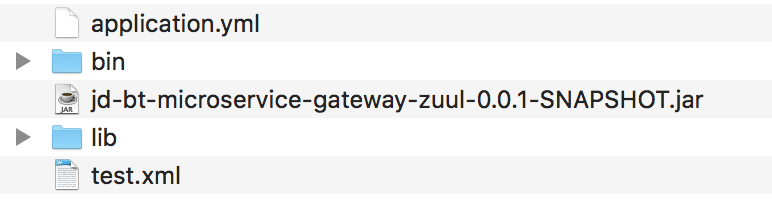
一、需求

　　打一个zip包，包含如下：



　　bin为程序脚本，启动和停止

　　lib为依赖包

　　根目录下为配置文件和项目jar包

二、知识储备

2.1、插件了解

| **plugin** | **function** |
| --- | --- |
| maven-jar-plugin | maven 默认打包插件，用来创建 project jar，负责将应用程序打包成可执行的jar文件  可在此处设置主类，manifest，排除对应的配置文件等 |
| maven-shade-plugin | 用来打可执行包，executable(fat) jar |
| maven-assembly-plugin | 支持定制化打包方式，负责将整个项目按照自定义的目录结构打成最终的压缩包，方便实际部署  可在此处设置打包拷贝路径，配置，以及打包好的jar文件等 |

三、配置

1、pom配置：

[IMG_257](javascript:void(0);)

<plugin>

<groupId>org.apache.maven.plugins</groupId>

<artifactId>maven-jar-plugin</artifactId>

<version>2.6</version>

<configuration>

<archive>

<manifest>

<mainClass>com.jd.bt.ZuulApplication</mainClass>

<!-- to create a class path to your dependecies you have to fill true in this field -->

<addClasspath>true</addClasspath>

<classpathPrefix>lib/</classpathPrefix>

</manifest>

<manifestEntries>

<Class-Path>./</Class-Path>

</manifestEntries>

</archive>

<excludes>

<!--注意从编译结果目录开始算目录结构-->

<exclude>/\*.yml</exclude>

<exclude>/\*.properties</exclude>

<exclude>/\*.xml</exclude>

</excludes>

</configuration>

</plugin>

<plugin>

<artifactId>maven-assembly-plugin</artifactId>

<configuration>

<!-- not append assembly id in release file name -->

<appendAssemblyId>false</appendAssemblyId>

<descriptors>

<descriptor>src/main/assembly/depolyment.xml</descriptor>

</descriptors>

</configuration>

<executions>

<execution>

<id>dist</id>

<phase>package</phase>

<goals>

<goal>single</goal>

</goals>

</execution>

</executions>

</plugin>

[IMG_258](javascript:void(0);)

对于maven-jar-plugin配置

　　其中manifest的部分是核心，在可执行的jar文件中，打包后会在jar文件内的META-INF文件夹下，生成一个MANIFEST.MF文件，里面记录了可执行文件的一些相关配置，比如像上面一段代码中所配置的内容，这里面就配置了可执行jar文件未来读取classpath的相对目录位置在什么地方，以及引入的jar文件都有哪些，上面的配置就是classpath目录是./   
　　mainClass配置表示，哪个class作为程序的入口来执行   
　　addClasspath配置表示，是否将依赖的classpath一起打包   
　　classpathPrefix配置表示，依赖的classpath的前缀，也就是打包后生成的MANIFEST.MF文件里，引入的jar文件都会加上前缀，lib/，比如fastjson-1.2.7.jar，在mainfest文件里就会是lib/fastjson-1.2.7.jar   
　　excludes配置表示，排除哪些文件夹不被打包进去

　　注意：其实maven-jar-plugin主要就是配置了MANIFEST.MF这个文件而已，就是让可执行文件知道自己怎么执行，加载哪些文件执行的描述，剩下的工作交由maven-assembly-plugin来处理

对于maven-assembly-plugin配置

　　地址：http://maven.apache.org/plugins/maven-assembly-plugin/assembly.html#class\_unpackOptions

　　　　http://liugang594.iteye.com/blog/2093607

　　　　https://blog.csdn.net/u011955252/article/details/78927175

　　最终压缩的文件格式，为zip，fileSets中配置了需要将那些文件打包到最终压缩包中，   
　　　　如配置文件包括了启动脚本bin文件夹，里面放着shell的启动脚本，相关的配置文件src/main/resources，里面放着整个程序提取的properties等相关的配置文件   
　　　　　最终可运行的jar文件，使用了${project.build.directory}变量，也就是通过maven-jar-plugin生成的那个jar文件   
　　dependencySets里面配置了依赖库最终输出到lib文件夹下，与上面的maven-jar-plugin配置生成的manifest文件路径相对应，这样可运行jar就会按照manifest的路径来找相应的文件进行加载

　　参看地址：https://blog.csdn.net/senpogml/article/details/52366518?locationNum=12

2、src/main/assembly/depolyment.xml配置

[IMG_259](javascript:void(0);)

<?xml version="1.0" encoding="UTF-8"?><assembly xmlns="http://maven.apache.org/plugins/maven-assembly-plugin/assembly/1.1.3" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="http://maven.apache.org/plugins/maven-assembly-plugin/assembly/1.1.3 http://maven.apache.org/xsd/assembly-1.1.3.xsd">

<id>dist</id>

<formats>

<format>zip</format>

</formats>

<includeBaseDirectory>true</includeBaseDirectory>

<fileSets>

<fileSet>

<directory>src/main/bin</directory>

<outputDirectory>bin/</outputDirectory>

</fileSet>

<fileSet>

<directory>src/main/resources</directory>

<outputDirectory>/</outputDirectory>

</fileSet>

<fileSet>

<directory>${project.build.directory}</directory>

<outputDirectory>/</outputDirectory>

<includes>

<include>\*.jar</include>

</includes>

</fileSet>

</fileSets>

<dependencySets>

<dependencySet>

<outputDirectory>lib</outputDirectory>

<scope>runtime</scope>

<excludes>

<exclude>${groupId}:${artifactId}</exclude>

</excludes>

</dependencySet>

</dependencySets></assembly>

[IMG_260](javascript:void(0);)

