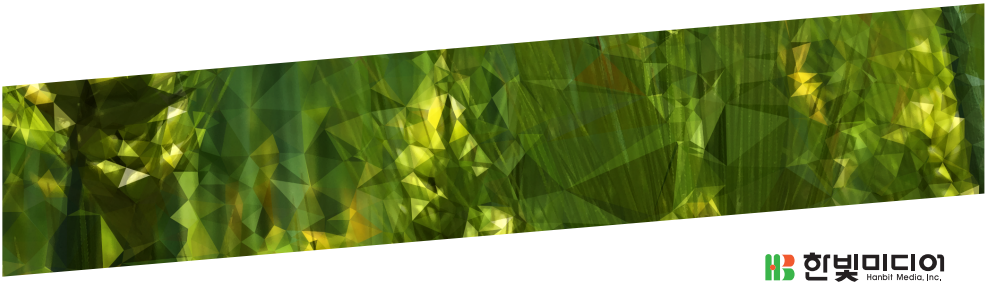
Hanbit RealTime 104

엔터프라이즈 빌드 자동화를 위한 Gradle

윤석진 지음



한빛미디어 Hanbit Media, Inc.

엔터프라이즈 빌드 자동화를 위한 Gradle

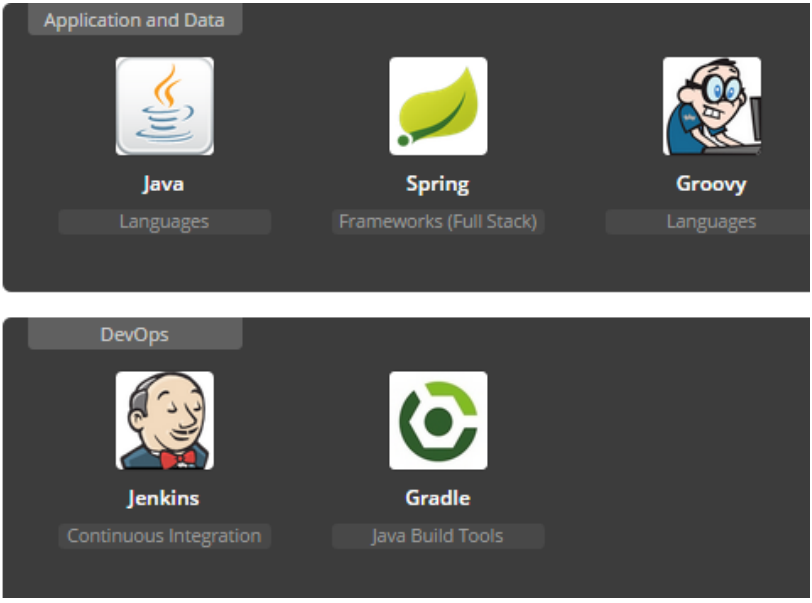
윤석진 지음

한빛미디어 Hanbit Media, Inc.

저자 소개

지은이_ **윤석진**

The_Coding_Live(https://www.facebook.com/thecodinglive) 페이스북 페이지를 운영하고 있으며 백오피스 구축과 프레임워크 설계 및 개발에 관심이 있습니다.



Tags: #로맨틱, #몰입하는습관, #추진력

개념의 실체화 - 좋은 옷인지 알려면 옷을 입어봐야 합니다.

새로운 기술에는 그 기술을 만든 사람이 겪었던 문제들을 해결하려는 마음 그리고 그 기술을 지탱하는 개념이 담겨있습니다. 옷을 입듯이 개념을 입어볼 수 있다면 얼마나 좋을까요?

이 책은 '배포는 쉬워야 한다'는 하나의 명제를 가지고 시작했습니다. 이 명제가 참이 되려면 반드시 수반되어야 하는 과정이 있습니다. Gradle을 중심으로 테스트 자동화, BDD, 통합 테스트, CI 도구들을 이용해 그 명제가 참에 이르는 과정을 함께 공유하고자 합니다.

옷을 입듯이 개념을 입어볼 수는 없지만, 이 책에 담긴 배포 과정의 공유를 통해 개발 도구에서 개인 PC로, 개인 PC에서 서버로의 코드 여정을 투명하게 체계화하려는 분들의 시간이 조금이나마 절약되길 희망합니다.

대상 독자 및 참고사항

이 책은 자바로 웹 개발을 하는 분 중 테스트 자동화에서 배포에 이르기까지 개발의 전 과정을 체계적으로 학습하고 싶은 분을 대상으로 합니다.

도서 구성 방식

이 책은 크게 세 부분으로 나눌 수 있습니다.

빌드 개념에서는 자동화 빌드 도구, 단위 테스트, BDD, 통합 테스트 등의 일반적인 개념을 다룹니다. 그리고 관련 도구 활용에서는 Jenkins, Spock, Junit, Geb와 같은 개념들을 Gradle과 함께 실제로 적용하고 사용하는 데 필요한 도구들을 설명합니다.

이 책의 예제는 Gradle 2.3, Java 8을 기준으로 작성되었으며, Gradle은 Java 6 이상이면 사용할 수 있습니다.

이 책의 예제는 다음 링크에서 내려받을 수 있습니다.

- https://github.com/thecodinglive/hanbit-gradle-book-example

한빛 리얼타임

한빛 리얼타임은 IT 개발자를 위한 eBook입니다.

