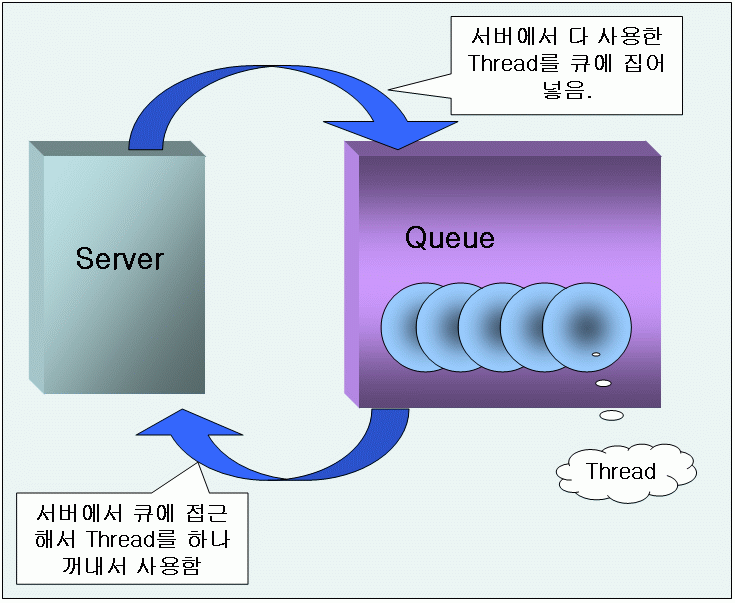
**주의: 본문서는 마이크로소프트웨어 2002~11월,12월호에 송지훈님이 기고한 글로써 본인의 동의없이 무단 배포하는 것을 금지함. 만약 글을 다른 곳에 포스팅하려 할 경우 반드시 강좌의 URL 링크를 사용해야함.   
  
  
ByteBufferPool과 ThreadPool을 추가해 성능 업그레이드하기**

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

**서버의 성능 향상을 위해 Pooling 기법을 도입하고 Command 패턴과 런타임 동적 로딩을 이용해 멈추지 않는 서버 만들기**  
이번 호에서는 서버의 성능 향상을 위해 고려해야 할 점들을 살펴보고 그 개선 방법들에 대해 언급할 것이다. 그리고 그 중 일부를 지난 시간에 만들었던 간단한 채팅 서버에 적용시켜 볼 것이다. 그럼 이 기사를 통해 상용 서버 프로그램 수준에 한걸음 더 다가서 보자.  
  
**ThreadPool 과 ByteBufferPool 로 서버에 날개를 달자**모바일 프로그램이 제한된 시스템 사양에서 만족할 만한 속도를 얻기 위해 코드를 최적화 시키듯이 네트워크 프로그래밍도 효율을 중시하기 때문에 코딩시 고려해야할 점들이 많다. 효율과 확장성, 안정성을 고려해야하는 네트워크 프로그래밍은 다른 프로그램에 비해 기술 집약적인 형태의 성능향상을 고려한 코드들로 이루어져야 한다. 이밖에도 네트워크라는 곳이 주무대이다 보니 예기치 못한 다양한 상황에 대비해야하는 것도 프로그래머 입장에선 골치 아픈 일이다. 지면관계상 이 글을 통해 서버 프로그래밍에서 고려해야할 모든 것들을 다 언급하지는 못할 것이다. 하지만 전 시간에 언급했듯이 효율적인 쓰레드 운영을 위한 ThreadPool, 버퍼의 효율적인 운영과 파일을 메모리로 사용하기 위한 ByteBufferPool을 중심으로 설명해 나갈 것이다. 또 Command 패턴과 자바의 언어적 특징인 런타임 동적 로딩을 이용해 기능 확장시 서버를 재부팅 하지 않아도 되는 기법을 설명할 것이다. 이외에 기타 필요하다고 생각되는 부분들에 대해서도 추가적인 설명과 코드를 소개해 나갈 것이다. 그럼 우리가 지난 시간에 만들었던 서버를 업그레이드 시켜보자.  
  
**효율적인 Thread 운영-관리를 위한 ThreadPool**동시에 수많은 요청을 처리하기 위해선 무엇보다도 효율적(적은 메모리 사용, 빠른 처리속도) 으로 서버를 만드는 것이 가장 중요하다. 쓰레드를 필요할 때마다 그때그때 만들어서 사용하게 되면 쓰레드 자체를 생성하는 것도 꽤(서버 프로그램 입장에선) 시간이 걸리는 느린 작업이고 쓰레드라는 객체가 자주 생성-해제 되는 상황 때문에 가비지 컬렉터(Garbage Collector)가 빈번하게 호출될 수 있기 때문이다.   
  
그래서 이런 문제점을 해결하기 위한 방법으로 GoF 가 쓴 명서 Design Pattern 에 소개된 Object Pool 패턴을 이용할 것이다. 생성할 객체가 너무 시간이 오래 걸리거나 많은 경우 등으로 인한 문제점이 있을 때 <그림 1> 과 같이 그 객체를 큐(컬렉션 객체)에 넣어놓고 재사용하는 것이 이 패턴의 핵심 원리다. 패턴 이름을 보고 짐작할 수 있겠지만 Object 는 범용적인 형태이다. 객체지향 언어에서 최상위의 표현단위가 바로 Object 아닌가...  
따라서 우리는 “Thread“를 재사용하려고 하기 때문에 ObjectPool Pattern 을 구체적으로 적용한 ThreadPool 이라는 클래스를 만들어서 사용할 것이다. ObjectPool Pattern 의 다른 적용 예로는 jsp/servlet 으로 웹프로그래밍을 할때 DB 접속에 흔히 사용하는 ConnectionPool 이 있다.   
  
