Лекция 2

Основы Apache Maven

Maven- это инструмент для сборки Java проекта: компиляции, создания jar, создания дистрибутива программы, генерации документации. Простые проекты можно собрать в командной строке. Если собирать большие проекты с командной строки, то команда для сборки будет очень длинной, поэтому её иногда записывают в bat/sh скрипт. Но такие скрипты зависят от платформы. Для того чтобы избавиться от этой зависимостии и упростить написание скрипта используют инструменты для сборки проекта.

Для платформы Java существуют два основных инструмента для сборки: Ant и Maven.

Основные преимущества Maven

- **Независимость от OS.** Сборка проекта происходит в любой операционной системе. Файл проекта один и тот же.
- **Управление зависимостями.** Редко какие проекты пишутся без использования сторонних библиотек(зависимостей). Эти сторонние библиотеки зачастую тоже в свою очередь используют библиотеки разных версий. Мавен позволяет управлять такими сложными зависимостями. Что позволяет разрешать конфликты версий и в случае необходимости легко переходить на новые версии библиотек.
- **Возможна сборка из командной строки.** Такое часто необходимо для автоматической сборки проекта на сервере (Continuous Integration).
- **Хорошая интеграция со средами разработки.** Основные среды разработки на java легко открывают проекты которые собираются с помощью maven. При этом зачастую проект настраивать не нужно - он сразу готов к дальнейшей разработке.
- Как следствие - если с проектом работают в разных средах разработки, то maven удобный способ хранения настроек. Настроечный файл среды разработки и для сборки один и тот же - меньше дублирования данных и соответственно ошибок.
- **Декларативное описание проекта.**

Создадим новый проект, выполнив команду:

mvn archetype:generate

Выполнив эту команду maven покажет список шаблонов(архетипов) для разных проектов. Выберите проект и его версию по умолчанию, нажав Enter, Enter далее команда спросит, *groupId* и *artifactId* введите данные:

- *uits.jv1502*
- *MavenTest*

В результате выполнения команды сгенерируется проект со стандартной структурой директорий

```
MavenTest
 |-- pom.xml
 `-- src
     |-- main
     |   `-- java
     |       `--uits
     |           `-- jv1502
     |                   `-- App.java
     `-- test
         `-- java
             `--uits
                 `-- jv1502
                         `-- AppTest.java
```

Чтобы скомпилировать, нужно перейти в директорию проекта MavenTest и набрать в консоли

mvn compile

Если в консоль выведется

...
[INFO] BUILD SUCCESS
.....

то компиляция прошла успешно и в созданной директории target/classes будут class файлы с нашей программой.

Если вы наберёте mvn package, в директории target будет создан jar файл **MavenTest-1.0-SNAPSHOT.jar**

Для запуска программы наберем в консоли следующую команду:

java -cp ./target/classes uits.jv1502.App

В результате успешного выполнения программы мы получим следующую строку:

Hello World!

Что такое pom.xml

pom.xml - это основной файл, который описывает проект. Вообще могут быть дополнительные файлы, но они играют второстепенную роль.

Давайте резберём из чего состоит файл pom.xml

Корневой элемент и заголовок.

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
                    http://maven.apache.org/xsd/maven-4.0.0.xsd">
<modelVersion>4.0.0</modelVersion>
.....
</project>
```

Корневой элемент <project>, схема, которая облегчает редактирование и проверку, и версия POM.

Внутри тэга project содержится основная и обязательная информация о проекте:

```
<!-- The Basics -->
 <groupId>...</groupId>
 <artifactId>...</artifactId>
 <version>...</version>
```

В Maven каждый проект идентифицируется парой groupId artifactId. Во избежание конфликта имён, groupId - наименование организации или подразделения и обычно действуют такие же правила как и при именовании пакетов в Java - записывают доменное имя организации или сайта проекта. artifactId - название проекта. Внутри тэга version, как можно догадаться хранится версия проекта. Тройкой groupId, artifactId, version (далее - GAV) можно однозначно идентифицировать jar файл приложения или библиотеки. Если состояние кода для проекта не зафиксировано, то в конце к имени версии добавляется "-SNAPSHOT" что обозначает что версия в разработке и результирующий jar файл может меняться. <packaging>...</packaging> определяет какого типа файл будет создаваться как результат сборки. Возможные варианты pom, jar, war, ear.

Рассмотрим на примере проекта powermock-core **groupId** - org.powermock, **artifactId** - powermock-core , **version** - 1.4.6

Также добавляется информация, которая не используется самим мавеном, но нужна для программиста, чтобы понять, о чём этот проект:

- <name>powermock-core</name> название проекта для человека
- <description>PowerMock core functionality.</description> Описание проекта
- <url>http://www.powermock.org</url> сайт проекта.

Зависимости

Зависимости - следующая очень важная часть pom.xml - тут хранится список всех библиотек (зависимостей) которые используюся в проекте. Каждая библиотека идентифицируется также как и сам проект - тройкой groupId, artifactId, version (GAV). Объявление зависимостей заключено в тэг <dependencies>...</dependencies>.