요즘 IT 업계에는 하루가 멀다 하고 수많은 기술이 나타나고 사라져 갑니다. 인터넷을 아무리 뒤져도 조금이나마 정리된 정보를 찾기도 쉽지 않습니다. 또한, 잘 정리되어 책으로 나오기까지는 오랜 시간이 걸립니다. 어떻게 하면 조금이라도 더 유용한 정보를 빠르게 얻을 수 있을까요? 어떻게 하면 남보다 조금 더 빨리 경험하고 습득한 지식을 공유하고 발전시켜 나갈 수 있을까요? 세상에는 수많은 종이책이 있습니다. 그리고 그 종이책을 그대로 옮긴 전자책도 많습니다. 전자책에는 전자책에 적합한 콘텐츠와 전자책의 특성을 살린 형식이 있다고 생각합니다.

한빛이 지금 생각하고 추구하는, 개발자를 위한 리얼타임 전자책은 이렇습니다.

1 eBook First – 빠르게 변화하는 IT 기술에 대해 핵심적인 정보를 신속하게 제공합니다

500페이지 가까운 분량의 잘 정리된 도서(종이책)가 아니라, 핵심적인 내용을 빠르게 전달하기 위해 조금은 거칠지만 100페이지 내외의 전자책 전용으로 개발한 서비스입니다. 독자에게는 새로운 정보를 빨리 얻을 기회가 되고, 자신이 먼저 경험한 지식과 정보를 책으로 펴내고 싶지만 너무 바빠서 엄두를 못 내는 선배, 전문가, 고수 분에게는 좀 더 쉽게 집필할 수 있는 기회가 될 수 있으리라 생각합니다. 또한, 새로운 정보와 지식을 빠르게 전달하기 위해 O'Reilly의 전자책 번역 서비스도 하고 있습니다.

2 무료로 업데이트되는 전자책 전용 서비스입니다

종이책으로는 기술의 변화 속도를 따라잡기가 쉽지 않습니다. 책이 일정 분량 이상으로 집필되고 정리되어 나오는 동안 기술은 이미 변해 있습니다. 전자책으로 출간된 이후에도 버전 업을 통해 중요한 기술적 변화가 있거나 저자(역자)와 독자가 소통하면서 보완하여 발전된 노하우가 정리되면 구매하신 분께 무료로 업데이트해 드립니다.

3 독자의 편의를 위해 DRM-Free로 제공합니다

구매한 전자책을 다양한 IT 기기에서 자유롭게 활용할 수 있도록 DRM-Free PDF 포맷으로 제공합니다. 이는 독자 여러분과 한빛이 생각하고 추구하는 전자책을 만들어 나가기 위해 독자 여러분이 언제 어디서 어떤 기기를 사용하더라도 편리하게 전자책을 볼 수 있도록 하기 위함입니다.

4 전자책 환경을 고려한 최적의 형태와 디자인에 담고자 노력했습니다

종이책을 그대로 옮겨 놓아 가독성이 떨어지고 읽기 어려운 전자책이 아니라, 전자책의 환경에 가능한 한 최적화하여 쾌적한 경험을 드리고자 합니다. 링크 등의 기능을 적극적으로 이용할 수 있음은 물론이고 글자 크기나 행간, 여백 등을 전자책에 가장 최적화된 형태로 새롭게 디자인하였습니다.

앞으로도 독자 여러분의 충고에 귀 기울이며 지속해서 발전시켜 나가도록 하겠습니다.

차례

chapter 1

그래들 소개

1.1 빌드의 중요성

회사에서는 업무 계획을 세우고 계획을 세분화해 주간, 월간으로 구분한 계획표를 만듭니다. 업무는 혼자 처리할 수 있는 일도 있지만 업무 분담 후 병합이 필요한 일도 있습니다. 분담해서 처리한 일은 결과를 취합하여 보고서를 작성하고, 때로는 결과 내용을 발표하기도 합니다.

그림 1-1 계획 세우기

개발 과정도 일반 회사의 업무 프로세스와 다르지 않습니다. 코드를 작성하고 결과물을 검증하기 위해 테스트하고, 실행되는 모습을 보여주기 위해 배포하며, 유

지보수를 위한 문서화 작업을 합니다. 이러한 일련의 과정을 빌드라고 합니다.

NOTE

빌드란 개발한 소프트웨어가 제품으로 만들어지는 일련의 과정으로 컴파일, 테스트, 배포, 문서화 등의 작업을 수반하는 절차입니다.

1.2 빌드의 자동화

기업의 프로젝트는 혼자서 모든 것을 처리하기에는 규모가 커서 협업이 동반되어야 합니다. 이때 빌드의 모든 과정을 자동으로 처리할 수 있는 도구가 필요합니다.

빌드에서 자동 처리, 즉 자동화란 무엇일까요? 『실용주의 프로그래머를 위한 프로젝트 자동화』(인사이트, 2005)에서는 자동화를 다음과 같이 구분합니다.

그림 1-2 빌드 자동화의 유형

지시 자동화

CI[01] 서버를 구축하고 솔루션을 도입해야만 빌드 시스템을 사용할 수 있는 것은 아닙니다. 그 시작은 지시 자동화를 가능하게 하는 것이라고 말할 수 있습니다.

할당문Assignment `int a = 5;`로 변수 a에 값을 할당한다면 누가 작성해도 모두 동일한 결과를 얻을 수 있습니다. 이처럼 빌드도 실행문을 통해 동일한 결과를 얻을 수 있습니다. 이를 '지시 자동화'라고 합니다.

