Kostenloses eBook

LERNEN gradle

Free unaffiliated eBook created from
Stack Overflow contributors.

#gradle

Inhaltsverzeichnis

Über

You can share this PDF with anyone you feel could benefit from it, downloaded the latest version from: gradle

It is an unofficial and free gradle ebook created for educational purposes. All the content is extracted from Stack Overflow Documentation, which is written by many hardworking individuals at Stack Overflow. It is neither affiliated with Stack Overflow nor official gradle.

Use the content presented in this book at your own risk; it is not guaranteed to be correct nor accurate, please send your feedback and corrections to info@zzzprojects.com

Kapitel 1: Erste Schritte mit gradle

Bemerkungen

Gradle ist ein Open-Source-Werkzeug für allgemeine Zwecke. Es ist beliebt in der Java-Community und ist das bevorzugte Build-Tool für Android .

Hervorgehobene Gradle-Funktionen

- Deklarative Build-Skripts *sind* in Groovy oder Kotlin geschriebener Code.
- Viele Core- und Community-Plugins, die einen flexiblen, auf Konventionen basierenden Ansatz verwenden
- Inkrementelle Builds , bei denen Aufgaben, deren Abhängigkeiten nicht geändert wurden, nicht erneut ausgeführt werden.
- Integrierte Abhängigkeitsauflösung für Maven und Ivy . Eingeschriebene Plugins bieten eine Abhängigkeitsauflösung von anderen `repositories` wie npm .
- Erstklassige Multiprojekt-Builds.
- Integration mit anderen Build-Tools wie Maven , Ant und anderen.
- Build-Scans , die die Fähigkeit der Entwickler verbessern , an Gradle-Builds zusammenzuarbeiten und diese zu optimieren.

Mehr Informationen

Wenn Sie mehr über die Gradle-Funktionen erfahren möchten, lesen Sie den Überblick- Teil des Gradle-Benutzerhandbuchs .

Wenn Sie Gradle ausprobieren möchten, können Sie sich die Anleitungen hier ansehen . Sie können eine Java-Kurzanleitung lesen, erfahren, wie Sie Gradle zum ersten Mal verwenden und von einem anderen Build-Tool aus migrieren.

Examples

Gradle Installation

Voraussetzungen: Installiertes Java JDK oder JRE (Version 7 oder höher für Gradle 3.x-Version)

Installationsschritte:

1. Laden Sie die Gradle-Distribution von der offiziellen Website herunter
2. Packen Sie die ZIP aus
3. Fügen Sie die Umgebungsvariable `GRADLE_HOME` . Diese Variable sollte auf die entpackten Dateien aus dem vorherigen Schritt zeigen.
4. Fügen `GRADLE_HOME/bin` Ihrer `PATH` Umgebungsvariablen `GRADLE_HOME/bin` , damit Sie Gradle über die Befehlszeilenschnittstelle (CLI) ausführen können
5. Testen Sie Ihre Gradle-Installation, indem Sie `gradle -v` in die CLI eingeben. Die Ausgabe

sollte die installierte Gradle-Version und die aktuellen Gradle-Konfigurationsdetails enthalten

Weitere Informationen finden Sie in der offiziellen Benutzeranleitung

Installation mit Homebrew unter OS X / macOS

Benutzer von Homebrew können Gradle durch Ausführen installieren

```
brew install gradle
```

Installation mit SdkMan

Benutzer von SdkMan können Gradle installieren, indem Sie Folgendes ausführen :

```
sdk install gradle
```

Bestimmte Version installieren

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

Versionen wechseln

```
sdk use gradle 2.12
```

Installieren Sie das Gradle-Plugin für Eclipse

Zum Installieren des Gradle-Plugins in Eclipse sind folgende Schritte erforderlich:

1. Öffnen Sie Eclipse und gehen Sie zu **Hilfe** -> **Eclipse Marketplace**
2. **Geben Sie** in der Suchleiste **buildship ein** und **drücken Sie** die Eingabetaste
3. Wählen Sie **"Buildship Gradle Integration 1.0"** und klicken Sie auf **Installieren**
4. Klicken Sie im nächsten Fenster auf **Bestätigen**
5. **Akzeptieren Sie dann** die Bedingungen und die Lizenz der Vereinbarung und klicken Sie dann auf **Fertig stellen**
6. Nach der Installation muss Eclipse neu gestartet werden. Klicken Sie auf **Ja**

Hallo Welt

Gradle-Tasks können mit Groovy-Code aus einer build.gradle-Datei eines Projekts geschrieben werden. Diese Tasks können dann mit `> gradle [taskname]` am Terminal oder durch Ausführen der Task innerhalb einer IDE wie Eclipse ausgeführt werden.

Um das Hello World-Beispiel in Gradle zu erstellen, müssen Sie eine Aufgabe definieren, die eine Zeichenfolge mit Groovy an die Konsole druckt. Wir werden `printLn` , um die Java-Methode `System.out.printLn` aufzurufen, um den Text auf der Konsole zu drucken.

build.gradle

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

Wir können diese Aufgabe dann mit `> gradle hello` oder `> gradle -q hello` . Mit `-q` werden unterdrückte Protokollnachrichten unterdrückt, sodass nur die Ausgabe der Aufgabe angezeigt wird.

