Java Web 开发内部培训(I)

张 亮 编撰

作者主页：https://github.com/pepstack

本文代码：git@github.com:pepstack/java-training.git

```
# /etc/hosts
192.30.255.112     github.com     git
185.31.16.184      github.global.ssl.fastly.net
```

（本资料适用于初级和中级 java 程序开发人员）

2019 年 9 月

1 环境准备

1.1 Linux

1) jdk-8u152-linux-x64.tar.gz

配置 /etc/profile.d/java-env.sh:

```
export JAVA_HOME=/usr/local/java/jdk1.8.0_152
export JRE_HOME=$JAVA_HOME/jre
export CLASSPATH=.:$JAVA_HOME/lib:$JAVA_HOME/lib/dt.jar:$JAVA_HOME/lib/tools.jar:$JRE_HOME/lib
export PATH=$PATH:$JAVA_HOME/bin:$JAVA_HOME/jre/bin
```

2) apache-maven-3.3.3

配置 /etc/profile.d/maven-env.sh:

```
export MAVEN_HOME=/usr/local/java/apache-maven-3.3.3
export PATH=$PATH:$MAVEN_HOME/bin
```

参考

- jdk 一键安装脚本：

  https://github.com/pepstack/common-tools/blob/master/scripts/install_jdk.sh

- Ubuntu 下安装 JDK 开发环境：

  https://blog.csdn.net/ubuntu64fan/article/details/6993637

- Java HelloWorld：

  https://blog.csdn.net/weixin_43149737/article/details/82724894

1.2 Windows

1) cygwin64

安装：

https://www.cygwin.com/setup-x86_64.exe

cygwin 多 tab 窗口页面：

https://blog.csdn.net/ubuntu64fan/article/details/80520699

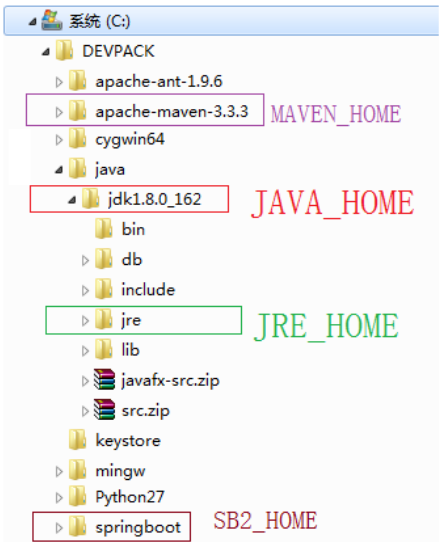
2） jdk-8u162-windows-x64.exe

3） apache-maven-3.3.3

Windows 系统环境变量：

JAVA_TOOL_OPTIONS =-Dfile.encoding=UTF8

MAVEN_OPTS=-Xms256m -Xmx512m -Dfile.encoding=UTF-8



CLASSPATH=.;%JAVA_HOME%\lib\dt.jar;%JAVA_HOME%\lib\tools.jar

Path=...;%JAVA_HOME%\bin;%JRE_HOME%\bin;%ANT_ROOT%;%MAVEN_HOME%\bin;%SB2_HOME%\bin

1.3 实验

实验 1

在自己常用的电脑上搭建出 jdk 运行时和开发环境，总结环境搭建过程中遇到的各种问题和解决手段。熟悉几种常用的 java 开发 IDE 环境：eclipse，vscode，npp。

提示：打开命令行，输入 java、javac、java -version、mvn --version 测试 java 和 maven 是否安装成功！

实验 2

在生产系统上，java 通常部署在 Linux 服务器上。选择一种 Linux 服务器（如：rhel/centos7，ubuntu18.04LTS）,分别用手工和自动化脚本 2 种方式部署 java 环境。熟悉 java，javac 命令。

2 Java 基础

2.1 基本概念

- **JVM**：Java Virtual Machine（Java 虚拟机）是通过在实际的计算机上仿真模拟各种计算机功能虚构出来的计算机（虚拟机）。引入 JVM 后，Java 语言在不同平台上运行时不需要重新编译。Java 语言使用 JVM 屏蔽了与具体平台相关的信息，使得 Java 语言编译程序只需生成在 JVM 上运行的目标代码（字节码），就可以在多种平台上不加修改地运行。