01 Continuous Integration : 지속적인 통합

많은 사람들이 이클립스나 인텔리제이와 같은 IDE를 사용합니다. 설정에 따라서 자동으로 컴파일할 수도 있고 아이콘 클릭 한 번으로 테스트하거나 결과를 확인할 수도 있습니다. 하지만 이렇게 개인 개발 도구 안에서 빌드 작업을 수행한다면 협업 시 많은 문제가 있습니다. 우선 다른 사람이 빌드하기 위해서는 서로 환경이 동일해야 합니다. 이것은 마치 파일 입출력이나 환경설정을 읽는 코드를 작성할 때 절대 경로를 하드코딩해서 사용하는 것과 같습니다. 가능하긴 하지만, 조그만 변경이 생겨도 오류가 날 가능성이 많습니다.

이런 상황을 개선하기 위해 스크립트로 빌드할 수 있습니다. 스크립트만 실행하면 누구나 동일하게 빌드할 수 있습니다. 이 책에서는 지시 자동화를 좀 더 편리하고 유연하게 하기 위해 그래들Gradle을 사용합니다.

예약 자동화

결과물을 만드는 과정을 빌드를 통해서 지시 자동화를 했다면 다음 단계는 예약 자동화입니다. 필요나 요구에 따라서 버그픽스 버전을 만들 수도 있고, 제품을 매주 목요일마다 릴리스할 수도 있습니다. 이러한 요구사항을 해결하기 위해서 8장에서 CI 도구인 젠킨스Jenkins와 연동한 후 원하는 시점을 예약해 빌드하는 법을 다룹니다.

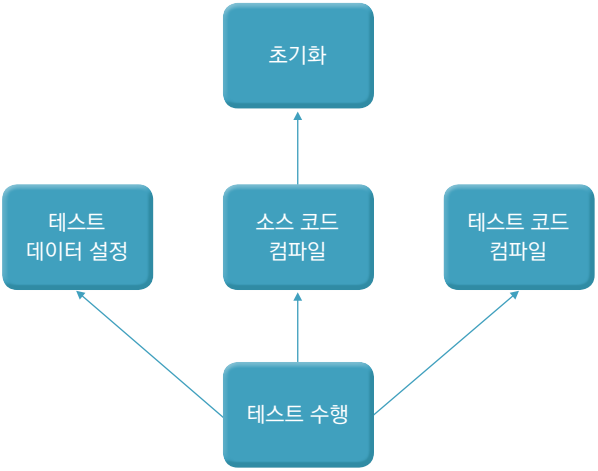
유발 자동화

솔루션을 만들다 보면 하나의 프로젝트가 아니라 여러 프로젝트를 묶어서 한 솔루션으로 관리하게 됩니다. 사용자의 작업에 따라 브랜치에서 체크아웃한 후에 빌드를 실행할 필요도 있습니다. 이렇게 사용자의 액션에 따라서 빌드하는 것이 유발 자동화입니다. 유발 자동화 역시 8장에서 다룹니다.

빌드 자동화와 의존성 네트워크

대부분 빌드 도구는 [그림 1-3]처럼 각 빌드 절차를 수행하기 위한 의존관계[02]가 있는데, 이러한 의존관계를 의존성 네트워크라고 부릅니다. 의존성 네트워크는 컴퓨팅 대안 모델의 하나로 대다수 빌드 도구의 근간이 되는 개념입니다. 트리에서 단계별로 순회 시 낮은 단계부터 차례대로 순회하는 것처럼 의존성 네트워크 모델을 사용하면 사람이 수동으로 실행하지 않아도 관계를 파악해서 테스트를 실행할 때 소스 코드 컴파일과 테스트 코드 컴파일을 먼저 실행합니다.

그림 1-3 빌드 프로세스의 의존관계[03]



1.3 그래들의 개념과 특성

그래들은 Gradle[04] 사에서 개발을 주도하는 자동화 빌드 도구로, 그루비를 기반

02 참고 : 제즈 험블,데이비드 팔리, 『신뢰할 수 있는 소프트웨어 출시』, 유석문, 김은하, 설현준 함께 옮김(에이콘 출판사, 2013)

03 참고: 마이크 클라크, 『실용주의 프로그래머를 위한 프로젝트 자동화』, 김정민, 김정훈 함께 옮김(인사이트, 2005)

04 https://gradle.org/company/

으로 하는 프로그래밍 방식과 다양한 플러그인을 지원하는 하나의 빌드 플랫폼입니다. 그래들은 자바 외에도 스칼라Scala, 그루비Groovy, C, C++ 등 다양한 언어를 지원합니다.

그래들 공식 홈페이지[05]에 따르면 앤트Ant의 유연함과 메이븐Maven 규약의 특성을 지닌, 의존성 관리가 되는 자동화 빌드 도구로 그래들을 소개하고 있습니다. 앤트는 Task 중심의 빌드 도구로, Task를 자바로 작성하고 이를 확장해 XML에서도 사용할 수 있어서 Task 확장성을 통한 유연함을 제공합니다. 그래들도 앤트처럼 Task 개념이 있고 메이븐처럼 플러그인을 이용해 새로운 작업을 수행할 수 있습니다. 간단히 표현하면, 그래들은 차세대 빌드 도구로 메이븐과 앤트의 장점을 더한 도구입니다.

그래들은 유연함을 극대화하기 위해 XML이 아닌 그루비로 작성되었습니다. 기존에는 같은 내용을 XML로 반복하여 작성했지만, 그래들은 반복문을 통해 좀 더 간결하게 작성할 수 있습니다. 그런데 그래들을 사용하기 위해 그루비를 능숙하게 다룰 필요는 없습니다. 그래들은 빌드 스크립트 언어Build Script Language이자 DSLDomain Specific Language[06]이기 때문입니다.

CSS를 사용하면 선언에 따라서 엘리먼트Element의 색상, 크기와 위치 등 웹 페이지의 전반적인 레이아웃을 관리할 수 있습니다. 그래들 또한 선언에 따라서 프로젝트를 관리할 수 있도록 해줍니다.

