EBook Gratuito

APPRENDIMENTO gradle

Free unaffiliated eBook created from
Stack Overflow contributors.

#gradle

Sommario

Di

Capitolo 1: Iniziare con Gradle

Osservazioni

Gradle è uno strumento di creazione open-source e generico. È popolare nella comunità Java ed è lo strumento di creazione preferito per Android .

Funzioni di Gradle evidenziate

- Gli script di compilazione dichiarativi *sono* codice scritto in Groovy o Kotlin .
- Molti plug-in di base e di comunità che utilizzano un approccio flessibile basato sulla convenzione
- Le build incrementali sono tali che le attività le cui dipendenze che non sono state modificate non vengono rieseguite.
- Risoluzione delle dipendenze integrata per Maven ed Ivy . I plugin forniti forniscono la risoluzione delle dipendenze da altri `repositories` come npm .
- Creazioni multiprogetto di prima classe.
- Integrazione con altri strumenti di costruzione come Maven , Ant e altri.
- Crea scansioni che aumentano la capacità degli sviluppatori di collaborare e ottimizzare le build di Gradle.

Maggiori informazioni

Se vuoi saperne di più sulle funzionalità di Gradle puoi consultare la parte Panoramica della Guida dell'utente di Gradle .

Se vuoi provare Gradle puoi dare un'occhiata alle guide qui . È possibile visualizzare una guida rapida all'avvio di Java, imparare come utilizzare Gradle per la prima volta e migrare da un altro strumento di creazione.

Examples

Installazione gradle

Requisiti: Java JDK o JRE installato (versione 7 o successiva per versione Gradle 3.x)

Passaggi di installazione:

1. Scarica la distribuzione di Gradle dal sito ufficiale
2. Disimballare lo ZIP
3. Aggiungi la variabile di ambiente `GRADLE_HOME` . Questa variabile dovrebbe puntare ai file decompressi dal passaggio precedente.
4. Aggiungi `GRADLE_HOME/bin` alla tua variabile d'ambiente `PATH` , quindi puoi eseguire Gradle dall'interfaccia della riga di comando (CLI)
5. Verifica l'installazione di Gradle digitando `gradle -v` nella CLI. L'output dovrebbe contenere la

versione di Gradle installata e i dettagli di configurazione di Gradle correnti

Ulteriori informazioni possono essere trovate nella guida utente ufficiale

Installazione con homebrew su OS X / macOS

Gli utenti di homebrew possono installare gradle eseguendo

```
brew install gradle
```

Installazione con SdkMan

Gli utenti di SdkMan possono installare Gradle eseguendo:

```
sdk install gradle
```

Installa una versione specifica

```
sdk list gradle
sdk install gradle 2.14
```

Passa alle versioni

```
sdk use gradle 2.12
```

Installa il plugin Gradle per Eclipse

Ecco i passaggi necessari per installare il plugin Gradle in Eclipse:

1. Apri Eclipse e vai su **Aiuto** -> **Mercato Eclipse**
2. Nella barra di ricerca, inserisci **buildship** e premi **invio**
3. Seleziona **"Buildship Gradle Integration 1.0"** e fai clic su **Installa**
4. Nella finestra successiva, fare clic su **Conferma**
5. Quindi, **accetta** i termini e la licenza di accordo, quindi fai clic su **Fine**
6. Dopo l'installazione, Eclipse dovrà riavviarsi, fare clic su **Sì**

Ciao mondo

Le attività Gradle possono essere scritte usando il codice Groovy all'interno del file build.gradle di un progetto. Queste attività possono quindi essere eseguite utilizzando `> gradle [taskname]` sul terminale o eseguendo l'operazione da un IDE come Eclipse.

Per creare l'esempio Hello World in gradle, dobbiamo definire un'attività che stamperà una stringa sulla console usando Groovy. Useremo Groovy's `printLn` per chiamare il metodo `System.out.printLn` di Java per stampare il testo sulla console.

build.gradle

```
task hello {
    doLast {
        println 'Hello world!'
    }
}
```

Possiamo quindi eseguire questa operazione utilizzando `> gradle hello` o `> gradle -q hello` . Il `-q` è usato per sopprimere i messaggi del registro gradle in modo che venga mostrato solo l'output dell'attività.

Uscita di `> gradle -q hello` :

```
> gradle -q hello
Hello world!
```

Ulteriori informazioni sulle attività

Prima di tutto: l'operatore `<<` (leftShift) è equivalente a `doLast {closure}` . Dal **gradle 3.2** è **deprecato** . Tutto il codice attività è scritto in un **build.gradle** .

> Un'attività rappresenta un pezzo di lavoro atomico eseguito da una build. Potrebbe essere la compilazione di alcune classi, la creazione di un JAR, la generazione di Javadoc o la pubblicazione di alcuni archivi in un repository.

Gradle supporta due grandi tipi di compiti: semplice e avanzato.

Osserviamo alcuni stili di definizione delle attività:

```
task hello {
    doLast{
       //some code
    }
}
```

O il:

```
task(hello) {
    doLast{
       //some code
    }
}
```

Questi compiti sopra sono equivalenti. Inoltre, è possibile fornire alcune estensioni all'attività, ad esempio: `dependsOn` , `mustRunAfter` , `type` ecc. È possibile estendere l'attività aggiungendo azioni dopo la definizione di attività, in questo modo:

```
task hello {
    doLast{
       println 'Inside task'
    }
}
```

```
hello.doLast {
    println 'added code'
}
```

Quando eseguiremo questo abbiamo ottenuto:

```
> gradle -q hello
    Inside task
    added code
```

Domande sulle dipendenze delle attività e l'ordine esaminati qui

Parliamo di due grandi tipi di attività.

Semplice:

Compiti che definiamo con una chiusura di azione:

```
    task hello {
        doLast{
        println "Hello from a simple task"
        }
    }
```

Migliorata

Migliorato è un compito con un comportamento preconfigurato. Tutti i plugin che utilizzi nel tuo progetto sono quelli *estesi* o **potenziati** . Creiamo la nostra e capirai come funziona:

```
task hello(type: HelloTask)

class HelloTask extends DefaultTask {
    @TaskAction
    def greet() {
        println 'hello from our custom task'
    }
}
```

Inoltre, possiamo passare parametri al nostro compito, in questo modo:

```
class HelloTask extends DefaultTask {
    String greeting = "This is default greeting"
    @TaskAction
    def greet() {
        println greeting
    }
}
```

E da ora in poi possiamo riscrivere il nostro compito in questo modo:

```
   //this is our old task definition style
task oldHello(type: HelloTask)
   //this is our new task definition style
task newHello(type: HelloTask) {
    greeting = 'This is not default greeting!'
}
```

Quando eseguiremo questo abbiamo ottenuto:

```
> gradle -q oldHello
This is default greeting

> gradle -q newHello
This is not default greeting!
```

Tutte le domande sui plugin di sviluppo gradle sul sito ufficiale

Leggi Iniziare con Gradle online: https://riptutorial.com/it/gradle/topic/894/iniziare-con-gradle

Capitolo 2: Compiti di ordinazione

Osservazioni

> Si noti che `mustRunAfter` e `shouldRunAfter` sono contrassegnati come "incubating" (come da Gradle 3.0), il che significa che si tratta di funzionalità sperimentali e il loro comportamento può essere modificato nelle versioni future.

