第3章

Ant 构建工具

Ant 是构建工具，构建命令定义在 build.xml 中，有利于重复执行。Ant 在 JDK 本身提供的 javac、jar、java 命令基础上进行了扩展，使得构建项目更简单，而且提供了更丰富的功能（文件操作、数据库读写、发送邮件等）。

3.1 Ant 的特点

1. Ant 的优点

Ant 有以下优点：

（1）跨平台：因为 Ant 是使用 Java 实现的，所以它可以跨平台。

（2）使用简单：Ant 比 make 更简单。

（3）语法清晰：Ant 比 make 语法更清晰。

（4）功能强大：Ant 可以做的事情很多。当开发 Ant 插件时，会发现它更多的功能。

2. Ant 与 make 的比较

Ant 所做的很多事情，以往大部分是由 make 做的，不过对象不同，make 更多应用于 C/C++，Ant 更多应用于 Java。当然这不是一定的，但大部分人习惯如此。

3.2 下载、安装和测试 Ant

3.2.1 下载 Ant

从 Apache 官方网站 http://ant.apache.org/可下载 Ant。

不同版本要求提供支持的 JDK 版本不一样，这里需要 Java 8 支持的版本，相应的安装包为 apache-ant-1.10.9-bin.zip。

3.2.2 安装 Ant

Ant 属于绿色软件，直接解压就可以。具体安装步骤如下：

(1) 将安装包解压到 D:\apache-ant-1.10.9 目录。

(2) 添加系统环境变量：ANT_HOME 为 D:\apache-ant-1.10.9。

(3) 将 bin 路径添加到环境变量 Path 中：%ANT_HOME%\bin。

3.2.3　测试 Ant

在命令行中输入 ant -version 命令查看 Ant 版本信息，测试安装是否成功，如图 3.1 所示。

图 3.1　查看 Ant 版本信息

3.3　初 识 Ant

在没有工具时，用 javac 编译，再用 jar 打包，最后用 java 运行，非常不方便。这里展示如何用 Ant 创建目录以及编译、打包和运行项目。Ant 默认的配置文件是 build.xml，它可以视为 Ant 的程序。

3.3.1　build.xml 文件

Ant 通过解析 build.xml 文件来运行。build.xml 文件根标签为＜project＞，示例如下：

```
<project name="HelloWorld" default="run" basedir=".">
</project>
```

Ant 的所有内容必须包含在＜project＞标签中，name 是程序的名字，basedir 是工作的根目录，"."代表当前目录。default 代表默认要做的事情。

＜property＞标签类似程序中定义的变量，以便全局使用。使用时的格式为 ${xxx}。＜property＞标签示例如下：

```
<property name="src" value="src"/>
```

把做的每一件事情写成一个＜target＞标签，示例如下：

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

compile 是它的名字，depends 是它依赖的 target。例如，在执行 compile 之前，Ant 会先检查 init 是否曾经被执行过。如果执行过，则直接执行 compile；否则会先执行它依赖的 target，例如这里的 init，然后再执行这个 target。

Ant 中的每一个任务都可以这样调用：ant target="xxx"。

本例中 build.xml 的详细定义如下：

```
<?xml version="1.0" encoding="utf-8" ?>
<project name="HelloWorld" default="run" basedir=".">
    <property name="src" value="src" />
    <property name="dest" value="classes" />
    <property name="hello_jar" value="hello1.jar" />
    <target name="init">
        <mkdir dir="${dest}" />
        <mkdir dir="${src}" />
    </target>
    <target name="compile" depends="init">
        <javac srcdir="${src}" destdir="${dest}" includeantruntime="on" />
    </target>
    <target name="build" depends="compile">
        <jar jarfile="${hello_jar}" basedir="${dest}" />
    </target>
    <target name="run" depends="build">
        <java classname="Hello" classpath="${hello_jar}" />
    </target>
    <target name="clean">
        <delete dir="${dest}" />
        <delete file="${hello_jar}" />
    </target>
    <target name="rerun" depends="clean,run">
        <ant target="clean" />
        <ant target="run" />
    </target>
    <target name="rerun1" depends="clean,run">
    </target>
    <target name="rerun2">
        <ant target="clean" />
        <ant target="run" />
    </target>
</project>
```

3.3.2 创建目录

在创建项目时，要规划项目的目录结构，例如，src 放源程序，classes 放编译的类。以前没有工具时，需要手工创建目录。这里利用 Ant 的＜mkdir＞创建目录标签，可以自动、重复建立，非常方便。首先定义源程序目录和编译的类目录，以便后面重复使用。目录定义用＜property＞标签，配置如下：

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

定义一个任务，任务名称为 init，任务内容为创建目录，配置如下：

```
<target name="init">
    <mkdir dir="${dest}" />
    <mkdir dir="${src}" />
</target>
```

