3장

구축 및 테스트

Tim O'Reilly가 말하는 웹2.0에 대해서도 마찬가지입니다. Tim O'Reilly는 웹 진화 과정을 개념적으로 명확히 함으로써 웹 진화를 촉진시켰습니다. 웹2.0이란 말이 널리 퍼지기 시작하면서 웹이나 업계 전체가 활기차게 된 것은 틀림없습니다. 웹2.0이 해 준 역할이 상당히 큽니다.

타인에게 설명하기는 어렵지만, 커뮤니케이션을 하기 위해 필요한 개념에 대해 이름을 붙입니다. 그렇게 함으로써 커뮤니케이션이 원활하게 되고, 막연하기만 했던 개념이 인식되는 "형태"가 되어 보급됩니다.

IT 아키텍트라는 말도 아작스나 웹2.0처럼 시스템 개발을 원활하게 하고, IT의 질을 향상시키기 위한 요소로서 필요한 개념입니다.

튜닝은 현상이 나타내는 메시지를 주의 깊게 파악하여 그 근본 원인을 해결하는 작업이라고 할 수 있다. 절대 단순히 현상 하나만 파악하여 일시적으로 부분적인 결론을 도출해서는 안 된다.

No.078 **현상만 보고 튜닝을 서둘러서는 안 된다**

No. 056

64비트 OS가 32비트 OS보다 우수하다고 생각해서는 안 된다

Intel 64(EM64T)나 AMD 64와 같은 64비트 아키텍처를 채택한 CPU가 보급되자, 윈도우즈나 리눅스에서도 64비트가 이용되는 경우가 많아졌다. 지금까지 주류였던 32비트 아키텍처와 비교해 보면, 한 번에 연산할 수 있는 비트 수가 증가되어 성능을 올릴 수 있기 때문에, 64비트 OS를 사용하고 있는 것 같다. 사실, 동작하는 어플리케이션이 32비트임에도 불구하고 64비트 OS를 채택하는 경우를 본 적이 있다. CPU가 64비트 아키텍처니까 64비트 OS를 선택하는 것이 베스트라고 단순하게 생각해서는 안 된다.

32비트 바이너리를 64비트 OS에서 동작시켜도 그다지 혜택은 없다

OS는 구축하고자 하는 서버의 하드웨어 스펙과 동작하는 소프트웨어의 특성을 보고 선택해야 하며, 64비트 OS를 고집할 필요는 없다. Intel 64나 AMD 64는 64비트 모드이지만, 32비트 명령세트로도 처리 성능을 떨어뜨리지 않고 실행할 수 있는 모드가 하드웨어 레벨에서 준비되어 있다. 그래서 32비트 OS도 64비트 OS도 동작할 수 있다. 또, 64비트 OS에서 32비트 어플리케이션을 동작시킬 수도 있지만, 32비트 어플리케이션은 OS가 64비트여도 64비트 모드의 혜택은 받을 수 없다.

64비트 모드는 레지스터를 이용하여 어플리케이션을 고속화할 수 있다. 단, 64비트 어플리케이션으로 컴파일이 되어 있어야 한다. 기존의 32비트 바이너리를 이용하는 어플리케이션은 64비트 OS에서 동작시켜 보았자 혜택은

없다. 어플리케이션에 따라서는 64비트 OS에서 제대로 동작되지 않은 경우도 있다.

프로그램 사이즈가 커진다

32비트 OS와 64비트 OS를 메모리 이용 관점에서 비교해 보자(그림 3-1).

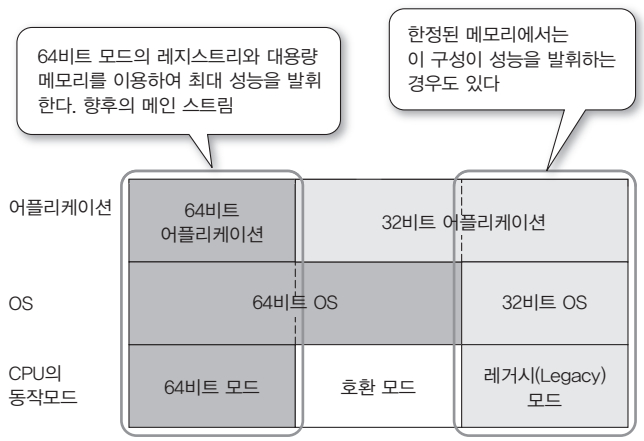

그림 3-1 OS와 어플리케이션의 선택

64비트 OS를 이용하는 가장 큰 장점은 메모리 제약을 받지 않는다는 점이다. 32비트 OS에서는 일반적으로 4GB(기가 바이트)까지의 물리 메모리밖에 취급할 수가 없다(32비트로 지정할 수 있는 메모리 주소가 2의 32승 = 4GB이므로). 한편, 64비트 OS의 경우는 논리적으로 약 1678만 TB(테라 바이트)의 메모리 공간을 사용할 수 있다. 실제 하드웨어 제약상 이보다는 훨씬 작은 사이즈지만, 그렇다고 해도 TB(테라 바이트) 사이즈의 메모리를 이용할 수 있다.