- **JDK**：Java Development Kit，java 的开发和运行环境，java 的开发工具和 jre。

- **JRE**：Java Runtime Environment，java 程序的运行环境，java 运行的所需的类库+JVM(java 虚拟机)。

- **JAVA_HOME** 环境变量：jdk 的主目录。

- **CLASSPATH** 环境变量：由系统帮我们去找指定的目录载入系统的包文件。

- **PATH** 环境变量：指定命令搜索路径，让$JAVA_HOME\bin 目录下的工具，可以在任意目录下运行。

- **javac**：负责的是编译的部分，当执行 javac 时，会启动 java 的编译器程序。对指定扩展名的.java 文件进行编译。生成了 Jvm 可以识别的字节码文件。也就是.class 文件，也就是 java 的库或运行程序。

- **java**：负责运行的部分。会启动 jvm。加载运行时所需的类库,并对 class 文件进行执行。

- **main** 函数：一个文件要被执行,必须要有一个执行的起始点,这个起始点就是 main 函数——主函数。不被执行的类不需要主函数。

2.2 语法知识

1）类和构造函数

类（class）：模版。

对象（object）：根据模版产生的实例—new—类实例化（创建对象）。

构造函数：对象初始化函数。函数的名称 = 类的名称。无返回值。对象创建时必须初始化。

对象封装：指隐藏对象的属性和实现细节，仅对外提供公共访问方式。将不需要对外提供的内容都隐藏起来，把属性都隐藏，提供公共方法对其访问。

```
public class Person {
    static {
        System.out.println("静态代码块");
    }
    {
        System.out.println("构造代码块");
    }

    public Person() {
        System.out.println("我是一个无参构造方法");
    }

    public Person(String name) {
        System.out.println("我是带有一个参数的构造方法");
    }
}
```

执行优先级从高到低：静态代码块 > main 方法 > 构造代码块 > 构造方法。其中静态代码块只执行一次。构造代码块在每次创建对象是都会执行。

创建一个对象都在内存中做了什么事情？

```
Person p = new Person();
```

1. 先将硬盘上指定位置的 Person.class 文件加载进内存。

2. 执行 main 方法时，在栈内存中开辟了 main 方法的空间(压栈-进栈)，然后在 main 方法的栈区分配了一个变量 p。

3. 在堆内存中开辟一个实体空间，分配了一个内存首地址值。new

4. 在该实体空间中进行属性的空间分配，并进行了默认初始化。

5. 对空间中的属性进行显示初始化。

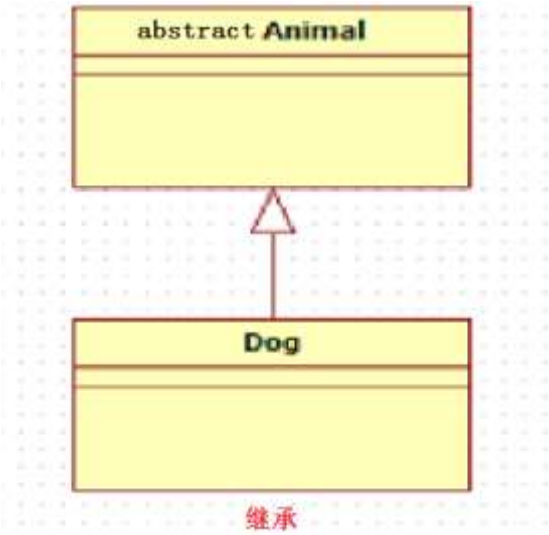
6. 进行实体的构造代码块初始化。

7. 调用该实体对应的构造函数，进行构造函数初始化。（）

8. 将首地址赋值给 p ，p 变量就引用了该实体。(指向了该对象)

2）类继承 extends

java 只支持单继承。java 虽然不直接支持多继承，但是可实现多接口。



如果想要调用父类中的属性值，需要使用一个关键字：super。

this：代表是本类类型的对象引用。

super：代表是子类所属的父类中的内存空间引用。

子类的实例化过程：子类在进行对象初始化时，先调用父类的构造函数。子类的所有构造函数中的第一行，其实都有一条隐身的语句 super()。调用父类中空参数的构造函数。

注意：

- 子类中所有的构造函数都会默认访问父类中的空参数的构造函数，因为每一个子类构造内第一行都有默认的语句 super()。

- 如果父类中没有空参数的构造函数，那么子类的构造函数内，必须通过 super 语句指定要访问的父类中的构造函数。
- 如果子类构造函数中用 this 来指定调用子类自己的构造函数，那么被调用的构造函数也一样会访问父类中的构造函数。

禁止继承：final class。

3）抽象类 abstract

抽象类特点：

- 包含抽象方法（只定义方法声明，并不定义方法实现）的类，只能定义在抽象类中，抽象类和抽象方法必须由 abstract 关键字修饰（没有抽象变量）。
- 抽象类不能实例化（创建对象）。
- 子类继承抽象类并实现了抽象类中的所有抽象方法后，该子类才可以实例化。否则，该子类还是一个抽象类。

