Ant 简明教程

◆ 作者：吕崇恩

一、什么是 Ant?

Apache Ant，是一个将软件编译、测试、部署等步骤联系在一起加以自动化的一个工具，大多于 Java 环境中的软件开发。由 Apache 软件基金会所提供。Ant 是由 Java 编写的，Ant 的用户可以编写包含 Ant 任务和数据类型的自定义 Ant 类库，也可以直接使用大量商业或开源的 Ant 类库。

教程配套视频课程：http://www.atstudy.com/course/15

二、如何安装 Ant?

1. 下载 Ant

在官网找到下载的 URL：http：//Ant.apache.org/bindownload.cgi，这里笔者选择 zip 包进行下载。在 Ubuntu 系统中创建了 Ant 文件夹，并使用 wget 命令进行下载。因为 Ant 的更新较快，请读者自行选择当前最新的 Ant 版本进行下载，这里以 1.9.4 为例。

```
mkdir Ant cd Ant
wget http: //mirrors.cnnic.cn/apache//Ant/binaries/apache-Ant-1.9.4-bin.zip
unzip apache-Ant-1.9.4-bin.zip
```

2. 设置环境变量

在路径/etc/profile.d 下创建文件 setAnt.sh 文件，内容如下，保存后并重启系统。其中 ANT_HOME 的路径为读者

解压 Ant 的路径。

```
export Ant_HOME=/home/lvchongen/Ant/apache-Ant-1.9.4
```

```
export PATH=$ANT_HOME/bin:  $PATH
```

3. 查看 Ant 版本

使用命令 Ant -version 来查看 Ant 的版本，若能成功显示，则表示 Ant 成功安装。笔者使用该命令后显示 如下结果:

```
lvchongen@iZ25hpb9g9uZ:~$ ant -version
Apache Ant(TM) version 1.9.4 compiled on April 29 2014
```

至此，我们就成功的安装了 Ant。之后我们将介绍如何使用 Ant。

三、如何使用 Ant?

1. 感性认识 Ant

在详细的介绍使用方法之前，大家可以通过一个简单的小例子感性的认识 Ant。

创建一个名字为 HelloAnt.java 的文件，内容如下:

```
public class HelloAnt{
public static void main(String[] args) { System.out.println("Hello Ant!");
}
}
```

在同级目录下创建一个名字为 build.xml 的文件，内容如下:

```
<? xml version="1.0"? >
<project name="Hello Ant" default="compile">

<property name="src" value="src"/>
<property name="dest" value="classes"/>

<target name="init">
<mkdir dir="${src}"/>
<mkdir dir="${dest}"/>
<copy todir="${src}/" file="HelloAnt.java"/>
</target>

<target name="compile" depends="init">
```

```
<javac srcdir="${src}" destdir="${dest}" includeAntruntime="on" />
<java classname="HelloAnt" classpath="${dest}" />
</target>

</project>
```

进入命令行，在 build.xml 和 HelloAnt.java 同级目录下执行命令 Ant ，将会有如下结果，我们可以看到创建了 两个文件夹 src 和 classes，在 src 文件夹下有 HelloAnt.java 文件，在 classes 文件夹下有 HelloAnt.class 文件，并且正确的输出了 Hello Ant 的结果。

```
→  temp git:(master) ✗ ls
HelloAnt.java build.xml
→  temp git:(master) ✗ ant
Buildfile: /Users/lvchongen/Desktop/temp/build.xml

init:
    [mkdir] Created dir: /Users/lvchongen/Desktop/temp/src
    [mkdir] Created dir: /Users/lvchongen/Desktop/temp/classes
     [copy] Copying 1 file to /Users/lvchongen/Desktop/temp/src

compile:
    [javac] Compiling 1 source file to /Users/lvchongen/Desktop/temp/classes
     [java] Hello Ant!

BUILD SUCCESSFUL
Total time: 0 seconds
→  temp git:(master) ✗ ls src
HelloAnt.java
→  temp git:(master) ✗ ls classes
HelloAnt.class
```

那么这里我们简单分析下这个 build.xml 文件，来对 Ant 有感性的认识。

```
<? xml version="1.0"? >
<!-- 注明这是一个 name 为 Hello Ant 的 project，需要默认执行"compile"的命令-->
<project name="Hello Ant" default="compile">
<!-- 定义 name 为 src 的属性，值为 src-->
<property name="src" value="src"/>
<!-- 定义 name 为 dest 的属性，值为 classes-->
<property name="dest" value="classes"/>
<!-- 定义名字为 init 的 target，  我们也可以理解为这是一个功能函数  -->
<target name="init">
<!-- 创建文件夹，并使用变量 src，使用方法是${src}，值为 src -->
<mkdir dir="${src}"/>
<!-- 创建文件夹，并使用变量 dest，使用方法是${dest}，值为 classes -->
<mkdir dir="${dest}"/>
<!-- 复制文件 HelloAnt.java 到目标路径 src 文件夹下  -->
```

```
<copy todir="${src}/" file="HelloAnt.java"/>
</target>
<!-- 定义名字为 compile 的 target， depends="init"指该 target 的执行依赖于 init，即执行前先执行
init -->
<target name="compile" depends="init">
<!-- 使用命令 javac 编译 java 文件，将生成的 class 文件存储在 classes 文件夹下，并将 Ant 相关库 加入构建中 -->
<javac srcdir="${src}" destdir="${dest}" includeAntruntime="on" />
<!-- 使用命令 java 执行编译后的 class 文件，路径在 classes 文件夹下 -->
<java classname="HelloAnt" classpath="${dest}" />
</target>
</project>
```

至此，我们已经对 Ant 有了感性的认识，只要正确的配置 build.xml 文件，Ant 就可以实现项目的自动构建和部署等 功能，从而在开发过程或自动化过程中节省大量的时间。

2. 理性认识 Ant

build.xml 介绍

我们可以创建一个简单的 build.xml 文件有如下内容:

```
<? xml version="1.0"? >
<project name="Hello World Project" default="info">
<target name="info">
<echo>Hello World - Welcome to Apache Ant!</echo>
</target>
</project>
```

在命令行执行 Ant 后由如下结果:

```
[echo] Hello World - Welcome to Apache Ant!
```

这就构成了一次简单的构建任务，那么我们首先来分析这个 Ant 的核心心任务描述文件 build.xml

project 元素

project 元素是 Ant 构件文件的根元素，Ant 构件文件至少应该包含一个 project 元素，否则会发生错误。在每个 project 元素下，可包含多个 target 元素。接下来向读者展示一下 project 元素的各属性。

属性	描述
name	指定 project 元素的名字
default	project 默认执行的 target 的名字
basedir	指定 project 的位置，默认路径构建文件所在路径

我们通过两个例子来说明这些属性的作用和用法

1. basedir 使用默值

```
<? xml version="1.0"? >
<project name="Hello World Project" default="info">
<target name="info">
<echo>${basedir}</echo>
</target>
</project>
```

我们并未对 basedir 进行赋值，故 basedir 的输出值应该为 build.xml 所在路径，现在使用 Ant 命令来验证下我 们的推论:

```
→ Test_basedir git:(master) ✗ ls
build.xml
→ Test_basedir git:(master) ✗ pwd
/Users/lvchongen/Desktop/temp/Test_basedir
→ Test_basedir git:(master) ✗ ant
Buildfile: /Users/lvchongen/Desktop/temp/Test_basedir/build.xml

info:
     [echo] /Users/lvchongen/Desktop/temp/Test_basedir

BUILD SUCCESSFUL
Total time: 0 seconds
```

可以看到输出的 basedir 路径就是 build.xml 的所在路径。

2. basedir 进行用户赋值

首先在 build.xml 同级目录下创建文件夹 basedir_folder，之后修改 build.xml 内容如下:

```
<? xml version="1.0"? >
<project name="Hello World Project" default="info" basedir="basedir_folder">

<target name="info">
<echo>${basedir}</echo>
</target>

</project>
```

这里使用了 basedir="basedir_folder" 对 basedir 进行赋值，使用 Ant 命令后 women 验证输出结果:

可以看到，basedir 被赋值后，改变了默认值，并被正确输出。

target 元素

target 是 Ant 的基本执行单元，它可以包含一个或多个具体的任务。多个 target 可以存在相互依赖关系，它有如下属性:

属性	描述
name	指定 target 的名字，是 target 的唯一标示符
depends	描述 target 之间的依赖关系，多个依赖时用逗号
description	对 target 的功能描述
if	当判定条件为真时，target 可以被执行
unless	当判定条件不成立时，target 可以被执行

我们通过几个例子来说明这些属性的作用和用法:

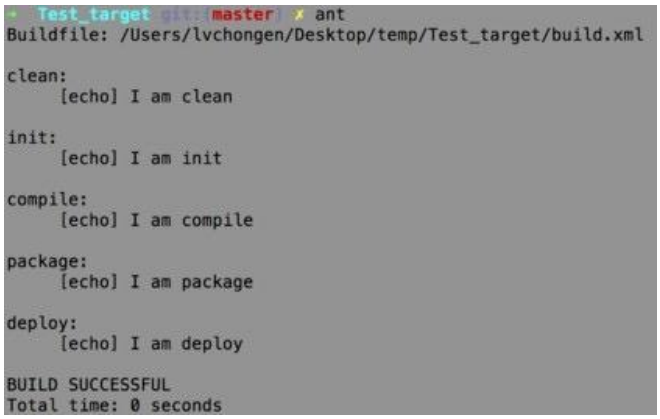
1. target 之间的依赖关系

首先创建 build.xml 文件，编辑内容如下

```
<? xml version="1.0"? >
<project name="Hello World Project" default="deploy">

<target name="deploy" depends="package">
<echo>I am deploy</echo>
</target>
<target name="package" depends="clean，compile">
<echo>I am package</echo>
</target>

<target name="clean">
<echo>I am clean</echo>
</target>

<target name="compile" depends="init">
<echo>I am compile</echo>
</target>

<target name="init">
<echo>I am init</echo>
</target>

</project>
```

这里我们创建了几个 target，分别有对应的依赖关系。target 之间的依赖逻辑是 A 依赖于 B，则先执行 B 再执行

A，以此类推，那么我们现在执行下命令 Ant 来看下结果:



可以看到，target 的执行结果是按照 depends 的顺序。

2. if 和 unless 的使用逻辑 在使用 if 和 unless 之前，我们首先简单介绍下 condition 和 os。

condition 是条件元素，可以通过调用其 property 来获得 condition 的判定结果---布尔值，判断的表达式写在与之间，笔者将会在后续章节详细介绍。os 是 Ant 的内置数据类型，有 family，name，arch，version 属性，其中 family 的值有:

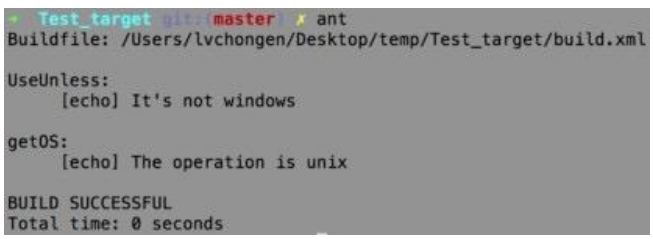
1. windows (for all versions of Microsoft Windows)
2. dos (for all Microsoft DOS based operating systems including Microsoft Windows and OS/2)
3. mac (for all Apple Macintosh systems)
4. unix (for all Unix and Unix-like operating systems)
5. netware (for Novell NetWare)
6. os/2 (for OS/2)
7. tandem (for HP's NonStop Kernel - formerly Tandem)
8. win9x for Microsoft Windows 95 and 98， ME and CE
9. winnt for Microsoft Windows NT-based systems， including Windows 2000， XP and successors
10. z/os for z/OS and OS/390
11. os/400 for OS/400

