5장. 하위 타겟과 빌드 변형

지금 까지는 소프트웨어가 하나의 빌드 프로세스만 갖고 있다는 점을 설명해왔다. 즉, 각 소스파일이 컴파일된 후 링크돼 하나의 실행 파일로 되는 방법이 하나일 뿐만 아니라. 릴리스 패키지도 한 유형만 생성될 수 있었다. 하지만 현실적으로 다양한 변형이 존재하며, 각 변형은 빌드 프로세스를 약간 변경하고, 약간 다른 릴리스 패키지를 생성한다. 여기서 약간이라는 것은 빌드 프로세스가 동일한 프로그램을 생성하지만, 그 프로그램의 동작에 있어 사소한 차이를 보인다는 뜻이다.

5장에서는 소스 트리에서 오브젝트 트리로의 매핑에 관한 다음과 같은 세가지 방법을 알아본다.

1. 하위 타겟 빌드하기

전형적인 빌드 프로세스에서는 모든 소스콰드를 컴파일해 오브젝트 트리를 만들고 이를 토대로 릴리스 패키지를 생성한다. 이는 타겟 머신에 소프트웨어를 설치하고 실행한다면 흔히 선행되는 일이다. 하지만 빌드 트리의 일부분을 변경한 개발자라면 빌드 트리의 모든 부분이 아닌 해당 부분만 재 빌드하길 선호한다. 이것이 하위 타겟 빌드하기로 알려져 있다.

2. 소프트웨어의 여러 다른 에디션 빌드하기

우리는 아직 소스 파일 전체를 컴파일 하겠지만, 그 결과물은 소프트웨어의 다양한 동작을 이끌어내게 수정돼야 한다. 이런 변형은 프랑스어나 독일어 등 같은 자연 언어, 홈 에디션이나 프로페셔널 에디션 같은 특징을 달리하는 데 지원하는 일을 포함할 수도 있다.

3. 여러 다른 타겟 아키텍처 빌드하기

여러 다른 타겟 머신의 소프트웨어 제품을 지원하기 위해 동일한 소스 파일을 다양한 CPU와 운영체제에서 동작하게 컴파일해야 한다. 이는 x86과 MIPS, PowerPC 같은 CPU, Linux, window, Mac OS X 같은 운영체제를 포함한다.

이런 접근법은 빌드 프로세스를 다양한 방법으로 수정할 수 있게 한다. 첫 번째 경우 소프트웨어 제품을 전부 컴파일하는 대신 그 중 일부를 빌드한다. 두 번째 경우 소프트웨어 제품 전부를 빌드하지만, 필요에 따라 전체 소스 파일 중 일부를 포함하거나 배제한다. 마지막 경우 소프트웨어 제품 전부를 빌드하지만, 컴파일 도구를 다양하게 사용해 추가로 일부 파일을 포함하거나 배제한다.

하위 타겟 빌드

소프트웨어의 어떤 부분이든 여러 작은 구성요소로 분할 될 수 있으며, 각 구성 요소는 프로그램의 전체 기능 중 일부를 지원하고, 기타 구성 요소들과 다소 독립적으로 개발된다. 그리고 큰 시스템에서 최종 소프트웨어는 서로 협력하는 여러 다른 실행 파일을 포함할 수 있으며, 이 경우 각 실행 파일이 하위 구성 요소가 된다.

만약 프로그램이 많은 소스 파일을 포함하고 있다면 그에 비례해 전체 소스 트리를 증분 빌드하는 데 아직 많은 시간이 필요하다. 빌드 도구가 변경된 파일만 다시 컴파일하더라도 makefile과 같은 build desription 파일을 모두 읽고 컴파일할 파일을 선정하는 데 시간이 필요하기 때문이다. 어쩌면 증분 빌드는 컴파일하기 전에 분석에만 2~3분이 소용될 수도 있다. 따라서 개발자는 전체 소스 트리를 다시 컴파일하는 대신에 빌드할 하위 구성 요소의 수를 제한할지도 모른다. 다시 말해 개발자가 math라이브러리에 있는 소스 파일의 일부만을 수정했다면 해당 디렉터리만 컴파일함으로써 전체 빌드 시간을 최적화할 수 있다.