抽象类的细节：

- 抽象类中是否有构造函数？有，用于给子类对象进行初始化。
- 抽象类中是否可以定义非抽象方法？ 可以。其实，抽象类和一般类没有太大的区别，都是在描述事物，只不过抽象类在描述事物时，有些功能不具体。所以抽象类和一般类在定义上，都是需要定义属性和行为的。只不过，比一般类多了抽象方法。
- 抽象关键字 abstract 和哪些不可以共存？

  final，private，static

- 抽象类中可不可以不定义抽象方法？可以。抽象方法目的仅仅为了不让该类实例化。

4）接口类 interface

```
interface DisplayInterface {
```

```
    public static final int dpi = 3;
    public abstract void show();
}
```

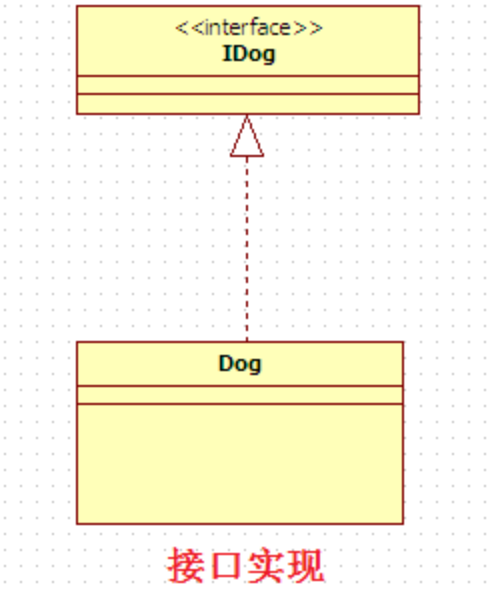
- 接口中可以包含全局常量、抽象方法：
  - 成员变量：public static final
  - 成员方法：public abstract
- 接口继承：接口不可以实例化。接口的子类必须实现（implements）了接口中所有的抽象方法后，该子类才可以实例化（否则，该子类还是一个抽象类）。
- 接口和类不同的地方：接口可以被多实现。java 将多继承机制通过多实现来体现（子类继承多个接口）。一个类在继承另一个类的同时，还可以实现多个接口。所以接口的出现避免了单继承的局限性。还可以将类进行功能的扩展。
- java 类的继承：
  - java 类是单继承的：class Child extends class Parent
  - java 接口可以多继承：class ImplClass implements InterfaceA, InterfaceB, ……
  - java 不允许类多重继承的主要原因是：如果 A 同时继承 B 和 C，而 B 和 C 同时有一个 Do 方法，A 如何决定该继承那一个呢？但接口不存在这样的问题，接口全都是抽象方法。
- 抽象类与接口：

抽象类：用于描述一个体系单元，将一组共性内容进行抽取。可以在类中定义抽象内容让子类实现，可以定义非抽象内容让子类直接使用。它里面定义的都是一些体系中的基本内容。是类对象实现上的统一和抽象。

接口：用于定义（或描述）实现对象应具备的一组功能（函数集）。是对象互操作性的一致性描述。

抽象类和接口的共性：都是不断向上抽取的结果。

- 抽象类和接口的区别：
  - 抽象类只能被继承，而且只能单继承。接口需要被实现，而且可以多实现。
  - 抽象类中可以定义非抽象方法，子类可以直接继承使用。接口中都是抽象方法，需要子类去实现。
  - 抽象类使用的是 is A 关系。接口使用的 like A 关系。
  - 抽象类的成员修饰符可以自定义。接口中的成员修饰符是固定的。全都是 public 的。



