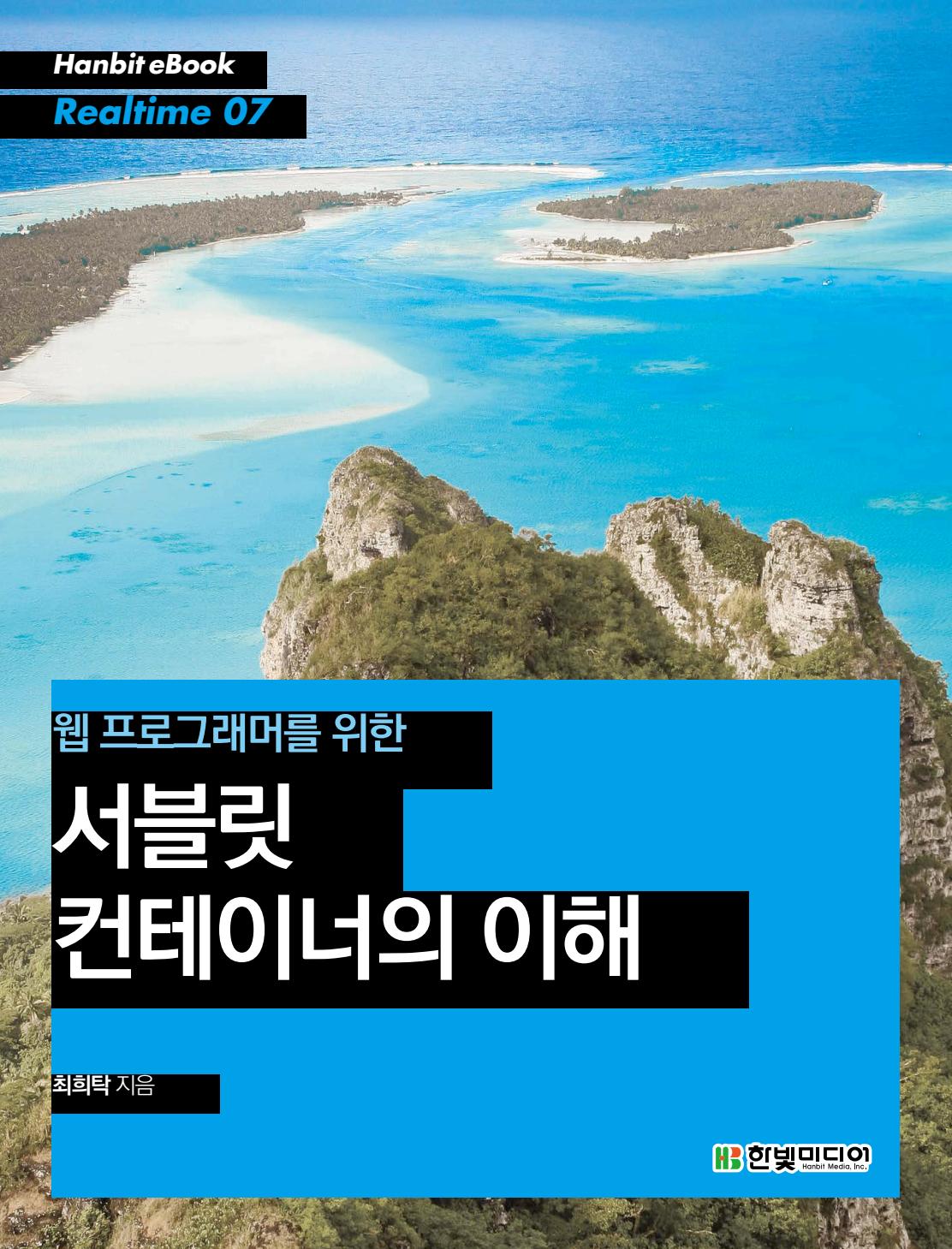


Hanbit eBook

Realtime 07



웹 프로그래머를 위한

# 서블릿 컨테이너의 이해

최희탁 지음

웹 프로그래머를 위한

# 서블릿 컨테이너의 이해

## 지은이 **최희탁**

서울대학교에서 수학을 전공하였으며, 티맥스소프트에서 룰 엔진과 웹 애플리케이션 서버 구현에 참여하였다. 현재는 NHN에서 소셜 관련 개발을 하고 있다.

● e-mail: endofhope@naver.com

## 웹 프로그래머를 위한 **서블릿 컨테이너의 이해**

---

초판발행 2012년 9월 27일

**지은이** 최희탁 / **펴낸이** 김태현

**펴낸곳** 한빛미디어(주) / **주소** 서울시 마포구 양화로 7길 83 한빛미디어(주) IT출판부

**전화** 02-325-5544 / **팩스** 02-336-7124

**등록** 1999년 6월 24일 제10-1779호

**ISBN** 978-89-7914-968-5 15560 / **정가** 13,000원

**책임편집** 배용석 / **기획** 김창수 / **편집** 김병희, 이순옥

**디자인** 표지 여동일, 내지 스튜디오 [밈], 조판 이순옥

**영업** 김형진, 김진불, 조유미 / **마케팅** 박상용, 박주훈

이 책에 대한 의견이나 오탈자 및 잘못된 내용에 대한 수정 정보는 한빛미디어(주)의 홈페이지나 아래 이메일로 알려주십시오.

**한빛미디어 홈페이지** [www.hanb.co.kr](http://www.hanb.co.kr) / **이메일** [ask@hanb.co.kr](mailto:ask@hanb.co.kr)

---

Published by HANBIT Media, Inc. Printed in Korea

Copyright © 2012 최희탁 & HANBIT Media, Inc.

이 책의 저작권은 최희탁과 한빛미디어(주)에 있습니다.

저작권법에 의해 보호를 받는 저작물이므로 무단 복제 및 무단 전재를 금합니다.

---

지금 하지 않으면 할 수 없는 일이 있습니다.

책으로 펴내고 싶은 아이디어나 원고를 메일([ebookwriter@hanb.co.kr](mailto:ebookwriter@hanb.co.kr))로 보내주세요.

한빛미디어(주)는 여러분의 소중한 경험과 지식을 기다리고 있습니다.

## 저자 서문

항상 자식의 선택을 믿고 기다려 주신 부모님께 먼저 감사드립니다. 그 신뢰가 충분히 가치 있었음을 보여 드리기 위해 노력하고 있으나 아직도 부족함이 많습니다.

프로그래머의 길로 들어서게 안내해준 이창신 군에게도 감사의 말을 전합니다. 항상 미지의 길을 찾는 그 여성에 즐거움이 가득하길 빕니다.

시스템 프로그래밍이란 무엇인지, 엔지니어가 지녀야 할 자긍심이란 무엇인지 알게 해 주신 티맥스 소프트의 박대연 교수님께 감사드립니다. 제가 부족해서 기대했던 바에 미치지 못함을 항상 죄송스럽게 생각합니다. 멀리서나마 견승을 빕니다.

옆에 있는 것만으로도 힘이 되는 아내 김현진에게 고맙고 사랑한다는 말을 전합니다. 오늘보다 내일이 반드시 더 좋을 거예요.

사랑하는 딸 은서에게, 네가 있어서 아빠는 행복하다. 엄마 아빠랑 같이하고 싶은 것들을 하나씩 빠짐없이 해 보자꾸나.

마지막으로 외할머니 최분선 여사의 안식을 빕니다. 항상 자신 있게 남들을 주도해 가라고 격려하셨는데 멀리서나마 손자가 잘 해내고 있는지 지켜보고 계시리라 믿습니다.

집필을 마치며  
저자 최희탁

# 대상 독자

초급

초중급

중급

중고급

고급

웹 프레임워크 사용이 대중화되면서 웹 프로그래머가 서블릿을 사용해 서비스를 구현할 기회가 점점 줄어들고 있습니다. 덩달아 웹 프레임워크를 작동시키는 서블릿 컨테이너에 대한 인식 수준 역시 그렇게 높지 않습니다. 하지만 자바 웹 프로그래밍에서 서블릿은 HTTP 요청의 최초 진입점일 뿐 아니라 멀티스레드 구조를 지원함으로써 기본적인 병렬처리 기능을 제공하므로 가장 중요한 구성요소라고 불리는 데 부족함이 없습니다. 무엇보다 웹 프레임워크를 사용해 구현되는 결과물이 결국 내부적으로는 서블릿으로 변환되어 서블릿 컨테이너 위에서 처리된다는 점에서 서블릿과 서블릿 컨테이너에 대한 정확한 이해가 자바 웹 프로그래밍에서 차지하는 중요성은 무엇보다 크다고 하겠습니다.

어느 정도 경력이 쌓여 매너리즘에 대해 고민한 자바 웹 프로그래머에게는 지금까지 자신이 별다른 자각 없이 간접적으로 사용하던 서블릿 API의 아래에서 벌어지는 여러 흥미진진한 일들에 대해 알게 되어 다시 한번 초년의 두근거림과 열정을 불러일으키는 계기가 되기를 기대합니다.

다른 언어를 사용하는 웹 프로그래머에게는 자바 언어 공동체에서 웹 서비스를 바라보는 관점에 대해 이해할 수 있게 되어, 해당 언어로 작성된 웹 애플리케이션 서버나 프레임워크에 대해 좀 더 명확한 관점을 가지는 데 도움이 될 수 있으리라 기대합니다.

## 예제 테스트 환경

사용 툴 및 장비	버전
아파치 TCPMon	apache-tcpmon-1.0
아파치 톰캣	apache-tomcat-7.0.29
아파치 앤트	apache-ant-1.8.4
자바 SDK	jdk1.6.0_35

- 아파치 TCPMon(Apache TCPMon)

: <http://ws.apache.org/commons/tcpmon/>

- 아파치 톰캣(Apache Tomcat)

: <http://tomcat.apache.org/download-70.cgi>

- 아파치 앤트(Apache Ant)

: <http://ant.apache.org/>

- 자바 SDK

: <http://www.oracle.com/technetwork/java/javase/downloads/index.html>

예제는 zip 형식으로 장 단위로 제공합니다.

예제 파일은 "<http://www.hanb.co.kr/exam/1968/>"에서 받으실 수 있습니다.

# 한빛전자책 알림

세상에는 수많은 종이책이 있습니다. 그리고 그 종이책을 그대로 옮긴 전자책도 많습니다. 그러나 전자책이 단지 종이책을 디지털 파일 형태로 변환해 놓은 것은 아니라고 생각합니다. 전자책에는 전자책에 적합한 콘텐츠와 형식이 있다고 생각합니다.

한빛이 지금 생각하고 추구하는, 개발자를 위한 전자책은 이렇습니다.

## 1. 빠르게 변화하는 IT 기술에 대해 핵심적인 정보를 신속하게 제공합니다.

500페이지 분량의 잘 정리된 도서가 아니라, 조금은 거칠지만 100페이지 내외로 핵심적인 내용을 빠르게 전달하여 독자의 소중한 시간을 아껴 주는 전자책 서비스입니다. 또한 새로운 정보와 지식을 빠르게 전달하기 위해 O'Reilly의 전자책 번역 서비스도 준비 중이며, 조만간 개시될 예정입니다.

## 2. 무료로 업데이트되는, 전자책 전용 서비스입니다.

종이책으로는 기술의 변화 속도를 따라잡기가 쉽지 않습니다. 책이 일정한 분량 이상으로 집필되고 정리되어 나오는 동안 기술은 이미 변해 있습니다.

한빛 eBook only 서비스는 전자책으로 출간된 이후에도 버전 업 등으로 인한 중요한 기술적 변화가 있거나, 저자(역자)와 독자가 소통하면서 보완되고 발전된 노하우가 정리되면 구매하신 분께 무료로 업데이트해 드립니다.

## 3. 전자책 환경을 고려한 최적의 형태와 디자인에 담고자 노력했습니다.

종이책을 그대로 옮겨 놓아 가독성이 떨어지고 읽기 힘든 전자책이 아니라, 전자책의 환경에 가능한 최적화하여 편리한 경험을 드리고자 합니다.

이를 위해 글자크기나 행간, 여백 등을 전자책 전용으로 새롭게 디자인했으며, ePUB이 아닌 PDF 포맷을 채택했습니다. 앞으로도 독자분들의 쟁고에 귀기울이며 지속적으로 발전시켜 나가도록 하겠습니다.

#### 4. 독자를 믿습니다.

한빛미디어의 eBook only 서비스는 구매한 전자책을 다양한 IT기기에서 자유롭게 활용하실 수 있게 DRM-Free PDF 포맷으로 제공됩니다.

이는 독자 여러분과 한빛이 생각하고 추구하고하는 전자책을 만들어 나가기 위해, 독자 여러분이 언제 어디서 어떤 기기를 사용하시더라도 편하게 전자책을 보실 수 있게 하기 위함입니다.

불법 복제가 많은 우리나라에서 DRM이 없는 한빛 eBook Only 서비스가 성공할 수 있을까 걱정해 주시는 분들도 많습니다. 그러나 한빛은 우리 독자 여러분이 지적 창조물과 저작권을 존중해 주실거라 믿습니다.

지금 보시는 전자책에 소유권한을 표시한 문구가 없거나 타인의 소유권한을 표시한 문구가 있다면 위법하게 사용하고 계실 가능성이 높습니다. 이 경우 저작권법에 의해 불이익을 받으실 수 있습니다.

구입하신 도서는 사본을 보관하지 않는다는 조건으로 다양한 기기에 사용할 수 있습니다. 또한 한빛미디어 사이트에서 구입하신 후에는 횟수에 관계없이 다운받으실 수 있습니다.

한빛미디어 전자책은 인쇄, 검색, 복사하여 붙이기가 가능합니다.

전자책은 오탈자 교정이나 내용의 수정보완이 이뤄지면 업데이트 관련 공지를 이메일로 알려드리며, 구매하신 전자책의 수정본은 무료로 내려받으실 수 있습니다.

이런 특별한 권한은 한빛미디어 사이트에서 구입하신 독자에게만 제공되며, 다른 사람에게 양도나 이전되지 않습니다.

# 차례

## 1부 웹 서비스 기초

0 1	서블릿 컨테이너를 학습해야 하는 이유	02
1.1	웹 애플리케이션 서버의 역할 .....	02
1.1.1	고성능, 고가용성은 웹 애플리케이션 서버의 이해로부터 .....	02
1.1.2	서블릿 컨테이너 .....	04
1.1.3	서블릿 컨테이너의 안내 .....	05
0 2	HTTP 프로토콜의 실제	06
2.1	아파치 TCPSMon .....	06
2.2	HTTP 프로토콜의 간략한 소개 .....	10
2.2.1	하나의 HTTP 메시지를 특정하는 방법 .....	10
2.2.2	메시지의 내용을 서로 구분하는 행 구분자 .....	12
2.2.3	부가 정보(메시지 헤더와 메시지 바디) .....	13
2.2.4	HTTP 메시지의 크기를 미리 알 수 없는 경우 .....	14
2.3	첫 번째 HTTP 메시지 분석 - 청크 인코딩 .....	15
2.3.1	요청 - GET / HTTP/1.1 .....	15
2.3.2	응답 - 청크 인코딩 사용 방식 .....	17
2.4	두 번째 HTTP 메시지 분석 - Content-Length 지정 .....	19
2.4.1	요청 - GET /css.....	19
2.4.2	응답 - Content-Length 지정 방식 .....	21
2.5	매개변수를 이용한 GET 요청 .....	23
2.6	매개변수를 동반한 요청 - POST 방식(x-www-form-urlencoded) .....	28

2.7 FORM은 POST 전용인가?.....	31
2.8 서블릿 컨테이너의 매개변수 처리 – GET/POST 방식의 차이점.....	33
2.9 바이너리 데이터 전송 – multipart/form-data.....	35
2.10 더 생각해 볼 문제.....	40

---

0 3 서블릿의 이해	41
3.1 서블릿이란 무엇인가.....	41
3.2 GenericServlet.....	42
3.3 HttpServlet.....	43
3.4 Apache Tomcat.....	47
3.5 웹 애플리케이션 배치.....	52
3.6 더 생각해 볼 문제.....	57

## 2부 서블릿 컨테이너

---

0 4 HTTP 프로토콜 분석기	60
4.1 메시지의 끝은 어디인가.....	60
4.2 HTTP GET 요청 처리기.....	64
4.3 메시지 바디 처리 – Content-Length 인식.....	68
4.4 매개변수의 처리 – 쿼리스트링과 x-www-form-urlencoded.....	74
4.5 성능 개선 1 – 버퍼의 사용.....	83
4.6 성능 개선 2 – 더 나은 I/O.....	93
4.7 더 생각해 볼 문제.....	94

0 5	<b>서블릿 관리자</b>	95
	5.1 웹 애플리케이션.....	96
	5.2 인터페이스를 사용한 컴포넌트와 컨테이너의 분리.....	97
	5.3 HTTP 요청이 서블릿에 가기까지.....	106
	5.4 서블릿 관리자.....	111
	5.5 더 생각해 볼 문제.....	118
0 6	<b>병렬처리</b>	119
	6.1 Stop/suspend와 wait/notify 메서드.....	119
	6.2 스레드 풀의 구성 요소 - jetty 6.x의 경우.....	121
	6.3 java.util.concurrent 패키지.....	127
	6.4 ThreadPoolExecutor의 사용.....	128
	6.5 적정 병렬 진행 수.....	133
	6.5.1 병렬화 가능하지 않은 영역의 존재.....	138
	6.5.2 병렬처리로 인해 부가되는 추가 제약조건.....	138
	6.5.3 컨텍스트 스위칭.....	139
	6.6 더 생각해 볼 문제.....	140
0 7	<b>BIO와 NIO의 비교</b>	142
	7.1 일반적인 프론트엔드 웹 서비스 구성.....	142
	7.2 직관적이고 개념적인 I/O.....	143
	7.3 좀 더 실제 물리적 전송과 근접한 I/O 방법.....	145
	7.4 프론트엔드 서버로서의 서블릿 컨테이너.....	148
	7.5 NIO 기반의 HTTP 프로토콜 상태 기계 구현.....	149
	7.6 더 생각해 볼 문제.....	163

0 8	서버 프로그램으로서의 서블릿 컨테이너	164
8.1	서블릿 컨테이너 분석.....	164
8.2	부팅과정에서 벌어지는 일들.....	175
8.3	생명주기 관리.....	183
8.4	남은 이야기.....	185
8.4.1	Shutdown Hook.....	185
8.4.2	System.out과 catalina.out.....	188
8.5	더 생각해 볼 문제.....	188
3부	남은 주제들	
0 9	Comet - HTTP 알림	191
9.1	단순한 시도 - 풀링.....	192
9.2	생각의 전환 - 스트림 방식.....	193
9.3	Neurasthenia - Comet 지원 서블릿 컨테이너.....	195
9.4	두 개의 연결.....	201
9.5	더 생각해 볼 문제.....	207
1 0	남은 이야기들	208
10.1	자존심과 두려움.....	208
10.2	개발자와 프로그래머.....	209

# 1부

## 웹 서비스 기초

1부에서는 서블릿 컨테이너의 실질적인 구현에 대해 다루기 전에, 왜 우리가 서블릿 컨테이너를 살펴봐야 하는지, 서블릿 컨테이너가 웹 서비스를 구성하는 데 어떤 역할을 하는지 알아보겠습니다.

그다음 예제를 통해 서블릿 컨테이너가 다루는 서블릿과 서블릿이 처리하는 HTTP 프로토콜에 대해 살펴보겠습니다. 이는 2부에서 전개될 서블릿 컨테이너의 기능이 어떤 맥락에서 구현됐는지 이해하고, 서블릿 컨테이너의 구조를 통찰하는 데 도움이 될 것입니다.

**1장** 서블릿 컨테이너를 학습해야 하는 이유

**2장** HTTP 프로토콜의 실제

**3장** 서블릿

# 1 | 서블릿 컨테이너를 학습해야 하는 이유

책을 쓸 때 가장 마지막에 결정해야 하는 것은 처음에 무엇을 쓸 것인가다

- 블레즈 파스칼

## 1.1 웹 애플리케이션 서버의 역할

웹 프로그램은 HTTP 프로토콜로 통신하는 네트워크 프로그램의 일종입니다. 하지만 오늘날의 웹 프로그래머 중에서 본인이 작성하는 프로그램이 네트워크 기반이라는 사실을 자각하면서 구현하는 사람은 그렇게 많지 않습니다. 현재의 웹 프로그래밍 작업이 HTTP 프로토콜을 다루는 코드를 직접 작성하기보다는 웹 애플리케이션 서버로 통칭되는 일종의 미들웨어 제품이 제공하는 실행 환경 위에서 동작하는 코드를 작성하는 경우가 대부분인 점에 기인합니다.

웹 애플리케이션 서버는 클라이언트와 서버 간의 소켓 통신에 필요한 TCP/IP 연결 관리와 HTTP 프로토콜 해석 등의 네트워크 기반 작업을 추상화해 일종의 실행 환경을 제공합니다. 이런 실행 환경에서 웹 프로그램을 작성하는 프로그래머는 웹 애플리케이션 서버에서 제공하는 ‘요청’과 ‘응답’이라는 개념 위에서 구현을 시작합니다. 따라서 웹 프로그래머는 TCP/IP 연결을 직접 생성하고 HTTP 프로토콜을 해석하는 과정을 생략해 웹을 쉽게 구현할 수 있습니다. 하지만 관련 기술을 깊이 알지 못한 채로 웹 애플리케이션 서버가 제공하는 추상화된 API로 네트워크에 접근합니다.

### 1.1.1 고성능, 고가용성은 웹 애플리케이션 서버의 이해로부터

현재는 그야말로 웹 기반이 아닌 곳을 찾기 어려울 정도로 웹 기반 프로그램이 성공적인 시대입니다. 이처럼 웹 기반 프로그램의 저변이 확대될 수 있었던 주요한 요인은 앞에서 이야기한 네트워크와 관련된 복잡한 하부 작업을 웹 애플리케이션

서버에 위임함으로써 웹 프로그래머는 처리하고자 하는 비즈니스 로직에 집중할 수 있게 되었기 때문일 것입니다.

“복잡한 실제 작업을 적절히 추상화해 사용하기 편하게 한다”는 개념은 대부분의 소프트웨어가 추구하는 목표일 것입니다. 당장 임의의 소프트웨어 업체 홈페이지의 소개 글에서 “복잡하고 난해한 하부 구조를 자기네 제품 안에 잘 녹여서 명료한 절차로 제공하므로, 이 제품을 사용하기만 하면 전문적인 지식이 없는 사용자라도 간단한 규칙 몇 개의 조합으로 훌륭한 결과를 얻을 수 있다”는 선전 문구를 찾기 어렵지 않습니다.<sup>01</sup>

웹 애플리케이션 서버가 위와 같은 개념을 성공적으로 충족하면서, 이를 사용하는 웹 기반 프로그래밍은 오히려 별것 아니다는 평판을 얻었습니다. 2000년대 초반 까지만 해도 웹 기반 프로그래밍은 비교적 단순한 작업의 연속이며 깊은 지식보다는 웹 애플리케이션 서버가 제공하는 여러 부가 기능을 폭넓게 아는 것이 더 도움 된다는 의견이 많았습니다. 이것은 웹 애플리케이션 서버가 기본적인 성능과 기능을 보장해주므로, 비교적 쉽게 어느 정도 이상 수준의 프로그램을 만들 수 있다는 시각이 반영된 결과라 할 수 있습니다. 또한, 초기에는 대용량 처리를 다루거나 중요한 기능은 웹 기반으로 제공하려고 시도하지 않았고 단순히 기업의 제품 소개나 제품 사용자 의견 청취 등의 부가적인 영역에 한해 제공됐다는 점도 있습니다.

하지만 웹 기반으로 제공되는 서비스의 영역은 지속적으로 넓어지고 있으며, 규모는 거대해지고 있습니다. 현재 금융, 항공 등의 미션 크리티컬한 영역에까지 웹 기반 시스템을 사용하고 있습니다. 이런 상황에서 웹 애플리케이션 서버가 제공하는 기능의 단순한 조합만으로 웹 서비스를 구현한다면 더욱 높아진 성능과 확장 가능성에 대한 요구를 만족시키기는 매우 어려울 것입니다. 특히 성능과 관련된 문제가

---

01 물론 지금까지 소프트웨어가 계속 개발된다는 사실은 이런 목표가 얼마나 달성하기 어려운 것인가를 반증하는 예일 수도 있습니다.

발생했을 때, 웹 기반 시스템의 하위 레벨 영역인 웹 애플리케이션 서버가 담당하는 부분을 모르고서는 근원적인 원인 규명 자체가 불가능합니다. 웹 애플리케이션 서버의 내부구조와 동작 원리를 이해하는지 못하는지가 웹 프로그램의 고성능, 고 가용성에 대한 요구를 충족시킬 수 있는지 결정한다고 할 수 있습니다.

### 1.1.2 서블릿 컨테이너

엄밀하게 말해 웹 애플리케이션 서버는 Java EE™(이하 ‘Java EE’로 표기함)<sup>02</sup> 명세를 만족시키는 Java™(이하 ‘자바’로 표기함)<sup>03</sup> 구현체를 의미하지만, 웹 프로그래밍을 위한 미들웨어라는 개념이 일반화되면서 자바 이외의 프로그래밍 언어로 작성한 서버도 비슷한 역할을 하면 웹 애플리케이션 서버라고 부릅니다. Java EE Java Platform, Enterprise Edition가 많은 부분을 포용하려다 보니, 그 명세 및 구현체가 지나치게 무거워지고 웹 서비스를 구현하는 방법도 필요 이상으로 복잡해졌다는 반성이 있습니다.<sup>04</sup> 이런 연유로 스프링 등을 필두로 여러 경량 프레임워크가 인기를 얻어, 널리 사용되고 있습니다.

하지만 경량 프레임워크도 Java EE 정의 중 웹 애플리케이션 기술<sup>05</sup> 위에서 동작하는 것이 일반적이며, 이런 웹 애플리케이션 기술에 대한 구현체가 바로 우리가 살펴볼 서블릿 컨테이너입니다. 다시 말해, 서블릿 컨테이너는 웹 애플리케이션 서버 중에서 HTTP 요청을 받아 처리하는 기초 역할을 맡고 있습니다.

대부분의 웹 프레임워크가 제공하는 기능은 서블릿 컨테이너 위에서 동작하는 서블릿, 필터, 이벤트 리스너 등을 적절히 조합해 구현한 것입니다. 따라서 사용자가 웹 프레임워크로 작성한 웹 애플리케이션은 결국 서블릿 컨테이너 위에서 동작합니다.

---

02 Java EE™은 Oracle 사의 트레이드 마크입니다.

03 Java™은 Oracle 사의 트레이드 마크입니다.

04 초기 J2EE 1.2는 정의 문서가 10여 개 정도였는데, 최근 버전인 Java EE™ 6는 30개가 넘는 정의 문서 목록을 확인할 수 있습니다(<http://www.oracle.com/technetwork/java/javaee/tech/index.html>).

05 서블릿, JSP, JavaServer Faces, JSTL 명세 등을 포함합니다. <http://www.oracle.com/technetwork/java/javaee/tech/index.html>

### 1.1.3 서블릿 컨테이너의 안내

아파치 톰캣(Apache Tomcat), 제티(Jetty), 그리즐리(Grizzly) 등은 서블릿 컨테이너로 현재 널리 사용되고 있습니다.<sup>06</sup> 아파치 톰캣은 오랫동안 서블릿 기술 명세에 대한 참조 구현체였으며, 가장 잘 알려진 오픈 소스 계열의 서블릿 컨테이너입니다. 제티는 실험적 시도를 과감하게 도입하는 것으로 유명하며, 특히 Comet에 대한 구현체로도 잘 알려져 있습니다. 그리즐리는 오랫동안 아파치 톰캣이 가진 Java EE 참조 구현체의 역할을 새로 맡은 글래스피시(glassfish.java.net)의 서블릿 프레임워크입니다(새로운 참조 구현체인 글래스피시가 아파치 톰캣이 이룩한 대중성을 이어받을 수 있을지도 지켜볼 만합니다).

서블릿 컨테이너의 주요 목표는 서블릿을 동작시키는 데 있습니다. 따라서 서블릿이 어떤 방식으로 동작하는지를 이해하면 서블릿 컨테이너가 제공해야 하는 기능을 역으로 유추할 수 있습니다. 그리고 이런 서블릿의 목적은 HTTP 프로토콜을 사용해 자바로 웹 서비스를 제공하는 것입니다.

서블릿 컨테이너로 들어가기 전에, 서블릿이 다루는 HTTP 프로토콜에 대해 알아본 다음 서블릿 컨테이너가 다루는 서블릿을 살펴보겠습니다. 이런 상향식 접근을 통해 서블릿 컨테이너가 왜 그런 기능을 제공하는지, 혹은 제공해야만 하는지 더 잘 이해할 수 있을 것입니다.

---

06 톰캣(apache.tomcat.org), 제티(jetty.codehaus.org/jetty/), 그리즐리(grizzly.java.net/)

## 2 | HTTP 프로토콜의 실제

2.0231: 세계의 실체는 단지 형식을 확정할 수 있을 뿐이고, 실질적 속성을 확정할 수는 없다. 왜냐하면 후자는 명제에 의해서만 비로소 묘사되기 때문, 즉 대상들의 배열에 의해서만 비로소 형성되기 때문이다.

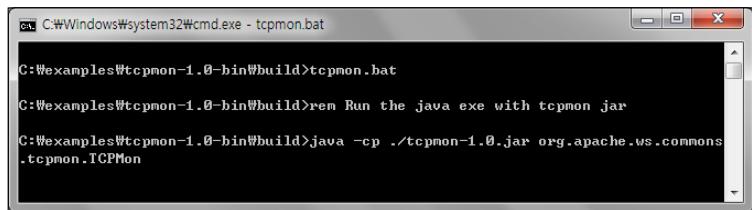
- 루드비히 비트겐슈타인(『논리 철학 논고』 중에서)

이 장에서는 웹 브라우저와 웹 서버 간에 오고 가는 네트워크 패킷 분석을 통해, HTTP 프로토콜에 대해 자세히 살펴보겠습니다. HTTP 1.1은 RFC 2616<sup>01</sup>으로 정의되어 있습니다.

### 2.1 아파치 TCPMon

웹 브라우저와 웹 서버 사이에서 오고 가는 HTTP 패킷을 살펴보기 위해 아파치 TPCMon을 사용하겠습니다. 먼저 아파치 TPCMon 홈페이지(<http://ws.apache.org/commons/tcpmon/>)에서 설치 파일을 내려받아 임의의 디렉터리에 압축을 풁니다(도서에는 examples 디렉터리를 사용하였습니다). 그리고 명령창에서 "tcpmon-1.0-bin\build" 디렉터리 아래에 있는 tcpmon.sh(\*nix 계열)나 tcpmon.bat(windows 계열)를 실행합니다.

그림 2-1 tcpmon.bat 명령어 실행

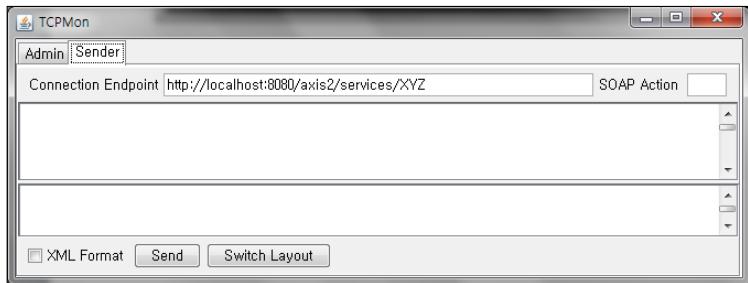


The screenshot shows a Windows Command Prompt window titled 'cmd C:\Windows\system32\cmd.exe - tcpmon.bat'. The command entered is 'C:\examples\tcpmon-1.0-bin\build>tcpmon.bat'. The output shows the command being run and a note: 'Run the java exe with tcpmon jar'. The final command shown is 'C:\examples\tcpmon-1.0-bin\build>java -cp ./tcpmon-1.0.jar org.apache.us.common.TCPMon'.

01 : <http://www.w3.org/Protocols/rfc2616/rfc2616.html>

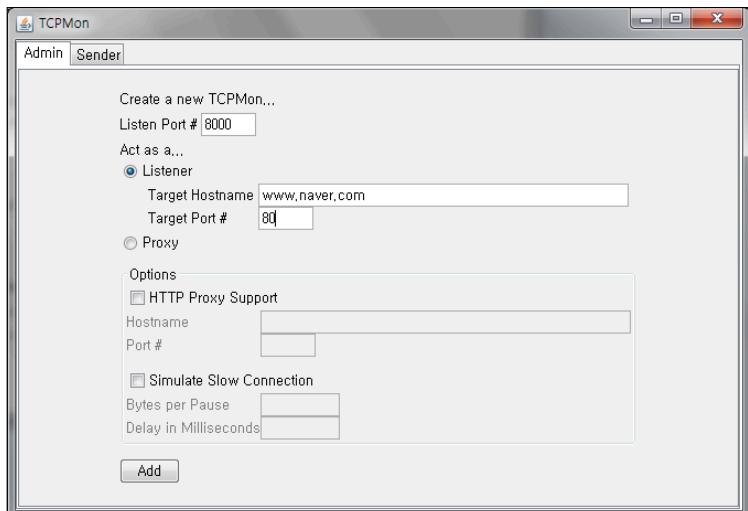
조금 후에 GUI 화면이 생성됩니다.

그림 2-2 TCPMon의 GUI 화면



[Admin] 탭을 선택합니다. 그리고 [Listen Port]에는 8000, [Target Hostname]에는 www.naver.com, [Target Port #]에는 80을 넣은 후 [Add] 버튼을 눌러 새 탭을 추가합니다.

그림 2-3 TCPMon 설정



추가된 [Port 8000] 탭은 다음과 같습니다.

그림 2-4 [Port 8000] 탭이 추가된 TCPMon

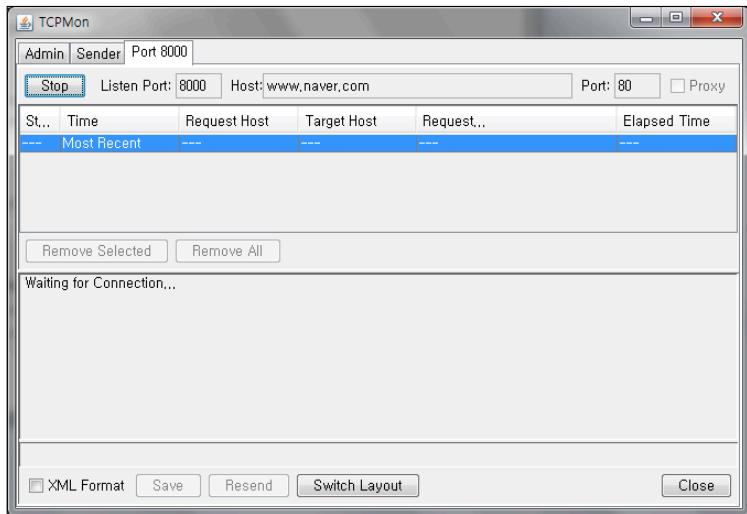
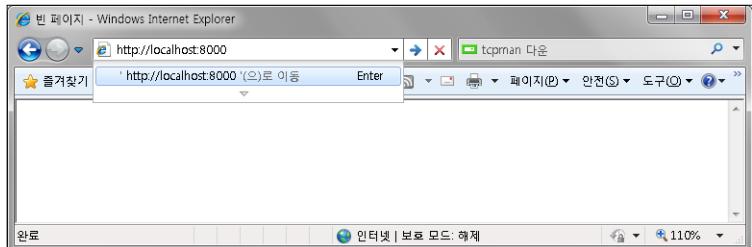


그림 2-4에서 알 수 있듯이, [Listen Port]로 지정된 포트 8000번을 이용해 listen(요청)을 대기합니다. 또한, 포트 8000번을 통해 받은 요청을 그대로 복제한 후, 포트 80번을 이용해 [Host]로 지정된 ‘www.naver.com’ 서버로 전송합니다. 그리고 ‘www.naver.com’ 서버가 반환한 결과를 받아 ‘Waiting for Connection’이라 표시된 하단 창에 나타냅니다.

이런 절차를 거쳐 웹 브라우저가 특정 웹 서버(www.naver.com)에 보낸 요청과 웹 서버가 반환한 응답을 가로채 보겠습니다. 웹 브라우저를 열어 ‘<http://localhost:8000>’을 호출합니다.

그림 2-5 웹 브라우저에서 'http://localhost:8000' 호출



웹 브라우저에서는 Host로 설정했던 'www.naver.com' 서버에 대한 응답, 즉 네이버 메인 페이지가 표시되는 것을 확인할 수 있습니다. 그런데 TCPMon의 [Port 8000] 탭상에서는 다음과 같은 결과가 반환됩니다.

그림 2-6 TCPMon에서 'http://localhost:8000' 호출 결과 확인

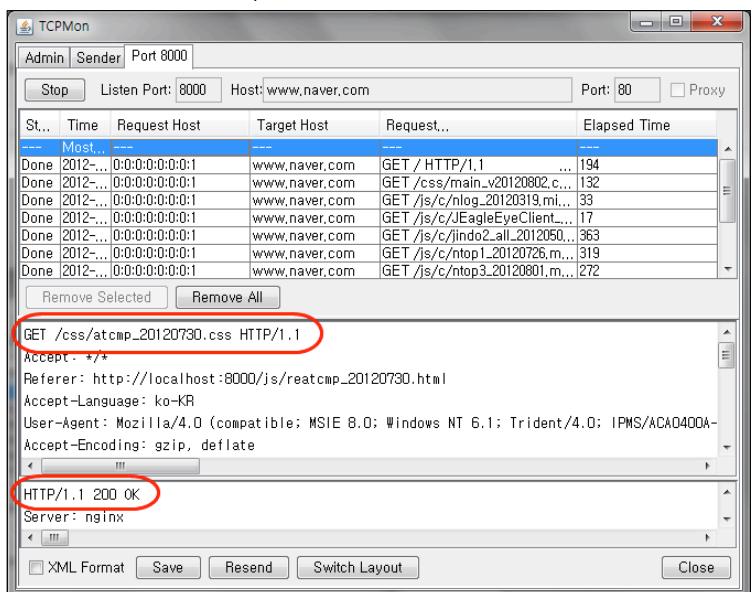


그림 2-6과 같은 결과가 나왔다면, 축하합니다! 여러분은 웹 브라우저와 웹 서버 사이의 통신을 들여다보는 것에 성공했습니다. 상단 테이블의 각 행은 각각 한 번의 HTTP 통신을 나타냅니다. 이 결과에 덧붙여 두 가지를 특별히 언급하고자 합니다.

첫째, 위와 같은 결과는 웹 브라우저에서 [www.naver.com](http://www.naver.com)을 한 번 호출해 발생한 것임에 주목해야 합니다. 이는 최종 사용자에게 하나로 보이는 요청을 처리하기 위해 웹 브라우저와 웹 서버 간에는 수회에 걸친 통신이 발생할 수 있음을 보여줍니다.

둘째, 통신을 위해 사용하는 HTTP 프로토콜로 작성된 실제 데이터를 눈여겨보십시오. 스크린샷 하단에 표시된 ‘GET/css’로 시작하는 문자열과 ‘HTTP/1.1 200 OK’로 시작하는 문자열은 전형적인 HTTP 요청과 응답입니다. 다음 절에서는 HTTP 메시지에 대해 좀 더 자세히 알아보겠습니다.

## 2.2 HTTP 프로토콜의 간략한 소개

기술적인 측면에서 웹 서비스는 HTTP 프로토콜로 메시지를 전송하는 시스템이라 할 수 있습니다. 조금 더 자세히 말하면, 클라이언트가 보낸 메시지는 HTTP 프로토콜에 실려 서블릿 컨테이너에 전송됩니다. 전송된 메시지를 서버 측에서 재구성 하려면 서블릿 컨테이너는 반드시 HTTP 프로토콜 해석 기계를 구현해야 합니다. 이는 HTTP 프로토콜을 이해해야 가능한 이야기입니다. 이제 HTTP 프로토콜이 어떤 구조인지 살펴보겠습니다.

### 2.2.1 하나의 HTTP 메시지를 특정하는 방법

HTTP 프로토콜은 문서를 전송하는 단순한 형태<sup>02</sup>에서 시작됐지만, 웹의 폭발적인

---

02 <http://www.w3.org/Protocols/HTTP/AsImplemented.html>에서 1991년 당시 정의된 초기 HTTP 프로토콜 규약을 찾아볼 수 있습니다.

확대에 대응하기 위해 프로토콜 내용이 확장되면서 현재의 형태<sup>03</sup>로 바뀌었습니다. 현시점에서 HTTP 프로토콜을 전체적으로 바라보면 일관성이 부족한 부분도 있지만, 이것은 하위 호환성을 유지하면서 기능을 추가하기 위해서는 불가피한 것이라 할 수 있습니다. 그중 대표적인 부분이 바로 “어디가 메시지의 끝인가”를 판단하는 기준입니다.

메시지 전송 프로토콜에 반드시 필요한 것은 전송받은 메시지의 끝을 결정하는 것입니다. 사람 간의 대화에서는 말의 끝에 일정한 휴지기를 두어, 자신이 이번 차례에 할 말은 끝났으며 이제 상대방이 말하기 시작함을 기다린다는 암시를 줄 수 있습니다. 하지만 컴퓨터 프로그램이 이런 모호함을 이해하고 통신할 수는 없습니다. 따라서 각각의 상황을 모두 포함하는 명확하고 일반적인 규약이 필요합니다. 이런 규약으로 대표적인 것이 다음에 보낼 데이터의 크기를 미리 전송하여 수신측에게 어디까지 읽어야 하는지를 알려주는 방식과 종료 기호를 서로 약속한 후 해당 기호로 메시지의 끝을 표시하는 방식입니다.

앞으로 보낼 데이터의 크기를 미리 전송하는 방식은 말 그대로 먼저 보낼 데이터의 크기를 전송한 후, 전송한 크기만큼 데이터를 보내겠다고 약속하는 것입니다. HTTP 프로토콜에서는 Content-Length라는 특별한 값의 메시지 헤더를 사용하여 이 방식을 지원합니다. 추후 HTTP 메시지를 설명하면서 자세히 다루겠습니다.

종료 기호<sup>termination symbol</sup>를 사용하는 방식은, 문장의 종료를 나타내기 위해 마치 마침표를 찍듯 매우 직관적이고 간단하게 적용할 수 있습니다. 하지만 이런 방식은 “문장 중에 마침표 기호를 인용하고 싶을 때 어떻게 하는가?”와 같은 질문<sup>04</sup>이 생깁니다.

---

03 <http://www.w3.org/Protocols/rfc2616/rfc2616.html>

04 종료 기호 자체를 문장 중에 사용하고 싶은 때를 의미합니다. 이 문제는 러셀의 이발사 역설, “모든 사람은 스스로 면도를 해서는 안 되며 반드시 이발사에게 가서 면도해야 한다. 그럼 이발사 본인은 어떻게 면도를 하는가?”와 일맥상통하는 면이 있습니다.

이에 대해 탈출 부호(역 슬래시처럼 일상적인 용도로 잘 사용하지 않는 부호)를 추가하는 방법이 있습니다. 하지만 또다시 탈출부호 자체를 메시지 내부에 어떻게 표현할지를 결정해야 합니다. 이렇다 보니 더 간단해서 좋아 보이던 종료 기호 사용 방식이 오히려 메시지 해석을 더 복잡하게(탈출 부호와 같은 예외 규칙의 추가로) 만들 수 있습니다.

메시지 끝에 종료 기호를 표시할 때 고려할 것이 하나 더 있습니다. 문자열 기반의 전송이라면 메시지 내용을 수정하여 메시지 내부에서 종료 기호를 표시(탈출 부호를 앞에 붙인다든가)할 수는 있지만, 대량의 바이너리 데이터를 전송할 때는 이런 종료 기호 방식은 적절하지 않습니다. 일반적으로 전송 전에 크기가 큰 바이너리 데이터를 해석하고 문제가 되는 데이터를 치환하는 전처리를 수행하는 것은 비효율적이기 때문입니다.

앞에서 언급한 몇몇 문제가 발생하지 않는다는 조건에서 종료 기호를 사용하는 방식은 매우 단순하고 직관적입니다. 이러한 이유로 초기 HTTP 프로토콜은 종료 기호를 사용하여 메시지의 끝을 나타냈습니다.

### 2.2.2 메시지의 내용을 서로 구분하는 행 구분자

HTTP Hypertext Transfer Protocol은 말 그대로 하이퍼링크가 포함된 문서를 전송하기 위한 규약에서 시작됐습니다. 따라서 HTTP 요청은 원격 기계에 있는 문서의 위치를 지정하는 방법이라 할 수 있습니다. HTTP/0.9로 대표되는 초기의 HTTP 규약은 원격 기계의 주소와 해당 원격 기계 안에서의 상대적인 문서의 경로를 조합한 문자열로 나타냈습니다. 그렇다 보니 해당 문자열의 끝이 어디인지, 즉 어디가 요청의 끝인지를 표시할 방법이 필요해졌습니다. 이를 위해 아스키코드 13번인 캐리지 리턴 Carriage Return(\r)과 10번인 라인 피드 Linefeed(\n)의 조합 CRLF(\r\n)을 써서 행 구분자를 정의했습니다.

HTTP 요청을 보내는 클라이언트는 요청에 필요한 문자열 뒤에 행 구분자를 붙여 하나의 HTTP 요청 메시지를 완성한 후 전송합니다. HTTP 요청을 받은 서버는 들어온 요청을 한 글자씩 읽다가 행 구분자가 들어오면 요청이 끝난 것으로 인식하고, 그때까지 들어온 데이터를 해석합니다. HTTP 응답은 종료 기호를 별도로 사용하지 않고, 요청에서 클라이언트가 지정한 문서 데이터(HTML 파일)를 서버가 보낸 다음 TCP 커넥션을 종료시켜 클라이언트에 메시지 전송이 완료되었음을 알리는 방법을 사용했습니다.<sup>05</sup> 대부분의 웹 서버는 HTTP/0.9 규약의 메시지 전송 방식을 아직도 잘 지원합니다.

### 2.2.3 부가 정보(메시지 헤더와 메시지 바디)

단순한 문자열 전송 이상의 기능이 요구되면서 HTTP/1.0이 추가로 정의됐습니다. 기존의 단일 행 요청 문자열을 시작행<sup>start-line</sup>이라고 재정의하고, 부가 정보를 전송하는 메시지 헤더와 메시지 바디를 선택적으로 추가할 수 있게 되었습니다.

메시지 헤더는 여러 개일 수 있으며, 가장 마지막에 내용이 없는 메시지 헤더<sup>06</sup>를 사용해 추가 헤더가 없음을 표시합니다.

앞서 메시지의 끝을 표시하는 방식 중 HTTP 프로토콜에서는 앞으로 전송할 데이터의 길이를 특정하는 방식을 특정한 이름의 메시지 헤더를 사용해 지원한다고 언급한 바 있습니다. 메시지 헤더 중 Content-Length라는 헤더값이 지정된 경우<sup>07</sup> 메시지 헤더가 모두 전송된 후에 해당 크기만큼 메시지 바디가 따라나온다는 의미입니다. 서버에서는 빈 메시지 헤더를 읽은 다음부터 Content-Length만큼 더 읽은 후 메시지를 완성합니다. 반면 Content-Length라는 이름의 헤더 값이 지정되

---

05 요청/응답쌍 하나에 무조건 TCP 커넥션 하나가 생성/종료되는 구조입니다. HTTP/1.1에서는 TCP 커넥션 생성, 유지 비용을 절약하기 위해 keep-alive 모드가 추가됩니다.

06 CRLF로 구분된 문자열이 한 행이므로 내용이 없는 메시지 헤더는 바로 앞 행의 행 구분자인 CRLF의 직후에 다시 CRLF가 나오는 형태, 즉 CRLFCRLF를 의미합니다.

07 메시지 헤더는 콜론(:)으로 구분된 이름/값 쌍으로 이뤄집니다.

지 않았다면, 메시지 바디의 길이가 지정되지 않았으므로 메시지 헤더가 종료된 시점에서 HTTP 요청 자체가 종료된 것으로 간주합니다.

#### 2.2.4 HTTP 메시지의 크기를 미리 알 수 없는 경우

Content-Length 헤더를 이용해 메시지 바디의 크기를 미리 알려주기 어려운 경우가 있습니다. 동적으로 결과가 변경되는 경우가 그에 해당됩니다. 외부 DBMS에 들어있는 데이터를 HTTP 메시지 바디에 지정하는 경우를 가정해 이를 살펴보겠습니다.

이때 외부에 있는 전송될 데이터를 실제로 구성해보지 않고 미리 알 수 있는 방법은 없습니다. 예를 들어, HTTP 바디에 출력될 내용을 메모리나 파일로 직접 써봐야만 실제 크기를 알 수 있습니다. 단지 메시지의 크기를 Content-Length 헤더에 지정하기 위해 HTTP 바디를 미리 모두 만든다는 것으로 이것은 명백히 서버 리소스를 낭비하게 됩니다. 이런 방법보다는 전송 버퍼를 사용해 HTTP 바디를 생성하는 도중 버퍼의 크기가 다 차면 해당 내용을 클라이언트로 부분 전송한 이후 다시 버퍼를 재활용하는 편이 더 좋습니다.

또한, 앞선 Content-Length를 사용하는 경우 응답이 요청 직후에 점진적으로 나갈 수 없고 일단 HTTP 바디 크기 계산이 종료된 이후에나 최초의 응답이 나갈 수 있습니다. 하지만 클라이언트로서는 응답 전체를 받지는 못하더라도 응답이 오기 시작했다는 사실 자체가 최소한 자신이 보낸 요청이 서버 시스템에 전달돼 처리된다는 의미입니다. 최대한 최초 응답에 걸리는 시간이 짧은 것이 더 나은 설계라 할 수 있습니다.<sup>08</sup> 그래서 HTTP/1.1에서는 메시지 바디를 점진적으로 전송하는 방법이 추가됐습니다.

---

08 고전적인 예로 큰 이미지를 전송할 때, 전체 파일을 전송받은 후 한 번에 전체 이미지를 보여주는 것과 사용자에게 전송되는 데이터를 일부라도 점진적으로 보여주는 방식 사이의 선택이 있습니다.

첨크 인코딩(chunked encoding)은 메시지 헤더에 transfer-coding 값으로 chunked 를 지정하고 메시지 바디에 첨크라는 단위의 데이터를 나열하는 방법입니다. 앞서 Content-Length 헤더에 전체 데이터의 크기를 지정하는 것과 비슷하게 전송할 데이터의 크기를 미리 전송하고 데이터를 보내는 전략을 메시지 바디에 반복적으로 적용한 것입니다.

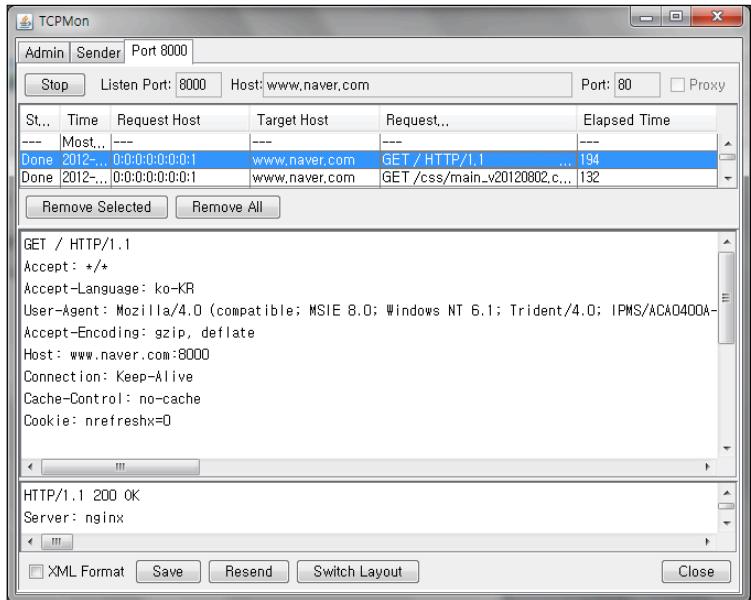
## 2.3 첫 번째 HTTP 메시지 분석 - 첨크 인코딩

TCPMon을 사용해 캡처한 HTTP 요청과 응답을 이용해 앞에서 설명한 내용에 대해 알아보겠습니다. 먼저 Request 칼럼 값이 'GET / HTTP/1.1...'인 요청의 전체 내용을 살펴보겠습니다.