12. openvms for OpenVMS

现在笔者创建 build.xml 文件并且内容如下:

```
<? xml version="1.0"? >
<project name="Get operation system" default="getOS" >

<target name="getOS" if="isUnix" depends="UseUnless">
<echo>The operation is unix</echo>
</target>

<target name="UseUnless" unless="isWindows">
<echo>It's not windows</echo>
</target>

<condition property="isUnix">
<os family="unix" />
</condition>

<condition property="isWindows">
<os family="windows" />
</condition>

</project>
```

笔者在 ubuntu 系统下执行 Ant 命令，得到如下结果:



至此，可以验证: * 对 target 使用 if 判断时，只有当判定条件为 true 时，target 才被执行。对 target 使用unless 判断时，只有当判定条件为 false 时，target 才被执行。

property 元素

property 是 Ant 的属性元素，一般来说有 7 种方法来设置 property。

1. By supplying both the name and one of value or location attribute，提供 name 和 value 或者 name 和 location

创建 build.xml文件有如下内容:

```
<? xml version="1.0"? >

<project name="Test property" default="TestProperty">

<property name="Test_value" value="HelloWorld"/>
<property name="Test_location" value="/bin" />

<target name="TestProperty">
<echo>${Test_value}</echo>
<echo>${Test_location}</echo>
</target>

</project>
```

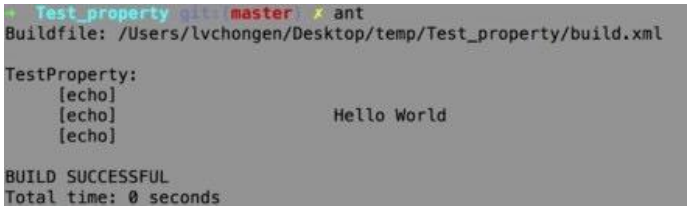
执行命令 Ant 后，有如下结果:

2. By supplying the name and nested text ，提供名字和嵌套的文本 创建 build.xml文件有如下内容，这里我们可以看到，特性 Result 使用了 value1 和 value2 组合而成。

```
<? xml version="1.0"? >
<project name="Test property" default="TestProperty">

<property name="value1" value="Hello"/>
```

```
<property name="value2" value="World" />
<property name="Result" value="${value1} ${value2}"/>

<target name="TestProperty">
<echo>
${Result}
</echo>
</target>

</project>
```

执行命令 Ant 后有如下结果:



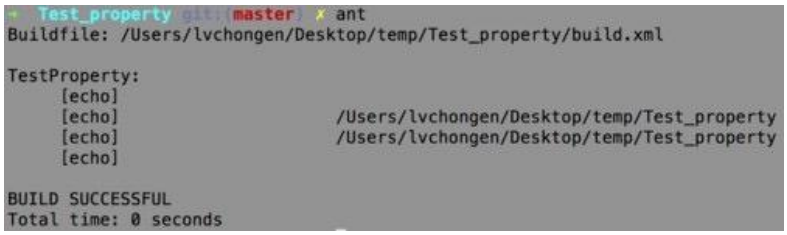
3. By supplying both the name and refid attribute，提供名字和引用属性的 id

创建 build.xml 文件有如下内容，这里我们可以使用了元素来作为引用，该元素是用来设置路径的，将在下文详细介绍。

```
<? xml version="1.0"? >
<project name="Test property" default="TestProperty">

<path id="project.path">
<pathelement location="./"/>
</path>

<property name="Result" refid="project.path"/>

<target name="TestProperty">
<echo>
${Result}
${basedir}
```

```
</echo>
</target>

</project>
```

现在我们使用命令 Ant 来查看 Result 的值，根据定义我们期望值是当前项目的路径，应该和 basedir 相同。



我们看到，结果是两个变量的值相同，均为当前项目的路径，这与我们的预期是相同的。以上便是如何使用 refid 进行赋值。

4. By setting the file attribute with the filename of the property file to load，通过设置文件的名字来导入该文件中的属性。

创建 build.xml 文件有如下内容，其中我们引入了 property file="build.properties，这意味着我们可以将一些属 性写在文件 build.properties 中，从而在 build.xml 中直接读取。

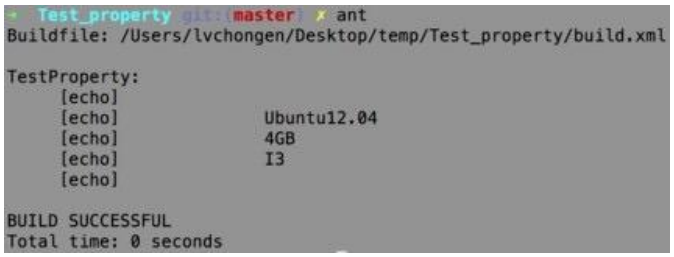
```
<? xml version="1.0"? >
<project name="Test property" default="TestProperty">

<property file="build.properties"/>

<target name="TestProperty">
<echo>
${OS}
${Memory}
${CPU}
</echo>
</target>
```

```
</project>
```

这里我们使用了三个未在 build.xml 中定义的属性：OS，Memort，CPU，那么我们需要创建文件 build.properties 来定义这些属性：

```
OS=Ubuntu12.04
Memory=4GB CPU=I3
```

现在我们执行 Ant 命令，根据定义我们应该能正确得到属性 OS，Memory，CPU 的值。



可以看到，三个属性的值是正确的。

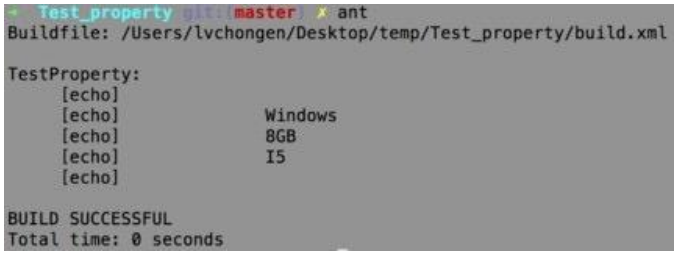
5. By setting the url attribute with the url from which to load the properties，通过 URL 导入属性文件，并使用其中的属性。 创建 build.xml 文件有如下内容，其中我们通过 url=http：//123.56.101.72：8080/build.properties 远程导入 build.properties 文件，根据 Ant 的定义，我们应该能够正确是使用远程文件中的各个属性。

```
<？xml version="1.0"？>
<project name="Test property" default="TestProperty">

<property url="http：//123.56.101.72：8080/build.properties"/>

<target name="TestProperty">
<echo>
${OS}
${Memory}
${CPU}
</echo>
</target>

</project>
```

而其中 build.properties 文件内容如下:

```
OS=Windows Memory=8GB CPU=I5
```

现在执行 Ant 命令，我们验证是否能够正确的获得这些属性的值:



可见，能够正确的读取配置文件。使用这种导入属性配置文件的方法，方便我们统一对 Ant 工程进行管理。

6. By setting the resource attribute with the resource name of the property file to load，通过设置资源文件的名字来导入该文件中的属性。 从定义上看，使用 resource 标签和使用 file 标签的功能是相似的，但实际上有些属性配置文件存储在一些不能直接访问的路径下，只有通过 resource 才能进行访问，我们需要通过配置 classpath 来使得 Ant 在该路径文件下寻找属性配置文件。

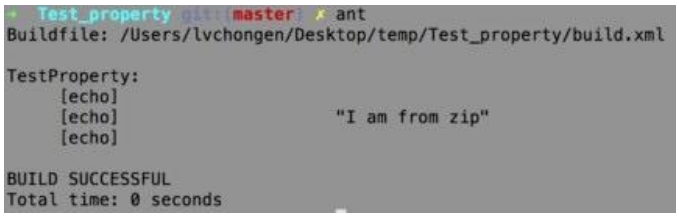
我们通过几个例子来演示这强大的特性。

加载 zip 文件内的属性配置文件，build.xml 内容如下，其中存储了属性 property_zip 的文件 zip.properties 被压缩在 resource.zip 中。而文件 zip.properties 中的内容为 property_zip="I am from zip"

```
<? xml version="1.0"? >
<project name="Test property" default="TestProperty">

<property resource="zip.properties" classpath="./resource.zip" />

<target name="TestProperty">
<echo>
${property_zip}
</echo>
</target>
```

```
</project>
```

现在执行 Ant 命令来查看下结果是否为预期的：



可以看到，正确的输出的属性的值：I am from zip。所以 resource 可以正确的读取 zip 内的属性配置文件。

加载 jar 文件内的属性配置文件，build.xml 内容如下，其中存储了属性 property_jar 的文件 jar.properties 被 打包在 properties.jar 中。而文件 jar.properties 中的内容为 property_jar="I am from jar"

```
<? xml version="1.0"? >
<project name="Test property" default="TestProperty">

<property resource="com/Ant/property/jar.properties"       classpath="properties.ja r" />

<target name="TestProperty">
<echo>
${property_jar}
</echo>
</target>

</project>
```

现在执行 Ant 命令来查看下结果是否为预期的：

可以看到，Ant 从 jar 包中正确读到了配置文件，并输出了属性的值。通过 resource，Ant 还可以访问 gzip，bzip2 等压缩文件，使用方法类似，这里就不一一举例

了。

By setting the environment attribute with a prefix to use，通过前缀使用环境变量的值。

我们通过引入 property environment="env"，就可以直接使用系统默认的环境变量了。创建 build.xml 文件内容 如下:

```
<? xml version="1.0"? >
<project name="Test property" default="TestProperty">

<property environment="env"/>

<target name="TestProperty">
<echo>
${env.ANT_HOME}
${env.JAVA_HOME}
</echo>
</target>

</project>
```

现在执行 Ant 命令查看实际的执行结果:

我们可以看到 Ant 已经正确的输出了两个环境变量 ANT_HOME 和 JAVA_HOME。至此，我们已经将 7 种设置 属性的方法全部介绍完成了，在实际使用中要灵活根据场景进行选择。

在使用 property 时，我们也可以使用 Ant 的预定义属性:

属性	描述

Ant.file	build.xml 文件的路径
Ant.version	Ant 的版本
basedir	默认为 project 的路径，若有自定义赋值，则显示自定义的值
Ant.java.version	Ant 使用的 JDK 的版本
Ant.project.name	当前 project 的名字
Ant.project.default-target	当前 project 默认的 target
Ant.project.invoked-targets	当前 project 使用命令行执行的 target 列表
Ant.core.lib	Ant.jar 文件的路径
Ant.home	安装 Ant 的路径
Ant.library.dir	Ant 库文件的路径，通常是${Ant.home}/lib