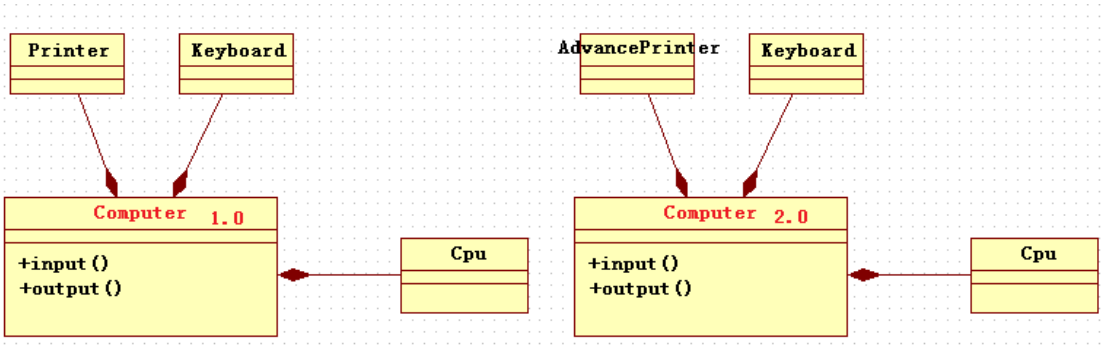
5）Java 中的面向接口编程

- 面向接口编程：是很多软件架构设计理论都倡导的编程方式。接口体现的是一种规范和实现分离的设计哲学，充分利用接口可以极好地降低程序各模块之间的耦合，从而提高系统的可扩展性和可维护性。基于这种原则，通常推荐“面向接口”编程，而不是面向实现类编程。
- 面向接口编程的优势：

（一）工厂模式

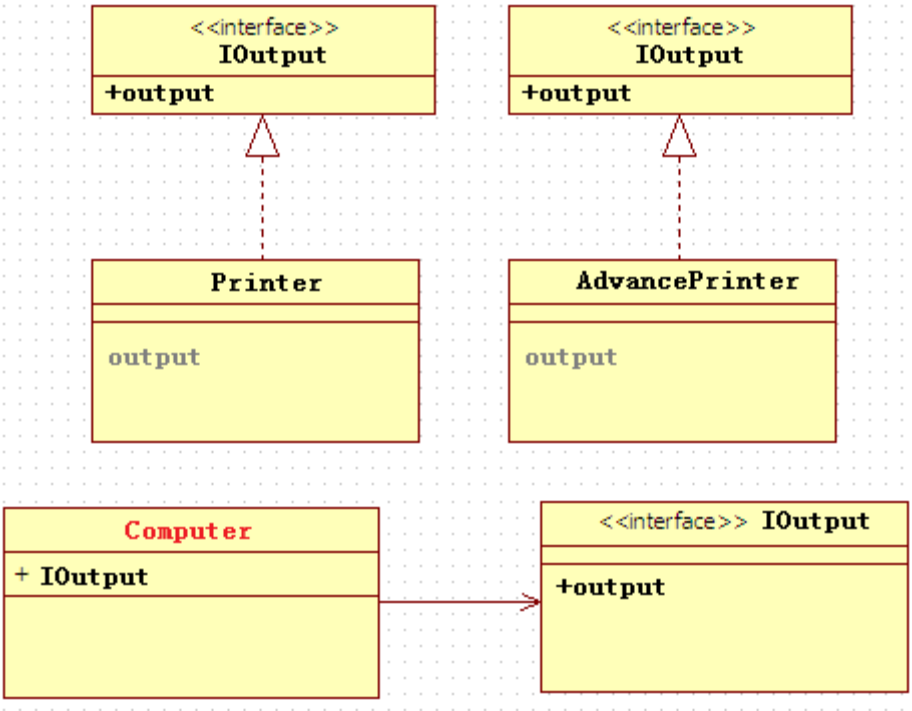
假设程序中有个 Comupter 类需要组合一个输出设备（output），现在有两个选择：直接让 Comupter 该类组合 Printer（Printer 具有 output 设备），或者让 Comupter 组合 Output 属性，那么到底采用哪种方式更好呢？

假设让 Computer 组合 Printer 属性，如果将来需要使用 AdvancePrinter 来代替 Printer，需要对 Computer 类源代码进行修改。如果系统中有更多的类组合了 Printer 属性,将意味着我们要修改每个要升级 Printer 到 AdvancePrinter 的类代码。



Computer 组合了 Printer的功能

为了避免这个问题，我们让 Comupter 组合一个 Output 属性，将 Comupter 类与 Printer 类完全分离。Computer 对象实际组合的是 Printer 对象，还是 AdvancePrinter 对象，对 Computer 而言完全透明。当 Printer 对象切换到 AdvancePrinter 对象时，系统完全不受影响。



Comupter类与Printer类完全分离

```
/**
 * file: IOutput.java
 */
package code.training;

public interface IOutput {
    void output(String content);
}


/**
 * file: Printer.java
 */
public class Printer implements IOutput {
    public void output(String content) {
        System.out.println("Printer:" + content);
    }
}
```

```java
/**
 * file: AdvancePrinter.java
 */
package code.training;

public class AdvancePrinter implements IOutput {
    public void output(String content) {
        System.out.println("AdvancePrinter:" + content);
    }
}


/**
 * file: OutputFactory.java
 */
package code.training;

public class OutputFactory {
    // Printer printer;
    private final AdvancePrinter adprinter;

    public static IOutput createOutput() {
        // 1.0
        // return new Printer;

        // 2.0
        return new AdvancePrinter();
    }
}


/**
 * file: Computer.java
 */
package code.training;

public class Computer {
    private final IOutput output;

    public Computer(IOutput output) {
        this.output = output;
    }

    public IOutput getOutput() {
```

```
        return output;
    }
}
```