### 2.3.1 요청 - GET / HTTP/1.1

HTTP 메시지의 각 행은 CRLF로 끝난다는 사실을 기억하십시오. 따라서 각 행의 마지막에는 CRLF가 생략되어 있습니다. 첫 번째 행이 시작행이고, 두 번째 행부터 메시지 헤더라고 설명한 바 있습니다. 따라서 이 요청의 시작행은 'GET / HTTP/1.1'입니다.

그림 2-7 GET / HTTP/1.1 요청 결과 : TCPMon



'GET'은 이 요청이 여러 HTTP 요청 형식 중 GET 방식의 요청임을 나타냅니다. 다음 단어인 '/'는 문서가 있는 서버상의 위치를 의미하며, '/'에 대응하는 문서 요청임을 나타냅니다. 마지막 'HTTP/1.1'은 이번 요청이 "HTTP 1.1 정의에 따른다"는 사실을 서버에 알려줍니다.

서버가 이번 요청의 시작행을 끝까지 읽었다면, 첫 번째 CRLF를 만난 순간 메시지 바디를 읽을 필요가 없다는 것을 자동으로 알게 됩니다. 왜냐하면, HTTP GET 방식 요청에는 메시지 바디가 없기 때문입니다. 이제 시작행 이후 CRLF로 구분된 각 행은 개별적인 메시지 헤더가 되며, 마지막 'Connection: keep-alive' 이후 빈 줄은 잘못된 것이 아니라 내용이 빈 행으로 이제 더 이상의 메시지 헤더가 없음을 의미합니다. 다시 말해, 이번 요청의 메시지 완료 조건은 CRLFCRLF까지 읽는 것입니다.

### 2.3.2 응답 - 청크 인코딩 사용 방식

앞에서 설명한 요청에 대한 응답은 다음과 같습니다.

그림 2-8 청크 인코딩을 이용한 요청 결과 ① : TCPMon

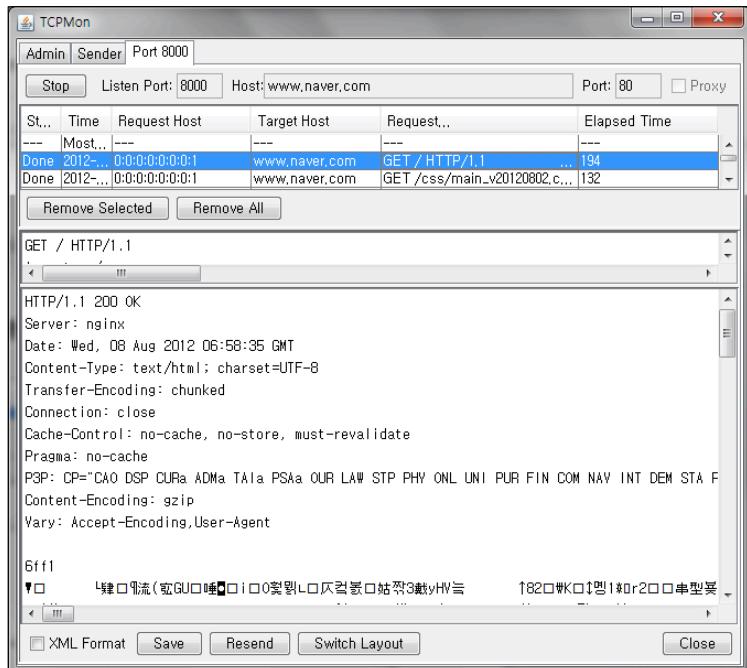
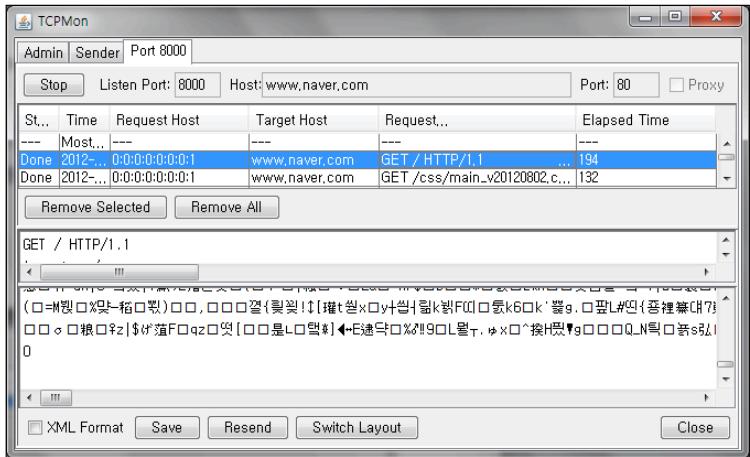


그림 2-9 청크 인코딩을 이용한 요청 결과 ② : TCPMon



서버가 반환한 HTTP 응답의 시작행은 ‘HTTP/1.1 200 OK’입니다. HTTP 요청의 시작행과는 모습이 다릅니다. 사용한 HTTP 버전이 먼저 나오고 응답 상태 코드와 응답 메시지가 차례로 반환됩니다. 이후 메시지 헤더에 응답한 시간, 서버 타입 등 부가 정보가 명시됩니다.

그중 ‘Transfer-Encoding: chunked’에 주목하십시오. 이것은 앞서 언급한 청크 인코딩 방식으로 메시지 바디를 전송할 것임을 표시한 것입니다. 메시지 바디에 ‘6ff1’이 나오고 그 다음 행에 읽기 힘든 문자들이 나옵니다. 청크 길이와 실제 청크 메시지를 표시한 것입니다. 청크 길이는 16진수로 표시하므로 10진법으로 전환하면  $28657 (= 6 \cdot 16^3 + 15 \cdot 16^2 + 15 \cdot 16 + 1)$  길이의 데이터가 뒤따라 나온다는 의미입니다.

청크의 크기나 메시지 바디에 나타날 수 있는 횟수는 제한이 없습니다. 메시지의 끝에 0과 빈 행이 있다는 점을 확인하십시오. 앞서 메시지 헤더의 끝을 표시하기 위해 내용이 없는 메시지 헤더, 즉 CRLF CRLF를 사용한 것과 마찬가지로 청크 인코딩 방

식에서도 크기가 0인 청크를 사용해 더 이상의 청크가 없다는 내용을 나타냅니다.

정리해보면 “Transfer-Encoding 헤더 값이 chunked되면 메시지 바디가 있고, 메시지의 종료 시점은 크기가 0인 청크를 만날 때다”라는 의미가 됩니다.

인코딩이 깨진 문자열처럼 전달된 메시지 청크의 내용이 읽기 힘든 모습인 것은 ‘Content-Encoding: gzip’ 때문입니다. 앞서 GET / HTTP/1.1 요청에서 ‘Accept-Encoding: gzip, deflate’가 있었다는 것을 확인하십시오. 이것은 웹 브라우저가 “나는 gzip이나 deflate 형식도 받아들일 수 있다”는 것을 메시지 헤더를 사용해 서버로 알려준 것입니다. 따라서 서버가 메시지 바디를 gzip으로 압축하고 Content-Encoding 헤더에 gzip을 지정한 것입니다. 이렇게 gzip으로 압축된 메시지이므로 각 청크의 내용이 읽지 못하는 형식으로 나타난 것입니다.

웹 브라우저에서는 Content-Encoding 헤더의 내용을 기반으로 받은 메시지 바디를 압축 해제해 화면을 그립니다. 처음 살펴본 HTTP 요청은 응답의 길이가 고정적이지 않아 청크 인코딩 방식을, 네트워크 전송량을 최소화하기 위해 압축을 사용했습니다.

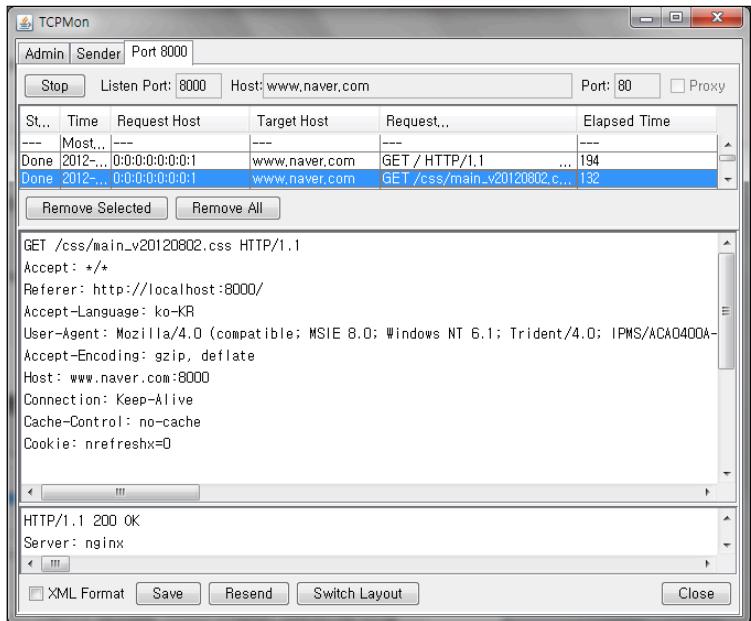
다음 절에서는 TCPMon상에서 나타난, 명시적으로 요청하지 않았던 요청/응답 쌍이 어떤 이유로 나타나는지를 알아보겠습니다.

## 2.4 두 번째 HTTP 메시지 분석 - Content-Length 지정

### 2.4.1 요청 - GET /css.....

이 요청은 원래 웹 브라우저 주소창에 수행한 요청이 아닙니다. 하지만 Referer 헤더에 ‘<http://localhost:8000/>’가 있는 것으로 보아 우리의 요청 때문에 간접적으로 수행됐을 것이라고 유추할 수 있습니다. 그렇다면 ‘GET / HTTP/1.1’ 요청에 대응되는 응답 내용에 이번 ‘GET /css/.....’를 촉발시킨 게 있다는 결론에 다다릅니다.

그림 2-10 GET /css..... 요청 결과 화면



'GET / HTTP/1.1' 요청으로 생성된 웹 브라우저 화면에서 소스 보기로 HTML 코드를 확인해 보겠습니다. 원칙적으로는 해당 요청에 대한 응답에서 바로 확인할 수 있으나, gzip으로 압축된 상태이므로 웹 브라우저의 소스 보기로 확인합니다.

<head> 태그 안에 <link> 태그를 확인할 수 있습니다. 해당 태크의 href 속성 값에 '/css/…'가 있습니다.

그림 2-11 웹 브라우저 화면에서 HTML 코드 확인

The screenshot shows a Microsoft Internet Explorer window with the URL 'http://localhost:8000/'. The title bar says 'http://localhost:8000/- 웹본'. The menu bar includes '파일(F)', '편집(E)', and '형식(O)'. The main content area displays the raw HTML code of a webpage. The code includes standard head elements like meta tags for character encoding, viewport, and compatibility, as well as a link to a favicon and a title. It also contains a script block that includes conditional comments for Internet Explorer, setting document.domain to 'naver.com' and history.navigationMode to 'compatible', and defining a variable nsc with the value 'navertop.v3'. The code is numbered from 1 to 18.

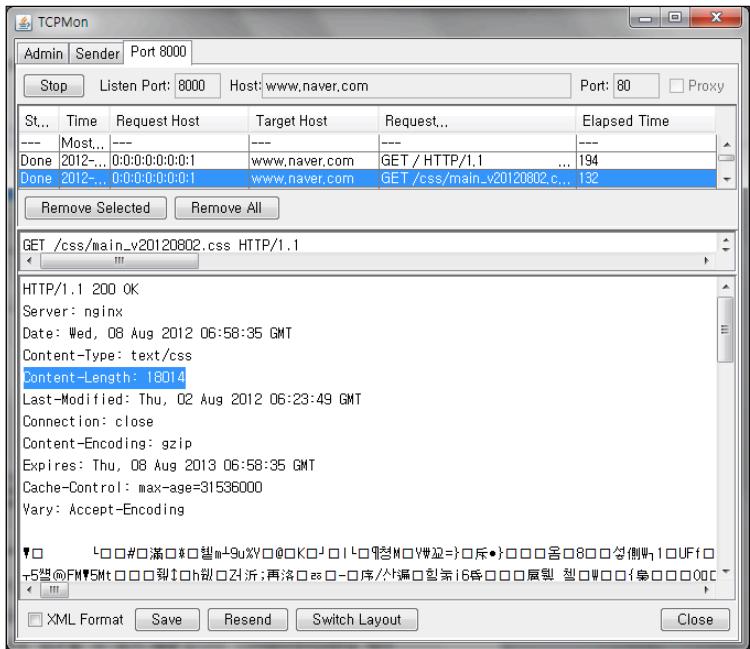
```
1 <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
2 <html xmlns="http://www.w3.org/1999/xhtml" lang="ko" xml:lang="ko">
3 <head>
4 <meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
5 <meta http-equiv="Content-Script-Type" content="text/javascript" />
6 <meta http-equiv="Content-Style-Type" content="text/css" />
7 <meta http-equiv="X-UA-Compatible" content="IE=edge" />
8 <meta name="viewport" content="width=880" />
9 <link rel="shortcut icon" type="image/x-icon" href="http://static.naver.net/www/favicon.ico" />
10 <title>네이버 :: 나의 경쟁력, 네이버</title>
11 <link rel="stylesheet" type="text/css" href="/css/main_v20120802.css"/>
12 <!--[if IE]><style type="text/css">.cast_list li{white-space: normal;}</style><![endif]-->
13 <script type="text/javascript">
14 //<![CDATA[
15 document.domain = "naver.com";
16 history.navigationMode = 'compatible';
17 var nsc = "navertop.v3";
18 //]]&gt;</pre>
```

정리해 보면, 먼저 웹 브라우저가 'GET / HTTP/1.1' 요청을 서버로 전송했고, 서버는 그에 대해 위와 같은 내용의 HTML을 반환했습니다. 웹 브라우저는 자신의 화면에 내용을 보여주려고 서버에서 받은 응답을 해석하던 중 원격 리소스를 표시하는 link 태그를 발견합니다. 해석을 완료하려고 웹 브라우저는 href 속성으로 지정된 '/css/.....' 파일을 다시 보내라는 요청을 자동으로 서버에 전송했습니다. 이렇게 원격 참조 링크가 유발시킨 전송 내역이, 앞서 TCPMon상에 여러 번 반복해서 나타난 요청/응답 목록입니다.

#### 2.4.2 응답 - Content-Length 지정 방식

이번 응답에는 Content-Length 헤더가 지정되었습니다. 이것은 메시지 바디의 크기를 응답으로 생성할 때 결정 가능했다는 의미입니다. 이번 요청은 스타일 시트 파일에 대한 요청이므로 대부분 서버상에서 파일로 접근할 수 있기 때문입니다. Content-Encoding 헤더 값이 gzip이므로 해당 CSS 파일을 gzip 압축 이후 해당 크기를 Content-Length 헤더 값으로 지정하고, 압축된 내용을 메시지 바디로 전송한 결과를 볼 수 있습니다.

그림 2-12 Content-Length 지정 방식을 이용한 요청 결과



주의해서 봐야 하는 메시지 헤더는 Connection입니다. Connection 헤더는 전송에 사용한 소켓 연결을 이번 요청/응답 전송이 완료된 이후 끊을 것인지(close) 아니면 일정 시간 유지해 다음 요청에 대비할 것인지(keep-alive) 여부를 지정하는 데 사용합니다. 앞선 요청에서 해당 헤더 값에 keep-alive를 사용했습니다. 반면, 응답에서는 Connection 헤더 값이 close로 지정되었습니다. 따라서 요청에선 keep-alive를 지정하고 소켓 연결을 유지할 것을 요구했더라도 서버에서 응답에 Connection: close를 지정하고 연결을 끊어버릴 수 있다는 점을 확인하십시오.

TCP 연결은 상당히 비싼 자원입니다. 따라서 대부분의 경우에 한 번 맺은 소켓 연결을 일정 시간 유지하면서 재활용의 기회, 즉 다음 요청이 올 때를 대비하는 것이

합리적입니다. 그럼에도 이렇게 연결을 유지하지 않는 것은 다음과 같은 이유 때문입니다.

CSS 파일처럼 특정 사이트 내에서 반복적으로 사용되는 정적 파일의 경우, 웹 브라우저 캐시와 같은 클라이언트 사이드 캐시에 한 번 적재되면 다시 서버로 요청이 올 가능성이 낮습니다.<sup>99</sup> 연결 하나에 요청/응답이 한 번만 수행되는 것이 대부분이라면, 가능한 요청처리 수는 서로 다른 클라이언트와의 연결 횟수에 비례해 증가할 것입니다. 그런데 서버가 동시에 사용할 수 있는 소켓 수는 한정되므로, 단위 시간 내에 연결 횟수를 최대화하려면 각 소켓이 연결된 시간을 최소화해야 합니다. 따라서 한 번 수행된 연결을 서버에서 바로 연결을 끊어버리는 것이 더 낫습니다.

남은 요청/응답은 ‘GET /css/…… 요청/응답’과 크게 다르지 않으므로 따로 설명하지 않습니다.

## 2.5 매개변수를 이용한 GET 요청

이제 고정된 URL 요청에 매개변수를 추가해 서버에 부가 정보도 같이 전달하는 경우에 대해 살펴보겠습니다. 일반적으로 웹 브라우저 주소창에 넣는 URL의 뒤에 '? 이름=값&이름2=값2'의 형태로 추가하면 URL과 같이 서버로 전송됩니다. 이 추가 정보를 쿼리스트링 Query String이라고 합니다. 그렇다면 이 요청은 어떻게 들어갈까요? TCPMon을 사용해 내용을 확인해 봅시다.

다시 [admin] 탭에서 다음과 같이 정보를 입력하여 이번에는 endofhope.com을 9000번 포트로 돌려서 확인해 보겠습니다.

[Port 9000] 탭은 그림 2-14와 같습니다.

---

99 해당 사이트 내에서 브라우징이 일어나는 동안 반복해서 호출되므로 지속적으로 캐시 히트(cache hit)이 일어날 것이라고 기대할 수 있습니다.

그림 2-13 TCPMon의 [Admin] 탭 설정

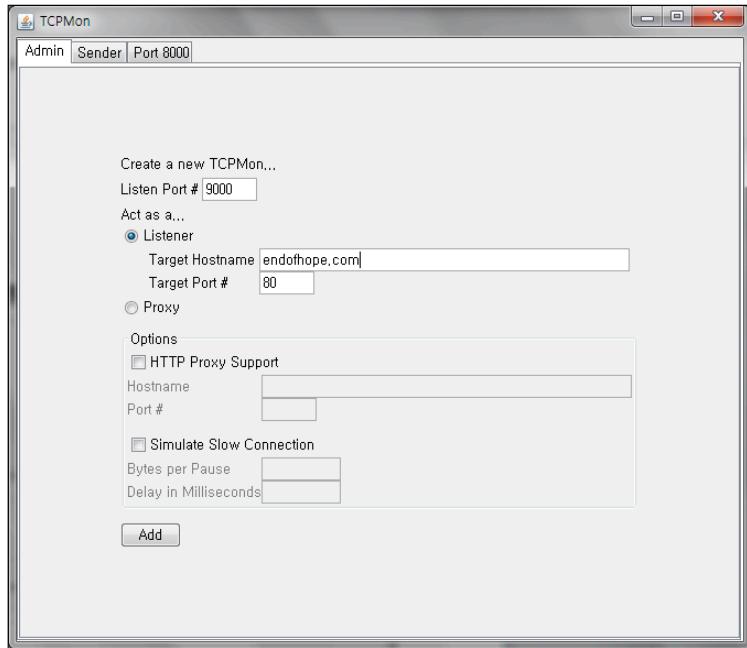
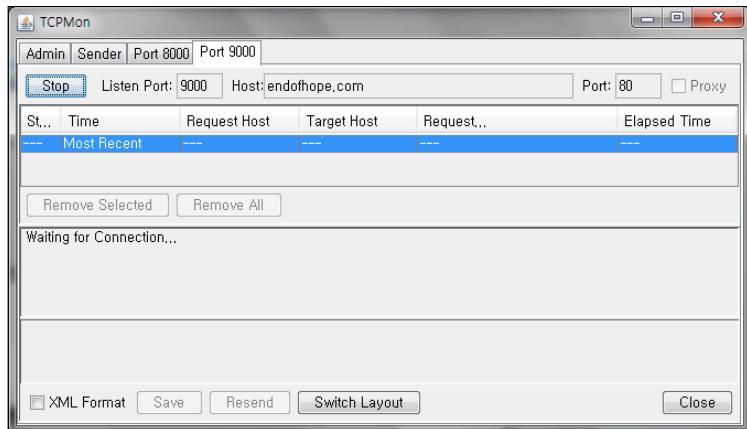


그림 2-14 TCPMon의 [Port 9000] 탭



웹 브라우저에서 'http://localhost:9000/info.cgi?a=b&c=d'를 입력합니다. 이 예제에서 쿼리스트링은 '?a=b&c=d' 부분입니다.

그림 2-15 매개변수를 이용한 GET 요청

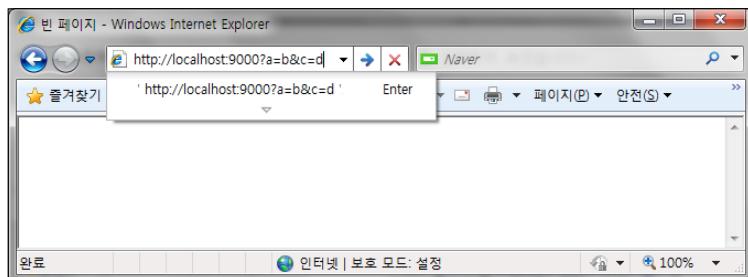
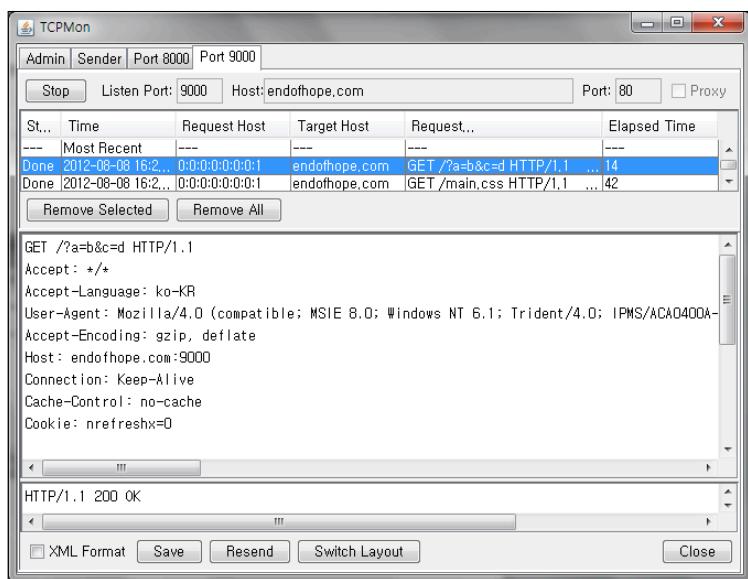


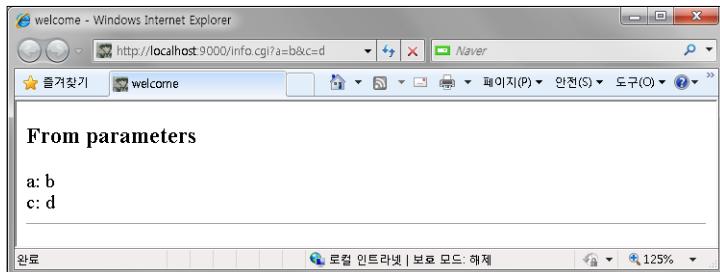
그림 2-16 매개변수를 이용한 GET 요청 결과 : TCPMon



먼저 GET 방식으로 웹 브라우저에서 서버로 데이터가 넘어간다는 사실을 확인할 수 있습니다. 요청 시작행 Request-Line의 URL 부분에 추가한 매개변수 a, c가 전송되는 것을 볼 수 있습니다. 그리고 서버에서는 ‘?’ 문자를 사용해 쿼리스트링을 파악한 후 ‘&’로 끊어내 매개변수를 나누고 ‘=’로 이름/값 쌍을 분리합니다.

endofhope.com/info.cgi는 요청에 들어온 파라미터 parameter를 표시하는 echo 서비스입니다. 위의 파라미터로 전송하면 응답은 다음과 같습니다.

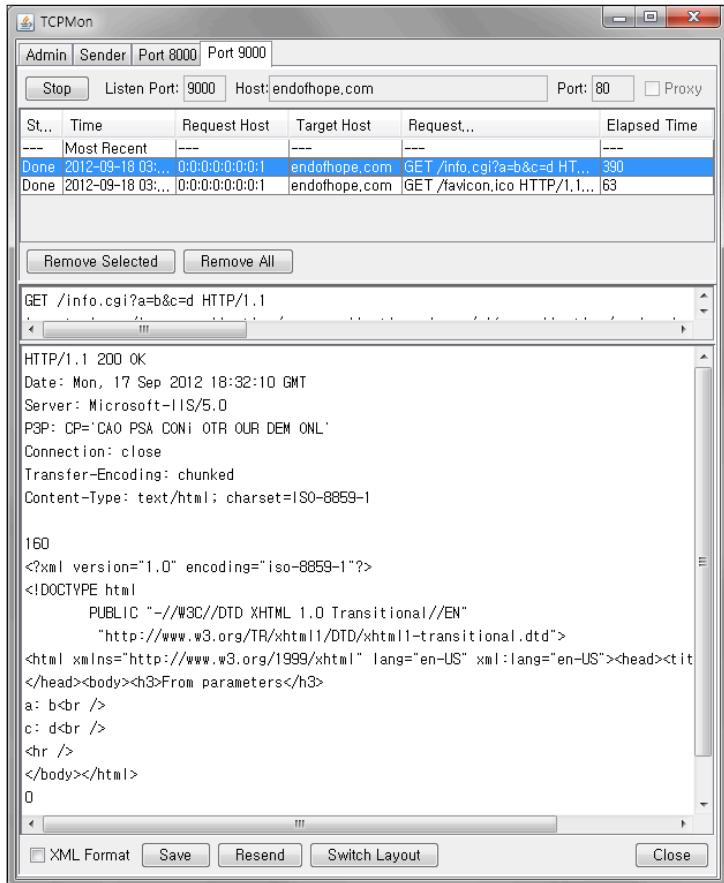
그림 2-17 파라미터 전송에 대한 응답 : 웹 브라우저



응답 내용에 대해 살펴보면 결과는 성공했고(200 OK), 이 응답 이후 커넥션은 끊길 것(Connection: close)입니다. 그리고 청크 방식으로 메시지 바디가 올 것(Transfer-Encoding: chunked)이라는 사실을 헤더 값을 통해 알 수 있습니다.

이하는 ‘16진수 크기 + 청크’의 반복으로 메시지 바디가 나오며 해당 청크 메시지는 크기가 0인 청크로 종료됩니다.

그림 2-18 파라미터 전송에 대한 응답 : TCPMon



## 2.6 매개변수를 동반한 요청

### - POST 방식(x-www-form-urlencoded)

지금까지 요청은 모두 GET 메서드를 사용한 것입니다. 이제 POST 메서드를 이용한 요청에 대해 알아보겠습니다.

먼저 다음과 같이 내용을 작성해 post.html로 저장합니다.

---

```
<html>
  <head></head>
  <body>
    <form name="formname" method="post" action="http://localhost:9000/
      info.cgi">
      <input type="text" name="name" value="1" />
      <input type="submit" name="submit" value="submit" />
    </form>
  </body>
</html>
```

---

웹 브라우저에서 방금 작성한 post.html을 열어 [submit]을 클릭하면 TCPMon에는 웹 브라우저가 서버로 요청을 전달한 것을 확인할 수 있습니다.

그림 2-19 POST 메서드를 이용한 요청

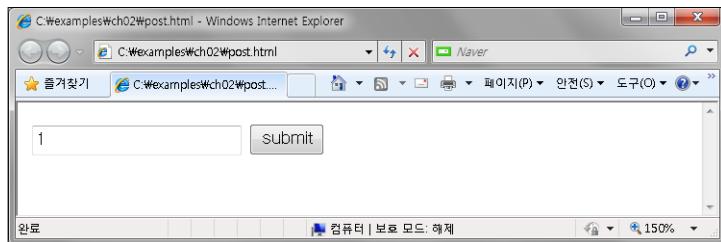


그림 2-20 POST 메서드를 이용한 요청 결과 : 웹 브라우저

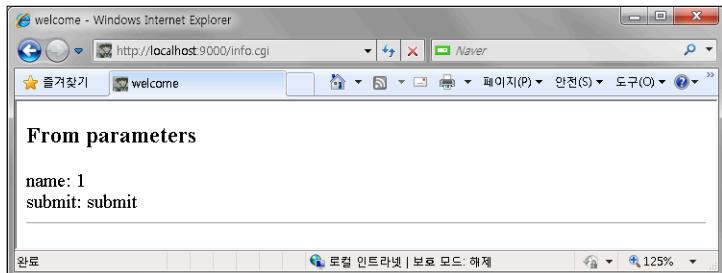
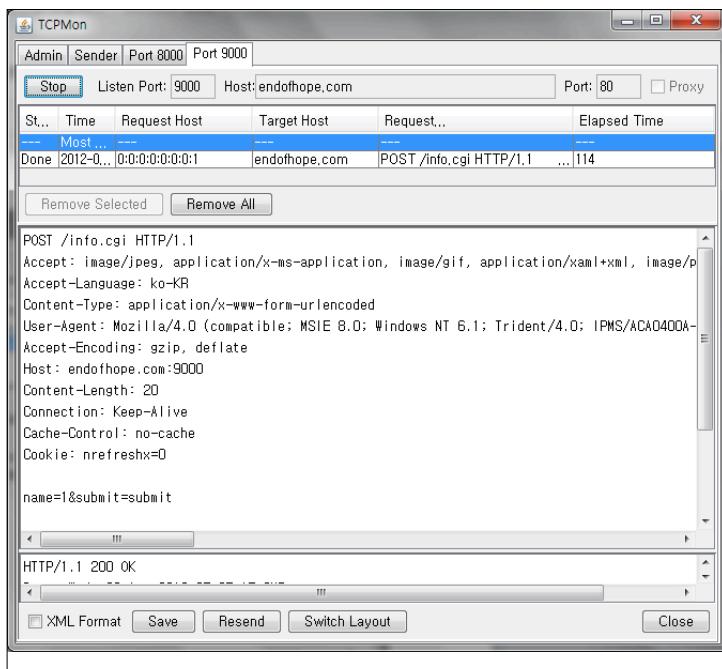


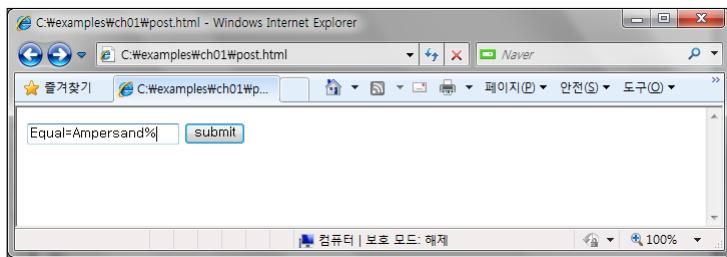
그림 2-21 POST 메서드를 이용한 요청 결과 : TCPMon



HTML Form 엘리먼트에 method 속성을 post로 지정해 매개변수를 전달하면 Content-Type이 application/x-www-form-urlencoded로 지정되고, 메시지 바디에 전달한 매개변수의 이름/값이 '='로 구분되며, 각 쌍은 &로 나뉘어 전달됩니다.

그렇다면 '='이나 '&'가 매개변수의 이름이나 값에 포함될 경우 문제를 예상할 수 있습니다. 먼저 post.html의 파라미터에 '='과 '&'를 넣어서 전달해 보겠습니다.

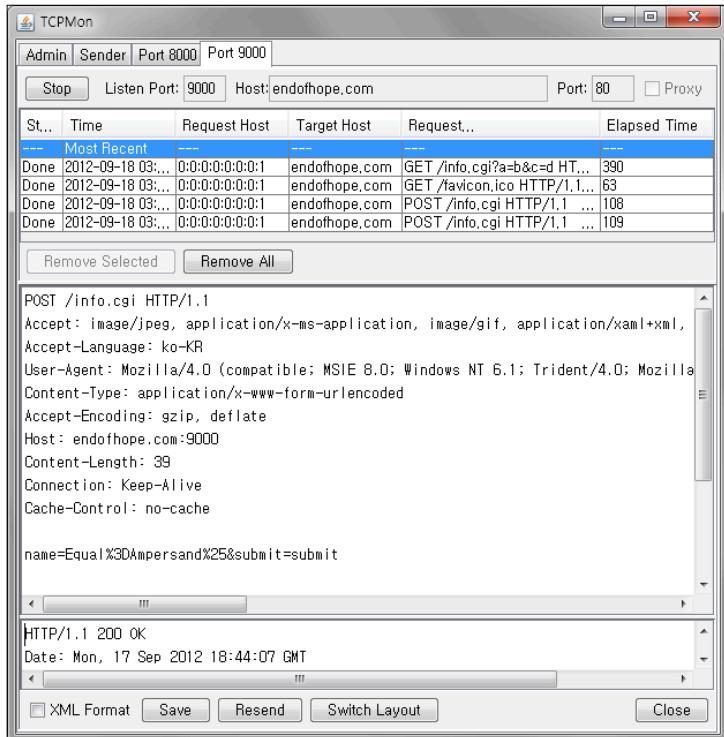
그림 2-22 파라미터 값에 '='과 '&'를 넣어서 전달 : POST 메서드



결과는 그림 2-23과 같습니다.

'='은 %3D, '&'는 %25로 변환돼 전달됩니다. 앞서 제기된 문제를 해결하기 위해 매개변수의 이름/값을 URL 인코딩이라는 특정한 인코딩처리를 합니다. 자바에서 URL.encode/decode 메서드로 비교적 쉽게 구현할 수 있습니다.

그림 2-23 파라미터 값에 '='과 '&'를 넣어서 전달한 결과



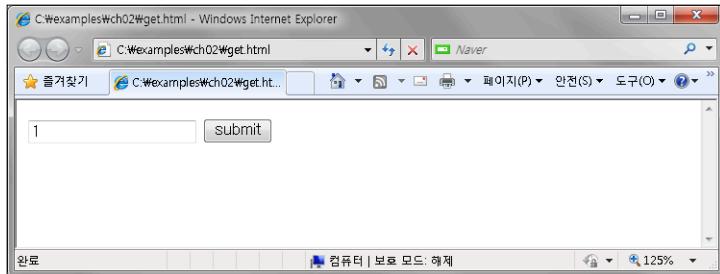
## 2.7 FORM은 POST 전용인가?

post.html에서는 form 엘리먼트를 사용해 post 메서드를 사용했습니다. 그러면 여기서 메서드에 get이라고 써보면 어떨까요? 한 번 시도해 봅시다. 다음과 같이 get.html을 만들고 다시 웹 브라우저에서 submit을 사용해 서버로 어떤 요청이 전송되는지 살펴보겠습니다.

```
<html>
  <head></head>
  <body>
    <form name="formname" method="get" action="http://localhost:9000/
info.cgi">
      <input type="text" name="name" value="1" />
      <input type="submit" name="submit" value="submit" />
    </form>
  </body>
</html>
```

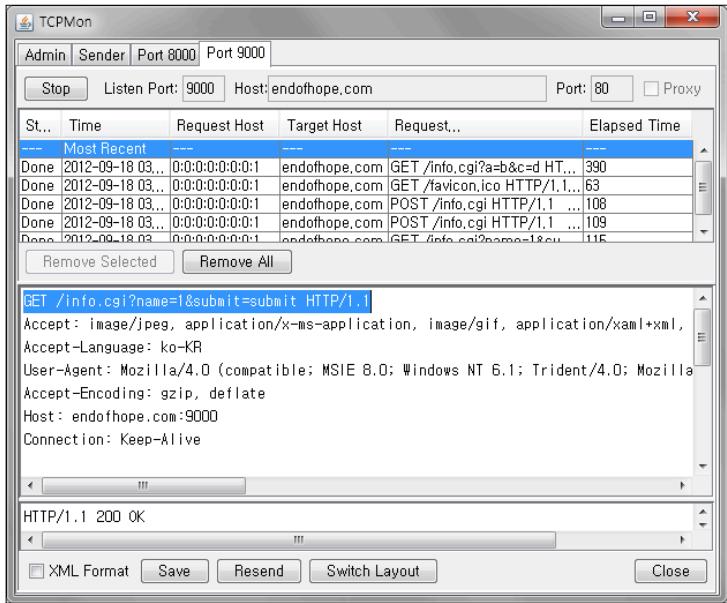
Form의 method가 post에서 get으로 바뀐 것에 주의하십시오.

그림 2-24 submit 메소드를 이용한 요청



Submit한 결과 다음과 같은 요청이 전송됩니다.

그림 2-25 submit 메소드를 이용한 요청 결과



Form을 사용해도 method에 get을 지정하면 쿼리스트링을 사용해 서버로 전송되는 것을 확인할 수 있습니다.

## 2.8 서블릿 컨테이너의 매개변수 처리 – GET/POST 방식의 차이점

자바 환경에서 웹 프로그래머는 HTTP 메서드의 종류<sup>10</sup>에 관계없이 서블릿 컨테이너가 제공하는 API 등을 사용해 웹 브라우저에서 넘어온 데이터에 접근합니다. 이런 편의를 웹 프로그래머에게 제공하기 위해 서블릿 컨테이너는 GET 방식의 요청 일 경우 쿼리스트링을 '?', '&', '=' 등의 구분자로 잘라내 각 매개변수로 할당해 서블릿 API로 접근할 수 있게 합니다.

10 HTTP 메서드는 GET, POST, OPTIONS, HEAD, PUT, DELETE, TRACE가 정의되어 있습니다.

POST 방식의 요청일 경우 데이터의 형태는 GET 방식과 비슷하지만 HTTP 헤더 뿐 아니라 HTTP 바디를 파싱하여 원래 데이터를 획득하는 과정이 필요합니다. 또한 HTTP 메시지 바디가 없어 헤더 정보만 읽어보면 되는 GET 방식 데이터의 경우 서블릿이 동작하기 전에 미리 헤더를 해석해 서블릿 API에서 바로 접근할 수 있게 메모리상에 적재하는 것이 가능합니다. 반면, POST로 넘어오는 매개변수는 해당 매개변수 값을 미리 준비하기보다는 서블릿에서 해당 매개변수를 요청하면 그 때 HTTP 바디를 읽기 시작해 매개변수 적재를 시작하는 것이 일반적입니다(이것은 서블릿 컨테이너의 구현에 따라 다를 수 있습니다).

왜냐하면 HTTP 바디를 사용하는 POST 방식은 매개변수의 값 크기 제한이 없기 때문에 매우 큰 용량의 데이터가 들어올 수 있습니다. 이때 해당 값을 사용하지 않는 경우 웹 브라우저에서는 넘겼으나 서버에서는 해당 값에 접근하지 못하는 경우가 발생합니다.

예를 들어, POST 형식으로 넘어온 1G나 되는 파일을 미리 해석한 후 메모리상에 적재하여 서블릿 API로 접근할 수 있게 준비해 놓았다고 합시다. 그런데 알고 보니 해당 서블릿에서는 사용자가 전달한 값에 접근하는 경우가 극히 드물거나, 혹은 클라이언트가 악의적으로 서버에 부하를 주기 위해 해당 요청을 수행했다면 서버에서는 쓸데없이 메모리/CPU 리소스를 낭비한 결과밖에 되지 않습니다.

그것뿐 아니라 서블릿이 동작하기 전에 매개변수를 준비해야 하므로 앞서 대량의 파일을 메모리에 올리기 위한 HTTP 메시지 바디 파싱이 끝나기 전까지 서블릿은 응답을 시작할 수 없을 것입니다. 따라서 해당 매개변수를 사용하지 않는 요청은 의미 없는 대기 상태를 피할 수 없게 됩니다. 대부분의 서블릿 컨테이너는 이런 상황을 방지하고 빠른 응답을 보장하기 위해 서블릿에서 해당 POST 방식의 파라미터를 직접 서블릿 API를 사용해 접근할 때 메시지 바디에 대한 파싱을 시작하는 전략을 취합니다.

## 2.9 바이너리 데이터 전송 - multipart/form-data

지금까지 이야기한 데이터 전달 방식은 URL에 쿼리스트링을 추가하거나 HTTP 바디에 URL 인코딩 방식으로 처리된 이름/값 쌍을 이용하는 것이었습니다. 그렇다면 바이너리 데이터 전달은 어떻게 처리할까요? 바이너리 데이터 전달의 대표적인 예인 파일 업로드를 통해서 HTTP 프로토콜상에서 어떻게 데이터가 전달되는지 살펴보겠습니다.

다음과 같이 upload.html을 작성하고 웹 브라우저에서 열어 보겠습니다.

---

```
<html>
  <head></head>
  <body>
    <form name="formname" method="post"
          enctype="multipart/form-data"
          action="http://localhost:9000/info.cgi">
      <input type="text" name="name" value="your name" />
      <input type="file" name="filename" value="" />
      <input type="submit" name="submit" value="submit" />
    </form>
  </body>
</html>
```

---

그림 2-26 upload.html 실행 화면

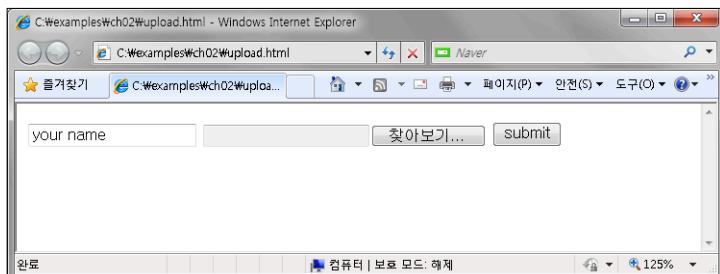
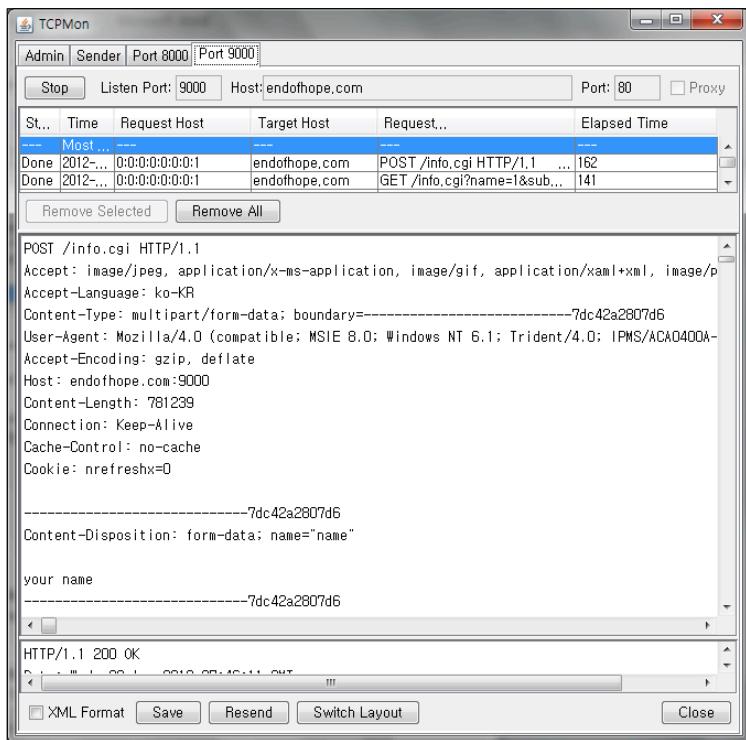


사진 파일을 하나 선택해 [submit]으로 전송하면 다음과 같은 결과를 TCPMon에서 볼 수 있습니다.

그림 2-27 그림 파일 전송 결과 화면



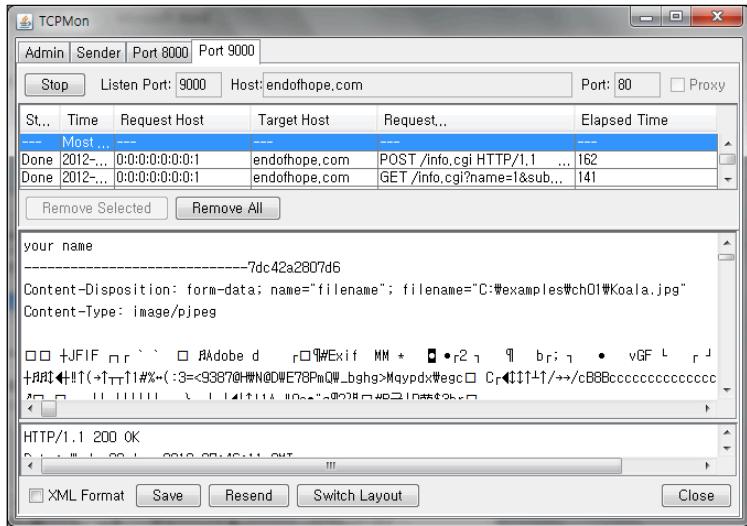
이번에는 POST 요청입니다. upload.html 안에 있는 form의 action에 ‘`http://localhost:9000/info.cgi`’를 지정했고, TCPMon이 해당 요청을 `endofhope.com`으로 다시 전달하는 것은 앞의 예제와 동일합니다. 그런데 Content-Type 헤더 값이 ‘`multipart/form-data; boundary=---`’로 지정돼 있습니다. 이것은 boundary 이름 그대로 구분자입니다.

이 구분자는 메시지를 전달하는 측, 여기서는 웹 브라우저가 생성하는 값이며, HTTP 바디에 없는 값을 생성합니다. 따라서 해당 내용을 받은 서버는 다음과 같이 행동합니다.

1. 요청 시작행을 읽어들인 후, POST 방식임을 알고 HTTP 바디가 들어올 것임을 예상합니다.
2. Content-Length가 781239이라는 것을 보고 HTTP 바디를 얼마나 읽으면 메시지가 완료되는지 파악합니다.
3. ‘Content-Type: multipart/form-data; boundary=-----  
----- 7dc42a2807d6’를 보고 ‘-----  
7dc42a2807d6’ 문자열을 구분자로 각 매개변수가 나눠짐을 알게 됩니다.  
다른 말로 표현하면 이번 메시지에서는 ‘-----  
7dc42a2807d6’가 구분 기호, 일종의 메시지 종료 기호처럼 사용될 것임을 알게 됩니다.
4. 이제 HTTP 바디를 781239바이트가 될 때까지 읽는 도중 ‘-----  
----- 7dc42a2807d6’가 나오면 하나의 매개변수 단위로 재구성합니다.

이를 토대로 메시지 바디 내용을 해석해보면 첫 번째 매개변수는 일반적인 form data 형식으로 넘어왔으며, 이름은 name이고 값은 ‘your name’이라는 값이 전달되었음을 알 수 있습니다.

그림 2-28 'your name' 값이 전달한 결과 화면

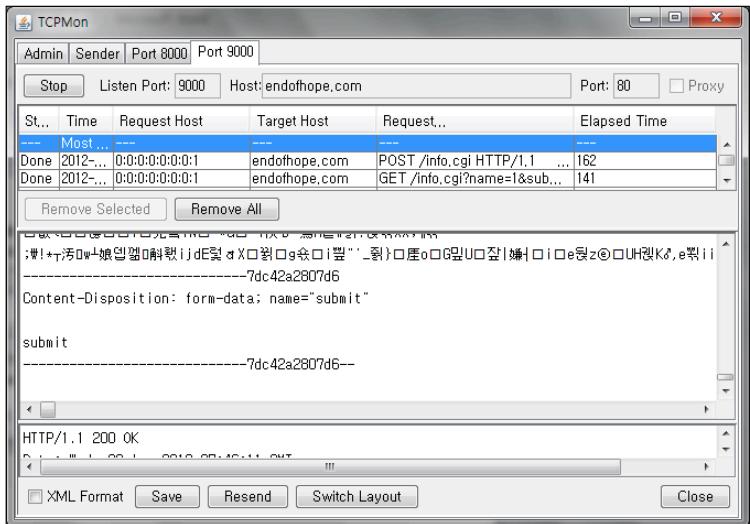


그 다음 부분 역시 동일한 방식으로 해석하면 이번에는 파일명이 Koala.jpg인 image/jpeg 타입의 바이너리 데이터가 전달됩니다.

마지막으로 submit이란 이름의 매개변수가 submit이란 값을 가짐을 알 수 있습니다.

이러한 방식에는 앞에서 살펴본 청크 인코딩 방식처럼 꾸러미가 더 이상 존재하지 않음을 표시하는 크기가 0인 청크가 필요 없습니다. 왜냐하면 헤더에 Content-Length가 지정되어 있으므로 전체 크기를 메시지 바디 해석 전에 알기 때문입니다. 단 하나 주의해야 할 점은 가장 마지막 구분자 '--'에는 '--'가 추가됐다는 점입니다. 이것이 바로 파일 업로드가 웹 브라우저에서 수행되는 방식입니다.

그림 2-29 image/jpeg 타입의 바이너리 데이터가 전달된 결과 화면



서블릿 컨테이너는 서블릿이 InputStream의 일종인 ServletInputStream으로 HTTP 메시지 바디를 읽어들일 수 있게 합니다. 달리 말하면 Servlet 안에서 HttpServletRequest.getInputStream이라는 API로 웹 브라우저에서 넘어온 HTTP 메시지 바디의 내용을 바이트 단위로 읽어들일 수 있게 지원합니다. 서블릿 스펙 2.5까지는 멀티파트 폼 데이터를 지원하는 API가 없었습니다.<sup>11</sup> 따라서 멀티파트 폼 데이터 형식의 HTTP 메시지 바디를 파싱하여 서블릿상에서 메모리상의 객체 형식의 접근을 가능하게 하는 공개/비공개 라이브러리가 여러 개 있습니다. 이런 라이브러리는 HTTP 헤더 값을 미리 읽어 Content-Type이 multipart/form-data인 경우 boundary 값을 알아낸 후 ServletInputStream으로 HTTP 바디를 읽어들입니다. 그다음 name/value쌍으로 해당 데이터를 메모리상에 올려 놓아 서블릿상에서 해당 라이브러리의 API를 사용해 접근할 수 있게 해 파일 업로드

11. 서블릿 스펙 3.0에서 멀티파트 폼 데이터 지원을 위한 Collection<Part> getParts(), Part getPart(String) 등의 API가 추가 되었습니다.

가 가능하도록 합니다.

이 장에서는 지금까지 몇몇 URL([www.naver.com](http://www.naver.com)과 [endofhope.com/info.cgi](http://endofhope.com/info.cgi))에 여러 종류의 HTTP 요청을 보내 서로 다른 응답을 살펴보았습니다. 전달하려는 데이터의 종류와 특성에 따라 조금씩 다른 HTTP 메시지 형식이 사용되는 것을 확인했습니다. 이런 관찰을 바탕으로 앞으로 2부에서는 앞서 열거된 HTTP 메시지를 파싱하여 처리할 수 있는 해석 기계를 작성해 보겠습니다.

## 2.10 더 생각해 볼 문제

1. 이 장에서 사용한 TCPMon은 HTTP 프로토콜을 캡처하는 데 사용됩니다. 이 외에 범용적으로 사용할 수 있는 네트워크 패킷 캡처 프로그램, 예를 들면 Wireshark, Tcpdump 등을 설치하고 TCPMon과 비교해 보십시오.
2. HTTP/1.1이 정의된 RFC 2616 문서를 찾아 읽어보십시오.

### 3 | 서블릿의 이해

아무것도 없이 재활용한 차에서 나온 강철 하나만 가지고, 우리는 전쟁을 치르는 동안 저걸

2,300만 개나 만들었소(그는 의기양양한 미소를 지었다).

그리고 지금도 생산 중이지.