추가적으로 라이브러리가 수정된 사항을 적용하기 위해 재차 실행 파일에 링크돼 컴파일 될 필요가 없는다는 뜻이다. 즉, 단순하게 수정된 라이브러리만을 다시 컴파일하고 프로그램을 다시 재실행하면 자동으로 코드 변경사항이 적용된다.

여러 다른 실행 파일을 포함하는 큰 소프트웨어에서는 일부 프로그램만을 컴파일 함으로써 빌드 프로세스를 최적화할 수 있다. 게다가 대개 타겟 머신이 이미 설치된 그 밖의 파일들은 새로이 추가될 파일과 호환된다. 따라서 빌드 도구를 호출할때마다 새로운 릴리스 패키지를 만들고 설치하는 대신, 타겟 머신에 수정된 파일만을 직접 복사함으로써 시간을 절약할 수 있다.

소프트웨어의 여러 에디션 빌드

세계 시장을 겨냥해서 소프트웨어를 개발할 때 최종 사용자의 요구 사항을 고려하는 일은 매우 중요한 사안이다. 특정 사용자에게만 국한된 소프트웨어를 만들지 않는 한, 언어와 문화, 하드웨어 차이, 소프트웨어 금액 차이를 고려해야 한다.

언어와 문화

모든 컴퓨터 사용자는 결코 동일한 언어로 말하거나 문화를 갖고 있지 않다. 그래서 사용자들은 소프트웨어를 편리하게 사용하기 위해 프로그램의 명령과 메뉴, 에러 메세지 등을 자신의 모국어로 보기를 원한다. 또한 그들은 서양식인 오른쪽으로 가로읽기 보다는 왼쪽으로 가로 읽기 또는 내려 읽기를 선호할 수도 있다.

이 localization를 가능하게 하려면 소프트웨어 개발자들은 다음과 같은 추가 작업을 해야 한다. 모든 텍스트 메세지를 지원할 언어로 번역하는 것 뿐만 아니라 소프트웨어 사용자 인터페이스 또한 다양한 형식으로 글과 이미지를 제공할 수 있어야 한다. 게다가 빌드 프로세스는 각 언어와 문화에 적합한 문자와 이미지를 선정해야 한다.

하드웨어 차이

다양한 하드웨어 플랫폼상에서 실행하게 설계된 소프트웨어는 필요한 기능을 포함할 수 있게 빌드 시에 수정될 수 있다. 리눅스 커널을 빌드하는 데 익숙한 이는 하드웨어 차이를 어떻게 빌드 시스템에 적용하는지 잘 알고 있다. 첫 번째 단계에서는 사용자에게 부여된 옵션 중 하나인 구성도구를 실행한다.

두 번째 단계에서는 수정된 커널을 생성하기 위해 빌드 프로세스를 호출한다.

마지막 단계에서는 타겟 하드웨어가 요구하는 모든 드라이브를 포함하고, 그 외 사용자가 컴파일하지 않은 것은 배제한다.

금액 차이

소프트웨어 벤더는 동일한 소프트웨어 일지라도 고객에 따라 지급하는 금액이 다르다는 점을 알고 있다. 예로 홈 유저가 재무회계 패키지를 200달러에 구매하는 반면, 회계사는 2000달러에 구매할지도 모른다. 소프트웨어 제공업체는 두 시장을 석권하기 위해 홈 유저에게는 홈 에디션을, 회계사에게는 프로페셔널 에디션을 공급한다. 유일한 차이라고는 고급 기능 몇 가지이다. 이 가격과 기능의 조합으로 두 에디션이 각 고객층을 이끌어내게 한다.

코드의 다양화

빌드 변형을 선택한 후 소프트웨어가 수행하는 일을 해당 변형에 적합하게 설정하기 위해 코드의 다양화를 활용한다. 이 코드의 다양화는 빌드 변형과 직접 관련 있는 특정 디렉터리나 파일, 코드 라인을 선택하는 것을 수반하며, 변화의 차이에 따라 수많은 방법으로 코드를 설정할 수 있다.

