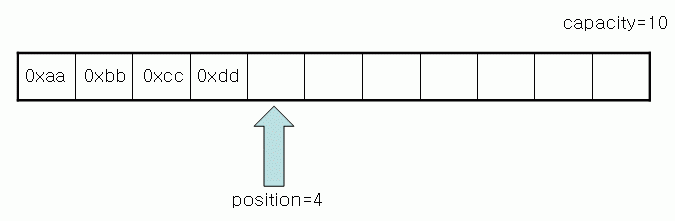
|  |  |
| --- | --- |
| **구분** | **내용** |
| java.nio.Buffer | abstract 클래스. 모든 Buffer의 super 클래스. |
| java.nio.ByteBuffer | byte 기반 버퍼. direct와 nondirect 두 가지 방식으로 생성할 수 있음. ReadableByteChannel로부터 읽을 수 있고 WritableByteChannel로 쓸 수 있음. |
| java.nio.MappedByteBuffer | byte 기반 버퍼. 항상 direct 로 생성됨. 파일의 메모리맵 영역을 내용으로 하는 버퍼. ByteBuffer의 서브 클래스임. |
| java.nio.CharBuffer | char 기반 버퍼. 채널에 쓸 수 없음. |
| java.nio.DoubleBuffer | double 기반 버퍼. 채널에 쓸 수 없음. |
| java.nio.FloatBuffer | float 기반 버퍼. direct와 nondirect 두 가지 방식으로 생성할 수 있음. |
| java.nio.IntBuffer | int 기반 버퍼. direct와 nondirect 두 가지 방식으로 생성할 수 있음. |
| java.nio.LongBuffer | long 기반 버퍼. direct와 nondirect 두 가지 방식으로 생성할 수 있음. |
| java.nio.ShortBuffer | short 기반 버퍼. direct와 nondirect 두 가지 방식으로 생성할 수 있음. |

Buffer는 세 가지 요소로 구성되는데 바로 capacity, position, limit이다.   
  
◆ **capacity** : 버퍼를 만들 때 사용하는 버퍼의 용량(크기)이다. 초기에 기입된 값이 고정되므로 적절한 용량으로 버퍼를 생성해야 한다.   
◆ **position** : 버퍼의 현재 위치가 어디인지를 가리키는 인덱스이다. position이 가질 수 있는 값은 limit 이하의 값이다.  
◆ **limit**  : 해당 버퍼에 더 이상 읽거나 쓸 수 없는 다음 요소의 인덱스이다. 그러므로 limit와 position의 차는 버퍼의 ‘남은 값’을 의미한다.   
  
abstract 클래스인 Buffer를 상속하는 서브클래스들은 공통적으로 allocate(int capacity) 또는 allocateDirect(int capacity)를 통해서 버퍼를 생성한다. 또한 이미 존재하는 배열을 wrap하는 방식(예 : ByteBuffer.wrap(byte[] buffer))으로 버퍼를 생성할 수도 있다. 특별한 경우로는 MappedByteBuffer의 경우 FileChannel.map(int mode, long position, int size)로 버퍼를 생성할 수 있다.   
allocateDirect(int capacity)로 생성하는 direct buffer는 메모리 블럭에 연속적으로 할당되고 버퍼의 데이터를 읽거나 쓰기 위해 네이티브 접근 메쏘드들을 사용하게 된다. direct buffer는 nondirect buffer에 비해 버퍼를 생성/해제 하는데 더 많은 비용이 들지만 속도가 더 빠른 특징이 있다. 따라서 direct buffer는 큰 용량의 버퍼를 오랜 시간동안 유지해야 하고 빠른 처리를 원할 경우에 생성하는 것이 바람직하다. 바로 우리가 만들 채팅 서버와 같은 서버 프로그램에서 ByteBufferPool(12월호에서 소개됨)을 만들어서 사용하는 것이 적절한 예가 될 것이다. allocate(int capacity)로 생성하는 nondirect buffer는 자바의 배열 접근자들(get/put 메소드들)을 통해 버퍼의 데이터에 접근한다.   
이 두 가지 타입의 버퍼 차이는 단지 앞서 설명한대로 메모리에 할당되는 것과 읽고 쓰기 위해 네이티브 접근을 하는가 아닌가 정도이다. 그럼 가장 흔히 쓰일 ByteBuffer를 예제로 해서 사용법을 알아보자.