Ausgabe von `> gradle -q hello` :

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

Mehr zu Aufgaben

Vor allem: Operator `<<` (leftShift) entspricht `doLast {closure}` . Ab **Grad 3.2 ist** es **veraltet** . Der gesamte **Taskcode wird** in ein **build.gradle geschrieben** .

> Eine Aufgabe stellt eine atomare Arbeit dar, die ein Build ausführt. Dies kann das Kompilieren einiger Klassen sein, das Erstellen einer JAR-Datei, das Generieren von Javadoc oder das Veröffentlichen einiger Archive in einem Repository.

Gradle unterstützt zwei große Aufgabentypen: einfache und erweiterte Aufgaben.

Betrachten wir einige Aufgabendefinitionsstile:

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

Oder der:

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

Diese Aufgaben sind äquivalent. Sie können aber auch einige Erweiterungen für die Aufgabe zur Verfügung stellen, wie zum Beispiel: `dependsOn` , `mustRunAfter` , `type` usw. Sie Aufgabe durch Hinzufügen von Aktionen nach Aufgabendefinition erweitern können, wie folgt aus :

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

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

Wenn wir das ausführen, haben wir:

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

Fragen zu Aufgabenabhängigkeiten und Reihenfolge werden hier geprüft

Sprechen wir über zwei große Arten von Aufgaben.

Einfach:

Aufgaben, die wir mit einem Aktionsabschluss definieren:

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

Erweitert

Verbessert ist es eine Aufgabe mit vorkonfiguriertem Verhalten. Alle Plugins, die Sie in Ihrem Projekt verwenden, sind *erweiterte* oder **erweiterte Aufgaben** . Lassen Sie uns unsere erstellen und Sie werden verstehen, wie es funktioniert:

```
task hello(type: HelloTask)

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

Außerdem können wir unserer Aufgabe Parameter wie folgt übergeben:

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

```
        println greeting
    }
}
```

Und ab jetzt können wir unsere Aufgabe so umschreiben:

```
   //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!'
}
```

Wenn wir das ausführen, haben wir:

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

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

Alle Fragen zu Development Gradle Plugins auf der offiziellen Website

Erste Schritte mit gradle online lesen: https://riptutorial.com/de/gradle/topic/894/erste-schritte-mit-gradle

Kapitel 2: Abhängigkeiten

Examples

Fügen Sie eine lokale JAR-Dateiabhängigkeit hinzu

Einzelnes Glas

Manchmal haben Sie eine lokale JAR-Datei, die Sie Ihrem Gradle-Build als Abhängigkeit hinzufügen müssen. So können Sie das machen:

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

Dabei ist `path` ein Verzeichnispfad in Ihrem Dateisystem und `local_dependency.jar` der Name Ihrer lokalen JAR-Datei. Der `path` kann relativ zur Build-Datei sein.

Verzeichnis der JARs

Es ist auch möglich, ein Verzeichnis von Gläsern zum Kompilieren hinzuzufügen. Dies kann wie folgt gemacht werden:

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

Wo wären `libs` das Verzeichnis, in dem sich die Jars befinden, und `*.jar` wäre der Filter der `*.jar` Dateien.

Verzeichnis der JARs als Repository

Wenn Sie nur Gläser in einem Repository suchen möchten, anstatt sie direkt als Abhängigkeit mit ihrem Pfad hinzuzufügen, können Sie ein flatDir-Repository verwenden.

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

Sucht nach jars im *`libs`* Verzeichnis und dessen untergeordneten Verzeichnissen.

Fügen Sie eine Abhängigkeit hinzu

Abhängigkeiten in Gradle folgen dem gleichen Format wie Maven . Abhängigkeiten sind wie folgt aufgebaut:

```
group:name:version
```

Hier ist ein Beispiel:

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

Fügen Sie diese Zeile in Ihrem `dependency` in der Gradle-Builddatei ein, um sie als Kompilierzeitabhängigkeit hinzuzufügen:

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

Eine alternative Syntax hierfür benennt jede Komponente der Abhängigkeit explizit wie folgt:

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

Dies fügt eine Abhängigkeit zur Kompilierzeit hinzu.

Sie können Abhängigkeiten auch nur für Tests hinzufügen. Hier ist ein Beispiel:

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

Verlassen Sie sich auf ein anderes Gradle-Projekt

Im Falle eines Gradle-Builds für mehrere Projekte müssen Sie manchmal auf ein anderes Projekt in Ihrem Build angewiesen sein. Um dies zu erreichen, geben Sie Folgendes in die Abhängigkeiten Ihres Projekts ein:

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

Dabei ist `':OtherProject'` der Abstufungspfad für das Projekt, der vom Stamm der Verzeichnisstruktur aus referenziert wird.

Um `':OtherProject'` in Zusammenhang mit der `build.gradle` Datei fügen Sie dies in dem entsprechenden `settings.gradle`

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

Eine ausführlichere Erklärung finden Sie hier .

Abhängigkeiten auflisten

Durch Aufrufen der `dependencies` können Sie die Abhängigkeiten des Stammprojekts anzeigen:

```
gradle dependencies
```

Die Ergebnisse sind Abhängigkeitsdiagramme (unter Berücksichtigung transitiver Abhängigkeiten), aufgeschlüsselt nach Konfiguration. Um die angezeigten Konfigurationen einzuschränken, können Sie die Option `--configuration` gefolgt von einer ausgewählten Konfiguration zur Analyse übergeben:

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

Um Abhängigkeiten eines Unterprojekts anzuzeigen, verwenden Sie die Task `<subproject>:dependencies` . Um beispielsweise Abhängigkeiten eines Unterprojekts mit dem Namen `api` :

```
gradle api:dependencies
```

Repositories hinzufügen

Sie müssen Gradle auf die Position Ihrer Plugins zeigen, damit Gradle sie finden kann. Dazu fügen Sie Ihrem `build.gradle` ein `repositories { ... } build.gradle` .

Hier ein Beispiel für das Hinzufügen von drei Repositorys, JCenter , Maven Repository und eines benutzerdefinierten Repositorys, das Abhängigkeiten im Maven-Stil bietet.

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

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

Hinzufügen einer .aar-Datei zum Android-Projekt mit Gradle

1. Navigieren Sie zum `app` Modul des Projekts und erstellen Sie das `libs` Verzeichnis.
2. `.aar` Ihre `.aar` Datei dort ab. Zum Beispiel `myLib.aar` .
3. Fügen Sie den folgenden Code zum `android` Block der `build.gradle` Datei der `app` Ebene `build.gradle` .

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

Auf diese Weise haben Sie ein neues zusätzliches Repository definiert, das auf den `libs` Ordner des `app` Moduls verweist.

4. Fügen Sie den folgenden Code zum Block für `dependencies` oder zur Datei `build.gradle` :

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

Abhängigkeiten online lesen: https://riptutorial.com/de/gradle/topic/2524/abhangigkeiten

Kapitel 3: Aufgaben bestellen

Bemerkungen

> Bitte beachten Sie, dass `mustRunAfter` und `shouldRunAfter` als "inkubierend" (ab Gradle 3.0) gekennzeichnet sind. Dies bedeutet, dass es sich hierbei um experimentelle Merkmale handelt, deren Verhalten in zukünftigen Versionen geändert werden kann.

Es gibt zwei Bestellregeln:

- `mustRunAfter`
- `shouldRunAfter`

Wenn Sie die `mustRunAfter` Anordnungsregel verwenden, geben Sie an, dass taskB immer nach taskA ausgeführt werden muss, wenn sowohl taskA als auch taskB ausgeführt werden.

Die Regel von `shouldRunAfter` ist ähnlich, aber weniger streng, da sie in zwei Situationen ignoriert wird:

- Wenn Sie diese Regel verwenden, wird ein Bestellzyklus eingeleitet.
- Wenn die parallele Ausführung verwendet wird und alle Abhängigkeiten einer Task abgesehen von der Task „ `shouldRunAfter` erfüllt wurden, wird diese Task unabhängig davon ausgeführt, ob die Abhängigkeiten von `shouldRunAfter` ausgeführt wurden oder nicht.

Examples

Bestellung mit der Methode mustRunAfter

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

B.mustRunAfter A
```

`B.mustRunAfter A` Zeile `B.mustRunAfter A` wird Gradle `B.mustRunAfter A` , Task nach dem als Argument angegebenen Task auszuführen.

Und die Ausgabe ist:

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

Die Bestellregel führt keine Abhängigkeit zwischen den A- und B-Tasks ein, sondern wirkt sich nur aus, wenn **beide Tasks** zur Ausführung geplant sind.

Das bedeutet, dass wir die Aufgaben A und B unabhängig voneinander ausführen können.

Die Ausgabe ist:

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

Aufgaben bestellen online lesen: https://riptutorial.com/de/gradle/topic/5550/aufgaben-bestellen

Kapitel 4: Aufgabenabhängigkeiten

Bemerkungen

doLast

Beachten Sie , dass in einem gradle 3.x mehr idiomatischen Weg Aufgabendefinition: **explizite doLast** mit **{} Schließung** Notation statt „Leftshift“ (<<) Operator bevorzugt **(Leftshift** in einer gradle 3.2 veraltet ist geplant in gradle 5.0 entfernt werden. .)

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

ist äquivalent zu:

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

Examples

Hinzufügen von Abhängigkeiten mithilfe von Tasknamen

Wir können die Ausführungsreihenfolge von Tasks mit der `dependsOn` .

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

Hinzufügen von 'hängt von: Ursachen ab:

- Aufgabe B hängt von Aufgabe A ab
- Absolvieren Sie eine `A` Task immer **vor** der `B` Task-Ausführung.