  
  
<그림 1> ThreadPool 도식도  
  
어떤 기술을 공부할 때 항상 생각해야 하는 것들 중 하나가 바로 그 기술의 장단점과 최적의 적용 가능 분야 등을 정확하게 파악하는 것이다. 그래야만 좀 더 그 기술에 대한 이해도도 높일 수 있고 그 기술이 필요한 곳에 최적화 시켜 사용할 수 있기 때문이다.   
그러므로 우선 ThreadPool을 만들기 전에 이 ThreadPool을 만들어 사용할 때 얻을 수 있는 장점이 구체적으로 무엇이 있는지 알아보자. 장점을 안다면 당연히 어느 곳에, 어떻게 쓰여야 할지도 쉽게 알 수 있을 것이다.  
  
첫 번째 이점으로는 쓰레드를 재사용함으로써 가비지 컬렉터의 호출을 좀 더 줄일 수 있다는 것이다. 쓰레드의 잦은 생성 소멸로 인해 생성될 가비지 컬렉션(Garbage Collection)의 대상을 줄임으로써 가비지 컬렉터의 호출을 줄이고 이로 인해 퍼포먼스에 악영향을 줄 수 있는 요소를 작게나마 사전에 예방한다는 것이다. 지난 회에서도 얘기했었지만 가비지 컬렉터로 메모리를 수거하는 것은 상당히 느린 작업이다. (1회 기사에서 설명했었다. 기억이 안난다면 다시 찾아보자)  
  
두 번째 이점은 쓰레드를 새로 생성하지 않고 이미 생성된 쓰레드를 가져다가 쓰기 때문에 쓰레드를 새로 생성하는 것에 비해 속도가 빠르다는 것이다. 메모리상에 존재하는 쓰레드를 그냥 가져오는 것이 당연히 새로운 쓰레드를 메모리에 할당해서 가져다 쓰는 것보다 빠를 것이다. 앞서도 언급했지만 쓰레드는 생성 시간이 결코 짧지 않다.  
  
세 번째 이점은 ThreadPool에 있는 적절히 설정된 개수(초기 생성할 쓰레드의 개수와 생성할 수 있는 최대의 쓰레드 개수)의 쓰레드만을 사용함으로써 너무 많은 쓰레드 생성에 의한 시스템 성능저하나 최악의 경우 OutOfMemoryException 을 피할 수 있다는 것이다. (1회 기사에서 각각의 쓰레드는 자신만의 CPU와 스택 영역(메모리)을 사용한다고 했다. 즉, 쓰레드 자체가 메모리를 소비한다는 말이다)  
  
<source 1> 은 ThreadPool 클래스의 코드이다. ThreadPool은 프로그램의 초기화 때 지정한 개수 만큼의 쓰레드를 미리 만들어서 큐(선입선출 큐 : FIFO Queue)에 넣어둔다. 그리고 필요할 때마다 큐에 접근해서 쓰레드를 꺼내서 사용하고 사용이 다 끝나면 다시 큐에 저장해서 나중에 다시 재사용하도록 하는 것이다. 그러나 큐에 접근해서 쓰레드를 꺼내려고 하는데, 만약 큐에 대기중인 쓰레드가 없다면 현재 생성된 쓰레드의 개수를 확인하고 생성할 수 있는 최대의 쓰레드 개수를 넘지 않았다면 새로 생성해서 건네주도록 할 것이다. 여기서 생성할 수 있는 최대의 쓰레드 개수에 도달했다면 사용중인 쓰레드가 큐에 반환되기를 기다렸다가 건네 줄 것이다. 또한 큐는 현재 대기중인 쓰레드의 개수가 초기에 생성한 쓰레드 개수보다 클 경우 큐로 반환되는 쓰레드를 보관하지 않고 폐기할 것이다. wait 변수는 큐에 대기중인 쓰레드가 없을 경우 생성할 수 있는 최대의 쓰레드 개수를 넘지 않았을 경우 바로 생성해서 건네줄지 아니면 큐에 사용이 끝난 쓰레드가 들어오기를 기다릴지를 결정한다. 여기서 주의 깊게 봐야할 점은 synchronized 키워드이다. 동기화 문제가 발생하지 않도록 synchronized를 사용하되 효율을 위해 사용 블록을 최소화 시켜야 한다.

<source 1> ThreadPool.java  
  
public class ThreadPool {  
private static final int MAX\_POOLSIZE = 15;  
private static int poolSize = 5;  
  
private final ArrayList queue = new ArrayList();  
  
private boolean wait = false  
private int total = 0;  
private int index = 0;  
  
private AdvancedNioServer server;  
  
public ThreadPool(AdvancedNioServer server) {   
this(poolSize, server);   
this.server = server;  
}  
  
public ThreadPool(int size, AdvancedNioServer server) {  
poolSize = size;  
  
for (index = 0; index < poolSize; index++) {  
WorkerThread thread = new WorkerThread(this, server);  
thread.setName("Worker" + (index + 1));  
thread.start();  
queue.add(thread);  
total++;  
}  
}  
  
public WorkerThread getThread() {  
WorkerThread worker = null  
  
if (queue.size() > 0) {  
synchronized (queue) {   
worker = (WorkerThread) queue.remove(0);  
}  
} else {  
if (wait) {  
return waitQueue();  
} else {  
if (index < MAX\_POOLSIZE) {  
worker = new WorkerThread(this, server);  
worker.setName("Worker" + (index + 1));  
worker.start();  
total++;  
return worker;  
} else {  
return waitQueue();  
}  
}   
}  
return worker;  
}  
  
private synchronized WorkerThread waitQueue() {  
while (queue.isEmpty()) {  
try {  
queue.wait();  
} catch (InterruptedException ignored) {  
}  
}  
return (WorkerThread) queue.remove(0);  
}  
  
public void putThread(WorkerThread thread) {  
if (queue.size() >= poolSize) {  
thread = null;  
--index;  
} else {  
synchronized (queue) {  
queue.add(thread);  
queue.notify();  
}  
}  
}  
  
public boolean isWait() { return wait; }  
public void setWait(boolean wait) { this.wait = wait; }  
}

필요성을 못느껴 ThreadPool 종료시 큐 안의 모든 쓰레드를 메모리에서 해제하는 메소드를 만들지는 않았지만 만약 필요하다면 간단하므로 직접 위 소스에 추가하면 될 것이다.  
  