Sono disponibili due regole di ordinazione:

- `mustRunAfter`
- `shouldRunAfter`

Quando si utilizza la regola di ordinamento `mustRunAfter` , si specifica che l'attività B deve sempre essere eseguita dopo l'attività A, ogni volta che vengono eseguiti sia taskA che taskB.

La regola di ordinamento `shouldRunAfter` è simile ma meno rigorosa in quanto verrà ignorata in due situazioni:

- se l'utilizzo di tale regola introduce un ciclo di ordinazione.
- quando si utilizza l'esecuzione parallela e tutte le dipendenze di un'attività sono state soddisfatte a prescindere dall'attività `shouldRunAfter` , questa attività verrà eseguita indipendentemente dal fatto che le dipendenze `shouldRunAfter` siano state eseguite o meno.

Examples

Ordinare con il metodo mustRunAfter

```
task A << {
    println 'Hello from A'
}
task B << {
    println 'Hello from B'
}

B.mustRunAfter A
```

`B.mustRunAfter A` riga indica a Gradle di eseguire l'attività dopo l'attività specificata come argomento.

E l'output è:

```
> gradle -q B A
Hello from A
Hello from B
```

La regola di ordinazione non introduce la dipendenza tra le attività A e B, ma ha un effetto solo quando **entrambe le attività sono pianificate** per l'esecuzione.

Significa che possiamo eseguire le attività A e B in modo indipendente.

L'output è:

```
> gradle -q B
Hello from B
```

Leggi Compiti di ordinazione online: https://riptutorial.com/it/gradle/topic/5550/compiti-di-ordinazione

Capitolo 3: dipendenze

Examples

Aggiungi una dipendenza file JAR locale

Singolo JAR

A volte hai un file JAR locale che devi aggiungere come dipendenza alla build di Gradle. Ecco come puoi fare questo:

```
dependencies {
    compile files('path/local_dependency.jar')
}
```

Dove `path` è un percorso di directory sul file system e `local_dependency.jar` è il nome del file JAR locale. Il `path` può essere relativo al file di build.

Elenco di JAR

È anche possibile aggiungere una directory di jar da compilare. Questo può essere fatto in questo modo:

```
dependencies {
        compile fileTree(dir: 'libs', include: '*.jar')
}
```

Dove `libs` sarebbe la directory contenente i jar e `*.jar` sarebbe il filtro di quali file includere.

Directory di JAR come repository

Se si desidera solo cercare i jar in un repository invece di aggiungerli direttamente come dipendenza con il loro percorso, è possibile utilizzare un repository flatDir.

```
repositories {
    flatDir {
        dirs 'libs'
    }
}
```

Cerca i jar nella directory *`libs`* e le relative directory child.

Aggiungi una dipendenza

Le dipendenze in Gradle seguono lo stesso formato di Maven . Le dipendenze sono strutturate come segue:

```
group:name:version
```

Ecco un esempio:

```
'org.springframework:spring-core:4.3.1.RELEASE'
```

Per aggiungere una dipendenza in fase di compilazione, aggiungi semplicemente questa riga nel blocco delle `dependency` nel file di build Gradle:

```
compile 'org.springframework:spring-core:4.3.1.RELEASE'
```

Una sintassi alternativa per questo chiama esplicitamente ciascun componente della dipendenza, in questo modo:

```
compile group: 'org.springframework', name: 'spring-core', version: '4.3.1.RELEASE'
```

Questo aggiunge una dipendenza al momento della compilazione.

È anche possibile aggiungere dipendenze solo per i test. Ecco un esempio:

```
testCompile group: 'junit', name: 'junit', version: '4.+'
```

Dipende da un altro progetto Gradle

Nel caso di una build gradle multi-progetto, a volte potresti dover dipendere da un altro progetto nella tua build. Per fare ciò, inserirai quanto segue nelle dipendenze del tuo progetto:

```
dependencies {
    compile project(':OtherProject')
}
```

Dove `':OtherProject'` è il percorso gradle per il progetto, referenziato dalla radice della struttura di directory.

Per rendere `':OtherProject'` disponibile nel contesto del file `build.gradle` , aggiungilo al corrispondente `settings.gradle`

```
include ':Dependency'
project(':Dependency').projectDir = new File('/path/to/dependency')
```

Per una spiegazione più dettagliata, puoi consultare la documentazione ufficiale di Gradle qui .

Elenco delle dipendenze

La chiamata all'attività `dependencies` consente di visualizzare le dipendenze del progetto root:

```
gradle dependencies
```

I risultati sono grafici di dipendenza (tenendo conto delle dipendenze transitive), suddivisi per configurazione. Per limitare le configurazioni visualizzate, è possibile passare l'opzione `--configuration` seguita da una configurazione scelta per analizzare:

```
gradle dependencies --configuration compile
```

Per visualizzare le dipendenze di un sottoprogetto, utilizzare `<subproject>:dependencies` attività delle `<subproject>:dependencies` . Ad esempio, per elencare le dipendenze di un sottoprogetto denominato `api` :

```
gradle api:dependencies
```

Aggiunta di repository

Devi indicare Gradle alla posizione dei tuoi plugin, in modo che Gradle possa trovarli. Fai questo aggiungendo `repositories { ... }` al tuo `build.gradle` .

Ecco un esempio di aggiunta di tre repository, JCenter , Maven Repository e un repository personalizzato che offre dipendenze nello stile Maven.

```
repositories {
  // Adding these two repositories via method calls is made possible by Gradle's Java plugin
  jcenter()
  mavenCentral()

  maven { url "http://repository.of/dependency" }
}
```

Aggiungi il file .aar al progetto Android usando gradle

1. Passare al modulo `app` del progetto e creare la directory `libs` .
2. Metti il tuo file `.aar` lì. Ad esempio `myLib.aar` .
3. Aggiungere il codice sottostante per `android` blocco di `app` di livello `build.gradle` file.

```
  repositories {
        flatDir {
            dirs 'libs'
        }
    }
```

In questo modo hai definito un nuovo repository extra che punta alla cartella `libs` del modulo `app` .