现在创建 build.xml 文件内容如下:

```
<? xml version="1.0"? >
<project name="I am project name" default="First_target">

<target name="First_target">
<echo>
Ant.file: ${Ant.file} Ant.version: ${Ant.version} basedir: ${basedir}
Ant.java.version: ${Ant.java.version} Ant.project.name: ${Ant.project.name}
Ant.project.default-target: ${Ant.project.default-target}
Ant.project.invoked-targets: ${Ant.project.invoked-targets}
Ant.core.lib: ${Ant.core.lib} Ant.home: ${Ant.home} Ant.library.dir:
${Ant.library.dir}
</echo>
```

```
</target>

<target name="Second_target">
<echo>I am second target</echo>
</target>
</project>
```

```
Test_property git:(master) ✗ ant First_target Second_target
Buildfile: /Users/lvchongen/Desktop/temp/Test_property/build.xml

First_target:
     [echo]
     [echo]                    ant.file:       /Users/lvchongen/Desktop/temp/Test_property/build.xml
     [echo]                    ant.version:    Apache Ant(TM) version 1.9.4 compiled on April 29 2014
     [echo]                    basedir:        /Users/lvchongen/Desktop/temp/Test_property
     [echo]                    ant.java.version:       1.7
     [echo]                    ant.project.name:       I am project name
     [echo]                    ant.project.default-target:     First_target
     [echo]                    ant.project.invoked-targets:    First_target,Second_target
     [echo]                    ant.core.lib:   /usr/local/apache-ant-1.9.4/lib/ant.jar
     [echo]                    ant.home:       /usr/local/apache-ant-1.9.4
     [echo]                    ant.library.dir:        /usr/local/apache-ant-1.9.4/lib
     [echo]

Second_target:
     [echo] I am second target

BUILD SUCCESSFUL
Total time: 0 seconds
```

执行命令 Ant First_target Second_target，返回结果如下:

```
Test_property git:(master) ✗ ant First_target Second_target
Buildfile: /Users/lvchongen/Desktop/temp/Test_property/build.xml

First_target:
     [echo]
     [echo]                    ant.file:       /Users/lvchongen/Desktop/temp/Test_property/build.xml
     [echo]                    ant.version:    Apache Ant(TM) version 1.9.4 compiled on April 29 2014
     [echo]                    basedir:        /Users/lvchongen/Desktop/temp/Test_property
     [echo]                    ant.java.version:       1.7
     [echo]                    ant.project.name:       I am project name
     [echo]                    ant.project.default-target:     First_target
     [echo]                    ant.project.invoked-targets:    First_target,Second_target
     [echo]                    ant.core.lib:   /usr/local/apache-ant-1.9.4/lib/ant.jar
     [echo]                    ant.home:       /usr/local/apache-ant-1.9.4
     [echo]                    ant.library.dir:        /usr/local/apache-ant-1.9.4/lib
     [echo]

Second_target:
     [echo] I am second target

BUILD SUCCESSFUL
Total time: 0 seconds
```

按照定义其中 Ant.project.invoked-targets 应输出两个 target 的名字，而其他的属性会根据当前 project 和系统环 境来进行显示。

日常使用过程中，我们可以直接将属性设置在 build.xml 文件中。但当工程较大的时候，推荐使用引入配置文 件的方式设置属性。在不同的工程环境下使用相同的 build.xml，通过不同的配置文件进行区分。例如 Dev，Product 环境下使用同一个 build.xml，通过 build.properties.dev 和 build.properties.product 进行区分达到复用的效果。

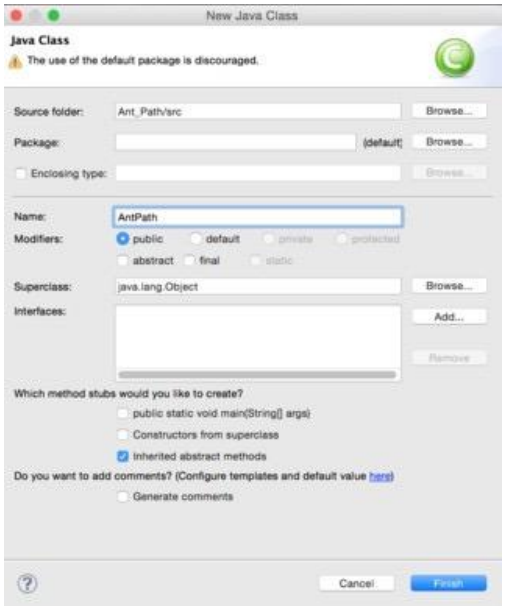
Datatypes

Datatypes 是 Ant 中的重要概念---数据类型，准确的说它不是一个元素，更像是数据结构，包含用户自定义的数据。Datatypes 是 build.xml 的重要组成结构。Path 和文件集合是 Ant 中非非常重要的 Datatypes。

Path 在 Ant 中经常被用来代替 classpath，路径元素通常使用 ： 和 ， 进行分割。下面的例子将演示如何使用 Path。

1. 笔者首先在 Eclipse 中创建 Ant_Path 的 Java 项目

2. 在 scr 文件夹下创一个名字为 AntPath 的类文件

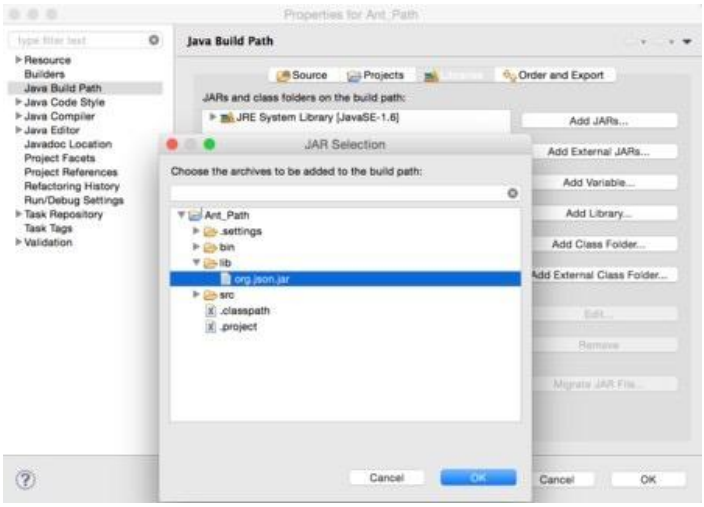


3. 下载 java-json.jar 文件，官方下载地址为:

http: //www.java2s.com/Code/JarDownload/java-json/java- json.jar.zip

4. 在 Project 根目录下创建空文件夹 lib

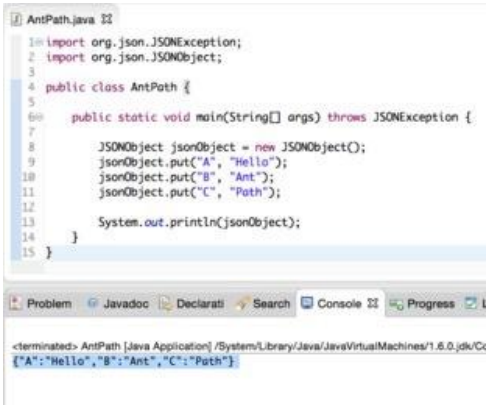
5. 将下载的 org.json.jar 复制到第三步创建的 lib 文件夹下，并配置项目的依赖



6. 在 AntPath 文件中编写如下代码:

```
import org.json.JSONException;
import org.json.JSONObject;
public class AntPath {
public static void main(String[] args) throws JSONException { JSONObject jsonObject = new JSONObject();
jsonObject.put("A", "Hello"); jsonObject.put("B", "Ant"); jsonObject.put("C", "Path");

System.out.println(jsonObject);
}
}
```

7. 执行后应打印出如下结果: {"A": "Hello", "B": "Ant", "C": "Path"}



8. 在项目根目录下创建 build.xml 文件

build.xml 文件编写如下代码:

```
<? xml version = "1.0" ? >
<project name="DataType_Ant" default="compile">

<target name="clean">
<delete dir="build"/>
<echo>Deleted</echo>
</target>

<target name="compile" depends="clean">
<mkdir dir="build/classes"/>
<javac srcdir="src" destdir="build/classes" includeAntruntime="on">
</javac>
</target>

</project>
```

根据之前学过的知识，读者应该可以了解这个 build.xml 文件的意图，它分为两个部分 clean 和 compile。其中 clean 的目的是删除文件夹 build，compile 的目的是编译 src 文件夹下的 java 文件，并将生成的 class 文件存储在新创建的文件夹 build 中。而 compile 依赖于 clean。现在我们在项目根目录下执行命令 Ant 来查看是否能够成功编译。

```
Path/build/classes
    [javac] /Users/lvchongen/Documents/workspace/Ant_Path/src/AntPath.java:1: 错
误: 程序包org.json不存在
    [javac] import org.json.JSONException;
    [javac]                ^
    [javac] /Users/lvchongen/Documents/workspace/Ant_Path/src/AntPath.java:2: 错
误: 程序包org.json不存在
    [javac] import org.json.JSONObject;
    [javac]                ^
    [javac] /Users/lvchongen/Documents/workspace/Ant_Path/src/AntPath.java:6: 错
误: 找不到符号
    [javac]     public static void main(String[] args) throws JSONException {
    [javac]                                                    ^
    [javac]   符号:   类 JSONException
    [javac]   位置: 类 AntPath
    [javac] /Users/lvchongen/Documents/workspace/Ant_Path/src/AntPath.java:8: 错
误: 找不到符号
    [javac]             JSONObject jsonObject = new JSONObject();
    [javac]             ^
    [javac]   符号:   类 JSONObject
    [javac]   位置: 类 AntPath
    [javac] /Users/lvchongen/Documents/workspace/Ant_Path/src/AntPath.java:8: 错
误: 找不到符号
    [javac]             JSONObject jsonObject = new JSONObject();
```

可以看到，编译失败了并返回了大量关于找不到 jsonobject 的错误，那么如何解决呢？请耐心看下去。

9. 使用 Path 来解决找不到依赖的编译失败问题。修改 build.xml 文件如下:

```
<? xml version = "1.0" ? >
<project name="DataType_Ant" default="compile">
<target name="clean">
<delete dir="build"/>
</target>

<target name="compile" depends="clean">
<mkdir dir="build/classes"/>
<javac srcdir="src" destdir="build/classes" includeAntruntime="on">

<!-- 在 javac 命令中通过 refid 使用定义的 path -->
<classpath refid="org.json"/>
</javac>
</target>
<!-- 添加 Path 元素，指向 org.json.jar -->
<path id="org.json">
<pathelement path="lib/org.json.jar"/>
</path>
</project>
```

代码中有了注解，可还是要重点强调下，使用定义的 Path 需要调用 refid=ID。

10. 执行命令 Ant，验证编译结果。

```
Ant_Path  ant
Buildfile: /Users/lvchongen/Documents/workspace/Ant_Path/build.xml

clean:
   [delete] Deleting directory /Users/lvchongen/Documents/workspace/Ant_Path/bu
ld
     [echo] Deleted

compile:
    [mkdir] Created dir: /Users/lvchongen/Documents/workspace/Ant_Path/build/cl
sses
    [javac] Compiling 1 source file to /Users/lvchongen/Documents/workspace/Ant_
Path/build/classes

BUILD SUCCESSFUL
Total time: 1 second
```

查看生成的 class 文件，可见已经生成了 AntPath.class。

```
classes  pwd
/Users/lvchongen/Documents/workspace/Ant_Path/build/classes
classes  ls
AntPath.class
```