더욱 더 중요한 것은, 하나의 프로세스가 취급할 수 있는 메모리 공간의 사이즈다. OS는 실제 탑재되어 있는 물리 메모리보다 많은 메모리 공간을 논리적으로 이용할 수 있도록 가상 기억 구조를 갖고 있다.

가상 기억의 메모리 주소는 가상 주소로 관리된다. 32비트 OS에서는 가상 메모리의 주소도 32비트로 제한된다(32비트 윈도우즈의 경우 절반을 OS에 할당하기 때문에, 어플리케이션은 2GB밖에 이용할 수 없다). 64비트 OS에서는 이 제약이 해소된다.

지금까지의 설명으로는, 64비트 OS를 이용하는 편이 좋다고 생각될지도 모른다. 다만, 문제는 메모리의 데이터 양이다. 64비트 어플리케이션을 실행시키면 주소를 보관하기 위한 포인터의 사이즈가 두 배가 되고, CPU의 명령 코드 사이즈도 커진다. 그래서, 프로그램의 바이너리 사이즈가 커진다.

프로그램을 실행할 때 바이너리를 메모리에 전개하는 경우도 32비트보다 많은 용량이 필요하고, 캐시의 히트율도 떨어진다.

RDBMS와 같이, 취급하는 데이터 사이즈가 큰 어플리케이션은 32비트 어플리케이션보다 64비트 어플리케이션이 데이터 파일(보존용)의 사이즈가 커진다는 것도 주의할 점이다. 조작해야 할 파일이 크기 때문에, 파일에서 데이터를 검색할 때 성능이 좋지 않다.

어플리케이션이 이용하는 메모리의 최대 용량이 32비트 OS의 가상 메모리가 이용할 수 있는 최대 사이즈를 넘지 않으면, 32비트 OS에서 32비트 어플리케이션을 이용하는 편이 처리 성능을 높일 수 있다. 64비트 OS에 집착할 필요는 없다.

소스코드

No. 057

기호 링크를 조심성 없이 이용해서는 안 된다

시스템에 산재한 설정 파일을 한 곳에 모아 관리 효율을 높이고 싶을 때가 있다. 유닉스계 OS에는 파일을 닉네임으로 참조하는 기호 링크 기능이 있어, 이것을 이용하여 관리 효율을 높일 수 있다.

링크할 실제 파일과 기호 링크는 동등하게 취급되므로, 어느 쪽으로 링크해도 괜찮다고 생각할지도 모른다. 그러나, 그렇지 않다.

링크에는 하드 링크와 기호 링크가 있다. 유닉스계 OS의 파일시스템은 하나의 파일에 여러 개의 파일명을 붙일 수 있다. 즉, 하나의 파일에 여러 개의 이름을 붙이는 처리가 하드 링크다. 하드 링크의 경우 여러 개의 파일명의 내용이 모두 똑같다.

이에 반해 기호 링크는 닉네임을 붙이는 것에 해당하며, 기호 링크를 하게 되면 파일의 외형만 링크되고, 링크된 파일명과 파일이 존재한 패스가 보관된다.

즉, 보관된 파일명과 패스 정보를 기반으로 링크를 한다.

2개의 파일명으로 동일한 파일에 접근해야 할 경우에는 하드 링크를 이용하는 편이 좋다. 다만, 하드 링크는 디바이스나 파티션이 다르면 링크할 수 없다. 또, 대부분의 파일시스템이 디렉토리 하드 링크를 쉽게 작성할 수 없다. 그래서 기호 링크가 많이 사용된다.

복사(copy)와 이동(move)에 따라 생성되는 파일이 다르다

기호 링크는 데이터와는 무관하게 파일의 외형만 링크되기 때문에 프로그램에서 링크된 파일명을 지정하게 되면 원본 파일과 똑같이 취급할 수 있다. 그러나, 링크된 파일명 자체를 이용하여 처리할 경우에는 원본 파일처럼 사용하면 문제가 발생하므로 주의가 필요하다.[1]

다시 말하면, 파일의 이동(mv), 복사(cp) 등의 조작이나 파일의 아카이브(tar) 처리 등이다.

