09.9주차Plugin

Plugin

Table of Contents

이번 문서에서는 Plugin과 Convention에 대해 살펴보고 사용자 정의 플러그인을 만드는 법을 알아본다.

프로젝트에 Plugin을 적용하는 법

Gradle 플러그인 적용은 Project.apply() 메소드를 통해 이루어진다.
build.gradle에 아래와 같이 작성한다.

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

java는 사실 축약형이다. 실제는 JavaPlugin이 적용된 것이다.
그래서 아래처럼 type으로 적용할 수 있다.

```
1 apply plugin: org.gradle.api.plugins.JavaPlugin
```

패키지명은 생략해도 된다. Gradle이 기본적으로 import를 하고 있기 때문이다.

```
1 apply plugin: JavaPlugin
```

Plugin으로 할 수 있는 일들

Gradle 플러그인을 적용하면 프로젝트에 다음과 같은 기능을 추가하여 확장할 수 있다.

- 프로젝트의 태스크 (compile, test)를 추가
- 유용한 기본값과 함께 태스크를 사전에 구성 (pre-configure)
- 프로젝트의 의존성 설정
- 확장을 통해 기존 타입에 새로운 속성과 메소드 추가

아래 예제는 java 플러그인를 적용해서 프로젝트에 새로운 태스크를 추가한 것이다.

```
apply plugin: 'java'

task show << {
    println relativePath(compileJava.destinationDir)
    println relativePath(processResources.destinationDir)
}
```

gradle -q show 의 결과는 아래와 같다.

```
build/classes/main
build/resources/main
```

java 플러그인이 프로젝트에 compileJava와 processResources 태스크를 추가하였고, 각 태스크가 가지고 있는 destinationDIr 속성을 설정한 것이다.

관례 (Conventions)

gradle은 메이븐처럼 관례에 기반한 프레임워크다. 그러면서도 그 관례를 마음껏 바꿀 수 있는 강력한 기능을 메커니즘을 제공하고 있다.
이번에는 사전에 제공하는 관례를 수정하여 프로젝트를 재구성하는 방법을 살펴보겠다.
위의 Plugin으로 할 수 있는 것들 예제에서 보았듯이 자바 플러그인의 compileJava와 processResources 태스크는 기본적으로 destinationDir 속성을 가지고 있다. 이번에는 그 기본값을 바꿔 본다.

플러그인 기본 속성 값 변경하기

```
apply plugin: 'java'

compileJava.destinationDir = file("$buildDir/output/classes")

task show << {
    println relativePath(compileJava.destinationDir)
}
```

gradle -q show 명령어를 실행해본다.

```
> gradle -q show
build/output/classes
```

단지 속성값을 overwrite하는 방법만으로 기본값이 build/classes/main 에서 build/output/classes 으로 변경되었다.

SourceSet

위 예제를 다시 한 번 살펴보자. compileJava의 destinationDir 속성에 클래스 패스 값을 할당하였다. 그런데 일반적인 프로젝라면 클래스 패스를 제어하는 태스크는 compileJava만 유일한 것이 아닐 것이다. 있다. 만약 compileJava 태스크가 아닌 다른 태스크에서도 클래스 패스를 참조한다면 예상치 못하는 버그가 나타날지도 모른다. 그래서 뭔가 공통적으로 속성 값을 변경할 수 있는 기능이 필요한데 이를 위해 자바 플러그인은 Source Set 이라는 개념을 가지고 있다.

source set은 자바의 source와 resource의 논리적인 그룹 단위이자 소스의 묶음을 나타내는 개념이다.
compileJava의 destinationDir 속성은 Source Set에 기본적으로 매핑되고 있는데, 이것을 이용하여 클래스 패스를 변경해본다.

```
apply plugin: 'java'

sourceSets.main.output.classesDir = file("$buildDir/output/classes")

task show << {
    println relativePath(compileJava.destinationDir)
}
```

```
> gradle -q show
build/output/classes
```

결과를 보면 compileJava.destinationDir의 직접적인 수정없이 기본값이 변경된 것을 확인할 수 있다.