이 책에서는 특정 개발 도구에 종속되지 않으면서 그루비를 익히지 않더라도 프로젝트 레이아웃을 만들고 프로젝트를 관리할 수 있는 방법을 다루겠습니다.

---

05 https://gradle.org/

06 DSL이란 특정한 도메인(분야, 영역)에서 발생하는 문제점을 해결하기 위해 도메인을 기준으로 모든 것을 해결하려고 제공되는 언어를 말한다. (참고 : http://martinfowler.com/bliki/DomainSpecificLanguage.html, http://gradle.org/docs/current/dsl/)

1.4 그래들 설치

그래들은 JVM 기반의 도구이므로 자바가 먼저 설치되어 있어야 합니다. 자바가 이미 설치되어 있다면 각 운영체제의 패키지 매니저 도구를 이용해 간편하게 설치할 수 있습니다.

맥 OS

```
# sudo brew install gradle
```

윈도우

윈도우에서는 Chocolatey[07]라는 오픈소스를 이용해서 설치할 수 있습니다. 먼저 다음 스크립트를 실행해 Chocolatey를 설치합니다.

```
# @powershell -NoProfile -ExecutionPolicy unrestricted -Command "iex ((new-
object net.webclient).DownloadString('https://chocolatey.org/install.ps1'))" &&
SET PATH=%PATH%;%ALLUSERSPROFILE%\chocolatey\bin
```

설치가 완료되면 brew처럼 `install` 명령으로 그래들을 설치합니다.

```
# choco install gradle
```

리눅스

데비안 계열(Ubuntu, CrunchBang, MintLinux)에서는 다음 명령으로 설치합니다.

```
# sudo apt-get install gradle
```

07 윈도우용 패키지 관리 도구, 홈페이지 https://chocolatey.org/

레드햇 계열(centOS, Fedora)에서는 다음과 같이 설치합니다.

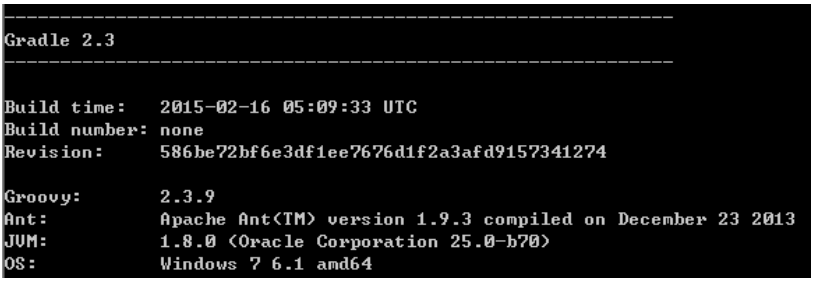
```
# sudo yum install gradle
```

그래들 설치가 끝나면 다음 명령으로 그래들의 버전을 확인합니다.

```
# gradle --version
```

명령을 실행하면 [그림 1-4]와 같이 그래들 버전 정보가 표시됩니다.

그림 1-4 그래들 버전 확인 결과



지금까지 자동화 빌드 도구의 기본적인 특성과 그래들의 특징을 살펴보았습니다. 다음 장에서는 그래들의 기본 Task를 사용하여 프로젝트를 생성하는 방법을 알아보겠습니다.

chapter 3

그래들로 자바 프로젝트 만들기

3.1 프로젝트 초기화

이번에는 그래들의 Java 플러그인[01]으로 자바 프로젝트를 구성하는 방법에 대해 알아보겠습니다. 예제에서는 초기화 명령을 사용하지만 이미 build.gradle 파일이 있는 경우에는 다음과 같이 플러그인 설정만 해도 됩니다.

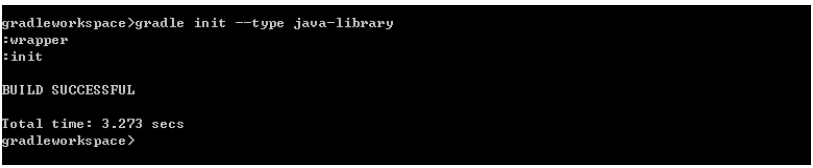
```
apply plugin : 'java'
```

`init` 명령에 `java-library` 타입으로 새로운 프로젝트를 생성합니다.

[프로젝트 초기화]

```
gradle init --type java-library
```

그림 3-1 Java-library 타입으로 프로젝트 초기화

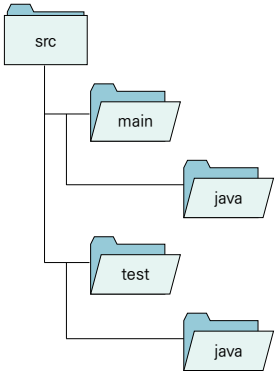


이렇게 새로운 프로젝트를 생성하면 src 디렉터리 구조는 [그림 3-2]가 됩니다. 이 구조는 기본 규격으로 변경할 수 있으며, 소스 파일은 /main/java 디렉터리

01 http://gradle.org/docs/current/userguide/java_plugin.html

에, 테스트 파일은 /test/java 디렉터리에 작성합니다.