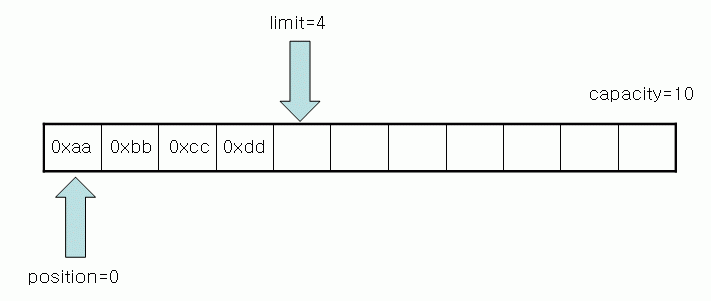
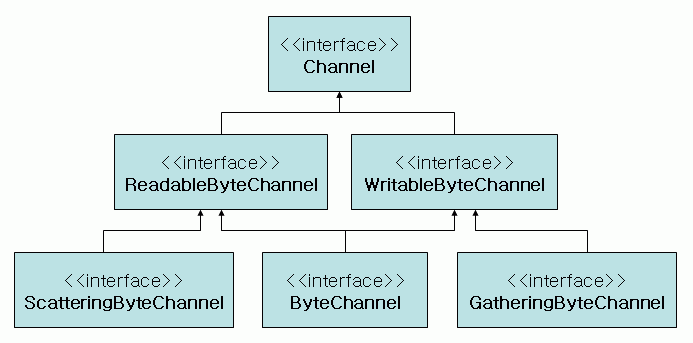
ByteBuffer buffer = ByteBuffer.allocateDirect(10);

이와 같이 direct 버퍼를 생성하면 <그림 2>와 같이 position=0, capacity=10, limit=10인 새로운 버퍼가 생성된다. 새로운 버퍼를 생성하면 capacity와 limit가 같게 된다.   
  
  
<그림 2> capacity가 10인 Direct 버퍼 생성  
  
  
버퍼의 position은 읽거나 쓰기 위한 다음 요소의 인덱스이다. 앞의 경우엔 버퍼를 새로 생성했으므로 position이 0번째 위치를 가리키게 된다. 또한 버퍼에 데이터를 추가하면 position은 limit 쪽으로 이동하게 된다. 그럼 버퍼에 데이터를 추가시켜 보자.

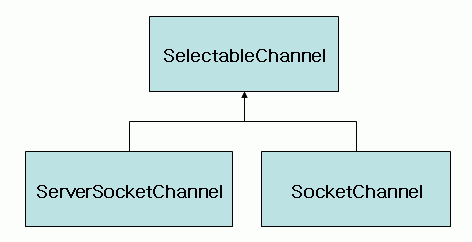
buffer.put( (byte)0xaa );  
buffer.putShort( (short)0xbbcc );  
buffer.put( (byte)0xdd );

이 처럼 버퍼에 데이터를 추가하면 1byte씩 차례대로 저장되기 때문에 <그림 3>과 같이 4byte의 공간에 순서대로 저장되고 position=4가 된다. putShort()로 저장한 부분을 주의해서 봐야 한다. short는 16bit이기 때문에 1byte 씩 순서대로 2byte가 저장된다.   
  
  
<그림 3> 버퍼에 put() 으로 데이터를 추가  
  
  
버퍼에 데이터를 넣을 때 주의할 것은 버퍼의 limit를 넘어서 버퍼에 데이터를 쓰려면 BufferOverflowException이 발생한다는 것이다. 비슷하게 버퍼의 limit를 넘어서 데이터를 읽으려면 BufferUnderflowException이 발생한다. 이런 경우 버퍼에 채워진 데이터를 읽으면 되며, 채널에 쓰기 위해서는 반드시 버퍼를 플립(flip)시켜야 한다. 

buffer.flip();

버퍼를 플립시키면 <그림 4>에서처럼 버퍼의 position을 맨 앞으로(0으로) 이동하고 플립하기 전의 position이 limit로 설정된다. 그러나 앞서 설명했듯이 capacity는 변하지 않고 고정되어 있다. 버퍼를 플립시키고 나면 버퍼를 읽을 수 있는 상태가 된다.  
  