4. Aggiungi il codice qui sotto al blocco delle `dependencies` o al file `build.gradle` :

```
compile(name:'myLib', ext:'aar')
```

Leggi dipendenze online: https://riptutorial.com/it/gradle/topic/2524/dipendenze

Capitolo 4: Dipendenze delle attività

Osservazioni

doLast

Nota che in un gradle 3.x la definizione del task in modo più idiomatico: usando la **notazione esplicita doLast {closure}** invece preferibile l'operatore "leftShift" (<<). (**LeftShift** è stato deprecato in un gradle 3.2 è programmato per essere rimosso in gradle 5.0 .)

```
task oldStyle << {
    println 'Deprecated style task'
 }
```

è equivalente a:

```
task newStyle {
    doLast {
    println 'Deprecated style task'
    }
 }
```

Examples

Aggiunta di dipendenze usando i nomi delle attività

Possiamo cambiare l'ordine di esecuzione delle attività con il metodo `dependsOn` .

```
task A << {
    println 'Hello from A'
}
task B(dependsOn: A) << {
    println "Hello from B"
}
```

Aggiungere `dependsOn: cause:

- l'attività B dipende dall'attività A
- Gradle eseguire `A` ogni compito **prima** del `B` esecuzione dell'attività.

E l'output è:

```
> gradle -q B
Hello from A
Hello from B
```

Aggiunta di dipendenze da un altro progetto

```
project('projectA') {
    task A(dependsOn: ':projectB:B') << {
        println 'Hello from A'
    }
}

project('projectB') {
    task B << {
        println 'Hello from B'
    }
}
```

Per fare riferimento a un'attività in un altro progetto, si **precede il nome dell'attività** con il percorso del progetto a cui appartiene `:projectB:B`

E l'output è:

```
> gradle -q B
Hello from A
Hello from B
```

Aggiunta di dipendenza utilizzando l'oggetto compito

```
task A << {
    println 'Hello from A'
}

task B << {
    println 'Hello from B'
}

B.dependsOn A
```

È un modo alternativo per definire la dipendenza anziché utilizzare il nome dell'attività.

E l'output è lo stesso:

```
> gradle -q B
Hello from A
Hello from B
```

Aggiunta di dipendenze multiple

È possibile aggiungere più dipendenze.

```
task A << {
    println 'Hello from A'
}

task B << {
    println 'Hello from B'
}
```

```
task C << {
    println 'Hello from C'
}

task D << {
    println 'Hello from D'
}
```

Ora puoi definire una serie di dipendenze:

```
B.dependsOn A
C.dependsOn B
D.dependsOn C
```

L'output è:

```
> gradle -q D
Hello from A
Hello from B
Hello from C
Hello from D
```

Altro esempio:

```
B.dependsOn A
D.dependsOn B
D.dependsOn C
```

L'output è:

```
> gradle -q D
Hello from A
Hello from B
Hello from C
Hello from D
```

Dipendenze multiple con il metodo dependsOn

È possibile aggiungere più dipendenze.

```
task A << {
    println 'Hello from A'
}

task B(dependsOn: A) << {
    println 'Hello from B'
}

task C << {
    println 'Hello from C'
}

task D(dependsOn: ['B', 'C'] << {
    println 'Hello from D'
```

```
}
```

L'output è:

```
> gradle -q D
Hello from A
Hello from B
Hello from C
Hello from D
```

Leggi Dipendenze delle attività online: https://riptutorial.com/it/gradle/topic/5545/dipendenze-delle-attivita

Capitolo 5: Gradle Performance

Examples

Creazione di profili di una build

Prima di iniziare a sintonizzare la build di Gradle per le prestazioni, dovresti stabilire una linea di base e capire quali parti della build impiegano più tempo. Per fare ciò, puoi profilare la tua build aggiungendo l'argomento `--profile` al comando Gradle:

```
gradle --profile
./gradlew --profile
```

Al termine della compilazione, verrà visualizzato un report del profilo HTML per la compilazione in `./build/reports/profile/` , con un aspetto simile al seguente:

Profile report

Profiled build: build

Started on: 2016/07/23 - 17:47:33

Summary Configuration Depend

Description	Duration
Total Build Time	20.654s
Startup	0.598s
Settings and BuildSrc	0.001s
Loading Projects	0.003s
Configuring Projects	0.061s
Task Execution	19.611s

Generated by Gradle 2.14.1 at Jul 23, 2016 5:47:53 PM

Facendo clic sulle schede accanto a **Riepilogo**

, è possibile visualizzare una ripartizione più dettagliata di dove viene speso il tempo.

Configura su richiesta

Se la creazione di profili della build mostra un notevole dispendio di tempo in **Configurazione progetti** , l'opzione Configura su richiesta potrebbe migliorare le prestazioni.

È possibile abilitare la modalità Configura su richiesta modificando `$GRADLE_USER_HOME/.gradle/gradle.properties` (`~/.gradle/gradle.properties` per impostazione predefinita) e impostando `org.gradle.configureondemand` .

```
org.gradle.configureondemand=true
```

Per abilitarlo solo per un progetto specifico, modifica invece il file `gradle.properties` quel progetto.

Se Configura su richiesta è abilitato, anziché configurare tutti i progetti in primo piano, Gradle configurerà solo i progetti necessari per l'attività da eseguire.

Dal manuale di Gralde :

> La modalità di configurazione on demand tenta di configurare solo i progetti rilevanti per le attività richieste, ovvero esegue solo il file `build.gradle` dei progetti che partecipano alla generazione. In questo modo, è possibile ridurre il tempo di configurazione di una grande build multi-progetto. A lungo termine, questa modalità diventerà la modalità predefinita, probabilmente l'unica modalità per l'esecuzione di build di Gradle.

Ottimizzazione dei parametri di utilizzo della memoria JVM per Gradle

È possibile impostare o aumentare i limiti di utilizzo della memoria (o altri argomenti JVM) usati per le build Gradle e il demone Gradle modificando `$GRADLE_USER_HOME/.gradle/gradle.properties` (`~/.gradle/gradle.properties` per impostazione predefinita) e impostando `org.gradle.jvmargs` .

Per configurare questi limiti solo per un progetto specifico, modifica invece il file `gradle.properties` quel progetto.