테스트 파일(리눅스에서 실행)
실제 파일 「real le_a」「real le_b」, 기호 링크 「symlink」가 존재한 디렉토리에서 실행

```
# ls –l
합계8
–rw–r––r–– 1 root root 20 6월 16 08:39 realfile_a
–rw–r––r–– 1 root root 10 6월 16 08:40 realfile_b
lrwxrwxrwx 1 root root 10 6월 16 08:40 symlink –〉 realfile_a
```

(1) cp 커맨드로 실제 파일 「real le_b」를 「symlink」로 복사했을 경우
→ 실제 파일 「real le_a」가 「real le_b」의 내용으로 덮어써진다.

```
$ cp realfile_b symlink
cp: ‘symlink’를 덮어 쓸까요?(yes/no)? yes
$ ls –l
합계8
–rw–r––r–– 1 root root 10 6월 16 08:40 realfile_a
–rw–r––r–– 1 root root 10 6월 16 08:40 realfile_b
lrwxrwxrwx 1 root root 10 6월 16 08:40 symlink –〉 realfile_a
```

(2) mv 커맨드로 실제 파일 「real le_b」를 「symlink」로 이동했을 경우
→ 「real le_b」의 이름이 「symlink」로 바뀐다.

```
$ mv realfile_b symlink
mv: ‘symlink’를 덮어 쓸까요?(yes/no)? yes
$ ls –l
합계8
–rw–r––r–– 1 root root 20 6월 16 08:40 realfile_a
–rw–r––r–– 1 root root 10 6월 16 08:40 symlink
```

그림 3-2 **cp커맨드와 mv커맨드의 차이**

1 파일을 저장하면 하드디스크의 어딘가에 저장한 파일의 내용이 기록된다. 그리고 하드디스크에 기록된 정보를 헤더에 저장한다. 즉, 헤더에 있는 위치 정보만을 갖고 있기 때문에 파일을 호출하면 호출한 파일이 갖고 있는 위치 정보를 이용하여 하드에서 내용을 찾아 사용하게 된다.
하드 링크는 이 위치 정보를 갖고 있는 이름을 여러 개 생성한다고 생각하면 된다. 그래서 하나를 지우더라도 하드에서 내용을 찾아 갈 수 있다. 하지만 기호 링크는 위치 정보를 갖고 있는 파일명을 또 한번 다른 이름으로 연결시키고 있기 때문에 원본 파일을 삭제하면 기호 링크 파일들은 위치 정보가 없어져서 무용지물이 된다.

원본 파일로 기호 링크 파일을 덮어 썼을 경우 복사 커맨드(cp)와 이동 커맨드(mv)의 결과는 다르다. 그림 3-2처럼 커맨드를 실행하면 cp 커맨드는 참조할 원본 파일이 갱신되는데 반해, mv 커맨드는 기호 링크에 덮어 써진다.

기호 링크는 읽기 전용으로 사용한다

커맨드 실행 결과의 차이를 의식하지 않기 위해서는 기호 링크는 읽기 전용으로만 사용하고, 변경이 필요하면 원본 파일을 변경한다는 방침을 세워야 한다.