  
<그림 4> 버퍼를 flip()시킨 경우  
  
  
<그림 4>에서 버퍼를 플립시키고 나서 position과 limit가 변한 것을 볼 수 있다(나머지 버퍼의 사용도 이것과 크게 다르지 않으니 API 문서를 통해 사용법을 익히기 바란다). CharBuffer와 java.nio.charset 패키지의 Charset 클래스로 인코딩하는 것은 지면 관계상 설명하지 않았지만 꼭 신경써 봐두기 바란다.  
  
**비동기식 소켓 통신을 위한 채널**채널은 하드웨어 디바이스, 파일, 네트워크 소켓 등이 상호간에 읽기나 쓰기 등의 입출력 작업을 한 개 이상 수행할 수 있는 추상화된 계층이다. 최상위의 java.nio.channel.Channel은 단지 Open 또는 Close 상태만을 나타내는데 만약 채널이 close된 상태에서 입출력 작업을 실행하려면 ClosedChannelException이 발생된다. 그리고 채널은 멀티 쓰레드 접근에 대해 안전하다. 채널의 장점은 네이티브 접근을 통해 속도가 빠르다는 것과 어떤 쓰레드가 특정 채널에 대한 작업 중 블럭됐을 때 다른 쓰레드가 그 채널을 종료시킬 수 있다는 것이다. 다른 쓰레드가 채널을 종료시키면 블록된 쓰레드는 채널이 종료할 때 관계된 Exception과 함께 깨어나게 된다. 즉, 어떤 작업 중에 블럭됐을 때 효과적인 대처가 가능한 것이다. <그림 5>는 채널의 계층도이다.   
  
  
<그림 5> 채널 계층도  
  
  
<그림 5>에서 볼 수 있듯이 ScatteringByteChannel은 데이터를 읽는데 사용되고 GatheringByteChannel은 write하는데 사용되는 채널이다. 예전부터 지금까지 오랫동안 유닉스와 윈도우 NT에서 고성능 I/O 작업을 위해 vectored IO로 알려진 Scatter/Gather를 사용해왔다. SCSI 컨트롤러 또한 전체적인 성능 향상을 위해서 Scatter/Gather를 사용한다. 자바에서 채널도 Scatter/Gather를 사용하는데, 이것은 채널의 io 작업에 대해 성능 향상을 위해서 Scatter/Gather 오퍼레이션들을 OS로 빠르게 전달함으로써 네이티브 OS 수준의 처리를 하는 것이다. 

// Gather 예.  
ByteBuffer header = ByteBuffer.allocateDirect(32);  
ByteBuffer body = ByteBuffer.allocateDirect(100)  
ByteBuffer attach = ByteBuffer.allocateDirect(100);  
ByteBuffer[] gatherBuffers = { header, body, attach };  
gatherChannel.write(gatherBuffers);

nio 패키지를 이용해서 서버를 만들 때 가장 먼저 이해를 요구하는 부분은 SelectableChannel, Selector, SelectionKey 이 세 개의 클래스 간의 협력관계이다. 이 세 개의 클래스가 서로 어떤 식으로 연관되어 어떻게 상호 작용하는지를 이해하면 나머지 기능에 대한 것들은 API를 통해 쉽게 사용할 수 있다. 그럼 먼저 SelectableChannel에 대해 알아보자.  
  