이제 ThreadPool 안에 저장되어 재사용되고 또 서버의 실제 서비스를 담당할 WorkerThread 만 만들어 추가하면 ThreadPool 컴포넌트가 완성된다. WorkerThread를 만들기 전에 우선 이 WorkerThread 에서 사용될 ByteBufferPool를 만들 것이다. 그리고 방화벽 통과를 위한 방법과 프로토콜을 XML 로 이용하는 것에 대해 잠시 설명할 것이다. 그런 후에 마지막 준비로 자바 특유의 언어적 특징을 이용해서 서버의 기능 확장시 다른 언어로는 상상할 수도 없는 “죽지 않는 서버”를 만들기 위한 테크닉에 대해 살펴보도록 하겠다. 그 후에 이것들을 종합하여 WorkerThread를 만들 것이다. 자, 그럼 계속 전진이다.  
  
파일 메모리의 도입과 효율적인 메모리 사용을 위한 ByteBufferPool

ByteBuffer buf = ByteBuffer.allocateDirect(4096);  
readCount = sc.read(buf);  
if (readCount < 0) {  
room.removeElement(sc);  
sc.close();  
}  
buf.flip();  
broadcast(buf);  
buf.clear();  
buf = null;

위의 소스는 1회 기사에서 사용했었던 소스의 일부분이다. 만약 1회 기사를 제대로 읽었다면 위 소스에 몇 가지 문제점이 있음을 곧바로 알 수 있을 것이다. 앞서 설명했던 ThreadPool 의 상황과 거의 비슷하다.  
  
첫 번째는 ByteBuffer 를 늘 직접 할당, 해제 하면서 사용하기 때문에 가비지 컬렉션 대상이 증가하므로 가비지 컬렉터가 자주 호출될 수 있다는 점이다.   
  
두 번째는 채널에 write 하기 위해서는 어차피 효율을 위해 allocateDirect 로 버퍼를 만들어 사용해야 하는데 이 녀석은 생성하는데 시간이 오래 걸린다는 점이다.   
  
세 번째는 만약 이 서버에 순간적으로 너무 많은 요청이 한꺼번에 몰리면 메모리만을 버퍼로 사용하기 때문에 OutOfMemoryException 으로 시스템이 다운될 위험이 있다는 점이다. 물론 흔히 발생하는 상황은 아니고 CPU 과부하로 먼저 서버가 죽지 않는다는 전제 안에서의 이야기지만 우리 개발자들은 항상 그 “만약”에 대비해야 하지 않는가!  
  
그럼 위의 세가지 문제를 모두 해결할 수 있는 비급을 소개하겠다. 바로 ByteBufferPool 이다.   
ByteBufferPool 은 앞서 설명한 ThreadPool 과 마찬가지로 GoF의 ObjectPool 패턴을 이용한다.   
  
이 녀석은 일반적으로 getMemoryBuffer() 를 통해서 메모리를 버퍼로 할당해서 사용한다. 하지만 정해진 메모리 버퍼를 모두 사용하고 있을 때에는 파일을 버퍼로 사용한다. 물론 파일을 메모리로 사용하면 실제 메모리를 사용하는 것보다는 느리다. 하지만 클라이언트가 거의 차이를 느끼지 못할 만큼의 퍼포먼스가 나오므로 속도 문제로 고민하지는 말자. 오히려 문제가 있다면 네트워크 트래픽이 클라이언트의 체감 속도에 더 영향을 줄 것이다. 그리고 무엇보다도 파일을 버퍼로 사용함으로써 메모리만을 버퍼로 사용할 때 발생할 수 있는 메모리 부족으로 인한 시스템 다운을 막을 수 있지 않은가. 마치 윈도우 운영체제에서 하드를 가상메모리로 설정해서 사용하는 것과 같은 원리다. 멋진 녀석이니 전체 소스를 살펴보도록 하자.

<source 2> ByteBufferPool.java  
  
public class ByteBufferPool {  
private static final int MEMORY\_BLOCKSIZE = 4096;  
private static final int FILE\_BLOCKSIZE = 10240;  
  
private final ArrayList memoryQueue = new ArrayList();  
private final ArrayList fileQueue = new ArrayList();  
  
private boolean wait = false;  
  
public ByteBufferPool(int memorySize, int fileSize, File file) throws IOException {  
if (memorySize > 0)   
initMemoryBuffer(memorySize);  
  
if (fileSize > 0)  
initFileBuffer(fileSize, file);  
}  
  
private void initMemoryBuffer(int size) {  
int bufferCount = size / MEMORY\_BLOCKSIZE;  
size = bufferCount \* MEMORY\_BLOCKSIZE;  
ByteBuffer directBuf = ByteBuffer.allocateDirect(size);  
divideBuffer(directBuf, MEMORY\_BLOCKSIZE, memoryQueue);  
}  
  
private void initFileBuffer(int size, File f) throws IOException {  
int bufferCount = size / FILE\_BLOCKSIZE;  
size = bufferCount \* FILE\_BLOCKSIZE;  
RandomAccessFile file = new RandomAccessFile(f, "rw");  
try {  
file.setLength(size);  
ByteBuffer fileBuffer = file.getChannel().map(FileChannel.MapMode.READ\_WRITE, 0L, size);  
divideBuffer(fileBuffer, FILE\_BLOCKSIZE, fileQueue);  
} finally {  
file.close();  
}  
}  
  