3、start.sh和stop.sh

start.sh

[IMG_261](javascript:void(0);)

#!/bin/bash

ARTIFACT\_ID=jd-bt-microservice-gateway-zuul

VERSION=0.0.1-SNAPSHOT

main\_jar=${ARTIFACT\_ID}-${VERSION}.jar

PIDS=`ps aux|grep ${main\_jar} |grep -v "grep" |awk '{print $2}'`if [ ! -z "$PIDS" ]; then

echo "${ARTIFACT\_ID} already started!"

echo "PID: $PIDS"

exit 0;fi

BIN\_PATH="${JAVA\_HOME}/bin"

LOG\_PATH="/Users/lihongxu/log/${ARTIFACT\_ID}/"

mkdir -p $LOG\_PATH

echo "ready start ${main\_jar}";

nohup $BIN\_PATH/java -server -Xms4096m -Xmx4096m -jar ../${main\_jar} >$LOG\_PATH/nohup.log 2>&1 &

sleep 5

PIDS=`ps aux|grep ${main\_jar} |grep -v "grep" |awk '{print $2}'`

if [ ! -z "$PIDS" ]; then

echo " ${ARTIFACT\_ID} Started Successed,pid:${PIDS}"

exit 0;else

echo " ${ARTIFACT\_ID} Started Failed"

exit 1;fi

[IMG_262](javascript:void(0);)

stop.sh

[IMG_263](javascript:void(0);)

#!/bin/bash

ARTIFACT\_ID=jd-bt-microservice-gateway-zuul

VERSION=0.0.1-SNAPSHOT

main\_jar=${ARTIFACT\_ID}-${VERSION}.jar

PIDS=`ps aux|grep ${main\_jar} |grep -v "grep" |awk '{print $2}'`

if [ ! -z "$PIDS" ]; then

kill -9 "$PIDS"

echo " ${ARTIFACT\_ID} is stop !"

echo " PID: $PIDS"

exit 0;fi

[IMG_264](javascript:void(0);)

四、java启动

java -cp 和 -classpath 一样，是指定类运行所依赖其他类的路径，通常是类库，jar包之类，需要全路径到jar包，window上分号“;”，linux使用“:”  
4.1、格式：  
　　java -cp .;myClass.jar packname.mainclassname      
　　表达式支持通配符，例如：  
　　java -cp .;c:\classes01\myClass.jar;c:\classes02\\*.jar  packname.mainclassname   
4.2、运行jar  
　　java -jar myClass.jar  
　　执行该命令时，会用到目录META-INF\MANIFEST.MF文件，在该文件中，有一个叫Main－Class的参数，它说明了java -jar命令执行的类。  
4.3、运行jar和cp  
　　用maven导出的包中，如果没有在pom文件中将依赖包打进去，是没有依赖包。  
　　　　1.打包时指定了主类，可以直接用java -jar xxx.jar。  
　　　　2.打包是没有指定主类，可以用java -cp xxx.jar 主类名称（绝对路径）。  
　　　　3.要引用其他的jar包，可以用java -classpath $CLASSPATH:xxxx.jar 主类名称（绝对路径）。其中 -classpath 指定需要引入的类。  
4.4、实例  
　　下面基于pom和META-INF\MANIFEST.MF两个文件的配置，进行了三种情况的测试：

　　pom.xml的build配置：

IMG_265

[IMG_266](javascript:void(0);)

<build>

<!--<finalName>test-1.0-SNAPSHOT</finalName>-->

<plugins>

<plugin>

<groupId>org.apache.maven.plugins</groupId>

<artifactId>maven-compiler-plugin</artifactId>

<version>3.1</version>

<configuration>

<source>1.8</source>

<target>1.8</target>

</configuration>

</plugin>

<plugin>

<artifactId>maven-assembly-plugin</artifactId>

<configuration>

<archive>

<manifest>

<mainClass>test.core.Core</mainClass>

</manifest>

</archive>

<descriptorRefs>

<descriptorRef>jar-with-dependencies</descriptorRef>

</descriptorRefs>

</configuration>

<!--下面是为了使用 mvn package命令，如果不加则使用mvn assembly-->

<executions>

<execution>

<id>make-assemble</id>

<phase>package</phase>

<goals>

<goal>single</goal>

</goals>

</execution>

</executions>

</plugin>

</plugins>

</build>

[IMG_267](javascript:void(0);)

META-INF\MANIFEST.MF的内容：  
Manifest-Version: 1.0  
Main-Class: test.core.Core

1.pom中build指定mainClass 但是 META-INF\MANIFEST.MF文件中没有指定Main-Class: test.core.Core  
java -jar test-jar-with-dependencies.jar //执行成功  
java -cp test-jar-with-dependencies.jar  test.core.Core  //执行失败，提示jar中没有主清单属性  
  
2.pom中build没有指定mainClass 但是 META-INF\MANIFEST.MF文件中指定了Main-Class: test.core.Core  
java -jar test-jar-with-dependencies.jar //执行失败，提示jar中没有主清单属性  
java -cp test-jar-with-dependencies.jar  test.core.Core  //执行成功  
  
3.pom中build指定mainClass && META-INF\MANIFEST.MF文件中增加了Main-Class: test.core.Core  
java -cp test-jar-with-dependencies.jar  test.core.Core  //执行成功  
java -jar test-jar-with-dependencies.jar  //执行成功