그림 3-2 Java-library 타입의 기본 디렉터리 구조



Java 플러그인을 사용하면 자바 관련 Task들을 사용할 수 있으며, 추가된 자바 Task를 확인하려면 tasks를 실행합니다.

```
gradle tasks
```

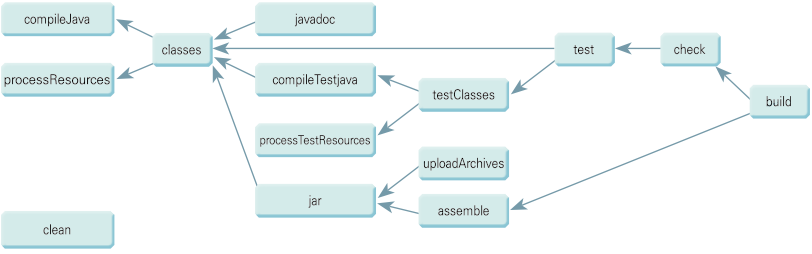
그림 3-3 자바 플러그인 추가 후 Task 목록

```
gradleworkspace>gradle tasks
:tasks

------------------------------------------------------------
All tasks runnable from root project
------------------------------------------------------------

Build tasks
-----------
assemble - Assembles the outputs of this project.
build - Assembles and tests this project.
buildDependents - Assembles and tests this project and all projects that depend on it.
buildNeeded - Assembles and tests this project and all projects it depends on.
classes - Assembles classes 'main'.
clean - Deletes the build directory.
jar - Assembles a jar archive containing the main classes.
testClasses - Assembles classes 'test'.

Build Setup tasks
-----------------
init - Initializes a new Gradle build. [incubating]
wrapper - Generates Gradle wrapper files. [incubating]

Documentation tasks
-------------------
javadoc - Generates Javadoc API documentation for the main source code.

Help tasks
----------
components - Displays the components produced by root project 'ch02-java'. [incubating]
dependencies - Displays all dependencies declared in root project 'ch02-java'.
dependencyInsight - Displays the insight into a specific dependency in root project 'ch02-java'.
```

```
help - Displays a help message.
projects - Displays the sub-projects of root project 'ch02-java'.
properties - Displays the properties of root project 'ch02-java'.
tasks - Displays the tasks runnable from root project 'ch02-java'.

Verification tasks
------------------
check - Runs all checks.
test - Runs the unit tests.

Rules
-----
Pattern: clean<TaskName>: Cleans the output files of a task.
Pattern: build<ConfigurationName>: Assembles the artifacts of a configuration.
Pattern: upload<ConfigurationName>: Assembles and uploads the artifacts belonging to a configuration.

To see all tasks and more detail, run gradle tasks --all

To see more detail about a task, run gradle help --task <task>

BUILD SUCCESSFUL

Total time: 2.802 secs
gradleworkspace>
```

assemble, build, clean, jar 등 빌드 관련 Task가 추가된 것을 확인할 수 있습니다. 이 Task들을 도식화하면 다음 그림과 같습니다.

그림 3-4 자바 플러그인의 Task 의존관계



출처: Gradle.org (http://gradle.org/docs/current/userguide/java_plugin.html)

좌측에 compileJava가 있고 우측은 build가 있습니다. build Task를 실행하면 의존관계가 있는 compileJava와 test Task가 실행됩니다. 그런데 이전 단계 Task가 실패하면 다음 단계는 진행되지 않습니다.

최근 이클립스나 인텔리제이와 같은 도구에서는 저장할 때 자동으로 컴파일이 수행되고 WAR 파일이나 JAR 파일 생성도 지원하므로 반드시 그래들의 모든 Task를 다 알고 사용할 필요는 없습니다. 다만, 각 Task 간의 의존관계를 파악한다면 나중에 빌드 문제가 발생할 때 효율적으로 문제를 해결할 수 있습니다.

3.2 레이아웃 구성하기

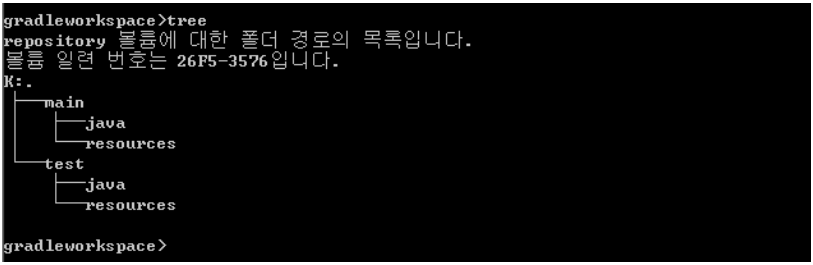
클래스는 아니지만 로그 설정이나 기타 다른 라이브러리 설정 파일이 클래스패스에 로드되어야 하는 경우에 다음과 같이 Task로 별도의 resources 디렉터리를 만들 수 있습니다.

다음 내용을 build.gradle 파일에 추가하고 `gradle makeResourceFolder`를 실행합니다.

```
task makeResourceFolder << {
    sourceSets*.resources.srcDirs*.each {it.mkdirs()}
}
```

Task가 정상적으로 실행되면 [그림 3-5]와 같이 resources 디렉터리가 test 디렉터리와 main 디렉터리에 생성됩니다.

그림 3-5 makeResourceFolder Task 실행 결과



`gradle init` 명령을 사용하지 않고 build.gradle 파일만 공유하여 사용하는 경우도 있을 것입니다. 이럴 때 한번에 소스 디렉터리와 리소스 디렉터리가 생성되도록 `makeResourceFolder` Task를 수정하겠습니다.