private void divideBuffer(ByteBuffer buf, int blockSize, ArrayList list) {  
int bufferCount = buf.capacity() / blockSize;  
int position = 0;  
for (int i = 0; i < bufferCount; i++) {  
int max = position + blockSize;  
buf.limit(max);  
list.add(buf.slice());  
position = max;  
buf.position(position);  
}  
}  
  
public ByteBuffer getMemoryBuffer() {  
return getBuffer(memoryQueue, fileQueue);  
}  
  
public ByteBuffer getFileBuffer() {  
return getBuffer(fileQueue, memoryQueue);  
}  
  
private ByteBuffer getBuffer(ArrayList firstQueue, ArrayList secondQueue) {  
ByteBuffer buffer = getBuffer(firstQueue, false);  
if (buffer == null) {  
buffer = getBuffer(secondQueue, false);  
if (buffer == null) {  
if (wait)  
buffer = getBuffer(firstQueue, true);  
else  
buffer = ByteBuffer.allocate(MEMORY\_BLOCKSIZE);  
}  
}  
return buffer;  
}  
  
private ByteBuffer getBuffer(ArrayList queue, boolean wait) {  
synchronized (queue) {  
if (queue.isEmpty()) {  
if (wait) {  
try {  
queue.wait();  
} catch (InterruptedException e) {  
return null;  
}  
} else {  
return null;  
}  
}  
return (ByteBuffer) queue.remove(0);  
}  
}  
  
public void putBuffer(ByteBuffer buffer) {  
if (buffer.isDirect()) {  
switch (buffer.capacity()) {  
case MEMORY\_BLOCKSIZE :  
putBuffer(buffer, memoryQueue);  
break;  
case FILE\_BLOCKSIZE :  
putBuffer(buffer, fileQueue);  
break;  
}  
}  
}  
  
private void putBuffer(ByteBuffer buffer, ArrayList queue) {  
buffer.clear();  
synchronized(queue) {  
queue.add(buffer);  
queue.notify();  
}  
}  
  
public synchronized void setWait(boolean wait) { this.wait = wait; }   
public synchronized boolean isWait() { return wait; }  
}

위의 소스에서 initFileBuffer(int size, File f) 메소드에서 FileChannel이 사용되었다. 전 시간에 살펴보았던 SocketChannel 이나 ServerSocketChannel 과 달리 파일채널은 SelectableChannel을 상속받지 않는다. 그말은 즉, 파일채널은 넌블럭킹 모드로 설정할 수 없다는 말이다. 그러나 파일채널은 기존의 io 로 파일을 제어하는 것보다 많은 장점을 갖고 있다. 그중 하나가 파일채널은 해당 OS 의 파일 캐쉬를 사용해서 파일에서 파일로 직접 전달할 수 있고 파일의 특정 부분을 잠글수(locking) 있다는 점이다. 또한 ByteBufferPool 클래스에서 사용했듯이 파일채널은 메모리로 파일의 일부 영역을 매핑(mapping) 시킬 수 있다. 파일을 메모리로 매핑할 때 파일의 내용을 메모리 위치들처럼 사용하기 위해 OS 의 네이티브 메모리 관리자를 사용한다. 이때 효율적인 매핑을 위해 OS는 디스크 페이징 시스템(disk paging system)을 사용한다. 애플리케이션 관점에서는 매핑된 파일 내용은 단지 특정 주소값을 갖고 메모리 안에 일렬로 쭉 늘어선 것으로 인식한다. 즉, 메모리와 별반 차이가 없이 인식한다는 말이다.  
파일채널은 파일을 메모리로 매핑할 때 메모리 영역을 표현하기 위해 MappedByteBuffer를 사용하는데 이것의 타입은 direct ByteBuffer 이다. MappedByteBuffer는 2가지 큰 장점이 있는데 첫째는 메모리에 매핑된 파일을 읽는 것이 상당히 빠르다는 것이다. 물론 파일을 순차적으로 읽는 것이 가장 빠르고 기존 io를 사용하는 것보다 성능이 많이 개선되지만 RandomAccessFile 과 같이 임의의 부분에 접근해서 읽는 것도 기존에 비해 성능이 많이 개선되었다. 이것은 기존의 BufferedInputStream 으로 특정 블록을 읽는 것보다 훨씬 더 많은 부분을 메모리 안으로 파일을 페이지(page) 해서 OS 가 읽기 때문이다. 두 번째는 MappedByteBuffer를 이용해서 파일을 보내는 것이 상당히 간단하다는 것이다.   
  
ByteBufferPool 소스를 보면 내부적으로 2개의 큐를 만든다. 메모리 버퍼와 파일 버퍼를 관리하기 위한 큐다. 생성자에서 전제적으로 사용할 메모리 크기와 파일 크기 및 파일을 받아서 각각의 버퍼를 만든다. wait 변수는 큐에 대기중인 버퍼가 없을 경우 기다릴지 여부를 결정하는 플래그다. 우리가 직접적으로 접근해서 사용할 수 있는 메소드는 public 으로 선언된 것들이다. 기본적으로 우리는 getMemoryBuffer()를 통해서 메모리 버퍼를 사용할 것이다. 소스를 보면 알 수 있겠지만 만약 메모리 버퍼를 얻으려 한 경우 큐에 대기중인 메모리 버퍼가 없을 경우에는 파일 버퍼를 얻기 위해 시도하고 그 반대의 경우에는 파일 버퍼를 얻기 위해 시도할 것이다. 따라서 대부분 메모리 버퍼를 먼저 사용할 것이므로 메모리 부족으로 인한 기다림이나 시스템 다운을 파일 버퍼가 커버해주는 한에서 예방할 수 있다.  
  