- 맥스 브룩스(『세계대전 Z』중에서)

1997년 SUN 사는 웹 서비스를 위한 기본 인터페이스로 서블릿을 제안했습니다. 또한 서블릿을 배치, 서비스하는 서버 구조로 서블릿 컨테이너의 개념을 전파하기 시작했습니다. 이것은 사용자가 원하는 비즈니스 로직을 서블릿 인터페이스 안에서 구현하고, 서블릿 컨테이너에 해당 서블릿 구현을 배치라는 특별한 절차를 거쳐 등록하면 서블릿 컨테이너가 네트워크 통신, 생명주기 관리, 스레드 기반의 병렬처리를 대행하겠다는 약속이었습니다. 추후 엔터프라이즈 자바빈즈 EJB(Enterprise JavaBeans), 자바서버 페이지 JSP(JavaServer Page) 등의 추가 스펙과 함께 웹 애플리케이션 서버를 정의하는 기반 명세가 됩니다.

이 장에서는 자바 웹 서비스의 기반 구성 요소인 서블릿에 대해 알아보겠습니다.

#### 3.1 서블릿이란 무엇인가

원칙적으로 javax.servlet.Servlet 인터페이스를 구현한 것이 서블릿입니다. 일반적인 자바 독립 실행 프로그램은 public static void main(String[] args) 메서드가 있어서, 해당 메서드가 시작되면 프로그램 수행을 시작하며 main 메서드가 끝나면 프로그램 역시 종료됩니다. 하지만 서블릿은 서블릿 컨테이너 내에 등록된 후 서블릿 컨테이너에 의해 생성, 호출, 소멸이 이뤄집니다. 다시 말해, 서블릿은 독립적으로 실행되지 않고, main 메서드가 필요하지 않으며, 서블릿 컨테이너에 의해 서 서블릿의 상태가 바뀝니다.

때때로 서블릿은 자신의 상태 변경 시점을 알아내 적절한 리소스 획득/반환 등의 처리를 해야 하므로 Servlet 인터페이스에 init/destroy 메서드가 정의됩니다. 다시 말해 서블릿 컨테이너는 서블릿의 생명주기에 따라 서블릿의 상태를 변경하면서 서블릿 인터페이스에 정의된 각 메서드를 불러줍니다.

사용자가 웹 브라우저를 사용해 웹 사이트로 접근하면 해당 요청이 웹 브라우저에 의해 HTTP 프로토콜로 변환돼 해당 사이트를 서비스하는 서블릿 컨테이너로 전달됩니다. 2장에서 설명한 바와 같이, 이 HTTP 프로토콜로 전달된 메시지는 서블릿 컨테이너에서 해석되고 재조합돼 웹 프로그래머가 작성한 서블릿으로 전달되는 과정을 거칩니다. 이때 해당 서블릿의 매개변수로 HttpServletRequest와 HttpServletResponse를 가지는 service 메서드가 호출되며 클라이언트로부터 전달받은 메시지는 HttpServletRequest 인터페이스를 통해 전달됩니다.

## 3.2 GenericServlet

서블릿 인터페이스만 구현하면 서블릿 컨테이너가 생성, 소멸 등의 생명주기 관리 작업을 수행할 수 있습니다. 그런데 서블릿 컨테이너가 서블릿 관리를 위해 필요 한 기능은 서블릿 스펙에 모두 정의돼 있으므로 서블릿 명세는 서블릿에 필요한 구현을 미리 작성해 GenericServlet이란 이름으로 제공합니다. 이 GenericServlet 클래스는 추상 클래스지만 abstract void service(ServletRequest req, ServletResponse res)를 제외하고는 모두 구현된 일종의 서블릿을 위한 어댑터 역할을 제공합니다.

따라서 서블릿을 작성하는 프로그래머들은 반복적으로 동일한 (서블릿 컨테이너의 관리를 받으려고) 서블릿 인터페이스를 구현하는 대신 GenericServlet을 상속 해서 사용합니다. 다음은 GenericServlet의 메서드입니다.

---

```
void destroy()
java.lang.String getInitParameter(java.lang.String name)
java.util.Enumeration<java.lang.String> getInitParameterNames()
ServletConfig getServletConfig()
ServletContext getServletContext()
java.lang.String getServletInfo()
java.lang.String getServletName()
void init()
void init(ServletConfig config)
void log(java.lang.String msg)
void log(java.lang.String message, java.lang.Throwable t)
abstract void service(ServletRequest req, ServletResponse res)
```

---

앞서 살펴본 바와 같이 서블릿 상태 변경 이벤트 리스너와 서블릿 초기화 매개변수 등 정보성 데이터에 대한 접근이 주된 내용입니다.

### 3.3 HttpServlet

일반적으로 서블릿이라 하면 거의 대부분이 HttpServlet을 상속받은 서블릿을 의미합니다. HttpServlet은 GenericServlet을 상속받으며, GenericServlet의 유일한 추상 메서드인 service를 HTTP 프로토콜 요청 메서드에 적합하게 재구현한 것입니다.

다음은 HttpServlet의 메서드입니다.

---

```
protected void doDelete(HttpServletRequest req, HttpServletResponse resp)
protected void doGet(HttpServletRequest req, HttpServletResponse resp)
protected void doHead(HttpServletRequest req, HttpServletResponse resp)
protected void doOptions(HttpServletRequest req, HttpServletResponse resp)
```

```
protected void doPost(HttpServletRequest req, HttpServletResponse resp)
protected void doPut(HttpServletRequest req, HttpServletResponse resp)
protected void doTrace(HttpServletRequest req, HttpServletResponse resp)
protected long getLastModified(HttpServletRequest req)
protected void service(HttpServletRequest req, HttpServletResponse resp)
void service(ServletRequest req, ServletResponse res)
```

---

HTTP 프로토콜의 요청 메서드인 Delete, Get, Head, Options, Post, Put, Trace를 처리하는 메소드가 모두 정의되어 있습니다.

서블릿 컨테이너는 받은 요청에 대해 서블릿을 선택한 후 Servlet 인터페이스에 정의된 service(ServletRequest, ServletResponse)를 호출합니다.<sup>01</sup> 그러면 클래스 상속 위계에 따라 그 처리가 부모 클래스인 GenericServlet에서 자식 클래스인 HttpServlet으로 넘어옵니다. HttpServlet의 service 메서드 내에서는 HTTP 요청 메서드에 의해 여러 doXXX 메서드로 분기돼 처리됩니다.

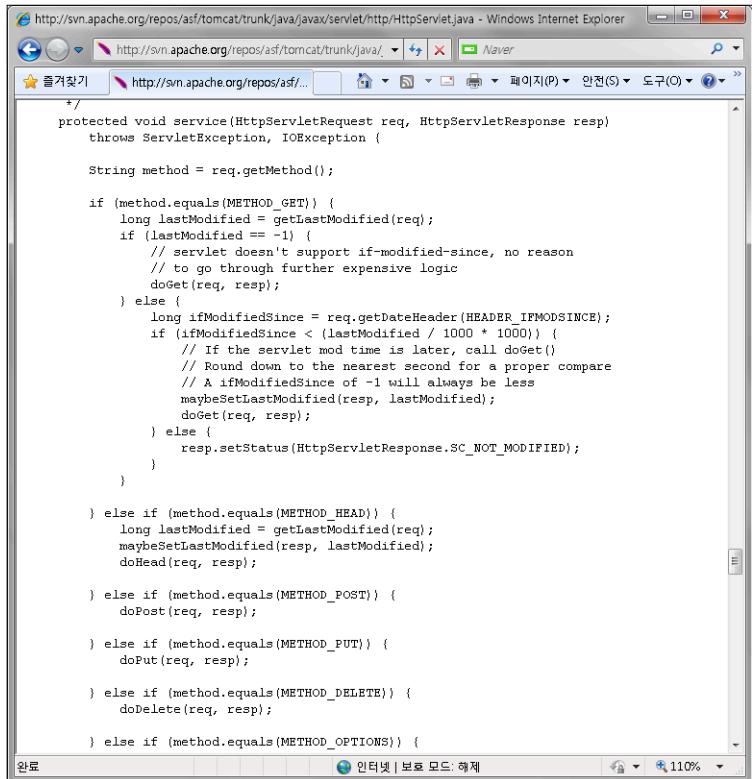
HttpServlet을 상속받아 웹 프로그래머가 작성한 서블릿은 이 doXXX 메서드를 다시 오버라이딩해 HTTP 메서드별로 서로 다른 처리를 수행합니다. 다음은 HttpServlet의 service 메서드 구현의 대표적인 예입니다.<sup>02</sup>

---

01. 뒤에서 설명하겠지만 서블릿 컨테이너는 모든 서블릿을 javax.servlet.Servlet 인터페이스로 처리합니다.

02. <http://svn.apache.org/repos/asf/tomcat/trunk/java/javax/servlet/http/HttpServlet.java>

그림 3-1 HttpServlet의 service 메서드 구현의 대표적인 예



The screenshot shows a Windows Internet Explorer window displaying the source code of the `service` method from the `HttpServlet` class. The URL in the address bar is `http://svn.apache.org/repos/asf/tomcat/trunk/java/javax/servlet/http/HttpServlet.java`. The code implements logic for various HTTP methods: GET, HEAD, POST, PUT, DELETE, and OPTIONS. It handles conditional responses based on last modified times and includes comments explaining its behavior for methods like HEAD and OPTIONS.

```
/*
 * protected void service(HttpServletRequest req, HttpServletResponse resp)
 * throws ServletException, IOException {
 *
 *     String method = req.getMethod();
 *
 *     if (method.equals(METHOD_GET)) {
 *         long lastModified = getLastModified(req);
 *         if (lastModified == -1) {
 *             // servlet doesn't support if-modified-since, no reason
 *             // to go through further expensive logic
 *             doGet(req, resp);
 *         } else {
 *             long ifModifiedSince = req.getDateHeader(HEADER_IFMODSINCE);
 *             if (ifModifiedSince < (lastModified / 1000 * 1000)) {
 *                 // If the servlet mod time is later, call doGet()
 *                 // Round down to the nearest second for a proper compare
 *                 // A ifModifiedSince of -1 will always be less
 *                 maybeSetLastModified(resp, lastModified);
 *                 doGet(req, resp);
 *             } else {
 *                 resp.setStatus(HttpStatus.SC_NOT_MODIFIED);
 *             }
 *         }
 *     } else if (method.equals(METHOD_HEAD)) {
 *         long lastModified = getLastModified(req);
 *         maybeSetLastModified(resp, lastModified);
 *         doHead(req, resp);
 *     } else if (method.equals(METHOD_POST)) {
 *         doPost(req, resp);
 *     } else if (method.equals(METHOD_PUT)) {
 *         doPut(req, resp);
 *     } else if (method.equals(METHOD_DELETE)) {
 *         doDelete(req, resp);
 *     } else if (method.equals(METHOD_OPTIONS)) {
 * }
```

HTTP 메서드를 보고 HttpServlet의 여러 doXXX 메서드로 분기하는 것을 확인할 수 있습니다. 지금까지의 내용을 바탕으로 첫 번째 서블릿을 만들어 보겠습니다. 다음은 전형적인 HelloWorld의 Servlet 버전입니다.

---

```
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import javax.servlet.ServletException;
import java.io.IOException;
import java.io.PrintWriter;

public class HelloServlet extends HttpServlet{
    @Override
    public void init(){
        System.out.printf("%s 이 초기화되었습니다.\n", getServletName());
    }
    @Override
    public void doGet(HttpServletRequest req,
                      HttpServletResponse resp) throws ServletException, IOException{
        PrintWriter out = resp.getWriter();
        out.println("Hello World\n");
    }
    @Override
    public void destroy(){
        System.out.println("이젠 안녕히");
    }
    @Override
    public String getServletName(){
        return "안녕 서블릿";
    }
}
```

---

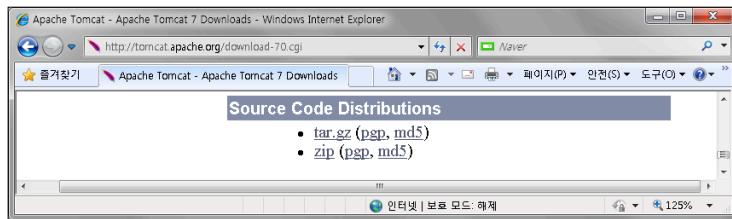
서블릿을 작성했으니 실제 서블릿 컨테이너에 올려서 동작시켜 보겠습니다.

## 3.4 Apache Tomcat

Apache Tomcat은 오픈 소스 서블릿 컨테이너입니다. 소스 코드를 내려받은 후 직접 서블릿 컨테이너를 빌드해 앞서 작성한 우리의 첫 번째 서블릿을 동작시켜 보겠습니다.

- Apache-tomcat : <http://tomcat.apache.org/download-70.cgi>  
(현 시점에서의 최신 버전인 Apache-tomcat 7 버전의 소스를 내려받습니다.)

그림 3-2 Apache-tomcat 내려받기



압축을 풀고 디렉터리 아래에 있는 BUILDING.txt의 내용을 확인합니다. 해당 파일의 내용을 간략하게 정리하면 다음과 같습니다.

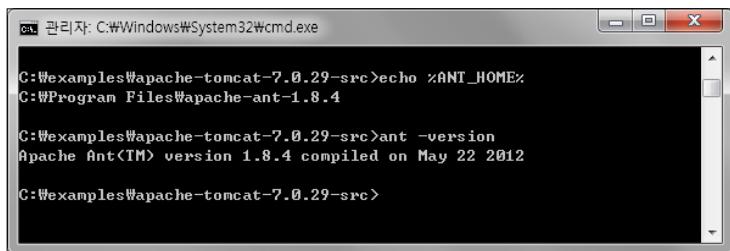
1. JDK를 설치하고 설치된 디렉터리를 JAVA\_HOME 환경 변수로 등록합니다.  
설치 디렉터리의 하위 디렉터리 bin을 PATH 환경 변수에 추가합니다.

그림 3-3 자바의 환경 변수 확인

2. Ant<sup>03</sup>를 설치하고 설치된 디렉터리를 ANT\_HOME 환경 변수로 등록합니다.

역시 설치 디렉터리의 하위 디렉터리인 bin을 PATH 환경 변수에 추가합니다.

그림 3-4 Ant의 환경 변수 확인



The screenshot shows a Windows Command Prompt window titled '관리자: C:\Windows\System32\cmd.exe'. The command 'echo %ANT\_HOME%' is run, showing the path 'C:\Program Files\Apache\ant-1.8.4'. Then, 'ant -version' is run, displaying the output: 'Apache Ant(TM) version 1.8.4 compiled on May 22 2012'.

```
C:\examples\apache-tomcat-7.0.29-src>echo %ANT_HOME%
C:\Program Files\Apache\ant-1.8.4

C:\examples\apache-tomcat-7.0.29-src>ant -version
Apache Ant(TM) version 1.8.4 compiled on May 22 2012

C:\examples\apache-tomcat-7.0.29-src>
```

3. 이미 소스코드를 내려받았으므로 SVN을 사용해 체크아웃할 필요는 없습니다.

하지만 Ant 빌드 도중 필요한 라이브러리를 내려받을 위치를 적절하게 수정하는 것이 좋습니다. build.properties.default 파일의 base.path 값을 기억하기 쉬운 위치로 바꿉니다.

---

```
# Location of GPG executable (used only for releases)
gpg.exec=/path/to/gpg

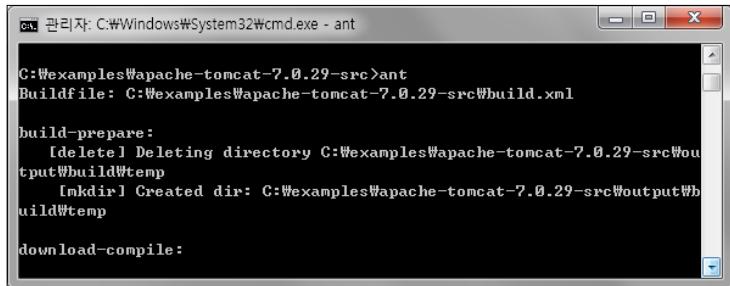
# ----- Default Base Path for Dependent Packages -----
# Please note this path must be absolute, not relative,
# as it is referenced with different working directory
# contexts by the various build scripts.
base.path=/usr/share/java
base.path=c:/examples/tomcat.lib
#base.path=C:/path/to/the/repository
#base.path=/usr/local
```

---

03 <http://ant.apache.org/>

04. 이제 ant 명령으로 필요한 라이브러리를 내려받거나 컴파일합니다.

그림 3-5 라이브러리 내려받기와 컴파일



```
cmd 관리자: C:\Windows\System32\cmd.exe - ant
C:\examples\Apache-tomcat-7.0.29-src>ant
Buildfile: C:\examples\Apache-tomcat-7.0.29-src\build.xml

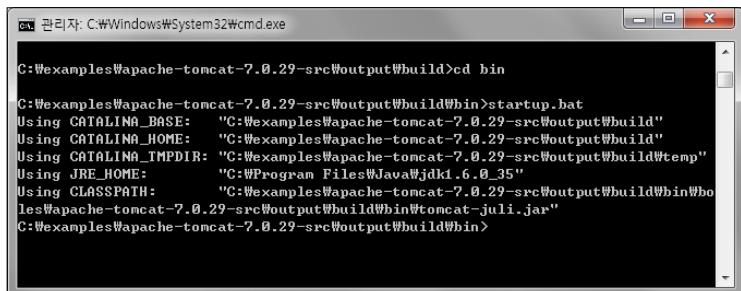
build-prepare:
  [delete] Deleting directory C:\examples\Apache-tomcat-7.0.29-src\output\build\temp
  [mkdir] Created dir: C:\examples\Apache-tomcat-7.0.29-src\output\build\temp

download-compile:
```

Ant를 빌드하면 base.path로 지정했던 디렉터리(c:\examples\tomcat.lib) 아래 여러 라이브러리 코드가 다운로드됩니다. 이 라이브러리는 톰캣 소스를 컴파일하는 데 사용됩니다. tomcat.source 아래에 output 디렉터리가 생성되고, output 디렉터리 아래에 build 디렉터리가 생성되면서 새롭게 빌드된 톰캣 서버가 생성됩니다.

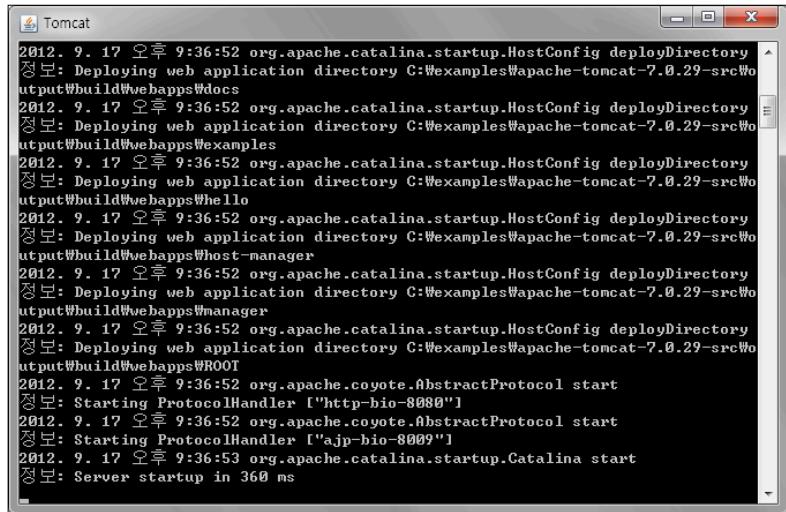
명령행에서 tomcat.source\output\build\bin으로 이동해 startup.bat(windows)나 startup.sh(\*nix)를 실행하면 (windows의 경우 새 창이 뜨면서) 톰캣이 시작됩니다.

그림 3-6 톰캣 서버 실행 ①



```
cmd 관리자: C:\Windows\System32\cmd.exe
C:\examples\Apache-tomcat-7.0.29-src\output\build\bin>startup.bat
Using CATALINA_BASE:  "C:\examples\Apache-tomcat-7.0.29-src\output\build"
Using CATALINA_HOME:  "C:\examples\Apache-tomcat-7.0.29-src\output\build"
Using CATALINA_TMPDIR: "C:\examples\Apache-tomcat-7.0.29-src\output\build\temp"
Using JRE_HOME:        "C:\Program Files\Java\jdk1.6.0_35"
Using CLASSPATH:       "C:\examples\Apache-tomcat-7.0.29-src\output\build\bin\bootstrap.jar"
C:\examples\Apache-tomcat-7.0.29-src\output\build\bin>
```

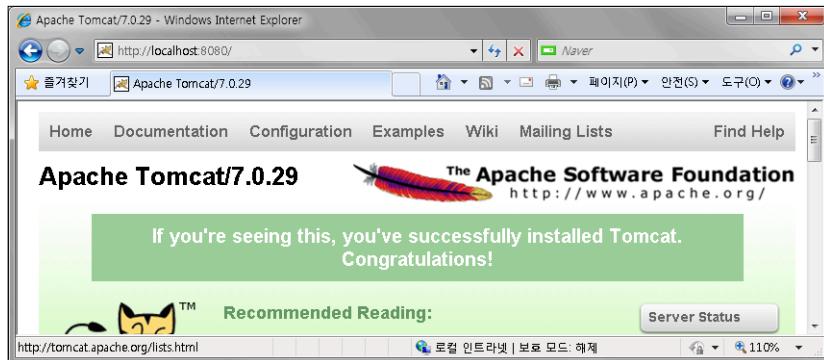
그림 3-7 톰캣 서버 실행 ②



```
2012. 9. 17 오후 9:36:52 org.apache.catalina.startup.HostConfig deployDirectory
정보: Deploying web application directory C:\examples\apache-tomcat-7.0.29-src\w
utput\build\weapps\docs
2012. 9. 17 오후 9:36:52 org.apache.catalina.startup.HostConfig deployDirectory
정보: Deploying web application directory C:\examples\apache-tomcat-7.0.29-src\w
utput\build\weapps\examples
2012. 9. 17 오후 9:36:52 org.apache.catalina.startup.HostConfig deployDirectory
정보: Deploying web application directory C:\examples\apache-tomcat-7.0.29-src\w
utput\build\weapps\hello
2012. 9. 17 오후 9:36:52 org.apache.catalina.startup.HostConfig deployDirectory
정보: Deploying web application directory C:\examples\apache-tomcat-7.0.29-src\w
utput\build\weapps\host-manager
2012. 9. 17 오후 9:36:52 org.apache.catalina.startup.HostConfig deployDirectory
정보: Deploying web application directory C:\examples\apache-tomcat-7.0.29-src\w
utput\build\weapps\manager
2012. 9. 17 오후 9:36:52 org.apache.catalina.startup.HostConfig deployDirectory
정보: Deploying web application directory C:\examples\apache-tomcat-7.0.29-src\w
utput\build\weapps\ROOT
2012. 9. 17 오후 9:36:52 org.apache.coyote.AbstractProtocol start
정보: Starting ProtocolHandler ["http-bio-8080"]
2012. 9. 17 오후 9:36:52 org.apache.coyote.AbstractProtocol start
정보: Starting ProtocolHandler ["ajp-bio-8009"]
2012. 9. 17 오후 9:36:53 org.apache.catalina.startup.Catalina start
정보: Server startup in 360 ms
```

웹 브라우저에서 'http://localhost:8080/'에 접근해 동작을 확인합니다.

그림 3-8 톰캣 서버 동작 확인



지금까지 톰캣 소스를 컴파일해 서블릿 컨테이너를 빌드한 후 HTTP 요청을 처리하는 과정을 확인해 보았습니다. 여기서 주의 깊게 살펴볼 점은, 톰캣 서버를 띄우기 위해 startup.bat 스크립트를 실행하고, 내부적으로 catalina.bat 파일을 호출한다는 것입니다.

다음은 catalina.bat 파일의 일부를 표시한 것이며, 실질적으로 어떤 변수를 사용해 JVM을 호출하는지 확인하기 위해 echo 명령을 추가해 콘솔 화면에 출력했습니다.

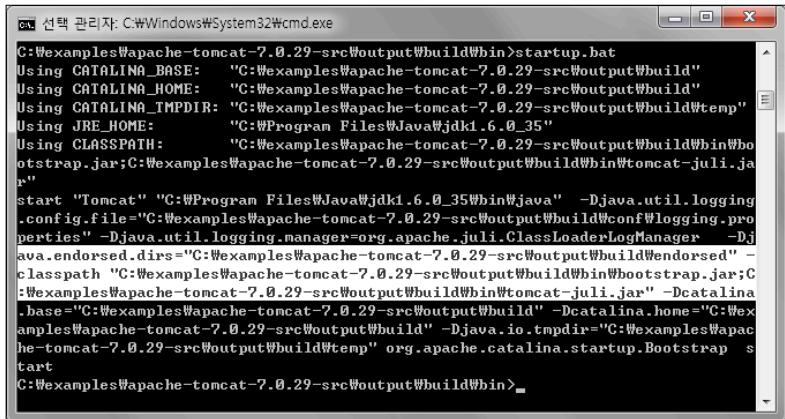
---

```
rem Execute Java with the applicable properties
if not "%JPDA%" == "" goto doJpda
if not "%SECURITY_POLICY_FILE%" == "" goto doSecurity
echo %_EXECJAVA% %JAVA_OPTS% %CATALINA_OPTS% %DEBUG_OPTS%
-Djava.endorsed.dirs="%JAVA_ENDORSED_DIRS%" -classpath "%CLASSPATH%" -Dcatalina.base="%CATALINA_BASE%" -Dcatalina.home="%CATALINA_HOME%"
-Djava.io.tmpdir="%CATALINA_TMPDIR%" %MAINCLASS% %CMD_LINE_ARGS% %ACTION%
goto end
```

---

echo를 추가했으므로 톰캣을 다시 시작하면, JVM을 구동하는 대신 다음과 같이 명령행을 출력합니다.

그림 3-9 JVM을 호출하는지 확인하기 위해 echo 명령 추가



```
C:\examples\apache-tomcat-7.0.29-src\output\build\bin>startup.bat
Using CATALINA_BASE:      "C:\examples\apache-tomcat-7.0.29-src\output\build"
Using CATALINA_HOME:       "C:\examples\apache-tomcat-7.0.29-src\output\build"
Using CATALINA_TMPDIR:     "C:\examples\apache-tomcat-7.0.29-src\output\build\temp"
Using JRE_HOME:            "C:\Program Files\Java\jdk1.6_0_35"
Using CLASSPATH:           "C:\examples\apache-tomcat-7.0.29-src\output\build\bin\bootstrap.jar;C:\examples\apache-tomcat-7.0.29-src\output\build\lib\tomcat-juli.jar"
start "Tomcat" "C:\Program Files\Java\jdk1.6_0_35\bin\java" -Djava.util.logging.config.file="C:\examples\apache-tomcat-7.0.29-src\output\build\conf\logging.properties" -Djava.util.logging.manager=org.apache.juli.ClassLoaderLogManager -Djava.endorsed.dirs="C:\examples\apache-tomcat-7.0.29-src\output\build\endorsed" -classpath "C:\examples\apache-tomcat-7.0.29-src\output\build\bin\bootstrap.jar;C:\examples\apache-tomcat-7.0.29-src\output\build\lib\tomcat-juli.jar" -Dcatalina.base="C:\examples\apache-tomcat-7.0.29-src\output\build" -Dcatalina.home="C:\examples\apache-tomcat-7.0.29-src\output\build" -Djava.io.tmpdir="C:\examples\apache-tomcat-7.0.29-src\output\build\temp" org.apache.catalina.startup.Bootstrap start
C:\examples\apache-tomcat-7.0.29-src\output\build\bin>
```

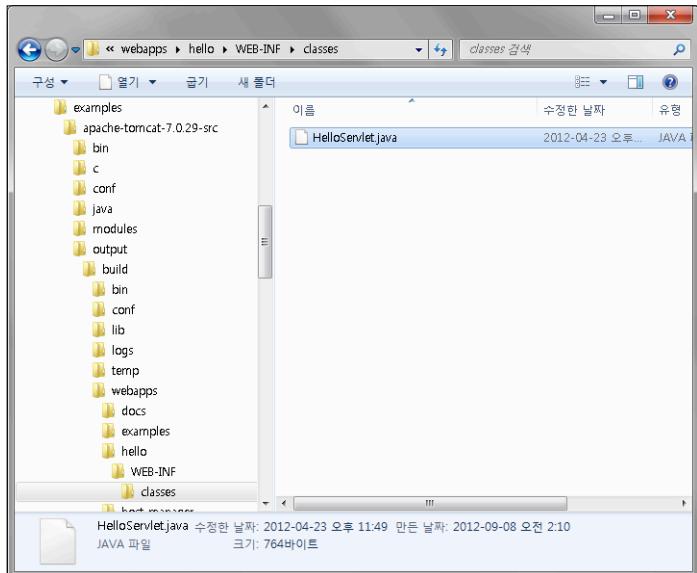
눈여겨 볼 점은 표시된 라인에서 CLASSPATH에 bootstrap.jar와 tomcat-juli.jar만 설정된 것입니다. 다시 말해, 방금 빌드한 톰캣 서버가 지금 당장 호출할 수 있는 클래스는 현재 클래스패스상에 있는 두 개의 jar 파일에 포함돼야 한다는 것입니다.

그렇다면 우리가 만든 HelloServlet를 서버 위에서 동작시키려면 톰캣 서버가 HelloServlet 클래스를 찾을 방법이 있어야 합니다. 또한 어떤 URL로 요청이 들어왔을 때 HelloServlet 서블릿을 실행해야 한다는 사실을 알려주어야 한다는 의미입니다. 서블릿 컨테이너가 제공하는 이런 과정을 배치라고 합니다.

### 3.5 웹 애플리케이션 배치

이제 간단한 웹 애플리케이션 구조를 만들고 HelloServlet을 추가한 다음 톰캣 서버에 배치해 결과를 확인해 보겠습니다. 톰캣 서버는 시작할 때 webapps 디렉터리 아래에서 새로운 웹 애플리케이션을 탐색합니다. 따라서 tomcat 소스 디렉터리(/output/build/webapps) 안에 hello 디렉터리를 생성합니다.

하위 디렉터리 WEB-INF와 classes 디렉터리를 생성하고, 앞서 만든 서블릿을 HelloServlet.java라는 이름으로 classes 안에 저장합니다.

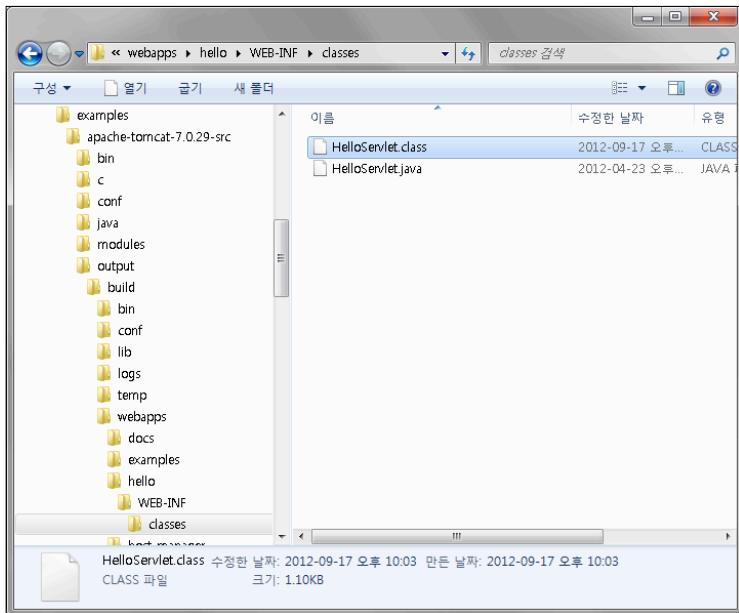


명령창에서 output/build/webapps/hello/WEB-INF/classes/HelloServlet.java 파일을 다음과 같이 컴파일합니다.

- 컴파일 명령: javac -classpath ..\..\..\lib\servlet-api.jar HelloServlet.java

```
C:\examples\apache-tomcat-7.0.29-src\output\build\webapps\hello\WEB-INF\classes>javac -classpath ..\..\..\lib\servlet-api.jar HelloServlet.java
C:\examples\apache-tomcat-7.0.29-src\output\build\webapps\hello\WEB-INF\classes>
```

결과 파일인 HelloServlet.class 파일이 생겼습니다.



이제 어떤 요청이 들어왔을 때 우리가 추가한 HelloServlet을 호출할지 결정하기 위해 hello/WEB-INF 디렉터리 아래에 다음과 같이 web.xml을 만들어 추가합니다.

---

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<web-app xmlns="http://java.sun.com/xml/ns/javaee"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
    http://java.sun.com/xml/ns/javaee/web-app_3_0.xsd"
  version="3.0"
  metadata-complete="true">
  <servlet>
    <servlet-name>helloservlet</servlet-name>
```

```
<servlet-class>HelloServlet</servlet-class>
</servlet>
<servlet-mapping>
    <servlet-name>helloservlet</servlet-name>
    <url-pattern>/Hello</url-pattern>
</servlet-mapping>
</web-app>
```

---

XML 엘리먼트명에서 유추할 수 있듯 helloservlet이란 이름으로 HelloServlet(.class)를 지정하고 /Hello URL 요청에 대해 helloservlet이란 이름으로 지정된 서블릿, 즉 HelloServlet이 처리하게 정의합니다.

이제 웹 브라우저에서 ‘[그림 3-10 ‘\[A screenshot of a Windows Internet Explorer window. The address bar shows the URL 'http://localhost:8080/hello>Hello'. The main content area of the browser displays the text 'Hello World'.\]\(http://localhost:8080/hello>Hello</a>’ 호출 결과</p></div><div data-bbox=\)](http://localhost:8080/hello>Hello</a>’를 호출하면 다음과 같은 내용을 확인할 수 있습니다.<sup>04</sup></p></div><div data-bbox=)

앞서 서블릿이 첫 요청을 처리하기 전에 서블릿 컨테이너는 해당 서블릿의 초기화하는 과정을 거치며 이때 서블릿의 init 메서드를 호출하여 준다고 언급한 바 있습니다. Tomcat 상태창에 다음과 같이 “안녕 서블릿이 초기화되었습니다.”라는 내용이 표시되는 것을 확인할 수 있습니다. 이는 HelloServlet의 init 메서드가 불렸다는 것을 의미합니다.

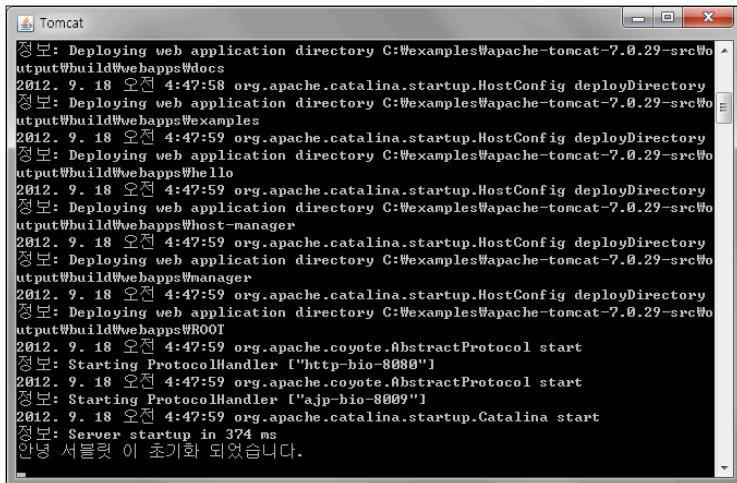
---

04 템켓이 띠 있지 않다면 output/build/bin/startup.bat로 서버를 시작합니다. 만약 서버가 끄지 않는다면 앞서 클래스 패스를 콘솔에서 확인하려고 추가한 catalina.bat의 echo 구문이 제거됐는지 확인합니다.

```
public class HelloServlet extends HttpServlet{
    @Override
    public void init(){
        System.out.printf("%s이 초기화되었습니다.\n", getServletName());
    }
}
```

하락

그림 3-11 서블릿 컨테이너의 서블릿 초기화



지금까지 살펴본 HelloServlet을 서블릿 컨테이너에 추가하는 과정을 요약하면 다음과 같습니다.

먼저 톰캣의 webapps 디렉터리 아래에 hello 디렉터리를 만들고 WEB-INF/classes/HelloServlet.class와 WEB-INF/web.xml을 생성해 hello 디렉터리를 웹 애플리케이션으로 만들었습니다. 또한 톰캣 서버는 webapps 디렉터리 아래 있는 웹 애플리케이션 구조를 자동으로 인식하므로 톰캣이 구동될 때, hello 디렉터리가 웹 애플리케이션으로 간주돼 톰캣 서버에 배치됩니다. 이제 톰캣 서버는 /

hello로 시작하는 요청을 hello 디렉터리가 표현하는 애플리케이션을 이용해 처리합니다.

요청 URL의 /hello 이하 부분은 ‘WEB-INF/web.xml’에 정의된 URL 패턴에 따라 처리할 서블릿이 결정되며, 작성한 web.xml에 따라 HelloServlet이 선택돼 결과적으로 웹 브라우저상에서 HelloWorld 문장이 보이게 됐습니다.

앞에서 톰캣이 시작될 때 jar 파일 두 개만 클래스패스에 지정돼 있으므로 새롭게 만든 HelloServlet을 찾을 방법이 필요하며, 그것이 배치라고 했습니다. 다시 말해 배치는 서블릿을 포함한 웹 애플리케이션을 서블릿 컨테이너에 알려 주는 과정입니다.

웹 애플리케이션은 서블릿 스펙에 정의된 특정한 구조를 가진 디렉터리 혹은 디렉터리 압축 파일이므로 서블릿 컨테이너는 해당 웹 애플리케이션이 가진 서블릿의 위치를 알 수 있습니다. 이번 예제에서는 hello/WEB-INF/classes 디렉터리 내에 있는 HelloServlet.class를 자동으로 로딩한 사실을 알 수 있습니다. 웹 애플리케이션의 구조와 서블릿 컨테이너가 어떤 방법으로 특정 클래스 파일을 런타임에 로딩하는지에 대해서는 5장에서 설명하겠습니다.

지금까지 서블릿 컨테이너가 다루는 HTTP 프로토콜과 서블릿에 대한 대강의 개요를 설명했습니다. 다음 장에서부터는 실질적으로 서블릿 컨테이너의 역할에 대해 자세히 살펴보고 그러한 역할을 위해 필요한 소스코드를 만들어 보겠습니다.

## 3.6 더 생각해 볼 문제

1. HelloServlet의 destroy 메서드를 오버라이딩해 “이제 종료합니다.”라는 문자열을 출력할 수 있게 수정해 보십시오. 이 문자열이 나오게 하려면 어떻게 해야 할지 생각해 보십시오.

2. 서블릿의 멤버 변수로 외부 리소스(예, JDBC connection)를 유지하는 것에 대해 고려해 보십시오. 만약 init/destroy 메서드에서 리소스 획득/반환 과정을 거치게 한다면 해당 리소스에 대한 leak이 발생하지 않는다는 주장의 정당성에 대해 논하여 보십시오.

## 2부

# 서블릿 컨테이너

지금까지는 서블릿 컨테이너에 대한 직접적인 기능과 구조를 설명하기보다 서블릿 컨테이너가 분석해야 하는 HTTP 프로토콜과 서블릿 컨테이너가 관리하는 서블릿에 대해 살펴보았습니다. 이런 접근을 통해 서블릿 컨테이너의 기능을 추측해봄으로써 실제 구현을 좀 더 쉽게 이해할 수 있으리라 기대합니다.

2부에서는 서블릿 컨테이너에 관한 이야기가 본격적으로 시작됩니다. 먼저 HTTP 요청의 의미를 파악하고(4장 HTTP 프로토콜 분석기), 해당 요청을 처리할 서블릿을 찾아내 연결하는 과정을(5장 서블릿 관리자) 살펴봅니다. 그다음 서블릿 컨테이너를 본격적으로 분석해 보겠습니다. 또한, 성능을 좋게 하려면 어떤 구조로 병렬 처리를 하는지, 서블릿 컨테이너는 병렬처리가 왜 중요한지(6장 병렬처리) 알아봅니다. HTTP 요청을 처리하는 두 가지 IO(7장 BIO와 NIO의 비교)의 장단점을 확인해 보겠습니다. 마지막으로 서버 프로그램으로서의 서블릿 컨테이너가 가져야 할 기능에 대해 알아보겠습니다.

**4장** HTTP 프로토콜 분석기

**5장** 서블릿 관리자

**6장** 병렬처리

**7장** BIO와 NIO의 비교

**8장** 서버 프로그램으로서의 서블릿 컨테이너

## 4 | HTTP 프로토콜 분석기

그들의 극장에서는 배우 하나가 막간이 시작된다는 것을 알린 뒤에 어김없이 이렇게  
덧붙이곤 했다. “이제 광고 시간입니다” 나는 무엇 때문에 봉가 인들이 정보의 정확성에  
그토록 집착하는 걸까 하고 오랫동안 생각해 보았다.  
- 움베르트 에코(『세상의 바보들에게 웃으면서 화내는 방법』 중에서)

이 장에서는 웹 브라우저에서 전송된 메시지를 의미 구조로 분석해 HTTP 메시지  
를 생성합니다. 2장에서 우리는 TCPMon을 이용해 몇몇 HTTP 요청과 응답을 확인  
해보았습니다. 이제 실제로 HTTP 요청을 받아들여 해석한 후 적절한 응답을  
보내는 방법에 대해 알아보겠습니다. 2장에서 살펴보았듯 메시지의 끝이 어디인지  
확인하는 것이 가장 첫 번째로 해결해야 할 문제입니다.

### 4.1 메시지의 끝은 어디인가

웹 브라우저에서 요청을 받는 서버를 하나 만들고, 웹 브라우저로 해당 서버에 요청을  
보낸 후 전달받은 HTTP 요청을 처리하는 코드를 하나씩 추가합니다.

---

```
package com.endofhope.scbook.ch04;

import java.io.IOException;
import java.io.InputStream;
import java.io.OutputStream;
import java.net.ServerSocket;
import java.net.Socket;

public class Server_0 {
    public static void main(String[] args) throws IOException{
        Server_0 server = new Server_0();
```

```

        server.boot();
    }

    private void boot() throws IOException{
        serverSocket = new ServerSocket(8000);
        Socket socket = serverSocket.accept();
        InputStream in = socket.getInputStream();
        OutputStream out = socket.getOutputStream();
        int oneInt = -1;
        while(-1 != (oneInt = in.read())){
            System.out.print((char)oneInt);
        }
        out.close();
        in.close();
        socket.close();
    }

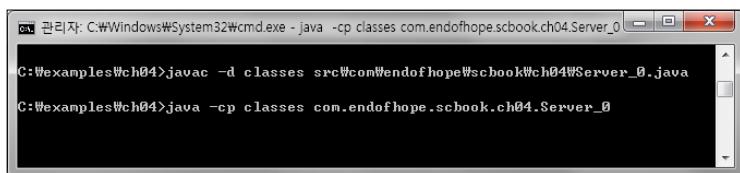
    private ServerSocket serverSocket;
}

```

---

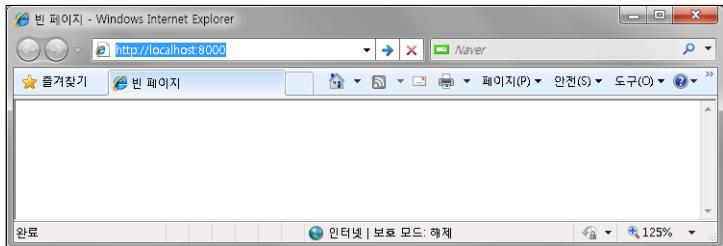
위와 같은 내용의 파일을 ‘C:\examples\ch04\src\com\endofhope\scbook\ch04’에 저장합니다. Server\_0은 8000번 포트를 연 후 Socket 연결에 대해 들어온 요청 내용을 표준 출력에 표시합니다. 먼저 다음과 같이 컴파일합니다.

- 컴파일 명령 : javac -d classes src\com\endofhope\scbook\ch04\Server\_0.java
- 실행 명령 : java -cp classes com.endofhope.scbook.ch04.Server\_0



이제 웹 브라우저로 'http://localhost:8000'을 호출합니다.

그림 4-1 'http://localhost:8000'을 호출



웹 브라우저가 8000번 포트로 HTTP 요청을 보낼 것이고, 명령창에서 해당 내용을 확인할 수 있습니다. 다음과 같이 명령창에는 익숙한 HTTP GET 요청 내용이 나타납니다.

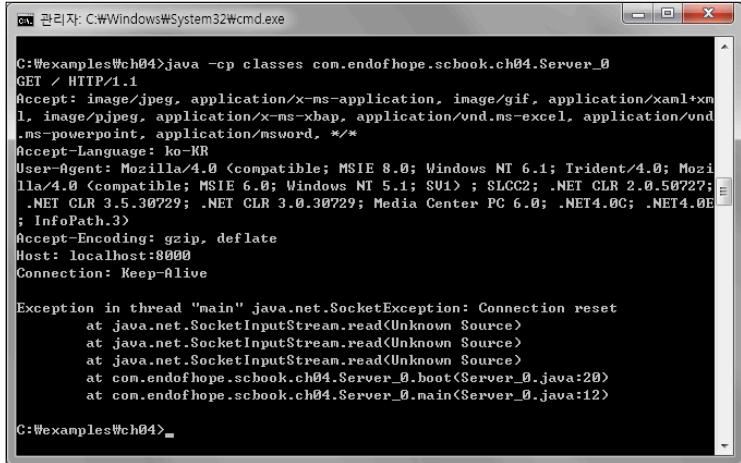
그림 4-2 HTTP GET 요청 내용

A screenshot of a Windows Command Prompt window titled '관리자: C:\Windows\System32\cmd.exe - java -cp classes com.endofhope.scbook.ch04.Server\_0'. The command entered is 'java -cp classes com.endofhope.scbook.ch04.Server\_0'. The output shows an incoming HTTP request:

```
C:\examples\ch04>java -cp classes com.endofhope.scbook.ch04.Server_0
GET / HTTP/1.1
Accept: image/jpeg, application/x-ms-application, image/gif, application/xaml+xml,
       image/pjpeg, application/x-ms-xbap, application/vnd.ms-excel, application/vnd.
       ms-powerpoint, application/msword, */*
Accept-Language: ko-KR
User-Agent: Mozilla/4.0 <compatible: MSIE 8.0; Windows NT 6.1; Trident/4.0; Mozi
lla/4.0 <compatible; MSIE 6.0; Windows NT 5.1; SV1> ; SLCC2; .NET CLR 2.0.50727;
       .NET CLR 3.5.30729; .NET CLR 3.0.30729; Media Center PC 6.0; .NET4.0C; .NET4.0E
       ; InfoPath.3>
Accept-Encoding: gzip, deflate
Host: localhost:8000
Connection: Keep-Alive
```

그런데 하나 꺼림칙한 부분이 있습니다. 이상하게도 웹 브라우저에서는 "연결 중....."이라는 표시가 계속 나옵니다. 또 코드를 보면 요청 하나를 처리한 후에는 프로그램이 정지해야 하는데 멈추지 않고, 웹 브라우저에서 [x] 버튼을 이용해 요청을 중지하면 그제서야 정지합니다.

그림 4-3 HTTP GET 요청 종료



```
C:\examples\ch04>java -cp classes com.endofhope.sbook.ch04.Server_0
GET / HTTP/1.1
Accept: image/jpeg, application/x-ms-application, image/gif, application/xaml+xml,
image/jpg, application/x-ms-xbap, application/vnd.ms-excel, application/vnd.
ms-powerpoint, application/msword, */*
Accept-Language: ko-KR
User-Agent: Mozilla/4.0 <compatible; MSIE 8.0; Windows NT 6.1; Trident/4.0; Mozi
lla/4.0 <compatible; MSIE 6.0; Windows NT 5.1; SUI>; SLCC2; .NET CLR 2.0.50727;
.NET CLR 3.5.30729; .NET CLR 3.0.30729; Media Center PC 6.0; .NET4.0C; .NET4.0E
; InfoPath.3>
Accept-Encoding: gzip, deflate
Host: localhost:8000
Connection: Keep-Alive

Exception in thread "main" java.net.SocketException: Connection reset
    at java.net.SocketInputStream.read(Unknown Source)
    at java.net.SocketInputStream.read(Unknown Source)
    at java.net.SocketInputStream.read(Unknown Source)
    at com.endofhope.sbook.ch04.Server_0.boot(Server_0.java:20)
    at com.endofhope.sbook.ch04.Server_0.main(Server_0.java:12)

C:\examples\ch04>_
```

2장에서 프로토콜을 해석할 때 어디가 메시지의 끝인가를 결정하는 것이 가장 기본이라고 설명했습니다. Server\_0은 다음과 같이 종료 조건이 결정됩니다.

---

```
int oneInt = -1;
while(-1 != (oneInt = in.read())){
    System.out.print((char)oneInt);
}
```

---

다시 말해, InputStream의 read 메서드가 -1을 반환하면 종료합니다. InputStream의 read 메서드는 스트림의 끝에 다다를 경우, 즉 소켓에서 얻은 스트림의 연결이 끊어질 경우에 -1을 반환합니다. 따라서 웹 브라우저의 [x] 버튼을 사용해 연결을 종료하면 종료 조건이 만족돼 Server\_0은 while 루프에서 탈출합니다.

이와 같은 종료 조건은 클라이언트-서버 구조에서 사용하기가 어렵습니다. 메시지의 끝을 연결 종료로 파악할 수밖에 없다면, 양방향 통신이 필요할 때 요청에 대한 응답을 보낼 수 없기 (이미 끊어졌으니) 때문입니다.<sup>01</sup> 따라서 서블릿 컨테이너는 스트림을 사용해 Socket에서 한 바이트씩 읽어들이면서 어느 시점에서 읽기를 중단하고 지금까지 받은 내용을 기반으로 요청 메시지를 구성할지 판단해야 합니다. HTTP 프로토콜에 대해 이런 상태를 분석하는 코드를 HTTP 상태 기계라고 합니다.

## 4.2 HTTP GET 요청 처리기

앞에서 HTTP 프로토콜을 설명하면서 HTTP 메시지는 CRLF로 행을 구분하고, 첫 번째 행은 특별한 구조를 가지며, 빈 내용의 행이 나오기 전까지 콜론으로 구분되는 메시지 헤더가 존재한다고 언급한 바 있습니다. 메시지 바디가 없는 GET 요청의 경우, HTTP 메시지의 종료 시점을 결정하는 데 있어 이런 행 구분자의 위치를 파악하는 것이 무엇보다 중요합니다.

다음은 CRLF를 고려한 Server\_1입니다.

---

```
package com.endofhope.scbook.ch04;

import java.io.IOException;
import java.io.InputStream;
import java.net.ServerSocket;
import java.net.Socket;

public class Server_1 {
    public static void main(String[] args) throws IOException{
```

---

01 요청하는 쪽에서 응답을 모두 받은 후에 끊으면 어떨까 생각할 수도 있겠지만, 응답이 다 왔다는 것을 판단하는 자체가 이미 메시지의 끝을 판단할 수 있는 것입니다.