`makeResourceFolder` Task에 `java`를 추가하여 새로운 Task를 만듭니다. 여기서 `sourceSets`는 디렉터리 전체를 의미하며, 실제로 `sourceSets`는 그래들 내부에서 인터페이스로 정의되어 있습니다. Java 플러그인을 사용할 때 `java`의 소스 디

렉터리와 테스트 디렉터리를 논리적으로 표현하는 단위가 sourceSets입니다.

```
task initJavaFolder << {
    sourceSets*.java.srcDirs*.each { it.mkdirs() }
    sourceSets*.resources.srcDirs*.each {it.mkdirs() }
}
```

그런데 기존 프로젝트의 디렉터리 위치가 그래들과는 다른 경우도 있어서 소스 디렉터리가 두 개가 되거나 별도의 디렉터리도 함께 지정해야 하는 경우가 있습니다. 이런 경우에는 sourceSets 속성 하위에 디렉터리를 추가합니다. 예를 들어, 자바 프로젝트의 소스 디렉터리 역할을 하는 디렉터리명이 'mysrc'라면 다음과 같이 추가하면 됩니다. build.gradle 파일에 이 내용을 추가한 후 initJava Folder Task를 실행합니다.

```
sourceSets {
    main{
        java{
            srcDirs = ['src', 'mysrc']
        }
    }
}
```

추가한 디렉터리를 그래들에서 잘 인식하는지 확인해 보겠습니다. 다음 내용의 printJavafolder Task를 build.gradle 파일에 추가하고 실행합니다.

```
task printJavafolder << {
    sourceSets {
        main{
            println "java.srcDirs = ${java.srcDirs}"
            println "resources.srcDirs = ${resources.srcDirs}"
        }
    }
}
```

srcDirs 디렉터리에 추가한 mysrc 디렉터리까지 함께 출력됩니다.

```
:printJavafolder
java.srcDirs = [K:\99.work\gradlesource\gradlejava\src, K:\99.work\gradlesource
gradlejava\mysrc]
resources.srcDirs = [K:\99.work\gradlesource\gradlejava\src\main\resources]

BUILD SUCCESSFUL

Total time: 3.951 secs
```

소스 디렉터리를 추가하는 것 외에도 test 디렉터리를 추가하거나 변경하는 것도 동일한 방법으로 처리할 수 있습니다. test 디렉터리를 추가하려면 다음과 같이 sourceSets 구문에 test를 추가하면 됩니다.

```
sourceSets{
    main{
        java{
            srcDirs = ['src', 'mysrc']
        }
    }
    test{
        java{
            srcDirs = ['test', 'mytest']
        }
    }
}
```

지금까지 다룬 자바 프로젝트의 build.gradle 파일 전체 내용은 다음과 같습니다.

[makeResource/build.gradle]

```
apply plugin: 'java'

sourceCompatibility = 1.8
targetCompatibility = 1.8

repositories {
    jcenter()
```

```
}
buildDir = 'build'

task makeResourceFolder << {
    sourceSets*.resources.srcDirs*.each {it.mkdirs()}
}

task initJavaFolder << {
    sourceSets*.java.srcDirs*.each { it.mkdirs() }
    sourceSets*.resources.srcDirs*.each {it.mkdirs() }
}

sourceSets {
    main {
        java {
            srcDirs = ['src', 'mysrc']
        }
    }

    test {
        java {
            srcDirs = ['test', 'mytest']
        }
    }
}

task printJavafolder << {
    sourceSets {
        main {
            println "java.srcDirs = ${java.srcDirs}"
            println "resources.srcDirs = ${resources.srcDirs}"
        }
    }
}

dependencies {
    compile 'org.slf4j:slf4j-api:1.7.5'
    testCompile 'junit:junit:4.11'
}
```

이제 간단한 자바 클래스를 만들어서 컴파일하겠습니다. 다음과 같이 First Gradle.java 파일을 작성합니다.

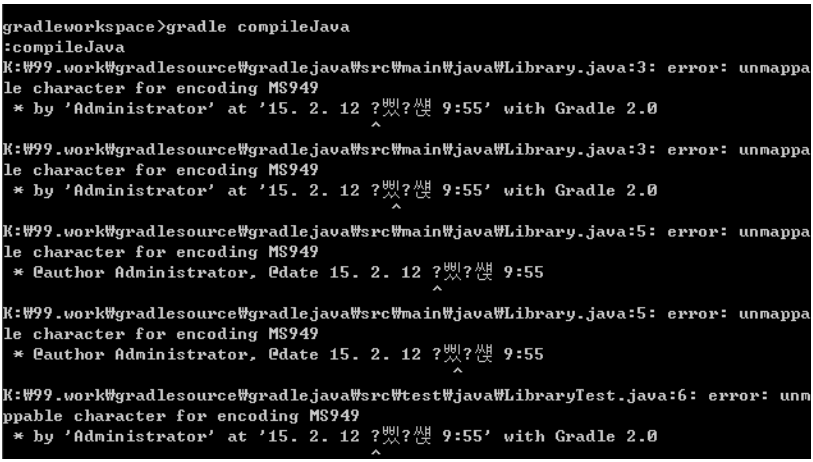
[makeResource/src/main/java/FirstGradle.java]

```
public class FirstGradle {
```

```
    public FirstGradle() {
    }

    public static void main(String ar[]) {
        FirstGradle gradle = new FirstGradle();
        System.out.println( "Hello Gradle with java" );
    }
}
```

내용은 최대한 단순하게 'Hello Gradle with Java'를 출력하도록 작성하였습니다. 이클립스나 인텔리제이와 같은 도구들은 저장 시 자동으로 컴파일하는 기능을 제공하기 때문에 자주 사용할 일은 없지만, 배우는 과정이므로 직접 gradle 명령으로 컴파일해 보겠습니다. compileJava Task를 실행하여 컴파일합니다.

