Възможностите на Ant за изграждане на големи проекти

Версия V0-0-4

Иван Иванов - rambiusparkisanius@gmail.com,
Петър Тахчиев - paranoiabla@gmail.com

25 октомври 2005 г.

Съдържание

1. Apache Ant: Уводни думи

Apache Ant е инструмент за изграждане (компилиране, пакетиране, тестване) на проекти. Основните му характеристики са:

- платформена независимост - тъй като е Java-базирано приложение;
- лесен за разширяване - много нови негови компоненти могат да се добавят, като се кодират на Java, както и на други скриптови езици като JavaScript, Python;
- с прост, но мощен XML синтаксис лесен за научаване;
- може да изгражда не само Java проекти, но и такива създавани на други езици.

1.1. Структура на build скриптовете

– Ant използва `build файлове`, които описват как да се компилира, пакетира и тества едно приложение. Build скриптът е XML файл, съдържащ стъпките за изграждане на един проект. Големи проекти могат да се състоят от по-малки проекти всеки от които може да има отделен build скрипт. Друг главен build скрипт може да координира build процесите на подпроектите.

– Всеки build скрипт съдържа няколко `target`-и (цели), които могат да зависят помежду си. В повечето случаи имаме target за компилиране на проекта, target за пакетиране и target за тестване, като target-ите за тестване и пакетиране е естествено да зависят от target-а за компилиране.

– Target-ите се състоят от `task`-ове, които извършват някаквa операция. Например, имаме task `<javac>`, която компилира Java файлове, както и task `<copy>`, която копира файлове.

Ето един елементарен build скрипт:

```
<project name="timestamp" basedir="." default="all">
    <description>Prints time and date</description>

    <tstamp/>

    <target name="print-date"
        description="Prints the current date">
        <echo>Current date is ${DSTAMP}</echo>
    </target>

    <target name="print-time"
        description="Prints the current time">
        <echo>Current time is ${TSTAMP}</echo>
    </target>

    <target name="all" depends="print-date,print-time"
        description="Prints the current date and time"/>
</project>
```

Извикваме target от този build скрипт по следния начин:

```
$ ant print-date
```

Резултатът е:

```
Buildfile: build.xml

print-date:
     [echo] Current date is 20051006

BUILD SUCCESSFUL
Total time: 0 seconds
```

Можем да извикаме ant и без да подадем target. Тогава се вика target-а по подразбиране, указан в `default` аттибута на проекта:

```
$ ant
Buildfile: build.xml

print-date:
     [echo] Current date is 20051006

print-time:
     [echo] Current time is 1919

all:

BUILD SUCCESSFUL
Total time: 0 seconds
```

Примерът показва:

- Използването на task-а `<echo>`, който извежда някакво съобщение. Тук част от съобщението са стойностите на `property`-та, наречени `DSTAMP`, `TSTAMP`. Property е двойка от име и стойност, като стойността може да се вземе чрез името с нотацията `${prop_name}`; веднъж установена, стойността на едно property не може да бъде променена;
- Използването на task-а `<tstamp>`, който установява стойностите на property-та с имена `DSTAMP`, `TSTAMP` съответно на текущата дата и текущото време;
- Не е задължително task-овете да са винаги в target-и. В нашия случай task-а `tstamp` е извън всички target-и. Тези task-ове се изълняват винаги, независимо кой target е стартиран;
- Зависимостите на един target от други се задават чрез `depends` атрибута на target-а, например target-а `all` зависи от `print-date` и `print-time` и те ще се изпълнят автоматично при извикване на all;
- Проектът може да има вложен таг `<description>`, който описва употребата на скрипта; `description` атрибут може да имат и target-ите, който описва действието им; това дава възможност за добавяне на документация на скрипта, която по-късно може да бъде изведена с командата `ant -projecthelp`.

Ant притежава и някои вградени property-тата, т.е. такива, които се генерират автоматично, без да е необходимо потребителят да ги дефинира. Това са:

- системните property-тата на JVM - същите property-ата, които достъпваме чрез метода System.getProperties();
- специфични за Ant property-та като:
  - `basedir` - абсолютния път до главната директория на проекта както е указано в basedir атрибута на <project> тага;
  - `ant.file` - абсолютния път на build скрипта, който се изпълнява в момента;
  - `ant.version` - версията на Ant;
  - `ant.project.name` - името на проекта, който се изпълнява в момента, както е указано в name атрибута на <project> тага;
  - `ant.java.version` - версията на Java виртуалната машина, с която се изпълнява build скрипта.