라인별 변형

이것은 소스코드에 변형을 유도해내는 가장 작은 단위 접근법이다. 이것이 허용되는 언어에서 조건부 컴파일은 각 변형에 적합한 각기 다른 구현을 가능케 하는 코드 라인을 명시한다.

첫 단계에서는 빌드 시스템이 컴파일러에 필요한 정의를 건네준다. C,C++에서는 이것은 전처리기 정의 preprocessor definition로 구현된다.

위의 방법은 간단함 때문에 많은 프로그래머가 사용하는 방식이다. 하지만 남용하면 프로그램 전체의 가독성이 안좋아지기 때문에 필요한 부분만 사용하자.

파일별 변형

한 변형의 소스코드가 그 외 변형의 소스코드와 현저히 다르다면 해당 소스코드를 파일별로 분리하는 것이 일목요연하다고 생각할지도 모른다. 예를 들어 근본적으로 동일한 기능을 수행하되 영어를 지원하는 함수를 포함한 소스 파일 english.c와 마찬가지로 독일어를 지원하는 함수를 포함하는 소스 파일을 가질 수 있다. 이방법은 동일한 파일 안에서 #ifdef 지시어를 사용해 여러 변형들이 섞여 있는 것보다 개발자가 소스코드의 구조를 파악하기 쉽게 해준다. 소스 파일을 조건부 컴파일하기 위해 다음과 같이 build descriptiond을 수정한다.

디렉터리별 변형

이 방법은 각 소스 파일 대신 하위 디렉터리 전체를 포함한다. 단순화 하기 위해 변형 뒤에 하위 디렉터리를 지정하는 일은 빌드 기술을 더욱 간결하게 한다.

이 방법은 라인별 조건부 컴파일이 지원되지 않는 자바와 같은 언어에서 널리 사용된다. 대신 각 변형은 프로그램으로 컴파일될 클래스를 포함하는 자신만의 고유한 하위 디렉터리를 가진다. 모든 하위 디렉터리에는 동일한 자바 소스 파일들이 있지만 각기 다르게 구현된다.

빌드 기술 파일별 변형

각 빌드 변형이 서로 다른 컴파일 플래그로 연계된다면 우리는 빌드 변형별로 각기 다른 빌드 기술 파일을 작성하려고 고려할지도 모른다. 최상위 빌드 기술 파일은 사용자가 설정한 변형별 기술 파일을 하나 이상 포함한다.

빌드 기술 파일을 여러 파일로 세분화하는 것은 새로운 변형에 대한 지원을 쉽게 하며, 주 빌드 기술 파일의 복잡성을 줄여준다. 반면 빌드 기술 파일이 가능한 모든 변형을 처리하기 위해 if,else문으로 난잡하게 작성된다면, 혼란스러워 질 수 잇다.

패키지 시 변형

변형이 적용될 수 있는 단계는 패키지를 만들 때다. 소프트웨어의 특정 에디션을 패키지화하기 위해 우리는 최종 릴리스 패키지에 어떤 파일을 복사할지 선택한다.

설치 시 변형

제품을 빌드하는 방법이 오직 하나 뿐일지라도 설치 시에 소프트웨어의 동작을 수정할 수 있다. 소프트웨어는 타겟 머신에 관련된 파일을 설치하기 전에 사용자의 지리적인 위치를 식별

한다. 또한 릴리스 패캐지는 모든 변형을 지원하는 데 필요한 모든 파일을 포함하지만, 선택된 변형에 관련된 파일만 설치한다.

런타임 시 변형

프로그램이 실행할 때 소프트웨어를 마지막으로 수정할 수 있다. 빌드 시스템은 모든 기능성을 포함하는 릴리스 패키지를 생성하고, 그 기능성 전체를 타겟 머신에 설치한다. 그렇지만 프로그램은 자신이 실행 할 때 어느 변형이 필요한지 결정하고, 상황에 맞게 자신의 동작을 변경한다.

변형을 제어하는 방법으로 사용자가 원하는 언어와 기능들을 선택할 수 있는 Tools menu나 option 메뉴를 이용하는 것을 들 수 있다.