그림 3-6 compileJava 실행 시 인코딩 오류



윈도우에서는 인코딩이 'MS949'로 지정되어 있어서 compileJava Task를 실행할 때 오류가 발생합니다. 인코딩 오류를 방지하기 위해서 build.gradle 파일에서 컴파일할 자바 파일에 대한 인코딩 옵션을 설정할 수 있습니다. 다음처럼 compileJava에 인코딩 옵션을 'UTF-8'로 추가합니다.

```
compileJava.options.encoding = 'UTF-8'
```

또는 gradle.properties 파일에서 그래들의 jvmargs 환경변수로 값을 추가할 수 있습니다.

```
org.gradle.jvmargs=-Dfile.encoding=UTF-8
```

인코딩 옵션을 추가한 후 compileJava를 다시 실행하면 다음과 같은 오류가 발생합니다.

그림 3-7 compileJava 실행 시 Test 라이브러리 의존성 오류

```
not find symbol
    @Test public void testSomeLibraryMethod() {
     ^
  symbol:   class Test
  location: class LibraryTest
K:₩99.work₩gradlesource₩gradlejava₩src₩test₩java₩LibraryTest.java:13: error: ca
not find symbol
        assertTrue("someLibraryMethod should return 'true'", classUnderTest.som
LibraryMethod());
        ^
  symbol:   method assertTrue(String,boolean)
  location: class LibraryTest
4 errors
:compileJava FAILED

FAILURE: Build failed with an exception.

* What went wrong:
Execution failed for task ':compileJava'.
> Compilation failed; see the compiler error output for details.

* Try:
Run with --stacktrace option to get the stack trace. Run with --info or --debu
ption to get more log output.

BUILD FAILED

Total time: 3.789 secs
```

여기서 compileJava를 실행하는 것은 자바 파일을 실행하려는 목적이므로 test를 무시하고 실행해 보겠습니다.

test를 무시하는 방법으로는 test 디렉터리를 삭제하는 방법도 있지만, 경우에 따라서는 test 코드가 작성된 기존 프로젝트에 코드를 추가하여 실행해야 할 수도 있습니다. 예를 들어, 이미 사용하고 있는 라이브러리로 개발을 진행하거나 오픈소스 프로젝트를 받아서 개발할 때 test를 먼저 실행하는 경우가 많습니다. 그런

데 지금처럼 변경 사항이 많지 않을 때 test를 무시하고 실행할 수 있다면, 기존 프로젝트의 설정을 보존하고 시간을 단축할 수 있습니다.

앞에서 프로젝트 레이아웃을 설정할 때 sourceSets 속성을 이용해서 디렉터리를 추가한 것처럼 test를 건너뛰고자 할 때도 마찬가지로 sourceSets에 설정을 추가하여 test를 무시하고 실행할 수 있습니다.

앞에서 오류가 발생한 것은 compileJava에 test 디렉터리가 포함되어 있기 때문이므로 test 실행 시 test 디렉터리에 대해서 제외하게 하는 설정이 필요합니다. 그래서 exclude를 추가합니다.

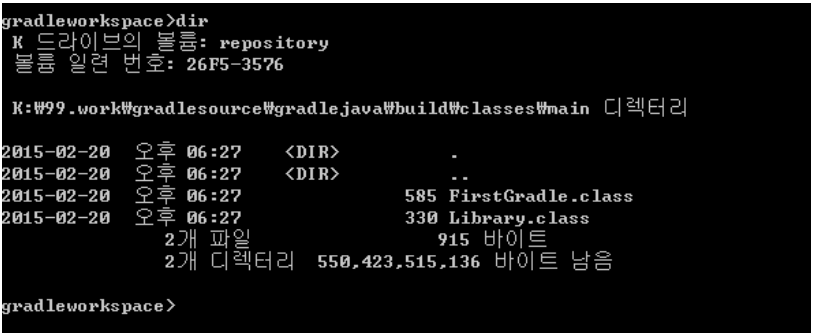
```
sourceSets{
    main{
        java{
            srcDirs = ['src', 'mysrc']
            exclude 'test/*'
        }
    }
}
```

그림 3-8 exclude를 추가한 후 compileJava 실행

```
gradleworkspace>gradle compileJava
:compileJava UP-TO-DATE

BUILD SUCCESSFUL

Total time: 3.719 secs
gradleworkspace>
```

test 디렉터리가 제외되었기 때문에 오류가 발생하지 않고 정상적으로 실행됩니다. 컴파일이 성공적으로 수행되면 클래스 파일은 build 디렉터리에 생성됩니다. build 디렉터리는 그래들에서 컴파일할 때 클래스 파일들이 위치하는 기본 경로로, 컴파일 이전의 디렉터리 구조와 동일하게 생성됩니다.

그림 3-9 compileJava 실행 후 build 디렉터리에 생성된 클래스 파일들



build 디렉터리는 반복해서 파일이 생성되는 디렉터리이므로 이전에 생성된 파일과 중복되지 않도록 파일을 삭제한 후에 컴파일해야 하는 경우가 있습니다. 이럴 때는 clean Task를 사용합니다. clean Task를 사용하면 빌드한 결과물이 생성되는 build 디렉터리와 디렉터리 내용 전체가 삭제됩니다. 또한, clean Task는 다른 Task와 조합해서 사용할 수 있습니다.