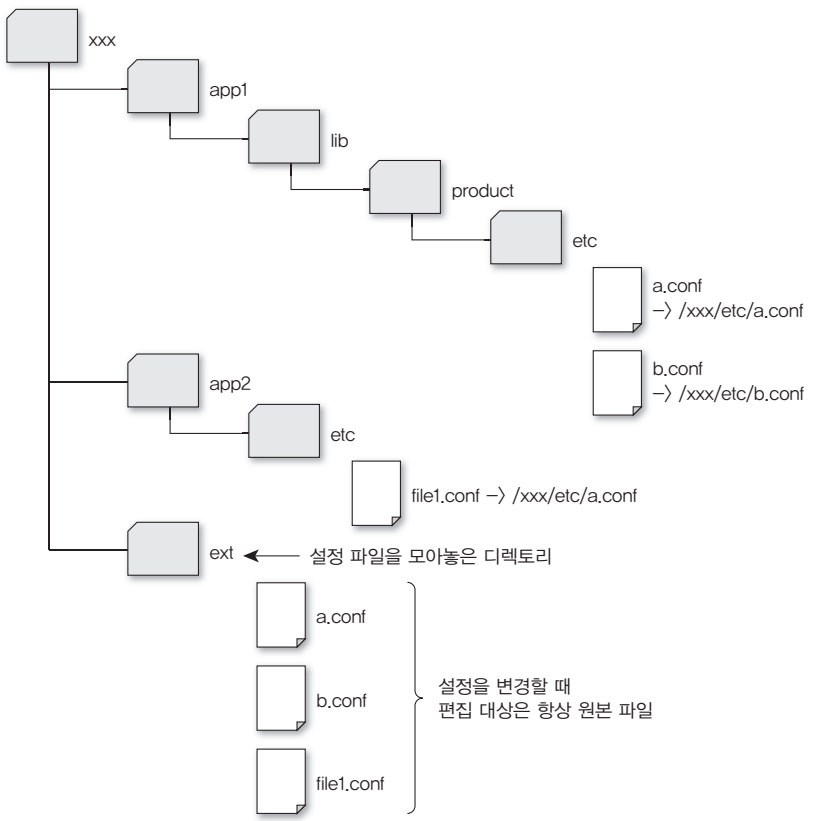

그림 3-3 **기호 링크를 이용한 설정 파일**

여러 디렉토리에 산재되어 있는 설정 파일을 효율적으로 갱신하기 위해, 하나의 디렉토리 안에 모아 놓는다(그림 3-3). 기호 링크를 이용한 파일 조작은 피한다는 방침에 따라 편집을 하기 위해 설정 파일을 모아 놓은 디렉토리(그림 3-3에서는 "ext")에 원본 파일을 넣고, 원본 파일에 기호 링크를 한다. 이렇게 하면, 설정 파일을 모아 놓은 디렉토리 안에 원본 파일들이 모이기 때문에 파일 조작은 신경 쓰지 않아도 된다.

또 한 가지, 기호 링크를 이용하는 데 있어 주의해야 할 점은 기호 링크의 지연 평가[1] 성질이다. 기호 링크를 작성한 시점에 링크한 파일이 존재했다고, 그 파일이 계속 존재한다고는 할 수 없다. 파일이 삭제되어, 이른바 링크가 끊어진 상태가 될 수 있다. 즉, 파일이 존재해도 다른 실체의 파일일지도 모른다.

1 지연평가: lazy evaluation으로 진짜 필요해 질 때까지 미루는 것

소스코드

No. 058

여러 가지의 OS를 이용할 때는 개행 코드를 무시해서는 안 된다

유닉스계의 OS로 구축된 시스템은 주로 여러 종류의 OS를 사용한다. 서비스를 제공하는 서버가 유닉스계 OS로 통일되어 있어도, 관리용 단말이나 개발용 단말 OS는 윈도우즈가 많다. 다른 종류의 OS로 작업할 때 조심해야 할 것 중의 하나가, OS간의 개행 코드의 차이다. 개행 코드는 유닉스계 OS에서는 "LF", 윈도우즈에서는 "CR+LF"다(CR: Carriage Return(0x0A), LF: Line Feed(0x0D)).

단말과 단말 사이에 파일을 전송할 때 쉘 스크립트나 펄 스크립트를 텍스트 파일 그대로 전송하고 있을 것이다. 그러나, 스크립트는 가독성이 있는 텍스트 형식으로 기재되기는 하지만, 프로그램이 번역되어 실행되기 때문에 개행 코드가 다르면 실행되지 않는 것이 있다.

확장자에 따라 전송 모드를 변경한다

파일 전송 툴 안에는 텍스트 형식 파일의 개행 코드를 자동으로 번역하는 기능을 갖고 있기 때문에, 텍스트 파일을 그대로 전송하려면 주의가 필요하다. 실제로, 개발 환경의 유닉스계 OS머신에서 작성하고 시험까지 통과했는데도, 윈도우즈 경유로 실제 환경에 전송하여 동작을 확인하려고 하자 개행 코드가 바뀌어서 실행할 수가 없었다(그림 3-4).

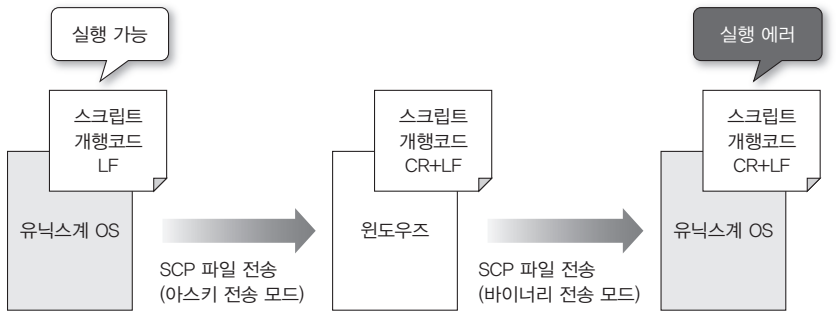

그림 3-4 **전송 모드의 설정 오류**

파일 전송은 전송용 프로토콜인 FTP나 SCP를 주로 이용하지만, 이러한 프로토콜에 대응한 클라이언트 툴의 대부분은 "ASCII 전송 모드"와 "바이너리 전송 모드" 이 두 종류의 전송 방식을 제공한다.

윈도우즈에서 동작하는 클라이언트는 파일 확장자에 따라 모드를 바꿔서 구축해야 하는 경우가 많기 때문에 주의해야 한다.

"README.txt"라는 파일은 개행 코드가 변환되어 전송되고, "README" 파일은 변환되지 않고 전송된 적이 있었다. README 파일처럼 텍스트 파일이라면 그렇게 심각할 정도는 아니지만, 스크립트 파일의 경우는 개행 코드가 다르면 실행할 수 없는 사태가 발생한다.

FTP나 SCP의 ASCII 전송 모드로 파일을 전송했을 때는, 전송 후에 파일의 개행 코드가 의도한 대로 되어 있는지 항상 확인해야 한다.