运行 ant init 命令，创建目录，然后运行 dir 命令列出目录，如图 3.2 所示，可以看出创建了两个目录。

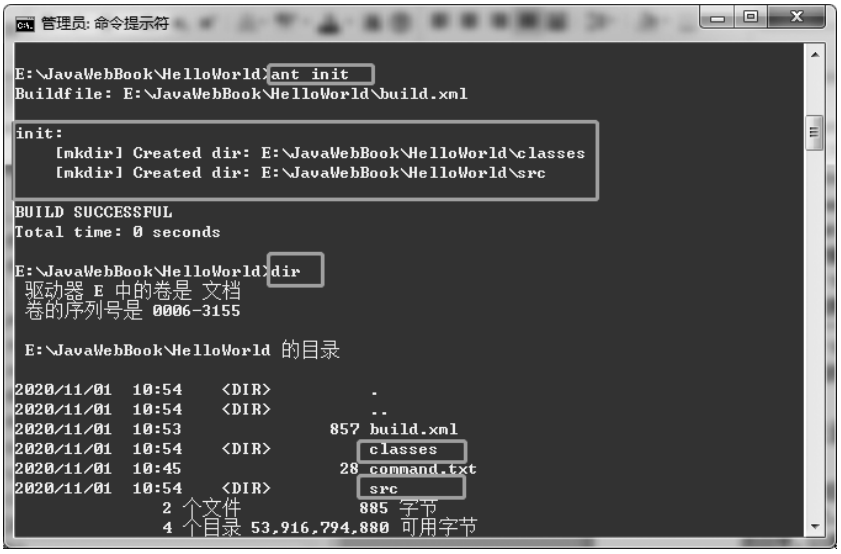

图 3.2　创建目录并列出目录

3.3.3　编译任务

编译不用手工输入 javac 命令完成，定义编译任务，然后运行编译任务即可。可以利用<javac>标签编译 Java 原代码，srcdir 属性表示源程序目录，destdir 属性表示编译输出目录，定义的编译任务如下：

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

这里定义编译任务依赖 init 任务，也就是说，要运行编译任务，首先要运行 init 任务。

利用 ant compile 命令运行编译任务，编译项目，并把编译的类存放到 classes 目录下，然后运行 dir 命令列出目录，如图 3.3 所示。

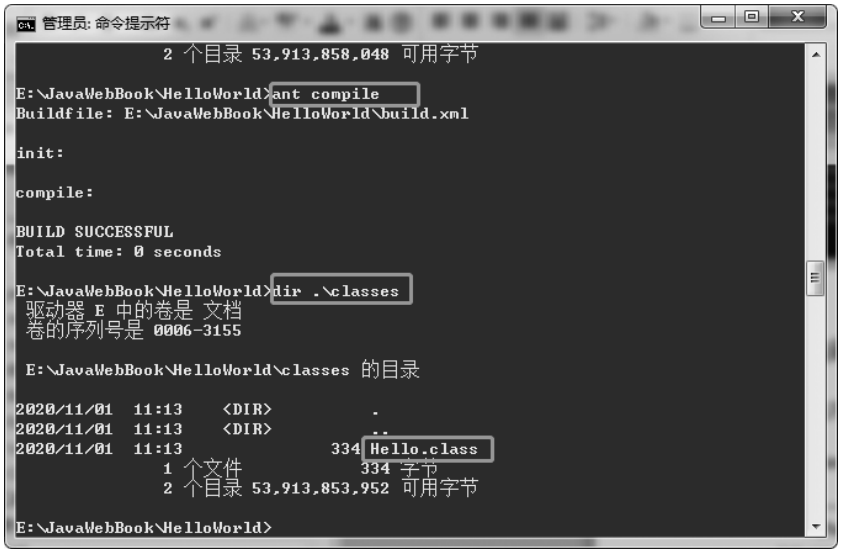

图 3.3 编译项目并列出目录

3.3.4 打包任务

同样,打包也不用手工输入 jar 命令,定义打包任务,然后运行打包任务即可。<jar>标签是打包命令,jarfile 属性为要打包成的 jar 文件,basedir 为被打包的目录,定义的打包任务如下:

```
<target name="build" depends="compile">
    <jar jarfile="${hello_jar}" basedir="${dest}" />
</target>
```

这里定义打包任务依赖 compile 任务。

利用 ant build 命令运行打包任务,打包项目,然后运行 dir 命令列出目录,如图 3.4 所示。

图 3.4 打包项目并列出目录

3.3.5 运行任务

也可以通过 Ant 运行 java 命令。<java>标签是运行命令,classname 属性为运行的

主类，classpath 为运行的库，定义运行任务如下：

```
<target name="run" depends="build">
    <java classname="Hello" classpath="${hello_jar}" />
</target>
```

利用 ant run 命令运行 Hello 程序，结果如图 3.5 所示。运行输出了“Hello World!。”另外也可以看出运行整个生命周期包括 init、compile、build 和 run。