PatternSet 是一种可以根据特定条件进行文件和文件夹进行筛选的模式的集合。

PatternSet 有以下四种属性:

属性	描述
includes	用逗号分隔，需要包含的文件模式列表
excludes	用逗号分隔，不需要包含的文件模式列表
includesfile	文件名。该文件的每一行都将作为一个 include 模式
excludesfile	文件名。该文件的每一行都将作为一个 exclude 模式

再介绍下字符匹配的规则:

？ 匹配一个字符，比如 ？Hello.java 可以匹配 AHello.java ，但不能匹配 AAHello.java 或者 AHelloA.java 等。

* 匹配任意字符，比如 *Hello.java 可以匹配 AHello.java ，AAHello.java 或者 Hello.java ，但不能匹配 AHello.cs 等。

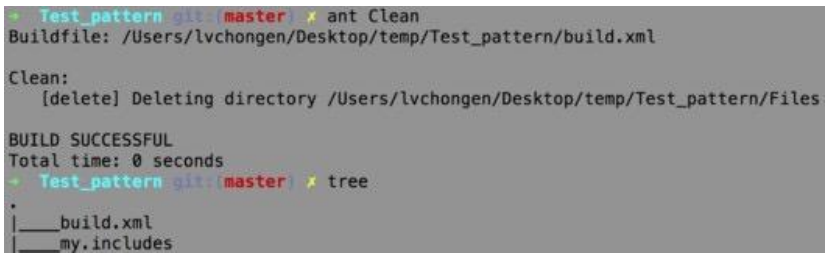
接下来，通过几个实际使用的例子让读者了解如何实际使用。首先创建 build.xml 文件内容如下:

```
<? xml version="1.0"? >
```

```
<project name="Test Pattern" default="">

<target name="Clean">
<delete dir="Includes_Folder"/>
<delete dir="Excludes_Folder"/>
<delete dir="Includesfile_Folder"/>
<delete dir="Excludesfile_Folder"/>
<delete dir="Files"/>
</target>

<target name="Init" depends="Clean">
<touch file="Files/Hello.java" mkdirs="True"/>
<touch file="Files/AHello.java"/>
<touch file="Files/HelloB.java"/>
<touch file="Files/AHelloB.java"/>
<touch file="Files/AAHello.java"/>
<touch file="Files/Hello.cs"/>
</target>

<target name="Test_includes" depends="Init">
<mkdir dir="Includes_Folder"/>
<copy todir="Includes_Folder">
<fileset dir="Files" includes="？Hello.java"/>
</copy>
</target>

<target name="Test_exclues" depends="Init">
<mkdir dir="Excludes_Folder"/>
<copy todir="Excludes_Folder">
<fileset dir="Files" excludes="*.*"/>
</copy>
<delete dir="Excludes_Folder" excludes="*Hello.java"/>
```

```
</target>

<target name="Test_includesfile" depends="Init">
<mkdir dir="Includesfile_Folder" />
<copy todir="Includesfile_Folder">
<fileset dir="Files" includesfile="my.includes"/>
</copy>
</target>

<target name="Test_excluesfile" depends="Init">
<mkdir dir="Excludes_Folder"/>
<copy todir="Excludes_Folder">
<fileset dir="Files" excludesfile="my.includes"/>
</copy>
</target>
</project>
```

clean 的作用是删除创建的文件夹，Includes_Folder ， Excludes_Folder ， Files ， Includesfile_Folder 和 Excludesfile_Folder 。我们执行命令 Ant Clean 后查看结果，我们的期望结果是不存在以上任何一个文件夹。



可见 Target Clean 能够正确的删除文件夹。

Init 的作用是创建文件夹 Files ， 以及待测试的文件， Hello.java ， AHello.java ， HelloB.java ， AHelloB.java ， AAHello.java 和 Hello.cs 。

我们执行命令 Ant Init 后查看结果，我们的期望结果是能够创建文件夹 Files，并且在该文件夹下成功创建待测试的文件。

```
→ Test_pattern git:(master) ✗ ls
build.xml   my.includes
→ Test_pattern git:(master) ✗ ant Init
Buildfile: /Users/lvchongen/Desktop/temp/Test_pattern/build.xml

Clean:

Init:
    [touch] Creating /Users/lvchongen/Desktop/temp/Test_pattern/Files/Hello.java
    [touch] Creating /Users/lvchongen/Desktop/temp/Test_pattern/Files/AHello.java
    [touch] Creating /Users/lvchongen/Desktop/temp/Test_pattern/Files/HelloB.java
    [touch] Creating /Users/lvchongen/Desktop/temp/Test_pattern/Files/AHelloB.java
    [touch] Creating /Users/lvchongen/Desktop/temp/Test_pattern/Files/AAHello.java
    [touch] Creating /Users/lvchongen/Desktop/temp/Test_pattern/Files/Hello.cs

BUILD SUCCESSFUL
Total time: 0 seconds
→ Test_pattern git:(master) ✗ tree
.
|____build.xml
|____Files
| |____AAHello.java
| |____AHello.java
| |____AHelloB.java
| |____Hello.cs
| |____Hello.java
| |____HelloB.java
|____my.includes
```

可以看到，执行后的文件结构与我们预期一致。

Test_includes 是用于测试 includes 属性的作用，创建文件夹 Includes_Folder 并将 Files 下的文件按照 includes=？Hello.java 的模式进行匹配和复制。

我们执行命令 Ant Test_includes 后查看结果，按照之前对 includes 和？的属性介绍，期望的结果是将以任意字符起始并后接 Hello.java 的文件复制到文件夹下。那么我们看下实际结果:

```
→ Test_pattern git:(master) ✗ ls
build.xml   my.includes
→ Test_pattern git:(master) ✗ ant Test_includes
Buildfile: /Users/lvchongen/Desktop/temp/Test_pattern/build.xml

Clean:

Init:
    [touch] Creating /Users/lvchongen/Desktop/temp/Test_pattern/Files/Hello.java
    [touch] Creating /Users/lvchongen/Desktop/temp/Test_pattern/Files/AHello.java
    [touch] Creating /Users/lvchongen/Desktop/temp/Test_pattern/Files/HelloB.java
    [touch] Creating /Users/lvchongen/Desktop/temp/Test_pattern/Files/AHelloB.java
    [touch] Creating /Users/lvchongen/Desktop/temp/Test_pattern/Files/AAHello.java
    [touch] Creating /Users/lvchongen/Desktop/temp/Test_pattern/Files/Hello.cs

Test_includes:
    [mkdir] Created dir: /Users/lvchongen/Desktop/temp/Test_pattern/Includes_Folder
     [copy] Copying 1 file to /Users/lvchongen/Desktop/temp/Test_pattern/Includes_Folder

BUILD SUCCESSFUL
Total time: 0 seconds
→ Test_pattern git:(master) ✗ tree
.
|____build.xml
|____Files
| |____AAHello.java
| |____AHello.java
| |____AHelloB.java
| |____Hello.cs
| |____Hello.java
| |____HelloB.java
|____Includes_Folder
| |____AHello.java
|____my.includes
```

可以看到，执行后的文件结构与预期一致。

Test_excludes 是用于测试 excludes 属性的作用，创建文件夹 Excludes_Folder 并将 Files 下的文件按照 includes="*.*" 的模式进行匹配和复制。

我们执行命令 Ant Test_excludes 后查看结果，按照之前对 exclues 的属性介绍，期望的结果是将 不将任何文件复制到文件夹 Files 下。下面我们看下实际的执行结果:

```
→ Test_pattern git:(master) ✗ ls
build.xml   my.includes
→ Test_pattern git:(master) ✗ ant Test_excludes
Buildfile: /Users/lvchongen/Desktop/temp/Test_pattern/build.xml

Clean:

Init:
    [touch] Creating /Users/lvchongen/Desktop/temp/Test_pattern/Files/Hello.java
    [touch] Creating /Users/lvchongen/Desktop/temp/Test_pattern/Files/AHello.java
    [touch] Creating /Users/lvchongen/Desktop/temp/Test_pattern/Files/HelloB.java
    [touch] Creating /Users/lvchongen/Desktop/temp/Test_pattern/Files/AHelloB.java
    [touch] Creating /Users/lvchongen/Desktop/temp/Test_pattern/Files/AAHello.java
    [touch] Creating /Users/lvchongen/Desktop/temp/Test_pattern/Files/Hello.cs

Test_excludes:
    [mkdir] Created dir: /Users/lvchongen/Desktop/temp/Test_pattern/Excludes_Folder

BUILD SUCCESSFUL
Total time: 0 seconds
→ Test_pattern git:(master) ✗ tree
.
|____build.xml
|____Excludes_Folder
|____Files
| |____AAHello.java
| |____AHello.java
| |____AHelloB.java
| |____Hello.cs
| |____Hello.java
| |____HelloB.java
|____my.includes
```

Test_includesfile 是用于测试 includesfile 属性的作用，创建文件夹 Includesfile_Folder ，并且编写 my.includes 文件，通过导入 my.includes 文件的模式匹配对 Files 下的文件进行匹配并复制。

在执行命令查看结果前，我们先确认下 includesfile 的含义，它的作用是导入我们自定义的配置文件，以下使我们自定义的配置文件:

```
→ Test_pattern git:(master) ✗ cat my.includes
?Hello.java
*Hello.java
```

所以按照之前对 includes 的介绍，我们的期望结果是将以 Hello.java 结尾的文件复制到 Includesfile_Folder 文件夹之下。执行命令 Ant Test_includesfile ，得到如下结果:

```
→  Test_pattern git:(master) ✗ ant Test_includesfile
Buildfile: /Users/lvchongen/Desktop/temp/Test_pattern/build.xml

Clean:

Init:
    [touch] Creating /Users/lvchongen/Desktop/temp/Test_pattern/Files/Hello.java
    [touch] Creating /Users/lvchongen/Desktop/temp/Test_pattern/Files/AHello.java
    [touch] Creating /Users/lvchongen/Desktop/temp/Test_pattern/Files/HelloB.java
    [touch] Creating /Users/lvchongen/Desktop/temp/Test_pattern/Files/AHelloB.java
    [touch] Creating /Users/lvchongen/Desktop/temp/Test_pattern/Files/AAHello.java
    [touch] Creating /Users/lvchongen/Desktop/temp/Test_pattern/Files/Hello.cs

Test_includesfile:
    [mkdir] Created dir: /Users/lvchongen/Desktop/temp/Test_pattern/Includesfile_Folder
     [copy] Copying 3 files to /Users/lvchongen/Desktop/temp/Test_pattern/Includesfile_Folder

BUILD SUCCESSFUL
Total time: 0 seconds
→  Test_pattern git:(master) ✗ tree
.
|____build.xml
|____Files
| |____AAHello.java
| |____AHello.java
| |____AHelloB.java
| |____Hello.cs
| |____Hello.java
| |____HelloB.java
|____Includesfile_Folder
| |____AAHello.java
| |____AHello.java
| |____Hello.java
|____my.includes
```

可以看到，Files 文件夹下有期望的文件。

Test_excludesfile 是用于测试 excludesfile 属性的作用，创建文件夹 Excludes_Folder ，并且编写 my.includes 文件，通过导入 my.includes 文件的模式匹配对 Files 下的文件进行匹 配并复制。