```

Server_1 server = new Server_1();
server.boot();
}

private void boot() throws IOException{
    serverSocket = new ServerSocket(8000);
    Socket socket = serverSocket.accept();
    InputStream in = socket.getInputStream();
    int oneInt = -1;
    byte oldByte = (byte)-1;
    StringBuilder sb = new StringBuilder();
    int lineNumber = 0;
    while(-1 != (oneInt = in.read())){
        byte thisByte = (byte)oneInt;
        if(thisByte == Server_1.LF && oldByte == Server_1.CR){ ❶
            // CRLF가 완성되었다. 따라서 직전 CRLF부터 여기까지가 한 행이다.
            // -2가 아니라 -1을 하는 이유는 아직 LF 가 버퍼에 들어가기 전이기 때문이다.
            String oneLine = sb.substring(0, sb.length()-1);
            lineNumber++; ❷
            System.out.printf("%d: %s\n", lineNumber, oneLine);
            if(oneLine.length()<=0){ ❸
                // 내용이 없는 행
                // 따라서 메시지 헤더의 마지막일 경우다.
                System.out.println("[SYS] 내용이 없는 헤더, 즉 메시지 헤더의 끝");
                // 현 상황에서는 메시지 바디는 처리하지 말기로 한다.
                break;
            }
            sb.setLength(0);
        }else{
            sb.append((char)thisByte);
        }
        oldByte = (byte)oneInt;
    }
}

```

```
    in.close();
    socket.close();
}

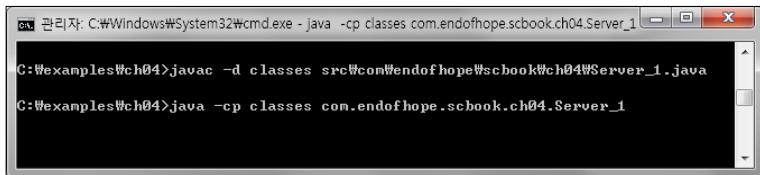
public static final byte CR = '\r';
public static final byte LF = '\n';

private ServerSocket serverSocket;
}
```

---

명령창에서 다음과 같이 컴파일하고 실행합니다.

- 컴파일 명령 : javac -d classes src\com\endofhope\scbook\ch04\Server\_1.java
- 실행 명령 : java -cp classes com.endofhope.scbook.ch04.Server\_1



이후 웹 브라우저로 'http://localhost:8000'을 호출하면 명령창에 웹 브라우저가 요청한 내용이 다음과 같이 표시됩니다.

그림 4-4 'http://localhost:8000'을 호출

```
cmd 관리자: C:\Windows\system32\cmd.exe
C:\examples\ch04>java -cp classes com.endofhope.sbook.ch04.Server_1
1: GET / HTTP/1.1
2: Accept: image/jpeg, application/x-ms-application, image/gif, application/xaml+xml, image/pjpeg, application/x-ms-xbap, application/vnd.ms-excel, application/vnd.ms-powerpoint, application/msword, */*
3: Accept-Language: ko-KR
4: User-Agent: Mozilla/4.0 <compatible; MSIE 8.0; Windows NT 6.1; Trident/4.0; Mozilla/4.0 <compatible; MSIE 6.0; Windows NT 5.1; SV1>; SLCC2; .NET CLR 2.0.50727; .NET CLR 3.0.30729; .NET CLR 3.0.30729; Media Center PC 6.0; .NET4.0C; .NET4.0E; InfoPath.3>
5: Accept-Encoding: gzip, deflate
6: Host: localhost:8000
7: Connection: Keep-Alive
8:
[SYS] 내용이 없는 헤더. 즉 메시지 헤더의 끝
C:\examples\ch04>
```

웹 브라우저가 보낸 HTTP 요청에 대해 각 행을 구분한 후 번호를 붙여 표준 출력에 표시했습니다. 또한 메시지 헤더가 종료되면 메시지의 끝으로 판단해 프로그램을 종료시킵니다.

Server\_0과 비교하여 Server\_1에서 추가된 부분은 다음과 같습니다. 먼저 읽어들인 데이터를 기억하고 있다가(①행) 두 바이트로 이뤄진 행 구분자 조합, 즉 CRLF 조합을, 만나면 그때까지 임시 저장한 데이터를 하나의 행으로 만들어 표시합니다(②, ③행).

2장에서 메시지 바디를 가지는 여러 HTTP 메시지 형식을 살펴보았습니다. 대표적으로 메시지 바디의 길이를 Content-Length 값으로 메시지 헤더에 지정하는 방식과 메시지 바디를 청크의 연속으로 표현하는 방식이 있었습니다. 이제 Content-Length를 처리할 수 있게 헤더 값을 관리하고 해당 Content-Length 값을 기반으로 메시지 바디를 처리해 보겠습니다.

## 4.3 메시지 바디 처리 - Content-Length 인식

---

```
package com.endofhope.scbook.ch04;

import java.io.IOException;
import java.io.InputStream;
import java.net.ServerSocket;
import java.net.Socket;
import java.util.ArrayList;
import java.util.HashMap;
import java.util.Iterator;
import java.util.List;
import java.util.Map;
import java.util.Set;

public class Server_2 {
    public static void main(String[] args) throws IOException{
        Server_2 server = new Server_2();
        server.boot();
    }
    private void boot() throws IOException{
        serverSocket = new ServerSocket(8000);
        Socket socket = serverSocket.accept();
        InputStream in = socket.getInputStream();
        int oneInt = -1;
        byte oldByte = (byte)-1;
        StringBuilder sb = new StringBuilder();
        int lineNumber = 0;
        boolean bodyFlag = false;
        String method = null;
        String requestUrl = null;
        String httpVersion = null;
```

```

int contentLength = -1;
int bodyRead = 0;
List<Byte> bodyByteList = null;
Map<String, String> headerMap = new HashMap<String, String>();
while(-1 != (oneInt = in.read())){
    byte thisByte = (byte)oneInt;
    if(bodyFlag){
        bodyRead++;
        bodyByteList.add(thisByte);
        if(bodyRead >= contentLength){
            break;
        }
    }else{
        if(thisByte == Server_2.LF && oldByte == Server_2.CR){
            String oneLine = sb.substring(0, sb.length()-1);
            lineNumber++;
            if(lineNumber == 1){
                // 요청의 첫 행, HTTP 메서드, 요청 URL, 버전을 알아낸다.
                int firstBlank = oneLine.indexOf(" ");
                int secondBlank = oneLine.lastIndexOf(" ");
                method = oneLine.substring(0, firstBlank);
                requestUrl = oneLine.substring(firstBlank+1, secondBlank);
                httpVersion = oneLine.substring(secondBlank+1);
            }else{
                if(oneLine.length()<=0){
                    bodyFlag = true;
                    // 헤더가 끝났다.
                    if("GET".equals(method)){
                        // GET 방식이면 메시지 바디가 없다
                        break;
                    }
                    String contentLengthValue = headerMap.get("Content-Length");

```

```

        if(contentLengthValue != null){
            contentLength = Integer.parseInt(contentLengthValue.trim());
            bodyFlag = true;
            bodyByteList = new ArrayList<Byte>();
        }
        continue;
    }
    int indexOfColon = oneLine.indexOf(":");
    String headerName = oneLine.substring(0, indexOfColon);
    String headerValue = oneLine.substring(indexOfColon+1);
    headerMap.put(headerName, headerValue);
}
sb.setLength(0);
}else{
    sb.append((char)thisByte);
}
}
oldByte = (byte)oneInt;
}
in.close();
socket.close();
System.out.printf("METHOD: %s REQ: %s HTTP VER. %s\n", method,
    requestUrl, httpVersion);
System.out.println("Header list");
Set<String> keySet = headerMap.keySet();
Iterator<String> keyIter = keySet.iterator();
while(keyIter.hasNext()){
    String headerName = keyIter.next();
    System.out.printf(" Key: %s Value: %s\n", headerName,
        headerMap.get(headerName));
}
if(bodyByteList != null){

```

```

        System.out.print("Message Body-->");
        for(byte oneByte : bodyByteList){
            System.out.print(oneByte);
        }
        System.out.println("<--");
    }

    System.out.println("End of HTTP Message.");
}

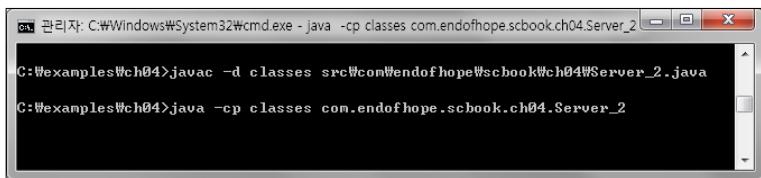
public static final byte CR = '\r';
public static final byte LF = '\n';
private ServerSocket serverSocket;
}

```

---

위와 같이 Server\_2 클래스를 생성한 후 다음과 같이 컴파일하고 실행시킵니다.

- 컴파일 명령 : javac -d classes src\com\endofhope\scbook\ch04\Server\_2.java
- 실행 명령 : java -cp classes com.endofhope.scbook.ch04.Server\_2



이제 메시지 바디가 있는 HTTP 요청을 보내기 위해 다음과 같이 post.html을 생성합니다.

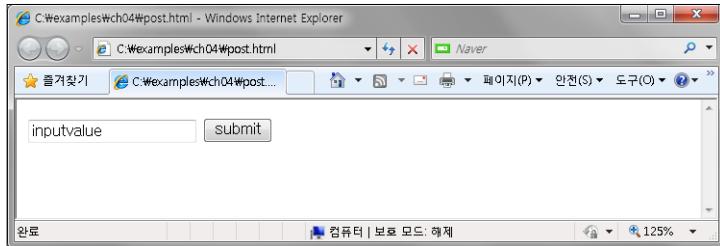
---

```
<html><head></head><body>
<form name="formname" method="post" action="http://localhost:8000">
<input type="text" name="inputname" value="inputvalue" />
<input type="submit" name="submit" value="submit" />
</form>
</body></html>
```

---

웹 브라우저로 해당 파일을 연 다음 [submit] 버튼을 클릭해 HTTP POST 메시지를 전송합니다.

그림 4-5 HTTP POST 메시지 전송



HTTP POST 메시지를 전송하면 Server\_2가 수행된 명령행에 다음과 같이 웹 브라우저에 전송된 결과가 표시됩니다.

그림 4-6 HTTP POST 메시지 전송 결과

```
C:\examples\ch04> java -cp classes com.endofhope.school.ch04.Server_2
METHOD: POST REQ: / HTTP VER: HTTP/1.1
Header list
Key: Accept-Language Value: ko-KR
Key: Host Value: localhost:8000
Key: Content-Length Value: 34
Key: Accept-Encoding Value: gzip, deflate
Key: User-Agent Value: Mozilla/4.0 (compatible; MSIE 8.0; Windows NT 6.1; Trident/4.0; Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.1; SV1) ; SLCC2; .NET CLR 2.0.50727; .NET CLR 3.5.30729; .NET CLR 3.0.30729; Media Center PC 6.0; .NET 4.0C; .NET4.0E; InfoPath.3)
Key: Connection Value: Keep-Alive
Key: Content-Type Value: application/x-www-form-urlencoded
Key: Accept Value: image/jpeg, application/x-ms-application, image/gif, application/xaml+xml, image/pjpeg, application/x-ms-xbap, application/vnd.ms-excel, application/vnd.ms-powerpoint, application/msword, */*
Key: Cache-Control Value: no-cache
Message Body-->1051101121171161109710910161105101121171161189710811710138115117981091051166111511798109105116<-
End of HTTP Message.

C:\examples\ch04>
```

어떤 부분이 Server\_2에 추가돼, 메시지 바디가 있는 HTTP POST 요청을 처리할 수 있게 됐는지 확인해 보겠습니다.

현재 읽는 HTTP 메시지가 메시지 바디인지 아닌지를 표시하는 bodyFlag라는 불리언boolean 값을 사용해 시작행과 메시지 헤더의 처리와 메시지 바디 처리 부분을 분리했습니다(37, 43행). 메시지 헤더는 문자열을 기반으로 하나의 행이라는 개념으로 각각을 구분할 수 있으나 메시지 바디는 이런 행 기반 구조를 사용하지 않으므로 이렇게 분리했습니다. 메시지 바디를 처리할 때는 단순히 바이트의 묶음으로 처리합니다.

메시지 헤더를 처리할 때 시작행을 특별하게 처리해 HTTP 메서드, 요청 URL, HTTP 버전 정보를 알아냅니다(47행).

메시지 헤더가 종료되면 메시지 바디를 읽을 것인지 종료할 것인지를 결정합니다. 이 말은 HTTP 메서드가 GET과 같이, 메시지 바디가 존재하지 않는 형식의 메서

드라면 더 이상의 읽기를 중지하고 바로 빠져나가고(58행), 메시지 바디가 있을 경우 Content-Length 헤더 값을 확인해 메시지 바디를 얼마나 읽어야 되는지를 판단합니다(62행). 이를 위해, 읽어들인 메시지 헤더를 이름/값 쌍으로 분리한 후 Map으로 저장합니다(73행).

일단 Content-Length 헤더 값을 확인해 메시지 바디의 크기를 확정합니다. 그리고 지금까지 읽은 메시지 바디의 크기를 유지해 미리 Content-Length로 전달받은 메시지 크기에 도달하기를 기다려서 메시지 읽기를 종료합니다(40행).

이후 시작행에서 분석한 HTTP 메서드, 요청 URL, HTTP 버전 등의 정보와 메시지 헤더, 메시지 바디의 값을 표준 출력으로 보냅니다(85행).

## 4.4 매개변수의 처리

### - 쿼리스트링과 x-www-form-urlencoded

GET 방식의 경우 매개변수를 전달하면 요청 URL의 뒤에 ‘?’ 문자를 붙여 매개변수를 전달할 수 있습니다. 단, 해당 매개변수의 이름과 값은 ‘=’로 분리하고 매개변수 쌍은 ‘&’로 구분합니다. 이런 방식을 쿼리스트링이라 합니다. 쿼리스트링의 매개변수는 요청 URL에 포함되므로 처리할 수 있는 요청 URL의 길이가 제한되는 서버인 경우<sup>02</sup> 전달 가능한 값이 적을 수도 있습니다.

POST 방식의 매개변수 전달은 쿼리스트링 방식의 길이 제한을 회피하기 위해 매개변수를 메시지 바디에 추가하는 방법입니다. 이를 위해, Content-Type 헤더 값을 application/x-www-form-urlencoded로 지정하고 메시지 바디에 쿼리스트링과 같은 포맷을 사용해 데이터를 전달합니다. 단, 매개변수의 각 이름/값을 URL 인코딩 방식으로 인코딩한 후에 전달됩니다.

---

02 현재 대부분의 웹 서버나 웹 애플리케이션 서버에는 URL의 길이 제한이 없거나 매우 긴 URL까지 지원합니다. 초창기 대부분의 웹 서버는 URL의 길이가 255바이트로 제한됐습니다.

매개변수를 처리할 수 있게 수정한 Server\_3은 다음과 같습니다.

---

```
package com.endofhope.scbook.ch04;

import java.io.IOException;
import java.io.InputStream;
import java.net.ServerSocket;
import java.net.Socket;
import java.net.URLDecoder;
import java.util.ArrayList;
import java.util.HashMap;
import java.util.Iterator;
import java.util.List;
import java.util.Map;
import java.util.Set;
import java.util.StringTokenizer;

public class Server_3 {

    public static void main(String[] args) throws IOException{
        Server_3 server = new Server_3();
        server.boot();
    }

    private void boot() throws IOException{
        serverSocket = new ServerSocket(8000);
        Socket socket = serverSocket.accept();
        InputStream in = socket.getInputStream();
        int oneInt = -1;
        byte oldByte = (byte)-1;
        StringBuilder sb = new StringBuilder();
        int lineNumber = 0;
```

```

boolean bodyFlag = false;
String method = null;
String requestUrl = null;
String httpVersion = null;
int contentLength = -1;
int bodyRead = 0;
List<Byte> bodyByteList = null;
Map<String, String> headerMap = new HashMap<String, String>();
while(-1 != (oneInt = in.read())){
    byte thisByte = (byte)oneInt;
    if(bodyFlag){
        bodyRead++;
        bodyByteList.add(thisByte);
        if(bodyRead >= contentLength){
            break;
        }
    }else{
        if(thisByte == Server_3.LF && oldByte == Server_3.CR){
            String oneLine = sb.substring(0, sb.length()-1);
            lineNumber++;
            if(lineNumber == 1){
                // 요청의 첫 행, HTTP 메서드, 요청 URL, 버전을 알아낸다.
                int firstBlank = oneLine.indexOf(" ");
                int secondBlank = oneLine.lastIndexOf(" ");
                method = oneLine.substring(0, firstBlank);
                requestUrl = oneLine.substring(firstBlank+1, secondBlank);
                httpVersion = oneLine.substring(secondBlank+1);
            }else{
                if(oneLine.length()<=0){
                    bodyFlag = true;
                    // 헤더가 끝났다.
                    if("GET".equals(method)){

```

```

        // GET 방식이면 메시지 바디가 없다
        break;
    }
    String contentLengthValue = headerMap.get("Content-Length");
    if(contentLengthValue != null){
        contentLength = Integer.parseInt(contentLengthValue.trim());
        bodyFlag = true;
        bodyByteList = new ArrayList<Byte>();
    }
    continue;
}
int index0fColon = oneLine.indexOf(":");
String headerName = oneLine.substring(0, index0fColon);
String headerValue = oneLine.substring(index0fColon+1);
headerMap.put(headerName, headerValue);
}
sb.setLength(0);
}else{
    sb.append((char)thisByte);
}
}
oldByte = (byte)oneInt;
}
in.close();
socket.close();

System.out.printf("METHOD: %s REQ: %s HTTP VER. %s\n", method,
    requestUrl, httpVersion);
Map<String, String> paramMap = new HashMap<String, String>();
int index0fQuotation = requestUrl.indexOf "?";
if(index0fQuotation > 0){
    StringTokenizer st = new

```

```
 StringTokenizer(requestUrl.substring(indexOfQuotation+1), "&");

    while(st.hasMoreTokens()){

        String params = st.nextToken();
        paramMap.put(params.substring(0, params.indexOf("=")),
                     params.substring(params.indexOf("=")+1));
    }
}

System.out.println("Header list");
Set<String> keySet = headerMap.keySet();
Iterator<String> keyIter = keySet.iterator();
while(keyIter.hasNext()){

    String headerName = keyIter.next();
    System.out.printf(" Key: %s Value: %s\n", headerName,
                      headerMap.get(headerName));
}

if(bodyByteList != null){

    if("application/x-www-form-urlencoded".equals(headerMap.get("Content-Type").trim())){
        int startIndex = 0;
        byte[] srcBytes = new byte[bodyByteList.size()];
        String currentName = null;
        for(int i=0; i<bodyByteList.size(); i++){
            byte oneByte = bodyByteList.get(i);
            srcBytes[i] = oneByte;
            if('=' == oneByte){
                byte[] one = new byte[i-startIndex];
                System.arraycopy(srcBytes, startIndex, one, 0, i-startIndex);
                currentName = URLDecoder.decode(new String(one), "CP949");
                startIndex = i+1;
            }else if('&' == oneByte){
                byte[] one = new byte[i-startIndex];
                System.arraycopy(srcBytes, startIndex, one, 0, i-startIndex);
            }
        }
    }
}
```

```
paramMap.put(currentName, URLDecoder.decode(new String(one), "CP949"));
startIndex = i+1;
}else if(i == bodyByteList.size()-1){
    byte[] one = new byte[i-startIndex+1];
    System.arraycopy(srcBytes, startIndex, one, 0, i-startIndex+1);
    paramMap.put(currentName, URLDecoder.decode(new String(one), "CP949"));
    startIndex = i+1;
}
}
}else{
    System.out.print("Message Body-->");
    for(byte oneByte : bodyByteList){
        System.out.print(oneByte);
    }
    System.out.println("<--");
}
}

Set<String>paramKeySet = paramMap.keySet();
Iterator<String> paramKeyIter = paramKeySet.iterator();
while(paramKeyIter.hasNext()){
    String paramName = paramKeyIter.next();
    System.out.printf("paramName: %s paramValue: %s\n", paramName,
        paramMap.get(paramName));
}
System.out.println("End of HTTP Message.");
}

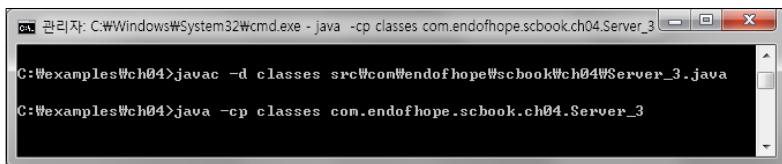
public static final byte CR = '\r';
public static final byte LF = '\n';

private ServerSocket serverSocket;
}
```

---

다음과 같이 명령창에서 컴파일하고 실행해 보겠습니다.

- 컴파일 명령 : javac -d classes src\com\endofhope\scbook\ch04\Server\_3.java
- 실행 명령 : java -cp classes com.endofhope.scbook.ch04.Server\_3



The screenshot shows a Windows Command Prompt window titled "관리자: C:\Windows\System32\cmd.exe". The command line shows the user navigating to the directory "C:\examples\ch04" and then running the command "java -cp classes com.endofhope.scbook.ch04.Server\_3". The output of the command is displayed below the command line.

```
C:\examples\ch04>javac -d classes src\com\endofhope\scbook\ch04\Server_3.java
C:\examples\ch04>java -cp classes com.endofhope.scbook.ch04.Server_3
```

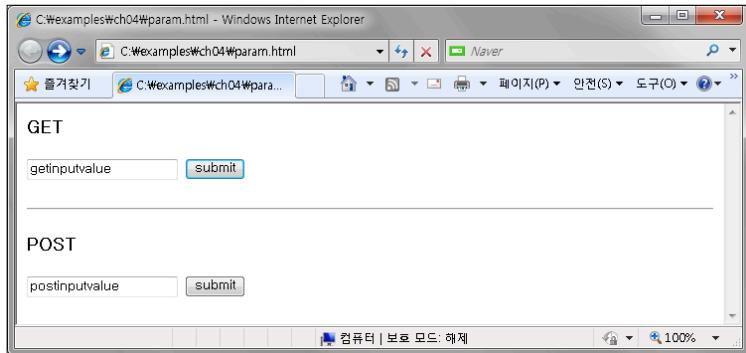
GET 방식과 POST 방식의 매개변수를 전달하기 위해 다음과 같이 param.html 을 만들고, 웹 브라우저에서 호출합니다.

---

```
<html><head></head><body>
<h3>GET</h3>
<form name="formname" method="get" action="http://localhost:8000">
<input type="text" name="getinputname" value="getinputvalue" />
<input type="submit" name="submit" value="submit" />
</form>
<hr />
<h3>POST</h3>
<form name="formname" method="post" action="http://localhost:8000">
<input type="text" name="postinputname" value="postinputvalue" />
<input type="submit" name="submit" value="submit" />
</form>
</body></html>
```

---

그림 4-7 웹 브라우저로 param.html 실행



GET 방식으로 보낸 결과는 다음과 같습니다.

그림 4-8 GET 방식으로 매개변수 전달

```
관리자: C:\Windows\System32\cmd.exe

C:\examples\ch04>java -cp classes com.endofhope.sbook.ch04.Server_3
METHOD: GET REQ: /?getinputname=getinputvalue&submit=submit HTTP/1.1
Header list
Key: Accept-Language Value: ko-KR
Key: Host Value: localhost:8000
Key: Accept-Encoding Value: gzip, deflate
Key: User-Agent Value: Mozilla/4.0 <compatible; MSIE 8.0; Windows NT 6.1; Trident/4.0; Mozilla/4.0 <compatible; MSIE 6.0; Windows NT 5.1; SV1> ; SLCC2; .NET CLR 2.0.50727; .NET CLR 3.5.30729; .NET CLR 3.0.30729; Media Center PC 6.0; .NET 4.0; .NET4.0E; InfoPath.3>
Key: Connection Value: Keep-Alive
Key: Accept Value: image/jpeg, application/x-ms-application, image/gif, application/xaml+xml, image/pjpeg, application/x-ms-xbap, application/vnd.ms-excel, application/vnd.ms-powerpoint, application/msword, */*
paramName: submit paramValue: submit
paramName: getinputname paramValue: getinputvalue
End of HTTP Message.

C:\examples\ch04>
```

POST 방식의 결과는 다음과 같습니다.

그림 4-9 POST 방식으로 매개변수를 전달한 결과

```
C:\examples\ch04>java -cp classes com.endofhope.sbook.ch04.Server_3
METHOD: POST REQ: / HTTP VER: HTTP/1.1
Header list
Key: Accept-Language Value: ko-KR
Key: Host Value: localhost:8000
Key: Content-Length Value: 42
Key: Accept-Encoding Value: gzip, deflate
Key: User-Agent Value: Mozilla/4.0 <compatible; MSIE 8.0; Windows NT 6.1; Trident/4.0; Mozilla/4.0 <compatible; MSIE 6.0; Windows NT 5.1; SV1>; SLCC2; .NET CLR 2.0.50727; .NET CLR 3.5.30729; .NET CLR 3.0.30729; Media Center PC 6.0; .NET 4.0C; .NET4.0E; InfoPath.3>
Key: Connection Value: Keep-Alive
Key: Content-Type Value: application/x-www-form-urlencoded
Key: Accept Value: image/jpeg, application/x-ms-application, image/gif, application/xaml+xml, image/pjpeg, application/x-ms-xbap, application/vnd.ms-excel, application/vnd.ms-powerpoint, application/msword, */*
Key: Cache-Control Value: no-cache
paramName: submit paramValue: submit
paramName: postinputname paramValue: postinputvalue
End of HTTP Message.

C:\examples\ch04>
```

동일한 매개변수 이름과 값을 전달했으므로 당연히 GET 방식과 POST 방식의 결과는 같습니다. 하지만 웹 브라우저가 GET과 POST 방식의 전달 방법은 쿼리스트링과 x-www-form-urlencoded 방식으로 전혀 다릅니다.

먼저 쿼리스트링을 처리하기 위해 요청 URL에 대한 처리 부분을 추가했습니다(91행). '=' 및 '&' 을 구분자로 매개변수 이름과 값을 분리해 paramMap을 구성합니다.

메시지 바디를 사용하는 x-www-form-urlencoded에 대한 처리는 좀 더 복잡합니다. 먼저 Content-Type 헤더 값을 확인하여 메시지 바디가 URL 인코딩된 값이란 것을 확인합니다(106행). 그리고 메시지 바디를 '='와 '&'로 구분한 후 해당 값을 URLDecoder를 사용해 변환하여 paramMap을 구성했습니다.

이후 GET/POST 방식 모두 매개변수에 대해 paramMap에 이름/값 쌍으로 저장했으므로 paramMap만 살펴보면 매개변수 값에 모두 접근할 수 있습니다(140행).

지금까지 포트를 열어 웹 브라우저의 요청을 표준 출력에 남기는 (Server\_0) 것에서 시작해 행 구분자를 사용해 메시지 헤더를 확인해 보았습니다(Server\_1). 시작 행과 메시지 헤더, 메시지 바디를 구분하는 방법도 알아보았습니다(Server\_2). 마지막으로 쿼리스트링 방식과 x-www-form-urlencoded 방식의 매개변수 전달에 대응할 수 있게, 특정 메시지 헤더를 체크하고 필요한 경우 메시지 바디를 분석해 웹 브라우저가 전달한 매개변수를 서버 사이드에서 재구성했습니다(Server\_3).

구현의 기본적인 접근 방식은 다음과 같습니다. 먼저 소켓 스트림을 열어 클라이언트가 전달한 데이터를 한 바이트씩 읽어갑니다. 이때 HTTP 메시지의 내부 상태를 계속 관찰하면서 메시지 종료 조건을 동적으로 판별해 메시지 읽기 종료 시점을 결정하고 HTTP 메시지를 구성합니다. 이제 웹 브라우저에서 받은 요청이 무엇인지 결정해 적절한 응답을 보냅니다. 우리는 받은 HTTP 요청 내용을 파악할 수 있게 됐으므로 다음에는 해당 요청을 처리할 서블릿을 선택하고, 요청을 서블릿에게 위임하는 방법에 대해 알아보겠습니다.

이 장을 마무리하기 전에 HTTP 프로토콜 분석과는 직접적인 관계는 없으나 실제로 구현할 때는 반드시 고려해야 할 두 가지에 대해 간략히 살펴보겠습니다.

## 4.5 성능 개선 1 – 버퍼의 사용

외부 자원에 접근할 때 한 번에 얻을 수 있는 양은 전체 접근 횟수를 결정하므로 성능에 큰 영향을 미칩니다. 다음은 로컬 디스크에서 파일을 한 바이트씩 복사하는 것과 버퍼를 사용하는 것에 대한 간단한 벤치 마크입니다.

---

```
package com.endofhope.scbook.ch04;

import java.io.FileInputStream;
import java.io.FileOutputStream;
```

```
import java.io.IOException;
import java.io.InputStream;
import java.io.OutputStream;

public class Copy {
    public static void main(String[] args) throws IOException{
        if(args == null || args.length < 1){
            System.out.println("파일이름 지정과 버퍼 사용 여부가 필요합니다.");
            System.exit(0);
        }
        Copy c = new Copy(args[0]);
        long before = System.currentTimeMillis();

        if(args.length > 1 && "no".equals(args[1])){
            c.noBufferAction();
        }else{
            c.bufferAction();
        }
        long after = System.currentTimeMillis();
        System.out.printf("%.3f\n", (float)((after-before)/1000f));
    }
    private Copy(String fileName){
        this.fileName = fileName;
    }
    private String fileName;
    private String targetFileName;
    private void noBufferAction() throws IOException{
        targetFileName = fileName.concat("-nob");
        InputStream in = new FileInputStream(fileName);
        OutputStream out = new FileOutputStream(targetFileName);
        int oneInt = -1;
        while(-1 != (oneInt = in.read())){

```

```
        out.write(oneInt);
    }
    in.close();
    out.close();
}

private void bufferAction() throws IOException{
    targetFileName = fileName.concat("-useb");
    InputStream in = new FileInputStream(fileName);
    OutputStream out = new FileOutputStream(targetFileName);
    byte[] buffer = new byte[1024];
    int readSize = 0;
    while(0<(readSize = in.read(buffer))){
        out.write(buffer, 0, readSize);
    }
    in.close();
    out.close();
}
}
```

---

매개변수로 전달받은 파일에 대해 noBufferAction일 때 한 바이트씩 읽은 다음  
다시 복제된 파일에 쓰고, bufferAction일 때는 내부 버퍼를 사용해 최대 1,024바  
이트까지 읽은 다음 파일에 읽어들인 바이트 배열을 씁니다.

먼저 다음과 같이 컴파일합니다.

- 컴파일 명령 : javac -d classes src\com\endofhope\scbook\ch04\Copy.

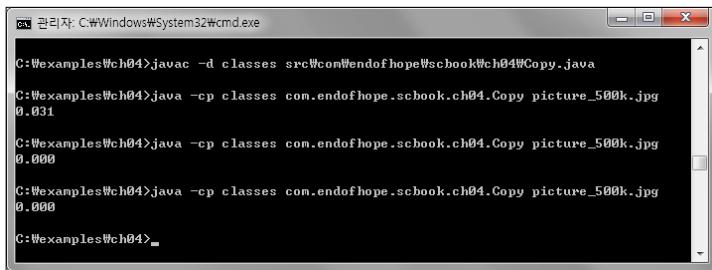
java



```
관리자: C:\Windows\System32\cmd.exe
C:\examples\ch04>javac -d classes src\com\endofhope\scbook\ch04\Copy.java
C:\examples\ch04>
```

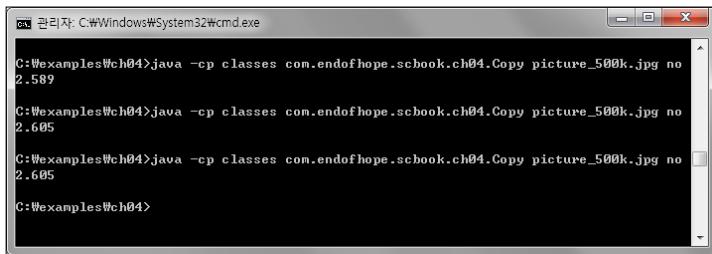
각 3회씩 수행한 결과는 다음과 같습니다. 먼저 내부 버퍼를 사용하는 경우 0.02초 내에 수행됩니다.

- 실행 명령 : java -cp classes com.endofhope.scbook.ch04.Copy picture\_500k.jpg



```
관리자: C:\Windows\System32\cmd.exe
C:\examples\ch04>javac -d classes src\com\endofhope\scbook\ch04\Copy.java
C:\examples\ch04>java -cp classes com.endofhope.scbook.ch04.Copy picture_500k.jpg
0.031
C:\examples\ch04>java -cp classes com.endofhope.scbook.ch04.Copy picture_500k.jpg
0.000
C:\examples\ch04>java -cp classes com.endofhope.scbook.ch04.Copy picture_500k.jpg
0.000
C:\examples\ch04>
```

내부 버퍼를 사용하지 않는 경우 대략 3초 걸렸습니다.



```
관리자: C:\Windows\System32\cmd.exe
C:\examples\ch04>java -cp classes com.endofhope.scbook.ch04.Copy picture_500k.jpg no
2.589
C:\examples\ch04>java -cp classes com.endofhope.scbook.ch04.Copy picture_500k.jpg no
2.605
C:\examples\ch04>java -cp classes com.endofhope.scbook.ch04.Copy picture_500k.jpg no
2.605
C:\examples\ch04>
```

결과적으로 한 바이트씩 읽어 복사하는 것보다 버퍼를 사용해 복제하는 것이 비교할 수 없을 정도로 성능면에서 뛰어난 것을 확인할 수 있습니다. 시스템 프로그래머가 메서드를 하나 사용할 때마다 해당 메서드가 내부적으로 시스템 호출을 유발하지는 않을지 언제나 전전긍긍하며 (이해하기 힘들 정도로) 걱정을 달고 사는지 이해할 수 있는 단초가 됐으면 좋겠습니다.

또한 HTTP 메시지와 같이 네트워크상으로 접근되는 자원은 로컬 머신을 기원으로 하는 자원과는 달리, 언제나 안정적으로 접근 가능할 것이라는 가정을 하기 힘듭니다. 이것은 원격 전송을 위해 물리적인 신호로 바꿔어 전달되는 동안 지연되거나 혹은 유실로 인한 전송 실패의 가능성성이 높다는 점에 기인합니다. 따라서 이번 호출이 성공했다고 해서 다음 호출이 성공하리라는 보장은 없다는 것을 의미합니다. 또 획득 가능한 시점에 가능한 만큼 (버퍼를 사용해서라도) 최대한 얻어오는 ‘제걸스런’ 방식을 사용하는 것에 대한 정당성을 뒷받침합니다.

소켓에서 스트림을 읽어들어올 때, 앞의 예제와 같이 하나의 바이트를 읽는 것보다 버퍼를 사용해 읽어들인 다음 처리하는 것으로 Server\_3을 수정해 보겠습니다.

---

```
package com.endofhope.scbook.ch03;

import java.io.IOException;
import java.io.InputStream;
import java.net.ServerSocket;
import java.net.Socket;
import java.net.URLDecoder;
import java.util.ArrayList;
import java.util.HashMap;
import java.util.Iterator;
import java.util.List;
import java.util.Map;
```

```
import java.util.Set;
import java.util.StringTokenizer;

public class Server_4 {

    public static void main(String[] args) throws IOException{
        Server_4 server = new Server_4();
        server.boot();
    }

    private void boot() throws IOException{
        serverSocket = new ServerSocket(8000);
        Socket socket = serverSocket.accept();
        InputStream in = socket.getInputStream();
        int oneInt = -1;
        byte oldByte = (byte)-1;
        StringBuilder sb = new StringBuilder();
        int lineNumber = 0;
        boolean bodyFlag = false;
        String method = null;
        String requestUrl = null;
        String httpVersion = null;
        int contentLength = -1;
        int bodyRead = 0;
        List<Byte> bodyByteList = null;
        Map<String, String> headerMap = new HashMap<String, String>();
        int readSize = 0;
        byte[] readBuffer = new byte[128];
        boolean isTerminal = false;
        while(0<(readSize = in.read(readBuffer))){
            for(int i=0; i<readSize; i++){
                byte thisByte = (byte)readBuffer[i];
                System.out.print((char)thisByte);
            }
        }
    }
}
```

```

if(bodyFlag){
    bodyRead++;
    bodyByteList.add(thisByte);
    if(bodyRead >= contentLength){
        isTerminal = true;
        break;
    }
}else{
    if(thisByte == Server_4.LF && oldByte == Server_4.CR){
        String oneLine = sb.substring(0, sb.length()-1);
        lineNumber++;
        if(lineNumber == 1){
            // 요청의 첫 행, HTTP 메서드, 요청 URL, 버전을 알아낸다.
            int firstBlank = oneLine.indexOf(" ");
            int secondBlank = oneLine.lastIndexOf(" ");
            method = oneLine.substring(0, firstBlank);
            requestUrl = oneLine.substring(firstBlank+1, secondBlank);
            httpVersion = oneLine.substring(secondBlank+1);
        }else{
            if(oneLine.length()<=0){
                bodyFlag = true;
                // 헤더가 끝났다.
                if("GET".equals(method)){
                    // GET 방식이면 메시지 바디가 없다
                    isTerminal = true;
                    break;
                }
            }
            String contentLengthValue = headerMap.get("Content-Length");
            if(contentLengthValue != null){
                contentLength = Integer.parseInt(contentLengthValue.trim());
                bodyFlag = true;
                bodyByteList = new ArrayList<Byte>();
            }
        }
    }
}

```

```

        }
        continue;
    }
    int indexOfColon = oneLine.indexOf(":");
    String headerName = oneLine.substring(0, indexOfColon);
    String headerValue = oneLine.substring(indexOfColon+1);
    headerMap.put(headerName, headerValue);
}
sb.setLength(0);
}else{
    sb.append((char)thisByte);
}
}

oldByte = (byte)thisByte;
}
if(isTerminal){
    break;
}
in.close();
socket.close();

System.out.printf("METHOD: %s REQ: %s HTTP VER. %s\n", method,
    requestUrl, httpVersion);
Map<String, String> paramMap = new HashMap<String, String>();
int indexOfQuotation = requestUrl.indexOf "?";
if(indexOfQuotation > 0){
    StringTokenizer st =
        new StringTokenizer(requestUrl.substring(indexOfQuotation+1), "&");
    while(st.hasMoreTokens()){
        String params = st.nextToken();

```

```

paramMap.put(params.substring(0, params.indexOf("=")),
             params.substring(params.indexOf("=")+1));
        }
    }
    System.out.println("Header list");
    Set<String> keySet = headerMap.keySet();
    Iterator<String> keyIter = keySet.iterator();
    while(keyIter.hasNext()){
        String headerName = keyIter.next();
        System.out.printf(" Key: %s Value: %s\n", headerName,
                          headerMap.get(headerName));
    }
    if(bodyByteList != null){
        if("application/x-www-form-urlencoded".equals(headerMap.get("Content-Type").trim())){
            int startIndex = 0;
            byte[] srcBytes = new byte[bodyByteList.size()];
            String currentName = null;
            for(int i=0; i<bodyByteList.size(); i++){
                byte oneByte = bodyByteList.get(i);
                srcBytes[i] = oneByte;
                if('=' == oneByte){
                    byte[] one = new byte[i-startIndex];
                    System.arraycopy(srcBytes, startIndex, one, 0, i-startIndex);
                    currentName = URLDecoder.decode(new String(one), "CP949");
                    startIndex = i+1;
                }else if('&' == oneByte){
                    byte[] one = new byte[i-startIndex];
                    System.arraycopy(srcBytes, startIndex, one, 0, i-startIndex);
                    paramMap.put(currentName, URLDecoder.decode(new String(one), "CP949"));
                    startIndex = i+1;
                }else if(i == bodyByteList.size()-1){

```

```
        byte[ ] one = new byte[i-startIndex+1];
        System.arraycopy(srcBytes, startIndex, one, 0, i-startIndex+1);
        paramMap.put(currentName, URLDecoder.decode(new String(one), "CP949"));
        startIndex = i+1;
    }
}
}else{
    System.out.print("Message Body-->");
    for(byte oneByte : bodyByteList){
        System.out.print(oneByte);
    }
    System.out.println("<--");
}
}

Set<String>paramKeySet = paramMap.keySet();
Iterator<String> paramKeyIter = paramKeySet.iterator();
while(paramKeyIter.hasNext()){
    String paramName = paramKeyIter.next();
    System.out.printf("paramName: %s paramValue: %s\n", paramName,
        paramMap.get(paramName));
}
System.out.println("End of HTTP Message.");
}

public static final byte CR = '\r';
public static final byte LF = '\n';

private ServerSocket serverSocket;
}
```

---

읽기 버퍼로 바이트 배열 `readBuffer`를 사용한 것과 하나의 메시지가 여러 버퍼로 나뉠 수 있으므로 불리언 값인 `isTerminal`을 추가해 스트림에서 버퍼에 값을 담는 메인 루프의 탈출 여부를 표시한 것에 유의해야 합니다.

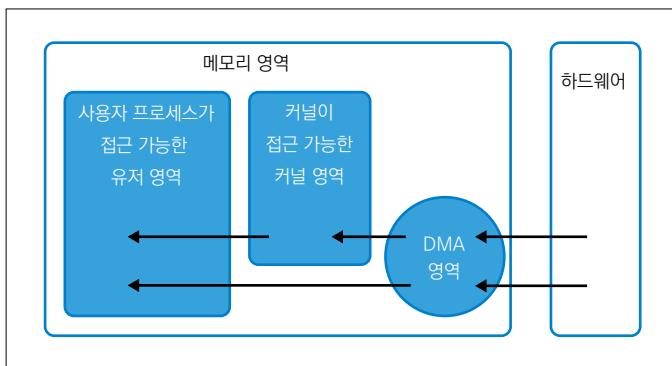
## 4.6 성능 개선 2 – 더 나은 I/O

앞에서 버퍼를 사용해 시스템 호출 수를 줄여 보았습니다. 이제 시스템 호출이 내부적으로는 어떻게 동작하는지 알아보고 ‘마른 수건을 더 쥐어짤’ 방법을 알아보겠습니다.

OS<sup>Operating System</sup>은 하드디스크나 마우스, 키보드, 프린터, 네트워크 카드, 메모리 등 의 자원에 대해 전적인 제어권을 가집니다. 프로그래머는 OS가 허용한 시스템 호출 함수를 사용해 해당 자원에 접근합니다. 다시 말해, 시스템 호출을 사용한다는 것은 OS에게 해당 작업에 대해 자기 대신 수행해 달라는 요청을 전달하는 것입니다.

이러한 위임 모델은 데이터가 OS의 메모리 영역(커널 영역이라고 통칭되는)에 복제된 이후 프로세스에 전달됩니다. 간략히 도식화하면 다음과 같습니다.

그림 4-10 위임 모델의 도식화



다시 말해 하드웨어를 사용해 데이터를 획득하려고 시스템 호출 함수를 호출하면 OS는 필요한 데이터를 하드웨어에서 읽어들여 커널 영역으로 복제하고 그 데이터를 다시 유저 영역으로 전달합니다. 그런데 DMA(Direct Memory Access) 지원이 가능한 버스는 미리 메모리 영역의 일부를 할당해 하드웨어상의 데이터를 메모리상의 DMA 영역으로 전달시킬 수 있습니다. 이때 DMA 영역에 접근할 수 있다면 커널 영역에서 유저 영역으로의 복사를 줄일 수 있습니다. 이런 기능은 NIO로 통칭되는 java.nio 패키지를 사용해 구현하며, NIO와 전통적인 블로킹 IO에 대한 비교 및 분석은 7장에서 다시 다룹니다.

## 4.7 더 생각해 볼 문제

- 1장에서 청크 인코딩을 사용해 점진적으로 데이터를 전달하는 방식에 대해 살펴보았습니다. 청크 인코딩된 요청을 처리할 수 있게 HTTP 프로토콜 분석기를 만들어 보십시오.
- 1장에서 multipart/form-data 형식을 사용해 바이너리 데이터를 전송하는 방식도 알아보았습니다. 이 형식을 분석하기 위한 HTTP 프로토콜 분석기를 만들어 보십시오.
- 성능 개선 1 - 버퍼의 사용 예제를 java NIO를 써서 구현하여 보고 성능을 확인하여 보십시오.

## 5 | 서블릿 관리자

리더에 대한 유일한 정의는 ‘추종자를 거느린 사람이다’라는 것이다.

어떤 리더를 신뢰하기 위해서 반드시 그를 인간적으로 좋아할 필요는 없다.

또한 그와 의견을 같이 할 필요도 없다. 신뢰라는 것은 리더가 언행을 일치한다는 데 대한 확신이다. 그것은 아주 낡은 표현 방식인 성실이라는 것에 대한 믿음이기도 하다.

- 피터 드러커(『프로페셔널의 조건』 중에서)

웹 서비스는 서블릿 컨테이너가 전달받은 HTTP 메시지를 서블릿에 전달하고, 이 서블릿이 작동함으로써 결과가 클라이언트에 반환되는 과정을 말합니다. 이 장에서는 서블릿을 구동하기 위해 서블릿 컨테이너가 수행하는 일들을 설명합니다.

4장에서는 웹 브라우저에서 전달받은 HTTP 요청을, 메시지 헤더와 메시지 바디로 분리하는 과정을 살펴보았습니다. 서블릿 컨테이너는 메시지 헤더의 첫 라인(시작 라인)을 HTTP 메서드, 요청 URL, HTTP 버전으로 다시 분리합니다. 그리고 요청 URL을 분석하여 자신이 유지관리하는 서블릿 중 하나를 선택한 후, 서블릿의 service 메서드를 호출해 HTTP 메시지를 처리하도록 위임합니다. 그렇다면 실제 서블릿에서 구현된 개별 doGet 메서드와 doPost 메서드는 어떻게 구별하여 호출하는 것일까요?

3장의 HttpServlet 절에서도 언급했듯이, doGet 메서드와 doPost 메서드 구분은 서블릿의 직전 상위 클래스인 javax.servlet.http.HttpServlet에 구현된 내용에 따라 결정됩니다. 다시 한번 강조하면, 서블릿 컨테이너는 최상위 Servlet 인터페이스만 인식하고 해당 메서드의 service 메서드를 호출합니다. 그러면 메서드 호출은 클래스의 상속구조를 따라 해당 Servlet 인터페이스의 service 메서드, GenericServlet 클래스의 service 메서드, HttpServlet 클래스의 service 메서드까지 내려갑니다. 그리고 전달받은 HTTP 요청의 메서드 종류에 따라

`HttpServlet doGet`이나 `HttpServlet doPost` 등으로 분기해 다시 위임됩니다. 따라서 사용자가 작성한 서블릿이 `HttpServlet` 클래스를 상속받은 후 `doGet` 메서드를 오버라이딩했다면 사용자가 작성한 `doGet`으로 웹 브라우저가 전달한 HTTP 메시지가 전달됩니다.

## 5.1 웹 애플리케이션

일반적으로 “웹 애플리케이션이 무엇인가?”라는 질문에 “하위에 WEB-INF 디렉터리를 가지는 디렉터리 구조이며 최상위 디렉터리는 해당 웹 컨텍스트의 최상위 문서 루트가 된다”는 내용과 WEB-INF 아래 있는 `web.xml`에 서블릿 설정이 들어 간다는 정도가 일반적인 답변입니다. 하지만 서블릿 컨테이너의 입장에서 웹 애플리케이션은 독립적인 클래스로 더러를 공유하는 웹 프로그램의 모임의 하나입니다. 웹 애플리케이션이 독립적인 클래스로 더러를 가지게 된 것은 다음과 같은 이유 때문입니다.

서블릿 명세에서 웹 애플리케이션을 정의할 때 디렉터리 구조를 위와 같이 결정하고 WAR<sup>Web application Archive</sup> 포맷으로 아카이빙 방법까지 정의한 것은, 웹 애플리케이션 역시 하나의 독립적인 애플리케이션으로 취급하려는 의도에서 기인했습니다. 물론, 서블릿과 웹 애플리케이션 자체만으로는 단독 실행이 불가능한데 어떻게 독립적인 단위로 볼 수 있을지 의문이 들 수 있습니다.

하지만 웹 애플리케이션과 서블릿이 지켜야 하는 규약인 서블릿 명세에는 웹 서비스를 위해 필요한 인터페이스뿐 아니라 서블릿 컨테이너 내에 초기화 방법과 구동에 필요한 설정이 모두 정의돼 있습니다. 결과적으로 서블릿 컨테이너는 임의의 웹 애플리케이션이나 서블릿도 배치, 구동할 수 있어 개별 서블릿 컨테이너들은 성능상의 차이는 있을지언정 기능적인 면에서는 서블릿 명세에 정의된 대로 동작하는 일종의 범용 실행 환경으로 간주될 수 있습니다(서블릿 명세를 기반으로 서블릿 컨테이너가 웹 애플리케이션을 내부로 끌여들이는 과정을 서블릿 컨테이너에 웹 애

플리케이션을 배치(deploy한다고 표현합니다)<sup>01</sup>.

웹 애플리케이션이 한 번 작성되면 다른 서블릿 컨테이너에서도 동일하게 동작해야 하므로 웹 애플리케이션은 필요한 모든 것<sup>02</sup>을 포함해야 합니다. 또한 서블릿 컨테이너는 서로 다른 웹 애플리케이션이 동일한 서블릿 컨테이너에서 동작할 때 서로 간의 간섭이 없이 독자적으로 동작하는 것도 보장해야 합니다. 간섭이 발생할 수 있는 상황의 예로는 풀 패키지 이름이 우연히도 같은 두 클래스<sup>03</sup>가 서로 다른 웹 애플리케이션 내에서 존재하는 경우입니다. 이런 경우 두 클래스가 모두 안전하게 서로 구별돼 호출되고 동작할 수 있어야 합니다.

이를 위해 서블릿 컨테이너는 웹 애플리케이션마다 독자적인 클래스로더를 유지합니다. 이런 방식으로 앞서 설명한 웹 애플리케이션 간의 간섭이 발생하지 않게 합니다. 따라서 두 개의 웹 애플리케이션에 내용은 다르지만 풀 패키지명이 같은 클래스가 동시에 있을 수 있고 동작할 수 있습니다.

## 5.2 인터페이스를 사용한 컴포넌트와 컨테이너의 분리

웹 애플리케이션 하나가 서블릿 컨테이너 안으로 배치되면 서블릿 컨테이너는 먼저 해당 웹 애플리케이션 전용 클래스로더를 생성합니다. 그후 WEB-INF/classes 디렉터리와 WEB-INF/lib 디렉터리 내에 있는 클래스 파일과 .jar 아카이브 파일들을 해당 클래스로더를 사용해 로딩합니다.

웹 애플리케이션을 호출하는 HTTP 요청이 들어왔을 때 WEB-INF/web.xml에 정의된 서블릿 매핑(어떤 요청 URL이 들어오면 특정 서블릿에게 넘겨주라는 정

---

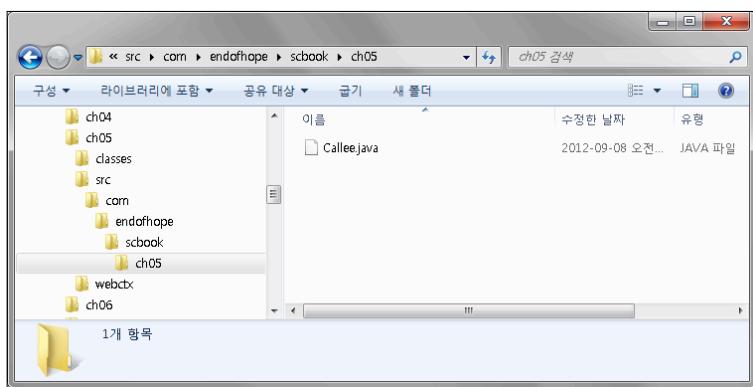
01 이러한 의도의 저변에는 여러 웹 애플리케이션이 일종의 웹 애플리케이션 시장에서 자유롭게 거래되는 것을 꿈꿨던 점도 있습니다. “Write once, run anywhere”라는 자바의 영원한 구호를 생각해보면 이 의도가 그렇게 낯설지는 않습니다. 어찌보면 웹 애플리케이션도 결국은 프로그램이니 한번 작성되면 어떤 실행 환경, 즉 서블릿 컨테이너 위에서도 동일한 동작을 해야 하는 것이 당연하게 생각될 것입니다.

02 내부적으로 호출하는 라이브러리와 설정 파일

03 fully-qualified class name

의)에 따라서 웹 애플리케이션 클래스로부터 서블릿을 찾아 요청을 전달합니다. 이런 방식은 서블릿 컨테이너에 특화된 것은 아니며 인터페이스를 사용해 독립적인 컴포넌트와 해당 컴포넌트를 담는 컨테이너를 분리하는 일반적인 전략입니다. 다음은 임의의 인터페이스인 Callee를 사용해 인터페이스의 컨테이너와 인터페이스 구현체가 어떻게 협업할 수 있을지 알아보겠습니다.

먼저 공통으로 사용할 Callee 인터페이스를 다음 위치에 작성합니다.



이 인터페이스는 javax.servlet.Servlet 인터페이스에 해당합니다.

---