소켓 계열 채널의 슈퍼 클래스, SelectableChannel  
  
  
<그림 6> SelectableChannel 계층도  
  
  
<그림 6>에서 나타나듯이 SelectableChannel은 ServerSocketChannel과 SocketChannel의 슈퍼 클래스이다. 뿐만 아니라 서버 프로그래밍에서 소켓 통신을 위해 사용할 다른 모든 채널들이 상속하는 클래스이다. 이해를 쉽게 하기 위해 SelectableChannel의 API를 살펴보자. 

public abstract class SelectableChannel extends AbstractInterruptibleChannel   
implements Channel {  
  
public abstract SelectableChannel configureBlocking(boolean block);  
  
public abstract SelectionKey register(Selector sel, int ops)  
throws ClosedChannelException;  
public abstract SelectionKey register(Selector sel, int ops, Object att)  
throws ClosedChannelException;   
...  
}

SelectableChannel 클래스에는 크게 두 가지의 핵심적인 역할을 하는 메쏘드가 있다. 그 중 하나는 서버의 블러킹 여부를 결정하는 configureBlocking 메쏘드이고, 다른 하나는 주어진 파라미터의 Selector에 자신이 어떤 오퍼레이션(ops)을 할지 등록하는 register 메쏘드이다. 이때 주의할 점은 반드시 register 메쏘드로 Selector에 등록하기 전에 먼저 configureBlocking으로 서버의 블러킹 여부를 정해줘야 한다. SelectableChannel은 디폴트로 블러킹 모드(true)가 지정 돼 있다. register 메쏘드에 두 번째 파라미터로 들어가는 ops는 <표 2>에서처럼 네 가지가 있다.   
  
<표 2> operation들과 그 역할

|  |  |
| --- | --- |
| **구분** | **내용** |
| OP\_ACCEPT | 클라이언트가 ServerSocketChannel에 접속을 시도했을 때 발생 |
| OP\_CONNECT | 서버가 클라이언트의 접속을 허락했을 때 발생 |
| OP\_READ | 서버가 클라이언트의 요청을 read할 수 있을 때 발생 |
| OP\_WRITE | 서버가 클라이언트에게 응답을 write할 수 있을 때 발생 |

SelectableChannel을 상속한 ServerSocketChannel 등은 register 메쏘드로 자신을 Selector에 등록해야하는데 이때 모든 ops를 사용해서 등록할 수는 없다. 각자 등록할 수 있는 ops가 <표 3>에서처럼 제한돼 있기 때문이다.  
  
<표 3> SelectableChannel 들의 등록 가능한 ops

|  |  |
| --- | --- |
| **구분** | **내용** |
| ServerSocketChannel | OP\_ACCEPT |
| SocketChannel | OP\_CONNECT, OP\_READ, OP\_WRITE |
| DatagramChannel | OP\_READ, OP\_WRITE |
| Pipe.SourceChannel | OP\_READ |
| Pipe.SinkChannel | OP\_WRITE |

register 메쏘는 Selector와 ops 외에 특정 객체를 같이 등록할 수 있는 register(Selector sel, int ops, Object att)도 제공해주는데, 이 메쏘드를 이용하면 특정 클라이언트의 세션, 비즈니스 로직 또는 다른 채널 등을 해당 채널과 연관시켜서 간편하게 사용할 수 있다. 그러나 이렇게 함께 등록된 객체는 해당 채널이 종료되더라도 제거되지 않으므로 반드시 채널 종료 시 명시적으로 직접 제거해줘야만 메모리 낭비를 막을 수 있다. 그리고 SelectableChannel은 여러 Selector에 등록할 수 있지만 하나의 Selector에는 특정 SelectableChannel이 한 개만 등록된다. 또한 SelectableChannel을 Selector에 등록할 때 SelectableChannel이 Selector에 직접 저장되는 것은 아니다. register 메쏘드의 리턴 값을 보고 몇몇 독자분은 짐작했겠지만 Selector에 등록할 때 SelectionKey라는 객체가 생성되고 이 객체를 Selector 내부의 Set에 저장해서 관리한다.   
  
ServerSocket에 대응하는 ServerSocketChannel  
ServerSocketChannel 은 ServerSocket과 같은 역할을 한다. 서버에 연결(connection)하기 위한 요청을 받아서 요청한 클라이언트와 서버의 연결을 맺어주는 것이다. 이 ServerSocketChannel을 만들기 위해서는 ServerSocket과 같이 new를 사용해서 직접적으로 만들 수 없고 open() 메쏘드를 사용해서 만든다. 이렇게 새로 만든 ServerSocketChannel은 오픈 상태지만 bind(ip, port에 연결)되지는 않는다. ServerSocketChannel을 bind시키기 위해서는 ServerSocket의 bind()를 사용해야 한다. 생성된 ServerSocketChannel에서 ServerSocket을 ServerSocketChannel의 socket()를 호출해서 얻은 후에 이 ServerSocket의 bind() 메쏘드를 호출해서 bind 시키는 것이다. 만약 ServerSocket의 옵션을 설정하고자 한다면 앞에서 처럼 얻은 ServerSocket을 이용해서 이전에 해왔던 것과 같은 방법으로 설정해줄 수 있다.