방화벽을 뚫기(Firewall Tunneling) 위해 Http 통신을 이용하자  
방화벽은 패킷 필터와 애플리케이션 계층 게이트웨이(프락시)라는 두가지 형태가 존재한다. 패킷 필터링은 보통 외부와 인터페이스라는 라우터의 IP 계층에서 일어난다. 패킷 필터링 시에는 접근 제어 리스트를 참조해서 내부 호스트로의 패킷을 허용할지 여부를 결정하게 된다. 애플리케이션 계층 게이트웨이는 보통 프락시라는 이름으로 알려져 있는데 클라이언트가 외부로의 접속을 맺으려 할 때 내부 클라이언트와 외부 서비스와의 사이를 중계하는 서버다. 보통은 외부 호스트의 80번 포트로 나가는 내부 클라이언트의 접속을 받아들인다.   
  
필자가 만든 서버는 4567번 포트를 사용하는데 만약 프락시 서버를 고려한다면 서버의 포트를 80번으로 바꾸고 클라이언트에서 System.setProperties() 메소드로 프락시 서버로 먼저 접속하도록 만들어야 한다. 하지만 편의상 필자는 프락시 서버를 이용한 방화벽은 고려하지 않았다. 또한 http 프로토콜의 다양한 코드에 대한 처리도 하지 않았다. 따라서 그냥 일반적인 http 프로토콜을 허용하는 방화벽 통과를 위해 클라이언트에서 서버로 요청을 보낼때와 서버에서 클라이언트로 응답할 때 다음과 같이 각각 http 요청-응답 헤더만 붙여서 사용했다.  
  
private static final String HttpRequestHeader = "POST / HTTP/1.1\r\n\r\n";  
private static final String HttpResponseHeader = "HTTP/1.1 200 OK\r\n\r\n";  
  
**프로토콜에 XML을 입히자**필자는 서버간 통신 메시지에 xml을 사용했다. 그 이유는 xml 은 사람과 기계(컴퓨터) 모두가 이해하기 쉬운 구조로 되어 있고 이미 널리 알려져 있듯이 W3C 에 의해 표준이 된 언어이다. 프로토콜에 xml을 사용함으로써 얻을 수 이점은 여러 가지가 있지만 우선 특정 언어나 시스템에 종속적이지 않다는 점이다. xml 로 작성된 우리의 통신 프로토콜은 어떤 운영체제나 프로그램 언어에서도 쉽게 인식 및 사용 가능하다. 두 번째로는 xml 언어 자체가 갖는 확장성이 부록으로 따라오기 때문이다. 만약 프로토콜의 확장이 필요할 때 이미 xml을 사용하고 있었다면 큰 고민없이 쉽게 확장 할 수 있을 것이다. 즉, xml 의 사용으로 프로토콜의 확장성과 유연성을 갖게 된다고 말할 수 있다. 세 번째로는 만약 우리의 통신 프로토콜이 잘 정의가 되어 있다면 SOAP 과 같이 표준으로서 인정되어 사용될 수 있다는 것이다. 아직 표준으로 인정되지는 않았지만 jabber 라는 플랫폼에선 이미 몇 년 전부터 잘 구조화된 프로토콜을 정의해서 사용하고 있다. 첨언이지만 필자는 개인적으로 현재 IT 업계의 싸움은 실제적인 기술이나 전략 보다도 자신의 기술을 표준으로 만들기 위한 힘 싸움으로 생각하고 있다. 따라서 표준이라는 말은 적어도 필자에게는 큰 의미로 다가온다. 마지막으로 xml 은 프로그래밍 언어에서 데이터로 사용하기에 상당히 단순하고 간편하다. 필자가 만든 서버에서 사용한 형식의 다음 xml 메시지 구조를 보면 알 수 있겠지만 xml을 모르더라도 쉽게 이해할 수 있을 것이다.

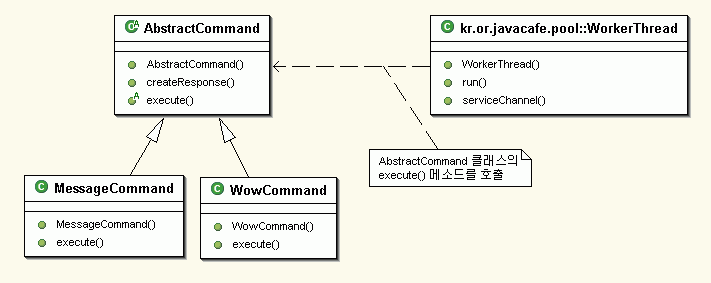
<?xml version='1.0' encoding='UTF-8'?>  
<request>  
<command>MessageCommand</command>  
<message>안녕하세요~!</message>  
</request>  
  
<?xml version='1.0' encoding='UTF-8'?>  
<response>  
<message>네. 반갑습니다.</message>  
</response>

필자는 예제로 만든 서버에서 단지 클라이언트가 보낸 메시지를 브로드캐스트 하는 방식으로만 구현했지만 실제 메신저 등을 만든다고 생각하면 xml을 이용함으로써 얼마나 명쾌하고 편리하게 구현할 수 있을지가 피부로 와 닿으리라 생각한다. 물론 전송 데이터의 양이 조금 더 많아지므로 속도면에서는 약간의 손해를 볼 것이다. 하지만 현명한 독자들은 xml 사용시 얻는 장점에 초점을 맞출거라 생각한다.  
  
Dynamic Class Loading 과 Command 패턴을 이용하여 "멈추지 않는 서버" 만들기  
지난 3년간 자바를 공부하면서 필자를 가장 흥분시켰던 자바의 특징이 바로 자바의 Dynamic Class Loading 이였다. 컴파일시에 어떤 클래스가 쓰일 것이라는 사실을 모르더라도 런타임에 필요한 클래스를 찾아서 사용한다는 것이 얼마나 필자의 가슴을 두근거리게 했었는지...   
  