```
<dependencies>
        <dependency>
            <groupId>junit</groupId>
            <artifactId>junit</artifactId>
            <version>4.4</version>
            <scope>test</scope>
        </dependency>

                ……..

        <dependency>
            <groupId>org.javassist</groupId>
            <artifactId>javassist</artifactId>
            <version>3.13.0-GA</version>
            <scope>compile</scope>
        </dependency>
    </dependencies>
```

Как вы могли заметить, кроме GAV при описании зависимости может присутствовать тэг <scope>. Scope задаёт для чего библиотека используется. В данном примере говорится, что библиотека с GAV junit:junit:4.4 нужна только для выполнения тестов.

Тэг <build>

Тэг <build> не обязательный, т. к. существуют значения по умолчанию. Этот раздел содержит информацию по самой сборке: где находятся исходные файлы, где ресурсы, какие плагины используются. Например:

```
<build>
        <outputDirectory>target2</outputDirectory>
         <finalName>ROOT</finalName>
        <sourceDirectory>src/java</sourceDirectory>
          <resources>
             <resource>
                <directory>${basedir}/src/java</directory>
                <includes>
                <include>**/*.properties</include>
                </includes>
             </resource>
          </resources>
              <plugins>
                 <plugin>
                    <groupId>org.apache.maven.plugins</groupId>
                    <artifactId>maven-pmd-plugin</artifactId>
                    <version>2.4</version>
                 </plugin>
              </plugins>
           </build>
```

Рассмотрим этот пример более подробно.

- <sourceDirectory> - определяет, откуда maven будет брать файлы исходного кода. По умолчанию это src/main/java, но вы можете определить, где это вам удобно. Директория может быть только одна (без использования специальных плагинов)
- <resources> - и вложенные в неё тэги <resource> определяют, одну или несколько директорий, где хранятся файлы ресурсов. Ресурсы в отличии от файлов исходного кода при сборке просто копируются . Директория по умолчанию src/main/resources
- <outputDirectory> - определяет, в какую директорию компилятор будет сохранять результаты компиляции - *.class файлы. Значение по умолчанию - target/classes
- <finalName> - имя результирующего jar (war, ear..) файла с соответствующим типу расширением, который создаётся на фазе package. Значение по умолчанию - artifactId-version.

Maven плагины позволяют задать дополнительные действия, которые будут выполняться при сборке. Например в приведённом примере добавлен плагин, который автоматически делает проверку кода на наличие "плохого" кода и потенциальных ошибок.

Репозитории.

Репозитории - это место где хранятся артефакты: jar файлы, pom -файлы, javadoc, исходники.

Существуют:

- Локальный репозиторий по умолчанию он расположен в <home директория>/.m2/repository - персональный для каждого пользователя.
- центральный репозиторий который расположен в http://repo1.maven.org/maven2/ и доступен на чтение для всех пользователей в интернете.
- Внутренний "Корпоративный" репозиторий- дополнительный репозиторий, один на несколько пользователей.

Локальный репозиторий

Локальный репозиторий по умолчанию он расположен в <home директория>/.m2/repository. Здесь лежат артефакты которые были скачаны из центрального репозитория либо добавлены другим способом. Например если вы наберёте команду

```
mvn install
```

в текущем проекте, то соберётся jar (или war, pom в зависимости от содержимого тэга packaging) который установится в локальный репозиторий. Найти его можно в <home директория>/.m2/repository/<groupIdPath>/<artifactId>/<version>/<artifactId>-<version>.jar где groupIdPath получается заменой всех точек на слеши. Например для проекта