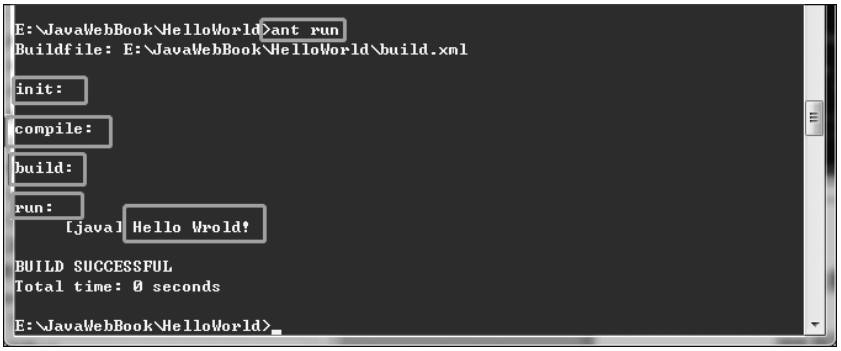

图 3.5　运行项目

3.3.6　清除任务

清除任务就是清除编译和打包的结果，便于下一次编译和打包。＜delete＞标签可以删除目录及文件，定义如下：

```
<target name="clean">
    <delete dir="${dest}" />
    <delete file="${hello_jar}" />
</target>
```

3.3.7　重新运行任务

重新运行任务就是先运行清除任务，再运行一次任务。重新运行任务定义如下：

```
<target name="rerun" depends="clean,run">
    <ant target="clean" />
    <ant target="run" />
</target>
```

这里示例了如何在任务中调用任务。上述定义会执行两遍，可以定义如下两种形式，只执行一遍。

(1) 将任务添加到依赖中，如下：

```
<target name="rerun1" depends="clean,run">
</target>
```

(2) 将任务添加到标签<ant>中,如下:

```
<target name="rerun2">
    <ant target="clean" />
    <ant target="run" />
</target>
```

3.3.8 生成清单文件

<manifest>是 Ant 内置任务,用于创建清单文件,或者替换、更新已有清单文件。清单文件由一组属性和组组成。<manifest>根据 Java 文件规范进行处理。

(1) file:要创建或更新的清单文件。

(2) mode:模式,update 或 replace,默认为 replace。

(3) encoding:更新时读取已有清单文件使用的编码,写清单文件时总是为 utf-8,默认为 utf-8。

(4) mergeClassPathAttributes:从 Ant 1.8 起,如果是 update 模式,还要确定是否合并在不同清单文件中找到的 class-path 属性。如果为 false,则只保留最新清单的属性;如果为 true,还需要将 flattenAttributes 属性设置为 true,否则可能会导致包含多个 class-path 属性而违反 Java 文件规范。默认为 false。

(5) flattenAttributes:从 Ant 1.8 起,要确定是否合并同一节中多次出现的 class-path 属性到单个属性中,默认为 false。

<manifest>标签支持以下嵌套元素:

(1) attribute:清单文件的一个属性,不会嵌套到其他的节中,即嵌套到主节中。它有以下属性:

- name:属性的名称,必须与"[A-Za-z0-9][A-Za-z0-9-_]*"模式匹配。
- value:属性的值。

(2) section:节,可以将属性嵌套到节中。在其 name 属性中设置节的名字。如果不设置,默认为主节。

示例如下:

```
<!-- 生成清单文件 -->
<manifest file="${meta.dir}/MANIFEST.BAK" mode="replace">
    <attribute name="Built-By" value="${user.name}" />
    <attribute name="Built-Date" value="${TODAY}" />
    <attribute name="Main-Class" value="${main-class}" />
    <attribute name="Class-Path" value="${quote.classpath}" />
</manifest>
```

3.4 Ant 文件命令

3.4.1 创建目录命令

mkdir 命令用于创建一个目录,如果其父目录不存在,会被同时创建。例如:

```
<mkdir dir="build/classes"/>
```

说明：如果 build 不存在，会被同时创建。

3.4.2　复制命令

copy 命令用于复制一个或一组文件、目录。以下是一些例子。

(1) 复制一个文件：

```
<copy file="myfile.txt" tofile="mycopy.txt"/>
```

(2) 复制一个文件到指定目录下：

```
<copy file="myfile.txt" todir="../some/other/dir"/>
```

(3) 复制一个目录到另一个目录下：

```
<copy todir="../new/dir">
<fileset dir="src_dir"/>
</copy>
```

(4) 复制一组文件到指定目录下：

```
<copy todir="../dest/dir">
<fileset dir="src_dir">
<exclude name="**/*.java"/>
</fileset>
</copy>
<copy todir="../dest/dir">
<fileset dir="src_dir" excludes="**/*.java"/>
</copy>
```

(5) 复制一组文件到指定目录下，文件名后增加“.bak”后缀：

```
<copy todir="../backup/dir">
<fileset dir="src_dir"/>
<mapper type="glob" from="*" to="*.bak"/>
</copy>
```

(6) 复制一组文件到指定目录下，替换其中的标签内容：

```
<copy todir="../backup/dir">
<fileset dir="src_dir"/>
<filterset>
<filter token="TITLE" value="Foo Bar"/>
</filterset>
</copy>
```

3.4.3　删除命令

delete 命令用于删除一个或一组文件、目录。以下是一些例子。

(1) 删除一个文件：

```
<delete file="/lib/ant.jar"/>
```

(2) 删除指定目录及其子目录：

```
<delete dir="lib"/>
```

(3) 删除指定一组文件：

```
<delete>
<fileset dir="." includes="**/*.bak"/>
</delete>
```

(4) 删除指定目录及其子目录，包括空目录：

```
<delete includeEmptyDirs="true">
<fileset dir="build"/>
</delete>
```