이제부터 Command Pattern 과 바로 자바의 언어적 특징 중 하나인 Dynamic Class Loading을 이용하여 서버를 리부팅 하지 않고도 기능 확장을 할 수 있는 서버를 만들기 위한 테크닉을 알아볼 것이다. 기대되지 않는가? 자, 그럼 마음을 가다듬고 정신을 집중해서 새로운 세상으로 한걸음 나아가보자.  
  
먼저 Dynamic Class Loading 과 Command Pattern 에 대해서 알아보고 이 둘을 이용해서 어떻게 우리의 서버를 기능 확장시에 “멈추지 않는 서버”로 만들 것인지를 살펴보도록 할 것이다.   
  
자바는 JVM(자바 가상 머신) 실행시에 java.lang.ClassLoader 클래스에 의해 코드가 링크되는 동적 링크 시스템이다. 모든 자바 클래스는 모두 예외 없이 ClassLoader에 의해 자바 가상 머신 내부로 로드된다. 이때 2가지 방식의 로딩을 사용하는데 바로 로드타임 동적 로딩과 런타임 동적 로딩이다.  
  
우선 우리가 일반적으로 사용하던 로드타임 동적 로딩을 보도록 하자.

public class HelloWorld {  
public static void main(String[] args) {  
System.out.println("Hello, world!");  
}  
}

위의 코드에선 JVM이 HelloWorld 클래스를 로드할 때, 이 클래스 안에서 System 클래스가 사용된다고 컴파일러가 JVM 에게 알려주고 그럼 JVM은 HelloWorld 클래스 로딩을 잠시 중단하고 System 클래스를 로딩한 후에 다시 HelloWorld 클래스를 로딩한다. 즉, HelloWorld 클래스 안에서 사용되는 모든 클래스들(System 이외에도 모든 클래스가 상속하는 Object 와 System 안에서 사용되는 PrintStream 클래스)이 로드타임에 미리 로드 되는 것이다.  
  
이에 반해 런타임 동적 로딩은 어려운 것은 아니지만 앞서 설명한 로드타임 동적 로딩과는 약간 다른 방식으로 사용된다. 아래 코드를 보고 “많이 보던건데..“ 하고 생각하시는 분들이 계실거라 짐작된다.

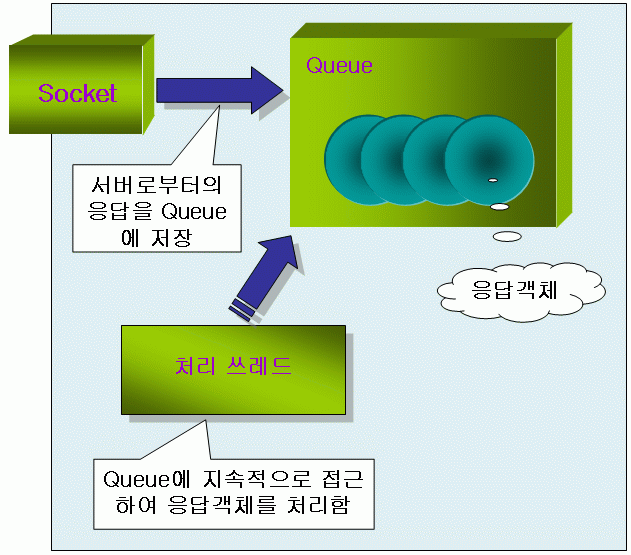
Class cls = Class.forName("java.lang.String");  
Object obj = cls.newInstance();  
String s = (String) obj;

“Class.forName("org.gjt.mm.mysql.Driver")” 같이 JDBC에서 특정 벤더의 JDBC 드라이버 클래스를 로딩할 때 많이 보았을 것이다. 위 코드에서는 forName() 메소드 안의 인자로 들어온 String 에 해당하는 클래스를 런타임시에 동적으로 로딩한다. 소스를 통해서도 알 수 있겠지만 forName() 메소드를 통해 리턴되는 것은 Class 다. 이것을 newInstance() 메소드를 통해 Object 타입으로 바꾸고 이것을 다시 원하는 적절한 타입으로 캐스팅해서 사용하면 되는 것이다. 자바의 런타임 동적 로딩의 한계가 있다면 바로 로딩할 클래스의 이름(패키지를 포함한)과 실제 타입(캐스팅할 타입)을 알아야만 한다는 것이다. 하지만 자바의 런타임 동적 로딩은 그런 한계를 극복하고도 남을 만큼의 충분한 가치가 있다.  
  
<그림 2>는 필자가 만든 예제 소스에서 Command 패턴의 사용 부분을 보여주고 있다. 이 기사는 UML 이나 DP(Design pattern)를 설명하는 것이 목적이 아니기에 그것들이 예제 소스에서 단순히 어떻게 사용되었는지에 초점을 맞춰서 설명하겠다. WorkerThread에서 클라이언트가 요청한 데이터를 파싱해서 커맨드와 메시지를 분리한 후에 커맨드 이름과 같은 클래스가 있는지를 동적으로 찾는다. 만약 있다면 그 클래스를 <그림 2>에서와 같이 AbstractCommand 형식의 상위 타입으로 캐스팅해서 execute() 메소드를 실행시켜 서비스를 한다. 실제로 수행되는 execute() 메소드는 AbstractCommand 클래스를 상속한 우리가 동적으로 찾은 서브 커맨드 클래스의 execute() 메소드이다. 또한 우리는 이렇게 사용할 커맨드 객체를 재사용하기 위해 서버에 HashMap 저장소를 하나 만들고 이곳에 처음 생성된 커맨드 클래스를 저장해서 재사용하려고 시도할 것이다. 만약 이미 저장되어 있다면 앞으로는 HashMap 저장소에서 저장된 커맨드를 꺼내와서 사용하게 된다. 이때 일부 독자는 멀티 쓰레드 환경에서 하나의 객체로 서비스 하는게 안전한가 하는 질문을 던지시는 분도 계시리라 생각된다. 그 답은 “안전하다” 이다. 왜냐하면 커맨드 객체를 이용하는 것은 쓰레드이기 때문이다. 앞서도 설명했듯이 쓰레드는 자신만의 스택 영역과 CPU를 점유해서 사용한다고 했다. 여기서 자신의 스택을 사용한다는 말은 데이터 처리에 있어 여러 쓰레드가 동시 접근을 한다고 해도 자신의 스택안에서 관련 데이터를 처리하므로 안전하다는 말이다.(물론 static 같은 정적 데이터나 클래스 멤버로 선언된 필드는 쓰레드들 사이에서 공유된다.) 지금까지 계속 살펴보았듯이 재사용할 수 있는 객체들은 최대한 재사용할 수 있게 하는 것이 중요하다.   
  