```
<groupId>uits.jv1502</groupId>
 <artifactId>site</artifactId>
 <version>1.0-SNAPSHOT</version>
```

jar файл будет лежать по пути: <home директория>/.m2/repository/uits/jv1502/site/1.0-

Центральный репозиторий

Чтобы самому каждый раз не создавать репозиторий, сообщество для Вас поддерживает центральный репозиторий. Если для сборки вашего проекта не хватает зависимостей, то они по умолчанию автоматически скачиваются с http://repo1.maven.org/maven2. В этом репозитории лежат практически все опенсорсные фреймворки и библиотеки.

Самому в центральный репозиторий положить нельзя. Т.к. этот репозиторий используют все, то перед тем как туда попадают артефакты они проверяются, тем более что если артефакт однажды попал в репозиторий, то по правилам изменить его нельзя.

Для поиска нужной библиотеки очень удобно пользоваться сайтами http://mavenrepository.com/ и http://findjar.com/

Корпоративный репозиторий

Если вы хотите создать свой репозиторий, содержимое которого вы можете полностью контролировать(как локальный), и сделать так, чтобы он был доступен для нескольких человек, вам будет полезен корпоративный репозиторий. Доступ к артефактам можно ограничивать настройками безопасности сервера так, что код ваших проектов не будет доступен из вне.

Чтобы добавить репозиторий в список, откуда будут скачиваться зависимости, нужно добавить секцию repositories в pom.xml, например:

```
<project>
  ...
  <repositories>
    <repository>
      <id>my-company-repo</id>
      <url>http://my-company-site.ru/repo</url>
    </repository>
  </repositories>
  ...
 </project>
```

Жизненный цикл сборки

Основные фазы сборки проекта

1. compile компилирование проекта
2. test тестирование с помощью JUnit тестов
3. package создание .jar файла или war, ear в зависимости от типа проекта
4. integration-test запуск интеграционных тестов
5. install копирование .jar (war , ear) в локальный репозиторий
6. deploy публикация файла в удалённый репозиторий

К примеру нам нужно создать jar проекта. Чтобы его создать набираем:

mvn package

Но перед созданием jar-файла будут выполняться все предыдущие фазы compile и test , а фазы integration-test, install, deploy не выполнятся. Если набрать

mvn deploy

то выполнятся все приведённые выше фазы.

Особняком стоят фазы **clean** и **site**. Они не выполняются если специально не указаны в строке запуска.

- **clean** удаление всех созданных в процессе сборки артефактов: .class, .jar и др. файлов. В простейшем случае результат — просто удаление каталога target
- **site** предназначена для создания документации (javadoc+сайт описания проекта).

Т. к . команда mvn понимает когда ему передают несколько фаз то для сборки проекта создания документации "с нуля" выполняют:

```
mvn clean package site
```

Профайлы

Основные сведения

Мавен изначально создавался , принимая во внимание портируемость. Но довольно часто приложение приходится запускать в разном окружении: например, для разработки используется одна база данных, в рабочем сервере используется другая. при этом могут понадобиться разные настройки, разные зависимости и плагины. Для этих целей в maven используются профайлы.

Давайте определим два профайла: один для разработки, другой для производственного сервера. Для разработки вполне подойдёт база hsqldb, которая хранит все данные в памяти. На производственном сервере же используется база данных postgres, которая сохраняет все данные на диск. В профайлах для каждой конфигурации определены свои проперти database.url и зависимости для разных jdbc драйверов.

Ниже приведён пример объявления таких профайлов.

```
<?xml version="1.0" encoding="UTF-8"?>
<project>
  <profiles>
(1)     <profile>
          <id>development</id>
          <properties>
             <database.url>jdbc:hsqldb:mem:testdb</database.url>
          </properties>
          <dependencies>
             <dependency>
                <groupId>org.hsqldb</groupId>
                <artifactId>hsqldb</artifactId>
                <version>2.0.0</version>
             </dependency>
          </dependencies>
       </profile>
```

(2)
```
    <profile>
      <id>productionServer</id>
      <properties>
        <database.url>jdbc:postgresql://databseserver/database</database.url>
      </properties>
      <dependencies>
        <dependency>
          <groupId>postgresql</groupId>
          <artifactId>postgresql</artifactId>
          <version>9.0-801.jdbc4</version>
        </dependency>
      </dependencies>
    </profile>
  </profiles>
  .....
</project>
```

Цифрами 1 и 2 обозначены начала объявления профайлов. каждый профайл имеет идентификатор в данном случае development и productionServer.

Внутри тэга <profile> содержатся все те же объявления что и внутри <project>: properties, dependencies, и др. Вот полный список тегов которые могут содержаться внутри профайлов:

- <repositories>
- <pluginRepositories>
- <dependencies>
- <plugins>
- <properties>
- <modules>
- <reporting>
- <dependencyManagement>
- <distributionManagement>
- <build> тэг, который может содержать
    1. <defaultGoal>
    2. <resources>
    3. <testResources>
    4. <finalName>

Активация профайла

Чтобы содержимое тэга профайла "работало", нужно профайл активировать. Когда профайл активирован, его содержимое объединяется с основной частью pom.xml. Нужно заметить, что активных профилей одновременно может быть несколько.

Активировать профайл можно несколькими способами:

во первых, это можно задать вручную в командной строке запуска maven, например: mvn package -P production

Во вторых при объявлении самого профайла можно задать тэг <activation>, который определяет профайл будет активирован: в нашем примере профайл development активный по умолчанию: <activation><activeByDefault>true</activeByDefault></activation>. Кроме активации по умолчанию можно задать активацию на основе операционной системы, установленных переменных окружения, версии JDK.

В коммандной строке можно задать, какие профили будут деактивированы: mvn goal -P !profile-1,!profile-2 //приоритет командной строки выше

Активные профайлы можно также задать в ~/.m2/settings.xml