```
package com.endofhope.scbook.ch05;

public interface Callee {
    public void setName(String name);
    public String getName();
}
```

---

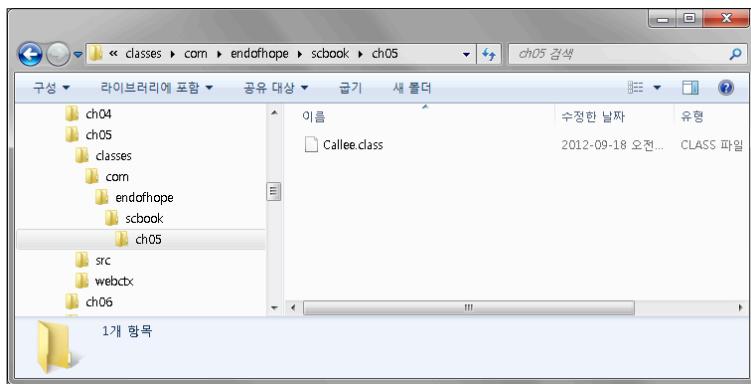
다음과 같이 컴파일합니다.

- 컴파일 명령 : javac -d classes src\com\endofhope\scbook\ch05\Callee.java

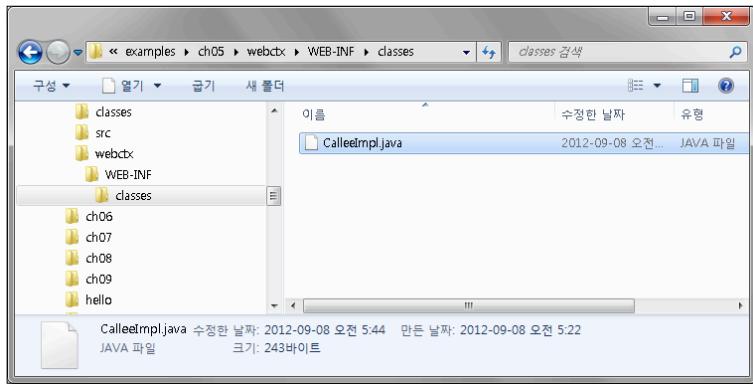


```
관리자: C:\Windows\System32\cmd.exe
C:\examples\ch05>javac -d classes src\com\endofhope\scbook\ch05\Callee.java
C:\examples\ch05>
```

ch05\classes 디렉터리 아래 다음과 같이 Callee의 바이트 코드가 생성됐습니다.



다음으로 Callee 인터페이스의 구현체인 CalleeImpl 클래스입니다. 웹 애플리케이션 안에 포함된 서블릿에 해당합니다.



---

```
import com.endofhope.scbook.ch05.Callee;

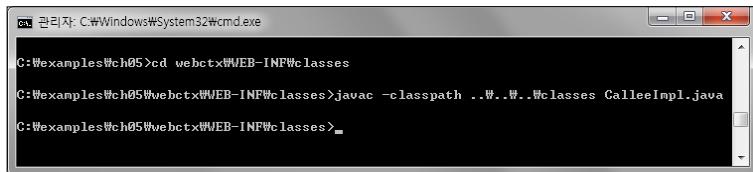
public class CalleeImpl implements Callee{
    private String name = "orgName";
    public void setName(String name){
        this.name = name;
    }
    public String getName(){
        return name;
    }
}
```

---

다음과 같이 컴파일합니다. 먼저 webctx\WEB-INF\classes 디렉터리로 이동 후 Callee 인터페이스를 구현했으므로 -classpath 옵션을 사용해 Callee의 바이트 코드 위치를 지정합니다.

- cd webctx\WEB-INF\classes

- Callee의 바이트코드 위치를 지정 명령: javac -classpath ..\..\..\classes  
CalleeImpl.java



```
관리자: C:\Windows\System32\cmd.exe
C:\examples\ch05>cd webctx\WEB-INF\classes
C:\examples\ch05\webctx\WEB-INF\classes>javac -classpath ..\..\..\classes CalleeImpl.java
C:\examples\ch05\webctx\WEB-INF\classes>_
```

이제 webctx\classes 디렉터리 아래 CalleeImpl의 바이트 코드가 생성된 것을 확인할 수 있습니다. 다음으로 컨테이너 역할을 담당할 CL을 작성하겠습니다. 컨테이너 구현은 다음과 같습니다.

---

```
package com.endofhope.scbook.ch05;

import java.io.File;
import java.io.FileFilter;
import java.net.MalformedURLException;
import java.net.URL;
import java.net.URLClassLoader;
import java.util.ArrayList;
import java.util.List;

public class CL {
    public static void main(String[] args) throws ClassNotFoundException{
        CL cl = new CL();
        cl.action();
    }
    private void action(){
        init();
        load();
    }
}
```

```
}

private void init(){
    String contextPath = "C:/examples/ch05/webctx";
    String classesPath =
contextPath.concat(File.separator).concat("WEB-INF").concat(File.separator
).concat("classes");
    String libPath =
contextPath.concat(File.separator).concat("WEB-INF").concat(File.separator
).concat("lib");
    File classes = new File(classesPath);
    List<URL> urlList = new ArrayList<URL>();
    if(classes.exists()){
        try {
            urlList.add(classes.toURI().toURL());
        } catch (MalformedURLException e) {
            e.printStackTrace();
        }
    }
    File lib = new File(libPath);
    if(lib.exists()){
        try {
            FileFilter ff = new FileFilter(){
                @Override
                public boolean accept(File pathname) {
                    boolean result = false;
                    if(pathname.getName().endsWith(".jar")){
                        result = true;
                    }
                    return result;
                }
            };
            File[] jarList = lib.listFiles(ff);
        }
    }
}
```

```
        for(File file : jarList){
            urlList.add(file.toURI().toURL());
        }
    } catch (MalformedURLException e) {
        e.printStackTrace();
    }
}

URL[] urls = new URL[urlList.size()];
for(int i=0; i<urls.length; i++){
    urls[i] = urlList.get(i);
}
urlCL = new URLClassLoader(urls,
Thread.currentThread().getContextClassLoader());
Thread.currentThread().setContextClassLoader(urlCL);
}

private void load(){
Class calleeClass = null;
try {
    calleeClass = urlCL.loadClass("CalleeImpl");
    Callee calleeInstance = (Callee)calleeClass.newInstance();

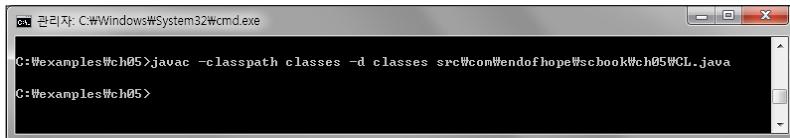
    System.out.println(calleeInstance.getName());
} catch (ClassNotFoundException e) {
    e.printStackTrace();
} catch (InstantiationException e) {
    e.printStackTrace();
} catch (IllegalAccessException e) {
    e.printStackTrace();
}
}

URLClassLoader urlCL = null;
}
```

---

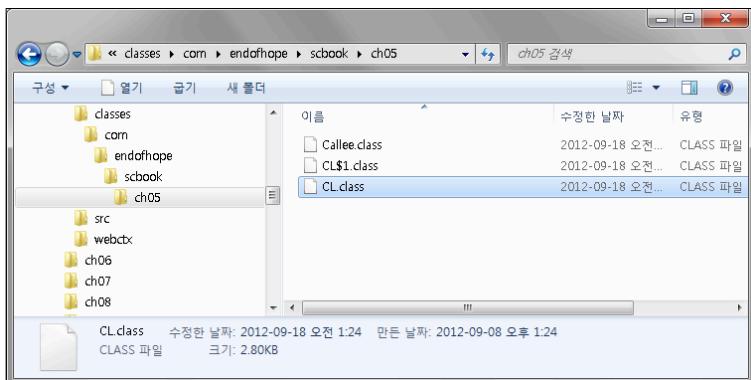
컨테이너인 CL.java를 ‘C:\examples\ch05\src\com\endofhope\scbook\ch05’에 두고 다음과 같이 컴파일합니다. Callee 인터페이스가 필요하므로 - classpath 옵션을 추가해 컴파일합니다.

- 컴파일 명령 : javac -classpath classes -d classes src\com\endofhope\scbook\ch05\CL.java

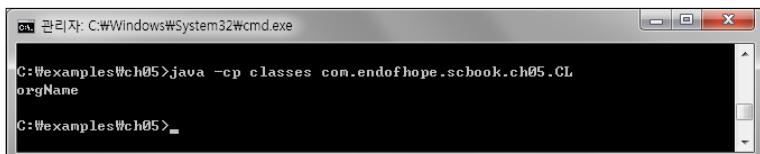


```
cmd 관리자: C:\Windows\System32\cmd.exe
C:\examples\ch05>javac -classpath classes -d classes src\com\endofhope\scbook\ch05\CL.java
C:\examples\ch05>
```

그림 5-1 Callee 인터페이스 컴파일 결과로 생성된 파일



CL을 호출하면 다음과 같은 결과를 볼 수 있습니다.



```
cmd 관리자: C:\Windows\System32\cmd.exe
C:\examples\ch05>java -cp classes com.endofhope.scbook.ch05.CL
orgName
C:\examples\ch05>
```

Callee 인터페이스의 구현체인 CalleeImpl과 컨테이너인 CL 사이에는 직접적인 상호 참조가 없다는 점에 주의해야 합니다. 다시 말해, CL.java 파일 내에 CalleeImpl 클래스를 임포트하지 않았으며 CL은 CalleeImpl 클래스 파일의 물리적인 위치<sup>04</sup>와 CalleeImpl이 Callee 인터페이스를 구현했다는 것만 알고 있습니다. 하지만 CL의 실행 결과는 CalleeImpl의 name인 orgName을 반환했습니다. 어떻게 이런 결과가 가능한지 CL을 자세히 살펴보겠습니다.

CL은 내부적으로 init과 load, 두 메서드로 구분됩니다. init 메서드에서는 웹 애플리케이션의 루트 아래 WEB-INF 디렉터리를 찾고, 그 아래에 있는 classes 디렉터리 자체를 URLClassLoader에 추가합니다. 그리고 lib 디렉터리 아래에서는 .jar 파일을 파일 필터를 이용하여 걸러낸 다음 찾아낸 파일을 URLClassLoader에 추가합니다.

이번 예제에서는 classes 디렉터리 아래에 CalleeImpl 클래스가 바이트코드로 있었으므로, 해당 URLClassLoader가 CalleeImpl 클래스를 찾을 수 있었습니다. 이제 load 메서드 안에서는 URLClassLoader의 load 메서드를 사용해, 직접 CalleeImpl을 꺼내어 인스턴스화 합니다. 단, 컴파일 타임에서는 CalleeImpl 클래스 자체에 접근할 수 없으므로(런타임 중에 URLClassLoader에 있는 load 메서드의 매개변수로 클래스명이 결정되므로 접근할 수 없음) 직접 CalleeImpl을 지정할 수 없습니다. 따라서 컨테이너에서는 미리 약속돼 공유된 Callee 인터페이스로 캐스팅해 인터페이스의 메서드인 getName을 호출합니다. 내부적으로 URLClassLoader에서 반환된 해당 인터페이스의 인스턴스는 CalleeImpl이므로 67 행의 calleeInstance.getName() orgName을 반환한 것을 확인할 수 있습니다.

---

04 22행의 String classesPath에 해당합니다.

위 예제에서 Callee 인터페이스, CalleeImpl 구현체, CL은 Servlet 인터페이스, 프로그래머가 작성하는 서블릿, 서블릿 컨테이너의 역할과 상응합니다(유사성을 더하기 위해 일부러 WEB-INF/classes 디렉터리를 구성했습니다). 다시 말해서, 서블릿 컨테이너는 웹 애플리케이션이 배치될 때 클래스로더를 생성해 특정 위치(WEB-INF/classes, WEB-INF/lib)에 있는 클래스 파일과 jar 파일을 해당 클래스로더에 로딩합니다. 이런 절차로 인해 서블릿 컨테이너는 자신보다 나중에 작성된 웹 애플리케이션을 배치 과정만 거치면 문제없이 구동할 수 있습니다.

지금까지 동작에 대해 알아보았습니다. 다음에는 실질적으로 서블릿 관리자가 제공해야 하는 추가 기능에 대해 알아보겠습니다.

### 5.3 HTTP 요청이 서블릿에 가기까지

4장에서 클라이언트 웹 브라우저가 전달한 HTTP 메시지를 분석해 보았습니다. 메시지 헤더와 메시지 바디를 구분하고 메시지 헤더의 첫 행을 시작행이라는 이름으로 특별하게 취급했다는 것을 기억하십시오. 요청 URL은 시작행의 일부이며, 이 URL을 분석해 요청이 어떤 웹 애플리케이션을 요청한 것인지, 그리고 웹 애플리케이션에 포함된 어떤 서블릿이 해당 요청을 처리하는지가 결정됩니다. 서블릿 컨테이너의 서블릿 관리자는 서블릿을 로딩한 후 해당 서블릿의 service 메서드를 호출합니다. 이때 전달받은 매개변수는 해당 메서드의 파라미터인 HttpServletRequest에 담겨 전달됩니다.

요청 URL은 일반적으로 다음과 같습니다.

- 요청 URL 포맷: /컨텍스트이름/서블릿매핑URL?매개변수

컨텍스트 이름은 서블릿 컨테이너에 배치된 웹 애플리케이션마다 유일하게 부여된 이름입니다. 이런 컨텍스트 이름 매핑의 실제적인 구현 방식은 서블릿 명세에서 다루지 않습니다. 따라서 서블릿 컨테이너 구현체마다 방식이 조금씩 다릅니다. 주로

서블릿 컨테이너마다 다른 설정파일을 사용해 시작할 때 읽어들이거나 배치 관리 용 툴을 제공해 런타임 시에 웹 애플리케이션을 특정한 컨텍스트로 배치합니다. 톰캣은 webapps라는 특별한 디렉터리를 감시하다가 해당 디렉터리 아래에 웹 애플리케이션이 있다는 사실을 포착하면 자동으로 배치하는 기능을 제공합니다.

서블릿 매팅 URL은 웹 애플리케이션이 가진 WEB-INF 디렉터리 안의 web.xml에 정의되어 있습니다. 다음은 해당 파일의 예입니다.

---

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<web-app xmlns="http://java.sun.com/xml/ns/javaee"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
        http://java.sun.com/xml/ns/javaee/web-app_3_0.xsd"
    version="3.0" metadata-complete="true">
    <servlet>
        <servlet-name>DummyName</servlet-name>
        <servlet-class>DummyServlet</servlet-class>
    </servlet>
    <servlet-mapping>
        <servlet-name>DummyName</servlet-name>
        <url-pattern>/dummy</url-pattern>
    </servlet-mapping>
</web-app>
```

---

web.xml이 포함된 웹 애플리케이션이 서블릿 컨테이너 안에 /WebApp이라는 컨텍스트로 매팅됐다고 가정하겠습니다. 앞서 언급한 바와 같이 웹 애플리케이션과 컨텍스트에 대한 매팅은 서블릿 컨테이너마다 제공하는 방법이 조금씩 다릅니다. '/WebApp/dummy?who=me'라는 요청 URL을 받은 서블릿 컨테이너가 어떻게 동작하는지 알아보겠습니다.

서블릿 컨테이너는 먼저 /WebApp을 보고 어떤 웹 애플리케이션인지 판단하고 그 웹 애플리케이션의 WEB-INF/web.xml을 확인합니다. WEB-INF/web.xml 안의 servlet-mapping 엘리먼트를 순회하면서 하위 엘리먼트인 url-pattern의 값에 일치하는 servlet-name 엘리먼트 값을 찾습니다. 이제 /dummy가 web.xml에 정의된 URL 패턴 중 서블릿 이름이 DummyName이라는 패턴과 일치하고 servlet-name의 값이 DummyName인 것을 확인합니다. 다음으로 servlet 엘리먼트 중에서 하위 엘리먼트인 servlet-name이 앞서 찾은 DummyName인 것을 찾아 servlet-class를 확인합니다. 이번 예에서는 DummyServlet입니다. 따라서 DummyServlet 클래스의 service 메서드를 호출하는 데 매개변수로 넘겨줄 HttpServletRequest 객체의 매개변수명 who에는 me라는 값이 들어있어야 합니다.

지금까지의 내용을 코드로 나타내면 다음과 같습니다.

---

```
// ContextPath 를 보고 servlet context 찾기를 시도한다.  
WebContextManager webContextManager =  
    server.getWebContainer().getWebContextManager();  
ServletContextImpl servletContextImpl =  
    webContextManager.getServletContext(panzerRequest.getContextPath()); ❶  
  
if(servletContextImpl == null){  
    // 오류 처리. 종료.  
}  
  
// HttpServletRequestImpl을 생성한다.  
HttpServletRequestImpl httpServletRequestImpl =  
    new HttpServletRequestImpl(panzerRequest, servletContextImpl); ❷  
// HttpServletResponseImpl을 생성한다.  
HttpServletResponseImpl httpServletResponseImpl =  
    new HttpServletResponseImpl(panzerResponse,
```

```
server.getMessageQueue("scatter_queue"), inMessage); ③

ServletManager servletManager = servletContextImpl.getServletManager(); ④
String servletName = panzerRequest.getServletName();
String filterServletPath = "/" + servletName;
httpServletRequestImpl.setServletPath(filterServletPath);
Servlet targetServlet = servletManager.getServlet(filterServletPath);

try {
    if(targetServlet == null){
        SystemSender.sendError(httpServletResponseImpl,
            HttpServletResponse.SC_NOT_FOUND, "Not found");
        return;
    }
    targetServlet.service(httpServletRequestImpl, httpServletResponseImpl);
} catch (ServletException e) {
    logger.log(Level.WARNING, "servlet execute error", e);
    try {

        httpServletResponseImpl.getWriter().write(StringUtil.getPrettyErrorHead(
            e.getMessage()));
        e.printStackTrace(httpServletResponseImpl.getWriter());

        httpServletResponseImpl.getWriter().write(StringUtil.getPrettyErrorTail());
    } catch (IOException e1) {
    }
} catch (IOException e) {
    logger.log(Level.WARNING, "servlet io error", e);
    try {
        httpServletResponseImpl.getWriter().write(StringUtil.getPrettyErrorHead(
            e.getMessage()));
        e.printStackTrace(httpServletResponseImpl.getWriter());
    }
```

```
        httpServletResponseImpl.getWriter().write(StringUtil.getPrettyErrorTail());
    } catch (IOException e1) {
    }
}catch (Throwable t){
    logger.log(Level.WARNING, "servlet error", t);
    try {
        httpServletResponseImpl.getWriter().write(StringUtil.getPrettyErrorHead(
            t.getMessage()));
        t.printStackTrace(httpServletResponseImpl.getWriter());
        httpServletResponseImpl.getWriter().write(StringUtil.getPrettyErrorTail());
    } catch (IOException e1) {
    }
} finally {
    try {
        httpServletResponseImpl.postService();
    } catch (IOException e) {
        logger.log(Level.WARNING, "postService io error", e);
    }
}
```

---

먼저 해당 요청이 어느 웹 애플리케이션에서 처리해야 하는지를 결정합니다(❶). 웹 애플리케이션이 결정되면 그에 의거하여 HttpServletRequest와 HttpServletResponse를 생성합니다(❷, ❸). 이 두 객체는 추후 처리할 서블릿이 결정되면 service 메서드의 매개변수로 전달될 것입니다. 이제 웹 애플리케이션의 서블릿 관리자에게 요청 URL에서 분석된 서블릿 이름을 넘겨주어 실재로 서블릿 관리자가 관리하던 서블릿 인스턴스를 넘겨받습니다(❹).

다음은 요청 URL의 일부인 filterServletPath를 사용해 서블릿 관리자에게 Servlet 인스턴스를 가져오는 예입니다.

---

```
ServletManager servletManager = servletContextImpl.getServletManager();
String servletName = panzerRequest.getServletName();
String filterServletPath = "/" + servletName;
httpServletRequestImpl.setServletPath(filterServletPath);
Servlet targetServlet = servletManager.getServlet(filterServletPath);
```

---

드디어 아래와 같이 서블릿의 service 메서드를 호출할 수 있게 되었습니다.

---

```
targetServlet.service(httpServletRequestImpl, httpServletResponseImpl);
```

---

지금까지의 내용을 다시 정리하면, 웹 애플리케이션이 서블릿 컨테이너에 배치되면 웹 애플리케이션의 몇몇 경로, 즉 WEB-INF/classes, WEB-INF/lib에서 찾을 수 있는 .class나 .jar 형식의 바이트 코드를 읽어들여 해당 웹 애플리케이션 고유의 클래스로더에 등록합니다. 이후 서블릿 컨테이너는 웹 애플리케이션 설정이 정의된 WEB-INF/web.xml에 따라 어떤 HTTP 요청 URL 패턴에 대해 어떤 서블릿이 불려야 하는지 결정한 후 해당 인스턴스의 service 메서드를 호출합니다.

## 5.4 서블릿 관리자

서블릿 명세의 요구 사항은 아니나 성능의 문제를 고려할 때 서블릿 컨테이너는 웹 애플리케이션이 배치될 때 미리 로딩해야 할 서블릿 인스턴스를 캐시 처리하는 것이 일반적입니다. 이렇게 캐시를 사용함으로써 요청 시마다 동일한 서블릿을 클래스로더에서 반복해서 인스턴스화하는 것을 회피할 수 있습니다. 이를 위해 웹 애플리케이션마다 해당 웹 애플리케이션의 서블릿을 관리하는 서블릿 관리자가 있습니다.

다음 소스코드는 웹 애플리케이션이 배치될 때 서블릿 관리자가 WEB-INF/web.xml에 정의된 서블릿 이름, 클래스명, 매개변수를 읽어내 Map 타입 객체인 servletInfoMap에 저장하는 과정을 설명합니다.

```
servletInfoMap = new HashMap<String, ServletInfo>();
servletInfoMap.put(①
    ServletManager.RESOURCE_SERVLET,
    new ServletInfo(
        ServletManager.RESOURCE_SERVLET,
        "com.endofhope.neurasthenia.webcontainer.servlet.ResourceServlet"));
ServletInfo jspServletInfo = new ServletInfo(②
    ServletManager.JSP_SERVLET, "org.apache.jasper.servlet.JspServlet");
servletInfoMap.put(ServletManager.JSP_SERVLET, jspServletInfo);

Document doc = saxb.build(new File(webinfPath + "web.xml"));
Element webAppElement = doc.getRootElement();

List<Element> contextParamElementList =
webAppElement.getChildren("context-param");
for(Element contextParamElement : contextParamElementList){
    servletContextImpl.getInitParamMap().put(
        contextParamElement.getChildTextTrim("param-name"),
        contextParamElement.getChildTextTrim("param-value"));
}

List<Element> servletElementList = webAppElement.getChildren("servlet");
for(Element servletElement : servletElementList){
    Element servletNameElement = servletElement.getChild("servlet-name");
    Element servletClassElement = servletElement.getChild("servlet-class");
    String servletName = servletNameElement.getTextTrim();
    String servletClass = servletClassElement.getTextTrim();
```

```
ServletInfo servletInfo = new ServletInfo(servletName, servletClass);
List<Element> initParamElementList =
servletElement.getChildren("init-param");
for(Element initParamElement : initParamElementList){
    servletInfo.getInitParamMap().put(
        initParamElement.getChildTextTrim("param-name"),
        initParamElement.getChildTextTrim("param-value"));
}
servletInfoMap.put(servletName, servletInfo);
}
```

---

정적 리소스에 대한 요청과 JSP에 대한 요청을 처리하는 특수 서블릿에 관한 정보를 먼저 등록합니다(❶, ❷). 그다음 WEB-INF/web.xml의 노드를 순회하면서 정의된 정보를 servletInfoMap에 추가합니다.

이제 클래스로더에 WEB-INF/classes, WEB-INF/lib 등의 바이트 코드가 존재하는 위치를 알려주어 추후 인스턴스화가 필요할 때 해당 위치에서 찾을 수 있게 합니다.

---

```
List<URL> urlList = new ArrayList<URL>();
String webinfPath = servletContextImpl.getRealContextPath() +
File.separator + "WEB-INF" + File.separator;
File classes = new File(webinfPath + "classes");
File lib = new File(webinfPath + "lib");

WebContextManager webContextManager =
servletContextImpl.getWebContextManager();
String commonClasspath = webContextManager.getCommonClassPath();
if(classes.exists()){
    try {
```

```
urlList.add(classes.toURI().toURL());
} catch (MalformedURLException e) {
    logger.log(Level.WARNING, "WEB-INF/classes invalid", e);
}
}

if(lib.exists()){
try {
    FileFilter ff = new FileFilter(){
        @Override
        public boolean accept(File pathname) {
            boolean result = false;
            if(pathname.getName().endsWith(".jar")){
                result = true;
            }
            return result;
        }
    };
    File[] jarList = lib.listFiles(ff);
    for(File file : jarList){
        urlList.add(file.toURI().toURL());
    }
} catch (MalformedURLException e) {
    logger.log(Level.WARNING, "WEB-INF/lib invalid", e);
}
}

URL[] urls = new URL[urlList.size()];
for(int i=0; i<urls.length; i++){
    urls[i] = urlList.get(i);
}
urlCL = new URLClassLoader(urls,
    Thread.currentThread().getContextClassLoader());
```

①

클래스로더는 URL 클래스로더를 사용했으며 서블릿 컨테이너가 로딩된 스레드의 클래스로더를 부모 클래스로더로 생성했습니다(❶).

J2EE 명세를 만족하는 웹 애플리케이션 서버는 EJB<sup>05</sup>(이하 EJB)에 대한 지원이 필요합니다. 지금까지 설명한 웹 애플리케이션과 비슷하게, EJB 역시 독립적인 아카이빙 방법이 존재하고 EJB 명세를 만족하기만 하면 서로 독립적으로 작성된 EJB 컨테이너와 EJB 컴포넌트가 추가 수정 없이 잘 작동합니다. 이를 위해, EJB 컴포넌트마다 독립적인 클래스로더를 유지하는 것이 일반적입니다. 여기서 발생하는 문제는 다음과 같습니다.

웹 애플리케이션은 각기 독립적인 클래스로더를 가지므로 서로 간에 호출할 수 없습니다. 그런데 EJB 컴포넌트도 독립적인 클래스로더를 가지고 있다면 특정 서블릿에서 EJB 컴포넌트를 어떻게 호출할 수 있을까 하는 문제가 발생합니다.

원칙적으로는 EJB 컴포넌트는 RMI<sup>06</sup> 기반의 원격 호출(네트워크 소켓 기반 통신) 방식으로 사용되므로 외부 웹 애플리케이션의 서블릿에서 호출 가능합니다. 하지만 성능상의 이유로 원격 호출이 아닌 메서드 콜로도 EJB를 호출 할 수 있습니다. 이와 같은 경우를 해결하는 방법은 여러 웹 애플리케이션 서버 벤더마다 조금씩 다릅니다.

일부 웹 애플리케이션 서버 벤더는 웹 애플리케이션의 클래스로더를 생성할 때 부모 클래스로더를 EJB의 클래스로더로 설정해 해당 문제를 해결했습니다. 이와 같은 경우에는 부모 클래스로더가 로딩한 클래스를 자식 클래스로더가 로딩한 클래스에서 접근할 수 있다는 클래스로더 상속 구조를 이용한 것입니다. 따라서 자식 클래스로더에서 로딩한 서블릿에서 부모 클래스로더가 로딩한 EJB를 호출해 사용 가능합

---

05 EJB<sup>TM</sup>는 Oracle 사의 트레이드 마크입니다. 본문에서는 EJB로 표기했습니다.

06 원격 메서드 호출(Remote Method Invocation)

니다. 이 외에도 OSGi<sup>07</sup>를 기반으로 해당 기능을 구현한 경우도 있습니다.

이제 servletInfoMap의 내용을 기반으로 서블릿 클래스를 로딩하고(❶), 서블릿 인스턴스를 생성하겠습니다(❷).

---

```
Set<String> servletNameSet = servletInfoMap.keySet();
Iterator<String> servletNameIter = servletNameSet.iterator();
Class<?> servletClass = null;
while(servletNameIter.hasNext()){
    String servletName = servletNameIter.next();

    ServletConfigImpl servletConfigImpl = new ServletConfigImpl(servletName,
        servletContextImpl);
    ServletInfo servletInfo = servletInfoMap.get(servletName);
    String servletClassName = servletInfo.getServletClass();
    Map<String, String> initParamMap = servletInfo.getInitParamMap();
    Set<String> keySet = initParamMap.keySet();
    Iterator<String> keyIter = keySet.iterator();
    while(keyIter.hasNext()){
        String key = keyIter.next();
        servletConfigImpl.addServletConfigInitParam(key, initParamMap.get(key));
    }
    try {
        servletClass = urlCL.loadClass(servletClassName); ❶
        Servlet cachedServlet = (Servlet)servletClass.newInstance(); ❷
        cachedServlet.init(servletConfigImpl);
        servletCache.put(servletName, cachedServlet);
        logger.log(Level.FINE, "servelt {0}, {1} initialized",
            new Object[]{servletName, cachedServlet});
    } catch (ClassNotFoundException e) { ❸

```

---

07 open service gateway initiative

```
        logger.log(Level.WARNING, "servlet class [" + servletName + "] not found", e);
    } catch (InstantiationException e) {
        logger.log(Level.WARNING, "servlet class [" + servletName + "] is
            not valid servlet", e);
    } catch (IllegalAccessException e) {
        logger.log(Level.WARNING, "servlet class [" + servletName + "] can not
            accessible", e);
    } catch (ServletException e) {
        logger.log(Level.WARNING, "servlet [" + servletName + "] exception", e);
    }
}
```

---

앞에서 서블릿 생명주기에 대해 언급하면서 서블릿 컨테이너가 서블릿을 처음 배치할 때 서블릿의 init 메서드가 호출된다고 설명했습니다. ❸행에서 서블릿 컨테이너가 init 메서드를 호출하는 것을 확인할 수 있습니다.

이후 servletCache에 해당 인스턴스를 넣어두고 앞으로 들어올 HTTP 요청에 대기합니다.

---

```
public Servlet getServlet(String servletPath){
    String servletName = null;
    if(servletPath != null
        &&(servletPath.endsWith(".jsp")
        || servletPath.endsWith(".JSP")
        || servletPath.endsWith(".jspx")
        || servletPath.endsWith(".JSPX"))){
        servletName = ServletManager.JSP_SERVLET;
    }else{
        servletName = getServletName(servletPath);
    }
}
```

```
if(servletName == null){  
    servletName = ServletManager.RESOURCE_SERVLET;  
}  
return servletCache.get(servletName);  
}
```

---

HTTP 요청 URL에 따라 web.xml에 정의된 서블릿이 결정되면 위와 같이 servletCache에서 해당 서블릿 인스턴스를 반환합니다. 만약 URL 패턴에 의해 정의된 서블릿이 없다면 해당 요청이 JSP에 대한 요청인지 확인해 JSP 전용 서블릿으로 넘기거나 혹은 정적 리소스에 대한 처리로 판단해 리소스 서블릿에게 해당 처리를 위임합니다.

## 5.5 더 생각해 볼 문제

1. 임의로 java.lang.String 클래스를 만들어 적용할 수 없는 이유를 클래스로더의 상속관계와 연관지어 설명해 보십시오.
2. 클래스 로더를 수정하면 앞서 1번에서 언급한 java.lang.String 클래스를 바꾸는 것도 가능합니다. 어떻게 수정하면 될지 생각해 보십시오(java.lang.Classloader의 loadClass 메서드 구현을 살펴보시오).

## 6 | 병렬처리

최신 이론에 따르면 동조는 협력이 아니라 경쟁의 산물이다. 반딧불은 남보다 먼저 발광하려고 한다(암컷이 이를 좋아하는 듯하다). 모두가 이런 전략을 따를다면 자동으로 서로 동조하는 결과가 나타난다.

- 스티븐 스트로가츠(『동시성의 과학 싱크』 중에서)

### 6.1 stop/suspend와 wait/notify 메서드

앞에서 성능을 높이기 위해 대기 상태의 스레드를 미리 여러 개 만들어 놓고 요청이 들어오면 활성화해 요청을 처리한 후, 다시 대기 상태로 관리한다고 했습니다. 이런 목적에 부합되는 자바 스레드 API에는 stop/suspend 메서드가 있지만 모두 deprecated<sup>01</sup> 되었습니다. 이유는 다음과 같습니다.

stop 메서드가 호출되면 해당 스레드가 즉시 멈춥니다. 그와 더불어 ThreadDeath 예외가 발생하면서 해당 스레드가 배타적으로 소유하던 모든 객체에 대한 모니터(혹은 뮤텍스<sup>02</sup>)가 즉시 해제되어 다른 스레드가 접근할 수 있게 됩니다. 문제는 소유하던 객체의 상태가 불안정한 상태일 수도 있다는 것입니다. 스레드가 특정 객체에 모니터 잠금을 사용하는 이유는 다른 스레드가 접근하지 못하게 배타적인 권한을 획득한 후 객체의 내부 상태를 변경하기 위해서입니다. 그런데 스레드가 객체의 상태를 변경하던 중 stop 메서드가 호출되면 외부 스레드에 모순된 상태의 객체, 즉 상태 변경이 완료되기 전의 형상<sup>03</sup>에 접근할 수 있어 원인을 매우 찾기 어려운 버그가 됩니다.

01 차후 버전에서 더 이상 지원하지 않을 수 있으므로 비슷한 기능의 신규 API로 변경이 필요하다는 표식입니다.

02 상호 배제 잠금(mutual exclusion lock)

03 예를 들어, 사용자의 성명을 '홍길동'에서 '성춘향'으로 변경하는 경우를 생각해보자. 성을 '홍'에서 '성'으로 바꾼 후, 이름을 저장하기 전에 외부에서 접근하면 '성길동'이라는 존재하지 않는 이름을 가진 객체로 보일 수 있습니다.

두 번째 메서드인 suspend는 문제가 좀 더 분명합니다. 스레드가 특정 리소스에 대한 뮤텍스를 획득한 상태에서 suspend 메서드가 호출돼 정지되면, 리소스에 접근하려는 스레드는 suspend된 스레드가 스스로 동작을 재개해 뮤텍스를 풀기 전 까지 기약 없이 대기해야 하는 문제(문이 잠겨 있는데 열쇠를 가진 사람이 안에 들어가 있는 형국)가 발생합니다. 이런 시나리오는 데드락이 발생할 수 있는 대표적인 예입니다.

위와 같은 상황을 피하고 스레드를 대기시키거나 진행하려면 wait/notify 메서드 조합을 이용합니다.

톰캣 4.x에서 제공하던 스레드 풀은 각 스레드에 lock 객체를 생성하고 해당 객체에 대해 wait 메서드를 호출함으로써 스레드를 대기 상태로 유지했습니다. 이런 대기 상태에 있는 스레드를 리스트 형태로 유지하다가 요청이 들어오면 스레드를 대기 리스트에서 빼내 서블릿 실행에 필요한 정보를 전달합니다. 그다음 해당 스레드를 대기 상태에서 활성화 상태로 바꾸기 위해 notify 메서드를 호출합니다. 이때 스레드는 정지됐던 wait 메서드 이후로 진행을 재개합니다. 서블릿 동작이 완료되면 스레드의 wait 메서드가 다시 호출되어, 스레드는 대기 상태로 돌아가며 다음 요청에 재사용할 수 있게 스레드는 대기 상태의 스레드 리스트로 재진입합니다. 이렇게 스레드마다 내부 lock을 사용해 개별적으로 대기를 구현하는 방법은 가장 직관적인 방식으로 스레드 풀링을 구현했다고 할 수 있습니다.

스레드를 먼저 다수 생성한 후 단일 Concurrent Queue를 사용해 각 스레드가 take 메서드를 호출해 스레드를 간접적으로 대기하는 좀 더 단순한 방법도 있습니다. 이 방법은 대기 리스트에 스레드를 유지, 관리할 필요가 없으며 단순히 요청을 Concurrent queue에 넣으면 queue.take 메서드에서 대기하던 스레드 중 하나가 동작합니다. 비교적 단순한 구조라는 것이 가장 큰 장점입니다.

## 6.2 스레드 풀의 구성 요소 - jetty 6.x의 경우

“바퀴를 다시 발명할 필요는 없다”라는 격언은 너무나 당연한 이야기겠지만, 도전적인 소프트웨어 엔지니어에게 그렇게 큰 영향력이 있는 것 같지는 않습니다. 특히 자바는 초기부터 스레드를 지원한 언어입니다. 자바의 장점으로 언어 레벨에서 스레드를 지원한다는 점이 강조되다 보니 많은 프로그래머가 스레드에 관심을 두게 되었고, 이런 이유로 지금까지 무수히 많은 스레드 풀 구현체가 만들어져 사용됩니다. 스레드 풀을 구현하려면 어떤 요소가 제공돼야 할까요?

쉽게 생각해 보면 `java.lang.Thread`를 여럿 생성해 컬렉션 객체에 저장시켜 놓은 뒤, 요청이 들어왔을 때 컬렉션 객체에서 스레드를 하나 꺼내 요청을 처리하면 됩니다. 아마도 다음과 같은 모습이 될 것입니다.

---

```
// 10개의 스레드를 생성해 threadList라는 이름의 List에 저장한다.
List<Thread> threadList = new ArrayList<Thread>();
for(int i=0; i<10; i++){
    Thread t = new Thread();
    t.start();
    threadList.add(t);
}
```

---

이렇게 단순히 `Thread` 객체 자체를 생성한 후 저장한다면 세 가지 문제가 발생합니다. 첫 번째는 스레드 풀로 관리하는 스레드는 한 번 수행되고 버려지는 것이 아니라 반복적으로 재사용 가능해야 합니다. 따라서 스레드 풀 내에서 생성/관리되는 스레드는 한 번 수행됐다고 종료돼서는 안 됩니다. 그러므로 다음과 같이 일종의 무한 루프를 돌게 구현된 특수한 `Thread` 객체가 스레드 풀링의 요소가 되어야 합니다. 다음 예는 jetty 6.x의 `QueuedThreadPool`에서 가져왔습니다. `PoolThread`는 해당 클래스의 내부 클래스로 정의돼 있습니다.

---

```
public class PoolThread extends Thread {  
    Runnable _job=null;  
    // 생략.....  
    public void run(){  
        // 생략.....  
        while (isRunning()){  
            // Run any job that we have.  
            if (job!=null){  
                final Runnable todo=job;  
                job=null;  
                idle=false;  
                todo.run();  
            }  
        }  
        // 하락.....
```

---

run 메서드 내에 while 루프가 존재하여 isRunning 메서드의 반환 값이 참이면 루프가 계속 도는 것을 확인할 수 있습니다.

여기에서 두 번째 문제가 발생합니다. 위 코드에 따르면 job이 null일 때, 즉 처리 할 요청이 당장 없으면 while 루프를 계속해서 순회합니다. 다시 말해, busy wait 를 합니다. 이를 막으려면 다음과 같이 요청을 기다리면서 스레드 실행을 대기해야 합니다(다음 코드는 PoolThread의 while 루프 내에서 가져왔습니다).

---

```
// 생략.....  
    While(isRunning()){  
        // 중략.....  
        synchronized (this){  
            if (_job==null)  
                this.wait(getMaxIdleTimeMs());
```

```
        job=_job;
        _job=null;
    }
}

// 하락.....
```

---

현재 자신이 처리할 job이 없는 경우 wait 메서드를 사용해 대기 상태로 들어가는 것을 확인할 수 있습니다. 물론 wait 메서드의 매개변수로 전달된 getMaxIdleTimeMs()의 반환 값만큼 시간이 흐르면 다시 while 루프로 job의 존재 여부를 체크합니다. 하지만 새로운 요청이 들어왔을 때, 즉시 PoolThread를 할당할 수 있게 다음과 같이 dispatch 메서드를 추가합니다.

---

```
void dispatch(Runnable job){
    synchronized (this){
        _job=job;
        this.notify();
    }
}
```

---

마지막 세 번째 문제는 병렬로 실행하려는 특정한 처리를 Thread 객체에 어떻게 전달하느냐는 것입니다. 다시 말해, HTTP를 처리하는 클래스 X를 병렬로 수행하고자 할 때 단순히 스레드 풀이 Thread 클래스의 인스턴스를 가지고 해당 클래스를 어떻게 동작시킬 것인가 하는 문제입니다.

객체지향 언어에서 발생하는 대부분의 문제가 그렇듯 이것 역시 인터페이스를 사용해 해결합니다. 스레드 풀의 작성자와 그 스레드 풀을 사용하는 사용자는 특정 인터페이스를 약속한 후 사용자는 병렬로 수행하고자 하는 역할을 해당 인터페이스를 구현한 클래스로 작성합니다. 그러면 작성자는 그 인터페이스 형식의 인자를

받아 병렬로 인터페이스의 메서드를 수행합니다.

이러한 실행 인터페이스는 스레드 풀 구현이 정하기 나름입니다. 그 예로 jetty 6.x 의 구현은 독자적인 인터페이스를 추가하기보다 Runnable 인터페이스를 재활용 하는 방식을 취합니다. 다시 말해, Runnable 인터페이스의 run 메서드를 원래 의미인 start 메서드 호출 시 병렬로 수행되는 메서드의 의미 외에 단순히 메서드로 간주하여 직접 호출해 사용했습니다.

앞의 PoolThread를 보면 Runnable인 \_job이 실행 스레드의 멤버 변수로 존재합니다. PoolThread 클래스의 dispatch(Runnable) 메서드를 확인하면, 처리할 요청은 Runnable 인터페이스를 구현한 인자로 실행 스레드로 넘겨지며 실행 스레드의 run 안에서 인자로 전달된 Runnable 구현체의 run 메서드를 호출하여 병렬처리합니다.

이제 요청이 들어올 때까지 대기하고(자체 락을 사용해 wait), 요청이 처리된 이후에도 다음 요청을 위해 루프를 돌며(isRunning())이 참을 반환하는 한 무한 루프), 대기 상태에서 새로운 요청에 대해 즉각적으로 대응하기 위한 처리(dispatch 메서드 내에서 해당 wait 조건을 탈출할 notify() 호출)가 추가된 PoolThread 구현을 살펴보았습니다.

PoolThread를 사용해 스레드 풀을 다시 구성해 보겠습니다. 다시 QueuedThreadPool 의 구현 내용을 살펴보면 다음과 같습니다.

---

```
protected void doStart() throws Exception{
    if (_maxThreads<_minThreads || _minThreads<=0)
        throw new IllegalArgumentException("!0<minThreads<maxThreads");
    _threads=new HashSet();
    _idle=new ArrayList();
```

```

.jobs=new Runnable[_maxThreads];
for (int i=0;i<_minThreads;i++){
    newThread();
}
}

protected void newThread(){
    synchronized (_threadsLock){
        if (_threads.size()<_maxThreads){
            PoolThread thread =new PoolThread();
            _threads.add(thread);
            thread.setName(thread.hashCode()+"@"+_name+"-"+_id++);
            thread.start();
        }else if (!_warned){
            _warned=true;
            Log.debug("Max threads for {}",this);
        }
    }
}

```

---

먼저 QueuedThreadPool이 스레드 풀로써 동작하면 doStart 메서드가 호출됩니다. 이 메서드는 설정 값에 따라 newThread 메서드를 호출합니다.

newThread 메서드는 앞에서 살펴본 PoolThread를 생성한 후 \_threads라는 이름의 HashSet에 저장하는 것을 확인할 수 있습니다. 이 PoolThread는 앞에서 살펴보았듯 요청이 없는 경우 자동으로 wait 상태에서 대기합니다. 요청이 새로 들어오면 PoolThread의 dispatch 메서드를 호출해 처리를 재개합니다.

다음은 스레드 풀에서 요청이 들어왔을 때 호출되는 QueuedThreadPool 클래스의 dispatch 메서드입니다.

---

```
public boolean dispatch(Runnable job) {
    if (!isRunning() || job==null)
        return false;
    PoolThread thread=null;
    boolean spawn=false;
    synchronized(_lock) {
        // Look for an idle thread
        int idle=_idle.size();
        if (idle>0)
            thread=(PoolThread)_idle.remove(idle-1);
        else {
            // 생략.....
            // idle 한 스레드가 모자라므로 추가합니다.
        }
        spawn=_queued>_spawnOrShrinkAt;
    }
    if (thread!=null) {
        thread.dispatch(job);
    }else if (spawn){
        newThread();
    }
    return true;
}
```

---

복잡하게 보이지만, 결국 현재 가용한 PoolThread 자원이 남아 있으면 그 스레드의 dispatch 메서드를 호출하고, 가용하지 않으면 newThread 메서드를 사용해 가용한 PoolThread를 만들어낸다는 내용입니다.

이외에도 스레드 풀에는 단순히 스레드 수를 일정하게 유지하는 기능뿐만 아니라 최대 가능 스레드 수, 최소 스레드 수, 저부하 상태에서 유지할 적절한 스레드 수와

같이 상태를 관리하는 기능이 추가돼야 합니다. 또한, 지나치게 오랜 시간 동안 동작하는 스레드<sup>04</sup>의 동작 시간을 체크하여 종료하는 기능도 필요합니다. 따라서 스레드 풀 내부에 관리를 위한 스레드를 두어 위에서 언급된 정리작업을 따로 해야 합니다.

### 6.3 java.util.concurrent 패키지

지금까지 스레드 풀을 구현하기 데 필요한 문제에 대해 알아보았습니다. 하지만 스레드 풀 구현의 가장 큰 문제는 병렬처리 자체가 그렇게 호락호락 한 분야가 아니라는 것입니다. 일단 스레드를 적절히 생성하고 수행하는 생명주기를 관리해야 할 뿐만 아니라, 병렬 진행되는 스레드 간에 동시성 문제가 발생하지 않게 조절하는 것은 쉽지 않습니다. 또한, 스레드와 같은 병렬 진행을 다루는 프로그래밍은 순차 진행 프로그래밍보다 디버깅 난이도가 급격하게 증가하므로 완성도 높은 구현체를 만드는 것은 아주 어려운 작업이라 할 수 있습니다.

이런 어려움을 해결하기 위해서 Java 1.5에는 `java.util.concurrent` 패키지가 추가되었습니다. 앞서 자바 언어가 스레드 지원을 시작한 초기 언어에 속한다고 언급했습니다. 하지만 자바의 초기 스레드 지원은 사용하기 쉽게 하려는 목적이 초점이 맞춰져 단순성에 중점을 두어 세세한 조정이 필요한 고급 스레드 프로그래밍에는 아쉬운 점이 있었습니다. 점점 자바로 구현하는 스레드 기반 프로그램이 늘어나면서 이에 대한 요구가 나오기 시작해 `java.util.concurrent` 패키지에 그 기능이 추가되었습니다.<sup>05</sup>

`java.util.concurrent` 패키지에는 `ThreadPoolExecutor`라는 이름의 범용

---

04 일반적으로 스레드가 종료되지 않고 예상되는 시간보다 길게 수행되는 경우 stuck 상태에 빠졌다고 말합니다. 대부분은 교착 상태와 같은 deadlock 상황이거나, DB나 소켓과 같은 외부 자원에 대한 접근에 대한 타임아웃이 설정되지 않았을 때 발생합니다.

05 스레드 관련 기능이 제공되기 전에는 대부분의 스레드 프로그래밍에서 Latch, Semaphore, Read Write Lock 같은 기능을 직접 구현하거나 몇몇 잘 알려진 사용자 작성 concurrent 라이브러리를 사용했습니다.

스레드 풀 구현체도 추가되었습니다. 해당 패키지 이전에는 모든 서블릿 컨테이너가 독자적인 스레드 풀을 구현해 사용했지만, 현재 대부분의 구현체는 ThreadPoolExecutor 구현체로 스레드 풀 구현을 교체하거나 최소한 ThreadPoolExecutor와 인터페이스를 맞춘 구현체를 제공합니다. 이는 ThreadPoolExecutor 스레드 풀 구현체 자체가 기존의 스레드 풀에 비해 월등한 성능을 가졌다고 확인돼서 그렇다기보다는 언어 자체에서 제공하는 표준 구현체를 사용하는 것이 서블릿 컨테이너를 구현하는 사람들 사이에서도 추가 커뮤니케이션 비용을 줄일 수 있기 때문이라 할 수 있습니다.

## 6.4 ThreadPoolExecutor의 사용

이번에는 ThreadPoolExecutor를 사용해 병렬로 수행하는 프로그램을 작성해 보겠습니다. 다음은 이상 작가의 시인 「오감도」의 첫 부분을 표시하는 프로그램입니다.

---

```
public class WalkSerial{
    public static void main(String[] args){
        WalkSerial ws = new WalkSerial();
        System.out.println("오감도 - 이상");
        ws.action(13);
    }
    private void action(int size){
        System.out.printf("%d인의 아해가 도로를 질주하오.\n", size);
        System.out.printf("(길은
막다른골목이적당하오.)\n\n");
        for(int i=0; i<size; i++){
            printMessage(i+1);
        }
    }
}
```

```

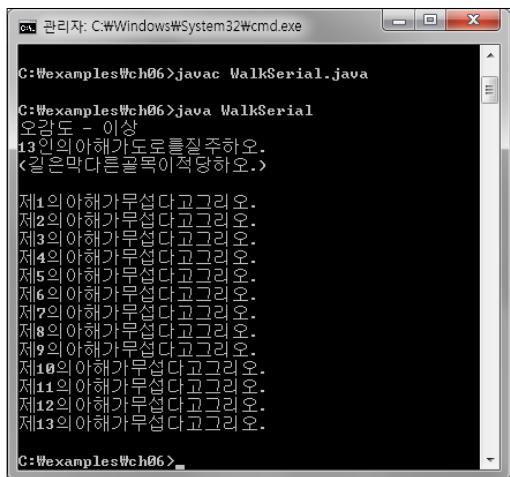
private Object lock = new Object();
private void printMessage(int index){
    synchronized(lock){
        try{
            lock.wait(1000L);
        }catch(InterruptedException e){
        }
        System.out.printf("제%d의 아해가 무섭다고 그리오.\n", index);
    }
}

```

---

printMessage 메서드 안에서 1초씩 기다리기 때문에 마지막까지 실행하려면 최소한 13초가 필요합니다. 실행 결과는 다음과 같습니다.

그림 6-1 WalkSerial의 실행 결과



```

cmd 관리자: C:\Windows\System32\cmd.exe
C:\examples\ch06>javac WalkSerial.java
C:\examples\ch06>java WalkSerial
오길도 - 이상
13인의아해가도로를질주하오.
<길은막다른골목이적당하오.>
제1의 아해가 무섭다고 그리오.
제2의 아해가 무섭다고 그리오.
제3의 아해가 무섭다고 그리오.
제4의 아해가 무섭다고 그리오.
제5의 아해가 무섭다고 그리오.
제6의 아해가 무섭다고 그리오.
제7의 아해가 무섭다고 그리오.
제8의 아해가 무섭다고 그리오.
제9의 아해가 무섭다고 그리오.
제10의 아해가 무섭다고 그리오.
제11의 아해가 무섭다고 그리오.
제12의 아해가 무섭다고 그리오.
제13의 아해가 무섭다고 그리오.
C:\examples\ch06>

```

i+1번째 아래는 i번째 아래가 무서운 후에야 무서울 수 있다는, 즉 1번부터 13번째 까지 차례차례 출력됨을 확인할 수 있습니다. 다음 내용은 ThreadPoolExecutor 을 이용해 병렬로 진행되게 구현한 것입니다.