这个例子我们仍旧使用之前自定义的 my.includes 文件，但按照之前对 excludes 属性的介绍，我们的期望结果是不复制以 Hello.java 为结尾的文件，现在执行命令 Ant Test_excludesfile 来查看结果:

```
Test_pattern git:(master) ✗ ant Test_excludesfile
Buildfile: /Users/lvchongen/Desktop/temp/Test_pattern/build.xml

Clean:

Init:
    [touch] Creating /Users/lvchongen/Desktop/temp/Test_pattern/Files/Hello.java
    [touch] Creating /Users/lvchongen/Desktop/temp/Test_pattern/Files/AHello.java
    [touch] Creating /Users/lvchongen/Desktop/temp/Test_pattern/Files/HelloB.java
    [touch] Creating /Users/lvchongen/Desktop/temp/Test_pattern/Files/AHelloB.java
    [touch] Creating /Users/lvchongen/Desktop/temp/Test_pattern/Files/AAHello.java
    [touch] Creating /Users/lvchongen/Desktop/temp/Test_pattern/Files/Hello.cs

Test_excludesfile:
    [mkdir] Created dir: /Users/lvchongen/Desktop/temp/Test_pattern/Excludes_Folder
     [copy] Copying 3 files to /Users/lvchongen/Desktop/temp/Test_pattern/Excludes_Folder

BUILD SUCCESSFUL
Total time: 0 seconds
Test_pattern git:(master) ✗ tree
.
|____build.xml
|____Excludes_Folder
| |____AHelloB.java
| |____Hello.cs
| |____HelloB.java
|____Files
| |____AAHello.java
| |____AHello.java
| |____AHelloB.java
| |____Hello.cs
| |____Hello.java
| |____HelloB.java
|____my.includes
```

在这里我们看到 Excludes_Folder 文件夹下只有 Hello.cs ， AHelloB.java 和 HelloB.java 三个文件，全部都不以 Hello.java 结尾，所以这符合我们的预期。

FileSet 一般是指某一个目录下的指定文件，也包含该目录下递归子目录的所有文件。FileSet 可以通过模式集合和选择器对文件进行筛选。FileSet 有两个子元素是用来帮助筛选文件的，patternset 和 selector。其中我们已经对 patternset 做了详细介绍，接下来我将重点介绍写 selector。

常用的 selector 有如下几个:

属性	描述
contains	选择的文件中包含待匹配的字符串
date	选择的文件在特定时间之前或之后有过修改
depend	比较两个不同目录下名称相同的文件，然后选择文件修改时间迟的文件
depth	指定选择文件的文件目录的深度
different	从两个目录中选择被认为不同的文件
filename	选择符合指定文件匹配模式的文件

present	选择在指定文件夹下不存在的或者多个目录下同时存在的文件
containsregexp	选择符合正则表达式的文件
size	选择大于或小于 size 的文件
type	指定选择的类型是文件还是目录
modified	选择的文件被修改过并匹配一些特定条件
signedselector	选择指定被签名的文件
scriptselector	用于创建自定义的 selector
readable	选择有可读权限的文件
writable	选择有可写权限的文件

下面我将对这些 selector 进行举例，让读者加深对它们的理解:

contains

根据定义我们知道，contains 是用来筛选文件内容是否包含指定的内容的，所以笔者创建如下几个文件 并包含相应的内容:

- 文件 Ant.txt:

  I am Ant.

  I wAnt to say "Hello Ant"

- 文件 Maven.txt:

  I am Maven.

  I wAnt to say "Hello Maven"

- 文件 Git.txt:

  I am Git.

  I wAnt to say "Hello Git"

- 文件 Jenkins.txt:

  I am Jenkins.

  I wAnt to say "Hello Jenkins"

接下来，笔者创建 build.xml 文件，内容如下:

```
<? xml version="1.0"? >
<project name="Test selector for contians" default="Test_contains">
<target name="Test_contains" >
<mkdir dir="DestFolder"/>
<copy todir="DestFolder">
<fileset dir="SourceFolder">
<contains text="Hello Ant"/>
</fileset>
</copy>
</target>
</project>
```

这里使用了 contains text="Hello Ant" ，根据文件内容及属性作用，得到期望结果是 DestFolder 文件夹下有文件 Ant.txt。现在执行命令 Ant 来查看实际结果:

```
+ Test_selector git:(master) ✗ ls
SourceFolder build.xml
+ Test_selector git:(master) ✗ ant
Buildfile: /Users/lvchongen/Desktop/temp/Test_selector/build.xml

Test_contains:
     [copy] Copying 1 file to /Users/lvchongen/Desktop/temp/Test_selector/DestFolder

BUILD SUCCESSFUL
Total time: 0 seconds
+ Test_selector git:(master) ✗ tree
.
|____build.xml
|____DestFolder
| |____Ant.txt
|____SourceFolder
| |____Ant.txt
| |____Git.txt
| |____Jenkins.txt
| |____Maven.txt
```

可见，完全符合我们的预期。

date

根据定义我们知道 date 是用来筛选在指定时间之前或之后修改的文件。首先我们查看下已经创建的文件的修改时间，之后以 MM/DD/YYYY HH: MM AM_PM 的格式为 date 进行赋值。

```
+ SourceFolder git:(master) ✗ ll
total 32
-rw-r--r--  1 lvchongen  staff    37B  3 24 17:16 Ant.txt
-rw-r--r--  1 lvchongen  staff    37B  3 24 17:15 Git.txt
-rw-r--r--  1 lvchongen  staff    45B  3 24 17:15 Jenkins.txt
-rw-r--r--  1 lvchongen  staff    41B  3 24 17:14 Maven.txt
```

接下来创建 build.xml 文件如下，其中 before 的作用是筛选 03/25/2015 17: 15 PM 之前修改的文件并复制，after 的作用是筛选 03/25/2015 17: 15 PM 之后修改的文件并复制

```
<? xml version="1.0"? >
<project name="Test selector for date" default="">

<target name="after" >
<mkdir dir="DestFolder_after"/>
<copy todir="DestFolder_after">
<fileset dir="SourceFolder">
<date datetime="03/25/2015 17: 15 PM" when="after"/>
</fileset>
</copy>
</target>

<target name="before" >
<mkdir dir="DestFolder_before"/>
<copy todir="DestFolder_before">
<fileset dir="SourceFolder">
<date datetime="03/25/2015 17: 15 PM" when="before"/>
</fileset>
</copy>
</target>
</project>
```

现在使用命令 Ant before 和 Ant after 来查看实际运行结果:

```
→ Test_selector git:(master) ✗ ant after
Buildfile: /Users/lvchongen/Desktop/temp/Test_selector/build.xml

after:
    [mkdir] Created dir: /Users/lvchongen/Desktop/temp/Test_selector/DestFolder_after

BUILD SUCCESSFUL
Total time: 0 seconds
→ Test_selector git:(master) ✗ tree
.
|____build.xml
|____DestFolder_after
|____SourceFolder
| |____Ant.txt
| |____Git.txt
| |____Jenkins.txt
| |____Maven.txt
```

可以看到因为不存在指定日期之后有修改的文件，所以没有任何文件被复制。

```
→ Test_selector git:(master) ✗ ant before
Buildfile: /Users/lvchongen/Desktop/temp/Test_selector/build.xml

before:
    [mkdir] Created dir: /Users/lvchongen/Desktop/temp/Test_selector/DestFolder_before
     [copy] Copying 4 files to /Users/lvchongen/Desktop/temp/Test_selector/DestFolder_before

BUILD SUCCESSFUL
Total time: 0 seconds
→ Test_selector git:(master) ✗ tree
.
|____build.xml
|____DestFolder_before
| |____Ant.txt
| |____Git.txt
| |____Jenkins.txt
| |____Maven.txt
|____SourceFolder
| |____Ant.txt
| |____Git.txt
| |____Jenkins.txt
| |____Maven.txt
```

因为所有的文件都在指定日期之前有修改，所以都被复制到了指定的文件夹下。

depend

根据定义我们知道 depend 是用于比较两个不同目录下名称相同的文件，然后选择文件修改时间最迟的那个文件。这是一个比较抽象的概念，笔者将会通过例子进行展示:

1. 创建两个文件夹 Source_1 和 Source_2，其中 Source_1 为指定的文件夹，Source_2 为目标文件夹。

2. 在 Source_2 中创建文件 TestFile.txt 并且内容为 I am older 。复制该文件到 Source_1 中并修改内容为 I am newer。这样文件 TestFile.txt 的修改时间在 Source_1 文件夹下是晚于 Source_2 的。

3. 在 Source_1 中创建文件 TestFile2.txt 并且内容为 I am older 。复制该文件到 Source_2 中并修改内容为 I am newer。这样文件 TestFile2.txt 的修改时间在 Source_2 文件夹下是晚于 Source_1 的。

4. 创建 build.xml 文件，内容如下，实现的内容是指定文件夹 Source_1，配置目标文件夹为 Source_2，并比较同名文件的修改时间进行复制。

```
<? xml version="1.0"? >
<project name="Test selector for depend" default="Test_depend">

<target name="Test_depend">
<mkdir dir="DestFolder"/>
<copy todir="DestFolder">
<fileset dir="Source_1">
<depend targetdir="Source_2"/>
</fileset>
</copy>
</target>

</project>
```

这里我们的期望结果是 DestFolder 文件夹下应该只有 TestFile2.txt 文件，并且来自 Source_1 中且内容为 I am newer。现在执行命令 Ant 来查看下实际结果:

可以看到，实际结果与期望一致。感兴趣的读者，可以再创建一个 target 并将 build.xml 中的 Source_1 与 Source_2 互换位置，来实现筛选最新文件的功能。

depth

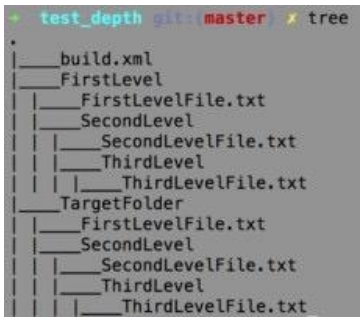
根据定义我们知道 depth 是通过文件所在文件夹的深度来进行筛选文件的。这里笔

者通过一个实例来演示如何使用 depth。

1. 通过 target 来创建三级文件夹及文件，target 内容如下:

```
<target name="init" depends="clean">
<mkdir dir="FirstLevel"/>
<mkdir dir="FirstLevel/SecondLevel"/>
<mkdir dir="FirstLevel/SecondLevel/ThirdLevel"/>
<touch file="FirstLevel/FirstLevelFile.txt"/>
<touch file="FirstLevel/SecondLevel/SecondLevelFile.txt"/>
<touch file="FirstLevel/SecondLevel/ThirdLevel/ThirdLevelFile.txt"/>
</target>
```