3.3 의존성 관리하기

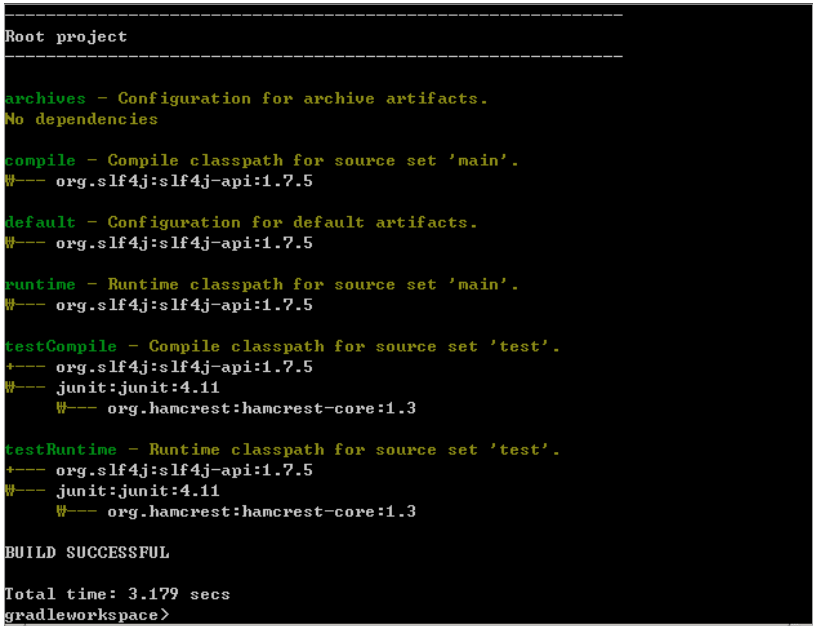
자바는 JVM을 통해 운영체제에 독립적으로 동작할 수 있고, 라이브러리를 만들 때에도 운영체제별로 만들지 않고 자바 런타임에 호환되도록 만들면 되므로 타 언어에 비해서 오픈소스나 라이브러리들이 많은 편입니다. 또한, 요즘 개발 트렌드도 오픈소스들을 조합해서 문제를 해결해 나가는 추세입니다. 그래들에서도 편리하고 쉽게 라이브러리를 관리할 수 있도록 지원합니다.

3.3.1 자바 프로젝트의 라이브러리 스코프

개발을 진행할 때 대표적으로 세 가지의 작업(컴파일, 테스트, 실행)을 합니다. 각 작업에서 사용되는 라이브러리가 매우 다양한다. 그래들에서는 사용하는 라이브러리가 중복되지 않도록 스코프를 지원합니다. 자바 프로젝트에서 지원하는 스코프

는 dependencies 명령으로 확인할 수 있습니다.

그림 3-10 자바 의존성 참조 확인



스코프별로 포함된 라이브러리가 출력되는데, 주로 사용하는 스코프는 compile, testCompile, runtime입니다.

표 3-1 스코프 종류

스코프 설정	관련 Task	설명
compile	compileJava	컴파일 시 포함해야 할 때
runtime	-	실행시점에 포함해야 할 때
testCompile	compileTestJava	테스트를 위해 컴파일할 때 포함해야 할 때
testRuntime	test	테스트를 실행시킬 때

컴파일과 테스트는 구분이 쉽지만, 컴파일과 실행시점(런타임)의 구분은 조금 혼란스러울 수 있습니다. 예를 들어, 서블릿으로 웹 페이지를 작성할 때 실행할 수 있는 클래스를 작성하기 위해서는 HttpServlet 클래스를 상속받아서 자

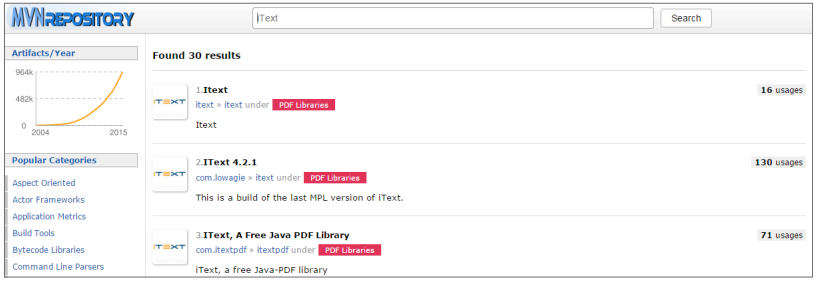
바 코드를 작성해야 합니다. 그리고 HttpServlet 클래스를 상속하기 위해서는 servlet-api.jar 파일이 컴파일 시 포함되어야 합니다. 이렇게 해서 컴파일을 하면 HTTP 요청을 처리할 수 있는 클래스가 준비됩니다.

하지만 웹 페이지에 데이터를 표현하기 위해서 JSTL과 같은 태그 라이브러리를 사용하는 경우에는 실행시점에만 사용해도 무방합니다. JSTL은 서블릿을 생성할 때는 필요하지 않고 서블릿 생성이 끝나고 HTTP 요청을 받아서 JSP에 전달되었을 때 브라우저에서 마크업Markup을 랜더링하면서 태그를 자바의 변수값으로 치환해주기 때문입니다. 이처럼 스코프를 이용하면 라이브러리를 각 단계에 따라 다르게 적용해서 사용할 수 있습니다.

3.3.2 라이브러리 추가

외부 라이브러리를 직접 추가해서 실행하는 프로그램을 작성해 보겠습니다. 사용할 라이브러리를 추가하기 위해서 http://mvnrepository.com/에 접속합니다. 사용할 라이브러리를 검색하면 검색 키워드와 매칭이 되는 라이브러리 목록이 표시됩니다. 여기에서 사용할 'iText'로 검색하면 같은 이름이 여러 개가 있는데 이중에서 'com.itextPdf' 패키지의 라이브러리를 사용하겠습니다.

그림 3-11 http://mvnrepository.com/에서 라이브러리 검색하기



Download(JAR) 버튼을 클릭해서 라이브러리를 직접 다운로드할 수도 있고 도구별로 탭으로 구분해서 골라 사용할 수 있게 되어 있습니다.