---

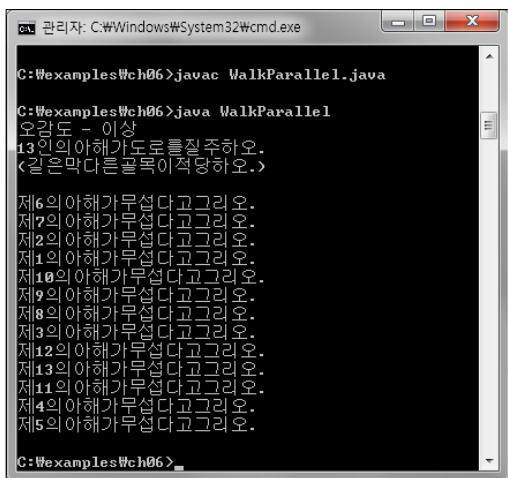
```
import java.util.concurrent.*;
public class WalkParallel{
    public static void main(String[] args){
        WalkParallel wp = new WalkParallel();
        System.out.println("오감도 - 이상");
        wp.action(13);
    }
    private void action(int size){
        System.out.printf("%d인의 아래가 도로를 질주하오.\n", size);
        System.out.printf("(길은 막다른 골목이 적당하오.)\n\n");
        ThreadPoolExecutor tpe = new ThreadPoolExecutor(10, 20, 60,
                TimeUnit.SECONDS, new SynchronousQueue<Runnable>());
        for(int i=0; i<size; i++){
            tpe.execute(new People(i+1));
        }
        tpe.shutdown();
    }
    class People implements Runnable{
        private int index;
        private Object lock = new Object();
        public People(int index){
            this.index = index;
        }
        public void run(){
            synchronized(lock){
                try{
                    lock.wait(1000L);

```

```
        }catch(InterruptedException e){
    }
    System.out.printf("제%d의 아해가 무섭다고 그리오.\n", index);
}
}
}
}
```

WalkParallel을 실행하면 1초 정도 기다렸다가 13명의 아해가 동시에 다발적으로 서로 무섭다고 말하는 것을 확인할 수 있습니다. 따라서 전체 수행은 2초 내로 종료되며, 순서에 상관없이 무작위로 나타납니다.

그림 6-2 WalkParallel 실행 결과



```
관리자: C:\Windows\System32\cmd.exe
C:\examples\ch06>javac WalkParallel.java
C:\examples\ch06>java WalkParallel
오강도 - 이상
13인의아해가도로를질주하오.
<길은막다른골목이적당하오.>
제6의아해가무섭다고그리오.
제7의아해가무섭다고그리오.
제2의아해가무섭다고그리오.
제4의아해가무섭다고그리오.
제10의아해가무섭다고그리오.
제9의아해가무섭다고그리오.
제8의아해가무섭다고그리오.
제3의아해가무섭다고그리오.
제12의아해가무섭다고그리오.
제13의아해가무섭다고그리오.
제11의아해가무섭다고그리오.
제4의아해가무섭다고그리오.
제5의아해가무섭다고그리오.
C:\examples\ch06>
```

또한, 스레드 풀은 내부에 요청이 들어왔을 때 처리하기 위해 대기 상태의 스레드를 계속 유지하므로 현재 작업 큐가 비었다고 자동으로 종료되지 않습니다. 위 예제에서는 shutdown 메서드를 호출해 스레드 풀 자체를 종료시켰습니다. 서블릿 컨테이너는 ThreadPoolExecutor의 라이프사이클을 적절히 관리해주어

야 합니다. 예제에서 볼 수 있듯이 ThreadPoolExecutor의 사용은 비교적 직관적으로 이해할 수 있습니다. 병렬로 수행할 작업을 Runnable 타입으로 생성해 ThreadPoolExecutor의 executor 메서드 인자로 넘겨주기만 하면 됩니다.

여기서 주의해야 할 점은 작업 큐로 사용할 BlockingQueue의 성질에 따라 미묘하게 3가지 다른 수행 방식이 있다는 것입니다. 예제에서 사용한 SynchronousQueue는 수행 요청이 들어오면, 요청을 쌓아두지 않고 풀 내에 준비된 스레드를 즉시 매치하여 처리합니다. 따라서 스레드 풀 내에 준비된 스레드가 모두 다른 일을 수행한다면, 최대 가능 스레드 수가 초과되지 않는 한 즉시 새 스레드를 생성해 요청을 처리합니다. 최대 가능 스레드 이상의 요청이 들어오면 해당 요청은 버려지므로 최대 가능 스레드 수는 제한이 없게 크게 설정하는 것이 일반적입니다. 따라서 처리되는 속도보다 요청이 들어오는 속도가 더 빠르면 스레드의 수가 비정상적으로 증가할 수 있습니다.

나머지 두 방식은 최대 가능 스레드 수에 제한 있는 큐를 사용하는 방식과 제한 없는 큐를 사용하는 방식입니다. 이 두 방식에는 실질적으로 최대 가능 스레드 수는 의미가 없습니다. 왜냐하면, ThreadPoolExecutor 생성자의 첫 번째 인자인 corePoolSize 수만큼만 스레드를 사용해 동작하며, 이상의 요청이 들어오면 스레드의 수를 늘리는 대신 추가 요청을 작업 큐에 쌓아 놓고 처리합니다. 앞의 경우와 같이 처리되는 속도보다 요청이 들어오는 속도가 더 빠를 경우 제한 없는 큐(LinkedBlockingQueue)는 큐의 크기가 비정상적으로 커지는 문제가 발생할 수 있습니다. 반면, 제한 있는 큐(ArrayBlockingQueue)는 미리 정해진 큐 사이즈를 넘는 요청을 모두 버립니다. 따라서 이 세 타입은 모두 서로 기능 및 안정성에 대한 트레이드오프<sup>trade-off</sup>가 있으며, 어떤 타입이 언제나 옳은 것이 아니라 사용되는 상황에 맞게 적절히 선택해야 합니다.

## 6.5 적정 병렬 진행 수

대부분의 서블릿 컨테이너에는 서블릿을 동작시키는 스레드의 숫자를 설정할 수 있습니다. 다시 말해, 서블릿을 처리하기 위해 스레드 풀이 내부적으로 유지하는 워커 스레드의 숫자를 지정할 수 있으며 이 값을 어떻게 설정하느냐에 따라 서블릿 컨테이너의 전체 처리량이 크게 영향을 받게 됩니다. 해당 값에 대한 설정치를 구하기 위해 고려해야 할 사항으로는 어떤 것이 있을지 알아보겠습니다.

컴퓨터의 연산 처리는 CPU가 수행합니다. 즉 CPU가 가진 코어<sup>core</sup>의 개수만큼 계산 작업이 동시에 처리할 수 있습니다. 이 말은 서블릿 컨테이너가 CPU 코어 수 미만의 스레드를 사용한다면 한 개 이상의 CPU 코어는 서블릿 컨테이너가 사용하지 못한다는 의미입니다. 따라서 최대 스레드 수 설정은 CPU 코어 수보다 작지 않아야 합니다.

또한, 병렬 진행되어야 하는 스레드의 수가 CPU 코어 수보다 큰 경우, 모든 스레드가 동시에 CPU 코어를 사용할 수는 없습니다.<sup>06</sup> 다시 말해, 다수의 스레드가 소수의 CPU 코어를 이용하려면 특정 시간에는 반드시 하나 이상의 스레드가 CPU 코어를 점유하지 않고 대기 상태에 있어야 합니다. 그러므로 처리하는 일이 CPU 연산 능력에만 의존하는 경우 CPU 코어마다 하나의 스레드가 할당되게 설정하는 것이 최적입니다. 이렇게 설정하면 일을 처리하는 모든 스레드는 대기 상태로 빠지지 않고 작업을 계속할 수 있습니다.

그렇다면 결국 CPU 코어의 수만 세면 최적의 스레드 수를 찾은 것일까요? 그렇지 않습니다. 앞선 내용 중 ‘CPU 연산 능력에만 의존하는 경우’라는 조건에 유의해야 합니다. 현실적으로 CPU 연산에만 의존하는 병렬처리 작업은 찾기가 어렵습니다. 좀 더 정확히는 CPU 연산에 주로 의존하는 작업을 병렬처리하기 위해 재구현하면

---

06 이해를 돋기 위해 CPU를 스레드가 접근하는 자원처럼 서술했습니다. 엄밀히 말하면 CPU가 스레드를 동작시키는 것이므로 특정 순간에 CPU 코어 개수 이상의 스레드는 활성화, 즉 동작할 수 없다는 의미가 됩니다.

원래의 제약조건인 CPU 연산 외의 병렬처리 자체에 내재된 다른 제약조건이 발생합니다.

다음은 10만 팩토리얼을 구하는 코드입니다. 10만 팩토리얼은 순수한 계산 문제로 CPU 연산에만 의존하는 작업입니다. 동작시킨 CPU 코어 수와 일치하는 스레드를 사용할 때 가장 빠른 결과를 나타낼 것으로 기대할 수 있습니다.

---

```
import java.math.*;

public class Factorial{
    public static void main(String[] args){
        Factorial f = new Factorial();
        // 인자값이 없는 경우 4개의 스레드를 사용한다.
        int threadSize = 4;
        if(args != null && args.length > 0){
            // 첫번째 인자 만큼 스레드를 만든다.
            threadSize = Integer.parseInt(args[0]);
        }
        f.action(threadSize);
    }

    private void action(int threadSize){
        System.out.println("use thread : "+threadSize);
        long start = System.currentTimeMillis();
        // 부분 곱에 대한 전역 저장소를 만든다.
        // 각각의 스레드 마다 접근하는 배열의 인자가 서로 다르므로
        // 서로간의 간섭, 즉 동시성 문제는 발생하지 않는다.
        BigInteger[] resultArray = new BigInteger[threadSize];
        SubFactorial[] subFactorialArray = new SubFactorial[threadSize];
        // 스레드의 수로 10만을 나누어 각각의 스레드가 처리해야 할 단위를 결정한다.
        int delta = 100000 / threadSize;
```

```

System.out.println("delta : " +delta);
for(int i=0; i<threadSize; i++){
    // 각각의 스레드에 부분 곱을 구해야 할 범위와 그 결과를 저장할 배열을 넘겨준다.
    subFactorialArray[i] = new SubFactorial(""+(i*delta + 1),
        ""+(i+1)*delta), resultArray, i);
}
for(SubFactorial subFactorial : subFactorialArray){
    // 스레드를 동작시킨다.
    subFactorial.start();
}
try{
    for(SubFactorial subFactorial : subFactorialArray){
        // 메인 스레드를 각각의 부분 곱을 구하는 스레드가 종료되기까지
        // join 메서드를 사용해 대기시킨다.
        subFactorial.join();
    }
}catch(InterruptedException e){
    e.printStackTrace();
}
// 여기까지 왔다는 것은 모든 부분 곱을 구한 SubFactorial 스레드가 종료했다는
// 뜻이고, 동시에 resultArray 배열 안에 부분 곱들이 계산되었다는 의미다.
BigInteger factorial = new BigInteger("1");
for(BigInteger subFactorial : resultArray){
    // 부분 곱에 대해 전체 곱을 구한다.
    // 단위가 크므로 java.math 패키지에 정의된 BigInteger를 사용한다.
    factorial = factorial.multiply(subFactorial);
}
long end = System.currentTimeMillis();
System.out.println("bit length : "+factorial.bitLength());
System.out.println("total time (ms) : "+ (end - start));
}

```

```

// java.lang.Thread 클래스를 상속받아 부분 곱을 구하는 클래스다.
class SubFactorial extends Thread{
    private String from;
    private String to;
    private BigInteger[] resultArray;
    private int resultIndex;
    private SubFactorial(
        String from,
        String to,
        BigInteger[] resultArray,
        int resultIndex){
        // 부분 곱을 구하는 시작값과 종료값, 부분 곱의 결과를 지정할
        // 배열과 위치를 생성자의 인자로 전달받는다.
        this.from = from;
        this.to = to;
        this.resultArray = resultArray;
        this.resultIndex = resultIndex;
    }
    // 클래스의 start 메서드가 호출되면 병렬적으로 아래의 run 메서드가 호출된다.
    @Override
    public void run(){
        BigInteger factorial = new BigInteger("1");
        BigInteger n = new BigInteger(to);
        BigInteger k = new BigInteger(from);
        BigInteger one = new BigInteger("1");
        // 초기값에서부터 시작하여 값을 하나씩 증가시켜가면서 부분 곱을 구한다.
        // 단 경계값에 도달하면 멈춘다.
        while(k.compareTo(n)<= 0){
            factorial = factorial.multiply(k);
            k = k.add(one);
        }
        // 계산된 부분 곱 값을 지정받은 위치에 저장한다.
    }
}

```

```

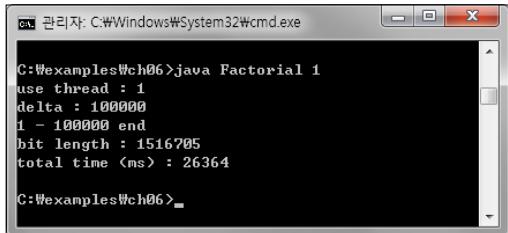
        resultArray[resultIndex] = factorial;
        System.out.println(from+" - "+to+" end");
    }
}
}

```

---

컴파일한 후 실행하면 다음과 같은 결과를 볼 수 있습니다.

그림 6-3 Factorial의 결과 화면



```

관리자: C:\Windows\System32\cmd.exe
C:\examples\ch06>java Factorial 1
use thread : 1
delta : 100000
1 - 100000 end
bit length : 1516705
total time <ms> : 26364

C:\examples\ch06>_

```

숫자를 인자로 받아 해당 숫자만큼의 스레드를 사용해 부분 곱을 구하고 모든 스레드가 계산을 마치면 부분 곱들의 곱을 사용해 답을 반환합니다. 단, 수치가 매우 큰 관계로 결과 비트의 자릿수를 반환합니다. 다음 표는 CPU 코어가 네 개인 시스템에서 나온 결과입니다.

표 6-1 CPU 코어가 네 개인 시스템의 10만 팩토리얼 연산 시간

스레드 수	걸린 시간(ms)
1	46433
2	16903
4	7924
8	6746
16	6908

스레드의 수가 8개일 때 가장 빨리 연산했습니다. 스레드가 4개일 때가 16개일 때 보다 더 좋은 결과가 나온 것에 유의하십시오. 예상과 다른 결과가 나온 원인을 알아보겠습니다.

### 6.5.1 병렬화 가능하지 않은 영역의 존재

34라인 이후부터는 단일 스레드인 메인 스레드에서 개별 스레드에서 계산한 부분 곱들에 대한 곱을 계산하여 답을 구합니다. 이렇게 병렬화되지 않는 구간이 존재할 수 있으며, 이런 구간이 전체 프로세스 중에 나타나는 위치와 빈도에 따라 전체 병렬처리 효율은 크게 영향을 받게 됩니다. CPU 코어 수와 스레드 수를 일치시킬 때 최대 처리량을 얻을 것이라는 추측은 모든 CPU 코어를 최대한 동작시킬 방법이 있다는 가정하에서 설득력이 있으며, 이런 병렬화가 가능하지 않은 영역이 존재하는 경우 그 정당성을 상실합니다.

### 6.5.2 병렬처리로 인해 부가되는 추가 제약조건

앞서 병렬처리를 하면 선형 처리에서는 존재하지 않던 제약조건이 추가된다고 했습니다. 생각해 보면 선형적으로 팩토리얼을 구하기 위해서는 단 2개의 저장소에 대한 접근, 즉 1씩 증가하는 현재 상태값과 지금까지의 상태값들을 곱해 온 중간 곱 결과값을 저장할 저장소와 저장소에 대한 접근이 필요합니다. 하지만 병렬처리를 위해서 Factorial 클래스는 개별 스레드 별로 부분 곱을 계산하기 위한 저장 장소를 따로 준비하여 접근하고 전체 곱을 구하기 위해 다시 한 번 부분 곱의 배열에 접근하는 것을 볼 수 있습니다. 다시 말해, 단순히 숫자들의 곱을 빨리 구하는 연산 능력이 유일한 성능 요소라고 가정할 수 없게 되었습니다.

지금까지 살펴본 팩토리얼 연산은 상당히 단순하고 계산 위주의 프로그램임에도 처리량을 최대로 만드는 스레드 수를 예상하기 쉽지 않았습니다. 우리의 주된 관심사인 서블릿 컨테이너가 다루는 서블릿은 팩토리얼 연산보다는 훨씬 복잡한 양상을 보입니다. 이제 서블릿의 어떤 특징이 서블릿 컨테이너의 동시 스레드 수 결정

을 어렵게 만드는지 알아보겠습니다.

이 장을 시작하면서 왜 서블릿 컨테이너에서 병렬처리가 중요한지에 대해 언급한 바 있습니다. 언급한 내용 중 하나가 서블릿 컨테이너의 최대 처리량에 대한 제약 조건이 CPU 처리량이 아니라 I/O 처리량<sup>07</sup>이라는 사실을 기억하십시오. 그렇다면 가능한 동시에 많은 스레드를 사용해 I/O 처리량을 극대화하는 전략이 더 나은 선택이 될 수 있겠습니다. 하지만 무작정 스레드의 수가 많을수록 처리량이 늘어나지는 않습니다. 스레드도 생성하고 유지하는 관리에 비용이 드는 자원이기 때문입니다. 따라서 실질적으로 생성, 유지 가능한 스레드의 수는 제한됩니다.

### 6.5.3 컨텍스트 스위칭

물리적인 제약 외에도 스레드의 수가 많아지면 그 자체가 성능 저하의 원인이 됩니다. 앞서 CPU 코어 수 이상의 스레드는 동시에 활성화되지 못하고 대기 상태에 머물러 있다고 언급했습니다. 멀티태스킹을 지원하는 대부분의 현대적인 운영체제는 CPU 코어 수보다 훨씬 많은 프로세스, 스레드들을 마치 동시에 수행되는 것처럼 보이게 합니다. 이를 위해 시분할<sup>time slicing</sup> 기법<sup>08</sup>을 사용하여 하나의 프로세스와 스레드가 일정 시간 CPU를 사용한 후에는 다른 프로세스나 스레드에 사용을 양보한 후 대기하다가 자기 차례가 오면, 앞서 수행한 단계 다음부터 진행을 계속 할 수 있어야 합니다. 따라서 양보의 시점에 수행하던 일의 차례, 계산해 놓았던 중간 결과물 등의 일 처리 문맥을 저장해 놓았다가 다시 자신의 차례가 오면 저장했던 정보에서 처리 문맥을 되돌리는 컨텍스트 스위칭<sup>Context switching</sup>이 발생합니다.

스레드의 수가 많아지면 CPU 점유에 대한 경쟁 조건이 심화되므로 자연히 컨텍스트 스위칭이 그만큼 더 자주 발생합니다. 그런데 컨텍스트 스위칭은 병렬로 처리되는 것처럼 보이게 하는 데 소요되는 비용이지 처리량을 높이는 것과는 무관합니다.

---

07 CPU bounded job과 IO bounded job으로 통칭됩니다.

08 동시에 병렬적으로 진행된 것처럼 보이는 결과를 얻기 위해 시간을 짧은 단위로 나눈 후 각 단위 시간마다 여러 프로세스, 스레드를 교대로 수행시키는 기법이다.

따라서 컨텍스트 스위칭의 발생을 억제할수록 성능이 개선됩니다. 이런 이유로 스레드의 개수는 특정한 한계가 분명히 있습니다.

지금까지 서블릿 컨테이너가 유지해야 하는 스레드의 적정 숫자를 찾기 위해 여러 요소를 살펴보았습니다. 만약 앞서 언급된 모든 요소(병렬화 불가능 영역에 대한 고려, 병렬처리로 부가된 추가 제약조건에 대한 고려, 컨텍스트 스위칭을 최소화 등)를 모두 고려한다면 특정 서블릿 컨테이너의 권장 스레드 수를 찾을 수 있을까요?

안타깝지만 그 답변은 부정적일 수밖에 없습니다. 왜냐하면, 서블릿 컨테이너가 서비스하는 웹 사이트의 성능은 결국 서블릿 컨테이너 위에서 동작하는 서블릿이 어떤 성능을 가지는가에 가장 의존성이 높기 때문입니다. 따라서 앞선 질문은 다음과 같이 바꿔야 합니다.

서블릿 컨테이너와 그 위에 동작하는 서블릿의 조합으로 구성된 특정 사이트의 성능을 최대화하기 위해서 서블릿 컨테이너가 가져야 하는 스레드의 수는 얼마인가? 그리고 그 답변은 해당 사이트에서 사용하는 서블릿의 특성에 따라 좌우되므로 실제 상황에 맞게 설정 값을 적절히 변경해 가면서 최적의 값을 찾아내야 한다가 되겠습니다.

## 6.6 더 생각해 볼 문제

1. 이 장에서 나온 내용을 참고 해서 직접 스레드 풀을 구현해 보십시오. 5초 이상 시간이 지난 작업은 강제로 정지되는 기능을 추가해 보십시오. 작업에 사용된 리소스를 적절하게 반환할 기회를 주었는지 확인해 보십시오.
2. 읽기 쓰기 잠금<sup>ReadWriteLock</sup>에 대해 알아보십시오. java.util.concurrent 패키지에서 제공되는 구현체를 참고 해서 직접 구현해 보십시오.

3. 그리드 컴퓨팅 Grid computing에 대해 알아보십시오.
4. 멀티스레딩 프로그램은 개발환경에서는 검출되지 않다가 실사용 환경에서만 발생하는 고약한 버그를 가진 경우가 있습니다. 실환경에서는 제거된 System.out.println의 사용과 연관 지어 유추해보십시오.

## 7 | BIO와 NIO의 비교

저마다 제가 훌륭하다고 말하지만, 누가 까마귀의 암수를 알겠는가?

(具曰予聖 誰知鳥之雌雄)

- 정민(『자웅난변』 중에서)

이 장에서는 블로킹 IO를 사용하는 방법과 네이티브 IO를 사용하는 방법을 비교하고 각각의 특징을 살펴보겠습니다.

### 7.1 일반적인 프론트엔드 웹 서비스 구성

앞서 살펴본 서블릿 컨테이너는 외부와 HTTP로 통신하는 서블릿을 관리하므로 웹 서비스 구성 요소 중 클라이언트와 가장 가까운 프론트엔드에 위치합니다. 하지만 실제로는 서블릿 컨테이너를 외부로 노출해 서비스하기보다는 Nginx, Apache, IIS와 같은 웹 서버를 앞에 세워 서블릿 컨테이너와 통신하는 구성이 좀 더 일반적입니다.<sup>01</sup> 이런 구조의 장점은 크게 보아 두 가지입니다.

첫 번째, 이미지 및 HTML 파일 같은 동적으로 생성되지 않는 static 파일은 전문적인 웹 서버가 좀 더 효율적으로 처리할 수 있습니다. 왜냐하면, 개별 OS에 특화된 시스템 콜<sup>02</sup>을 호출하는 데 제약이 없는 네이티브 바이너리로 제공되는 웹 서버에 비해 JVM이 추상화된 레벨로 한 번 더 감싼 I/O를 사용해야 하는 서블릿 컨테이너는 성능상 제약이 분명히 있기 때문입니다.

두 번째, 동시에 동작 가능한 스레드의 수가 전체 웹 서비스의 단위 시간당 처리량

---

01 Apache tomcat connector(구 mod\_jk) 모듈이 있어 특정 URL 패턴의 요청을 다른 포트로 전달할 수 있습니다. 이때 받은 HTTP 요청 자체를 그대로 리다이렉트하는 것이 아니라 고유의 프로토콜(Ajp13)을 사용해 좀 더 효율적으로 서블릿 컨테이너와 통신합니다. 좀 더 자세한 내용은 톰캣 커넥터 안내 페이지를 참조하십시오.  
<http://tomcat.apache.org/connectors-doc/>

02 sendfile과 같은 특정 OS에서만 지원 가능한 기능입니다.

throughput을 제한합니다. Java 1.4가 java.nio를 제공하기 전까지 모든 I/O는 블로킹 모드로만 동작했습니다. 따라서 클라이언트가 접근하면 서블릿 컨테이너는 해당 클라이언트와 연결된 소켓을 처리하는 스레드를 할당해야 했습니다. 일반적으로 한 대의 기계가 연결할 수 있는 소켓의 수는 유지할 수 있는 스레드의 수보다 훨씬 큽니다. 스레드가 프로세스보다 공유하는 메모리가 크다 할지라도 필요한 리소스의 크기를 무시할 수 없기 때문입니다.

또한, 네트워크의 I/O 성능은 CPU 연산 처리 능력에 비해서는 매우 낮으므로, 할당된 스레드는 네트워크 지연 network latency 때문에 대부분의 시간을 대기하는 데 보냅니다.

따라서 일반적인 웹 서비스 시스템은 서블릿 컨테이너의 앞단에 OS 레벨에서 지원되는 non-blocking I/O를 사용해 동시 처리 능력을 높인 웹 서버를 세우고, 서블릿 컨테이너에는 서블릿을 동작하게 하는 데 집중시켜 전체 시스템의 처리량을 최대한 높입니다.

## 7.2 직관적이고 개념적인 I/O

다음은 Blocking I/O를 사용해 소켓에서 데이터를 읽는 예제입니다. 소켓에서 InputStream을 꺼낸 다음 종료되기 전까지 한 바이트씩 읽어들입니다.

---

```
InputStream in = socket.getInputStream();
int oneInt = -1;
while(-1 != (oneInt = in.read())){
    System.out.print((char)oneInt);
}
in.close();
```

---

이러한 방식의 데이터 전달을 블로킹Blocking 모드라고 하는 이유는 `in.read` 메서드에서 진행이 중단될 수 있기 때문입니다. 예를 들어, 어떤 이유로<sup>03</sup> 데이터 전송이 중단됐을 때 `in.read` 메서드는 다음 데이터를 읽기 위해 계속 대기 상태가 되어 전체 수행이 중단됩니다.

얼핏 생각할 때, 송신측과 수신측은 소켓으로 서로 연결되어 있으므로 한 쪽의 문제를 다른 쪽에서 알아차릴 수 있을 것으로 생각할 수 있습니다. 이것은 연결(커넥션), 흐름(스트림) 등의 용어가 의도하지 않게 잘못된 유추를 이끌어낸 것입니다. 다시 말해, 수신측에서는 송신측에서 데이터를 전달하지 않으면 어떤 문제가 있는지 파악할 방법이 없으므로 데이터가 오기만을 기다릴 수밖에 없습니다. 실질적으로 네트워크상에서 데이터는 패킷 단위로 물리적인 신호의 뮤음으로 전송됩니다. 따라서 전송받는 측에서는 이런 물리적인 신호가 오지 않으면 상대방의 이상 여부를 파악할 아무런 방법이 없습니다.<sup>04</sup>

이렇게 네트워크상의 실제 전송과정과 무관한 연결, 흐름과 같은 용어를 사용하는 이유는 무엇일까요? 그것은 단순히 익숙한 개념이므로 이해하기 쉽기 때문입니다. 마치 전자의 이동을 설명하면서 전류(물처럼 흘러감)라는 개념을 도입하는 것과 대동소이합니다.

통나무를 강물에 내려보내면 하류에서 모을 수 있듯, 보내고 싶은 데이터를 차례로 스트림에 쓰면 받는 측은 스트림을 통해서 데이터 내용을 차례로 읽어들일 수 있다는 개념입니다. 따라서 Blocking IO는 직관적으로 데이터 통신을 이해하고 사용

---

03 네트워크상에서 발생할 수 있는 문제의 원인으로는 전송측의 고부하로 인한 지연, 프로그램 자체의 오류로 인한 데이터 손실, 전송 네트워크상의 오류, 누군가가 겉어찬 충격으로 랜선이 뽑히는 등의 물리적인 차단 등 수많은 경우가 있습니다.

04 소켓을 사용해 송/수신 프로그램을 작성한 후, 송/수신 중에 송신 프로세스를 Kill 등의 유ти리티를 사용해 제거하면 즉각 수신 프로그램에서 연결이 끊어진 것을 알 수 있다고 생각할 수도 있습니다. 그런데 이것은 송신 프로세스가 종료하면서 EOF 메시지를 전송하기 때문입니다. 송신측의 물리적 네트워크 연결을 단절시켜 보면, 즉 랜선을 뽑아보면 수신 프로세스가 연결이 끊어진 사실을 알지 못한다는 것을 가장 확실하게 확인할 수 있습니다.

할 수 있게, 송/수신되는 패킷들이 마치 전송한 순서대로 한 바이트씩 줄지어 송/수신하는 것으로 간주하고 사용할 수 있게 합니다.<sup>05</sup>

### 7.3 좀 더 실제 물리적 전송과 근접한 I/O 방법

Blocking I/O의 기능은 직관적으로 이해할 수 있습니다. 하지만 스트림, 즉 흐름을 기반으로 의미를 부여하다보니 실제 개별 메시지의 전달인 네트워크상의 전송 과정이 API 상으로는 숨겨졌습니다. 그래서 메시지가 전송되지 않는 시점의 제어가 쉽지 않게 되었습니다. 이 때문에 Blocking 상태에 빠지면, 이를 확인하거나 회피할 수 있는 수단이 없습니다.

Non-blocking I/O는 실재 네트워크 전송 상태에 더 근접한 사용법을 제공합니다. 스트림 대신 전달할 메시지를 담을 버퍼ByteBuffer를 생성하고 수신 지점으로 버퍼의 담긴 내용을 전달하면, 수신측에서는 네트워크상에서 벌어지는 이벤트에 반응할 수 있는 셀렉터Selector를 사용해 대기하다가 수신된 메시지를 받아 처리합니다.

다음은 소켓 연결을 받아들인 후 셀렉터를 이용해 데이터를 읽어들이는 예입니다.

---

```
selector = Selector.open();
ServerSocketChannel serverSocketChannel = ServerSocketChannel.open();
ServerSocket serverSocket = serverSocketChannel.socket();
serverSocket.bind(new InetSocketAddress(port));
serverSocketChannel.configureBlocking(false);
serverSocketChannel.register(selector, SelectionKey.OP_ACCEPT); ❶
while(running){
    selector.select();
    Iterator<SelectionKey> iterator = selector.selectedKeys().iterator();
    while(iterator.hasNext()){
        SelectionKey key = iterator.next();
        if(key.isAcceptable()){
            SocketChannel channel = key.accept();
            channel.configureBlocking(false);
            channel.register(selector, SelectionKey.OP_READ);
        } else if(key.isReadable()){
            SocketChannel channel = (SocketChannel)key.channel();
            ByteBuffer buffer = ByteBuffer.allocate(1024);
            channel.read(buffer);
            String message = new String(buffer.array());
            System.out.println("Received message: " + message);
        }
    }
}
```

05 IP 프로토콜은 네트워크상의 특정한 위치를 지정하는 방법입니다. UDP는 IP 프로토콜을 사용해 데이터를 전송하는 프로토콜입니다. 이 프로토콜은 데이터가 유실 없이 차례대로 전송됨을 보장해 주지 않습니다. 데이터가 순서대로 모두 전달됨을 보장하는 프로토콜은 TCP입니다. 이 TCP 프로토콜은 데이터를 구성하는 패킷에 일련번호를 부여하고 전송 받은 패킷을 이 번호대로 재 구성하여 빠진 데이터 패킷은 재 요청 처리로 채워 넣어 결과적으로 수신된 데이터가 순서가 맞게 재구성됨을 보장합니다.

```

try {
    selector.select((long)readSelectTimeout * 1000L); ❷
    Set<SelectionKey> selectionKeySet = selector.selectedKeys();
    Iterator<SelectionKey> selectionKeyIter = selectionKeySet.iterator();
    while(selectionKeyIter.hasNext()){
        SelectionKey selectionKey = selectionKeyIter.next();
        selectionKeyIter.remove();
        if(!selectionKey.isValid()){
            continue;
        }
        try{
            if(selectionKey.isAcceptable()){ ❸
                ServerSocketChannel oneServerSocketChannel
                    = (ServerSocketChannel)selectionKey.channel();
                SocketChannel socketChannel = oneServerSocketChannel.accept();
                socketChannel.configureBlocking(false);
                socketChannel.register(selector, SelectionKey.OP_READ); ❹
                // 생략.....
            }
            if(selectionKey.isReadable()){
                SocketChannel socketChannel = SocketChannel(selectionKey.channel()); ❺
                // 처리할 데이터가 들어왔으므로 스레드 풀에서 워커 스레드를 하나
                // 빼내어 처리를 수행합니다.
            }
        }catch(CancelledKeyException e){
            .....
        }
    }
} catch (IOException e) { ❻
    .....
}
}

```

클라이언트 소켓을 받아들이기 위에 서버 소켓 채널을 설정한 후 셀렉터에 등록합니다. 이때 소켓이 처음 접속을 시도할 때 이벤트를 받아 깨어나기 위해 셀렉터에 서버 소켓 채널을 등록할 때 SelectionKey.OP\_ACCEPT를 매개변수로 같이 넘겨 줍니다(❶).

이제 셀렉터의 select 메서드로 진입하는 데(❷) 마치 InputStream.read 메서드와 비슷하게 보입니다. 이 둘의 차이는 메서드의 매개변수로 최대 대기 시간을 설정한다는 것입니다. 따라서 데이터가 들어오지 않으면 최대 타임아웃 시간만큼만 기다립니다.

셀렉터에 등록된 채널에 데이터가 들어오는 등의 사건은 SelectionKey로 표현합니다. SelectionKey를 보고 해당 채널에 어떤 사건이 벌어졌는지 확인하여 그에 대응한 처리를 합니다. 지금은 셀렉터에 OP\_ACCEPT만 등록되어 있습니다. 따라서 외부 소켓이 서버 소켓에 접근하려고 하면 셀렉터의 select 메서드가 리턴되어 ❸ 라인의 if(selectionKey.isAcceptable()) 이하가 수행됩니다.

외부 소켓을 다시 소켓 채널로 만들고 OP\_READ 매개변수로 다시 셀렉터에 등록합니다(❹). 이것은 해당 소켓 채널에 데이터가 들어오면 이벤트를 발생시키라는 의미며, 이벤트를 잡아내는 부분이 if(selectionKey.isReadable()) 이하입니다(❺).

이제 데이터가 들어왔을 때만 워커 스레드가 할당되므로(28라인), BIO 방식과 같이 앞으로 들어올 데이터를 위하여 개별 스레드가 동기적으로 대기함으로써 블로킹 상태가 되지는 않습니다.

## 7.4 프론트엔드 서버로서의 서블릿 컨테이너

이 장을 시작하면서 자바가 Non-blocking IO를 지원하지 않았던 시기에는 웹 서버를 앞에 세울 수밖에 없었다는 이야기를 했습니다. 이제 대부분의 서블릿 컨테이너가 Non-blocking IO를 지원하므로, 굳이 웹 서버를 앞에 세우지 않아도 될 것처럼 보입니다. 하지만 지금까지도 서블릿 컨테이너를 클라이언트에 직접 노출하는 경우는 그렇게 많지 않습니다.

대표적인 이유 중 하나는 서블릿 처리 자체가 원래 동기적이라면 것입니다. Non-blocking IO를 사용하면 IO 대기를 최소화하여 CPU 사용량을 극대화할 수 있지만, 이것이 가능해지려면 작업 동기화가 필요한 위치, 일반적으로는 IO 대기가 발생하는 지점을 경계로 나눌 수 있어야 하고 각 나뉜 작업이 동시에 병렬로 처리할 수 있어야 합니다. 하지만 서블릿은 HttpServlet을 상속하여 doXXX 메서드를 오버라이딩하도록 구현하며, doXXX 메서드는 HTTP 요청을 동기적<sup>06</sup>으로 수행합니다. 따라서 Non-blocking IO의 장점을 살릴 수 있는 부분은 서블릿이 수행되기 전에 HTTP 요청을 받아들여 프로토콜에 따라 파싱하고 서블릿 요청ServletRequest을 생성하는 커넥터/리스너 모듈이 주된 대상입니다. 하지만 doXXX 메서드 내에 존재하는 비즈니스 로직<sup>07</sup> 실행은 앞서 말한 바와 같이 동기적으로 수행하기 때문에 하나의 워커 스레드를 할당하여 처리할 수밖에 없습니다.

다시 말해, Non-blocking IO를 사용해 많은 요청을 받아들려도 동시에 실행할 수 있는 서블릿 인스턴스의 수는 해당 기계의 가용 스레드 수를 초과할 수 없으므로 HTTP 요청을 처리하는 전체 단위 시간당 처리량throughput의 관점에서 blocking IO를 사용하는 방식과 큰 차이는 없습니다.<sup>08</sup>

---

06 메서드 내에서는 차례로 코드가 진행됩니다.

07 서블릿 작성자가 추가하는 애플리케이션 코드를 의미합니다.

08 가용 스레드의 수에 의하여 제한을 받는다는 점 이외에도 프로토콜 분석 및 메시지 작성에 Non-blocking IO를 사용하는 것 만으로는 서블릿을 실행시키는 워커 스레드가 블로킹 되는 경우 해당 서블릿 수행이 블로킹되는 것을 피할 수 없다는 점에서 그러합니다.

하지만 BIO 방식의 커넥터/리스너가 가진 I/O상에서 블로킹이 될 수 있는 문제를 NIO에서 해결할 수 있다는 점에서 앞으로 점점 더 NIO 기반의 구현이 일반화되고 사용될 것으로 예상합니다.

## 7.5 NIO 기반의 HTTP 프로토콜 상태 기계 구현

4장에서 BIO 기반의 HTTP 프로토콜 상태 기계를 구현했습니다. 이번에는 NIO를 사용해 HTTP 프로토콜 상태 기계를 구현하겠습니다.

기존 프로그램이 요청을 받는 서버 모듈이었으므로 이번에는 HTTP 요청을 특정 웹 서버로 전달하고 그 응답을 파싱하여 표시하는 클라이언트 모듈을 구현하겠습니다. 다음은 NIO를 사용해 HTTP 프로토콜로 원격 서버에 요청을 전달하고, selector를 사용해 비동기로 응답을 전달받는 프로그램입니다.

---

```
package com.endofhope.scbook.ch06;

import java.io.IOException;
import java.net.InetSocketAddress;
import java.nio.ByteBuffer;
import java.nio.channels.SelectionKey;
import java.nio.channels.Selector;
import java.nio.channels.SocketChannel;
import java.util.ArrayList;
import java.util.HashMap;
import java.util.Iterator;
import java.util.List;
import java.util.Map;
import java.util.Set;

public class HttpClient {
```

```
public static void main(String[] args){
    // HTTP 요청을 전달할 호스트명과 IP를 설정합니다.
    String host = "endofhope.com";
    int port = 80;
    // HTTP 요청의 시작 줄을 설정합니다.
    // GET 방식으로 /를 HTTP 1.1 버전을 사용해 요청합니다.
    String requestLine = "GET / HTTP/1.1";
    // HTTP 헤더를 설정합니다.
    // 먼저 Host 헤더에 앞서 입력받은 호스트명:포트를 지정합니다.
    Map<String, String> headerMap = new HashMap<String, String>();
    headerMap.put("Host", host+":"+port);
    // 이번 예에서는 GET 방식 호출이므로 HTTP 바디는 없습니다.
    byte[] bodyBytes = null;
    // 지금까지 생성한 정보를 생성자에 추가해 객체를 생성하고 send 메서드를 호출합니다.
    HttpClient hc = new HttpClient(host, port, requestLine, headerMap, bodyBytes);
    hc.send();
}

private String host;
private int port;
private byte[] sendBytes;
public HttpClient(String host, int port, String requestLine,
    Map<String, String> headerMap, byte[] bodyBytes){
    this.host = host;
    this.port = port;
    // 생성자에서는 전달받은 인자를 조합하여 HTTP 요청을 생성합니다.
    // request-line과 헤더를 \r\n을 사용해 나열합니다.
    StringBuilder sb = new StringBuilder();
    sb.append(requestLine).append("\r\n");
    Set<String> headerKeySet = headerMap.keySet();
    Iterator<String> headerKeyIter = headerKeySet.iterator();
    while(headerKeyIter.hasNext()){
        String name = headerKeyIter.next();
```

```

        String value = headerMap.get(name);
        sb.append(name).append(": ").append(value).append("\r\n");
    }

// 전송할 바디가 있다면 Content-Length 헤더에 바디의 크기를 값으로 추가합니다.
if(bodyBytes != null) sb.append("Content-Length:
").append(bodyBytes.length).append("\r\n");

// 연결을 재사용하지 않음을 Connection 헤더 값에 close를 설정해 알려줍니다.
// 만약 재사용할 예정이라면 서버가 요청을 임의로 끊지 말 것을 요청하는 의미로
// close 대신 keep-alive를 전송합니다.
sb.append("Connection: close\r\n");

// 헤더부의 끝을 의미하기 위해 빈 라인 (\r\n\r\n)이 들어갔습니다.
sb.append("\r\n");

byte[] headerBytes = sb.toString().getBytes();
// 전송할 내용을 의미하는 sendBytes에 지금까지 구성한 내용을 저장합니다.

if(bodyBytes == null){
    sendBytes = headerBytes;
} else{
    sendBytes = new byte[headerBytes.length + bodyBytes.length];
    System.arraycopy(headerBytes, 0, sendBytes, 0, headerBytes.length);
    System.arraycopy(bodyBytes, 0, sendBytes, headerBytes.length,
bodyBytes.length);
}
}

public static final long SELECT_INTERVAL = 200L;
public static final int BUFFER_SIZE = 256;
private static final ByteBuffer readBuffer =
ByteBuffer.allocate(HttpClient.BUFFER_SIZE);

// 생성자에서 세팅된 값을 기반으로 실제 전송을 시도합니다.
public void send(){

    SocketChannel socketChannel = null;
    Selector selector = null;
    boolean isEnd = false;
}

```

```

try {
    // 먼저 소켓 채널을 생성한 후, 비동기 설정인 non-blocking을 추가합니다.
    socketChannel = SocketChannel.open(new InetSocketAddress(host, port));
    socketChannel.configureBlocking(false);
    // 앞서 보낼 내용을 지정한 sendBytes를 ByteBuffer 형태로 바꾼 후 전송합니다.
    socketChannel.write(ByteBuffer.wrap(sendBytes));
    // 이제 본격적으로 비동기적으로 읽기가 시작됩니다.
    // 셀렉터를 하나 열고, 읽기 이벤트가 발생하면 깨어나겠다는 것을 지정합니다.
    selector = Selector.open();
    socketChannel.register(selector, SelectionKey.OP_READ);
    while(!isEnd){
        // 이제 SELECT_INTERVAL 시간 만큼 위에서 지정한 (READ) 이벤트를 기다립니다.
        selector.select(HttpClient.SELECT_INTERVAL);
        // 두 가지 경우 select 메서드가 반환됩니다.
        // 첫 번째는 대기하던 이벤트가 왔다면 반환됩니다.
        // 두 번째는 이벤트가 오지 않아도 지정한 타임아웃 시간(SELECT_INTERVAL)이
        // 지나면 반환됩니다.
        // 두 번째 경우에는 selectionKeys.iterator().hasNext()가 false이므로
        // 다시 위의 while 루프로 돌아가 다시 select 메서드에서
        // 이벤트를 대기하게 되겠습니다.

        Set<SelectionKey> selectionKeys = selector.keys();
        Iterator<SelectionKey> selectionKeyIter = selectionKeys.iterator();
        while(selectionKeyIter.hasNext()){
            SelectionKey selectionKey = selectionKeyIter.next();
            // 만약 지정된 이벤트가 들어왔다면 SelectionKey이 반환되게 되고
            // 이제 이벤트 타입 별로 각각의 처리가 시작됩니다.
            // 위에서 READ 이벤트만 Selector에 등록했으므로
            // 이벤트가 들어왔다면 아래 if 구문이 true가 됩니다.

            if(selectionKey.isReadable()){
                // 이제 읽기 이벤트가 발생했으며 소켓 채널에서 실질적으로 읽게 됩니다.
                // 문제는 읽어들였을 때 이번에 읽은 양이 전체 메시지라는 보장이 없습니다.
                // 이론적으로 IP 패킷의 최대 크기는 65535입니다만

```

```

// 실질적으로는 그보다 훨씬 작은 크기가 사용됩니다.09
// 따라서 많은 경우 하나의 HTTP 메시지는 여러 개의 패킷으로 이루어집니다.
// 먼저 소켓 채널에서 읽어 보고, 아직 메시지가 완성되지 않아
// 더 전달되어야 할 내용이 있다면 지금까지 읽은 내용을 저장하여 두고
// 다음 읽기 이벤트가 발생했을 때 다시 읽은 후
// 앞에서 저장된 내용과 합쳐 전체 메시지가 전달됐는지 확인하게 됩니다.
// 이를 위해 저장하여 두는 곳이 바로 SelectionKey.attachment가 되겠습니다.

    MessageBag messageBag = (MessageBag)selectionKey.attachment();
    if(messageBag == null){
        messageBag = new MessageBag();
        selectionKey.attach(messageBag);
    }

// 이제 읽기를 시도합니다.
// 읽기 버퍼를 초기화하고 읽습니다.

    readBuffer.clear();
    socketChannel.read(readBuffer);
    readBuffer.flip();
    while(readBuffer.hasRemaining()){

// 읽어들인 내용을 앞에서부터 한 바이트씩 보고
// HTTP 메시지에서 현재 어떤 상태인지 확인하며 처리하게 됩니다.

        byte oneByte = readBuffer.get();
// 처음 시작입니다. 상태를 요청 라인으로 바꿉니다.

        if(messageBag.status == Status.INIT){
            messageBag.add(oneByte);
            messageBag.status = Status.REQUEST_LINE;
// 첫 줄인 요청 라인을 읽어들이다가 \r이 나오면 한 행이 끝남을,
// 곧 요청 라인이 종료되었음을 알 수 있습니다.

        }else if(messageBag.status == Status.REQUEST_LINE){
            if(oneByte == CR){


```

---

09 네트워크로 연결된 양쪽에서 한 번에 읽어들일 수 있는 패킷의 크기가 다를 수 있으므로 TCP/IP로 양쪽이 통신을 시작할 때에는 자신의 MSS(Maximum Segment Size)를 상대에게 알려주도록 약속돼 있습니다. 이 정보를 기반으로 전송 시 메시지를 상대의 MSS 값보다 작은 단위로 나눠 보냅니다.

```

        messageBag.status = Status.REQUEST_LINE_CR;
        messageBag.setRequestLine();
    }else{
        messageBag.add(oneByte);
    }
}else if(messageBag.status == Status.REQUEST_LINE_CRLF){
    messageBag.status = Status.REQUEST_LINE_CRLF;
}else if(messageBag.status == Status.REQUEST_LINE_CRLFCR){
// 요청 라인이 종료된 이후에는 헤더로 간주합니다.
    messageBag.add(oneByte);
    messageBag.status = Status.HEADER;
}else if(messageBag.status == Status.HEADER){
    if(oneByte == CR){
        messageBag.addHeader();
        messageBag.status = Status.HEADER_CR;
    }else{
        messageBag.add(oneByte);
    }
}else if(messageBag.status == Status.HEADER_CR){
// 헤더 상태에서 \r을 만나면 하나의 헤더값이 완성되었다고 간주합니다.
    if(oneByte == LF){
        messageBag.status = Status.HEADER_CRLF;
    }else{
        throw new IllegalStateException("LF must be followed.");
    }
}else if(messageBag.status == Status.HEADER_CRLF){
    if(oneByte == CR){
        messageBag.status = Status.HEADER_CRLFCR;
    }else{
        messageBag.add(oneByte);
        messageBag.status = Status.HEADER;
    }
}

```

```

}else if(messageBag.status == Status.HEADER_CRLF_CRF){
// 빈 헤더값을 만나면 이제 헤더부가 끝났고
// 지금까지 들어온 요청 라인과 헤더를 파싱하여
// 메시지 바디 유무를 판단하여 더 필요하면 읽기를 계속합니다.

    if(oneByte == LF){
        BodyStyle bodyStyle = messageBag.afterHeader();
        if(bodyStyle == BodyStyle.NO_BODY){
            messageBag.status = Status.TERMINATION;
            break;
        }else{
            messageBag.status = Status.BODY;
        }
    }else{
        throw new IllegalStateException("LF must be followed.");
    }
}else if(messageBag.status == Status.BODY){
// 메시지 바디가 있는 경우
    if(messageBag.bodyStyle == BodyStyle.CONTENT_LENGTH){
// Content-Length 헤더가 있다면 해당 헤더 값만큼 바디를 추가로 읽습니다.
        messageBag.add(oneByte);
        if(messageBag.getContentLength() <= messageBag.getBytesSize()){
// 메시지 바디를 끝까지 다 읽었다면
// 지금까지 읽은 값을 bodyBytes에 넣고, 상태를 종료로 표시하고 나갑니다.
            messageBag.setBodyBytes();
            messageBag.status = Status.TERMINATION;
            break;
        }
    }else if(messageBag.bodyStyle == BodyStyle.CHUNKED){
// 메시지 청크 방식인 경우 HTTP 바디에 대해 다시 상태 기계를 만듭니다.
        if(messageBag.chunkStatus == ChunkStatus.CHUNK_NUM){
            if(oneByte == CR){
                messageBag.setChunkSize(Integer.parseInt(new

```

```

        String(messageBag.toBytes(), 16));
    messageBag.chunkStatus = ChunkStatus.CHUNK_NUM_CR;
}else{
    messageBag.add(oneByte);
}
}else if(messageBag.chunkStatus == ChunkStatus.CHUNK_NUM_CR){
    if(oneByte == LF){
        if(messageBag.getChunkSize() == 0){
            // 크기가 0인 청크는 청크 방식 메시지 바디의 끝을 의미합니다.
            // 메시지 상태를 종료로 표시하고 나갑니다.
            messageBag.setChunkBodyBytes();
            messageBag.status = Status.TERMINATION;
            break;
        }else{
            messageBag.chunkStatus = ChunkStatus.CHUNK_BODY;
        }
    }else{
        throw new IllegalStateException("LF must be followed by CR");
    }
}else if(messageBag.chunkStatus == ChunkStatus.CHUNK_BODY){
    if(messageBag.getBytesSize() == messageBag.getChunkSize()-1){
        messageBag.add(oneByte);
        messageBag.addChunk();
        messageBag.chunkStatus = ChunkStatus.CHUNK_END;
    }else{
        messageBag.add(oneByte);
    }
}else if(messageBag.chunkStatus == ChunkStatus.CHUNK_END){
    if(oneByte == CR){
        messageBag.chunkStatus = ChunkStatus.CHUNK_CR;
    }else{
        throw new IllegalStateException("CR must be followed by chunk");
    }
}

```

```
        }
    }else if(messageBag.chunkStatus == ChunkStatus.CHUNK_CR){
        if(oneByte == LF){
            messageBag.chunkStatus = ChunkStatus.CHUNK_CRLF;
        }else{
            throw new IllegalStateException("LF must be followed by CR");
        }
    }else if(messageBag.chunkStatus == ChunkStatus.CHUNK_CRLF){
        messageBag.add(oneByte);
        messageBag.chunkStatus = ChunkStatus.CHUNK_NUM;
    }
}
}

}

}

// 읽기 버퍼를 끝까지 다 읽었습니다.
// 지금까지 읽어들인 값을 다시 SelectionKey.attachment에 넣어둡니다.
selectionKey.attach(messageBag);
if(messageBag.status == Status.TERMINATION){
    // 메시지를 끝까지 다 읽은 것이라면
    // SelectionKey.attachment 를 제거하고
    // 지금까지 읽어들인 값을 표시합니다.
    selectionKey.attach(null);
    isEnd = true;
    messageBag.process();
    break;
}
}

}

}

socketChannel.close();
selector.close();
} catch (IOException e) {
```

```

        e.printStackTrace();
    } finally {
        if(socketChannel != null){ try{ socketChannel.close(); }
            }catch(IOException e){} };
        if(selector != null){ try{ selector.close(); }catch(IOException e){} };
    }
}

public static final byte CR = '\r';
public static final byte LF = '\n';

// HTTP 메시지가 가질 수 있는 상태를 표시합니다.

public enum Status{
    INIT, REQUEST_LINE, REQUEST_LINE_CR, REQUEST_LINE_CRLF,
    HEADER, HEADER_CR, HEADER_CRLF, HEADER_CRLFCR,
    BODY,
    TERMINATION
}
// HTTP Body의 상태를 세분화합니다.

public enum BodyStyle{
    NO_BODY, CONTENT_LENGTH, CHUNKED
}

// HTTP Body 중에서 청크 탑입일 때의 상태를 표시합니다.

// 청크가 숫자, CR, LF, 앞선 숫자만큼의 바이트 배열로 구성됩니다.

public enum ChunkStatus{
    CHUNK_NUM, CHUNK_NUM_CR, CHUNK_NUM_CRLF, CHUNK_BODY,
    CHUNK_END, CHUNK_CR, CHUNK_CRLF
}

class MessageBag{
    // 이 클래스는 HTTP 메시지를 구성하는 구성 요소를 표현할 수 있게 구성되었습니다.

private Status status = Status.INIT;

private List<Byte> byteList = new ArrayList<Byte>();

protected byte[] toBytes(){
    byte[] bytes = new byte[byteList.size()];

```

```
for(int i=0; i<byteList.size(); i++){
    bytes[i] = byteList.get(i);
}
byteList.clear();
return bytes;
}

protected void add(byte oneByte){
    byteList.add(oneByte);
}

protected int getBytesSize(){
    return byteList.size();
}

private String requestLine;
protected String getRequestLine(){ return requestLine; }
protected void setRequestLine(){
    requestLine = new String(toBytes());
}

private Map<String, String> headerMap = new HashMap<String, String>();
protected void addHeader(){
    String headerLine = new String(toBytes());
    int indexOfColon = headerLine.indexOf(":");
    headerMap.put(headerLine.substring(0, indexOfColon).trim(),
        headerLine.substring(indexOfColon+1).trim());
}

private int contentLength;
protected int getContentLength(){ return contentLength; }

private BodyStyle bodyStyle;
private ChunkStatus chunkStatus;
private int chunkSize = -1;
protected void setChunkSize(int chunkSize){
    this.chunkSize = chunkSize;
}
```

```

protected int getChunkSize(){
    return chunkSize;
}

// 헤더가 다 들어온 것이 확인되면
// 각각의 이름/값 쌍으로 헤더를 재 구성하고
// 메시지 바디가 있는지, 얼마나
// 혹은 어떻게 메시지 바디를 읽어야 하는지 확인하는 역할을 하는 메서드입니다.

protected BodyStyle afterHeader(){
    bodyStyle = BodyStyle.NO_BODY;

    Set<String> headerKeySet = headerMap.keySet();
    Iterator<String> headerKeyIter = headerKeySet.iterator();
    while(headerKeyIter.hasNext()){

        String headerName = headerKeyIter.next();
        String headerValue = headerMap.get(headerName);
        if("Content-Length".equals(headerName)){
            contentLength = Integer.parseInt(headerValue);
            bodyStyle = BodyStyle.CONTENT_LENGTH;
        }else if("Transfer-Encoding".equals(headerName) &&
                 "chunked".equals(headerValue)){
            bodyStyle = BodyStyle.CHUNKED;
            chunkStatus = ChunkStatus.CHUNK_NUM;
        }
    }

    return bodyStyle;
}

private byte[] bodyBytes;
protected void setBodyBytes(){
    bodyBytes = toBytes();
}

protected byte[] getBodyBytes(){
    return bodyBytes;
}

```

```

// 청크 방식 바디의 경우 여러 청크들이 하나의 바디를 이루므로
// 리스트 형태로 유지합니다.
private List<byte[]> chunkList = new ArrayList<byte[]>();
protected void addChunk(){
    chunkList.add(toBytes());
}

// 여러 청크 바디를 하나의 바디로 묶어주는 유ти리티 메서드입니다.
protected void setChunkBodyBytes(){
    int bodyBytesLength = 0;
    for(int i=0; i<chunkList.size(); i++){
        bodyBytesLength += chunkList.get(i).length;
    }
    bodyBytes = new byte[bodyBytesLength];
    int destPos = 0;
    for(int i=0; i<chunkList.size(); i++){
        System.arraycopy(chunkList.get(i), 0, bodyBytes, destPos,
            chunkList.get(i).length);
        destPos += chunkList.get(i).length;
    }
}

protected void process(){
    // 읽어들인 HTTP 메시지를 보여주기 위한 메서드입니다.
    System.out.printf("%s\n", requestLine);
    Set<String> headerKeySet = headerMap.keySet();
    Iterator<String> headerKeyIter = headerKeySet.iterator();
    while(headerKeyIter.hasNext()){
        String headerName = headerKeyIter.next();
        String headerValue = headerMap.get(headerName);
        System.out.printf("%s: %s\n", headerName, headerValue);
    }
    System.out.printf("\n");
    if(bodyBytes != null){

```

```
        System.out.println(new String(getBodyBytes()));
    }
}
}
}
```

---

다음과 같이 컴파일하고 실행하면 다음과 같은 결과를 얻습니다.