사용자 정의(Custom) 플러그인 만들기

Gradle 플러그인은 빌드 로직의 부분을 패키징해서 다른 여러 프로젝트와 빌드를 같이 사용할 수 있도록 해준다. Gradle 은 자신만의 고유한 플러그인을 구현할 수 있도록 허용하기 때문에 그것을 빌드 로직에 재사용하여 다른 사용자들에게 공유할 수 있다.
커스텀 플러그인은 프로그래밍 언어의 구애 없이 구현할 수 있다. 참고로 여기서는 Groovy로 구현하지만 Java나 Scala 등으로도 얼마든지 가능하다.

플러그인 패키징

플러그인의 소스를 배치하는 방법에는 3가지가 있다.

1.빌드 스크립트

빌드 스크립트 소스 내에 플러그인을 직접 포함시킨다. 이 방식은 별다른 작업없이 자동으로 컴파일되고 빌드 스크립트의 클래스패스에 포함되는 이득이 있지만, 플러그인을 선언한 빌드 스크립트 외부에서는 접근할 수 없다는 단점이 있다.

2.buildSrc 프로젝트

플러그인 소스를 rootProjectDir/buildSrc/src/main/groovy 디렉토리에 배치한다. 그러면 Gradle이 알아서 컴파일, 테스팅해주고 빌드 스크립트의 클래스패스에 추가해준다.
플러그인은 빌드에서 사용되는 모든 빌드스크립트에서 참조 가능하다. 그러나 빌드 외부에서는 참조가 불가능하다.

3.독립형(Standalone) 프로젝트

독립적으로 분리된 프로젝트로 만드는 것이다. 분리된 프로젝트의 archives인 JAR파일를 배포하여 다른 프로젝트에서 공유하는 방법이다. 이렇게 하면 멀티 빌드에서 사용하거나 또는 다른 사용자와 공유할 수 있다.

빌드 스크립트 플러그인

간단한 플러그인 작성해보기

빌드 스크립트 소스 내에 플러그인을 작성해본다.
플러그인을 작성하기 위해서는 Plugin 인터페이스를 구현해서 작성한다. 프로젝트에서 플러그인이 사용될때 Gralde은 플러그인의 인스턴스를 생성한 뒤, 인스턴스의 Plugin.apply() 메소드를 호출한다.
이 때 project 객체는 파라미터로 전달되어 설정하는 데 사용할수 있다.
다음 예제는 GreetingPlugin 이라는 플러그인에 hello 태스크를 추가해본다. build.gradle에 아래와 같이 작성한다.

```
apply plugin: GreetingPlugin
// Plugin 인터페이스를 상속한다.
class GreetingPlugin implements Plugin<Project> {
    // gradle에 의해서 apply 메소드가 호출된다.
    void apply(Project project) {
        // 태스크를 정의한다.
        project.task('hello') << {
            println "Hello from the GreetingPlugin"
        }
    }
}
```

gradle -q hello 를 실행하면 다음과 같이 출력된다.

```
Hello from the GreetingPlugin
```

플러그인이 적용되는 프로젝트마다 새로운 플러그인 인스턴스가 생성되므로 주의한다.

빌드에서 입력값 얻기

대부분의 플러그인은 빌드 스크립트에서 설정을 할 필요가 있다. 이것을 하기 위한 한 가지 방법으로 확장(extension) 객체를 사용한다.
다음 예제는 간단한 확장 객체를 프로젝트에 추가해본다. greeting이라는 확장 객체를 추가해서 설정 값을 수정해본다.

```
apply plugin: GreetingPlugin

// message에 설정 값을 입력받는다.
greeting.message = 'Hi from Gradle'

class GreetingPlugin implements Plugin<Project> {
    void apply(Project project) {
        // greeting 확장 객체를 추가한다.
        project.extensions.create("greeting", GreetingPluginExtension)
        // 입력받은 설정값을 사용하는 태스크를 추가한다.
        project.task('hello') << {
            println project.greeting.message
        }
    }
}

// POGO 를 정의한다.
class GreetingPluginExtension {
    def String message = 'Hello from GreetingPlugin'
}
```