3.4.4 移动命令

move 命令用于移动或重命名一个或一组文件、目录。以下是一些例子。

(1) 移动或重命名一个文件：

```
<move file="file.orig" tofile="file.moved"/>
```

(2) 移动或重命名一个文件到另一个目录下：

```
<move file="file.orig" todir="dir/to/move/to"/>
```

(3) 将一个目录移到另一个目录下：

```
<move todir="new/dir/to/move/to">
<fileset dir="src/dir"/>
</move>
```

(4) 将一组文件移动到另一个目录下：

```
<move todir="some/new/dir">
<fileset dir="my/src/dir">
<include name="**/*.jar"/>
<exclude name="**/ant.jar"/>
</fileset>
</move>
```

(5) 在移动文件的过程中增加“.bak”后缀：

```
<move todir="my/src/dir">
<fileset dir="my/src/dir">
<exclude name="**/*.bak"/>
```

```
</fileset>
<mapper type="glob" from="*" to="*.bak"/>
</move>
```

3.5　其他命令

3.5.1　时间戳命令

tstamp 是 Ant 内置任务，用于在当前项目中设置 DSTAMP、TSTAMP 和 TODAY 属性。默认情况下，DSTAMP 属性的格式为 yyyyMMdd，TSTAMP 属性的格式为 hhmm，TODAY 属性的格式为 MMMM dd yyyy。可以嵌套 format 元素创建新的日期属性。

可以用 prefix 设置属性的前缀，以避免属性命名冲突。默认无前缀。

tstamp 支持 format、property、pattern、timezone、offset、unit、locale 嵌套元素。

（1）format：将属性值设置为指定格式的当前日期和时间。还可以将偏移量应用于时间，以生成不同的时间值。

（2）property：定义属性名称，接收以指定格式生成的时间/日期字符串。

（3）pattern：使用的时间/日期格式，时间/日期格式定义在 java.text.SimpleDateFormat 类中。

（4）timezone：显示时间使用的时区，时区定义在 java.util.TimeZone 类中。

（5）offset：当前时间的偏移数量。

（6）unit：偏移量单位，可选值为 millisecond、second、minute、hour、day、week、month、year。

（7）locale：用于创建时间/日期字符串的区域设置。通常格式为“language,country,variant”，variant 或者 variant 和 country 可以省略。具体可参考 java.util.Locale 类。

例如，下面生成一个新的日期属性 age，并显示 DSTAMP、TSTAMP、TODAY、ago 4 个日期属性。

```
<tstamp prefix="time">
    <format property="ago" pattern="MM/dd/yyyy hh:mm " offset="-1" unit=
        "hour" />
</tstamp>
<echo message="time.DSTAMP=${time.DSTAMP},
               time.TSTAMP=${time.TSTAMP},
               time.TODAY=${time.TODAY},
               time.ago=${time.ago}" />
```

运行结果如图 3.6 所示。

```
     [echo] time.DSTAMP=20201102,time.TSTAMP=2111,time.TODAY=November 2 2020,tim
e.ago=11/02/2020 08:11
```

图 3.6　tstamp 任务生成日期属性

3.5.2 执行 SQL 语句

部署数据库时往往需要更新数据库，Ant 支持对数据库的操作，通过 JDBC 执行 SQL 语句。

Ant 使用＜sql＞标签执行 SQL 语句或 sql 文件，在这个标签中必须有的属性如下：

（1）driver：数据库驱动程序名，MySQL 的数据库驱动程序名是"com.mysql.jdbc.Driver"，Oracle 的数据库驱动程序名是"oracle.jdbc.driver.OracleDriver"。

（2）url：数据库 URL，MySQL 的数据库 URL 是"jdbc：mysql://IP/数据库名"，Oracle 的数据库 URL 是"jdbc:oracle:thin:@IP:port:SID"。

（3）userid：数据库用户

（4）password：数据库用户密码。

（5）classpath：数据库连接的 jar 包，MySQL 的 jar 包是"mysql-connector-java-版本号-bin.jar"，Oracle 的 jar 包是"ojdbc6-版本号.jar"。

print 属性如果置为 true，则会打印执行 SQL 语句的详细输出信息。

要执行的 SQL 语句（可以多个）可直接放在＜sql＞标签中，也可放在一个 sql 文件中，然后将文件名赋值给＜sql＞的子标签＜transaction＞的 src 属性。

下面定义一个 SQL 任务 sqlselect，获取学生表信息并输出到 a.txt 文件中。

```
<target name="sqlselect">
    <sql driver="com.mysql.jdbc.Driver" url="jdbc:mysql://127.0.0.1:3306/
        test?characterEncoding=utf-8"
        userid="root" password="888"
        output="a.txt" print="true" encoding="utf-8">
        select * from student_inf;
    </sql>
</target>
```