그림 7-1 HttpClient의 결과 화면

The screenshot shows a Windows Command Prompt window with the title '관리자: C:\Windows\System32\cmd.exe'. The command entered is 'java -cp classes com.endofhope.sbook.ch07.HttpClient'. The output is a rendered HTML document containing various links and titles, such as 'END OF HOPE', 'Tractatus Logico-Philosophicus', and 'Neurasthenia'.

```
C:\examples\ch07>java -cp classes com.endofhope.sbook.ch07.HttpClient
HTTP/1.1 200 OK
Transfer-Encoding: chunked
Date: Wed, 19 Sep 2012 14:09:26 GMT
P3P: CP="CAO PSA COMI OTR OUR DEM ONL"
Content-Type: text/html
Connection: close
X-Powered-By: PHP/5.1.6
Server: Microsoft-IIS/5.0

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.1//EN" "http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="ko-KR">
<head>
    <link rel="stylesheet" type="text/css" href="main.css">
    <meta http-equiv="Content-Type" content="text/html; charset=UTF-8" />
    <title>END OF HOPE</title>
</head>
<body>
    <div class="doc_wrapper">
        <div class="base">
            <ul class="mine_link_list">
                <li>EOH <a href="/tc/">BLOG</a> Tractatus Logico-Philosophicus</li>
                <li>Neurasthenia <a href="http://dev.naver.com/projects/neurasthenia/">Project Home</a></li>
                <li>Neurasthenia <a href="http://endofhope.linuxstudy.pe.kr">Demo site</a></li>
                <li><a href="http://endofhope.blogspot.com">OLD BLOG</a> 대장 紬鬱委沮</li>
            </ul>
        </div>
    </div>
</body>
</html>

C:\examples\ch07>
```

## 7.6 더 생각해 볼 문제

1. 위 예제 코드를 자세히 살펴본 독자라면 HTTP 응답은 비동기적으로 받지만, 요청은 동기적으로 전송했다는 사실을 알아차렸을 것으로 생각합니다. 그렇다면 요청까지 비동기적으로 보내려면 어떻게 하면 될까요? 힌트는 우리가 셀렉터에 관심 이벤트를 등록할 때 “쓰기 가능한가” 여부를 추가하는 것이 첫 단추입니다.
2. HTTP 메시지를 전송하고 응답을 받아 표시하는 기능 정도면 동기적 IO, 즉 Blocking IO를 사용하면 같은 기능을 하는 코드를 더 쉽게 작성할 수 있을 텐데 굳이 Non-blocking IO를 사용하는 이유는 무엇일까요?
3. Blocking IO를 사용해 동일한 역할을 하는 코드를 작성해 보고, 특정 시간 이상이 걸리면 즉시 오류를 반환하는 타임아웃 기능을 가지도록 변경하여 보십시오.
4. 비동기를 사용한 예제에서 동일한 타임아웃 기능을 추가하려면 어디를 수정해야 할지 찾아보십시오.

## 8 | 서버 프로그램으로서의 서블릿 컨테이너

대저, 물이 깊지 못하면 큰 배를 띠울 만한 힘이 없다. 한 잔의 물을 뜰의 패인 곳에 붓고 거자씨를 배 삼아 띠우면 뜨지만, 거기에 술잔을 놓으면 땅에 닿을 것이다. 물은 얹고, 배는 크기 때문이다. 바람의 쌩임이 두텁지 못하면, 봉새의 큰 날개를 지탱할 만한 힘이 없다.

- 장자(『소요유』 중에서)

지금까지 클래스 파일을 읽어들여 서블릿 인스턴스로 만들고, HTTP 요청을 받아들여 적절한 서블릿을 할당하는 등의 서블릿을 동작시키기 위해 서블릿 컨테이너가 제공해야 하는 기능에 대해 알아보았습니다. 물론 서블릿 컨테이너는 서블릿을 동작시키는 것이 주요한 목적이지만, 이런 목적을 달성하려면 서블릿 컨테이너가 가져야 하는 서버 기능도 중요합니다. 이 장에서는 서블릿을 담아두고 관리하는 컨테이너의 역할에 초점을 맞춰 살펴보겠습니다.

어떻게 구현하느냐에 따라 컨테이너의 역할이 조금씩 다르므로, 이 장에서는 톰캣 서블릿 컨테이너에 대해서만 살펴보겠습니다.

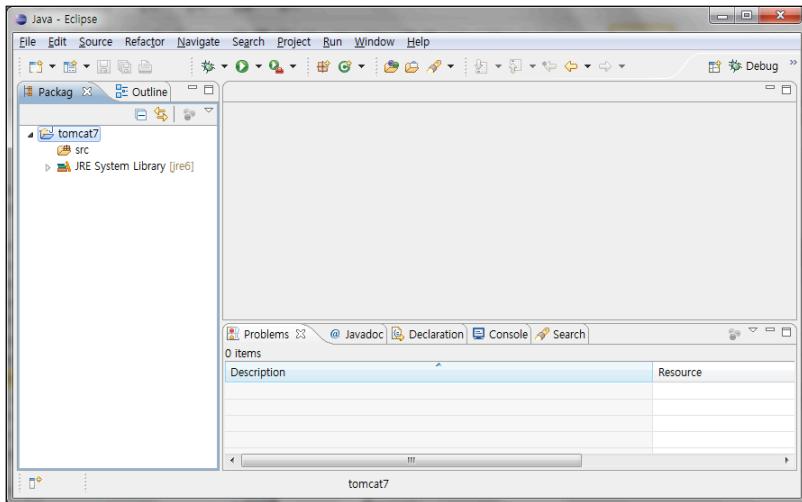
### 8.1 서블릿 컨테이너 분석

서블릿 컨테이너를 분석할 때 가장 처음으로 만나는 장벽은 프로그램 시작점을 파악하는 것입니다. 클라이언트 프로그램은 명시적으로 시작되는 지점(사용자가 해당 프로그램을 구동시키면 처음으로 불리는 곳)이 비교적 분명하지만, 서버 프로그램인 서블릿 컨테이너는 시작과 동시에 사용자의 개입 없이 초기화 과정을 수행하므로 어디부터 분석을 시작해야 할지 막막하게 느껴지는 경우가 많습니다. 하지만 서블릿 컨테이너도 main 메서드를 가진 자바 애플리케이션입니다. 초기화 코드의 시작 시점에서 따라가 본다면 충분히 분석할 수 있고, 또 그럴 만한 가치가 있습니다. 대중적으로 많이 사용하는 아파치 톰캣 서블릿 컨테이너의 부팅 과정을 따라가 보

겠습니다.

3장에서 Apache-tomcat 7.x의 소스코드를 이용해 톰캣 서버를 구동했습니다.  
이클립스 IDE를 사용해 해당 서버에 디버거를 걸어<sup>01</sup> 보겠습니다.<sup>02</sup>

먼저 tomcat7이라는 이름으로 Java Project를 생성합니다.

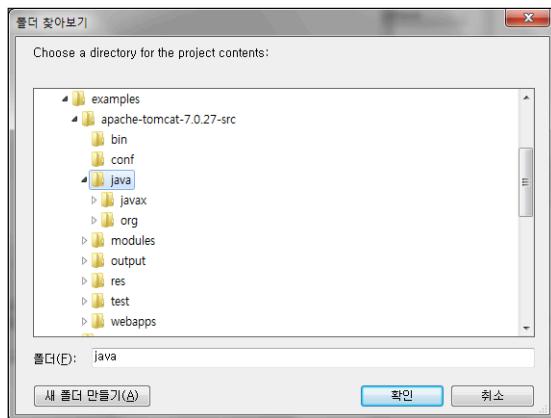
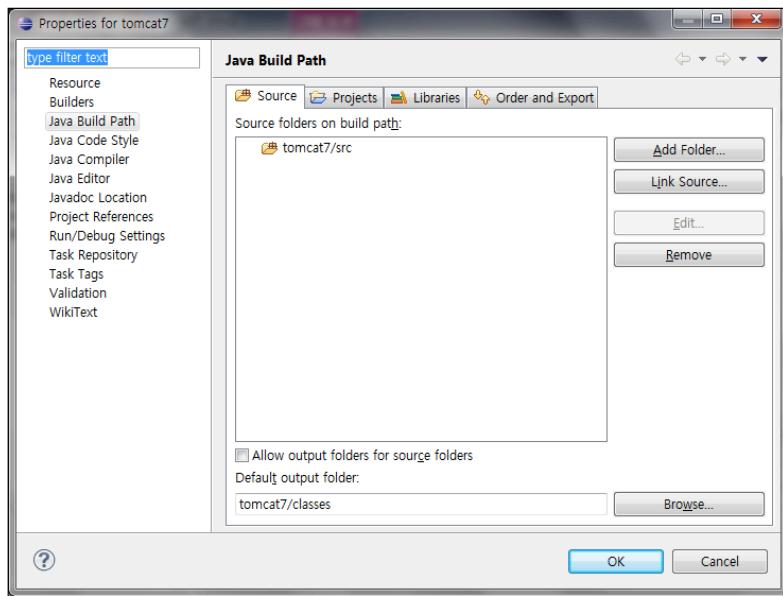


[Link Source] 메뉴를 이용해 해당 프로젝트의 소스 디렉터리를 기존의 tomcat7 java 소스 파일이 들어있는 디렉터리로 지정합니다.

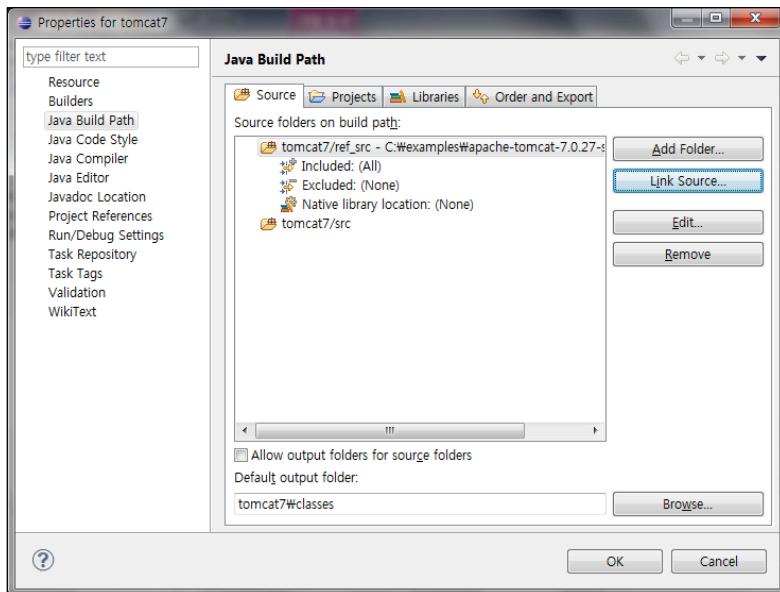
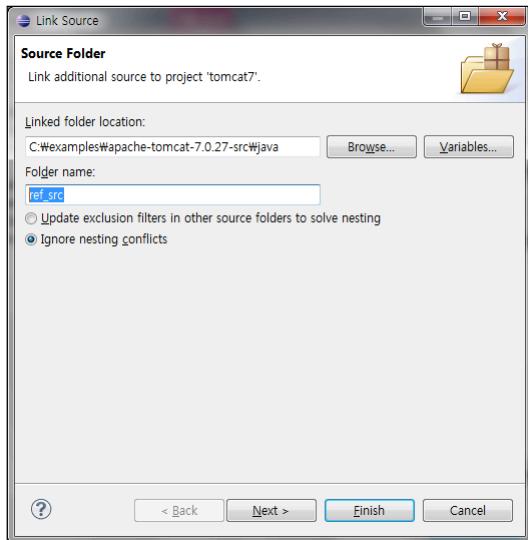
---

01 디버깅 툴을 이용해 프로그램 진행을 살펴보겠다는 의미입니다.

02 이클립스 IDE에 대한 자료는 이클립스 프로젝트(<http://eclipse.org>)에서 찾아볼 수 있습니다.

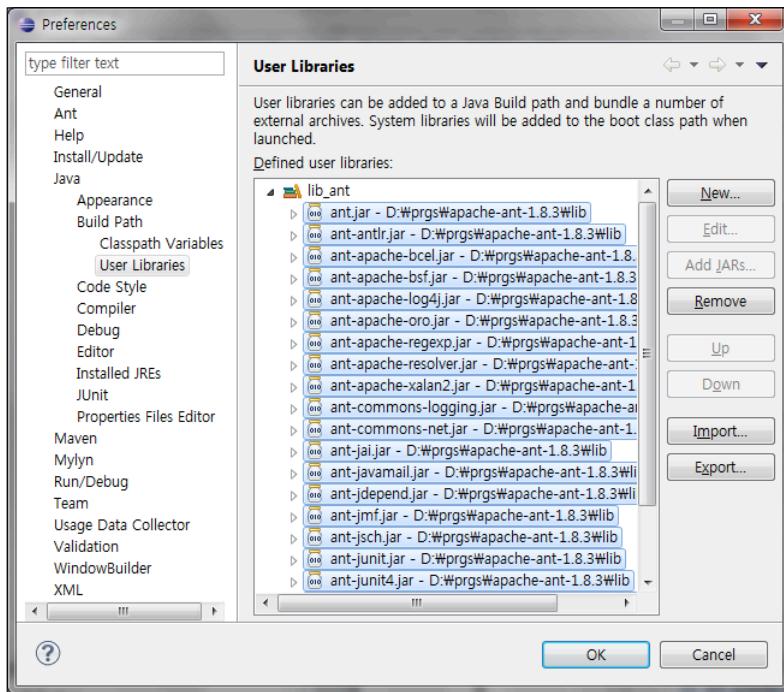


이름을 ref\_src로 지정하겠습니다.

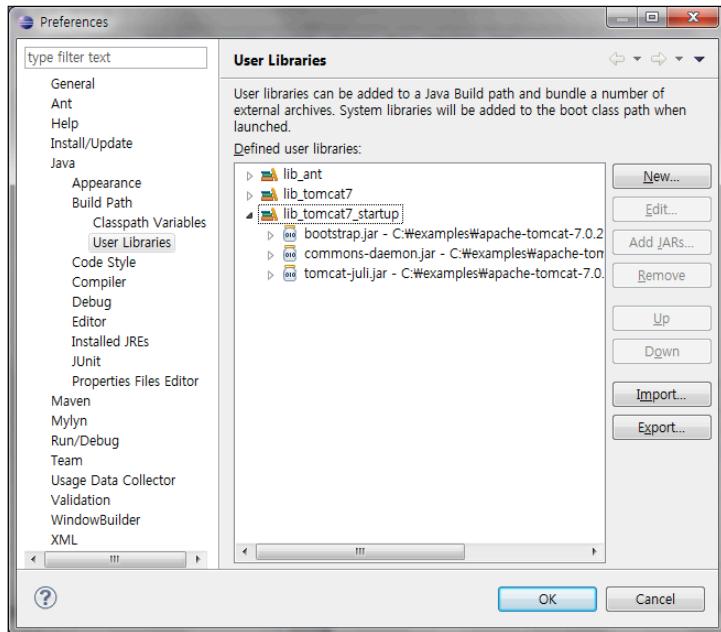


만약 자동 빌드 옵션이 켜져 있다면 [OK] 버튼을 클릭합니다. 컴파일하면 많은 에러를 보고할 것입니다. 이제 tomcat7 컴파일에 필요한 여러 라이브러리를 추가하겠습니다.

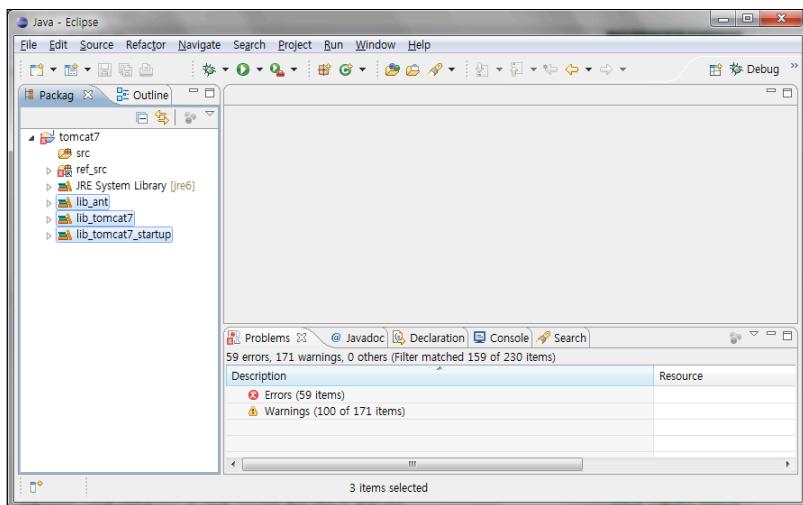
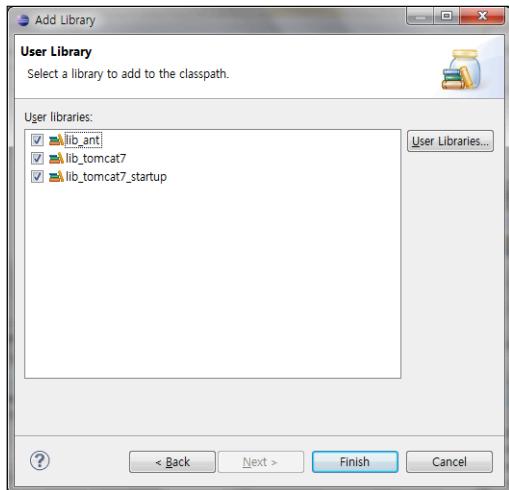
먼저 ant가 설치된 디렉터리의 하위 lib 디렉터리에 있는 .jar 파일을 사용자 라이브러리로 등록합니다.



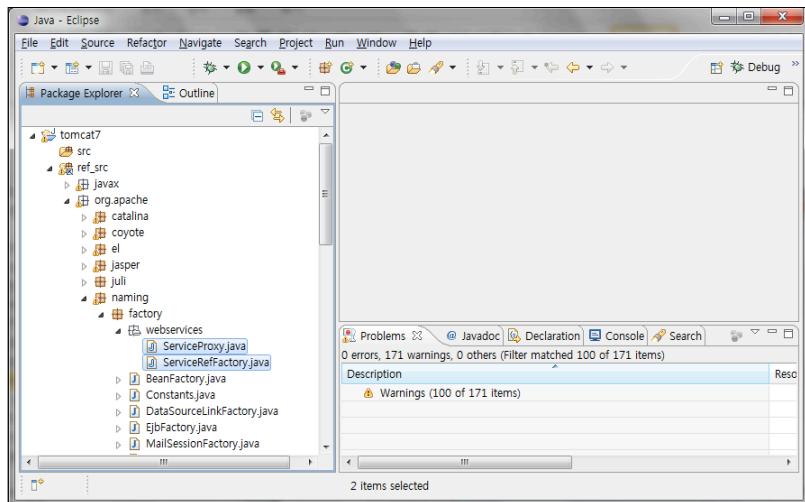
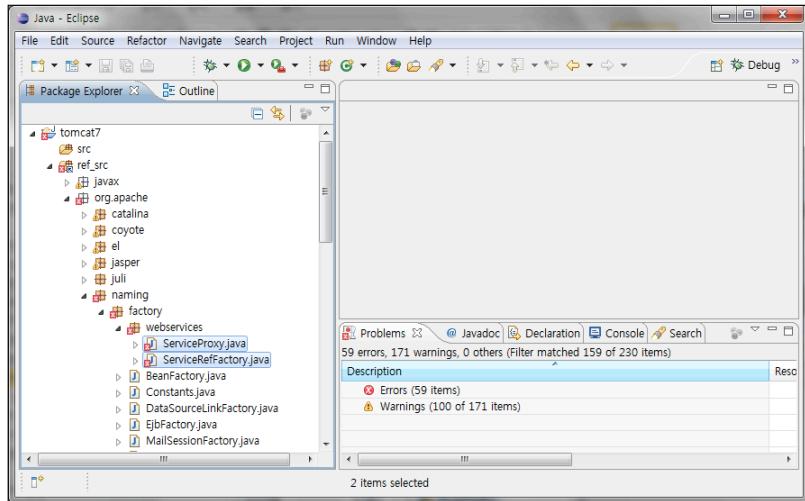
빌드한 tomcat7의 lib 디렉터리 내의 jar 파일을 lib\_tomcat7이란 이름으로 등록하고, bin 디렉터리 내의 jar 파일을 lib\_tomcat7\_startup으로 등록합니다.



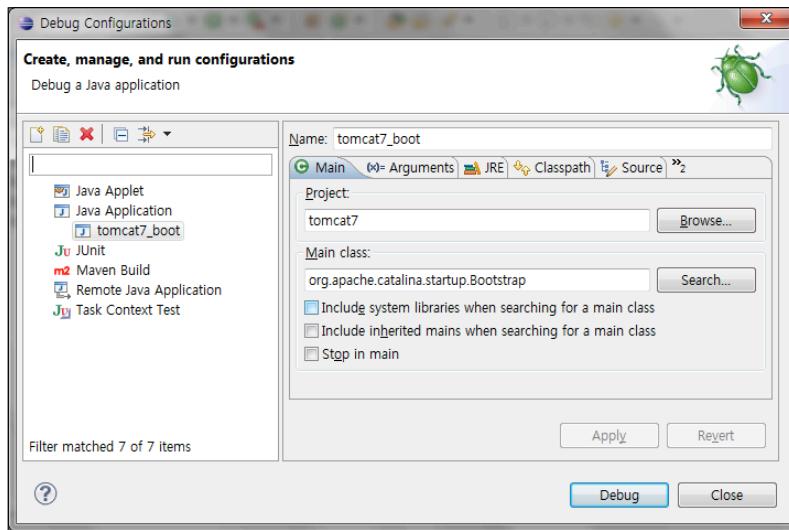
등록한 사용자 라이브러리를 tomcat7 프로젝트로 추가합니다.



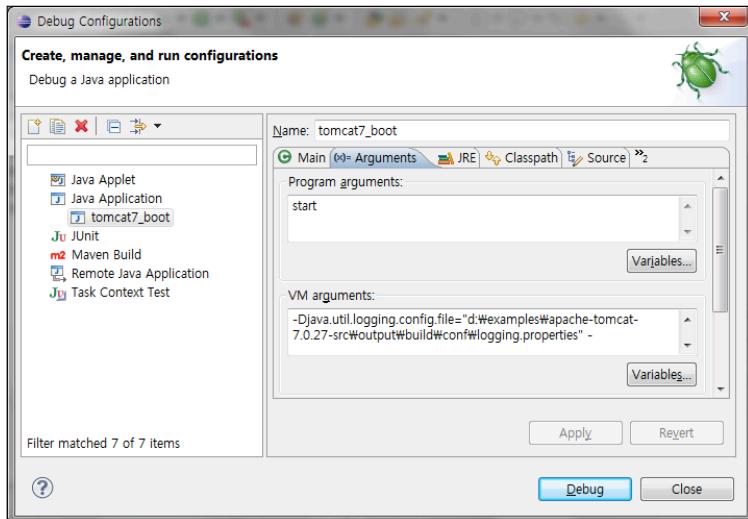
컴파일에 실패한 부분은 웹 서비스 지원을 위해 필요한 부분입니다. Axis 라이브러리를 추가해 처리할 수도 있고, 단순히 exclude를 사용해 컴파일 범위에서 처리 할 수도 있습니다.



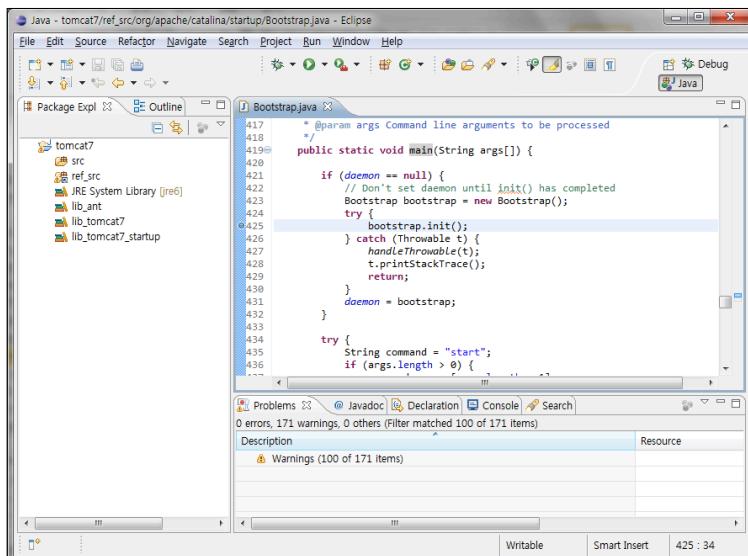
이제 컴파일은 성공했으니 톰캣을 구동해 보겠습니다. 3장에서 catalina.bat 스크립트 내에 echo 구문을 추가해 jvm 구동 옵션을 살펴보았습니다. 표시된 내용을 적절히 가감하여 이클립스 IDE 디버깅 설정을 추가합니다. main 클래스가 org.apache.catalina.startup.Bootstrap인 것을 확인할 수 있습니다.



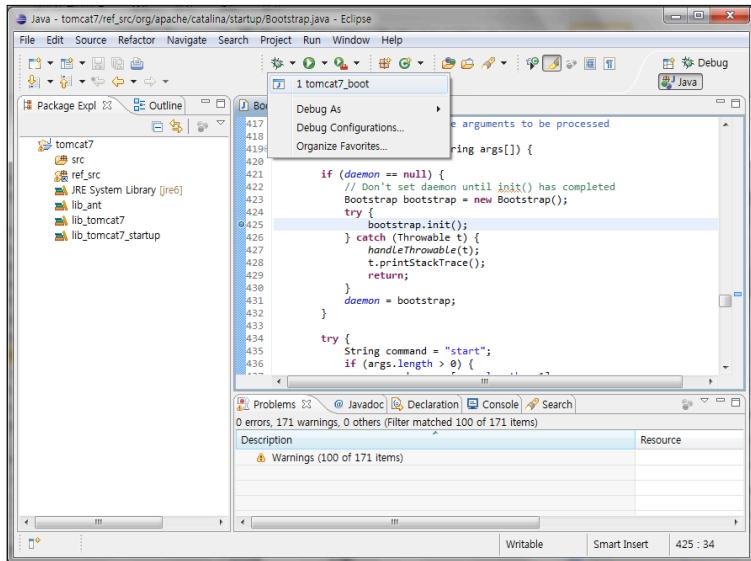
또한, main 클래스에 전달한 인자인 start 문자열과 VM에 전달할 인자창에 나머지를 추가합니다.



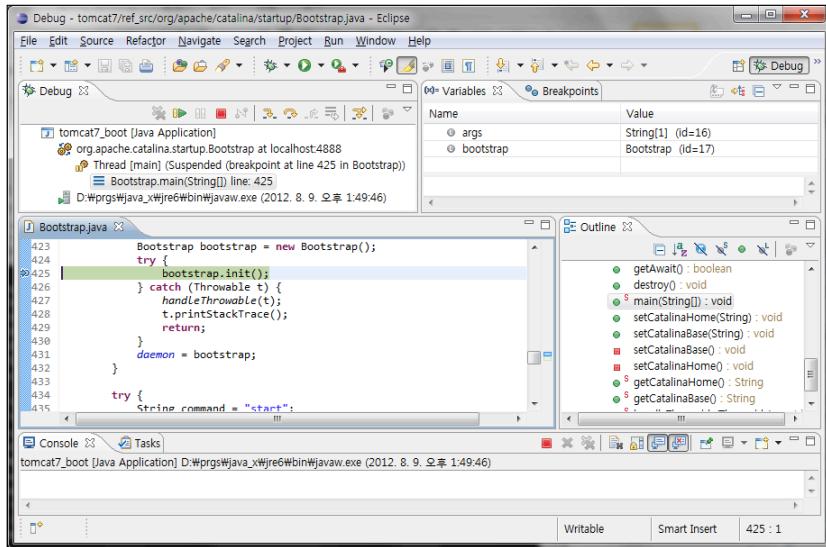
이제 디버거를 걸기 위해 Bootstrap 클래스의 main 메서드 내에 다음과 같이 브레이크포인트를 설정합니다.



별레 모양의 디버거 버튼을 클릭해 디버깅을 시작합니다.



축하합니다! 이제 시간과 의지만 있다면 현재 가장 많이 사용되는 서블릿 컨테이너의 동작을 한 발자국씩 따라가며 빠짐없이 살펴볼 수 있습니다.



## 8.2 부팅과정에서 벌어지는 일들

다음은 앞에서 살펴본 tomcat 7 서블릿 컨테이너 시작 클래스인 Bootstrap의 main 메서드입니다.

---

```

public static void main(String args[]) {
    if (daemon == null) {
        // Don't set daemon until init() has completed
        Bootstrap bootstrap = new Bootstrap(); // Bootstrap 클래스를 생성
        try {
            bootstrap.init();
        } catch (Throwable t) {
            handleThrowable(t);
            t.printStackTrace();
        }
    }
}

```

```
        }

        daemon = bootstrap; // Bootstrap 인스턴스를 할당
    }

    try {
        String command = "start";
        if (args.length > 0) {
            command = args[args.length - 1];
        }

        if (command.equals("startd")) {
            args[args.length - 1] = "start";
            daemon.load(args);
            daemon.start();
        } else if (command.equals("stopd")) {
            args[args.length - 1] = "stop";
            daemon.stop();
        } else if (command.equals("start")) {
            daemon.setAwait(true);
            daemon.load(args);
            daemon.start();
        } else if (command.equals("stop")) {
            daemon.stopServer(args);
        } else if (command.equals("configtest")) {
            daemon.load(args);
            if (null==daemon.getServer()) {
                System.exit(1);
            }
            System.exit(0);
        } else {
            log.warn("Bootstrap: command \"" + command + "\" does not exist.");
        }
    }
```

```
        } catch (Throwable t) {
            // Unwrap the Exception for clearer error reporting
            if (t instanceof InvocationTargetException && t.getCause() != null) {
                t = t.getCause();
            }
            handleThrowable(t);
            t.printStackTrace();
            System.exit(1);
        }
    }
}
```

---

Bootstrap 클래스를 생성하고 init 메서드를 호출해 초기화합니다. 멤버 변수인 daemon에 방금 생성한 Bootstrap 인스턴스를 할당합니다. 이후 main 메서드의 args 배열로 전달받은 명령행 인자 값에 따라 Bootstrap 클래스의 인스턴스인 daemon의 load, start, stop 등의 메서드가 호출됩니다.

만약 예외가 발생하지 않았다면 main 메서드는 종료됩니다.<sup>03</sup> 따라서 프로그램은 종료돼야 합니다. 하지만 수많은 웹 사이트가 톰캣 서블릿 컨테이너로 서비스합니다. 지금까지 지나쳐 온 여러 메서드(init, load, start 등)에 서블릿 컨테이너를 종료하지 않고 HTTP 요청을 받아들일 수 있게 소켓을 열고, 서블릿을 초기화하며 스레드를 관리하는 등 웹 서비스를 동작하게 하는 기능이 있으리라 짐작할 수 있을 것입니다. 먼저 init 메서드를 알아보겠습니다.

---

```
public void init() throws Exception{
    // Set Catalina path
    setCatalinaHome();
```

---

03 예를 들어, start라는 문자열을 인자로 Bootstrap 클래스를 시동했다면, 다시 말해 main 메서드를 호출했다면 Bootstrap 클래스의 인스턴스를 만들고 해당 인스턴스의 init, load, start 메서드가 차례로 불린 후 main 메서드가 끝납니다.

```
setCatalinaBase();
initClassLoaders(); // initClassLoaders 메서드

Thread.currentThread().setContextClassLoader(catalinaLoader);
// 설정한 클래스로더를 현재 클래스로더로 바꾼다.

SecurityClassLoad.securityClassLoad(catalinaLoader);

// Load our startup class and call its process() method
if (log.isDebugEnabled())
    log.debug("Loading startup class");

Class<?> startupClass =
catalinaLoader.loadClass("org.apache.catalina.startup.Catalina");
Object startupInstance = startupClass.newInstance();

// Set the shared extensions class loader
if (log.isDebugEnabled())
    log.debug("Setting startup class properties");

String methodName = "setParentClassLoader";
Class<?> paramTypes[] = new Class[1];
paramTypes[0] = Class.forName("java.lang.ClassLoader");
Object paramValues[] = new Object[1];
paramValues[0] = sharedLoader;
Method method = startupInstance.getClass().getMethod(methodName, paramTypes);
method.invoke(startupInstance, paramValues);
catalinaDaemon = startupInstance;
}
```

---

주목해야 하는 곳은 initClassLoaders 메서드입니다. 이 메서드는 모두 세 개의 클래스로더를 생성합니다. 생성하는 클래스로더는 common, server, shared입니다. common 클래스로더는 루트 클래스로더로, 부모가 없는 최상위 클래스로더입니다. server와 shared 클래스로더는 common 클래스로더를 부모 클래스로더<sup>04</sup>로 가집니다.

common, shared, server 클래스로더는 catalina.properties 파일에 정의된 디렉터리의 jar 파일을 읽어들여, 파일 안에 포함된 클래스를 로딩합니다. 기본으로 제공되는 catalina.properties에는 common 클래스로더만 설치 디렉터리 아래 lib 디렉터리로 지정되어 있고, shard, server 클래스로더는 위치가 지정되어 있지 않습니다. 따라서 설정을 변경하지 않으면 shard나 server 클래스로더에 아무런 영향이 없습니다.<sup>05</sup>

Thread.currentThread().setContextClassLoader(catalinaLoader)는 지금까지 설정한 클래스로더를 현재 클래스로더로 바꿉니다.

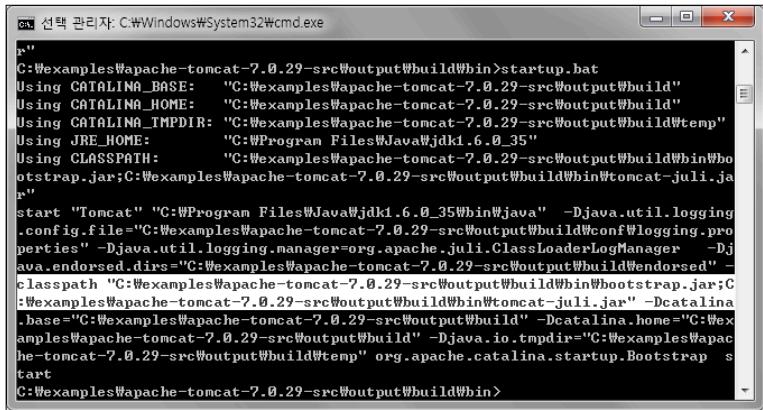
이 부분은 특별히 강조할 만한 가치가 있습니다. 앞서 살펴본 톰캣 구동 스크립트를 다시 한번 살펴보겠습니다.

---

04 shard나 server 클래스로더에 로딩된 클래스는 상위 클래스로더인 common 클래스로더에 로딩된 클래스에 접근할 수 있습니다.

05 다수의 클래스로더를 설정할 수 있게 하는 것은 하위 호환을 위해서입니다. 지난 버전의 톰캣은 설치 디렉터리 아래에 common, server, shared라는 디렉터리가 기본으로 제공되고, 각 클래스로더는 이름이 같은 디렉터리 내의 jar 파일을 로딩합니다. 따라서 톰캣 서블릿 컨테이너에 전역적으로 사용될 라이브러리는 common, server, shared 디렉터리 중 하나에 넣어서 사용할 수 있습니다.

그림 8-1 톰캣 구동 스크립트 화면



```
ca: 선택 관리자: C:\Windows\System32\cmd.exe
r"
C:\examples\apache-tomcat-7.0.29-src\output\build\bin>startup.bat
Using CATALINA_BASE:      "C:\examples\apache-tomcat-7.0.29-src\output\build"
Using CATALINA_HOME:       "C:\examples\apache-tomcat-7.0.29-src\output\build"
Using CATALINA_TMPDIR:     "C:\examples\apache-tomcat-7.0.29-src\output\build\temp"
Using JRE_HOME:            "C:\Program Files\Java\jdk1.6_0_35"
Using CLASSPATH:           "C:\examples\apache-tomcat-7.0.29-src\output\build\bin\bootstrap.jar;C:\examples\apache-tomcat-7.0.29-src\output\build\bin\tomcat-juli.jar"
start "Tomcat" "C:\Program Files\Java\jdk1.6_0_35\bin\java" -Djava.util.logging.config.file="C:\examples\apache-tomcat-7.0.29-src\output\build\conf\logging.properties" -Djava.util.logging.manager=org.apache.juli.ClassLoaderLogManager -Djava.endorsed.dirs="C:\examples\apache-tomcat-7.0.29-src\output\build\vendorsed" -classpath "C:\examples\apache-tomcat-7.0.29-src\output\build\bin\bootstrap.jar;C:\examples\apache-tomcat-7.0.29-src\output\build\bin\tomcat-juli.jar" -Dcatalina.base="C:\examples\apache-tomcat-7.0.29-src\output\build" -Dcatalina.home="C:\examples\apache-tomcat-7.0.29-src\output\build" org.apache.catalina.startup.Bootstrap start
C:\examples\apache-tomcat-7.0.29-src\output\build\bin>
```

클래스 패스에 jar 파일 두 개만 설정된 것을 확인할 수 있습니다. 그렇다면 jar 파일 이외의 라이브러리는 어떻게 사용할 수 있을까요? 그보다 먼저 서블릿 인터페이스는 lib 디렉터리 아래 servlet-api.jar에 포함되어 있습니다. 클래스패스에 포함되어 있지 않다면 서블릿 컨테이너는 서블릿 인터페이스를 어떻게 알고 사용할 수 있을까요?

5장에서 서블릿보다 서블릿 컨테이너가 먼저 작성하는 데도 불구하고 서블릿 컨테이너가 사용자가 작성한 서블릿을 어떻게 호출(어디 있는지, 어떤 메서드를 가졌는지 알아내서)하는지에 대해서 설명했습니다. 간략히 정리하면 다음과 같은 단계로 요약할 수 있겠습니다.

1. 서블릿 컨테이너는 서블릿이 포함된 웹 애플리케이션을 동작시키기 위해 전용 클래스로더를 추가로 생성합니다.
2. 웹 애플리케이션을 위한 클래스로더는 서블릿 명세에 미리 약속된 특정 위치 WEB-INF/lib, WEB-INF/classes에 있는 jar나 class 파일을 로딩합니다.

3. 모든 서블릿은 javax.servlet.Servlet 인터페이스를 구현하므로 서블릿 컨테이너는 사용자가 작성한 다양한 형태의 서블릿을 Servlet.service 메서드로 호출할 수 있습니다.

톰캣과 같은 서블릿 컨테이너는 위와 같은 전략(로딩할 리소스의 위치와 호출할 메서드를 미리 약속)을 컨테이너 자신을 구동시키는 데에도 사용합니다. 왜냐하면, 서블릿 컨테이너에 추가하고 싶은 라이브러리가 있을 때, 매번 JVM을 기동하는 스크립트 파일을 열어 클래스패스 인자를 수정하는 일은 능률적이지 않습니다. 뿐만 아니라 라이브러리에 포함된 모든 jar 파일을 일일이 나열한다면 관리가 힘들어지기 때문입니다. 이런 이유로 톰캣은 JVM을 시작할 수 있는 최소한의 클래스만 bin 디렉터리 아래 bootstrap.jar 파일에 모으고 기동 스크립트 내에서는 jar 파일을 클래스패스로 설정합니다. 그리고 JVM을 구동한 다음 프로그램상에서 실제 서블릿 컨테이너 역할을 할 클래스로더를 새로 생성합니다. 이 클래스로더는 앞서 살펴본 바와 같이 특정 디렉터리(lib) 아래 모든 jar 파일을 등록하게 함으로써, 사용자가 서블릿 컨테이너에 추가하고 싶은 라이브러리를 lib 디렉터리 아래 추가하면 별 다른 설정 없이도 기동 시 클래스패스에 추가되는 효과를 얻었습니다.

이제 실질적인 톰캣 서버 클래스인 카탈리나 클래스를 찾아 로딩을 시작합니다.

---

```
Class<?> startupClass =
    catalinaLoader.loadClass("org.apache.catalina.startup.Catalina");
```

---

이후 Bootstrap 클래스의 load, start, stop 메서드가 불리지만 내부적으로는 catalinaDaemo,n, 즉 Catalina 클래스에 있는 같은 이름의 메서드가 호출됩니다.

카탈리나 클래스를 클래스로더에서 로딩한 후 startup 인스턴스를 만듭니다.

---

```
Class<?> startupClass =
catalinaLoader.loadClass("org.apache.catalina.startup.Catalina");
Object startupInstance = startupClass.newInstance();
```

---

앞서 생성한 startup 인스턴스를 catalinaDaemon으로 지정합니다(253라인).

---

```
/**
 * Start the Catalina daemon.
 */
public void start()
throws Exception {
if( catalinaDaemon==null ) init();

Method method = catalinaDaemon.getClass().getMethod("start", (Class [] )null);
method.invoke(catalinaDaemon, (Object [])null);

}
```

---

이제 catalinaDaemon(앞서 statup 인스턴스로 만든 Catalina 클래스)의 start 메서드를 호출합니다.

---

```
Method method = catalinaDaemon.getClass().getMethod("start", (Class [])
)null;
```

---

이렇게 부팅 과정을 관리하는 클래스(Bootstrap)의 서버 역할을 수행하는 클래스(Catalina)를 분리해 놓으면 추후 서버 클래스 개선이 필요할 때 Bootstrap 클래스의 변경을 최소화하면서 서버 역할 수행 클래스를 교체할 수 있습니다.

### 8.3 생명주기 관리

톰캣 서블릿 컨테이너를 사용한 경험이 있는 웹 프로그래머라면 한 번쯤 config 디렉터리 안에 있는 XML 파일이 어떤 의미가 있으며, 어떤 시점에 어떻게 로딩되는지 의문을 가졌을 것이라 생각합니다. 이런 설정 값을 읽어들이는 과정은 바로 Catalina 클래스의 load 메서드에서 찾아볼 수 있습니다.<sup>06</sup>

---

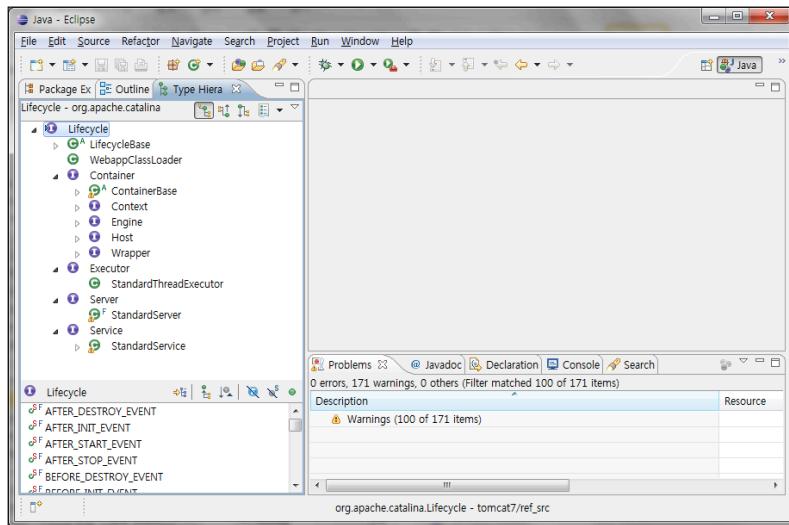
```
/**  
 * Start a new server instance.  
 */  
public void load() {  
  
    long t1 = System.nanoTime();  
  
    initDirs();  
  
    // Before digester - it may be needed  
  
    initNaming();  
  
    // Create and execute our Digester  
    // 설정 파일을 개체화해 접근할 수 있도록 한다.  
    Digester digester = createStartDigester();
```

---

06 앞서 Bootstrap 클래스의 main 메서드에서 init, load, start의 순으로 Catalina 클래스의 메서드를 호출하는 것을 확인한 바 있습니다.

```
InputSource inputSource = null;  
InputStream inputStream = null;  
  
// (이하 생략)...
```

Jaxb 같은 XML - Object 매핑 라이브러리를 사용하는 것이 일반화되기 전부터 톰캣 서블릿 컨테이너는 Digester<sup>07</sup>를 사용해 부팅 시 설정 파일을 객체화해 접근하는 방식을 지원했습니다. 그러므로 server.xml 파일이 로딩되면 XML 태그에 해당하는 객체가 생성됩니다.



톰캣 서블릿 컨테이너는 설정 파일에 지정된 객체들을 시동 시, 자동으로 생성될 뿐만 아니라 유지, 관리, 소멸 등의 생명주기를 관리함으로써 동작 중인 서버를 재시동하지 않고서 기능 변경을 하는 방법을 제공합니다. 이것은 마치 서블릿 관리자

07 Apache commons 프로젝트에 포함되어 있습니다. (<http://commons.apache.org/digester/>)

가 서블릿의 생명주기를 관리하는 것과 일맥상통하며, 서블릿 인터페이스의 역할을 하는 Lifecycle 인터페이스가 있습니다.

서블릿 인터페이스에 init, destroy 메서드가 있어, 해당 서블릿이 초기화되거나 소멸할 때 서블릿 컨테이너가 해당 메서드를 호출했다는 사실을 기억하십시오. 서블릿은 외부의 HTTP 요청이 들어올 때, service 메서드가 호출됐던 것에 반해 Lifecycle 인터페이스는 관리자가 능동적으로 활성화되거나 정지할 수 있습니다. 이를 위해 start/stop 메서드가 추가됐습니다. 또한, 앞서 server.xml이 가진 엘리먼트의 이름을 Lifecycle 인터페이스를 상속한 여러 클래스에서 찾아볼 수 있음을 확인하십시오.

정리하면 사용자가 설정 파일에 정의된 각 XML 엘리먼트는 Digester에 의해 서버 객체로 변경돼 로딩되며, 이런 객체는 Lifecycle 인터페이스를 구현했으므로 프로그래밍적으로 초기화, 시작, 종료, 소멸 등을 컨트롤할 수 있다는 의미가 됩니다.

## 8.4 남은 이야기

### 8.4.1 Shutdown Hook

경험이 짧은 시절에는 누구나 ‘완벽한 사용자<sup>08</sup> 이론’에 따라 프로그램을 작성하기 마련입니다. 하지만 점점 경험을 쌓으면서 세상에는 매뉴얼 대로 행동하는 사용자가 드물다는 것을 체득하게 됩니다.

톰캣 서블릿 컨테이너를 종료하려면 bin/shutdown 스크립트를 사용합니다. 해당 스크립트가 호출되면 Lifecycle 인터페이스를 구현한 객체에 stop 메서드와 destroy 메서드를 차례로 호출해 개별 객체가 수행하던 작업을 종료시키고 리소스를 반환합니다. 하지만 매번 [Control-C]를 눌러 종료하는 사용자도 분명 있을 것

---

08 프로그래머가 의도한 순서대로 사용자가 정확히 입력하는 경우에만 정상적으로 동작하는 프로그램. 입력 값 범위, 타입 체크가 없는 경우를 말합니다.

입니다. 이런 경우를 위해 Shutdown Hook를 추가해 최소한의 리소스 반환 처리를 도모합니다.

다음은 Catalina의 start 메서드의 일부입니다.

---

```
// Register shutdown hook
if (useShutdownHook) {
    if (shutdownHook == null) {
        shutdownHook = new CatalinaShutdownHook();
    }
    Runtime.getRuntime().addShutdownHook(shutdownHook);

    // If JULI is being used, disable JULI's shutdown hook since
    // shutdown hooks run in parallel and log messages may be lost
    // if JULI's hook completes before the CatalinaShutdownHook()
    LogManager logManager = LogManager.getLogManager();
    if (logManager instanceof ClassLoaderLogManager) {
        ((ClassLoaderLogManager) logManager).setUseShutdownHook(
            false);
    }
}

if (await) {
    await();
    stop();
}
```

---

소스코드에서 볼 수 있듯 런타임 객체에 JVM이 종료될 때, 호출되기를 원하는 클래스를 addShutdownHook 메서드를 사용해 파라미터로 등록합니다.

JVM 종료 시 호출되는 CatalinaShutdownHook 클래스는 다음과 같습니다.

