

Build Automation

Riferimenti

- Sommerville, Capitolo 29
- <http://grokcode.com/538/java-build-systems-a-sad-state-of-affairs/>
- <http://mrbook.org/tutorials/make/>
- <http://ant.apache.org/manual/tutorial-HelloWorldWithAnt.html>
- <http://maven.apache.org/guides/getting-started/maven-in-five-minutes.html>
- <http://maven.apache.org/guides/getting-started/>

Build Automation

- Build Automation è l'insieme di attività legate all'automatizzazione di alcuni task del ciclo di vita del software
 - Compilazione;
 - Linking;
 - Esecuzione di test;
 - Deployment;
 - Creazione di documentazione;
 - ...
- Per poter automatizzare queste attività è spesso necessario scrivere del codice in un linguaggio di scripting (oppure utilizzare programmi visuali)
 - Il codice di build automation è da considerare a pieno titolo nell'insieme del codice del programma!

Build Automation

- Tramite un efficace codice di Build Automation è possibile, ad esempio:
 - Generare e deployare automaticamente diverse versioni del programma
 - Versioni in lingue diverse
 - Versioni con insiemi di feature diverse
 - Versioni adatte a diversi sistemi operativi
 - Versioni di testing/debugging con alcune parti di codice sostituite da moduli fittizi
 - Reperire automaticamente le risorse necessarie a completare l'esecuzione del programma

Strumenti di Build Automation

- Make
- Apache Ant
- Apache Maven
- Visual Build
- http://en.wikipedia.org/wiki/List_of_build_automation_software

Shell Script

- La soluzione più semplice ai problemi di build automation passa per la scrittura di script di shell
 - La maggior parte degli strumenti necessari (compilatore, linker) sono eseguibili da linea di comando
 - Tutte le shell (es. Bash, DOS shell, etc.) forniscono comandi a sufficienza per la gestione del file system
 - I linguaggi di shell scripting sono interpretati: se vengono utilizzati per realizzare elaborazioni complesse, queste ultime sono difficili da testare
 - I linguaggi di shell offrono poco supporto alla programmazione strutturata e nessun supporto alla programmazione a oggetti
 - I linguaggi di shell non hanno tipi (al più variabili stringhe)

Comandi Shell DOS

- Call, per chiamare un altro batch file (.bat)
- Start, per avviare un eseguibile, eventualmente in un'altra istanza di shell
- Choice, per implementare una sorta di switch
- For, per implementare un ciclo for (in un insieme predefinito di valori)
 - Forfiles, per implementare un ciclo su di un insieme di files
- Goto, salto incondizionato
- If, che supporta come condizioni l'identità tra stringhe (==), l'esistenza di un file (EXIST) e la restituzione di un certo valore di ritorno di errore (ERRORLEVEL)
- Set, per le assegnazioni di variabile
- Tipici comandi di gestione file (copy, xcopy, dir, delete, rename, cd, md, rd, ...)
- Le shell di Unix (bash o altre shell) forniscono tipicamente più feature ma seguono la stessa filosofia
- Un qualsiasi programma utente può arricchire quest'insieme a patto però di essere eseguibile da linea di comando.

Make

- Sotto il nome 'make' è possibile raggruppare parecchie utility diffuse sui vari sistemi Linux o Windows
 - La più nota è GNU Make
- E' distribuita in forma di eseguibile a linea di comando
 - Le sue elaborazioni dipendono da un file di scripting nominato *makefile*
 - L'unico parametro, opzionale, è un TARGET che rappresenta la label del punto del makefile da cui comincerà l'esecuzione
- Il makefile è uno script in un linguaggio dichiarativo, non imperativo, che permette di specificare come debbano avvenire le operazioni di deployment e installazione di un software

Makefile

- Uno script makefile è organizzato in regole (rules)
- Ogni regola è individuata da:
 - un'etichetta (Target) seguita da:
 - una lista (opzionale) di componenti da cui la regola dipende
 - Un elenco di comandi (commands) di shell collegati a quella regola

```
targets : prerequisites ; command
```

- Uno script makefile è organizzato in regole (rules)

```
copia: originale.txt; copy originale.txt copia.txt
```

Esempio Makefile

#Variabili

CC = gcc

CFLAGS = -g

all: helloworld

helloworld: helloworld.o

\$(CC) \$(LDFLAGS) -o \$@ \$^

helloworld.o: helloworld.c

\$(CC) \$(CFLAGS) -c -o \$@ \$<

clean:

rm -f helloworld helloworld.o

- CC e CFLAGS sono variabili
- All (target predefinito) per essere eseguito prevede come prerequisito l'esistenza del file helloworld: se non esiste, allora viene eseguito preventivamente il target che omonimo
- helloworld ha bisogno che esista helloworld.o, altrimenti esegue preventivamente il target chiamato helloworld.o
- \$@ è una variabile che rappresenta il nome del target mentre \$< rappresenta i parametri in ingresso alla chiamata make