运行 ant sqlselect 命令，可以在 a.txt 中查看到获取的信息，如图 3.7 所示。

图 3.7 执行 SQL 语句后获取的信息

3.6 深入 Ant

在第 2 章中，用手工方法构建项目，这里用 Ant 工具重构项目，以帮助读者加深对 Ant 的理解。

3.6.1　创建目录

用 Ant 的＜mkdir＞标签创建项目目录，创建步骤如下：

(1) 定义目录结构名称：

```
<property name="src" value="src" />
<property name="dest" value="target/classes" />
<property name="lib" value="lib" />
<property name="meta-inf" value="META-INF" />
```

(2) 定义创建目录任务 init：

```
<target name="init">
    <mkdir dir="${dest}" />
    <mkdir dir="${src}" />
    <mkdir dir="${lib}" />
    <mkdir dir="${meta-inf}" />
</target>
```

(3) 在命令行运行 ant init 命令，创建项目目录。

3.6.2　清除项目

为了能够反复编译，这里先定义清除任务，清除编译目标目录。利用 Ant 的＜delete＞标签清除编译的 jar 包，任务定义如下：

```
<target name="clean">
    <delete dir="${dest}" />
    <delete file="${appjar}" />
</target>
```

3.6.3　编译项目

编译项目操作步骤如下：

(1) 编译项目需要第三方依赖库，这里先定义 classpath 属性：

```
<property name="classpath" value="lib/lunar-1.2.jar" />
```

(2) 定义编译任务：

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

(3) 在命令行运行 ant compile 命令，编译项目。这比手工编译效率提高很多，直接用 javac 编译时，需要编译源文件清单，这里只需要源文件目录。

(4) 加上-verbose 和-debug 选项运行,显示执行细节,执行命令如下:

```
ant -verbose -debug clean compile
```

先执行 clean 清除,然后执行 compile 重新编译,如图 3.8 所示。

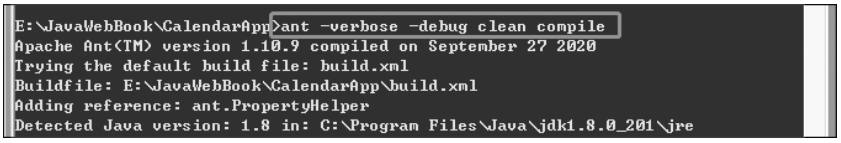

图 3.8 在命令行执行编译命令

(5) 命令执行分析。

编译命令执行的细节如图 3.9 所示。可以看出,Ant 首先扫描源目录 src 的 Java 源文件,把要编译的源文件找出来(ATest.java、LunarApp.java、Test.java)。javac 不能编译一个目录,这样 Ant 间接实现了编译一个目录。

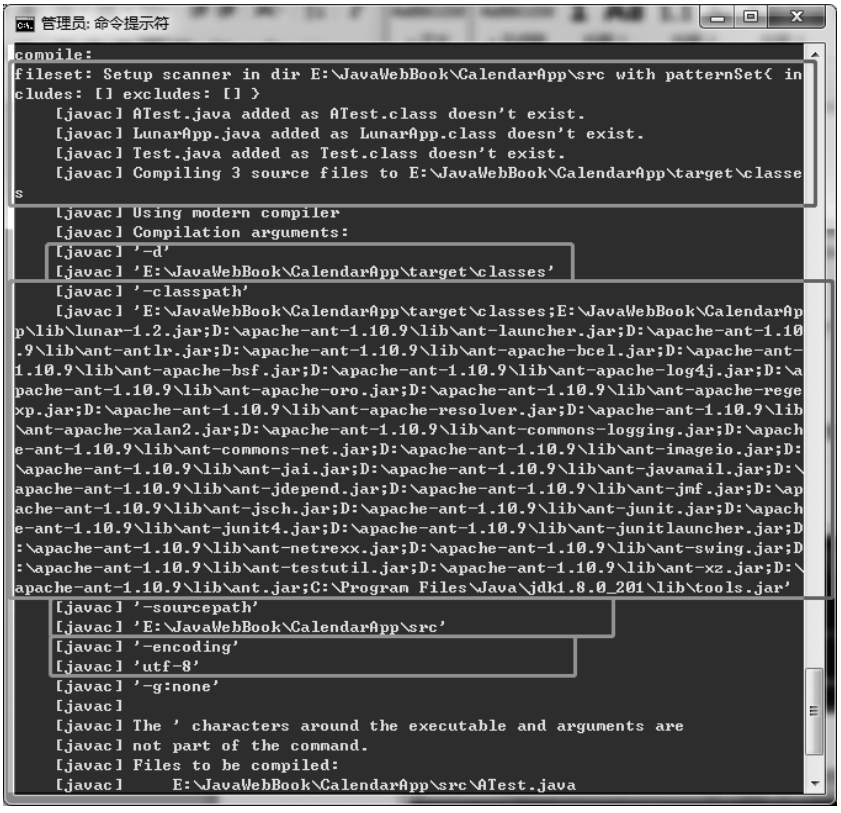

图 3.9 编译命令执行细节

图 3.9 中显示了编译参数,可以看出,这些编译参数正是 javac 的编译参数(-d、-classpath、-sourcepath、-encodeing),也就是说 Ant 编译时就是调用 javac,它封装了 javac,使编译更方便。