gradle -q hello 를 실행해보면 다음과 같다.

```
Hi from Gradle
```

이 예제에서 GreetingPluginExtension 는 message라는 필드를 지닌 POGO (Plan Old Groovy Object)이다. 이것을 확장객체로 사용하기 위해 greeting이라는 이름으로 플러그인 리스트에 추가하였다.
종종 하나의 플러그인에 몇가지의 관련 속성을 지정해야 하는 경우가 있다. Gradle는 각각의 확장 객체에 구성(configuration) 클로저 블록을 추가하기 때문에 설정을 한꺼번에 할 수 있도록 해준다.
다음은 설정 클로저를 사용한 커스텀 플러그인 예제이다.

```
apply plugin: GreetingPlugin

// 속성 값을 greeting의 configuration 클로저로 묶어준다.
greeting {
    message = 'Hi'
    greeter = 'Gradle'
}

class GreetingPlugin implements Plugin<Project> {
    void apply(Project project) {
        // POGO를 확장 객체 등록
        project.extensions.create("greeting", GreetingPluginExtension)
        // 태스크 정의
        project.task('hello') << {
            // 확장 객체 이름(greeting) 으로 설정값을 접근할 수 있다.
            println "${project.greeting.message} from ${project.greeting.greeter}"
        }
    }
}

// POGO 정의
class GreetingPluginExtension {
    String message
    String greeter
}
```

gradle -q hello 를 실행해보면 다음과 같다.

```
Hi from Gradle
```

이 예제에서는 속성 값을 클로저를 통해 그룹화해서 구성하였다. 실제 값들을 접근할 때는 블록 이름(greeting)과 extention(확장 객체 이름)이 동일해야 한다. 클로저가 실행되면 확장 객체의 필드는 그루비의 표준 위임 기능에 따라 클로저의 변수와 매핑된다.

사용자 정의 태스크와 플러그인에서 파일 처리

사용자 정의 태스크를 개발할 때 파일 위치의 입력값을 설정받도록 하는 것은 매우 좋은 아이디어이다. Project.file() 메소드를 활용해서 최대한 늦게 file의 값을 결정할 수있다.

다음 예제는 파일의 속성을 최대한 지연하여(lazy하게) 결정(evaluating)한다.

```
class GreetingToFileTask extends DefaultTask {

    def destination

    // destination을 파일로 읽어들인다.
    File getDestination() {
        project.file(destination)
    }

    @TaskAction
    def greet() {
        // 파일을 읽어들여서 내용을 기록한다.
        def file = getDestination()
        file.parentFile.mkdirs()
        file.write "Hello!"
    }
}

// 타입이 GreetingToFileTask 인 태스크
task greet(type: GreetingToFileTask) {
    //  destination 의 값을 클로저로 정의해서 lazy하게 처리함
    destination = { project.greetingFile }
}

task sayGreeting(dependsOn: greet) << {
    // greetingFile 내용을 파일에 쓴다.
    println file(greetingFile).text
}

// 파일 위치를 입력받는다.
greetingFile = "$buildDir/hello.txt"
```

gradle -q sayGreeting 을 실행해보면 다음과 같다.

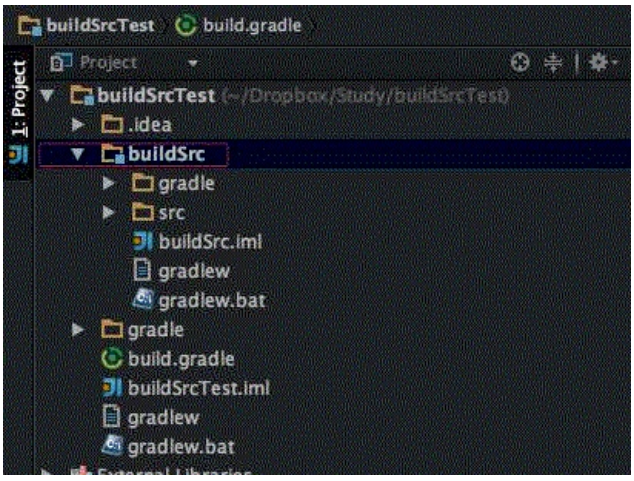
Hello!