ServerSocketChannel ssc = ServerSocketChannel.open();  
ServerSocket ss = ssc.socket();  
// 필요한 경우 ss를 이용해서 ServerSocket 옵션 설정.  
InetAddress ia = InetAddress.getLocalHost();  
InetSocketAddress isa = new InetSocketAddress(ia, PORT);  
ss.bind(isa);  
  
ServerSocketChannel에서 클라이언트의 연결 요청을 받아 서버와 연결을 맺어주기 위해서는 ServerSocket과 같이 accept() 메쏘드를 사용한다. 하지만 return 값은 소켓이 아니라 SocketChannel이다.  
  
SocketChannel sc = ssc.accept();

소켓에 대응하는 SocketChannel  
SocketChannel은 소켓과 비슷하게 클라이언트와 직접적이고 실질적인 통신을 한다. 하지만 소켓과 달리 데이터를 주고받을 때 read(ByteBuffer bb)/write(ByteBuffer bb) 메쏘드를 사용한다. SocketChannel을 생성하는 것은 다음과 같이 세 가지 방법이 있다.  
  
? 연결되지 않은 SocketChannel이 만들어지는 경우

SocketChannel sc = SocketChannel.open();

? 주어진 SocketAddress로 접속하는 SocketChannel이 만들어지는 경우

SocketChannel sc = SocketChannel.open(SocketAddress remote);

? 서버에 접속한 클라이언트와 연결된 SocketChnnel이 만들어지는 경우

SocketChannel sc = ServerSocketChannel.accept();

또한 ServerSocketChannel과 마찬가지로 socket() 메쏘드로 소켓을 얻어올 수 있고 이렇게 얻은 소켓으로 옵션을 설정할 수 있다. 

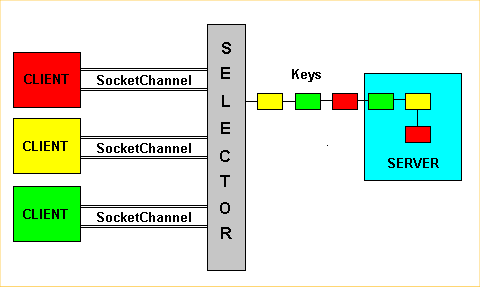
Socket s = sc.socket();

// 필요한 경우 s를 이용해서 Socket 옵션 설정.  
  
**마법캡슐 SelectionKey**SelectionKey는 특정 채널과 selector 사이의 등록 관계를 캡슐화한다. 즉, 어떤 SelectableChannel이 특정 Selector에 register()로 등록하면 그들의 연관 관계를 표현하는 SelectionKey 객체가 생성되고, 이 객체는 Selector와 애플리케이션이 각각의 역할을 수행하는데 사용된다. 이렇게 생성된 객체는 두 가지의 핵심적인 역할을 하는 set을 갖고 있다. 첫 번째는 SelectableChannel이 register()로 Selector에 등록한 오퍼레이션들(ops)을 저장하는 interest set이다. 두 번째는 SelectableChannel에서 이벤트가 발생하면 그 이벤트들을 저장하는 ready set이다. 그럼 API를 살펴보자. 

public abstract class SelectionKey {  
public static final int OP\_READ = 1 << 0; // => 1  
public static final int OP\_WRITE = 1 << 2; // => 4  
public static final int OP\_CONNECT = 1 << 3; //=> 8  
public static final int OP\_ACCEPT = 1 << 4; // => 16  
  
public abstract int readyOps();  
  
public boolean isAcceptable();  
public boolean isConnectable();  
public boolean isReadable();  
public boolean isWritable();  
  
public abstract void cancel();  
  
public abstract SelectableChannel channel();  
public abstract Selector selector();  
}