개행 코드의 영향을 받지 않는 전자 메일

전자 메일에 첨부해서 서로 다른 OS간에 파일을 송수신할 경우, 기본적으로 첨부 파일은 바이너리 형식으로 송부되고 OS의 개행 코드의 영향은 받지 않는다. 메일의 첨부 파일은 메일 전송 프로토콜SMTP로 첨부 파일을 취급하기 위해, 확장 규격인 MIME으로 전송되기 때문이다.

SMTP에서는 기본적으로 7비트 이하의 코드만을 취급하는 제약이 있기 때문에, MIME 인코드로 바이너리 파일을 일단 7비트 이내의 ASCII 코드로 변경해서 송부한다. 수신 측은 송부된 내용을 디코드해서 바이너리 데이터로 복원한다.

MIME은 첨부 파일의 종류를 나타내기 위한 Content-Type에 텍스트 파일임을 나타내는 "text"를 지정할 수도 있지만, 메일러(이메일의 송수신 기능을 수행하는 프로그램)로 Content-Type을 해석한다. 개행 코드를 변환하는지 알 수는 없지만, 변환 처리가 생길 수도 있으므로 전혀 없다고 단언할 수 없다.

이와 같이 전송에 이용하는 툴로 인해 개행 코드가 변경되는 위험이 존재한다. 개행 코드가 변경되는 위험을 회피하기 위해 파일을 작성한 기기에서 tar나 zip 커맨드로 전송 대상 파일을 압축하여 바이너리 파일로 전송하고, 전송 대상 기기에서 해제하는 것을 룰로 정하면 개행 코드까지 확실하게 전해 줄 수 있게 된다.

로그 파일이나 소스코드에도 요주의

여러 종류의 OS가 혼재하는 시스템을 개발할 때는 문자 코드도 주의해야 한다. UI는 물론이고 네트워크로 교환하는 데이터 내용까지, 데이터베이스를 설계할 때는 문자 코드가 가장 신경이 쓰이는 부분이다.

반면, 로그나 소스코드의 설계는 소홀해지기 십상이다. 프로그램의 보수성 측면에서 로그나 소스코드도 명확하게 규정해야 한다.

로그를 출력할 때 로컬의 로그 파일에는 OS 내에서 사용할 수 있는 문자 코드로 출력하기 때문에 그다지 문제가 되지 않지만, 로그를 모아 놓는 서버의 경우에는 문제가 된다. 로그를 송신한 곳, 혹은 로그 서버를 수신한 곳에서 문자 코드의 변환 처리가 필요하기 때문에, 대응하고 있는 문자 코드를 조사하여 송신 측에서 변환할지, 수신 측에서 변환할지 정해 두어야 한다. 로그 출력은 바이트 코드를 많이 사용하지 않고, ASCII 코드만으로 설계하는 것도 좋은 방법이다.

소스코드는 형상관리 소프트웨어로 관리할 때 문제가 생기기 쉽다. 개발 기기의 OS 종별에 따라 문자 코드가 다른 상태로 저장되는 사태는 가능한 피해야 한다. 통합 개발 환경에 따라서는 저장 장소에서 소스코드를 취득한 시점에 자동적으로 문자 코드나 개행 코드가 환경에 맞춰 변경될 때가 있다. 자동으로 변환되고 있는 것도 모르고, 변환된 채로 저장해 버리면 통합 개발 환경 이외에서는 빌드할 수 없는 사태가 발생할 수도 있기 때문에 특히 주의가 필요하다.

소스코드

No. 059

정의된 것 이외의 것을 가볍게 보아서는 안 된다

상세 설계나 프로그램 개발에 정의되어 있지 않은 "미지 상태"에 대해 사전에 고려를 해 놓으면, 트러블 발생을 미리 막을 수 있다. 설계나 구축 시점에서는 있을 수 없는 일일지도 모르나, 횟수를 거듭할 수록 상황이 바뀌어 일어날 수 있기 때문이다. 미리 정의를 해 두는 것이 중요하다.

미지 상태를 고려하지 않아 오류가 발생

미지 상태란 요구 사양에 기록되지 않은 상태를 말한다. 예를 들면, 다음의 요구 사양이 있었다고 하자.

상태state가 100이면 X처리(테이블 A의 갱신)를 하고, 200이면 Y처리(테이블 B의 갱신)를 한다.

이 요구 사양에는 상태가 100도 200도 아닌 그 이외의 경우에 대해서는 어떻게 취급해야 할 지 기록되지 않았다. 설계자나 프로그래머가 이 요구 사양을 "100과 200 이외의 상태는 고려하지 않아도 된다"라고 생각했다면 리스트 3-1과 같이 구축했을 것이다.