Освен това имаме достъп до всички променливи на обкръжението, например ако искаме да вземем стойността на променлива на обкръжението наречена CATALINA_HOME използваме следната конструкцията:

```
<project>
    <property environment="env"/>
    <echo>$${env.CATALINA_HOME} is ${env.CATALINA_HOME}</echo>
</project>
```

1.2. Изграждане и тестване на HelloWorld приложение

Следващата стъпка е да направим HelloWorld приложение, което да компилираме, тестваме и пакетираме. Приложението се състои от:

– Java код:

```
package net.sourceforge.emaria.openfest;

public class Message {

    public String getMessage(String name) {
        return "Hello World, " + name;
    }
}
```

```
package net.sourceforge.emaria.openfest;

public class Main {

    public static void main(String[] args) {
        Message message = new Message();
        System.out.println(message.getMessage("Sharo"));
    }
}
```

– Кода на unit тестовете:

```
package net.sourceforge.emaria.openfest;

import junit.framework.TestCase;
import junit.textui.TestRunner;
import net.sourceforge.emaria.openfest.Message;

public class MessageTest extends TestCase {

    public void testHelloWorld() {
        Message helloWorld = new Message();
        assertEquals(helloWorld.getMessage("Sharo"),
            "Hello World, Sharo");
        assertNotNull(helloWorld.getMessage(null));
    }

    public void testHelloWorldFails() {
        Message helloWorld = new Message();
        assertEquals(helloWorld.getMessage("Sharo"),
            "Hello World, OpenFest");
    }
}
```

А ето и самия код, от build.xml-а, който:

– компилира приложението:

```
<target name="compile" depends="-init"
    description="Compiles the project's sources">
    <mkdir dir="${project.classes}"/>
    <javac srcdir="${project.src}"
        destdir="${project.classes}"/>
</target>
```

– компилира тестовете:

```
    <delete dir="${project.tests.classes}"/>
    <delete dir="${project.tests.data}"/>
    <delete dir="${project.tests.reports}"/>
</target>

<target name="test.compile" depends="-test.init,compile"
    description="Compiles the project's unit tests">
    <mkdir dir="${project.tests.classes}"/>
```

– стартира тестовете:

```
            destdir="${project.tests.classes}">
            <classpath refid="compile.tests.cp"/>
        </javac>
    </target>

    <target name="test" depends="test.compile"
        description="Runs the project's unit tests">
        <mkdir dir="${project.tests.reports}"/>
        <mkdir dir="${project.tests.data}"/>
        <junit printsummary="yes" haltonfailure="no"
                haltonerror="no" errorproperty="test.error"
                failureproperty="test.error">
            <classpath refid="runtime.tests.cp"/>
            <formatter type="plain" usefile="false"/>
            <formatter type="xml"/>
            <batchtest fork="yes" todir="${project.tests.data}">
                <fileset dir="${project.tests.classes}">
                    <include name="**/*Test.class"/>
                </fileset>
            </batchtest>
        </junit>
        <junitreport todir="${project.tests.data}">
            <fileset dir="${project.tests.data}">
                <include name="TEST-*.xml"/>
            </fileset>
            <report format="frames"
```

Тези примери демонстрират:

- Task-а `<junit>`, който изпълнява всички unit тестове, избрани чрез вложеният елемент `<batchtest>`;
- Елемента `formatter`, който трансформира резултата от тестовете във формата указан от атрибута му `type`;
- Атрибутите `haltonerror`, `haltonfailure`, които указват дали изпълнението на build-а да бъде прекратено, ако тестовете не са успешни;
- Атрибутите `failureproperty` и `errorproperty`, указващи имената на property-тата, които се установяват ако тестовете не са успешни;
- Task-а `<junitreport>`, който преобразува резултатите от тестовете от XML формат в HTML формат чрез XSLT трансформация;
- Task-а `<fail>`, който прекратява build-а като неуспешен ако някакво условие е изпълнено;
- В нашия пример указваме build-а да не спира при неуспех на тестовете, но задаваме при евентуален неуспех да се инициализира property-то `test.error`, което ни послужи ръчно да спрем build-а като неуспешен, но едва след като сме генерирали HTML резултатите с <junitreport>.