-d 指定编译输出目录，参数如下：

```
[javac] '-d'
[javac] 'E:\JavaWebBook\CalendarApp\target\classes'
```

- classpath 指定编译的类路径，参数如下：

```
[javac] '-classpath'
[javac] 'E:\JavaWebBook\CalendarApp\target\classes;E:\JavaWebBook\CalendarAp
p\lib\lunar-1.2.jar;D:\apache-ant-1.10.9\lib\ant-launcher.jar…
```

类路径包括编译输出目录 E:\JavaWebBook\CalendarApp\target\classes 和第三方依赖库 E:\JavaWebBook\CalendarApp\lib\lunar-1.2.jar，还包括 Ant 的依赖库。

（6）includeantruntime 属性分析。

其实编译不需要 Ant 依赖库，编译命令有个参数用于设置是否包括 Ant 依赖库，即 includeantruntime。在上面将其设置为 on，因此，把 Ant 依赖库也加入了 classpath。新增编译任务 compile1，把 includeantruntime 设置为 off，定义如下：

```
<target name="compile1" depends="init">
    <javac srcdir="${src}" destdir="${dest}" encoding="utf-8"
        includeantruntime="off" classpath="${classpath}">
    </javac>
</target>
```

在命令行运行 compile1 任务，命令如下：

```
ant -verbose -debug clean compile1
```

结果如图 3.10 所示，可以看出 classpath 中已经不包括 Ant 依赖库了。

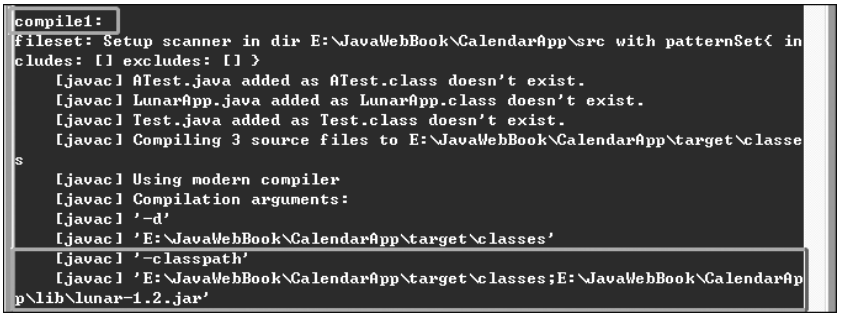

图 3.10　includeantruntime="off"时的 classpath

3.6.4　classpath 构建

用 javac 编译项目，在指定 classpath 时，需要一个一个地指定依赖库，不能指定一个目录，当库多时，比较麻烦。Ant 可以自动生成 classpath 需要的依赖库列表。

定义<path>标签，通过<fileset>元素指定路径，获得该路径下的多个文件。下面

的<path>定义将把 lib 目录中的 jar 包构造为一个文件名字符串，文件名之间的分隔符为分号。

```
<path id="classpathid">
    <fileset dir="${lib}">
        <include name="**/*.jar" />
    </fileset>
</path>
```

定义新的编译任务 compile2，利用上面定义的 path(refid="classpathid")，任务定义如下：

```
<target name="compile2" depends="init">
    <javac srcdir="${src}" destdir="${dest}" encoding="utf-8"
           includeantruntime="off">
        <classpath refid="classpathid" />
    </javac>
</target>
```

为了演示多个库文件，在 lib 目录下再复制一个 fastjson-1.2.18.jar，这个库没有用处，只是用于演示。在命令行运行 ant -verbose -debug clean compile2 命令，编译项目，这样在编译时就不用一个一个指定 jar 包。编译结果如图 3.11 所示，可以看出 classpath 自动加了 lib 目录下两个 jar 包。

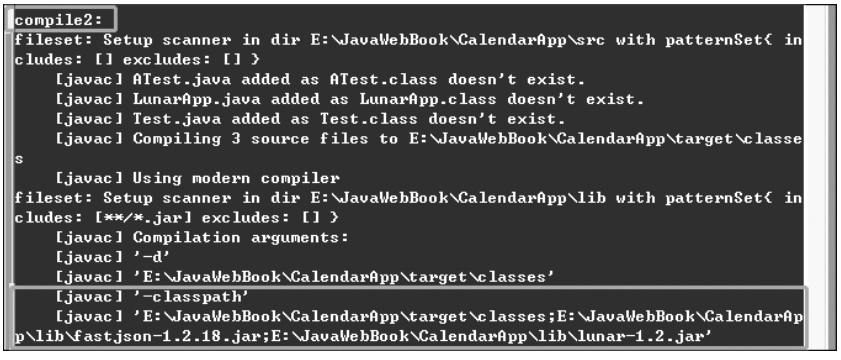

图 3.11　用<path>构建 classpath 的编译结果

3.6.5　打包项目

用 JDK 自带的 jar 命令打包可以实现整个目录打包，但不具备对目录中的文件进行筛选的能力，打包不够灵活。Ant 打包更加灵活，可以有 includes excludes 等过滤参数。这里只打包 classes 目录，任务定义如下：

```
<target name="build" depends="compile2">
    <jar jarfile="${appjar}" basedir="${dest}" />
</target>
```