  
  
<그림 2> Command 패턴의 Class Diagram  
  
**실제 서비스를 담당하는 WorkerThread**ByteBuffer 에 데이터를 넣을 때 2가지 문제점을 맞이하게 된다. 바로 바이트 순서(byte ordering)와 캐릭터 변환(character conversion)이다. ByteBuffer 는 내부적으로 ByteOrder 클래스를 사용해서 바이트 순서를 결정한다. Big-endian 과 Little-endian 두가지로 설정할 수 있는데 기본적으로 ByteBuffer 는 Big-endian을 사용한다. 만약 다른 언어나 이기종의 시스템간의 통신을 해야 한다면 바이트 순서를 고려해야 하지만 자바간의 통신이라면 별도의 처리가 필요하지 않으므로 더 이상 언급하지 않겠다. 이것에 대한 자세한 내용은 선의 공식 문서 등을 참고하기 바란다.   
이제 캐릭터 변환에 대해 이야기 할 차례다. 영어권 나라들의 경우에는 하나의 문자를 1바이트로 표현하고 있고 또 대부분의 원천 기술이 그들에게서 나오기 때문에 우리 나라와 같은 2바이트 문화권의 프로그래머들은 늘 인코딩 문제로 고심하게 된다. 필자는 예제 소스에서 xml을 사용하기 때문에 UTF-8 형식의 유니코드 인코딩을 사용했다.(또한 자바에서는 디폴트로 유니코드를 사용하기 때문에)   
보통 인코딩은 getBytes() 메소드에 파라미터로 인코딩 타입을 명시해서 사용한다. 또한 대부분의 경우 디코딩은 ByteBuffer를 직접 제어해서 사용하게 되는데 인코딩 및 디코딩 사용법은 다음과 같이 상당히 간단하다. (java.nio.charset 패키지에 포함되어 있음)

Charset charset = Charset.forName("UTF-8");  
CharsetEncoder encoder = charset.newEncoder();  
// 인코드 후 리턴되는 버퍼는 non-direct 버퍼임에 주의.  
ByteBuffer encodedBuffer = encoder.encode(buffer);  
  
Charset charset = Charset.forName("UTF-8");  
CharsetDecoder decoder = charset.newDecoder();  
CharBuffer charBuffer = decoder.decode(buffer);  
String result = charBuffer.toString();

<source 3>은 실제 서비스를 담당하는 WorkerThread 다. WorkerThread 의 핵심 메소드는 requestProcess() 이다. 이 메소드는 클라이언트의 요청을 읽어서 http 헤더를 제거하고 몸체로 딸려온 xml 부분을 파싱해서 커맨드와 메시지로 나눈다. 이때에도 xml을 파싱하는 SaxHandler 클래스를 싱글턴 패턴을 사용해서 재사용한다. 소스를 보면 알 수 있겠지만 패턴이라고 해서 어렵게 생각 할 것은 없다. 아마 첨부된 소스를 보면 쉽게 이해가 갈 것이다.   
그 후 커맨드와 같은 이름의 커맨드 클래스를 동적으로 찾는다. 그리고 그 클래스의 execute() 메소드를 호출해서 클라이언트에게 응답을 보낸다. 이 커맨드 객체도 앞서 설명했듯이 HashMap 에 저장해서 재사용한다. 또한 서비스를 마치고 난 후에는 ByteBufferPool에서 가져온 ByteBuffer를 finish() 메소드에서 반환하고 WorkerThread 도 run() 메소드 끝부분에서 ThreadPool 로 반환한다.(전체 소스는 ‘이달의 디스켓’에 첨부했다)

<source 3> WorkerThread.java  
  
public class WorkerThread extends Thread {   
  
...  
  
private void requestProcess(SelectionKey key) throws IOException {  
SocketChannel sc = (SocketChannel) key.channel();  
ByteBuffer buffer = bufferPool.getMemoryBuffer();  
int count;  
  
AbstractCommand command = null;  
  
count = sc.read(buffer);   
buffer.flip();  
  
if (count < 0) {  
server.removeUser(sc);  
sc.close();  
bufferPool.putBuffer(buffer);  
key.selector().wakeup();  
server.info("클라이언트가 접속을 종료했습니다.");  
return;  
}  
  
charBuffer = decoder.decode(buffer);   
String temp = charBuffer.toString();  
byte[] bb = temp.substring(temp.indexOf("\r\n\r\n")).trim().getBytes("UTF-8");  
String[] param = parsingXML(bb);  
if (param == null) {  
finish(buffer);  
return;  
}  
  
if (server.isContainCommand(param[0])) {  
command = server.getCommand(param[0]);  
} else {  
try {  
command = (AbstractCommand) Class.forName(CommandPath + param[0]).newInstance();  
} catch (Exception e) {  
server.info("명령을 수행하는 커맨드 클래스가 존재하는지 확인해보세요.");  
finish(buffer);  
return;  
}   
  