리스트 3-1은 100 이외는 모두 200으로 간주한 것이다. 요구 사양을 만족하고는 있지만, 100이나 200 이외의 상태에 대해서는 고려하고 있지 않기 때문에, 다음과 같은 트러블이 발생할 가능성이 높다.

예를 들면, 메소드나 함수의 파라미터는 state 변수고, 외부에서 100이나 200 이외의 상태, 예를 들어 state에 500이 지정되면 state가 200일 때 처리되어야 할 Y처리를 하게 되어 테이블 B가 갱신되게 된다.

또, "300이 추가되면 Z처리(테이블 C의 갱신)를 한다"는 사양 추가가 생겼다고 하자. 이 때 다른 소스 코드는 전부 수정했는데, 리스트 3-1만 수정하는 것을 깜박 잊었다면 300에서도 Y처리를 하게 되어 테이블 B가 잘못 갱신되게 된다.

리스트 3-1 state가 100, 200 이외를 고려하지 않는 구축의 예

```
if( state == 100 ){     // state가 100의 경우
    X 처리 (테이블 A의 갱신 처리)
}else{      // 그 이외의 경우
    Y 처리 (테이블 B의 갱신 처리)
}
```

이렇게, 누락이나 오류에 의한 버그는 생각 외로 많다. 그리고 이러한 버그는 늦게 발견될 가능성이 높기 때문에, 설계자나 프로그래머는 정의되지 않은 것까지도 빠트리지 않도록 주의해야 한다. 설계자는 정의되지 않은 것에 대해 어떻게 처리해야 할지 설계서에 명기해 주어야 한다. 그리고, 프로그래머는 "그 이외의 경우는?"이라고 하는 의문을 갖고 구축하는 습관이 필요하다. 정의되지 않은 상태를 고려하여 구축했을 때는 리스트 3-2와 같다.

리스트 3-2 state가 100, 200 이외를 고려한 구축의 예

```
if( state == 100 ){     // state가 100의 경우
    X 처리
}else if( state == 200 ){     // state가 200의 경우
    Y 처리
}else {     // 정의되지 않는 state의 경우
     에러 (예를 들면, 예외를 발생시킨다)
}
```

소스코드

No. 060

공개 기능 클래스의 인스턴스를 직접 생성해서는 안 된다

일반적으로 자바로 클래스의 인스턴스를 생성하려면 리스트 3-3과 같이 구축한다. 여기서 Supplier는 공개 기능을 제공하는 인터페이스이고, SupplierA는 구축할 클래스, User는 공개 기능 클래스를 이용하는 클래스다. User 클래스는 Supplier 인터페이스가 제공하는 공개 기능을 이용하기 위해 SupplierA에 new 연산자를 이용하여 인스턴스를 생성하고 있다(리스트 3-3❶).

리스트 3-3 인스턴스를 생성하는 자바 코드의 예

```
[Supplier.java]
public interface Supplier {
    void doSomething();
}
[SupplierA.java]
pulibc class SupplierA implements Supplier {
    public void doSomething(){
     ....
    }
}
[User.java]
public class User {
    private Supplier supplier;
    public void process(){
        // ❶ SupplierA의 인스턴스를 생성
        Supplier supplier = new SupplierA();
        // ❷ Supplier의 doSomething 기능을 이용한다
        supplier.doSomething();
....
    }
}
```


그리고 나서, 인터페이스 Supplier의 구축 클래스(SupplierA)의 인스턴스로 제공된 doSomething 메소드를 이용하고 있다(리스트 3-3❷).

그런데, 이 구축에는 아무런 문제가 없는 것처럼 보인다. 그러나, 새로운 SupplierB 클래스가 추가되어 인스턴스의 생성 대상 클래스가 SupplierA에서 SupplierB로 무조건(혹은 특정 조건에서) 바뀐다면 어떻게 될까?

SupplierA에서 SupplierB로 바뀐 경우는 ❶의 부분을 다음과 같이 수정하게 된다.

```
// ❶ SupplierB의 인스턴스를 생성
Supplier supplier = new SupplierB();
```

일반적으로 공개된 공통 기능은 다양한 클래스에서 이용된다. 만약, Supplier의 공통 기능을 이용하는 클래스가 User 이외에도 있었다고 하면, 그것들에 대해서도 똑같이 수정이 필요하다.

리스트 3-3에 나타낸 코드에서 문제점은, 공통 기능을 이용하는 User 클래스가 직접 구축 클래스의 인스턴스를 생성하고 있는 점이다. User 클래스는 인터페이스 Supplier가 제공하는 공통 기능을 이용하고 싶은 것뿐이며, 인터페이스 Supplier의 구축 클래스인 SupplierA나 SupplierB에 의존하고 싶지 않다. 공통 기능을 제공하는 측은 구축 클래스에 의존하지 않고 공통 기능을 이용할 수 있도록 배려해야 한다.