SelectionKey 에는 API에서 볼 수 있듯이 SelectableChannel이 등록할 수 있는 네 개의 오퍼레이션이 static 필드로 정의돼 있다. 또 readyOps()으로 이 SelectionKey 객체가 캡슐화하고 있는 SelectableChannel이 이벤트가 발생됐는지를 체크할 수 있다. 이것은 다음과 같이 두 가지 방식으로 가능하다. 다음의 부분은 Accept 오퍼레이션을 체크하는 것인데 보다시피 다른 오퍼레이션들도 어떻게 이뤄질지 쉽게 짐작할 수 있을 것이다.

if (key.isAcceptable())  
or  
if ((key.readyOps() & SelectionKey.OP\_ACCEPT) != 0)  
  
또한 channel() 메쏘드와 selector() 메쏘드로 SelectableChannel과 Selector의 인스턴스를 리턴 값으로 받을 수 있다.  
  
SocketChannel sc = key.channel();  
Selector selector = key.selector();

cancel() 메쏘드는 해당 SelectionKey가 캡슐화하고 있는 Selector와 SelectableChannel의 관계를 종료시킨다. 즉, SelectableChannel가 Selector에서 등록해제 되는 것이다. 그리고 그 SelectionKey는 유효하지 않게 된다. 만약 어떤 SelectableChannel이 close되면 이 SelectableChannel이 등록한 모든 Selector에서 즉시 해당 SelectionKey가 유효하지 않게 되고 적절한 시점에서 등록해제 된다. 여기서 주의해야 할 것은 cancel() 메쏘드를 호출하거나 SelectableChannel이 close됐다고 그 즉시 Selector에서 등록해제 되는 것이 아니고 Selector의 cancelled key set에 넣어지게 된다. 하지만 그 SelectionKey는 즉시 유효하지 않게 된다. 이렇게 유효하지 않게 된 SelectionKey의 어떤 메쏘드를 호출하면 CancelledKeyException이 발생한다.   
  
**이벤트 중계자 Selector**1996년에 출판된 POSA2(Pattern Oriented Software Architecture, Volume2)에 Reactor 패턴이 소개됐다. Reactor 패턴은 이벤트 중심 애플리케이션이 하나 이상의 클라이언트로부터 한 애플리케이션으로 동시에 전달되는 서비스 요청들을 나눠 각 요청에 상응하는 서비스 제공자에게로 구별해서 보내준다. 좀 더 자세하게 설명하면 클라이언트들의 모든 요청을 우선 앞단의 큐에 저장하고 큐를 모니터링하는 쓰레드에게 이벤트를 보낸다. 그러면 큐를 모니터링하는 쓰레드는 큐에 저장된 요청의 방향을 분석해서 적절한 프로세스로직으로 보내주어 해당 요청이 처리되도록 해주는 것이다.  
  
nio에서 비동기식 서버 구현의 밑바탕이 되는 것이 바로 Reactor 패턴이다. Selector는 바로 Reactor 역할을 한다. 즉, 여러 SelectableChannel을(더 정확히 말하자면 그 채널과의 관계를 표현하는 SelectionKey를) 자신에게 등록하게 하고 등록된 SelectableChannel의 이벤트 요청들을 나눠 적절한 서비스 제공자에게 보내어 처리하는 것이다.   
Selector는 내부적으로 세 개의 set을 관리한다. 첫 번째는 자신에게 등록한 SelectableChannel의 SelectionKey를 저장하는 Registered key set이다. 두 번째는 selection 메쏘드(select(), select(long timeout), selectNow()) 중 하나를 호출했을 때 Registered key set에 등록된 SelectableChannel 중에서 이벤트가 발생한 것들의 SelectionKey를 저장하는 Selected key set이다. 세 번째는 Selector에서 등록해제하기 위해 SelectionKey의 cancel() 메쏘드를 호출하거나 SelectableChannel의 close() 메쏘드를 호출한 SelectionKey를 저장하는 Cancelled key set이다.   
Selector 역시 SelectableChannel의 서브 클래스처럼 open() 메쏘드를 사용해 생성하고 close() 메쏘드를 사용해 종료한다. 그리고 wakeup() 메쏘드는 select()나 select(long timeout) 메쏘드의 호출로 블럭된 쓰레드를 깨우는데 사용된다.  
Selector도 채널과 Buffer 클래스들과 마찬가지로 네이티브 메쏘드를 사용해서 빠른 처리를 한다.  
  