---

```
protected class CatalinaShutdownHook extends Thread {

    @Override
    public void run() {
        try {
            if (getServer() != null) {
                Catalina.this.stop();
            }
        } catch (Throwable ex) {
            ExceptionUtils.handleThrowable(ex);
            log.error(sm.getString("catalina.shutdownHookFail"), ex);
        } finally {
            // If JULI is used, shut JULI down *after* the server shuts down
            // so log messages aren't lost
            LogManager logManager = LogManager.getLogManager();
            if (logManager instanceof ClassLoaderLogManager) {
                ((ClassLoaderLogManager) logManager).shutdown();
            }
        }
    }
}
```

---

Thread를 상속하고 stop 메서드를 호출해 정상 종료할 때와 동일한 프로세스를 거쳐 종료될 수 있게 설정했습니다.

#### 8.4.2 System.out과 catalina.out

System.out.println 메서드는 동기적으로 동작해야 할 합리적인 이유가 있습니다. 만약 비동기적으로 동작하면 두 개 이상의 스레드에서 동시에 System.out에 문자열을 출력할 때 두 문자열이 섞일 수 있기 때문입니다.

이렇게 System.out.println 메서드가 동기적으로 동작해 웃지 못할 사건이 벌어지기도 했습니다. 예를 들면, 콘솔 화면에 올라가는 로깅 메시지를 살펴보다 마우스 클릭을 잘못해 선택되면 System.out.println이 블록으로 처리돼 전체 시스템이 멈춘 것과 같은 효과가 나타나곤 했습니다.<sup>09</sup> 이런 일이 의외로 잦았는지 톰캣 서블릿 컨테이너는 System.out과 System.err를 재할당해 직접 관리합니다. 이런 처리로 서블릿 작성자가 System.out에 쓴 로깅 메시지가 log/catalina.out 파일에 추가되는 것입니다. 다음은 initStreams의 내용입니다.

---

```
protected void initStreams() {  
    // Replace System.out and System.err with a custom PrintStream  
    System.setOut(new SystemLogHandler(System.out));  
    System.setErr(new SystemLogHandler(System.err));  
}
```

---

## 8.5 더 생각해 볼 문제

- 최신의 톰캣 서블릿 컨테이너 코드를 source repository에서 체크아웃해 빌드해 보십시오.

---

09 스레드가 System.out.println을 호출하려고 하면 대기 상태에 빠지게 되므로 결국은 전체 스레드가 언젠가는 모두 대기 상태에 빠지게 됩니다. 물론 요즘은 직접 System.out을 사용하기보다는 Logging 라이브러리를 사용하지만 2000년 전후에는 쓸만한 Logging 라이브러리가 없었을 뿐 아니라 Logging을 사용할 의식 자체가 없었던 시기였습니다.

2. Java 1.5부터 원격 디버깅을 지원합니다. catalina.bat에서 JPDA 부분을 참조해 원격 디버깅이 가능하게 설정해 보고, 어떤 점이 일반적인 디버거 사용 환경과 다른지 살펴보십시오.
3. 다른 서버 객체(Server, Service 등)는 Lifecycle 인터페이스를 구현하지만 Catalina는 아무런 인터페이스도 구현하지 않았습니다. Lifecycle 인터페이스를 사용해야 하는지, 사용하지 않는다면 왜 그런지 생각해 보십시오.

# 3부

## 남은 주제들

지금까지 우리는 서블릿을 동작시키는 기능을 위주로 서블릿 컨테이너를 살펴보았습니다. 이 외에도 서블릿 컨테이너는 서블릿 명세에는 없지만, 웹 서비스를 위해 필요한 부가 기능을 몇 가지 제공할 수 있습니다. 다음 장에서는 이런 기능 중 대표적인 Comet에 대해 살펴보겠습니다.

**9장** Comet – HTTP 알림

**10장** 남은 이야기들

## 9 | Comet - HTTP 알림

처음부터 인간이 아닌 줄 알았어. 인간은 그렇게 인간적이지 않거든.

- ‘에어리언 4’ 중에서

웹 서비스가 대중화되면서 기존 시스템을 웹 기반으로 바꾸는 작업이 많이 행해졌습니다. 이런 전환 작업에서 어려운 점은 기존의 설치형 클라이언트가 제공하던 기능을 웹 서버와 범용 웹 브라우저를 이용해 구현해야 한다는 사실입니다. 이미 클라이언트 서버 환경인 레거시 시스템에 익숙한 사용자들에게 만족스러운 웹 환경을 제공하기란 쉽지 않았습니다. 그중에서 빠지지 않고 제기되던 불만 중 하나는 “왜 웹 환경에서는 서버와 연결된 사용자를 알 수 없는가?”였습니다.

사실 이것은 HTTP 프로토콜 자체 특성에 기인합니다. HTTP 프로토콜은 원래 문서를 요청하고 문서의 내용을 응답하는 목적으로 구상됐습니다. 따라서 클라이언트가 기존에 요청을 몇 번 했든 특정 문서에 대한 요청은 언제나 해당 문서의 내용이 반환돼야 합니다. 또한, 같은 문서에 대한 요청은 클라이언트에 상관없이 응답 내용이 같아야 합니다.

전송 이력과 전송 대상에 대한 정보는 응답 내용에 영향을 미치지 않으므로, 이를 유지할 이유가 없었습니다. 따라서 HTTP 프로토콜은 웹 서버가 클라이언트를 특징지어 구별하는 방법을 제공하지 않습니다.<sup>01</sup> 이것을 HTTP 프로토콜을 이용하는 웹의 한계 혹은 웹으로는 구현 불가 등으로 이야기되곤 했습니다.

앞서 제기된 지금 서버와 연결된 사용자가 누구인지를 알아내는 문제는 자연스럽게 특정 사건이 발생했을 때 해당 사용자에게 실시간으로 알림을 전달하고자 하는

---

01 이러한 단순성이 오히려 다수의 사용자를 동시에 처리할 수 있는 대용량 시스템에 적합했고 웹 서비스가 이토록 성공적으로 확대될 수 있게 한 근본적인 원동력이 되었다 하겠습니다.

고민으로 이어졌습니다. 일반적으로 알림 기반의 서비스<sup>02</sup><sup>02</sup>는 전용 프로그램을 설치한 후에 사용할 수 있었는데, 만약 범용 웹 브라우저상에서 실시간으로 알림을 전송할 수 있다면 사용자 접근성이 더 높아질 것입니다. 이런 이유로 웹 브라우저가 이해할 수 있는 HTTP 프로토콜을 사용하면서도 상태 정보를 유지하는 여러 방법이 꾸준히 시도됐습니다. 이 장에서는 HTTP 프로토콜을 이용해 웹 브라우저에서 알림을 전송하는 방법이 있을지, 또 해당 기능을 사용하려면 서블릿 컨테이너에 어떤 기능을 추가해야 하는지에 대해 오픈 소스 서블릿 컨테이너인 Neurasthenia 코드를 사용해 알아보겠습니다.

## 9.1 단순한 시도 - 폴링

웹 브라우저를 사용해 사용자 간 채팅 기능을 제공하는 가장 단순한 방법으로 요청 폴링request polling, 즉 주기적으로 화면을 새로고침 처리를 생각할 수 있습니다. 물론 메시지창 깜박임을 피하고자 Ajax나 숨겨진 아이프레임iframe 등을 사용해 요청을 보내고, 기존 내용에 추가된 메시지 내용만을 Javascript 등을 이용해 메시지 창에 추가하는 처리는 필요합니다.

이 폴링 방식의 문제점은 사용자가 채팅 창을 열어 놓으면 새로운 메시지 추가 여부를 알기 위해 주기적으로 서버로 요청이 전달된다는 점에 있습니다. 특히 채팅처럼 전송 지연에 민감한 경우 그 주기를 길게 가져가기 어려워 다수의 사용자를 대상으로 이런 방식을 사용하기는 어렵습니다.

또한, 사용자가 채팅 창을 열어 대화하지 않더라도 채팅방 초대 같은 시스템 메시지의 발생 여부를 알기 위해 새로고침이 발생하므로, 로그인된 상태의 사용자 숫자만 큼 부하가 가중될 것임을 짐작할 수 있습니다. 다시 말해, 로그인한 사용자는 채팅방 입장과 관계없이 주기적으로 [F5] 키를 사용해 새로고침을 시도하는 것과 같은

---

02 모바일 메신저나 다중 사용자 채팅 등이 있습니다.

서버 부하가 발생합니다. 이런 이유로 어느 정도 규모가 큰 서비스에서 폴링 방식을 사용하는 경우는 찾아보기 어렵습니다.

## 9.2 생각의 전환 - 스트림 방식

앞서 살펴본 폴링 방법의 문제인 과다한 요청을 제거하기 위해 스트림 방식이 제안되었습니다. 기본적인 아이디어는 다음과 같습니다.

HTTP 프로토콜이 클라이언트에서 요청을 보내야 서버가 응답할 수 있는 구조라면, 다시 말해 서버가 클라이언트로 요청을 보낼 수 없고 응답만 보낼 수 있는 구조라면 서버가 응답을 주도적으로 보내는 전용 연결을 하나 더 생성/유지하는 것입니다. 이 방식은 주의 깊게 살펴볼 부분이 세 곳입니다.

먼저 클라이언트가 서버의 알림을 받는 TCP 연결을 추가로 생성하고 HTTP 요청을 보냅니다. 일반적인 HTTP 요청/응답 구조라면, 서버는 즉시 요청에 대한 응답을 보내 HTTP 요청/응답 통신을 완료해야 합니다. 하지만 이 방식은 서버가 즉각적으로 응답을 보내지 않는다는 점이 첫 번째 핵심입니다.

서버는 즉각적인 응답을 유보하고 TCP 연결을 맺은 상태를 유지합니다. 이후 서버가 클라이언트로 메시지를 전달할 때마다 미리 생성해 둔 연결에 HTTP 응답을 실어 전달합니다.

여기서 의문이 생깁니다. HTTP 응답은 요청/응답쌍이 완성돼 다시 요청이 들어와야 추가 응답이 가능한데, 어떤 방법으로 응답만을 지속해서 전달할 수 있을까요? 그 대답은 2장에서 살펴본 HTTP 메시지 구조 중 청크 인코딩 방식에서 찾아볼 수 있습니다.

청크 인코딩 방식은 transfer-encoding 헤더 값을 chunked로 설정한 후, 청크라는 메시지 덩어리의 집합으로 HTTP 메시지를 구성하는 방식입니다. 이런 방식은

메시지의 크기가 유동적일 때 사용하며, 응답 메시지의 끝에 크기가 0인 청크를 표시해 더 이상의 청크가 없음을 나타냅니다. 이 방식을 역으로 생각하면, 청크 인코딩으로 전달 중인 응답은 크기가 0인 청크가 전달되지 않는 한 전체 응답이 종료된 것이 아니란 의미가 됩니다.

스트림 방식의 두 번째 강조할 만한 점이 바로 이 청크 인코딩 방식을 사용해 끝나지 않는 HTTP 응답을 사용한다는 것입니다. 서버의 응답 전용 연결은 클라이언트에서 전달된 요청에 대한 응답을 청크 인코딩으로 설정한 후 응답을 종료시키지 않고(크기가 0인 청크를 보내지 않음), 이후 서버가 클라이언트로 특정 메시지를 전달하고자 할 때 해당 메시지의 내용을 청크로 만들어 보냅니다. 다행히 대다수 웹 브라우저에서는 청크 인코딩 타입의 응답 조각을 그때마다 해석하는 기능을 제공합니다. 따라서 이벤트가 발생할 때마다 전달되는 청크 내용을 분석해 서버가 전달한 실시간 메시지를 파악할 수 있습니다.

앞서 언급한 스트림 방식에서 살펴볼 가치가 있는 세 가지 중 마지막은 비동기 처리와 관련 있습니다. 지금까지 살펴본 바에 따라 스트림 방식을 구현하려면 특정한 HTTP 요청을 받았을 때 응답을 유보하는 기능이 필요합니다. 기본적으로 서블릿은 동기적이므로 하나의 워커 스레드가 할당돼 동작하는 것이 일반적입니다. 그러므로 서블릿 내에서 스트림 방식을 처리하면 서블릿을 동작시키는 워커 스레드를 블로킹하는 방법이 있습니다.

이러한 방식은 한정된 자원인 스레드 수가 제한될 수밖에 없어 확장성에 문제가 됩니다. 이를 해결하려면 TCP 연결 자체를 관리해야 합니다.

서블릿 컨테이너 레벨에서는 외부에서 들어온 TCP 연결을 관리하다가 응답 전용이라고 판단되는 연결에는 워커 스레드를 할당하지 않는 방법을 사용할 수 있습니다. 이후 특정 연결에 청크 메시지를 전달할 때 관리하던 소켓 연결을 찾아내 직접 청크 메시지를 추가한다면 불필요한 스레드 블로킹을 피할 수 있습니다.

## 9.3 Neurasthenia - Comet 지원 서블릿 컨테이너

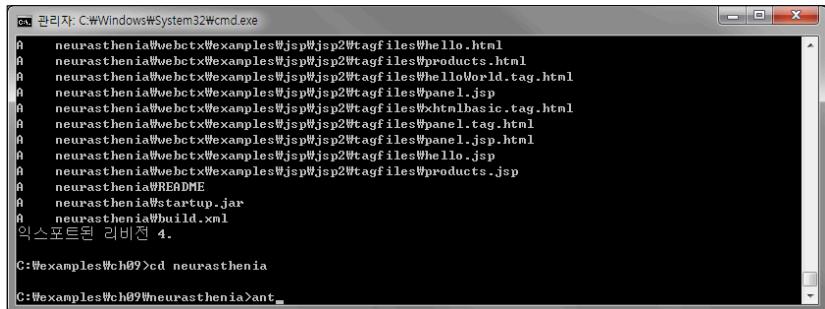
앞서 설명한 스트림 방식의 Comet 처리를 지원하는 서블릿 컨테이너 (neurasthenia 프로젝트<sup>03</sup>) 구현을 살펴보고 어떻게 동작하는지 설명하겠습니다.

먼저 Neurasthenia 프로젝트의 소스코드는 다음과 같이 접근할 수 있습니다.<sup>04</sup>



```
cmd 관리자: C:\Windows\System32\cmd.exe
C:\examples\ch09>svn export --username anonsvn https://dev.naver.com/svn/neurasthenia/trunk neurasthenia --password anonsvn
```

이제 생성된 neurasthenia 디렉터리 안으로 들어가 ant 명령으로 빌드합니다.



```
cmd 관리자: C:\Windows\System32\cmd.exe
A neurasthenia\WebContext\Weexamples\jsp\tagfiles\Hello.html
A neurasthenia\WebContext\Weexamples\jsp\jsps2>tagfiles\products.html
A neurasthenia\WebContext\Weexamples\jsp\jsps2>tagfiles\HelloWorld.tag.html
A neurasthenia\WebContext\Weexamples\jsp\jsps2>tagfiles\panel.jsp
A neurasthenia\WebContext\Weexamples\jsp\jsps2>tagfiles\htmlbasic.tag.html
A neurasthenia\WebContext\Weexamples\jsp\jsps2>tagfiles\panel.tag.html
A neurasthenia\WebContext\Weexamples\jsp\jsps2>tagfiles\Hello.jsp
A neurasthenia\WebContext\Weexamples\jsp\jsps2>tagfiles\products.jsp
A neurasthenia\WEBADM
A neurasthenia\startupt.jar
A neurasthenia\build.xml
익스포트된 리비전 4.

C:\examples\ch09>cd neurasthenia
C:\examples\ch09\neurasthenia>ant
```

boot.bat(유닉스 계열에서는 boot.sh)를 실행해 시작합니다.

03 <http://dev.naver.com/projects/neurasthenia/>, neurasthenia는 신경쇠약이란 뜻입니다. 이 이름은 외부의 변경에 항상 신경써야 하는 현대인의 삶을 대표적으로 보여 주는 기능이 실시간 알림이라고 생각해 실시간 알림을 지원하는 서블릿 컨테이너의 이름으로 선택했습니다.

04 Subversion은 '<http://subversion.apache.org/>'에서 배포합니다.

```

C:\Windows\System32\cmd.exe 관리자: C:#Windows#System32\cmd.exe
ava uses unchecked or unsafe operations.
[javac] Note: Recompile with -Xlint:unchecked for details.
[javac] C:\examples\ch09\neurasthenia\build.xml:31: warning: 'includeantruntime' was not set, defaulting to build.sysclasspath=last; set to false for repeatable builds
[javac] Compiling 110 source files to C:\examples\ch09\neurasthenia\classes_jasper
[javac] Note: Some input files use or override a deprecated API.
[javac] Note: Recompile with -Xlint:deprecation for details.
[javac] Note: Some input files use unchecked or unsafe operations.
[javac] Note: Recompile with -Xlint:unchecked for details.

jar:
[jar] Building jar: C:\examples\ch09\neurasthenia\lib\neurasthenia.jar
[jar] Building jar: C:\examples\ch09\neurasthenia\lib\neurasthenia_jasper.jar
[jar] Building jar: C:\examples\ch09\neurasthenia\startup.jar

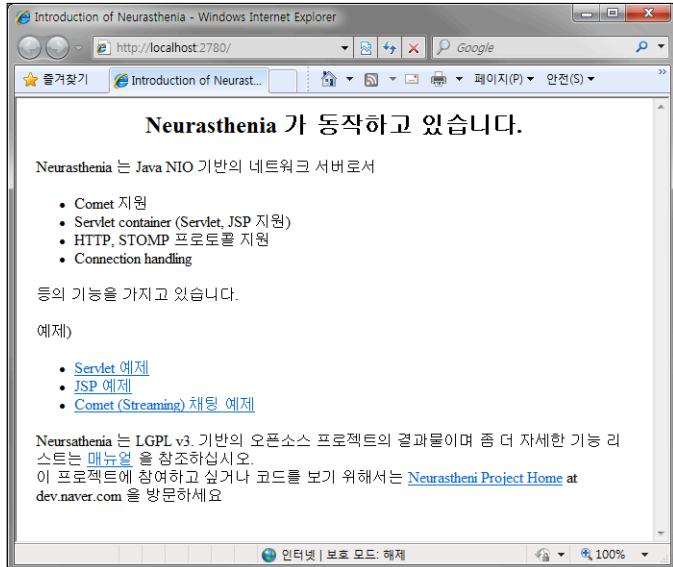
BUILD SUCCESSFUL
Total time: 2 seconds

C:\examples\ch09\neurasthenia>boot.bat

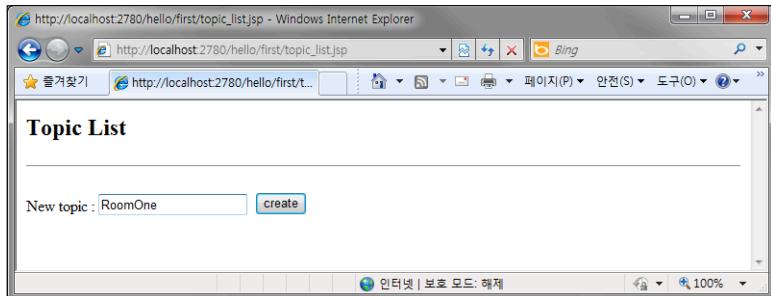
```

브라우저에서 'http://localhost:2780'을 호출해 다음과 같은 화면이 나오면 정상적으로 부팅된 것입니다.

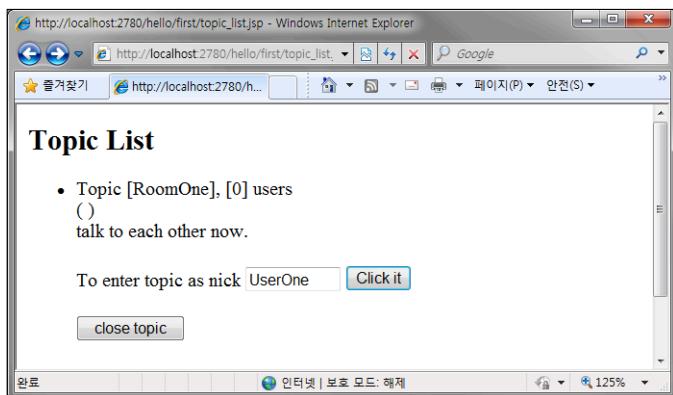
그림 9-1 브라우저로 'http://localhost:2780'에 접속한 결과



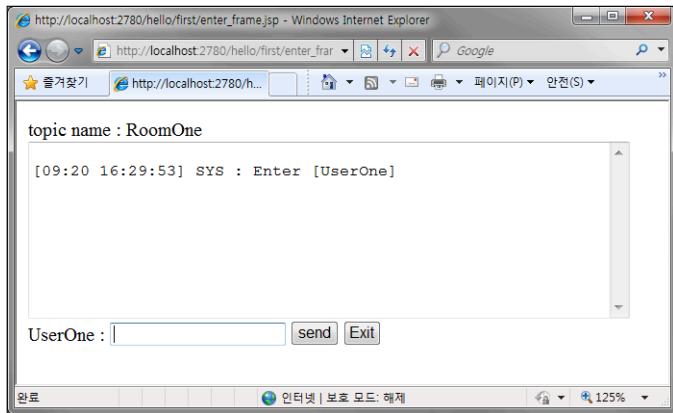
[Comet (Streamming) 채팅 예제] → [start(Chat demo 1 - use frame, streaming)]를 차례로 클릭하면 채팅룸을 만들 수 있습니다. 브라우저를 두 개 띄운 다음 채팅 예제를 확인해 보십시오. 먼저 RoomOne이라는 이름의 채팅룸을 만듭니다.



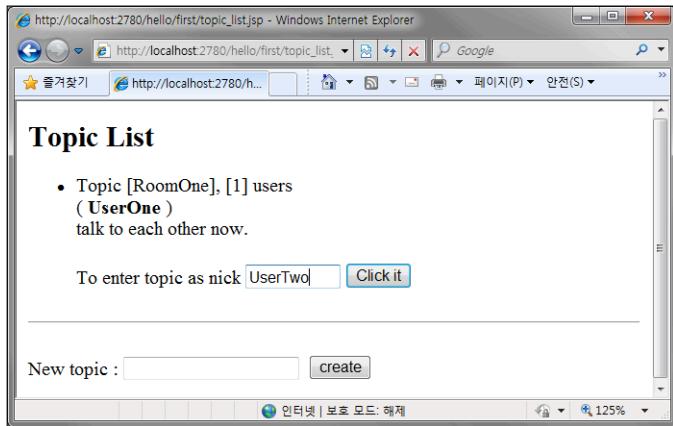
그리고 UserOne이라는 사용자를 만들어 입장합니다.



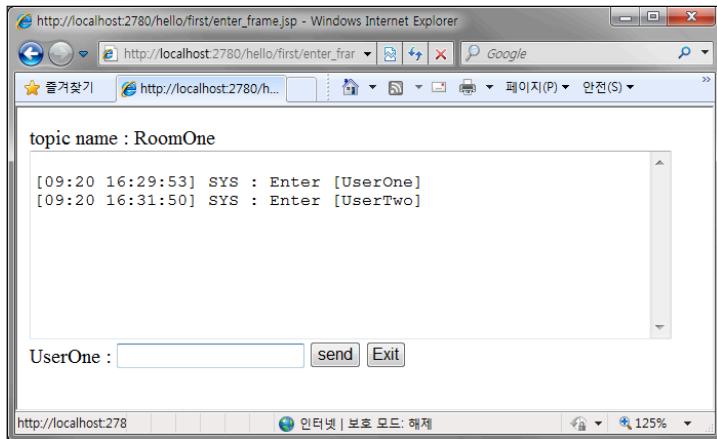
채팅방에 입장하면 입장했다는 안내와 함께 시작됩니다.



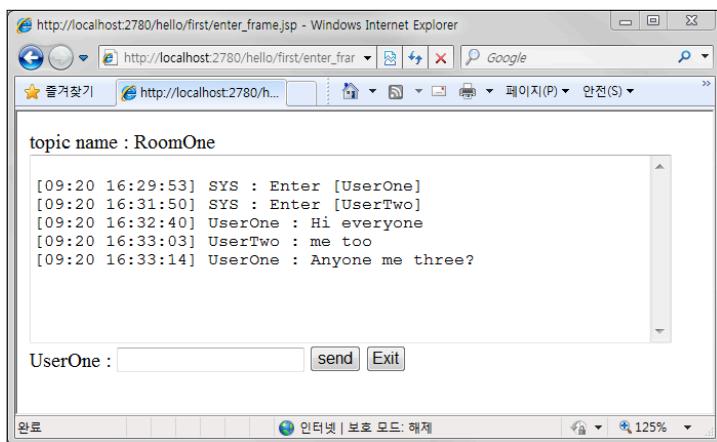
브라우저를 하나 더 띄워서 채팅 예제 페이지로 접근하면 다음과 같이 RoomOne이라는 채팅방을 확인할 수 있으며, 이미 채팅방에 UserOne이 있음을 볼 수 있습니다.

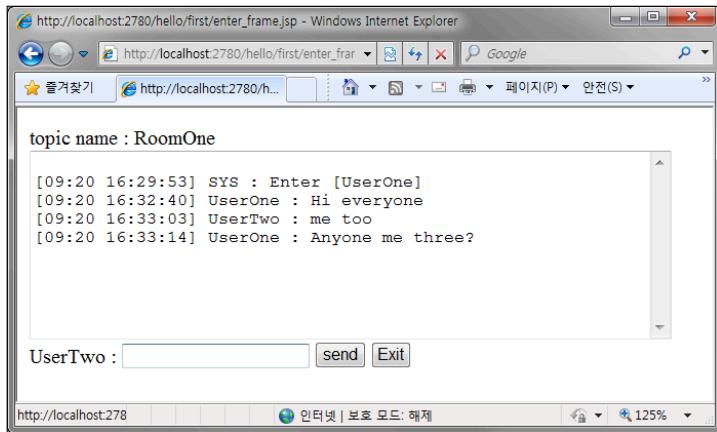


UserTwo라는 이름으로 RoomOne 채팅방에 입장하면 다음과 같이 두 명이 입장한 채팅방 메시지를 볼 수 있습니다.



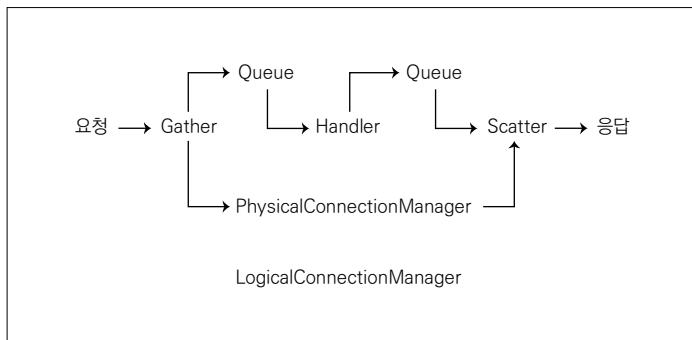
브라우저에 채팅 메시지가 즉각 반영되는 것을 확인할 수 있습니다.





이런 기능이 어떻게 구현됐는지 살펴보겠습니다. 다음은 메시지 흐름의 관점에서 본 neurasthenia의 대략적인 구성입니다.

그림 9-2 neurasthenia의 구성



크게 보아 Gather, Handler, Scatter로 나눠져 있습니다. Gather는 네트워크 통신에 특화된 영역을 처리합니다. 다시 말해, 네트워크 연결을 받아들이고 프로토콜에 따라 요청을 분석한 후 메시지 큐에 넣습니다. 그러면 Handler가 메시지를 받아서 처리합니다. 이때 네트워크 연결 정보를 PhysicalConnectionManager에 등록해 추후 Scatter가 응답을 반환할 때 사용할 수 있게 합니다.

Handler는 Gather가 추가한 메시지 큐에서 메시지를 꺼내 실질적인 처리를 합니다. 이후 응답을 되돌려 줄 필요가 있으면, 전달할 내용을 메시지로 만들어 Scatter가 대기 중인 큐에 넣습니다. 이렇게 Gather와 Handler를 분리함으로써 서로 다른 프로토콜로 들어온 요청을 같은 업무로직으로 수행할 수 있습니다.

Scatter는 대기 중인 큐에 메시지가 도착하면 PhysicalConnectionManager를 이용해 응답할 TCP 연결을 찾아 메시지를 반환합니다.

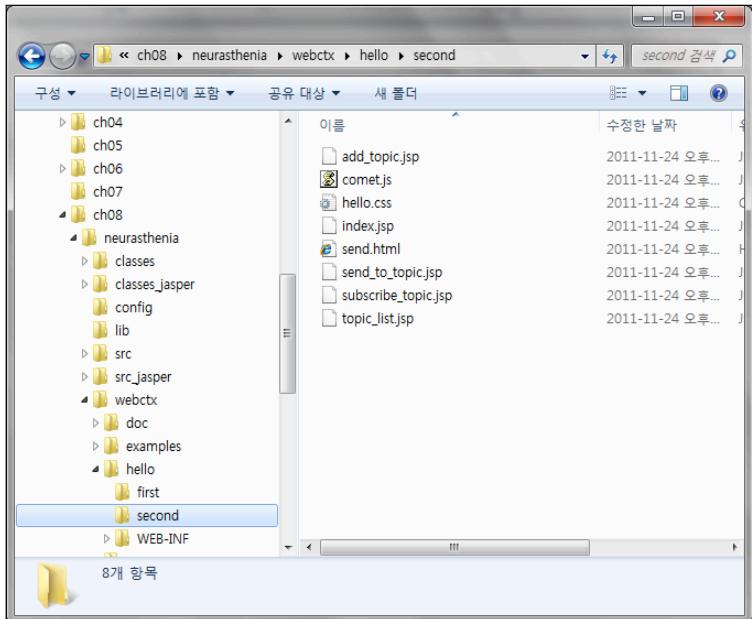
이렇게 요청을 받아들이고(Gather), 논리적인 처리를 하며(Handler), 응답하는(Scatter) 부분을 비동기로 동작할 수 있게 분리하고, TCP 연결 정보를 총괄적으로 관리하는 부분 (Physical/LogicalConnectionManager)을 둠으로써 스트림 방식의 Comet을 지원하는 것입니다.

## 9.4 두 개의 연결

알림을 전송받기 위해 전용 연결을 하나 더 유지하는 것이 스트림 방식입니다. 실제로 위 예제에서 웹 브라우저를 사용해 어떻게 연결을 생성, 유지하는지 알아보겠습니다.

Neurasthenia가 설치된 디렉터리 아래 있는 webctx 디렉터리는 웹 애플리케이션이 설치되어 있습니다. 앞에서 살펴본 채팅 예제는 hello 디렉터리 아래에서 찾을 수 있습니다.

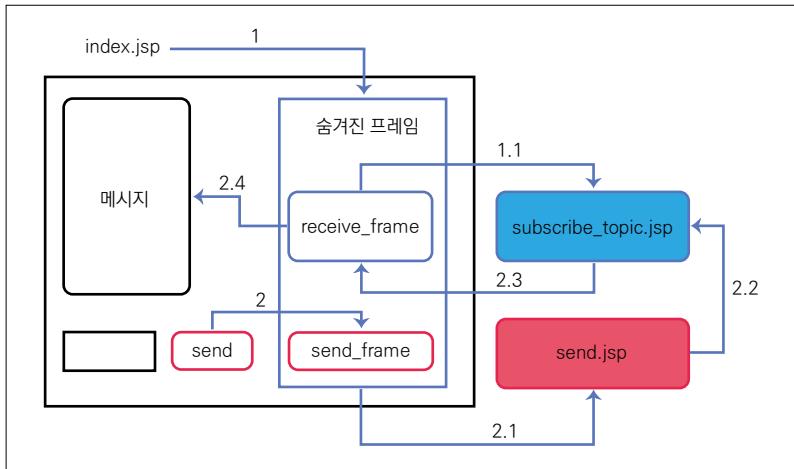
그림 9-3 Neurasthenia 디렉터리 구조



채팅방에 진입한 후 발생하는 이벤트의 흐름을 그림으로 나타내면 그림 9-4와 같습니다.

먼저 index.jsp가 호출되면 로딩과 동시에 숨겨진 프레임 receive\_frame과 send\_frame이 생성됩니다. 생성과 동시에 receive\_frame은 subscribe\_topic.jsp를 호출하며(1), 이 subscribe\_topic.jsp를 호출한 연결이 응답 전용으로 사용됩니다(1.1). 따라서 응답을 바로 반환하지 않고 대기 상태에 있습니다. subscribe\_topic.jsp는 현재 사용자를 topic에 등록해 추후 해당 topic에 메시지가 추가되면 알림을 전달할 대상에 추가합니다.

그림 9-4 채팅방에 진한 후 발생하는 이벤트 흐름도



이후 사용자가 특정 메시지를 보내면 숨겨진 send\_frame에 해당 처리를 위임합니다. 이것은 전체 창이 새로고침되서 깜박거리지 않게 하려는 것입니다. send\_frame은 send.jsp에 추가된 메시지를 전달합니다(2.1).

send.jsp는 topic에 등록한(subscribe\_topic.jsp를 호출한) 사용자에게 메시지를 보냅니다(2.2). 앞서 연결한 subscribe\_topic.jsp 응답 일부로 receive\_frame에 청크 메시지를 보냅니다(2.3). 청크 메시지는 스크립트 코드로, 브라우저가 스크립트 코드를 즉시 해석해 index.jsp 메시지 창에 추가된 메시지를 덧붙입니다(2.4).

이러한 구성에서 사용자를 topic에 등록하고 해당 topic에 추가된 알림을 전송하는 핵심은 subscribe\_topic.jsp가 내부적으로 호출하는 TopicManager입니다. 이 TopicManager는 각 알림을 받을 사용자를 추가/제거하며 이벤트를 전달합니다.

다음은 TopicManager에서 사용자를 등록하는 subscribe 메서드입니다.

---

```
public boolean subscribeTo(
    final int topicId, String userName,
    PhysicalConnectionKey physicalConnectionKey,
    ConnectionEventHandler connectionEventHandler){
    boolean subscribeFlag = false;
    Topic topic = topicMap.get(new Integer(topicId));
    if(topic == null){
        subscribeFlag = false;
    }else{
        LogicalConnectionManager logicalConnectionManager =
            server.getLogicalConnectionManager();
        List<PhysicalConnectionKey> duplicatedPhysicalConnectionKeyList =
            logicalConnectionManager.addLogicalConnection(userName,
                LogicalConnection.CONNECTION_TYPE_COMET_SUBSCRIBED,
                physicalConnectionKey, connectionEventHandler
            );
        // 같은 connectionType 중 같은 category이고 중복된 것은 끊어버린다.
        for(PhysicalConnectionKey duplicatedPhysicalConnectionKey :
            duplicatedPhysicalConnectionKeyList){
            logicalConnectionManager.removeLogicalConnectionByPhysical
                ConnectionKey(duplicatedPhysicalConnectionKey);
        }
        topic.addUser(userName);
        subscribeFlag = true;
    }
    return subscribeFlag;
}
```

---

매개변수로 topicId와 사용자명 외에 물리적인 연결 정보를 나타내는 Physical ConnectionKey와 이벤트 리스너로 ConnectionEventHandler를 받는 것을 확인할 수 있습니다. PhysicalConnectionKey는 특정 사용자에게 알림을 전송하기 위해 어떤 물리적 대상에 청크 메시지를 전달해야 하는지를 판단하는 기준입니다. 또한, ConnectionEventHandler를 등록함으로써 추후 물리적인 연결이 끊어졌을 때처럼 연결 상태의 변동을 감지했을 때 대응할 방법을 제공합니다.

주요한 처리 내용은 LogicalConnectionManager에 어떤 사용자가 Comet 타입의 연결(종료되지 않는 응답 전용 연결)을 어떤 물리 연결에 대해 맺고 있는지 등록하는 것입니다. 이렇게 알림을 전송하는 메타 정보를 LogicalConnectionManager에 등록하면 sendMessageToTopic 메서드로 알림을 전송할 수 있습니다.

다음은 sendMessageToTopic 메서드의 일부입니다.

---

```
public void sendMessageToTopic(String msg, int topicId){  
    byte[] dataBytes = StringUtil.getUTF8Bytes(msg);  
    String hexSize = String.format("%x", dataBytes.length);  
    String hexSizeStr = hexSize + Constants.CRLFStr;  
    byte[] hexSizeBytes = StringUtil.getUTF8Bytes(hexSizeStr);  
    List<byte[]> bytesList = new ArrayList<byte[]>();  
    bytesList.add(hexSizeBytes);  
    bytesList.add(dataBytes);  
    bytesList.add(Constants.CRLF);  
    byte[] chunkedBytes = StringUtil.copyBytes(bytesList);
```

---

(이하 생략)

---

매개변수로 알림 내용과 알림을 보낼 topic의 ID를 받은 후, String 형식의 내용을 청크 인코딩 방식에서 사용되는 16진수 형식의 크기와 byte 배열, 그리고 캐리지 리턴과 라인 피드 형태로 변경한 후 네트워크상으로 전송하기 위해 byte 배열로 만듭니다.

이후 topic에 등록된 사용자 정보를 가져옵니다. 이 정보에는 해당 사용자가 현재 연결된 물리적인 연결에 접근하는 방법을 포함하고 있습니다. 다음은 관련 소스코드입니다.

---

```
Topic topic = topicMap.get(new Integer(topicId));
topic.addMessage(msg);
List<String> userList = topic.getUserList();
// topic에 등록된 사용자 정보를 가져온다.
Iterator<String> userIter = userList.iterator();
ConfigManager configManager = ConfigManager.getInstance();
BlockingQueue<Message> messageQueue =
    server.getMessageQueue(
        configManager.getHandlerInfoList().get(0).getMessageQueueId());
LogicalConnectionManager logicalConnectionManager =
    server.getLogicalConnectionManager()
```

---

이제 전송할 메시지에 도착 지점, 다시 말해 topic에 등록한 사용자의 물리적인 연결을 의미하는 PhysicalConnectionKey를 추가하고 Scatter를 대기 중인 메시지 큐에 추가합니다. 이후 Scatter는 해당 메시지를 클라이언트로 전달하면 알림 전송이 완료됩니다.

---

```
LogicalConnectionManager logicalConnectionManager =
server.getLogicalConnectionManager();
```

---

```

while(userIter.hasNext()){
    String userName = userIter.next();
    LogicalConnection logicalConnection =
        logicalConnectionManager.getLogicalConnection(userName,
        LogicalConnection.CONNECTION_TYPE_COMET_SUBSCRIBED);
    if(logicalConnection != null){
        PhysicalConnectionKey physicalConnectionKey =
            logicalConnection.getPhysicalConnectionKey();

        // PhysicalConnectionKey를 추가
        Message message = new MessageImpl(getTopicName(topicId)+"_"+userName,
            Message.MSG_TYPE_BYPASS, physicalConnectionKey, null, chunkedBytes);
        try {
            messageQueue.put(message); // Scatter를 대기 중인 메시지 큐에 추가
        } catch (InterruptedException e) {
        }
    }
}

```

---

## 9.5 더 생각해 볼 문제

1. 지금의 Neurasthenia는 하나의 물리적인 기계에서 동작하도록 구현되어 있습니다. 이말은 하나의 기계에서 사용 가능한 소켓 수 이상의 동시 사용자는 서비스할 수 없다는 의미이며 이런 제한을 뛰어넘기 위해 어떤 모듈을 수정하면 될지 생각하여 보십시오.
2. Comet 구현에는 이 장에서 살펴본 두 방식(폴링 및 스트림 방식) 외에 롱 폴링 방식이 있습니다. 롱 폴링 방식에 대해 알아보십시오.
3. 이런 방식의 장/단점에 대해 생각해 보십시오.

# 10 | 남은 이야기들

말할 수 있는 것은 분명하게 말해질 수 있고, 말할 수 없는 것에 대해서는 침묵해야 한다.

- 루드비히 비트겐슈타인(『논리 철학 논고』 중에서)

## 10.1 자존심과 두려움

웹 서비스를 구축하면서 발생하는 문제 중에서 서비스를 개발/유지하는 측에서 도저히 알 수 없다고 판단하면, 결국 웹 애플리케이션 서버 제조사로 문의가 들어옵니다. 각종 특이한 증상을 호소하는 요청을 처리하다 보면 세상에는 참으로 별난 일도 많다는 느낌이 절로 듭니다.

어느 금융권에서 프로젝트가 진행되던 중 웹 애플리케이션 서버가 처음에는 잘 동작하는 듯싶더니 언제부턴가 자꾸 아무런 징후 없이 JVM 자체가 종료되는 현상이 발생했습니다. 이렇다 보니 해당 프로젝트 수행을 담당하는 조직은 발칵 뒤집어졌습니다. 사용하던 웹 애플리케이션 서버가 국산 제품이다 보니 ‘국산은 이래서’라는 이야기가 나왔고 ‘이래서 안 돼’는 물건을 개발한 회사에서는 비상사태로 돌입, 원인 규명에 들어갔습니다. 웹 애플리케이션 서버 코드를 쥐 잡듯 뒤집다 못해 실제 해당 프로젝트 소스코드를 요청해 살펴보던 중 참으로 믿기 어려운 장면이 발견되었습니다.

누군가 JSP 코드 끝에 `System.exit(0);`라고 곱게 넣어 둔 것이었습니다. 이렇다 보니 멀쩡히 작동하다가도 해당 JSP가 호출되기만 하면 바로 JVM이 종료되는 치명적인 결과가 나왔던 것입니다.

어쩌면 그런 코드를 작성한 것이 이해되기도 하는 것은 대다수의 C 프로그래밍 책의 예제에서 `exit(0);` 함수를 호출하는 것으로 마치고, 2000년 전후에 출판된 몇몇 서적에서는 `System.exit` 메서드를 호출하는 것을 예제 코드마다 넣기도 했기 때문

입니다. 문제는 JSP란 물건이 스탠드 얼론 프로그램으로 자기 수행이 끝나면 모두 종료되는 게 아니라 웹 애플리케이션 서버 안에서 그 생명주기가 조절되는 것인데, JSP가 하나의 요청을 처리하고는 JVM 자체를 종료시켜 웹 애플리케이션 서버 자체가 종료되는 대형 사고로 이어졌던 것입니다.

결국, 이 사건의 여파로 내장된 JSP 컴파일러를 수정해 JSP 소스코드 내에 System.exit가 포함된 경우 해당 부분을 무시하고 서블릿으로 생성하게 하는 방어 코드가 제품에 추가되었습니다.

돌이켜 보면 해당 버그를 만든 사람은 자신이 작성한 JSP가 호출되면 시스템이 종료된다는 사실을 가장 먼저 알았을 텐데 타인에 의해 밝혀지기 전까지 힘구했다는 사실이 놀랍습니다. 인간이 자신의 체면이 깨일까 두려워하는 마음이 정말 무섭구나 하고 느꼈습니다.

## 10.2 개발자와 프로그래머

언제부터인가 ‘개발자’라는 말이 자연스레 프로그래머가 자신을 지칭하는 말이 된 것 같습니다. 일단 ‘놈 자(者)’가 붙은 직종치고 대우받는 게 드물다고 한탄한 어느 기자記者의 말은 차지하고서라도, 웬지 이 개발자란 어휘가 썩 마음에 들지 않습니다.

20세기 말 소프트웨어 업계에는 패턴이라는 광풍이 불어쳤습니다. 1920년대 시카고도 아닐진대 누구나 갱들의 전설적인 이야기를 했었지요. 그들의 총구 앞에 모든 문제는 손을 들 수밖에 없으며 장구 앞에 쌓아놓은 돈을 패턴에 따라 가져가기만 하면 되는 것처럼 보였습니다.

물론 대부분의 추종자는 그 전설의 네 갱들처럼 문제를 쉽게 털진 못했지만 그건 단순히 패턴을 잘못 적용했기 때문이므로 문제에 사로잡힌 불운한 추종자는 죽음의 행진 중에서도 패턴에 대한 믿음은 깊어져만 갔지요.

그때 일련의 쾌락주의자들이 나타났습니다. 갱들이 그들의 문제 해결을 위해 패턴 이란 기관총으로 클라이언트에게 “너희 문제는 이것이고 또한 이것이라야 한다.”라고 육박질렀던 것에 반하여, 클라이언트를 행복하게 해 주어야 문제가 해결된다 고 쾌락주의자들은 믿었습니다. 또한 클라이언트란 자신이 무엇을 원하는지조차 모르므로 그들과 계속 대화하면서 그들이 진정으로 원하는 것을 알아차릴 때까지 입안의 혀처럼 살살 굴러주다 보면 클라이언트 모두가 행복해지는 날이 오고 따라서 문제는 자연스럽게 해결된 것과 같다고 주장했습니다. 모든 문제는 바로 클라이언트가 만족하지 않는다는 한 가지만 중요했던 것이지요.

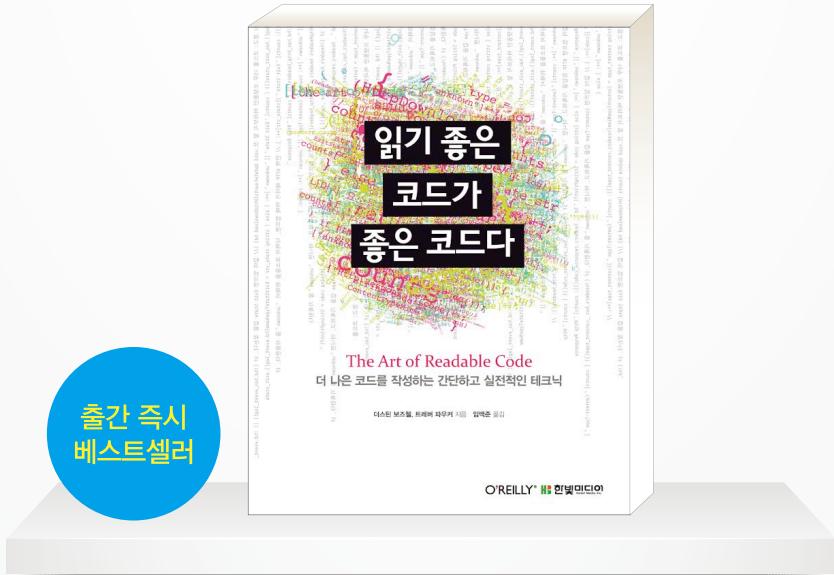
네. 그렇습니다. 모두가 만족해하고 일정도 맞추고 돈도 챙겨서 떠나갈 수 있으니 흄잡을 곳 없이 좋네요. 하지만 마음 깊숙이 웬지 이렇게 사는 것이, 다른 이의 욕망을 채워주는 것을 자신의 목적으로 사는 삶 이것으로 괜찮은 건지 웬지 모를 불안감은 가시지 않습니다.

프로그래밍이란 특정한 입력값을 받아 동작하는 프로그램을 만드는 일련의 작업을 의미합니다. 이 프로그램을 만드는 프로그래머는 자신의 설계대로 자기 손으로 만들어진 하나의 창조물이 자신의 의도대로 동작하는 것을 보며 만족감을 느낍니다.

개발자란 말을 들을 때마다 무엇이 어떻게 돼야 한다라는 목적에 대한 결정권을 다른 사람에게 맡기고 그저 부여받은 책무에 대해 처리하면 된다는 의기소침한 소시민의 이미지가 떠오르곤 합니다. 요즈음 개발자란 언급을 직간접적으로 받으면서 내가 진정 원하는 것에 대해 한 번 더 생각해 보곤 합니다.

# The Art of Readable Code

자신의 코드를 남에게 보여주기가 꺼려집니까?



## 읽기 좋은 코드가 좋은 코드다 : 더 나은 코드를 작성하는 간단하고 실전적인 테크닉

“이 책을 번역하기로 결정한 과정은 다소 우연이었지만, 번역 과정에서 내가 평소에 생각하던 부분들이 이렇게 책으로 정리되어 나왔다는 사실에 안도감과 고마움을 느꼈다. 내가 지금까지 경험한 바에 의하면, 이 책을 읽지 않아도 좋은, 원래부터 간결하고 효율적인 코드를 작성하는 능력을 가진 프로그래머는 열에 하나에 불과하다. 자신이 그 하나에 속한다는 확신이 없으면, 이 책을 꼭 읽어보기 바란다”

– 임백준, 소프트웨어 개발자 & IT 라이터

더스틴 보즈웰, 트레버 파우커 지음 | 임백준 옮김 | 18,000원

# 독자와 함께 만듭니다

한빛미디어의 독자 서비스에 참여해 보세요

## \* 한빛미디어 도서 구매 후 인증하세요!

-  **한빛이코인 적립!**
-  **부가서비스 이메일 자동 전송!**
-  **구매 도서 독자평가 실시!**

혜택1. 도서 구매 시 현금처럼 사용할 수 있는 한빛이코인을 적립해 드립니다.

혜택2. 구매 도서에 대한 정오표, 부록 등 관련자료를 이메일로 자동 전송해 드립니다.

혜택3. 구매 도서에 대한 독자평기를 실시, 기획에 적극 반영합니다.



인증방법: 한빛미디어 홈페이지▶로그인▶도서인증

[www.hanb.co.kr](http://www.hanb.co.kr)

## \* 한빛리더스가 되세요!

### 한빛리더스(Readers&Leaders)란?

한빛미디어에서 출간된 도서에 대해 리뷰 등 객관적인 도서 사후 평가, 출판사의 기획과 마케팅 활동을 돋기 위한 미션을 수행하는 독자들의 모임입니다.

한빛리더스는 독자들의 경험과 지식을 공유하여 더 좋은 도서를 출간하기 위한 독자들과의 커뮤니케이션 장입니다. 한빛미디어를 사랑하는 많은 독자님들의 참여 바랍니다.



**신간도서 리뷰 및 오픈자 등록**



**한빛도서 토론회**



**번개 미션 수행**



**설문조사 참여**



**미출간 도서 읽고 미션 수행**



**기획의견 및 벤치마킹 아이디어**



매년 2회 모집하는 한빛리더스는 페이스북을 통해 활동합니다.

지금 홈페이지에서 모집 현황을 확인하세요!

<http://facebook.com/hanbitreaders>

# 책으로 여는 IT 세상

한빛미디어는 IT 개발자 여러분과 함께 발전해 왔습니다



## 프로그래머로 사는 법 : 프로그래머의 길을 걸어가는 당신을 위한 안내서

샘 라이트스톤 지음 | 서환수 옮김 | 25,000원 | **베스트셀러**

소프트웨어 업계에서 취직하는 방법, 높은 자리로 올라가는 방법, 최고 자리로 가는 방법처럼 프로그래머로 경력을 시작하는데 필요한 정보가 모두 담겨 있다. 스티브 워즈니악, 제임스 고슬링 등 세계적 프로그래머 구루의 독점 인터뷰가 담겨 있으며, 한국 개발자 9인의 인터뷰도 포함했다



## 읽기 좋은 코드가 좋은 코드다

: 더 나은 코드를 작성하는 간단하고 실전적인 테크닉

더스틴 보즈웰, 트레버 파우커 지음 | 임백준 옮김 | 18,000원 | **베스트셀러**

이 책은 코드를 작성할 때 언제나 적용할 수 있는 기본적인 원리와 실전적인 기술에 초점을 맞추고 있다. 누구나 쉽게 이해할 수 있는 코드를 예제로 사용하고, 각 장은 코딩과 관련한 다양한 주제를 다룬다. 그리하여 여러분이 어떻게 이해하기 쉬운 코드를 작성할 수 있는지를 보여준다.



## Make : Technology on Your Time Volume 04

메이크 미디어 지음 | 메이크 코리아 옮김 | 15,000원

3D 프린터를 중심으로, 탁상 제조 시대를 여는 툴과 기술을 '책상 위 공작소'라는 주제로 소개한다. 오라일리 미디어에서 2005년 창간한 MAKE 매거진은 한국에서는 한빛미디어에서 2011년 5월에 출간하였으며, 2012년에 국내 최초의 메이커페어인 '메이커페어 서울'을 개최했다.



## 린 스타트업 : 실리콘밸리를 뒤흔든 IT 창업 가이드

애시 모리아 지음 | 위선주 옮김 | 최환진 감수 | 18,000원

최근 벤처창업은 출발 당시 세웠던 플랜 A를 시의 적절하게 플랜 B, 플랜 C로 발전시키는 것을 '성공하는 비결'로 제시한다. 이 책은 플랜 A를 플랜 B, 플랜 C로 발전시키기 위한 사업 계획 수립과 고객 인터뷰 방법, 스타트업의 효율을 위한 업무 지침까지 저비용 고효율을 가능케 하는 린 스타트업의 진수를 보여준다.