Le impostazioni di utilizzo della memoria predefinite per build Gradle e Demone Gradle sono:

```
org.gradle.jvmargs=-Xmx1024m -XX:MaxPermSize=256m
```

Ciò consente un'allocazione di memoria massima generale (dimensione heap) di 1 GB e un'allocazione di memoria massima per oggetti "interni" permanenti di 256 MB. Quando vengono raggiunte queste dimensioni, si verifica Garbage Collection, che può ridurre significativamente le prestazioni.

Supponendo che tu abbia la memoria di riserva, potresti facilmente raddoppiarli in questo modo:

```
org.gradle.jvmargs=-Xmx2024m -XX:MaxPermSize=512m
```

Nota che smetterai di vedere i benefici derivanti dall'aumento di `XX:MaxPermSize` prima di quando `Xmx` aumenta e diventa `Xmx` .

Usa il demone Gradle

Puoi abilitare il demone Gradle per migliorare le prestazioni delle tue build.

Il demone Gradle mantiene il Gradle Framework inizializzato e funzionante, e memorizza nella cache i dati del progetto in memoria per migliorare le prestazioni.

Per una singola costruzione

Per abilitare il demone per una singola build, è sufficiente passare l'argomento `--daemon` al comando `gradle` o allo script Gradle Wrapper.

```
gradle --daemon
./gradlew --daemon
```

Per tutte le build di un progetto

Per abilitare il demone per tutte le build di un progetto, puoi aggiungere:

```
org.gradle.daemon=true
```

Al file `gradle.properties` del progetto.

For All Builds

Per abilitare il demone Gradle per impostazione predefinita, per ogni build creata dal tuo account utente sul tuo sistema, modifica `$GRADLE_USER_HOME/.gradle/gradle.properties` (`~/.gradle/gradle.properties` per impostazione predefinita) e aggiungi questa riga:

```
org.gradle.daemon=true
```

Puoi anche farlo in un singolo comando su sistemi Mac / Linux / * nix:

```
touch ~/.gradle/gradle.properties && echo "org.gradle.daemon=true" >>
~/.gradle/gradle.properties
```

Oppure su Windows:

```
(if not exist "%USERPROFILE%/.gradle" mkdir "%USERPROFILE%/.gradle") && (echo
org.gradle.daemon=true >> "%USERPROFILE%/.gradle/gradle.properties")
```

Disabilitare il demone

È possibile disabilitare il daemon per una build specifica utilizzando l'argomento `--no-daemon` o disattivarlo per un progetto specifico impostando esplicitamente `org.gradle.daemon=false` nel file `gradle.properties` del progetto.

Fermare il demone

Se si desidera interrompere manualmente un processo Daemon, è possibile interrompere il processo tramite il task manager del sistema operativo o eseguire il comando `gradle --stop` . L' `--stop` fa in modo che Gradle richieda che tutti i processi di Daemon in esecuzione, della stessa versione di Gradle utilizzata per eseguire il comando, si interrompano. Normalmente, i processi demone terminano automaticamente * dopo * *3 ore di inattività o meno* .

Gradle Parallel costruisce

Gradle eseguirà solo un'attività alla volta per impostazione predefinita, indipendentemente dalla struttura del progetto. Usando l' `--parallel` , puoi forzare Gradle ad eseguire sottoprogetti indipendenti - quelli che non hanno dipendenze tra progetti implicite o esplicite tra di loro - in parallelo, permettendogli di eseguire più attività contemporaneamente fintantoché quelle attività sono in diversi progetti.

Per costruire un progetto in modalità parallela:

```
gradle build --parallel
```

È anche possibile creare in parallelo l'impostazione predefinita per un progetto aggiungendo la seguente impostazione al file gradle.properties del progetto:

```
org.gradle.parallel=true
```

Usa l'ultima versione di Gradle

Il team di Gradle lavora regolarmente per migliorare le prestazioni dei diversi aspetti delle build di Gradle. Se stai usando una vecchia versione di Gradle, stai perdendo i benefici di quel lavoro. Prova ad aggiornare all'ultima versione di Gradle per vedere che tipo di impatto ha. Fare così è a basso rischio perché pochissime cose si interrompono tra versioni minori di Gradle.

Il file delle proprietà per il wrapper Gradle può essere trovato nella cartella del progetto sotto `gradle/wrapper/` e si chiama `gradle-wrapper.properties` . Il contenuto di quel file potrebbe assomigliare a questo:

```
distributionBase=GRADLE_USER_HOME
distributionPath=wrapper/dists
zipStoreBase=GRADLE_USER_HOME
zipStorePath=wrapper/dists
distributionUrl=https\://services.gradle.org/distributions/gradle-X.X.X.zip
```

È possibile modificare manualmente il numero di versione `XXX` (versione corrente) su `YYY` (versione più recente) e la volta successiva che si esegue il wrapper, la nuova versione viene scaricata automaticamente.

Leggi Gradle Performance online: https://riptutorial.com/it/gradle/topic/3443/gradle-performance

Capitolo 6: Gradle Plugin

Examples

Semplice plugin gradle da `buildSrc`

Semplice esempio di come creare un plugin personalizzato e DSL per il tuo progetto gradle.
Questo esempio utilizza uno dei tre modi possibili per creare plugin.
I tre modi sono:

- in linea
- buildSrc
- plugin standalone

Questo esempio mostra la creazione di un plug-in dalla cartella **buildSrc** .

Questo esempio creerà cinque file

```
// project's build.gradle
build.gradle
// build.gradle to build the `buildSrc` module
buildSrc/build.gradle
// file name will be the plugin name used in the `apply plugin: $name`
// where name would be `sample` in this example
buildSrc/src/main/resources/META-INF/gradle-plugins/sample.properties
// our DSL (Domain Specific Language) model
buildSrc/src/main/groovy/so/docs/gradle/plugin/SampleModel.groovy
// our actual plugin that will read the values from the DSL
buildSrc/src/main/groovy/so/docs/gradle/plugin/SamplePlugin.groovy
```

build.gradle:

```
group 'so.docs.gradle'
version '1.0-SNAPSHOT'

apply plugin: 'groovy'
// apply our plugin... calls SamplePlugin#apply(Project)
apply plugin: 'sample'

repositories {
    mavenCentral()
}

dependencies {
    compile localGroovy()
}

// caller populates the extension model applied above
sample {
    product = 'abc'
    customer = 'zyx'
}
```

```
// dummy task to limit console output for example
task doNothing <<{}
```

buildSrc / build.gradle

```
apply plugin: 'groovy'

repositories {
    mavenCentral()
}

dependencies {
    compile localGroovy()
}
```