在命令行中用调试方法运行打包任务 build，命令如下：

```
ant -verbose -debug clean build
```

打包结果如图 3.12 所示。可以看出，打包时扫描打包目录并且进行筛选(includes 和 excludes)，从而获取打包的文件集合。另外 Ant 打包并没有调用 JDK 的 jar 命令，而是自身实现的打包功能，jar 文件格式以流行的 zip 文件格式为基础，Ant 就是利用 ZIP 工具进行打包的。

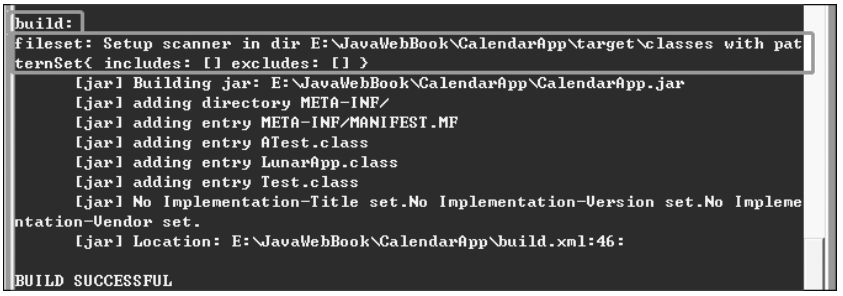

图 3.12　打包结果

3.6.6　运行项目

打包文件 CalendarApp.jar 不能单独运行，它运行时需要依赖库。在＜java＞标签中配置 classpath，通过设置＜pathelement location＝" ${appjar}"添加生成的 jar 包，还通过＜fileset dir＝" ${lib}"＞添加 lib 中的第三方依赖库。运行需要指明主类，通过 classname＝" ${main-class}"指定。创建一个新的进程运行 fork＝'true'。具体 run 任务配置如下：

```
<target name="run" depends="build">
    <java classname="${main-class}" fork='true'>
        <classpath>
            <pathelement location="${appjar}" />
            <fileset dir="${lib}">
                <include name="**/*.jar" />
            </fileset>
        </classpath>
    </java>
</target>
```

在命令行的调试方式下运行 ant -debug -verbose clean run 命令，运行结果如图 3.13 所示，可以看出，run 任务就是调用 JDK 的 java 命令运行的，结果中给出了参数：

```
[java] '-classpath'
[java] 'E:\JavaWebBook\CalendarApp\CalendarApp.jar;E:\JavaWebBook\CalendarA
pp\lib\fastjson-1.2.18.jar;E:\JavaWebBook\CalendarApp\lib\lunar-1.2.jar'
[java] 'ATest'
```

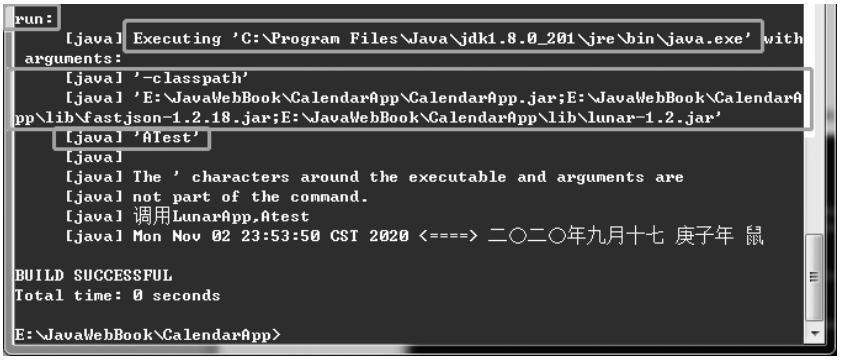

图 3.13 项目运行结果

3.6.7 打包可执行的 jar(依赖外部)

对于可执行的 jar 包,设置清单文件的 Class-Path 属性,指定外部的依赖库。这里希望自动添加库文件名,利用 pathconvert 路径转换,把前面生成的路径 classpathid 中的文件都去掉路径名称,然后都加上前缀./lib,变成相对路径,库文件名用空格分隔(pathsep=" ")。

(1) 路径转换命令如下:

```
<pathconvert property="quote.classpath" pathsep=" ">
    <mapper>
        <chainedmapper>
            <!-- jar 包文件只保留文件名,去掉目录信息 -->
            <flattenmapper />
            <!-- add lib/ prefix -->
            <globmapper from="*" to="./lib/*" />
        </chainedmapper>
    </mapper>
    <path refid="classpathid" />
</pathconvert>
```

这样就得到符合要求的 Class-Path 值,把这个值添加到清单文件中:

```
<attribute name="Class-Path" value="${quote.classpath}" />
```