소프트웨어 변형을 구현하는 라인별 변형이나 파일별 변형과 같은 방법은 결코 상호 베타적이지 않다. 따라서 빌드 시스템은 이치에만 맞는다면 이 방법들을 어떤 조합으로도 자유로이 사용 가능하다.

서로 다른 타겟 아키텍처 빌드

빌드 프로세스의 결과물을 다양케 하는 세 번째 방법은 하나 이상의 타겟 아키텍처를 대상으로 코드를 작성하는 일이며, 이는 소프트웨어가 여러 CPU와 운영체제를 지원하는 점을 암시한다. 일반적으로 프로그램의 기능은 어느 경우에도 동일하지만 타겟 컴퓨터는 그렇지 않다. native code로 컴파일 하는 언어로 프로그래밍 할 대는 변형이 관련되지만, 하드웨어와 독립적인 가상머신을 사용하는 자바와 C#은 그렇지 않다.

다중 컴파일러

타겟 아키텍처의 다양화를 위한 첫 번째 중요한 기술은 소스코드 컴파일러를 여러개 사용하는 일이다. 예를 들어 여러분의 제품이 리눅스 환경과 윈도우 환경 둘 다를 대상으로 한다면 리눅스용 코드를 만들기 위해서는 GNU 컴파일러를, 윈도우용 코드를 만들기 위해서는 비주얼 스튜디오 컴파일러를 사용한다. 각 컴파일러는 자신만의 커맨드라인 옵션을 요구하지만 다음과 같은 방법으로 처리할 수 있다.

```
1 ifeq ($(TARGET), Linux)
2     CC := gcc-4.2
3     CFLAGS := -g -o
4 endif
5 ifeq($(TARGET), Windows)
6     CC := cl.exe
7     CFLAGS := /02 /Zi
8 endif
```

위에서 TARGET변수를 HOST로만 고쳐준다면 운영체제를 참조하게 된다.

이런 자동 검출은 개발자가 TARGET=값 을 일일이 기술할 필요가 없으므로 네이티브 컴파일에서 바람직한 방법이라고 할 수 있다.

반면에 크로스 컴파일은 상황이 많이 달라진다. 단독 빌드 머신이 하나 이상의 플랫폼을 대상으로 코드를 생성하는 데 사용될 수 있으므로 개발자는 그들이 원하는 변형을 명시해야 한다.

플랫폼에 특성화된 파일이나 함수

타겟 아키텍처의 다양화를 위한 두 번째 중요한 기술은 모든 소스코드가 모든 플랫폼에 반드시 관련돼 있지 않다는 점이다. 하지만 여러분은 모드 머신상에서 실행되는 소스코드를 작성하려고 시도하지만, 코드는 결국 운영체제에 특성화된 기능을 사용할 수 밖에 없다.

실제로 모든 타겟 머신상에서 동일한 표준 라이브러리에 의존하지 않는 한 상당히 많은 조건부 컴파일을 해야 한다. 따라서 다음과 같은 방법이 적절하다.

- C, C++에서의 #ifdef와 같은 라인별 조건부 컴파일을 사용한다.
- 특정 아키텍처에 관련된 소스코드를 선택하는 파일별 변형을 사용한다.
- 특정 아키텍처에 관련된 소스코드 디렉터리를 선택하는 디렉터리별 변형을 사용한다.

다중 오브젝트 트리

하나 이상의 운영체제와 CPU를 대상으로 코드를 생성한다면 그와 동시에 여러 변형의 오브젝트 코드를 원할지도 모른다. 이는 여러 타겟에서 신중히 검증돼야 할 소스코드의 일부를 수정할 때 특히 유용하다. 어느 한 아키텍처에서는 잘 동작하지만, 그 외에서는 동작하지 않을지도 모르기 때문이다.

오직 하나의 오브젝트 트리만 있다면 그 외의 변형에서 코드를 검증할 때마다 전체 트리를 재빌드해야 한다. 이는 개발자의 생산성을 저하 시키고 개발자가 모든 타겟 머신상에서 검증하는 일을 철저하게 수행하지 않는다면 코드의 정합성이 유지되기 어렵다.