이 예제에서는 greet 태스크의 destination 속성을 클로저로 정의했다. 마지막 순간(sayGreeting)에 이 클로저는 리턴 값을 file객체로 돌려줌으로써 Project.file() 메소드와 함께 (lazy 하게) 평가되었다.

buildSrc 프로젝트 플러그인

플러그인 소스를 rootProjectDir/buildSrc/src/main/groovy 디렉토리에 배치하는 방법이다.

먼저 buildSrc 디렉토리를 만든다. 그러면 gradle은 자동으로 buildSrc 아래의 소스를 컴파일하게 된다.



intellij에서는 buildSrc를 디렉토리를 만드는 것만으로도 source 디렉토리로 인식한다.

build/src/main/groovy/GreetingPlugin

greeting 태스크를 가지고 있는 GreetingPlugin 클래스를 만든다.

```
iimport org.gradle.api.Plugin
import org.gradle.api.Project

class GreetingPlugin implements Plugin<Project> {

    @Override
    void apply(Project target) {
        target.task('greeting') << {
            println "greeting!!"
        }
    }
}
```

build.gradle

build.gradle 에서는 위 greeting을 의존하는 hello 태스크를 만든다. GradlePlugin 플러그인을 적용한 것에 유의한다.

```
apply plugin: GreetingPlugin

task hello(dependsOn: 'greeting') {
    println 'hello!'
}
```

gradle -q hello 를 실행해보면 다음과 같이 실행되는 것을 알 수 있다.

```
hello!
greeting!!
```

독립(standalone) 프로젝트 플러그인

독립 프로젝트 예제 소스는 git에 올려놓았습니다.
git clone https://github.com/sjune/gradle-tutorial.git
cd customPlugin

plugin 플러그인 작성

우리가 만든 플러그인을 독립 프로젝트로 옮겨서 다른 사용자들과 공유할 수 있다. 이 프로젝트는 플러그인 classes가 포함된 JAR를 가진 간단한 그루비 프로젝트이다.
먼저 다음은 간단한 예제이다. build.gralde에는 groovy 플러그인을 적용하고 Gradle API와 컴파일 시점 의존성을 추가 하였다. 그리고 jar 배포를 위해 uploadArchives도 정의하였다.

h4. customPlugin/plugin/build.gradle

```
apply plugin: 'groovy'

dependencies {
    compile gradleApi()
    groovy localGroovy()
}
..
// jar 배포를 위함
uploadArchives {
    repositories {
        mavenDeployer {
            repository(url: uri('../repo'))
        }
    }
}
```

Gradle은 Plugin 구현체를 JAR 안의 META-INF/gradle-plugins 디렉토리에서 플러그인의 이름에 해당하는 속성 파일을 통해 찾는다.
src/main/resources/META-INF/gradle-plugins/greeting.properties 안에 다음과 같이 작성된다.

```
implementation-class=org.gradle.GreetingPlugin
```

GreetingPlugin은 플러그인 이름이 된다.

다른 프로젝트(consumer)에서 plugin 플러그인 사용

먼저 customPlugin/plugin 에서 gradle -q uploadArchvices 명령어를 사용하여 jar를 로컬 레퍼지토리(../repo)에 배포한다.

```
Uploading: org/gradle/customPlugin/1.0-SNAPSHOT/customPlugin-1.0-20140410.171843-1.jar to repository remote at file:/Users/sjune/github/gradle-tutorial/customPlugin/repo
Transferring 1K from remote
Uploaded 1K
```