(2) 定义打包任务 build1,任务定义如下:

```
<target name="build1" depends="compile2">
    <!-- 指定时间戳可以调用 TODAY -->
    <tstamp>
        <format property="TODAY" pattern="yyyy-MM-dd HH:mm:ss" />
    </tstamp>
    <pathconvert property="quote.classpath" pathsep=" ">
        <mapper>
            <chainedmapper>
```

```
            <!-- jar 包文件只保留文件名,去掉目录信息 -->
            <flattenmapper />
            <!-- add lib/ prefix -->
            <globmapper from="*" to="./lib/*" />
        </chainedmapper>
    </mapper>
    <path refid="classpathid" />
</pathconvert>
<!-- 生成清单文件 -->
<manifest file="${meta.dir}/MANIFEST.BAK" mode="replace">
    <attribute name="Built-By" value="${user.name}" />
    <attribute name="Built-Date" value="${TODAY}" />
    <attribute name="Main-Class" value="${main-class}" />
    <attribute name="Class-Path" value="${quote.classpath}" />
</manifest>
<jar jarfile="${appjar1}" basedir="${dest}" manifest="${meta.dir}/
    MANIFEST.BAK">
</jar>
</target>
```

(3) 定义运行任务 run1,运行 CalendarApp1.jar 包,任务定义如下:

```
<target name="run1" depends="build1">
    <java jar="${appjar1}" fork='true'>

    </java>
</target>
```

运行 ant clean run1 命令,运行结果如图 3.14 所示,run1 任务能够正确运行。CalendarApp1.jar 包的结构如图 3.15 所示。

```
build1:
     [jar] Building jar: E:\JavaWebBook\CalendarApp\CalendarApp1.jar

run1:
     [java] 调用LunarApp.Atest
     [java] Tue Nov 03 08:41:56 CST 2020 <====> 二〇二〇年九月十八 庚子年 鼠

BUILD SUCCESSFUL
Total time: 0 seconds

E:\JavaWebBook\CalendarApp>
```

图 3.14　运行 run1 任务的结果

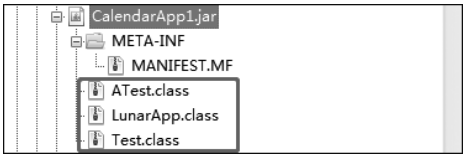

图 3.15　CalendarApp1.jar 包的结构

3.6.8 打包可执行的 jar(独立运行)

依赖包可以用<zipfileset>解压方式打入 jar 包,这样可执行的 jar 包就不再需要外部依赖库。这种方式需要在清单文件中添加主类。打包之前先生成清单文件,在清单文件中设置 Main-Class。

(1) 定义打包任务 build2,任务定义如下:

```
<target name="build2" depends="compile2">
    <!-- 指定时间戳可以调用 TODAY -->
    <tstamp>
        <format property="TODAY" pattern="yyyy-MM-dd HH:mm:ss" />
    </tstamp>
    <!-- 生成清单文件 -->
    <manifest file="${meta.dir}/MANIFEST.BAK" mode="replace">
        <attribute name="Built-By" value="${user.name}" />
        <attribute name="Built-Date" value="${TODAY}" />
        <attribute name="Main-Class" value="${main-class}" />
    </manifest>
    <jar jarfile="${appjar2}" basedir="${dest}" manifest="${meta.dir}/
          MANIFEST.BAK">
        <zipfileset src="${lib}/lunar-1.2.jar">
        </zipfileset>
    </jar>
</target>
```

(2) 定义任务 run2,执行 CalendarApp2.jar 包,任务定义如下:

```
<target name="run2" depends="build2">
    <java jar="${appjar2}" fork='true'>
    </java>
</target>
```

在命令行方式下运行 ant clean run2 命令,结果如图 3.16 所示,说明 build2 打包正确。CalendarApp2.jar 包的结构如图 3.17 所示,jar 包中打入的第三方依赖库已经解压。

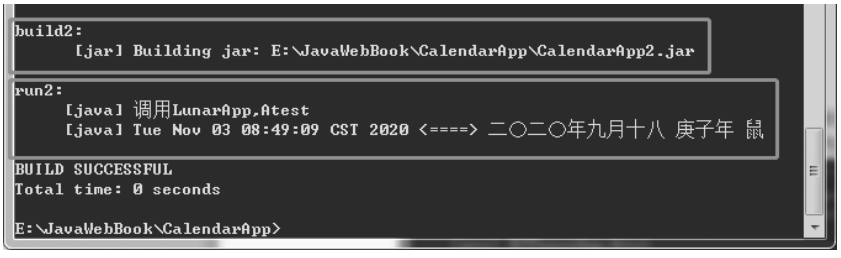

图 3.16 运行 run2 任务的结果

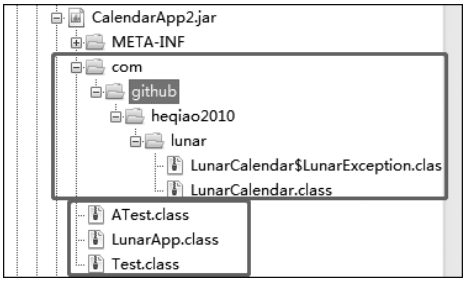

图 3.17　CalendarApp2.jar 包结构