Und die Ausgabe ist:

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

Abhängigkeiten aus einem anderen Projekt hinzufügen

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

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

Um auf eine Aufgabe in einem anderen Projekt zu verweisen, geben Sie dem Pfad des Projekts, zu dem er gehört, **den Namen der Aufgabe** an `:projectB:B`

Und die Ausgabe ist:

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

Abhängigkeit mithilfe eines Aufgabenobjekts hinzufügen

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

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

B.dependsOn A
```

Es ist eine alternative Methode, um die Abhängigkeit zu definieren, anstatt den Aufgabennamen zu verwenden .

Und die Ausgabe ist die gleiche:

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

Mehrere Abhängigkeiten hinzufügen

Sie können mehrere Abhängigkeiten hinzufügen.

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

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

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

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

Jetzt können Sie eine Reihe von Abhängigkeiten definieren:

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

Die Ausgabe ist:

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

Anderes Beispiel:

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

Die Ausgabe ist:

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

Mehrere Abhängigkeiten mit der abhängigen Methode

Sie können mehrere Abhängigkeiten hinzufügen.

```
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'
}
```

Die Ausgabe ist:

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

Aufgabenabhängigkeiten online lesen:
https://riptutorial.com/de/gradle/topic/5545/aufgabenabhangigkeiten

Kapitel 5: Einschließlich nativer Quelle - experimentell

Parameter

Parameter	Einzelheiten
model.android.ndk.toolchain	native toolchain im ndk-bundle-ordner

Examples

Grundlegende JNI Gradle Config

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"
                    }
                }
            }
        }
    }
}
```

Verwenden von vorgefertigten Bibliotheken und 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
```

Einschließlich nativer Quelle - experimentell online lesen:
https://riptutorial.com/de/gradle/topic/4460/einschlie-lich-nativer-quelle---experimentell

Kapitel 6: Gradle Init-Skripte

Examples

Fügen Sie für alle Projekte ein Standard-Repository hinzu

Fügen Sie Ihrem Benutzer-Gradle-Ordner ein init.gradle hinzu. Das init.gradle wird in jedem Projekt erkannt.

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

> Dies sind auch alternative Speicherorte, an denen das Init-Skript automatisch platziert und geladen werden kann: -
>
> - Beliebige * **.gradle-** Datei in **USER_HOME / .gradle / init.d**
> - Beliebige * **.gradle-** Datei im **init.d-** Verzeichnis der Gradle-Installation

init.gradle mit mavenLocal als Repository in allen Projekten.

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

Damit haben Sie Ihren lokalen Maven-Cache in allen Repositorys zur Verfügung. Ein Anwendungsfall könnte darin bestehen, ein jar zu verwenden, das Sie mit "gradle install" in einem anderen Projekt einsetzen, ohne das mavenLocal-Repository zum build.gradle oder einen nexus / artifactory-Server hinzuzufügen.

Gradle Init-Skripte online lesen: https://riptutorial.com/de/gradle/topic/4234/gradle-init-skripte

Kapitel 7: Gradle Performance

Examples

Erstellen eines Profils

Bevor Sie mit der Optimierung Ihres Gradle-Builds beginnen, sollten Sie eine Baseline festlegen und herausfinden, welche Teile des Builds die meiste Zeit in Anspruch nehmen. Dazu können Sie Ihr Build profilieren, indem Sie dem Gradle-Befehl das Argument `--profile` hinzufügen:

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

Nachdem der Build abgeschlossen ist, wird unter `./build/reports/profile/` ein HTML-Profilbericht für den Build `./build/reports/profile/` . Dieser sieht etwa wie `./build/reports/profile/` :

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

Wenn Sie auf die Registerkarten neben **Zusammenfassung**

klicken, können Sie eine detailliertere Aufschlüsselung des Zeitaufwands anzeigen.

Konfigurieren Sie bei Bedarf

Wenn die Profilerstellung Ihres Builds viel Zeit für das **Konfigurieren von Projekten bedeutet** , kann die Option Auf Anforderung konfigurieren die Leistung verbessern.

Sie können den On-Demand-Modus `$GRADLE_USER_HOME/.gradle/gradle.properties` indem Sie `$GRADLE_USER_HOME/.gradle/gradle.properties` (standardmäßig `~/.gradle/gradle.properties`) `org.gradle.configureondemand` und `org.gradle.configureondemand` .