若执行任务并创建成功后，应有如下的文件结构，这里文件夹的深度是 3。

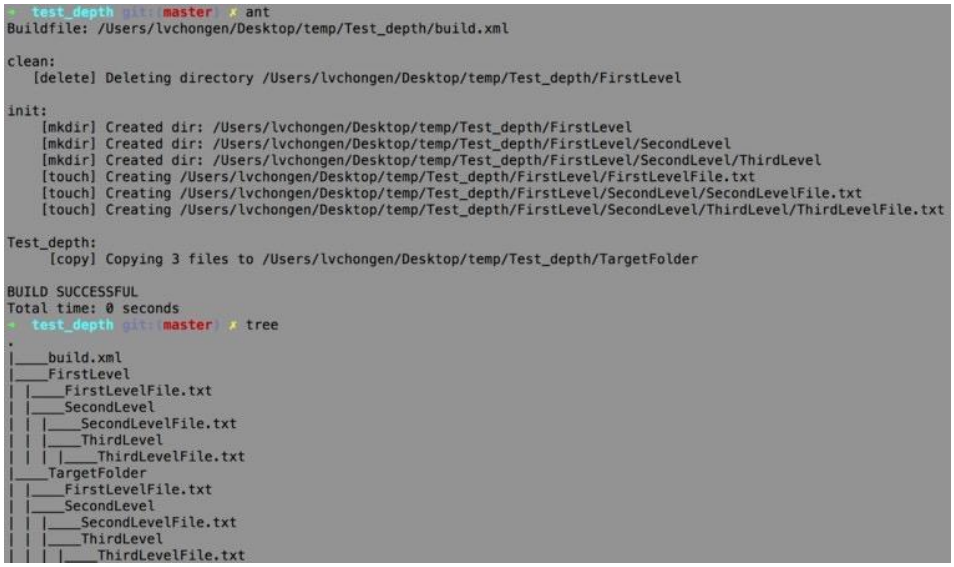


2. 创建任务来实现检索创建的文件夹，并根据文件的深度来对文件进行筛选和复制。完整 build.xml 内 容如下:

```
<? xml version="1.0"? >
<project name="Test selector for depth" default="Test_depth">

<target name="init" depends="clean">
<mkdir dir="FirstLevel"/>
<mkdir dir="FirstLevel/SecondLevel"/>
<mkdir dir="FirstLevel/SecondLevel/ThirdLevel"/>
<touch file="FirstLevel/FirstLevelFile.txt"/>
<touch file="FirstLevel/SecondLevel/SecondLevelFile.txt"/>
<touch file="FirstLevel/SecondLevel/ThirdLevel/ThirdLevelFile.txt"/>
</target>
<target name="clean">
```

```
<delete dir="FirstLevel"/>
</target>
<target name="Test_depth" depends="init">
<mkdir dir="TargetFolder"/>
<copy todir="TargetFolder">
<fileset dir="FirstLevel" includes="**/*">
<depth max="3"/>
</fileset>
</copy>
</target>

</project>
```

3. 执行 Ant 命令查看实际结果



可以看到，Ant 根据 3 级文件深度进行筛选后，复制了对应的文件夹和文件，使得 TargetFolder 与 FirstLevel 有相同的文件结构。其中有参数 max 和 min，作用是对文件深度进行筛选，分享相当于小于等于和大约等于，感兴趣的读者可以自己尝试下 min。

different

根据定义我们知道 different 将会根据一定的规则将不同的文件选择出来。而 different 有如下属性:

属性	描述
targetdir	用于指定一个目录，这个目录下的文件与 fileset 中的 dir 属性指定的目录中的文件进行比较
ignoreFileTimes	是否忽略文件的修改时间，默认忽略
ignoreContents	是否对每个字节进行比较，默认开启
granularity	用于指定一个文件修改时间的毫秒数据的允许误差。因为不是所有的文件系统 的修改时间都是精确到毫秒数。默认时为 0

如何判断两个文件是不同的呢？different 根据以下几个规则来进行判断：

1. 如果文件只存在于 fileset 中的 dir 属性指定的目录下，而不存在与 target 目录下，则文件被认为是不同的。

2. 如果文件只存在于 target 文件夹下，则文件被忽略。

3. 若文件大小不同，则文件被认为是不同的。

4. 如果属性 ignoreFileTimes 被设置为开启，则修改时间不同的两个文件被认为是不同的。

下面，笔者通过例子来展示如何使用 different ：

1. 创建 build.xml 文件，其中 init 实现初始化文件夹的作用。

创建文件夹 Base 以及文件 OnlyInBase.txt。创建文件夹 Target 以及文件 OnlyInTarget.txt。 在 Base 和 Target 下创建不同时间戳的同名文件 Timestamp.txt。

2. 其中 clean 用于清除已经创建的文件夹

3. Test 用于设置 Base 和 Target 文件夹并执行测试。代码如下:

```
<? xml version="1.0"? >
<project name="Test Selector different" default="Test">

<target name="init" depends="clean">
```

```
<mkdir dir="Base"/>
<mkdir dir="Target"/>
<mkdir dir="Result"/>
<touch file="Base/OnlyInBase.txt"/>
<touch file="Target/OnlyInTarget.txt"/>
<touch file="Base/Timestamp.txt" datetime="05/29/2015 04:  00 pm"/>
<touch file="Target/Timestamp.txt" datetime="05/29/2015 05:  00 pm"/>
</target>

<target name="clean">
<delete dir="Base"/>
<delete dir="Target"/>
<delete dir="Result"/>
</target>

<target name="Test" depends="init">
<copy todir="Result">
<fileset dir="Base">
<different targetdir="Target" ignoreFileTimes="False"/>
</fileset>
</copy>
</target>

</project>
```

根据 different 的判断规则，我们期望最后被复制的文件是 OnlyInBase.txt 和 Timestamp.txt，现在执行 Ant 命令查看实际结果:

```
→ Test_different git:(master) ✗ ant
Buildfile: /Users/lvchongen/Desktop/temp/Test_different/build.xml

clean:

init:
    [mkdir] Created dir: /Users/lvchongen/Desktop/temp/Test_different/Base
    [mkdir] Created dir: /Users/lvchongen/Desktop/temp/Test_different/Target
    [mkdir] Created dir: /Users/lvchongen/Desktop/temp/Test_different/Result
    [touch] Creating /Users/lvchongen/Desktop/temp/Test_different/Base/OnlyInBase.txt
    [touch] Creating /Users/lvchongen/Desktop/temp/Test_different/Target/OnlyInTarget.txt
    [touch] Creating /Users/lvchongen/Desktop/temp/Test_different/Base/Timestamp.txt
    [touch] Creating /Users/lvchongen/Desktop/temp/Test_different/Target/Timestamp.txt

Test:
     [copy] Copying 2 files to /Users/lvchongen/Desktop/temp/Test_different/Result

BUILD SUCCESSFUL
Total time: 0 seconds
→ Test_different git:(master) ✗ tree
.
|____Base
| |____OnlyInBase.txt
| |____Timestamp.txt
|____build.xml
|____Result
| |____OnlyInBase.txt
| |____Timestamp.txt
|____Target
| |____OnlyInTarget.txt
| |____Timestamp.txt
```

根据 different 的判断规则，我们期望最后被复制的文件是 OnlyInBase.txt 和 Timestamp.txt，现在执行 Ant 命令查看实际结果:

```
→ Test_different git:(master) ✗ ant
Buildfile: /Users/lvchongen/Desktop/temp/Test_different/build.xml

clean:

init:
    [mkdir] Created dir: /Users/lvchongen/Desktop/temp/Test_different/Base
    [mkdir] Created dir: /Users/lvchongen/Desktop/temp/Test_different/Target
    [mkdir] Created dir: /Users/lvchongen/Desktop/temp/Test_different/Result
    [touch] Creating /Users/lvchongen/Desktop/temp/Test_different/Base/OnlyInBase.txt
    [touch] Creating /Users/lvchongen/Desktop/temp/Test_different/Target/OnlyInTarget.txt
    [touch] Creating /Users/lvchongen/Desktop/temp/Test_different/Base/Timestamp.txt
    [touch] Creating /Users/lvchongen/Desktop/temp/Test_different/Target/Timestamp.txt

Test:
     [copy] Copying 2 files to /Users/lvchongen/Desktop/temp/Test_different/Result

BUILD SUCCESSFUL
Total time: 0 seconds
→ Test_different git:(master) ✗ tree
.
|____Base
| |____OnlyInBase.txt
| |____Timestamp.txt
|____build.xml
|____Result
| |____OnlyInBase.txt
| |____Timestamp.txt
|____Target
| |____OnlyInTarget.txt
| |____Timestamp.txt
```

Filename

filename 是用来匹配指定文件的 selector，它有几个常用的属性:

属性	描述
name	必要的属性，用于指定文件的名字，可以使用 Ant 的通配符

regex	用于进行文件名称筛选的正则表达
casesensitive	是否忽略文件名的字母母大小写，默认开启
negate	对结果进行翻转，默认为 false。设置为 true 时，表示不包含选中的文件

下面通过通过实际的例子来展示如何使用 filename:

1. 创建 init 用于初始化测试环境
2. 创建 Test_filename 用于使用 Ant 模式匹配的方式进行文件选择。
3. 完整的 build.xml 代码如下:

```
<? xml version="1.0"? >
<project name="Test filename" default="Test_filename">

<target name="init" depends="clean">
<mkdir dir="Result"/>
<mkdir dir="Source"/>
<touch file="Source/Hello.txt"/>
<touch file="Source/Ant.txt"/>
</target>

<target name="clean">
<delete dir="Result"/>
<delete dir="Source"/>
</target>
<target name="Test_filename" depends="init">
<copy todir="Result">
<fileset dir="Source" includes="**/*">
<filename name="**/*nt*.txt"/>
</fileset>
</copy>
</target>
</project>
```

根据定义，期望结果是将文件名称包含 nt 的文件拷贝到 Result 文件夹下，现在我们执行 Ant 命令查看实际结果:

可以看到，在 Result 文件夹下只有一个文件 Ant.txt，与我们的期望一致。

5. present present 用来选择有相同结构文件夹下的文件。该方法会选择 fileset 指定目录存在但 target 目录中不存在的文件，或两者都存在的文件。 下面使用实例来演示该方法的用法:

1. 创建文件夹 Source，Base 和 Result。其中 Base 存储文件 A.txt 和 B.txt，其中 A.txt 的内容为“I am from Base”。 Source 文件夹中存储文件 A.txt，其中 A.txt 的内容为“I am from Source”。

2. 创建 build.xml 文件，内容如下:

```
<? xml version="1.0"? >
<project name="Test present" >

<target name="Test_present_both">
<mkdir dir="Result_both"/>
<copy todir="Result_both">
<fileset dir="Base" includes="**/*">
<present targetdir="Source" present="both" />
</fileset>
</copy>
</target>
```

```
<target name="Test_present_srconly">
<mkdir dir="Result_srconly"/>
<copy todir="Result_srconly">
<fileset dir="Base" includes="**/*">
<present targetdir="Source" present="srconly" />
</fileset>
</copy>
</target>
</project>
```

1. 其中属性 present 有两个参数， both 和 srconly 。其中 present=srconly 表示只选取 fileset 中指定目录中存在但 target 目录中不存在的文件。而 present=both 表示选择指定目录和 target 目录中都存在的文件。