**nio 서버의 전체적인 처리 흐름**<그림 7> 은 비동기식 서버의 전체 구조를 보여준다.  
  
  
<그림 7> 비동기식 서버 아키텍처  
  
  
SelectableChannel은 자신이 발생시키고 싶은 오퍼레이션과 함께 Selector에 등록한다. 그러면 Selector의 Registered key set에는 해당 SelectableChannel의 정보를 캡슐화하고 있는 SelectionKey가 저장되고, 만약 이렇게 등록된 SelectableChannel에서 어떤 이벤트가 발생하면 곧바로 애플리케이션(Selector를 이용해 이벤트를 처리하는 서버)에 알리지 않고 발생한 이벤트들을 해당 SelectionKey의 ready set에 기록해둔다.   
그런 후에 애플리케이션이 Selector의 selection 메쏘드 중 하나를 호출했을 때 먼저 Cancelled key set 안의 SelectionKey를 등록해제하고 그 set을 비우게 된다. 그 다음 Selector는 내부의 Registered key set에 저장된 SelectionKey를 검사한다. 이때 SelectionKey의 ready set이 empty가 아닌(이벤트가 발생한) SelectionKey들을 selected key set 에 넣는다. 다음으로 애플리케이션이 Selector의 selectedKeys() 메쏘드를 호출해서 Selector에 저장된 selected key set을 가져오고 그 안에 저장된 각각의 SelectionKey의 오퍼레이션 타입에 따라 처리한다. 다음의 소스는 nioServer의 핵심적인 처리 부분이다(전체 소스는 ‘이달의 디스켓’에 첨부했다).

while (true) {  
int n = selector.select();   
Iterator it = selector.selectedKeys().iterator();   
while (it.hasNext()) {  
SelectionKey key = (SelectionKey) it.next();   
if (key.isAcceptable()) {  
ServerSocketChannel server = (ServerSocketChannel) key.channel();  
SocketChannel sc = server.accept();   
boolean isRegist = registerChannel(selector, sc, SelectionKey.OP\_READ);  
if (isRegist)   
room.addElement(sc);  
} else if (key.isReadable()) {  
service(key);  
}  
it.remove();  
}  
}

**nio로 비동기식 서버를 만들 때 고려할 점**

|  |
| --- |
| ? 프로세스 로직을 ThreadPool로 처리하게 만들어라. 동시에 많은 처리를 요하는 프로그램에서 쓰레드는 필수적으로 사용된다. 100명의 관객이 영화 티켓을 사기 위해 하나의 창구를 이용하는 것과 10개의 창구를 이용하는 것은 분명 많은 차이가 있다. 결국 많은 양의 동시 처리를 요구하는 서버 프로그램에서도 성능 향상을 위해 반드시 기본적으로 구현해서 사용해야 하는 것이 ThreadPool이다.  ? direct ByteBuffer를 사용해라. 채널에 direct 버퍼로 write한 경우에는 네이티브 호출을 위해 즉시 전달되지만, nondirect 버퍼로 write를 하면 채널은 새로운 direct 버퍼를 만들어서 nondirect 버퍼에 있던 데이터를 새로 만든 direct 버퍼로 복사해서 전달한다. 이것은 데이터 복사에 따른 시간 지연과 가비지 생성, 새로운 direct 버퍼의 생성으로 인한 시간 지연 및 메모리 낭비를 의미한다. 그러므로 채널을 이용할 때는 반드시 direct 버퍼를 사용하고 direct 버퍼의 할당 해제는 시간이 걸리는 작업이므로 효율을 위해 direct ByteBufferPool을 만들어서 사용하는 것이 바람직하다.   ? SelectorPool을 만들어서 사용해라. 윈도우 OS의 경우 하나의 Selector가 63개까지의 SelectableChannel을 등록할 수 있다. 다른 OS의 경우 Integer.MAX\_VALUE(2147483647)까지 등록이 가능하다. 하지만 효율을 위해 하나의 쓰레드가 하나의 Selector를 관리하는 SelectorPool을 만들어서 여러 개의 Selector에 SelectableChannel들을 균등하게 등록해서 멀티 쓰레드 처리를 하도록 사용하는 것이 바람직하다.   ? Selector의 내부 key set에 대한 멀티 쓰레드 접근을 주의하라.  Selector는 쓰레드에 대해 안전하지만 Selector의 key set은 그렇지 못하다. Selector의 내부 key set은 private 접근자를 갖고 있는 내부 필드를 직접적으로 참조하기 때문에 만약 어떤 쓰레드가 key set을 처리하는 도중에 다른 쓰레드가 그 key set을 변경하면 예상하지 못할 결과가 발생할 수도 있으므로 주의해야 한다. |