Limiti dei makefile

- Esistono alcune (poche) altre possibilità di implementare strutture di controllo nell'ambito di un makefile
- Make dipende dal sistema operativo: con sistemi operativi diversi non è possibile riutilizzare identicamente gli stessi script (perchè dipendono dal linguaggio di shell) e sono anche necessari diversi porting di make

ANT

- Ant è il primo e più diffuso strumento di build per programmi Java
- Ant è indipendente dalla piattaforma e dal sistema operativo
- Può essere utilizzato anche per elaborazioni più complesse di un semplice build
- Gli script Ant sono scritti in XML
- Ant fornisce ricche “librerie” di task predefiniti
- Ant fornisce la possibilità di realizzare task riusabili
- Ant può essere chiamato da linea di comando in modo da poter essere facilmente integrato in altri programmi
- Ant è parte del progetto open source Apache
- Ant non necessita di installazione (è sufficiente copiarne i file ed, eventualmente, settarne il percorso tra i path predefiniti)

ANT

- La sintassi di ant è praticamente la stessa di make:
 - ant [target]
- Ant cerca ed esegue il file *build.xml* che è analogo al makefile di make
- Esempio:

```
<?xml version="1.0"?>
<project name="Hello World Project" default="info">
  <target name="info">
    <echo>Hello World - Welcome to Apache Ant!</echo>
  </target>
</project>
```
- Col tag project si specifica il nome del progetto e il target predefinito (in questo caso info)

Target

Attributes	Description
name	The name of the target (Required)
depends	Comma separated list of all targets that this target depends on. (Optional)
description	A short description of the target. (optional)
if	Allows the execution of a target based on the trueness of a conditional attribute. (optional)
unless	Adds the target to the dependency list of the specified Extension Point. An Extension Point is similar to a target, but it does not have any tasks. (Optional)

Variabili

- Col tag *property* è possibile definire una variabile e settarne nome (attributo *name*) e valore (attributo *value*)
- Per utilizzare il valore è sufficiente scrivere, in un qualsiasi altro punto `{$nomevariabile}`
- Tipicamente è possibile elencare tutte le property e i loro valori in un file separato denominato *build.properties*

Comandi

- I comandi possono essere scritti sotto forma di tag i cui attributi sono gli attributi del comando

```
<target name="build" description="Compile source tree java files">  
  <mkdir dir="${build.dir}"/>  
  <javac destdir="${build.dir}" source="1.5" target="1.5">  
    <src path="${src.dir}"/>  
    <classpath refid="master-classpath"/>  
  </javac>  
</target>
```

- Numerosi filtri sono disponibili per facilitare la ricerca di file o insiemi di file sui quali applicare i comandi
 - Fileset
 - Patternset
 - Filterset
 - ...

Limiti di Ant

- Nella maggior parte dei casi, gli script ant sono generati automaticamente da altri programmi, ad esempio da IDE come Eclipse
- Gli script ant sono un pò limitati in quanto a strutture di controllo e ad altre possibilità di introdurre comportamenti dinamici
- Gli script ant possono indirizzare soltanto file su disco, non risorse remote
 - Da verificare

Maven

- Maven non è soltanto un tool per la build automation (come Ant) ma:
 - *“attempt to apply patterns to a project's build infrastructure in order to promote comprehension and productivity by providing a clear path in the use of best practices”*
 - In particolare fu pensato per unificare il modo di sviluppare e organizzare diversi progetti della famiglia Apache
- Non si occupa solo della costruzione, installazione e deployment dei progetti, ma anche della generazione della documentazione, di metriche, reports e casi di test
- Allo stesso modo di make e ant, può essere eseguito da linea di comando
- Allo stesso modo di ant, legge uno o più file xml di configurazione

Maven

- Uno schema di massima di partenza per il file di configurazione può essere generato col comando:
`mvn archetype:generate -DgroupId=com.mycompany.app -DartifactId=my-app -DarchetypeArtifactId=maven-archetype-quickstart -DinteractiveMode=false`
- Gli unici input necessari e variabili sono il nome dell'applicazione e il dominio di riferimento degli sviluppatori
 - La necessità di indicare sia il nome che il dominio è legata alla possibilità di poter fornire nomi universali (URI) ai progetti, in modo che possano essere univocamente reperiti sulla rete
 - In ant le risorse necessarie al deployment di un progetto potevano essere solo locali
 - Attenzione: nella maggior parte dei casi maven può funzionare solo e soltanto in presenza di connessione ad Internet
 - In particolare, avviene per la prima esecuzione di Maven susseguente la sua installazione