2. 分别执行命令 Ant Test_present_both 和 Ant Test_present_srconly 查看实际结果。根据属性的定义，Result_srconly 文件夹下的文件应仅存于 Base 下;Result_both 文件夹 下的文件应在 Base 和 Source 下同时存在。

```
→ Test_present git:(master) ✗ ant Test_present_both
Buildfile: /Users/lvchongen/Desktop/temp/Test_present/build.xml

Test_present_both:
    [mkdir] Created dir: /Users/lvchongen/Desktop/temp/Test_present/Result_both
     [copy] Copying 1 file to /Users/lvchongen/Desktop/temp/Test_present/Result_both

BUILD SUCCESSFUL
Total time: 0 seconds
→ Test_present git:(master) ✗ ant Test_present_srconly
Buildfile: /Users/lvchongen/Desktop/temp/Test_present/build.xml

Test_present_srconly:
    [mkdir] Created dir: /Users/lvchongen/Desktop/temp/Test_present/Result_srconly
     [copy] Copying 1 file to /Users/lvchongen/Desktop/temp/Test_present/Result_srconly

BUILD SUCCESSFUL
Total time: 0 seconds
→ Test_present git:(master) ✗ tree
.
|____Base
| |____A.txt
| |____B.txt
|____build.xml
|____Result_both
| |____A.txt
|____Result_srconly
| |____B.txt
|____Source
| |____A.txt
```

可以看到，实际的结果与预期的一致。

6. containsregexp

筛选符合正则表达式的文件。该 selector 有核心属性 expression，其值是正则表达式

用来匹配文件的 内容。下面以实际的例子展示 containsregexp 的用法。

1. 创建 build.xml 文件，其中 init 和 clean 用于创建和删除文件夹和文件。

2. 因为 containsregexp 是匹配文件内容的，所以将文件内容通过属性 <echo> 写入文件。

3. build.xml 的代码如下:

```
<? xml version="1.0"? >
<project name="Test present" default="Test_containsregexp" >

<target name="clean">
<delete dir="Source"/>
<delete dir="Result"/>
</target>

<target name="init" depends="clean">
<mkdir dir="Source" />
<mkdir dir="Result" />
<echo file="Source/Windows10_OS.txt">Windows10_OS</echo>
<echo file="Source/Windows8_OS.txt">Windows8_OS</echo>
<echo file="Source/Windows7_OS.txt">Windows7_OS</echo>
</target>
<target name="Test_containsregexp" depends="init">
<copy todir="Result">
<fileset dir="Source" includes="*.txt">
<containsregexp expression="Windows\d+.*" />
</fileset>
</copy>
</target>

</project>
```

4. 正则表达式 Windows\d+.* 可以匹配 Windows 起始后接数字，并且数字后匹配

任意字符。根据文件的内容，期望的结果是可以将三个文件都复制到 Result 文件夹下，现在执行命令 Ant 查看实际的运行结果:

```
Test_Containsregexp git:(master) ✗ ant
Buildfile: /Users/lvchongen/Desktop/temp/Test_Containsregexp/build.xml

clean:

init:
    [mkdir] Created dir: /Users/lvchongen/Desktop/temp/Test_Containsregexp/Source
    [mkdir] Created dir: /Users/lvchongen/Desktop/temp/Test_Containsregexp/Result

Test_containsregexp:
     [copy] Copying 3 files to /Users/lvchongen/Desktop/temp/Test_Containsregexp/Result

BUILD SUCCESSFUL
Total time: 0 seconds
Test_Containsregexp git:(master) ✗ tree
.
|____build.xml
|____Result
| |____Windows10_OS.txt
| |____Windows7_OS.txt
| |____Windows8_OS.txt
|____Source
| |____Windows10_OS.txt
| |____Windows7_OS.txt
| |____Windows8_OS.txt
```

可以看到成功的复制了三个文件到 Result 文件夹下。

7. size size 是通过文件的大小对文件进行筛选，它有如下几个属性

属性	描述
value	用于比较文件的大小
units	属性 value 的单位，K=1000bytes，M=1000K，G=1000M
when	可以指定为 more(大于)，less(小于)，equal(等于)，用于配合判断文件大小，条件成立时选文件

下面通过实际的例子展示 size 的使用:

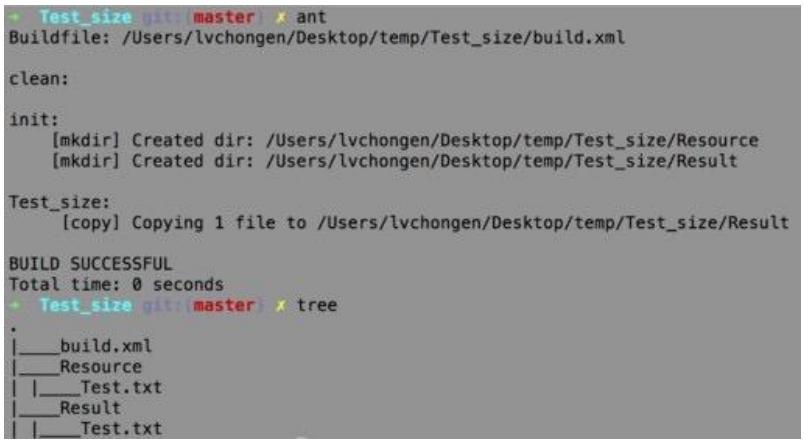
1. 创建 build.xml 文件，使用 clean 和 init 用于清理和初始化测试环境，其中 init 会在 Resource 文件夹下创建名为 Test.txt 的文件，内容仅仅为 Hello Size ，所以文件的大小仅仅是 10byte

2. 设置判断条件 <size value="100" units="K" when="less"/> 来进行文件大小的比对。

3. 完整的代码如下:

```
<? xml version="1.0"? >
<project name="Test size" default="Test_size" >

<target name="clean">
<delete dir="Resource"/>
<delete dir="Result"/>
</target>

<target name="init" depends="clean">
<mkdir dir="Resource"/>
<mkdir dir="Result"/>
<echo file="Resource/Test.txt">Hello Size</echo>
</target>

<target name="Test_size" depends="init">
<copy todir="Result">
<fileset dir="Resource" includes="*.*">
<size value="100" units="K" when="less"/>
</fileset>
</copy>
</target>
</project>
```

4. 按照文件的实际大小和判断的条件，期望结果是将文件 Test.txt 复制到文件夹 Result 下。现在执行命令 Ant 查看实际结果:



可见，实际结果是将文件复制到了文件夹 Result 下，与期望结果一致。

8. type

根据文件的类型进行选择，该 selector 只有两个属性 dir 和 file 分别代表文件夹和文件。因使用较为单，这里就不一一举例了。

至此，核心的 selector 就全部介绍完毕了，掌握好 selector 能够帮助我们准确的选择文件进行操作，在整个自动构建过程中起到最重要的作用。

FilterSet

从字面可以看出这是一个包含 filter 的集合。filter 可以被定义为 token-value 的形式，也可以从文件中读取。

FilterSet

属性	描述
begintoken	指定一个特殊字符，用于过滤字符串的起始位置
endtoken	指定一个特殊字符，用于过滤字符串的结束位置
filtersfile	引入外部记录过滤内容的文件
recurse	表示是否可以查找更多的替换标志，默认为 true
onmissingfiltersfile	若 filtersfile 不存在后的操作，包括‘fail’,'warn'和'ignore'

Filter

属性	描述
token	指定要被替换的标示符
value	指定替换后的值

Filtersfile

属性	描述
file	通过 name-value 方式描述过滤器的文件

现在通过实际例子来展示这几个过滤器的使用方法:

1. 创建 filter.properties 文件，内容为 CONTENT=I am from file 。
2. 创建 build.xml 文件，内容为:

```
<? xml version="1.0"? >
<project name="Test filterset" default="Test_filterset" >

<target name="clean">
<delete dir="Source"/>
<delete dir="Result"/>
</target>

<target name="init" depends="clean">
<mkdir dir="Source"/>
<mkdir dir="Result"/>
<echo file="Source/Test.txt">*ENVIRONMENT%，*CONTENT%</echo>
</target>

<target name="Test_filterset" depends="init">
<copy file="Source/Test.txt" toFile="Result/Test.txt">
```

```
<filterset begintoken="*" endtoken="%">
<filter token="ENVIRONMENT" value="${Ant.home}"/>
<filtersfile file="filter.properties"/>
</filterset>
</copy>
</target>
</project>
```

3. 根据过滤器及其属性的定义，我们可以明白 Test_filterset 的意图，那么期望结果是复制文件后并将对应的变量替换为 Ant.home 的实际值和"I am from file"，我们执行命令 Ant 来查看实际结果:

```
→  Test_filterset git:(master) ✗ ant
Buildfile: /Users/lvchongen/Desktop/temp/Test_filterset/build.xml

clean:

init:
    [mkdir] Created dir: /Users/lvchongen/Desktop/temp/Test_filterset/Source
    [mkdir] Created dir: /Users/lvchongen/Desktop/temp/Test_filterset/Result

Test_filterset:
     [copy] Copying 1 file to /Users/lvchongen/Desktop/temp/Test_filterset/Result

BUILD SUCCESSFUL
Total time: 0 seconds
→  Test_filterset git:(master) ✗ tree
.
|____build.xml
|____filter.properties
|____Result
| |____Test.txt
|____Source
| |____Test.txt
→  Test_filterset git:(master) ✗ cat Result/Test.txt
/usr/local/apache-ant-1.9.4,I am from file
→  Test_filterset git:(master) ✗ cat Source/Test.txt
*ENVIRONMENT%,*CONTENT%
```

可以看到成功复制了文件并修改了内容。

filelist

顾名思义，这是一个 file 的集合，下面通过实际例子来展示如何使用该过滤器。

创建 build.xml，其中 Test_filelist 的作用是根据 filelist 的集合进行文件筛选并复制。代码如下:

```
<? xml version="1.0"? >
<project name="Test filelist" default="Test_filelist" >


<target name="clean">
<delete dir="Source"/>
```

```
<delete dir="Result"/>
</target>

<target name="init" depends="clean">
<mkdir dir="Source"/>
<mkdir dir="Result"/>
<touch file="Source/A.txt"/>
<touch file="Source/B.txt"/>
<touch file="Source/C.txt"/>
<touch file="Source/D.txt"/>
</target>

<target name="Test_filelist" depends="init">
<copy todir="Result">
<filelist id="source" dir="Source">
<file name="A.txt"/>
<file name="B.txt"/>
<file name="C.txt"/>
</filelist>
</copy>
</target>

</project>
```

2. 期望结果是将 Source 下的 A.txt，B.txt，C.txt 复制到文件夹 Result 下。执行命令 Ant 查看实际结果:

```
Test_filelist git:(master) ✗ ant
Buildfile: /Users/lvchongen/Desktop/temp/Test_filelist/build.xml

clean:

init:
    [mkdir] Created dir: /Users/lvchongen/Desktop/temp/Test_filelist/Source
    [mkdir] Created dir: /Users/lvchongen/Desktop/temp/Test_filelist/Result
    [touch] Creating /Users/lvchongen/Desktop/temp/Test_filelist/Source/A.txt
    [touch] Creating /Users/lvchongen/Desktop/temp/Test_filelist/Source/B.txt
    [touch] Creating /Users/lvchongen/Desktop/temp/Test_filelist/Source/C.txt
    [touch] Creating /Users/lvchongen/Desktop/temp/Test_filelist/Source/D.txt

Test_filelist:
     [copy] Copying 3 files to /Users/lvchongen/Desktop/temp/Test_filelist/Result

BUILD SUCCESSFUL
Total time: 0 seconds
Test_filelist git:(master) ✗ tree
.
|____build.xml
|____Result
| |____A.txt
| |____B.txt
| |____C.txt
|____Source
| |____A.txt
| |____B.txt
| |____C.txt
| |____D.txt
```