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

Um es nur für ein bestimmtes Projekt zu aktivieren, bearbeiten `gradle.properties` stattdessen die Datei `gradle.properties` dieses Projekts.

Wenn Nach Bedarf konfigurieren aktiviert ist, konfiguriert Gradle nicht alle Projekte im Voraus, sondern konfiguriert nur Projekte, die für die auszuführende Aufgabe erforderlich sind.

Aus dem Gralde-Handbuch :

> Der Modus "Configuration on Demand" versucht, nur Projekte zu konfigurieren, die für angeforderte Aufgaben relevant sind, dh er führt nur die `build.gradle` Datei von Projekten aus, die am Build teilnehmen. Auf diese Weise kann die Konfigurationszeit eines großen Multiprojekt-Builds reduziert werden. Langfristig wird dieser Modus zum Standardmodus, möglicherweise der einzige Modus für die Ausführung von Gradle Build.

Anpassen der JVM-Speicherverwendungsparameter für Gradle

Sie können die für Gradle-Builds und den Gradle-Daemon verwendeten Speicherbenutzungsgrenzwerte (oder andere JVM-Argumente) festlegen oder erhöhen, indem Sie `$GRADLE_USER_HOME/.gradle/gradle.properties` (standardmäßig `~/.gradle/gradle.properties`) `~/.gradle/gradle.properties` und die `org.gradle.jvmargs` .

Um diese Grenzwerte nur für ein bestimmtes Projekt zu konfigurieren, bearbeiten `gradle.properties` stattdessen die Datei `gradle.properties` dieses Projekts.

Die Standardeinstellungen für die Speichernutzung für Gradle-Builds und den Gradle-Daemon sind:

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

Dies ermöglicht eine allgemeine maximale Speicherzuordnung (Heap-Größe) von 1 GB und eine maximale Speicherzuordnung für permanente "interne" Objekte von 256 MB. Wenn diese Größen erreicht werden, erfolgt eine Garbage Collection, wodurch die Leistung erheblich beeinträchtigt werden kann.

Angenommen, Sie haben den Speicher übrig, könnten Sie diese wie folgt verdoppeln:

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

Beachten Sie, dass Sie nicht mehr von einer Erhöhung von `XX:MaxPermSize` profitieren können `XX:MaxPermSize` früher, als wenn `Xmx` zunimmt, wird der Nutzen zunehmen.

Benutze den Gradle Daemon

Sie können den Gradle-Daemon aktivieren, um die Leistung Ihrer Builds zu verbessern.

Der Gradle-Daemon lässt das Gradle-Framework initialisiert und ausgeführt werden und speichert Projektdaten im Arbeitsspeicher, um die Leistung zu verbessern.

Für einen einzelnen Build

Um den Daemon für einen einzelnen Build zu aktivieren, können Sie einfach den Pass `--daemon` Argument zu Ihrem `gradle` Befehl oder Gradle Wrapper - Skript.

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

Für alle Builds eines Projekts

Um den Daemon für alle Builds eines Projekts zu aktivieren, können Sie Folgendes hinzufügen:

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

In die Datei `gradle.properties` Ihres Projekts.

Für alle Builds

Um den Gradle-Daemon standardmäßig zu aktivieren, bearbeiten `$GRADLE_USER_HOME/.gradle/gradle.properties` für jedes Build, das von Ihrem Benutzerkonto in Ihrem System erstellt wurde, `$GRADLE_USER_HOME/.gradle/gradle.properties` (standardmäßig `~/.gradle/gradle.properties`) und fügen Sie diese Zeile hinzu:

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

Sie können dies auch in einem einzigen Befehl auf Mac / Linux / * nix-Systemen tun:

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

Oder unter Windows:

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

Deaktivieren des Daemons

Sie können den Daemon für ein bestimmtes Build mit dem Argument `--no-daemon` deaktivieren oder für ein bestimmtes Projekt deaktivieren, indem Sie `org.gradle.daemon=false` explizit in der Datei `gradle.properties` des Projekts `gradle.properties` .

Den Daemon stoppen

Wenn Sie einen Daemon-Prozess manuell beenden möchten, können Sie den Prozess entweder über den Task-Manager Ihres Betriebssystems `gradle --stop` oder den `gradle --stop` . Mit `--stop` Schalter `--stop` Gradle an, dass alle `--stop` Daemon-Prozesse derselben Gradle-Version, die zum Ausführen des Befehls verwendet wurde, sich selbst beenden. Normalerweise beenden sich Daemon-Prozesse automatisch * nach * *3 Stunden Inaktivität oder weniger* .

Gradle Parallel baut auf

Gradle führt unabhängig von der Projektstruktur standardmäßig immer nur eine Aufgabe aus. Mit der `--parallel` können Sie Gradle zwingen, unabhängige Teilprojekte parallel auszuführen, `--parallel` solche, die keine impliziten oder expliziten Projektabhängigkeiten aufweisen, sodass mehrere Aufgaben gleichzeitig ausgeführt werden können verschiedene Projekte.

Projekte parallel erstellen:

```
gradle build --parallel
```

Sie können das parallele Erstellen auch als Standard für ein Projekt festlegen, indem Sie die folgende Einstellung zur Datei gradle.properties des Projekts hinzufügen:

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

Verwenden Sie die neueste Version von Gradle

Das Gradle-Team arbeitet regelmäßig an der Verbesserung der Leistung verschiedener Aspekte von Gradle-Builds. Wenn Sie eine alte Version von Gradle verwenden, verpassen Sie die Vorteile dieser Arbeit. Versuchen Sie, ein Upgrade auf die neueste Version von Gradle durchzuführen, um zu sehen, welche Auswirkungen dies hat. Dies ist ein geringes Risiko, da nur sehr wenige Dinge zwischen den kleineren Versionen von Gradle auftreten.

Die Eigenschaftendatei für den Gradle-Wrapper befindet sich in Ihrem Projektordner unter `gradle/wrapper/` und heißt `gradle-wrapper.properties` . Der Inhalt dieser Datei könnte folgendermaßen aussehen:

```
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
```

Sie können die Versionsnummer `xxx` (aktuelle Version) manuell in `yyy` (neuere Version) ändern. `yyy` Sie den Wrapper das nächste Mal ausführen, wird die neue Version automatisch heruntergeladen.

Gradle Performance online lesen: https://riptutorial.com/de/gradle/topic/3443/gradle-performance

Kapitel 8: Gradle Plugins

Examples

Einfaches Gradle-Plugin von `buildSrc`

Ein einfaches Beispiel, wie Sie ein benutzerdefiniertes Plugin und DSL für Ihr Gradle-Projekt erstellen.
Dieses Beispiel verwendet eine der drei Möglichkeiten, Plugins zu erstellen.
Die drei Möglichkeiten sind:

- in der Reihe
- buildSrc
- eigenständige Plugins

Dieses Beispiel zeigt das Erstellen eines Plugins aus dem Ordner **buildSrc** .

In diesem Beispiel werden fünf Dateien erstellt

```
// 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
```

Mit diesem Setup können wir die vom Anrufer in Ihrem DSL-Block gelieferten Werte sehen

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

Wie schreibe ich ein eigenständiges Plugin?

Um ein benutzerdefiniertes eigenständiges Gradle-Plug-In mit Java zu erstellen (Sie können auch Groovy verwenden), müssen Sie eine Struktur wie folgt erstellen:

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

Richten Sie die Konfiguration ein

In der Datei `build.gradle` definieren Sie Ihr Projekt.

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

dependencies {
    compile gradleApi()
}
```

Das `java` Plugin wird zum Schreiben von Java-Code verwendet.
Die Abhängigkeit von `gradleApi()` gibt uns alle Methoden und Eigenschaften, die zum Erstellen eines Gradle-Plugins erforderlich sind.

In der Datei " `settings.gradle` ":

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

Es definiert die **Artefakt-ID** in Maven.
Wenn die Datei " `settings.gradle` " nicht im Plugin-Verzeichnis vorhanden ist, ist der Standardwert der Name des Verzeichnisses.

Erstellen Sie das Plugin

Definieren Sie eine Klasse in der `src/main/java/org/sample/MyPlugin.java` die die `Plugin` Schnittstelle implementiert.

```
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);
    }

}
```

Definieren Sie die Task, die die `DefaultTask` Klasse erweitert:

```
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");
    }
}
```

Deklaration der Plugin-Klasse

Im Ordner `META-INF/gradle-plugins` Sie eine Eigenschaftendatei erstellen, die `implementation-class` Eigenschaft der `implementation-class` definiert, die die `implementation-class` `META-INF/gradle-plugins` identifiziert.

In den `META-INF/gradle-plugins/testplugin.properties`

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

Beachten Sie, dass der **Eigenschaftendateiname mit der Plugin-ID übereinstimmt** .

Wie man es baut und veröffentlicht

Ändern Sie die Datei `build.gradle` fügen Sie einige Informationen hinzu, um das Plugin in einem Maven-Repo hochzuladen:

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

dependencies {
    compile gradleApi()
```

```
}

repositories {
    jcenter()
}


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

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

Sie können das Gradle-Plug-in mit dem folgenden Befehl erstellen und im Maven-Repo veröffentlichen, das in der Datei `plugin/build.gradle` definiert ist.

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

Wie man es benutzt

So verwenden Sie das Plugin-Add im `build.gradle` Ihres Projekts:

```
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
```

Dann können Sie die Aufgabe aufrufen mit:

```
 $ ./gradlew myTask
```

Gradle Plugins online lesen: https://riptutorial.com/de/gradle/topic/1900/gradle-plugins

Kapitel 9: Gradle wird initialisiert

Bemerkungen

Terminologie

- Aufgabe - eine atomare Arbeit, die ein Build ausführt. Aufgaben haben `inputs` , `outputs` und Aufgabenabhängigkeiten.
- `dependencies {}` - Deklariert `File` oder Binärabhängigkeiten, die zur Ausführung von Aufgaben erforderlich sind. Zum Beispiel `org.slf4j:slf4j-api:1.7.21` ist eine Abkürzung Koordinaten zu einer Maven Abhängigkeit.
- `repositories {}` - Wie Grads Dateien für externe Abhängigkeiten findet. Wirklich nur eine Sammlung von Dateien, sortiert nach Gruppe, Name und Version. Beispiel: `jcenter()` ist eine praktische Methode für `maven { url 'http://jcenter.bintray.com/' } }` , ein Bintray Maven-Repository .

Examples

Initialisieren einer neuen Java-Bibliothek

Voraussetzung: Gradle installieren

Nachdem Sie Gradle installiert haben, können Sie ein neues oder ein vorhandenes Projekt einrichten, indem Sie es ausführen

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

Beachten Sie, dass es andere Projekttypen wie Scala gibt, mit denen Sie anfangen können. In diesem Beispiel wird jedoch Java verwendet.