Esempio di base: pom.xml

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
    http://maven.apache.org/xsd/maven-
4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.mycompany.app</groupId>
  <artifactId>my-app</artifactId>
  <packaging>jar</packaging>
  <version>1.0-SNAPSHOT</version>
  <name>Maven Quick Start Archetype</name>
  <url>http://maven.apache.org</url>
  <dependencies>
    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>3.8.1</version>
      <scope>test</scope>
    </dependency>
  </dependencies>
</project>
```

- Pom sta per Project Object Model
- Il tag packaging sta ad indicare che il risultato eseguibile del building verrà messo in forma di file compresso jar
- version è la versione del progetto che verrà generata da questo build. Dovrebbe ricalcare il numero di release mantenuto dal cvs
- url è il sito del progetto (bisogna modificare il valore di default che rappresenta il sito di maven stesso)
- L'unica dipendenza dichiarata nell'archetipo è quella da Junit necessario per il testing

Albero dei file generati

```
my-app
|-- pom.xml
'-- src
    |-- main
    |   |-- java
    |       |-- com
    |           |-- mycompany
    |               |-- app
    |                   |-- App.java
    '-- test
        |-- java
        |   |-- com
        |       |-- mycompany
        |           |-- app
        |               |-- AppTest.java
```

- Lo script di esempio può essere eseguito con
 - mvn compile
- Eseguendo maven nella cartella che contiene pom.xml, viene generato quest'albero dei file
- I file Java sono creati sotto forma di template da completare

Altre modalità di esecuzione di mvn

- Per eseguire i test è sufficiente scrivere:
 - `mvn test`
 - Test è il valore di un tag `scope`, che è equivalente al concetto di target visto in make ed ant
- Per creare l'eseguibile (jar):
 - `mvn package`
- Altre modalità di esecuzione standard:
 - `mvn site` → genera il sito web di documentazione
 - `mvn clean` → cancella tutti i file generati
 - `mvn eclipse:eclipse` → genera file di progetto per eclipse
- Così come in ant, variabili possono essere dichiarate in un file separato *application.properties*

Dependencies

- E' possibile dichiarare dependencies da altri progetti non disponibili in locale

```
<dependencies>
```

```
...
```

```
<dependency>
```

```
<groupId>com.mycompany.app</groupId>
```

```
<artifactId>my-app</artifactId>
```

```
<version>1.0-SNAPSHOT</version>
```

```
<scope>compile</scope>
```

```
</dependency>
```

```
</dependencies>
```

- Come si può notare, NON viene nominata esattamente l'URL del progetto da cui dipende ciò che dobbiamo costruire, ma il groupId, l'artifactId e la version saranno sufficienti a maven per reperirlo nel caso esso sia stato correttamente pubblicato nell'ambito del dominio com.mycompany.app
- Junit faceva eccezione: in assenza della dichiarazione della URL, maven cerca in locale e sul sito ufficiale del progetto Maven

Dependencies

- E' possibile indicare in pom.xml e nel file aggiuntivo di configurazione settings.xml le modalità di pubblicazione del nostro progetto in un repository remoto, allo scopo di renderlo disponibile ad altri

```
<distributionManagement>
  <repository>
    <id>mycompany-repository</id>
    <name>MyCompany Repository</name>
    <url>scp://repository.mycompany.com/repository/maven2</url>
  </repository>
</distributionManagement>
</project>
<settings xmlns="http://maven.apache.org/SETTINGS/1.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/SETTINGS/1.0.0
    http://maven.apache.org/xsd/settings-1.0.0.xsd">
```

```
<servers>
  <server>
    <id>mycompany-repository</id>
    <username>jvanzyl</username>
    <!-- Default value is ~/.ssh/id_dsa-->

    <privateKey>/path/to/identity</privateKey> (default is ~/.ssh/id_dsa)

    <passphrase>my_key_passphrase</passphrase>
  </server>
</servers>
...
</settings>
```

Maven vs Ant

- I maggiori vantaggi di Maven rispetto ad Ant sono:
 - Possibilità di gestire numerosi aspetti del ciclo di vita dell'applicazione, oltre a build e deployment
 - Maggiori librerie e plug-in di utilità
 - Possibilità di estendere il comportamento tramite plug-in
 - Maggiore facilità d'uso (le funzionalità sono sempre utilizzabili tramite dichiarazione, mentre negli altri erano descritti più nella forma di comandi imperativi)
 - Possibilità di pubblicare progetti per il riuso
 - Possibilità di includere progetti disponibili in remoto e di collegarli dinamicamente a tempo di compilazione
 - Migliore integrazione con gli IDE di sviluppo