if (command == null) {  
finish(buffer);  
return;  
} else {  
server.putCommand(param[0], command);  
}  
}  
buffer.clear();  
command.execute(server, buffer, param[1]);  
finish(buffer);   
}  
  
private void finish(ByteBuffer buffer) {  
bufferPool.putBuffer(buffer);  
key.interestOps(key.interestOps() | SelectionKey.OP\_READ);   
key.selector().wakeup();   
}  
  
private String[] parsingXML(byte[] xml) {  
String[] param = null;  
ArrayList list = null;  
  
if (xml == null)   
return param;  
  
in = new ByteArrayInputStream(xml);  
try {   
handler = SaxHandler.getInstance();   
parser.parse(in, handler);   
list = handler.getContents();   
} catch (SAXException e) {  
server.log(Level.WARNING, "WorkerThread/parsingXML()", e);  
} catch (IOException ex) {  
server.log(Level.WARNING, "WorkerThread/parsingXML()", ex);  
}  
  
param = new String[list.size()];  
for (int i = 0; i < list.size(); i++) {   
param[i] = (String) list.get(i);  
}   
handler.clearSaxHandler();   
return param;  
}   
}

**클라이언트 애플리케이션**필자는 지난번에 말했던 것처럼 클라이언트도 nio(니오)를 사용해서 만들었다. 서버와 달리 클라이언트는 많은 양의 동시처리가 필요하지 않으므로 꼭 nio를 사용할 필요는 없다. 물론 nio를 사용하면 블러킹이 없어 좀 더 효율적이지만 말이다. 그러나 안정성을 생각해서 클라이언트는 기존의 소켓으로 만드는 것이 더 좋을 듯 하다. 필자가 시간 관계상 클라이언트는 거의 신경을 쓰지 못하고 예제 소스를 만들었지만 만약 제대로 된 구조를 갖추게 설계한다면 클라이언트도 <그림 3>과 같이 이벤트 중심 설계를 하는 것이 바람직하다. 즉, 서버에서 도착한 모든 응답(이벤트)을 커맨드와 메시지 등으로 나눠서 객체로 포장하고 그 객체를 큐에 쌓아두고 큐에 접근하는 별도의 소비자 쓰레드가 지속적으로 큐를 체크하면서 처리해 나가는 것이다. 이렇게 하는 이유는 클라이언트 프로그램이 반드시 서버와 단 하나의 연결만을 갖게 된다는 보장이 없고 대개의 경우 주로 여러개의 연결을 유지하는 경우가 많기 때문이다. 예를 들어 MSN 메신저 클라이언트를 개발한다고 했을 경우 MSN 메신저는 자신의 상태 및 대화명 등을 관리해주는 Dispatcher 서버와 하나의 연결이 생성되고 친구와 대화를 할 때에는 Switchboard 서버에 연결해서 처리를 한다. 이때 서버로부터 들어오는 모든 응답을 한 곳에서 관리하는 것이 유지보수나 확장성 등에서 좀 더 유용하기 때문이다. 따라서 필자는 대부분의 클라이언트 소켓 애플리케이션의 경우 이벤트 중심 설계가 훌륭한 모델이 될 것이라고 생각한다.   
그리고 서버와 같이 Command 패턴을 도입해서 손쉬운 확장이 가능하도록 만들고 여기에 런타임 동적 로딩을 추가해 클라이언트를 업그레이드 할 때 UI 부분이 아닌 기능상의 변화라면 굳이 프로그램을 재시작 하지 않아도 되게 하면 좀 더 좋을 것이다.   
  
  
  
<그림 3> 이벤트 중심 아키텍쳐  
  
  
  
  
**연재를 마치며...**지금까지 서버의 성능을 향상시키기 위해 객체의 재사용에 중점을 두고 필요한 것들을 하나하나씩 만들어 추가해봤다. 성능향상을 위해 특별한 기술을 사용한 것은 없었다. 단지 재사용할 수 있는 객체들을 재사용함으로써 좀 더 효율적으로 메모리를 사용하고 또한 객체의 생성으로 인한 시간을 절약할 수 있었다. 또 이렇게 함으로써 눈에 보이지는 않지만 가비지 컬렉터의 호출도 조금은 줄 일 수 있어서 퍼포먼스 향상에 도움이 되었다. 지금까지 우리가 개발한 것들은 소프트웨어의 개발시에 할 수 있는 튜닝 작업에 해당한다. 2차적으로는 실제로 서비스를 하기 위해 JVM 옵션을 이용하여 힙(heap)이나 쓰레드 초기 스택(stack) 사이즈 등을 지정함으로써 성능을 개선해야 할 것이다. 이것은 추후 사용 서비스를 할 때 많은 테스트를 통해 JVM을 최적화해야 하는 것으로 독자분들의 몫이다. 마지막으로 독자들에게 바람이 있다면 우선 필자가 제공한 예제 소스에 그치지 말고 좀 더 나은 채팅이나 메신저 등의 솔루션으로 발전시켰으면 하고 또 그것을 넘어서 분산처리가 가능한 대용량의 확장성 있는 서버를 만들었으면 한다. 그럼 독자분들의 행운을 빌며 2회에 걸친 기사를 마무리 하겠다.   
  
**참고자료**1. Java Nio by Ron Hitchens, 2002 O'reilly  
  
2. Patterns in Java Volume 1, Wiley  
  
3. Server-Based Java Programming, 인포북  
  
4. Java Network Programming, 인포북  
  
5. <http://www.jabber.org>   
  
6. <http://developer.java.sun.com/developer/technicalArticles/InnerWorkings/JDCPerformTips>  
  
7. <http://www.javaworld.com/javaworld/jw-09-2001/jw-0907-merlin.html>   
  
8. [소스 자료](http://www.javacafe.or.kr/lecture/cafeLecture/network/threadpool/maso%202th%20source.zip)