1.3. Първа употреба на `<presetdef>`

Task-ът `<presetdef>` създава нова дефиниция на вече съществуващ task, като предефинира с фиксирани стойности някои от атрибути или елементи й. Ето един възможен сценарий за употреба на <presetdef>:

1. Task-а <javac> по подразбиране не генерира debug информация, която по време на разработка е необходима за проследяване на грешки; както видяхме в HelloWorld приложението <javac> се използва поне два пъти в build скрипта: за компилиране на приложението и за компилирането на unit тестовете;
2. Можем всеки път, когато използваме <javac>, да цитираме едни и същи стойности на атрибутите `debug`, `debuglevel` и `deprecation`, но това води до буквално повтаряне на код;
3. Решението е да използваме <presetdef>:

```
<presetdef name="javac">
    <javac debug="on" debuglevel="vars,lines,source"
        deprecation="on"/>
</presetdef>
```

   Така всеки път когато използваме <javac>, използваме горната предефинирана версия;
4. Това води и до следния проблем: debug информацията увеличава размера на компилираните класове, което не е желано за официални build-ове, а всяко ползване на <javac> генерира такава информация;

5. Решението е вместо фиксирани стойности за атрибутите да изпозваме <property>-та. Тъй като стойностите им не могат да се променят, всичко което трябва да осигурим, е при правене на официални build-ове, стойностите на debug property-тата да се установяват на off преди всичко друго.
6. В нашия случай това постигаме с добавянето на target-а

```
<target name="javadoc" depends="-init"
    description="Generates project's javadoc">
</target>

<target name="doc" depends="javadoc"
```

и указваме target-а за създаване на официална дистрибуция `dist` да зависи от `-dist.init`:

```
</target>

<target name="dist.clean" depends="-init"
```

2. Описание на task-овете <subant>, <import>, <macrodef>

Тези task-ове, както и <presetdef> се въведоха в Ant 1.6, за да улеснят създаването на нови task-ове, без да е необходимо да се пише кода на Java, за опростяването на build процеса при големи проекти и за повторната употреба на съществуващи build скриптове.

2.1. Дефиниране на нови task-ове чрез <macrodef>

Task-ът **<macrodef>** дефинира нови task-ове, като входните данни на тези нови task-ове задаваме с елементите **<attribute>** и **<element>**.
Ето един полезен <macrodef>, който отваря зададен HTML файл в браузъра, по подразбиране използвайки Linux-ката команда **htmlview**:

```
<macrodef name="browse">
    <attribute name="file"/>
    <sequential>
        <exec executable="htmlview" spawn="yes">
            <arg file="@{file}"/>
        </exec>
    </sequential>
</macrodef>
```

Той се използва като всички останали task-ове:

```
<target name="test-browser" depends="browserDef"
    description="Tests the macrodef">
    <browse file="test-macrodef.html"/>
</target>
```

Можем да напишем и версия на този <macrodef>, която да работи под Windows:

```
<macrodef name="browse">
    <attribute name="file"/>
    <sequential>
        <exec executable="rundll32.exe">
            <arg line="url.dll, FileProtocolHandler"/>
            <arg file="@{file}"/>
        </exec>
    </sequential>
</macrodef>
```

Тези две версии могат да се комбинират в един и същи build скрипт, така че <browse> task-a да работи и под Windows, и под Linux:

```
<target name="checkOS"
    description="Checks the operating system">
    <condition property="isWindows">
        <os family="windows"/>
    </condition>
    <condition property="isLinux">
        <os name="Linux"/>
    </condition>
</target>

<target name="winBrowserDef" depends="checkOS" if="isWindows">
    <macrodef name="browse">
        <attribute name="file"/>
        <sequential>
            <exec executable="rundll32.exe">
                <arg line="url.dll, FileProtocolHandler"/>
                <arg file="@{file}"/>
            </exec>
        </sequential>
    </macrodef>
</target>

<target name="linBrowserDef" depends="checkOS" if="isLinux">
    <macrodef name="browse">
        <attribute name="file"/>
        <sequential>
            <exec executable="htmlview" spawn="yes">
                <arg file="@{file}"/>
            </exec>
        </sequential>
    </macrodef>
</target>

<target name="browserDef" depends="winBrowserDef,linBrowserDef"/>
```

2.2. Повторна употреба на build скриптове чрез <import>

В горния пример видяхме, че след извикването на target-а `browserDef` macrodef-ът <browse> се дефинира за съответната операционна система. Естествено е да поискаме да използваме този <macrodef> и в други build скриптове.