buildSrc / src / main / Groovy / SO / docs / Gradle / plugin / SamplePlugin.groovy:

```
package so.docs.gradle.plugin

import org.gradle.api.Plugin
import org.gradle.api.Project

class SamplePlugin implements Plugin<Project> {
    @Override
    void apply(Project target) {
        // create our extension on the project for our model
        target.extensions.create('sample', SampleModel)
        // once the script has been evaluated the values are available
        target.afterEvaluate {
            // here we can do whatever we need to with our values
            println "populated model: $target.extensions.sample"
        }
    }
}
```

buildSrc / src / main / Groovy / SO / docs / Gradle / plugin / SampleModel.groovy:

```
package so.docs.gradle.plugin

// define our DSL model
class SampleModel {
    public String product;
    public String customer;

    @Override
    public String toString() {
        final StringBuilder sb = new StringBuilder("SampleModel{");
        sb.append("product='").append(product).append('\'');
        sb.append(", customer='").append(customer).append('\'');
        sb.append('}');
        return sb.toString();
    }
}
```

buildSrc / src / main / resources / META-INF / Gradle-plugins / sample.properties

```
implementation-class=so.docs.gradle.plugin.SamplePlugin
```

Usando questa configurazione possiamo vedere i valori forniti dal chiamante nel tuo blocco DSL

```
 $ ./gradlew -q doNothing
SampleModel{product='abc', customer='zyx'}
```

Come scrivere un plugin standalone

Per creare un plug-in graduale standalone personalizzato usando java (puoi usare anche Groovy) devi creare una struttura come questa:

```
plugin
|-- build.gradle
|-- settings.gradle
|-- src
    |-- main
    |   |-- java
    |   |-- resources
    |       |-- META-INF
    |           |-- gradle-plugins
    |-- test
```

Configura la configurazione gradle

Nel file `build.gradle` definisci il tuo progetto.

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

dependencies {
    compile gradleApi()
}
```

Il plugin `java` verrà utilizzato per scrivere il codice java.
La dipendenza `gradleApi()` ci fornirà tutti i metodi e le proprietà necessarie per creare un plugin Gradle.

Nel file `settings.gradle` :

```
rootProject.name = 'myplugin'
```

Definirà l' **id artefatto** in Maven.
Se il file `settings.gradle` non è presente nella directory plugin, il valore predefinito sarà il nome della directory.

Crea il plugin

Definire una classe in `src/main/java/org/sample/MyPlugin.java` implementando l'interfaccia `Plugin` .

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

public class MyPlugin implements Plugin<Project> {

    @Override
    public void apply(Project project) {
         project.getTasks().create("myTask", MyTask.class);
    }

}
```

Definire l'attività che estende la classe `DefaultTask` :

```
import org.gradle.api.DefaultTask;
import org.gradle.api.tasks.TaskAction;

public class MyTask extends DefaultTask {

    @TaskAction
    public void myTask() {
        System.out.println("Hello World");
    }
}
```

Dichiarazione sulla classe dei plugin

Nella cartella `META-INF/gradle-plugins` è necessario creare un file delle proprietà che definisca la proprietà della `implementation-class` che identifica la classe di implementazione del plugin.

Nel `META-INF/gradle-plugins/testplugin.properties`

```
implementation-class=org.sample.MyPlugin.java
```

Si noti che il **nome file delle proprietà corrisponde all'id del plug-in** .

Come costruirlo e pubblicarlo

Cambia il file `build.gradle` aggiungendo alcune informazioni per caricare il plug-in in un repository:

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

dependencies {
    compile gradleApi()
}

repositories {
    jcenter()
```

```
}


group = 'org.sample'
version = '1.0'

uploadArchives {
    repositories {
        mavenDeployer {
        repository(url: mavenLocal().url)
        }
    }
}
```

È possibile creare e pubblicare il plug-in Gradle sul repository Maven definito nel file `plugin/build.gradle` utilizzando il seguente comando.

```
$ ./gradlew clean uploadArchives
```

Come usarlo

Per usare il plugin aggiungi nel `build.gradle` del tuo progetto:

```
buildscript {
     repositories {
         mavenLocal()
     }
 dependencies {
    classpath group: 'org.sample',    // Defined in the build.gradle of the plugin
              name: 'myplugin',       // Defined by the rootProject.name
              version: '1.0'
    }
 }

apply plugin: 'testplugin'            // Defined by the properties filename
```

Quindi puoi chiamare l'attività usando:

```
 $ ./gradlew myTask
```

Leggi Gradle Plugin online: https://riptutorial.com/it/gradle/topic/1900/gradle-plugin

Capitolo 7: Gradle Wrapper

Examples

Gradle Wrapper e Git

Come discusso nell'introduzione, la funzionalità di gradle wrapper funziona perché un jar viene scaricato nel progetto da utilizzare quando viene eseguito il comando `gradlew` . Tuttavia, questo potrebbe non essere eseguito e dopo la prossima estrazione del progetto, `gradlew` non verrà eseguito con l'errore:

```
Error: Could not find or load main class org.gradle.wrapper.GradleWrapperMain
```

Questo perché il tuo .gitignore probabilmente includerà `*jar` per i progetti Java. Quando il wrapper gradle è stato inizializzato, copia nel file `gradle/wrapper/gradle-wrapper.jar` . Quindi è necessario aggiungerlo all'indice git e commetterlo. Fatelo con:

```
git add -f gradle/wrapper/gradle-wrapper.jar
git ci
```

Con il `-f` essere per costringerlo.

Introduzione di Gradle Wrapper

Gradle ha la possibilità di aggiungere un wrapper ai progetti. Questo wrapper riduce la necessità per tutti gli utenti o sistemi di integrazione continua di avere Gradle installato. Inoltre previene i problemi di versione in cui vi è incompatibilità tra la versione utilizzata dal progetto e quella installata dagli utenti. Lo fa installando una versione di gradle localmente nel progetto.

Gli utenti del progetto eseguono semplicemente:

```
> ./gradlew <task> # on *Nix or MacOSX
> gradlew <task>   # on Windows
```

Per configurare un progetto per utilizzare un wrapper, gli sviluppatori:

1. Eseguire:

```
gradle wrapper [--gradle-version 2.0]
```

Dove `--gradle-version X` è facoltativa e se non viene fornita (o l'attività wrapper non è inclusa, come mostrato di seguito), la versione utilizzata è la versione di gradle in uso.

1. Per forzare il progetto a utilizzare una versione specifica, aggiungi quanto segue a `build.gradle` :