빌드 스크립트 플러그인을 사용하기 위해서는 빌드 스크립트의 classpath에 플러그인 class를 추가해야한다. 이때 buildscript 블럭을 사용한다. 다음 예제는 로컬 레퍼지토리에서 라이브러리를 가져오는 방법을 보여준다.

```
buildscript {
    repositories {
        maven {
            url uri('../repo')
        }
    }
    dependencies {
        classpath group: 'org.gradle', name: 'customPlugin', version: '1.0-SNAPSHOT'
    }
}

apply plugin: 'greeting' // greeting 플러그인을 적용한다.
```

customPlugin/consumer/ 에서 gradle -q greeting을 실행해보면 다음과 같이 GreetingPlugin의 greeting 태스크가가 호출된것을 알 수 있다.

```
howdy!
```

플러그인 테스트 만들기

플러그인 구현 테스트를 할 때는 ProjectBuilder 클래스를 사용해서 Project 인스턴스를 만든다.

다음 코드는 src/test/groovy/org/gradle/GreetingPluginTest.groovy 에 존재한다.

```
class GreetingPluginTest {
    @Test
    public void greeterPluginAddsGreetingTaskToProject() {
        Project project = ProjectBuilder.builder().build()
        project.apply plugin: 'greeting'

        assertTrue(project.tasks.hello instanceof GreetingTask)
    }
}
```

다중 도메인 객체 관리

Gradle은 객체 관리를 위한 몇가지 유틸리티성 클래스를 제공한다. 이것은 Gradle 빌드 언어에서 활용할 수 있다.

```
apply plugin: DocumentationPlugin

books {
    quickStart {
        // quickStart는 값을 오버라이딩하였다.
        sourceFile = file('src/docs/quick-start')
```

```
    }
    userGuide {

    }
    developerGuide {

    }
}

task books << {
    books.each { book ->
        println "$book.name -> $book.sourceFile"
    }
}

// 커스텀 플러그인을 정의한다.
class DocumentationPlugin implements Plugin<Project> {
    void apply(Project project) {
        // Project.container() 메서드는 NamedDomainObjectContainer 의 인스턴스를 생성한다.
        // 새로운 인스턴스를 생성할 때 인자 값으로 받은 클래스의 name 속성에 이름을 지정한다. 그래서 name이 고유해야하는 것이다.
        def books = project.container(Book)

        // books의 요소를 반복하면서 file객체를 sourceFile에 할당한다. $name은 요소의 name을 가리킨다.
        books.all {
            sourceFile = project.file("src/docs/$name")
        }
        project.extensions.books = books
    }
}

class Book {
    final String name // 'name' 속성은 반드시 상수이자 유니크한 값이어야 한다.
    File sourceFile

    Book(String name) {
        this.name = name
    }
}
```

gradle -q books 를 실행해보면 결과는 다음과 같다.

```
developerGuide ->
/home/user/gradle/samples/userguide/organizeBuildLogic/customPluginWithDomainObjectContainer/src/docs/developerGuide
quickStart ->
/home/user/gradle/samples/userguide/organizeBuildLogic/customPluginWithDomainObjectContainer/src/docs/quick-start
userGuide ->
/home/user/gradle/samples/userguide/organizeBuildLogic/customPluginWithDomainObjectContainer/src/docs/userGuide
```

NamedDomainObjectContainer 인스턴스는 객체를 관리하고 설정에 편리한 많은 메소드를 제공한다. 위 예제는 제공되는 메소드 중에서 all() 이 사용되었다

참고 링크

- http://www.gradle.org/docs/current/userguide/plugins.html
- http://www.gradle.org/docs/current/userguide/custom_plugins.html
- http://www.gradle.org/docs/current/userguide/organizing_build_logic.html
- http://www.gradle.org/docs/current/userguide/groovy_plugin.html

---

링크 목록

- Plugin으로 할 수 있는 것들 - http://gliderwiki.org/wiki/201/#Plugin으로 할 수 있는 일들
- Source Set - http://www.gradle.org/docs/current/dsl/org.gradle.api.tasks.SourceSet.html

관련 키워드

Gradle