Един от начините за повторна употреба на Ant build скриптове (или части от тях) е чрез XML entity-та, като този начин не е свързан с Ant, а с XML формата на build скриптовете.

Другият начин е да използваме task-а `<import>`. Пример за синтаксиса на този task е:

```
<import file="imported.xml" optional="true"/>,
```

където атрибутът `file` указва друг build файл, който искаме да включим в настоящия build скрипт, а атрибутът `optional` указва дали build-ът да спре, ако импортваният build файл не съществува.

Build скриптът, съдържащ macrodef-а <browse>, се импортва по следния начин:

```
<project name="test-import" default="test-browser" basedir=".">

    <import file="../macrodef/browser.macrodef" optional="false"/>

    <target name="test-browser" depends="browserDef">
        <browse file="test-import.html"/>
    </target>
</project>
```

Target-ът test-browser зависи от target-а browserDef (който е дефиниран в импортвания файл), тъй като macrodef-ът <browse> се създава след като се изпълни target-а browserDef. Така <import> позволява повторното ползване и на macrodef-ове и на target-и.

В основата си той работи както работят XML entity-та, т.е. <import> добавя в импортващият файл всичко от импортвания без <project> тага. Разликите обаче са, че <import> въвежда предефиниране на target-и и някои специални property-та.

2.3. Стартиране на няколко build скрипта чрез <subant>

Да разгледаме следната ситуация: нека имаме даден продукт, който се състои от няколко проекта, и изграждането на продукта изисква изграждането на тези проекти. В повечето случаи всеки от проектите има свой собствен build скрипт и като build скриптът на продукта вика build скриптовете на проектите.
В древността преди Ant 1.6 това се постига с task-а <ant>. Ето псевдокод, който илюстрира тази идея:

```
<target name="compile-product">
    <ant target="compile" antfile="project1/build.xml"/>
    <ant target="compile" antfile="project2/build.xml"/>
    ...
    <ant target="compile" antfile="projectN/build.xml"/>
</target>
```

Един удобен заместител на task-а <ant> е task-а <subant>, който изглежда по следния начин:

```
<target name="compile-product">
    <subant target="compile">
        <dirset dir="${basedir}">
            <include name="project*"/>
        </dirset>
    </subant>
</target>
```

3. Демонстрация на проект използващ <subant>, <import> и <macrodef>

Проектът, чрез който ще илюстрираме горните идеи, се нарича `antutils`. Той реализира няколко разширения на Apache Ant, отделени в компоненти, които са:

- FileInfo - Ant task, който извлича информация за даден файл, като размер, дали е скрит и т.н;
- JazzyTask - Ant task, който проверява правописа на файлове, използвайки Java библиотеката Jazzy;
- Input - този компонент позволява маскирането на пароли, въвеждани през текстова консола;
- MailAttachLogger - Ant listener, който наследява оригиналния MailLogger и дава възможност за прикачване на файлове към съобщението с резултатите от build процеса.

Всеки от тези компоненти трябва да се компилира, тества и пакетира. Тези три операции могат да се отделят в един общ build скрипт - `common.xml`, който чрез <import> да се използва в build скрипта на всеки компонент. При необходимост build скрипта на даден компонент може да предефинира някое property или task от common.xml. Имаме и един главен build скрипт `build.xml`, който използва <subant>, за да извика build скриптовете на компонентите.

Примерът с antutils показва случая, при който компонентите са независими един от друг. Същата техника обаче с някои изменения може да се използва и при изграждане на проекти, които зависят един от друг.

4. Достъп до настоящата презентация и примерния код

Можете да свалите презентацията под формата на архив наречен `OFP2005.tar.bz2` от адреса

http://sourceforge.net/project/showfiles.php?group_id=103509

Последната й версия може да се изтегли от CVS хранилището на SF.net чрез командите:

```
cvs -d:pserver:anonymous@cvs.sourceforge.net:/cvsroot/emaria login
cvs -d:pserver:anonymous@cvs.sourceforge.net:/cvsroot/emaria co -P \
OFP2005
```

Кодът на проекта `antutils` можете да свалите пак от CVS хранилището на SF.net чрез командите:

```
cvs -d:pserver:anonymous@cvs.sourceforge.net:/cvsroot/emaria login
cvs -d:pserver:anonymous@cvs.sourceforge.net:/cvsroot/emaria co -P
antutils
```

При поискване на парола за CVS сървъра, дайте празна такава, т.е. просто натиснете Enter.