```
task wrapper(type: Wrapper) {
    gradleVersion = '2.0'
}
```

Quando viene eseguito il comando `gradle wrapper` , crea i file:

```
the_project/
  gradlew
  gradlew.bat
  gradle/wrapper/
    gradle-wrapper.jar
    gradle-wrapper.properties
```

La documentazione ufficiale su questa funzione è disponibile su https://docs.gradle.org/current/userguide/gradle_wrapper.html .

Usa Gradle localmente servito nel wrapper Gradle

Se si desidera mantenere la copia locale di Gradle e lasciare che Wrapper lo utilizzi nelle build, è possibile impostare `distributionUrl` punta alla propria copia sull'attività `wrapper` :

```
task wrapper(type: Wrapper) {
    gradleVersion = '2.0'
    distributionUrl = "http\://server/dadada/gradle-${gradleVersion}-bin.zip"
}
```

dopo aver eseguito `gradle wrapper` , viene creato lo script di shell `gradlew` e il `gradle/wrapper/gradle-wrapper.properties` è configurato per utilizzare l'URL fornito per scaricare Gradle.

Usando il Gradle Wrapper dietro un proxy

La prima volta che un utente esegue il `gradlew` un progetto, dovrebbe essere realizzato che farà due cose fondamentali:

1. Controlla se la versione del gradle utilizzata dal wrapper è già in ~ / .gradle / wrapper / dists
2. In caso contrario, scaricare l'archivio della versione da Internet

Se ci si trova in un ambiente che richiede tutto il traffico esterno per passare attraverso un proxy, il passaggio 2 fallirà (a meno che non si tratti di un ambiente proxy trasparente). Di conseguenza, è necessario assicurarsi che siano impostati i parametri del proxy *JVM* .

Ad esempio, se hai una configurazione proxy base senza autenticazione, imposta semplicemente la variabile d'ambiente `JAVA_OPTS` o `GRADLE_OPTS` con:

```
-Dhttps.proxyPort=<proxy_port> -Dhttps.proxyHost=<hostname>
```

Quindi un esempio completo su Windows sarebbe:

```
set JAVA_OPTS=-Dhttps.proxyPort=8080 -Dhttps.proxyHost=myproxy.mycompany.com
```

Se tuttavia l'ambiente richiede anche l'autenticazione, è anche necessario rivedere le altre opzioni su https://docs.oracle.com/javase/8/docs/api/java/net/doc-files/net-properties. html .

*NOTA: questa configurazione proxy è in **aggiunta** a qualsiasi configurazione proxy per l'accesso al repository delle dipendenze.*

Leggi Gradle Wrapper online: https://riptutorial.com/it/gradle/topic/3006/gradle-wrapper

Capitolo 8: Inclusa sorgente nativa - sperimentale

Parametri

parametri	Dettagli
model.android.ndk.toolchain	toolchain nativo trovato nella cartella ndk-bundle

Examples

Configurazione JNI Gradle di base

root: build.gradle

```
buildscript {
    repositories {
        jcenter()
    }
    dependencies {
        classpath 'com.android.tools.build:gradle-experimental:0.8.0-alpha4'
    }
}

allprojects {
    repositories {
        jcenter()
    }
}
```

app: build.gradle

```
apply plugin: 'com.android.model.application'

dependencies {
    compile "com.android.support:support-v4:23.3.0"
    compile fileTree(dir: 'libs', include: '*.jar')
}

model {
    android {
        compileSdkVersion = 23
        buildToolsVersion = '23.0.3'

        defaultConfig {
            applicationId = 'com.example.hello'
            minSdkVersion.apiLevel = 9
            targetSdkVersion.apiLevel = 23

            buildConfigFields {
```

```
                create() {
                    type "int"
                    name "VALUE"
                    value "1"
                }
            }
        }

        ndk {
            platformVersion = 9
            moduleName "hello"

            toolchain "clang"

            stl "gnustl_static"
            CFlags.add("-DANDROID_NDK")
            cppFlags.add("-std=c++11")

            ldLibs.add("android")
            ldLibs.add("dl")
            ldLibs.add("log")
        }

        sources {
            main {
                jni {
                    exportedHeaders {
                        srcDirs "../../common/headers"
                    }
                    source {
                        srcDirs "../../common/src"
                    }
                }
            }
        }
    }
}
```

Utilizzo di librerie predefinite e OpenGL ES 2.0

root: build.gradle

```
buildscript {
    repositories {
        jcenter()
    }
    dependencies {
        classpath 'com.android.tools.build:gradle-experimental:0.8.0-alpha4'
    }
}

allprojects {
    repositories {
        jcenter()
    }
}
```

app: build.gradle

```
apply plugin: 'com.android.model.application'

dependencies {
    compile "com.android.support:support-v4:23.3.0"
    compile fileTree(dir: 'libs', include: '*.jar')
}

model {
    android {
        compileSdkVersion = 23
        buildToolsVersion = '23.0.3'

        defaultConfig {
            applicationId = 'com.example.glworld'
            minSdkVersion.apiLevel = 9
            targetSdkVersion.apiLevel = 23

            buildConfigFields {
                create() {
                    type "int"
                    name "VALUE"
                    value "1"
                }
            }
        }

        buildTypes {
            release {
                minifyEnabled = false
                proguardFiles.add(file('proguard-rules.txt'))
            }
        }

        ndk {
            platformVersion = 9
            moduleName "glworld"

            toolchain "clang"

            stl "gnustl_static"
            CFlags.add("-DANDROID_NDK")
            CFlags.add("-DDISABLE_IMPORTGL")
            CFlags.add("-DFT2_BUILD_LIBRARY=1")
            cppFlags.add("-std=c++11")

            ldLibs.add("EGL")
            ldLibs.add("android")
            ldLibs.add("GLESv2")
            ldLibs.add("dl")
            ldLibs.add("log")
        }

        sources {
            main {
                jni {
                    dependencies {
                        library "freetype2" linkage "shared"
                    }
                    exportedHeaders {
                        srcDirs "../../common/headers"
                    }
```

```
                    source {
                        srcDirs "../../common/src"
                    }
                }
            }
        }
    }

    repositories {
        prebuilt(PrebuiltLibraries) {
            freetype2 {
                headers.srcDir "../../common/freetype2-android/include"
                binaries.withType(SharedLibraryBinary) {
                    def localLib = "../../common/freetype2-android/Android/libs"
                    sharedLibraryFile =
                            file("$localLib/${targetPlatform.getName()}/libfreetype2.so")
                }
            }
        }
    }
}