```
<settings>
        ...
        <profiles>
          <profile>
            <id>appserverConfig</id>
            <properties>
              <appserver.home>/path/to/appserver</appserver.home>
            </properties>
          </profile>
        </profiles>

        <activeProfiles>
          <activeProfile>appserverConfig</activeProfile>
        </activeProfiles>
        ...
</settings>
```

Отладка

Чтобы проверить работу и, возможно, найти ошибки, полезны следующие плагины:

- для того, чтобы показать какие профайлы сейчас активны, можно набрать:
  mvn help:active-profiles

причём опции командной строки принимаются во внимание.

- команда

mvn help:active-profiles -P productionServer,development

выведет, как и положено

The following profiles are active:

- development (source: pom)

- productionServer (source: pom)

- Также можно посмотреть pom, полученный после объединения основной части и активных профайлов:
  mvn help:effective-pom -P productionServer

Сравнение Maven и Ant

Интеграция со средами разработки

Во-первых, в отличии от Ant, Maven хорошо интегрируется со всеми основными средами разработки. Если в вашей команде предпочитают работать в разных средах разработки, то не нужно ограничивать - пусть каждый пользуется тем, что ему нравится. В любой среде проект открывается сразу уже настроенный.

Во вторых, хорошо, если сборка проекта может происходить совсем без IDE. Для сборки Ant'ом это значит поддерживать скрипты для сборки, которые дублируют информацию в файлах проекта конкретной IDE. Зачастую build.xml (файл сборки в Ant) устаревает, и перестаёт работать т.к. им каждодневно не пользуются.

Управление зависимостями в Ant

Во всех Ant проектах в которых я участвовал работа с библиотеками организована следующим образом: в директории проекта создавалась папка lib и туда копировались все jar файлы библиотек.

Проблем нет до тех пор пока вся библиотека содержится в одном jar файле. Но дело в том что сложные библиотеки могут включать в свой состав другие библиотеки(зависимости). К примеру hibernate содержит внутри себя 16 библиотек. Вот управлять такими библиотеками с зависимостями сложнее.

Представьте что в вашем проекте давно используется Hibernate и Struts. Тут вы решили обновить версию Hibernate чтобы воспользоваться новыми возможностями новой версии. Кажется всё просто - сравниваем версии hibernate, копируем в директорию lib все библиотеки из нового hibernate. Если в lib есть такие же файлы библиотек со старыми номерами в названии файла то пожалуй их нужно удалить. Всё вроде просто, но ранние версии hibernate содержали в качестве зависимости commons-logging, а в новой версии вместо него используется slf4j. Получается commons-logging уже не нужен, можно удалить. А вы точно уверены что можно удалить? А если commons-logging успользуется в другой библиотеке? И действительно, внутри struts есть commons-logging. удалять нельзя - иначе получишь ClassNotFoundException во время выполнения программы. Тут я вам описал случай с двумя библиотеками Hibernate и Struts. А если библиотек много? Пожалуй тут лучше воспользоваться ivy..

Управление зависимостями в Maven

Maven хранит зависимости в pom.xml. И в отличии от Ant'овской системы сборки* здесь информация о зависимостях не теряется. В большинстве случаев апгрейд библиотеки сводится к изменению номера версии в pom.xml. Всё остальное Maven сделает сам.

Здесь приведён часто встречающийся способ сборки проекта в Ant, но он может значительно отличаться.

Недостатки maven

К недостаткам maven следует отнести его большую сложность и дополнительное время для изучения если вы ещё не знаете Maven.

Есть опенсорсные библиотеки, которые собираются не Maven'ом, и они попадают в центральный репозиторий обычно позже, чем выйдет официальный релиз. Тут можно либо подождать, когда они всё-таки попадут в центральный репозиторий, либо добавить их вручную влокальный/корпоративный репозиторий.