Sie werden am Ende mit:

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

```
            └── LibraryTest.java
```

Sie können jetzt laufen `gradle tasks` und sehen Sie, dass Sie einen bauen kann `jar` , führen `test` s, produzieren `javadoc` s und vieles mehr , obwohl Ihre `build.gradle` Datei ist:

```
apply plugin: 'java'

repositories {
    jcenter()
}

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

Gradle wird initialisiert online lesen: https://riptutorial.com/de/gradle/topic/2247/gradle-wird-initialisiert

Kapitel 10: Gradle Wrapper

Examples

Gradle Wrapper und Git

Wie in der Einführung beschrieben, funktioniert die Gradle-Wrapper-Funktion, weil in das Projekt eine JAR- `gradlew` wird, die beim `gradlew` Befehls `gradlew` wird. Dies kann jedoch nicht festgeschrieben werden. Nachdem das Projekt das nächste Mal `gradlew` wird `gradlew` nicht mit dem Fehler ausgeführt:

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

Dies liegt daran, dass Ihre .gitignore `*jar` -Datei wahrscheinlich für Java-Projekte enthalten wird. Wenn der Gradle-Wrapper initialisiert wurde, wird er in die Datei `gradle/wrapper/gradle-wrapper.jar` . Daher müssen Sie es dem Git-Index hinzufügen und es festschreiben. Tun Sie dies mit:

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

Mit dem `-f` zu zwingen.

Gradle Wrapper Einführung

Gradle kann Projekten einen Wrapper hinzufügen. Mit diesem Wrapper wird die Installation von Gradle für alle Benutzer oder Systeme mit kontinuierlicher Integration vermieden. Außerdem werden Versionsprobleme vermieden, bei denen eine Inkompatibilität zwischen der vom Projekt verwendeten Version und der von den Benutzern installierten Version besteht. Dazu wird eine Version von Gradle lokal im Projekt installiert.

Benutzer des Projekts werden einfach ausgeführt:

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

So richten Sie ein Projekt für die Verwendung eines Wrappers ein:

1. Ausführen:

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

Wenn `--gradle-version X` optional ist und wenn nicht angegeben (oder die Wrapper-Task nicht enthalten ist, wie unten gezeigt), ist die verwendete Version die Version von Gradle.

1. Um zu erzwingen, dass das Projekt eine bestimmte Version verwendet, fügen Sie dem `build.gradle` Folgendes `build.gradle` :

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

Wenn der Befehl `gradle wrapper` ausgeführt wird, werden die Dateien erstellt:

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

Die offizielle Dokumentation zu dieser Funktion finden Sie unter https://docs.gradle.org/current/userguide/gradle_wrapper.html .

Verwenden Sie im Gradle-Wrapper lokal servierte Gradle

Wenn Sie eine lokale Kopie des Gradle behalten und den Wrapper in den Builds verwenden lassen möchten, können Sie die `distributionUrl` für Ihre Kopie in der `wrapper` Task `wrapper` :

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

Nach dem Ausführen des `gradle wrapper` wird das Shell-Skript `gradlew` erstellt, und `gradle/wrapper/gradle-wrapper.properties` ist so konfiguriert, dass die bereitgestellte URL zum Herunterladen des Gradle verwendet wird.

Verwenden des Gradle-Wrappers hinter einem Proxy

`gradlew` ein Benutzer zum ersten Mal die `gradlew` eines Projekts `gradlew` , sollte klar sein, dass er zwei wichtige Dinge tun wird:

1. Prüfen Sie, ob die Version des von dem Wrapper verwendeten Gradels bereits in ~ / .gradle / wrapper / dists enthalten ist
2. Wenn nicht, laden Sie das Archiv der Version aus dem Internet herunter

Wenn Sie sich in einer Umgebung befinden, in der der gesamte externe Datenverkehr einen Proxy durchläuft, schlägt der zweite Schritt fehl (es sei denn, es handelt sich um eine transparente Proxy-Umgebung). Aus diesem Grund müssen Sie sicherstellen, dass die *JVM*- Proxy-Parameter festgelegt sind.

Wenn Sie beispielsweise über ein einfaches Proxy-Setup ohne Authentifizierung verfügen, setzen Sie einfach die Umgebungsvariable `JAVA_OPTS` oder `GRADLE_OPTS` mit:

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

Ein abgeschlossenes Beispiel für Windows wäre also:

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

Wenn Ihre Umgebung jedoch auch eine Authentifizierung erfordert, sollten Sie auch Ihre anderen Optionen unter https://docs.oracle.com/javase/8/docs/api/java/net/doc-files/net-properties überprüfen . html .

*ANMERKUNG: Diese Proxy-Konfiguration wird **zusätzlich** zu einer beliebigen Proxy-Konfiguration für den Zugriff auf Ihr Repository für Abhängigkeiten verwendet.*

Gradle Wrapper online lesen: https://riptutorial.com/de/gradle/topic/3006/gradle-wrapper

Kapitel 11: IntelliJ IDEA Aufgabenanpassung

Syntax

- groovy.util.Node = node.find {childNode -> return true || falsch
- node.append (nodeYouWantAsAChild)
- groovy.util.Node parsedNode = (neuer XmlParser ()). parseText (someRawXMLString)
- " 'mehrzeiliger String (nicht interpoliert)' "

Bemerkungen

Auf die drei grundlegenden Dateien eines IntelliJ-Projekts - die Dateien ipr, iws und iml - kann wie in der Ideenaufgabe schrittweise zugegriffen werden

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

Mit der .withXml können Sie auf die XML-Datei zugreifen. Durch die Verwendung von .asNode () wird daraus ein grooviger XML-Knoten.

Ex:

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

Von da aus ist es ziemlich einfach - Gradle zu ändern, um IntelliJ-Projekte für Sie zu konfigurieren, die Datei beim Start zu nehmen, die gewünschten Aktionen auszuführen (in IntelliJ) und dann die neue Datei mit der alten Datei zu vergleichen. Sie sollten sehen, welches XML Sie benötigen, um den Ideenauftrag anzupassen. Sie müssen auch beachten, wo sich die XML-Datei befindet.

Eine andere Sache, die Sie berücksichtigen sollten, ist, dass Sie keine doppelten Knoten in den IntelliJ-Dateien wünschen, wenn Sie die Gradle-Idee mehrmals ausführen. Sie möchten also nach dem Knoten suchen, den Sie erstellen möchten. Wenn er nicht vorhanden ist, können Sie ihn erstellen und einfügen.

Fallstricke:

Wenn bei der Suchmethode == für den Zeichenfolgenvergleich verwendet wird, schlägt dies manchmal fehl. Beim Testen und ich finde, dass dies der Fall ist, verwende ich .contains.

Bei der Suche nach Knoten haben nicht alle Knoten das Attribut, das Sie als Kriterien verwenden. Überprüfen Sie daher unbedingt den Wert null.

Examples

Fügen Sie eine Grundkonfiguration hinzu

Annahmen für dieses Beispiel:

- Sie haben eine Klasse, `foo.bar.Baz` .
- Sie möchten eine Laufkonfiguration erstellen, die die Hauptmethode ausführt.
- Es ist in einem Modul namens `fooBar` .

In deiner Gradle-Datei:

```
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}"
        }
    }
}
```

IntelliJ IDEA Aufgabenanpassung online lesen: https://riptutorial.com/de/gradle/topic/2297/intellij-idea-aufgabenanpassung

Kapitel 12: Versionsnummer für automatisches Inkrementieren mit Gradle Script für Android-Anwendungen

Examples

So rufen Sie die automatische Inkrementierungsmethode beim Erstellen auf

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

Definition der automatischen Inkrementmethode

```
 /*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!")
    }
}
```

Versionsnummer aus einer Eigenschaftendatei lesen und einer Variablen zuweisen

def versionPropsFile = Datei ('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!")
}
```

Versionsnummer für automatisches Inkrementieren mit Gradle Script für Android-Anwendungen online lesen: https://riptutorial.com/de/gradle/topic/10696/versionsnummer-fur-automatisches-inkrementieren-mit-gradle-script-fur-android-anwendungen

Kapitel 13: Verwendung von Plugins von Drittanbietern

Examples

Hinzufügen eines Drittanbieter-Plugins zu build.gradle

Gradle (All Versions) *Diese Methode funktioniert für alle* **Gradle-** *Versionen*

Fügen Sie den Buildscript-Code am Anfang Ihrer build.gradle-Datei ein.

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

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

Gradle (Versionen *2.1 und höher* **)** *Diese Methode funktioniert nur für Projekte, die Gradle 2.1 oder höher verwenden.*

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

build.gradle mit mehreren Plugins von Drittanbietern

Gradle (Alle Versionen)

Wenn Sie mehrere Plugins von Drittanbietern hinzufügen, müssen Sie sie nicht in verschiedene Instanzen des Buildscript-Codes (All) oder des Plug-Ins (2.1+) unterteilen. Neue Plug-Ins können neben bereits vorhandenen Plug-Ins hinzugefügt werden.

```
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 (Versionen 2.1+)

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

Verwendung von Plugins von Drittanbietern online lesen:
https://riptutorial.com/de/gradle/topic/9183/verwendung-von-plugins-von-drittanbietern

Credits

S. No	Kapitel	Contributors
1	Erste Schritte mit gradle	Afterfield, bassim, Community, Emil Burzo, Eric Wendelin, Hamzawey, Hillkorn, Matthias Braun, Nikem, Pepper Lebeck-Jobe, Sergey Yakovlev, Stanislav, user2555595, vanogrid, Will
2	Abhängigkeiten	Afshin, Andrii Abramov, GameScripting, Hillkorn, leeor, Matthias Braun, mcarlin, mszymborski, Will
3	Aufgaben bestellen	Gabriele Mariotti
4	Aufgabenabhängigkeiten	Gabriele Mariotti, Sergey Yakovlev, Stanislav
5	Einschließlich nativer Quelle - experimentell	iHowell
6	Gradle Init-Skripte	ambes, Hillkorn
7	Gradle Performance	ambes, Sergey Yakovlev, Will
8	Gradle Plugins	Gabriele Mariotti, JBirdVegas
9	Gradle wird initialisiert	Eric Wendelin, Will
10	Gradle Wrapper	ajoberstar, Fanick, HankCa, I Stevenson
11	IntelliJ IDEA Aufgabenanpassung	IronHorse, Sam Sieber, Will
12	Versionsnummer für automatisches Inkrementieren mit Gradle Script für Android-Anwendungen	Jayakrishnan PM
13	Verwendung von Plugins von Drittanbietern	Afterfield