// The next tasks compile a freetype library using a make file.
// These `.so`'s are then used as the shared libraries compiled above.
tasks.withType(JavaCompile) {
    compileTask -> compileTask.dependsOn buildNative
}

// Call regular ndk-build (.cmd) script from the app directory
task buildNative(type: Exec) {
    def ndkDir = "/Development/android-sdk-macosx/ndk-bundle"
    commandLine "$ndkDir/ndk-build",
            '-C',
            file('../../common/freetype2-android/Android/jni').absolutePath
}

task cleanNative(type: Exec) {
    def ndkDir = "/Development/android-sdk-macosx/ndk-bundle"
    commandLine "$ndkDir/ndk-build",
            '-C',
            file('../../common/freetype2-android/Android/jni').absolutePath,
            "clean"
}

clean.dependsOn cleanNative
```

Leggi Inclusa sorgente nativa - sperimentale online:
https://riptutorial.com/it/gradle/topic/4460/inclusa-sorgente-nativa---sperimentale

Capitolo 9: Inizializzare Gradle

Osservazioni

Terminologia

- Compito : un lavoro atomico eseguito da una build. Le attività hanno `inputs` , `outputs` e dipendenze delle attività.
- `dependencies {}` - Dichiara le dipendenze `File` o binarie necessarie per l'esecuzione delle attività. Ad esempio, `org.slf4j:slf4j-api:1.7.21` è una coordinata abbreviata per una dipendenza Maven.
- `repositories {}` - How Gradle trova i file per le dipendenze esterne. In realtà, solo una raccolta di file organizzati per gruppo, nome e versione. Ad esempio: `jcenter()` è un metodo comodo per `maven { url 'http://jcenter.bintray.com/' } }` , un repository Maven Bintray .

Examples

Inizializzazione di una nuova libreria Java

Prerequisito: installazione di Gradle

Una volta installato Gradle, puoi configurare un progetto nuovo o esistente eseguendo

```
cd $PROJECT_DIR
gradle init --type=java-library
```

Nota che ci sono altri tipi di progetti come Scala con cui puoi iniziare, ma useremo Java per questo esempio.

Finirai con:

```
.
├── build.gradle
├── gradle
│   └── wrapper
│       ├── gradle-wrapper.jar
│       └── gradle-wrapper.properties
├── gradlew
├── gradlew.bat
├── settings.gradle
└── src
    ├── main
    │   └── java
    │       └── Library.java
    └── test
        └── java
            └── LibraryTest.java
```

Ora puoi eseguire `gradle tasks` e vedere che puoi creare un `jar` , eseguire `test` , produrre `javadoc`

e molto altro anche se il tuo file `build.gradle` è:

```
apply plugin: 'java'

repositories {
    jcenter()
}

dependencies {
    compile 'org.slf4j:slf4j-api:1.7.21'
    testCompile 'junit:junit:4.12'
}
```

Leggi Inizializzare Gradle online: https://riptutorial.com/it/gradle/topic/2247/inizializzare-gradle

Capitolo 10: Numero di versione dell'incremento automatico utilizzando lo script Gradle per le applicazioni Android

Examples

Come chiamare il metodo di incremento automatico durante la compilazione

```
gradle.taskGraph.whenReady {taskGraph ->
    if (taskGraph.hasTask(assembleDebug)) {  /* when run debug task */
        autoIncrementBuildNumber()
    } else if (taskGraph.hasTask(assembleRelease)) { /* when run release task */
        autoIncrementBuildNumber()
    }
}
```

Definizione del metodo di incremento automatico

```
 /*Wrapping inside a method avoids auto incrementing on every gradle task run. Now it runs
only when we build apk*/
ext.autoIncrementBuildNumber = {

    if (versionPropsFile.canRead()) {
        def Properties versionProps = new Properties()
        versionProps.load(new FileInputStream(versionPropsFile))
        versionBuild = versionProps['VERSION_BUILD'].toInteger() + 1
        versionProps['VERSION_BUILD'] = versionBuild.toString()
        versionProps.store(versionPropsFile.newWriter(), null)
    } else {
        throw new GradleException("Could not read version.properties!")
    }
}
```

Leggi e assegna il numero di versione da un file di proprietà a una variabile

def versionPropsFile = file ('version.properties') def versionBuild

```
/*Setting default value for versionBuild which is the last incremented value stored in the
file */
if (versionPropsFile.canRead()) {
    def Properties versionProps = new Properties()
    versionProps.load(new FileInputStream(versionPropsFile))
    versionBuild = versionProps['VERSION_BUILD'].toInteger()
} else {
    throw new GradleException("Could not read version.properties!")
}
```

Leggi Numero di versione dell'incremento automatico utilizzando lo script Gradle per le

applicazioni Android online: https://riptutorial.com/it/gradle/topic/10696/numero-di-versione-dell-incremento-automatico-utilizzando-lo-script-gradle-per-le-applicazioni-android

Capitolo 11: Personalizzazione dell'attività IDEA IntelliJ

Sintassi

- groovy.util.Node = node.find {childNode -> return true || falso}
- node.append (nodeYouWantAsAChild)
- groovy.util.Node parsedNode = (nuovo XmlParser ()). parseText (someRawXMLString)
- " 'stringa a più linee (non interpolata)' "

Osservazioni

I tre file di base di un progetto IntelliJ - ipr, iws e iml - sono accessibili come gradle nel task idea tramite

```
project.ipr
module.iml
workspace.iws
```

usando il .withXml puoi accedere a xml. Usando il .asNode () su quello si trasforma in un nodo xml groovy.

Ex:

```
project.ipr.withXml { provider ->
    def node = provider.asNode()
```

Da lì è piuttosto semplice - modificare gradle per configurare i progetti IntelliJ per te, prendere il file all'avvio, eseguire le azioni che desideri gradle (all'interno di IntelliJ), quindi diffare il nuovo file con il vecchio file. Dovresti vedere quale XML ti servirà per personalizzare il lavoro idea. Dovrai anche prendere nota di dove si trova il xml.

Un'altra cosa da considerare è che non si desidera duplicare i nodi all'interno dei file IntelliJ se si esegue l'idea gradle più volte. Quindi, dovrai cercare il nodo che desideri creare e, se non è lì, puoi crearlo e inserirlo.

insidie:

A volte, quando si usa == per il confronto delle stringhe nel metodo find, fallisce. Quando collaudo e trovo che sia il caso, io uso .contains.

Durante la ricerca di nodi, non tutti i nodi hanno l'attributo che stai utilizzando come criterio, quindi assicurati di verificare la presenza di null.

Examples

Aggiungi una configurazione di esecuzione di base

Presupposti per questo esempio:

- Hai una classe, `foo.bar.Baz` .
- Vorresti creare una configurazione di esecuzione che esegua il metodo principale.
- È in un modulo chiamato `fooBar` .

Nel tuo file gradle:

```
idea {
    workspace.iws.withXml { provider ->
        // I'm not actually sure why this is necessary
        def node = provider.asNode()

        def runManager = node.find { it.@name.contains('RunManager')}

        // find a run configuration if it' there already
        def runner = runManager.find { it.find ({ mainClass ->
            return mainClass.@name != null && mainClass.@name == "MAIN_CLASS_NAME" &&
mainClass.@value != null && mainClass.@value.contains('Baz');
        }) != null }

        // create and append the run configuration if it doesn't already exists
        if (runManager != null && runner == null){
            def runnerText = '''
                <configuration default="false" name="Baz" type="Application"
factoryName="Application" nameIsGenerated="true">
                  <extension name="coverage" enabled="false" merge="false" runner="idea">
                    <pattern>
                      <option name="PATTERN" value="foo.bar.Baz" />
                      <option name="ENABLED" value="true" />
                    </pattern>
                  </extension>
                  <option name="MAIN_CLASS_NAME" value="foo.bar.Baz" />
                  <option name="VM_PARAMETERS" value="" />
                  <option name="PROGRAM_PARAMETERS" value="" />
                  <option name="WORKING_DIRECTORY" value="file://$PROJECT_DIR$" />
                  <option name="ALTERNATIVE_JRE_PATH_ENABLED" value="false" />
                  <option name="ALTERNATIVE_JRE_PATH" />
                  <option name="ENABLE_SWING_INSPECTOR" value="false" />
                  <option name="ENV_VARIABLES" />
                  <option name="PASS_PARENT_ENVS" value="true" />
                  <module name="foobar" />
                  <envs />
                  <method />
                </configuration>'''
            runner = (new XmlParser()).parseText(runnerText)
            runManager.append(config);
        }

        // If there is no active run configuration, set the newly made one to be it
        if (runManager != null && runManager.@selected == null) {
            runManager.@selected="${runner.@factoryName}.${runner.@name}"
        }
    }
}
```

Leggi Personalizzazione dell'attività IDEA IntelliJ online:
https://riptutorial.com/it/gradle/topic/2297/personalizzazione-dell-attivita-idea-intellij

Capitolo 12: Script di Gradle Init

Examples

Aggiungi repository predefinito per tutti i progetti

Aggiungi un init.gradle alla cartella gradle dell'utente. Init.gradle è riconosciuto su ogni progetto.

```
Unix: ~/.gradle/init.gradle
```

> Questi sono anche percorsi alternativi dove lo script di init può essere inserito e caricato automaticamente: -
>
> - Qualsiasi file * **.gradle** in **USER_HOME / .gradle / init.d**
> - Qualsiasi file * **.gradle** nella directory **init.d** dell'installazione di Gradle

init.gradle con mavenLocal come repository in tutti i progetti.

```
allprojects {
    repositories {
        mavenLocal()
    }
}
```

Con questo hai a disposizione la cache del tuo maven locale in tutti i repository. Un caso d'uso potrebbe essere quello di usare un jar che si inserisce in "gradle install" in un altro progetto senza aggiungere il repository mavenLocal a build.gradle o aggiungendo un server nexus / artifactory.

Leggi Script di Gradle Init online: https://riptutorial.com/it/gradle/topic/4234/script-di-gradle-init

Capitolo 13: Utilizzando plugin di terze parti

Examples

Aggiunta di un plug-in di terze parti a build.gradle

Gradle (Tutte le versioni) *Questo metodo funziona per tutte le versioni di gradle*

Aggiungi il codice buildscript all'inizio del tuo file build.gradle.

```
buildscript {
  repositories {
    maven {
      url "https://plugins.gradle.org/m2/"
    }
  }
  dependencies {
    classpath "org.example.plugin:plugin:1.1.0"
  }
}

apply plugin: "org.example.plugin"
```

Gradle (Versioni 2.1+) *Questo metodo funziona solo per i progetti che utilizzano Gradle 2.1 o* **versioni** *successive.*

```
plugins {
  id "org.example.plugin" version "1.1.0"
}
```

build.gradle con più plug-in di terze parti

Gradle (Tutte le versioni)

Quando si aggiungono più plug-in di terze parti non è necessario separarli in diverse istanze del codice buildscript (Tutti) o plug-in (2.1+), è possibile aggiungere nuovi plug-in insieme a plug-in preesistenti.

```
buildscript {
  repositories {
    maven {
      url "https://plugins.gradle.org/m2/"
    }
  }
  dependencies {
    classpath "org.example.plugin:plugin:1.1.0"
    Classpath "com.example.plugin2:plugin2:1.5.2"
  }
}

apply plugin: "org.example.plugin"
```

```
apply plugin: "com.example.plugin2"
```

Gradle (versioni 2.1+)

```
plugins {
  id "org.example.plugin" version "1.1.0"
  id "com.example.plugin2" version "1.5.2"
}
```

Leggi Utilizzando plugin di terze parti online: https://riptutorial.com/it/gradle/topic/9183/utilizzando-plugin-di-terze-parti

Titoli di coda

S. No	Capitoli	Contributors
1	Iniziare con Gradle	Afterfield, bassim, Community, Emil Burzo, Eric Wendelin, Hamzawey, Hillkorn, Matthias Braun, Nikem, Pepper Lebeck-Jobe, Sergey Yakovlev, Stanislav, user2555595, vanogrid, Will
2	Compiti di ordinazione	Gabriele Mariotti
3	dipendenze	Afshin, Andrii Abramov, GameScripting, Hillkorn, leeor, Matthias Braun, mcarlin, mszymborski, Will
4	Dipendenze delle attività	Gabriele Mariotti, Sergey Yakovlev, Stanislav
5	Gradle Performance	ambes, Sergey Yakovlev, Will
6	Gradle Plugin	Gabriele Mariotti, JBirdVegas
7	Gradle Wrapper	ajoberstar, Fanick, HankCa, I Stevenson
8	Inclusa sorgente nativa - sperimentale	iHowell
9	Inizializzare Gradle	Eric Wendelin, Will
10	Numero di versione dell'incremento automatico utilizzando lo script Gradle per le applicazioni Android	Jayakrishnan PM
11	Personalizzazione dell'attività IDEA IntelliJ	IronHorse, Sam Sieber, Will
12	Script di Gradle Init	ambes, Hillkorn
13	Utilizzando plugin di terze parti	Afterfield