至此，我们将 Ant 中的 Data Type 全部介绍了，希望读者切实掌握这些在 Ant 中是重中之重的技能。

Ant 实际使用之构建 Java 文档

Javadoc 是 Sun 公司提供的一个技术，它能够从程序源代码中抽取类，方法，成员等注释生成一个和源代码对应的 API 帮助文档。在编写程序程序时以特定的标签作注释，通过 javadoc 就可以同时生成程序的开发文档。

javadoc 注释规范

属性	描述
@author	标明开发该类模块的作者
@version	标明该类模块的版本
@parama	标明方法的参数名称及描述
@return	标明函数返回的注释
@throws	标明构造函数或方法所会抛出的异常
@exception	标明构造函数或方法所会抛出的异常

@see	标明查看相关内容，如类、方法、变量等
@since	标明 API 在什么程序的什么版本后开发支持

示例

1. 创建 Java 项目，并使用注释。文件 HelloAnt.java 代码如下:

```
public class HelloAnt {

public HelloAnt(){
}

/**
* @author lvchongen
* @version 1.0
* @param  输入两个 string
* @return  返回拼接的字符串
*/
public String getString(String a，  String b) {
return a+b;
}
}
```

2. 创建 build.xml 文件进行文档编译。代码内容如下:

```
<? xml version="1.0"? >
<project name="Test javadoc" default="Test_javadoc" >

<target name="clean">
<delete dir="doc"/>
</target>

<target name="init" depends="clean">
```

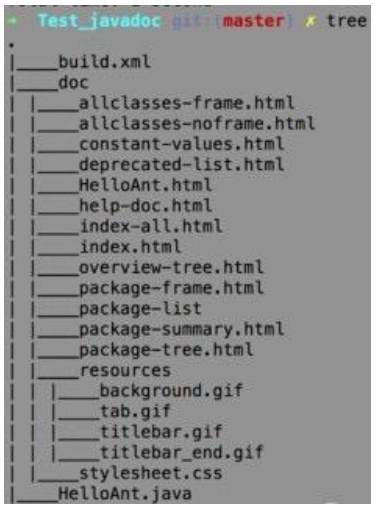
```
<mkdir dir="doc"/>
</target>
<target name="Test_javadoc" depends="init">
<javadoc  destdir="doc" windowtitle="Hello Ant" charset="UTF-8" enc oding="UTF-8" docencoding="UTF-8">
<fileset dir="./"/>
</javadoc>
<echo message = "java doc has been generated!" />
</target>
</project>
```

3. 标签 <javadoc> 的作用是对选中 java 文件生成文档。

属性	描述
windowtitle	文档的题目
fileset	java 文件所在目录
charset="UTF-8" encoding="UTF-8" docencoding="UTF-8"	支持中文

4. 执行 Ant 命令查看实际的运行结果:

可以看到在 doc 文件夹下生成了若干 html 文件:



用浏览器打开 index.html 查看实际的生成结果

良好的注释习惯以及直观的帮助文档能够帮助理解代码，为项目的稳定性，可复用性提高了保证。

Ant 实际使用之项目打包 Jar

什么是 Jar

Jar(Java Archive，Java 归档文件)是与平台无无关的文件格式。JAR 通常聚合大量的 Java 类文件，相关的元数据和资源等到一个文件，以便发布 Java 平台应用软件或者库。

示例

这里我们以打包一个可运行的 Jar 文件为例:

1. Manifest

在 Java 平台中，Manifest 资源配置文件是 JAR 归档中所包含的特殊文件。Manifest 文件被用来定义扩展或档案打包相关数据。Manifest 文件是一个元数据文件，它包含了不同部分中的键-值对数据。如果一个 JAR 文件被 当作可执行文件，则其中的 Manifest 文件需要指出该程序的主类文件。通常 Manifest 文件的文件名为 MANIFEST.MF。

可见 Manifest 可打包成可执行 Jar 的必备文件之一。

2. Ant 中的 Jar 属性

在 Ant 中可以使用 <jar> 对文件进行打包，它有如下几个属性:

属性	描述
basedir	指定需要被打包文件的目录
compress	指定打包过程中压缩文件，默认为 True

keepcompression	指定已经压缩的文件保持原先的压缩格式，默认为 False
destfile	指定打包生成文件的名字
duplicate	定义当出现重复文件时的处理方式。可取值 add、preserve 和 fail。add 代表依然添加（覆盖）文件，preserve 代表不打包重复文件，fail 代表将打包失
excludes	指定一个或多个不被打包进入 jar 的文件
excludesfile	指定一个或多个不被打包进入 jar 的文件，但可以通过模式匹配进行筛选
inlcudes	指定需要被打包进入 jar 的文件，属性忽略时代表所有文件都被打包
includesfile	指定需要被打包进入 jar 的文件，属性忽略时代表所有文件都被打包，但可以通过模式匹配进行筛选
update	定义是否更新或覆盖目标文件，当目标文件已存在时。默认为 false

3. 实际使用

1. 创建 Hello.java 文件，用于打印 Hello + 参数 。内容如下:

```
public class Hello {

public static void main(String[] args) { String content = "Hello "; if(args.length > 0) {
for(int i=0; i<args.length;i++){
content= content + " " + args[i];
}
}
System.out.println(content);
}
}
```

2. 创建 build.xml 文件用于打包 jar 文件，因为需要打包可执行的 jar 文件，所以需要创建 Mainfest 文件进行配置入口，于是我们引入了标签 <manifest> 来指定入口，根据 Mainfest 的定义，内容应是 key-value 的格式，而描述函数入口的属性是 Main-class， 根据主函数 main 所在的类我们知道入口的值应为 Hello。

build.xml 的内容如下:

```
<? xml version="1.0"? >
<project name="Test jar" default="Test_jar" >

<property name="src" value="src"/>
<property name="dest" value="classes"/>

<target name="clean">
<delete dir="${src}"/>
<delete dir="${dest}"/>
</target>

<target name="init" depends="clean">
<mkdir dir="${src}"/>
<mkdir dir="${dest}"/>
<copy todir="${src}/" file="Hello.java"/>
</target>

<target name="compile" depends="init">
<javac srcdir="${src}" destdir="${dest}" includeAntruntime="on" /
>
</target>

<target name="Test_jar" depends="compile">
<jar destfile="Hello.jar" basedir="${dest}" >
<manifest>
<attribute name="Main-class" value="Hello"/>
</manifest>
</jar>
</target>
</project>
```

3. 执行 Ant 命令查看实际的运行结果:

```
→ Test_jar git:(master) ✗ ant
Buildfile: /Users/lvchongen/Desktop/temp/Test_jar/build.xml

clean:

init:
    [mkdir] Created dir: /Users/lvchongen/Desktop/temp/Test_jar/src
    [mkdir] Created dir: /Users/lvchongen/Desktop/temp/Test_jar/classes
     [copy] Copying 1 file to /Users/lvchongen/Desktop/temp/Test_jar/src

compile:
    [javac] Compiling 1 source file to /Users/lvchongen/Desktop/temp/Test_jar/classes

Test_jar:
      [jar] Building jar: /Users/lvchongen/Desktop/temp/Test_jar/Hello.jar

BUILD SUCCESSFUL
Total time: 1 second
→ Test_jar git:(master) ✗ tree
.
|____build.xml
|____classes
| |____Hello.class
|____Hello.jar
|____Hello.java
|____src
| |____Hello.java
```

4. 执行 java -jar Hello.jar ， My name is lvchongen 来查看是否输出了正确地结果:

```
→ Test_jar git:(master) ✗ java -jar Hello.jar , My name is lvchongen
Hello  , My name is lvchongen
```

可以看到我们已经成功的使用 Ant 创建了 jar 包。读者可以根据自己的项目结构对需要打包的文件进行选择，创建需要的 jar 包。也可以阅读开源项目中的 build.xml 文件进行学习，如 JMeter。

Ant 实际使用之 Android 渠道包编译

什么是渠道包

Android 应用的发布需要面对各种各样的市场，我们称之为渠道。有的时候，我们需要知道应用是从哪个渠道下载的。比如，我们可能需要统计哪些市场带来的用户量比较大。再比如，我们可能有一些盈利需要和具体的渠道进行分成。这些都是统计渠道的信息。

如何修改渠道包

想知道如何修改渠道，我们先要了解在哪里配置对应的渠道。笔者以常用的友盟统计为例，该工具需要在 AndroidManifest.xml 文件中进行配置，一般渠道名称出现在这样的标签中 <meta-data android: name="UMENG_CHANNEL" android: value="渠道名称" /> ，可以通过修改 value 的值成不同的名称达到渠道统计的目的。

看到这里我想读者已经有了不同的解决方案，其中比较直接的就是进行字符串的替

换。对 AndroidManinfest.xml 进行对应字段的正则表达式匹配后，替换渠道名称。

1. 介绍 Ant 中的正则匹配功能 ReplaceRegExp

ReplaceRegExp 是根据指定的正则匹配表达式在选中的文件中进行匹配及替换。ReplaceRegExp 有如下属性:

属性	描述
file	指定需要被匹配的文件
match	指定需要匹配的正则表达式
replace	指定需要被替换的表达式
flags	指定匹配的模式，G 表示全局替换，M 表示匹配以"^"或"$"为起始或结束的字符串
byline	指定一行一行进行筛选替换
encoding	指定文件的编码格式
preserveLastModified	文件修改后保存时间戳

2. 创建 custom_rules.xml 文件作为 build.xml 文件的引用，其主要的职责是进行正则表达式的匹配以及匹配后的替换。我们需要使用标签 <ReplaceRegExp> 来进行正则匹配的一系列操作。核心代码如下:

```
<replaceregexp flags="g" byline="false" encoding="UTF-8">
<regexp pattern="android: name=&amp;quot;UMENG_CHANNEL&amp;quot;(.*)\s+android: value=&amp;quot;(.*)&amp;quot;" />
<substitution expression="android: name=&amp;quot;UMENG_CHANNEL&amp;quot;&amp;#x000A;android: value=&amp;quot;${channel}&amp;quot;" />
<fileset dir="" includes="AndroidManifest.xml" />
```

</replaceregexp>

3. 这里简单分析下这段代码的作用

android: name="UMENG_CHANNEL"(.*)\s+android: value="(.*)"

是用来进行匹配的正则表达式，会在 AndroidManifest.xml 全局查找并匹配。

匹配到 android: value="渠道名称"后，使用变量 ${channel} 进行替换，而该变量是以 <property file="local.properties" /> 的形式引入的，该文件中已 key-value 的形式描述了不同的渠道，例如 market_channels=qq，xiaomi，wandoujia 。

4. 编译打包的过程中会完成渠道的替换。

到这里，关于 Ant 的实用教程就全部介绍完毕了。因为篇幅所限，难以将 Ant 中的每个标签都一一介绍，所以 这里只介绍了常用的重要的标签，以及几个自动构建中常用的例子，希望能够起到抛砖引玉的作用。笔者认为在持续集成的过程中，Ant 起到了自动构建的作用，希望读者能够认真学习并锻炼使用。