Factory Method로 간접화한다

이 과제를 해결하려면 Factory Method라는 디자인 패턴을 이용하면 좋다. Factory Method란 오브젝트의 생성을 맡고 있는 메소드factory method를 이용하여 간접적으로 오브젝트를 생성하는 방법을 말한다. 개별적으로 인스턴스를 생성하지 말고 인터페이스를 이용하여 클래스의 인스턴스를 생성할 수 있도록 메소드를 준비한다.

리스트 3-4의 구축 예를 보자. SupplierFactory 클래스로 인스턴스를 생성하는 클래스 메소드 createSupplier를 준비했다. User 클래스는 클래스의 인스턴스를 직접 생성하는 것이 아니고 createSupplier를 이용하여 인스턴스를 생성한다. 인스턴스의 생성을 SupplierFactory 클래스에 맡긴다는 얘기이다.

리스트 3-4 Factory Method의 구축 예

```
[SupplierFactory.java]
public class SupplierFactory {
    public static Supplier  createSupplier(){
        // Supplier의 구축 클래스의 인스턴스를 생성하여 돌려준다
    }
}
[User.java]
public class User {
    public void process(){
        Supplier supplier = SupplierFactory.createSupplier();
        supplier.doSomething();
    }
}
```


소스코드

No. 061

거대한 정수 클래스를 만들어서는 안 된다

정수의 정의는 정수 클래스(정수 정의 클래스)에 모여 있다. 자바나 C++은 리스트 3-5와 같이 정수 클래스를 구축한다.

리스트 3-5 정수 클래스의 구축 예

```
【 Java의 경우】
public class Construct {
    public static final int MY_CODE = 100;
}
【 C++의 경우】
class Construct {
public
    static const int MY_CODE = 100;
}
```

정수 클래스에 정수를 정의하려면, 정적 변수를 나타내는 static 수식자를 붙이거나 final(C++의 경우는 const) 수식자로 고정 변수라고 정의한다.

정수 클래스에 대한 지침은 대부분 어플리케이션 개발 지침 등에 간결하게 기록되어 있다. 그런데, 대규모 프로젝트임에도 불구하고 역할이나 목적에 따른 정수 클래스에 대한 방침이 없을 때가 있다. 정수가 여기 저기 분산되지 않도록 정수 전용 클래스를 준비한다는 식의 기록이 있으면, 하나의 정수 클래스에 모든 정수 정의가 집중되어 거대한 정수 클래스가 완성된다.

일반적으로, 프로그래머가 필요로 하는 정수는 1개 내지 기껏해야 몇 개 정도다. 프로그래머가 리스트 3-6과 같이 거대한 정수 클래스에서 필요한 정수를 찾아야 한다고 상상해 보기 바란다.

리스트 3-6 거대한 정수 클래스의 예

```
【Java의 경우】
public class Construct {
    /** 처리 결과 코드 : 대상 없음 */
    public static final int NOT_FOUND = -1;
    /** 처리 결과 코드 : 정상 */
    public static final int SUCCESS = 0;
    /** 응답 메시지 : 변경 없음 */
    public static final String NOT_CNANGE = "Not change.";
    /** 응답 메시지 : 변경 있음 */
    public static final String UPDATED = "Updated.";
    /** 처리 옵션 : 일반 */
    public static final int NORMAL_OPT = 100;
    /** 처리 옵션 : 확장 */
    public static final int SPECIAL_OPT = 101;
··· (이하, 다양한 정수의 정의가 끝없이 계속된다)···
}
```

프로그래머는 개발 지원 툴로 오로지 IDE(통합 개발 환경)를 이용하고 있다. IDE에는 프로그래밍을 효율적으로 할 수 있도록 코드 보조 기능(입력 보완 혹은 입력 후보를 제시하는 기능)을 갖추고 있다. 그런데, 정수 클래스가 거대하다면 코드 보조 기능으로 많은 입력 후보들이 표시된다. 그 중에서 적절한 정수를 선택하기란 너무 어렵고 작업 효율 또한 크게 떨어진다.

정수 클래스를 분할하는 네이밍도 중요하다

본래 정수 클래스를 마련하는 목적은 다양한 정수를 정수 클래스에 정리하는 것에 있다. 혼재된 상태의 거대한 정수 클래스는 본래의 모습이 아니다.

정수 클래스는 기능이나 목적에 맞게 적절한 사이즈가 되도록 분할해야 한다.

앞에서 말한 거대한 정수 클래스는 "처리 결과 코드", "응답 메시지", "처리 옵션" 등과 같이 분류하면 된다.

가령, 고객 구분 등 코드 분류마다 독립된 정수 클래스를 작성하는 방침이 있었다면, 필요한 정수 클래스를 설계서에서 모두 찾아 내어 자동 생성하는 툴을 만드는 것이 좋다. 대규모 프로젝트라면 설계서에서 자동으로 정수 클래스를 생성해 주면 가장 알기 쉽고, 정착하기 쉽다.