**nio에 친숙해지길 바라며…**앞서도 계속 강조했지만 서버 프로그램에서 최우선 고려 대상은 바로 효율이다. 그런데 지면관계상 직접 소스를 설명하지 못했으며, 필자가 예제로 만들어 첨부한 서버에서는 몇 가지 개선할 점이 있다. 비록 우리가 만든 서버가 비동기식 서버이지만 어디까지나 클라이언트들의 서버에 대한 Connection과 C/S간 데이터 전송에 대한 블러킹이 없었을 뿐이고 그 처리 로직은 순차적 처리를 하는 방식이었다. 그러므로 첫 번째 개선점은 프로세스 로직을 쓰레드(효율을 위해 ThreadPool을 만들어 사용)를 사용해서 동시에 병렬처리를 함으로써 효율을 높여야 하는 것이다.   
두 번째는 ByteBufferPool을 만들어 사용함으로써 direct ByteBufferPool의 할당?해제하는 시간을 절약하는 것이다. 여기서 좀 더 나아가 OS가 하드디스크를 가상 메모리로 설정해서 사용하듯이 FileChannel을 이용해서 메모리 부족 시 파일을 메모리로 사용하도록 만들 수 있다. 다음 호에서는 앞서 우리가 만든 채팅 서버에 위에서 언급한 ThreadPool과 ByteBufferPool을 추가시켜 효율 극대화와 메모리 부족에 의한 서버 다운을 최대한 예방할 수 있는 견고한 서버를 만들어보자.  
  
마소에 기사를 쓰는 다른 필자들의 글을 읽을 때 많은 분이 제한된 지면 때문에 하고 싶은 말을 제대로 쓰지 못했다며 아쉬워하는걸 봐왔다. 지금 이렇게 기사를 쓰면서 나 또한 그 분들과 같은 생각을 하게 된다. 제한된 지면관계상 설명하지 못한 부분과 좀 더 자세하게 설명하고 싶지만 그렇지 못한 부분들이 있어서 많은 아쉬움이 남는다. 하지만 나름대로 최선을 다해 쉽게 설명하려고 노력했으므로 많은 독자들이 본 기사를 통해 nio에 좀 더 친숙해진다면 그것만으로도 큰 기쁨이 될 것이다. 다음 호에서 더 좋은 글로 찾아뵐 것을 약속드리며 이만 글을 마무리하겠다.  
  
정리 : 조규형 ([jokyu@korea.cnet.com](http://www.javacafe.or.kr/lecture/cafeLecture/network/nio/jokyu@korea.cnet.com) )  
  
**참고자료**1. Java Nio by Ron Hitchens, 2002 O'reilly  
2. <http://www.onjava.com/pub/a/onjava/2002/10/02/javanio.html>  
  
3. <http://www.onjava.com/pub/a/onjava/2002/09/04/nio.html>  
  
4. <http://java.sun.com/j2se/1.4/docs/guide/nio/index.html>  
  
5. <http://developer.java.sun.com/developer/technicalArticles/releases/nio>  
  
6. <http://www.javaworld.com/javaworld/jw-09-2001/jw-0907-merlin.html>  
  
7. [소스 자료](http://www.javacafe.or.kr/lecture/cafeLecture/network/nio/src.zip)