정수 클래스의 작성과 더불어 클래스의 이름이나 정수 이름의 네이밍도 중요하다. 어떤 분류 기준으로 정수 클래스를 분할했다고 하더라도, 각 정수 클래스에 어떤 정수가 모여 있는지 클래스명으로 예측할 수 없다면 클래스를 분할한 의미가 희미해진다.

분할에 관한 지침(가이드라인)을 조기에 수립한다

적절한 사이즈가 되도록 정수 클래스를 분할하는 작업을 시스템을 구축하는 도중에 하는 것은 바람직하지 못하다. 프로젝트 멤버의 혼란을 가중시키기 때문이다.

정수 클래스의 분할 지침은 설계 공정 초기에 결정하는 것이 바람직하다. 늦어도 구축 공정에 들어가기 전까지는 수립해야 한다. 방침을 결정하면 어플리케이션 개발 지침 등에 명기해서, 프로젝트 멤버에게 주지시켜야 한다.

소스코드

No. 062

분량이 많은 코딩 규칙을 만들어서는 안 된다

코딩 규칙은 프로젝트 멤버에게 프로그래밍 룰을 알려 주기 위해 작성하는 문서다. 일반적으로 소프트웨어 아키텍트가 정리한다. 코딩 규칙이 너무 많으면 프로그래머가 그 모든 것을 이해할 수 없게 되어, 결국 준수되지 못하게 되므로 주의가 필요하다.

실제 상황에 맞춰 기존의 규칙을 수정한다

코딩 규칙 작성은 귀찮고 노력이 많이 드는 작업이다. 프로젝트마다 새로 작성할 필요는 없다. 다른 프로젝트에서 사용되고 있는 기존의 코딩 규칙을 기본으로, 사용하는 플랫폼이나 개발 언어에 맞춰 수정하면 된다.

코딩 규칙에 기술하는 항목은 일반적으로는 코딩 스타일이나 코멘트 서식, 식별자의 네이밍, 프로그래밍의 금지 사항, 관례나 팁들, 이 4가지다.

이 중, 코딩 스타일이나 코멘트 서식과 식별자 네이밍 규약을 제정하는 목적은 소스코드의 가독성 향상에 있다. 소스코드의 가독성이 높으면 프로그램을 작성한 당사자, 리뷰 담당자 모두 버그를 찾아내기 쉽다. 또, 가독성이 높으면 다른 프로그래머가 유지보수 할 때 작업 효율이 높아진다. 프로그래밍의 제약 사항을 적어 놓는 이유는, 프로그래머에 따라 별생각 없이 부적절하게 코딩을 해 버리는 경우가 있기 때문이다. 프로그래머가 재차 의식을 하면서 코딩할 수 있도록, 코딩 규칙에서 사용하는 플랫폼이나 개발 언어의 특성을 고려하여 위험한 내용을 표시한다.

관례나 팁들을 나타내는 것은, 알고리즘 등에 따른 차이를 프로그래머에게 인식시켜 처리 효율이 높은 소스코드를 만들기 위해서다(리스트 3-7).

리스트 3-7 관례나 팁들의 기재 예

```
【나쁜 예】 루프의 조건식에서 메소드를 호출한다. 성능 문제가 일어나기 쉬워 바람직하지 않다.
List list = new ArrayList();
...
for(int index = 0; index < list.size(); index++){
// 조건부에서 size 메소드를 매회 호출하고 있다!
...
}
【좋은 예】 사전에 size()를 호출, 로컬 변수에 저장하고 나서 그 로컬 변수를 참조한다.
List list = new ArrayList();
...
int size = list.size();
for(int index = 0; index < size; index++){
...
}
```

필요한 내용을 충분히 알기 쉽게 표시한다

이러한 내용을 모두 포함하게 되면 코딩 규칙의 양이 늘어나서 인쇄했을 때 두꺼워지기 십상이다. 완성도가 높은 코딩 규칙이란 필요한 내용을 충분히 알기 쉽게 나타내야 하지만, 필요 이상으로 상세하고 두꺼운 것은 바람직하지 않다.

프로그래머가 구축 작업을 할 때는 설계서를 참조하여 IDE(통합 개발 환경) 등의 툴과 서로 대조하게 된다. 코딩 중에 코딩 규칙을 참조할 필요는 없다. 프로그래머가 코딩 규칙을 읽는 것은 구축 작업을 하기 전이어야 하며, 그 때 규칙을 충분히 이해하고 나서 코딩 작업을 해야 한다. 다시 한번 강조하지만 코딩 규칙이 두꺼우면 전부 기억할 수 없고, 금지 사항 등 중요한 규칙마저 지킬 수 없게 된다. 그렇게 되면, 어렵게 만들어 놓은 코딩 규칙이 아무런 의미가 없게 된다.