（二）命令模式

某个方法需要完成某一个行为，但这个行为的具体实现无法确定，必须等到执行该方法时才可以确定。例如：假设有个方法需要遍历某个数组的数组元素，但无法确定在遍历数组元素时如何处理这些元素，需要在调用该方法时指定具体的处理行为。类似 C 语言的回调函数。

对于这样一个需求，我们必须把“处理行为”作为参数传入该方法，实现 process 方法和“处理行为”的分离。这个“处理行为”用编程来实现就是一段代码。那如何把这段代码传入该方法呢？

```
public interface Command {
    // 接口里定义的 process 方法用于封装“处理行为”
    void process(int[] target);
}


public class ProcessArray
{
    public void process(int[] target, Command cmd) {
        cmd.process(target);
    }
}


public class PrintCommand implements Command {
    public void process(int[] target) {
        for (int tmp : target ) {
            System.out.println("迭代输出目标数组的元素:" + tmp);
        }
    }
}


public class AddCommand implements Command {
    public void process(int[] target) {
```

```
            int  sum  =  0;
            for  (int  tmp  :  target) {
                sum  +=  tmp;
            }
            System.out.println("数组元素的总和是:"  +  sum);
        }
    }
```

主程序：

```
    public  class  TestApp {
        public  static  void  main(String[]  args)
        {
            ProcessArray  pa  =  new  ProcessArray();
            int[]  target  =  {3,  -4,  6,  4};

            // 第一次处理数组，具体处理行为取决于 PrintCommand
            pa.process(target  ,  new  PrintCommand());

            // 第二次处理数组，具体处理行为取决于 AddCommand
            pa.process(target  ,  new  AddCommand());
        }
    }
```

6）Java 单例设计模式 -- Singleton

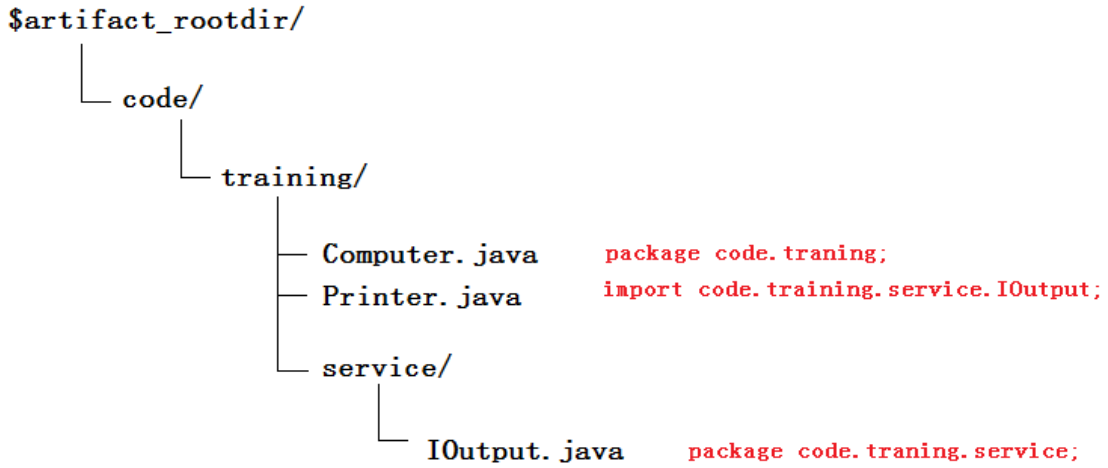
Singleton 是一种创建性模型,它用来确保在进程范围内只产生一个实例,并提供一个访问它的全局访问点。应用该模式的类一个类只有一个实例。即一个类只有一个对象实例。

```
    public class SingletonProcess {
        private static enum Singleton {
            INSTANCE;

            private static final SingletonProcess singleton = new
    SingletonProcess();

            public SingletonProcess getSingleton() {
                return singleton;
            }
        }
```

```
    private SingletonProcess() {
    }

    public static SingletonProcess getInstance() {
        return SingletonProcess.Singleton.INSTANCE.getSingleton();
    }

    public void startupService(String arg) {
        System.out.println("SingletonProcess startupService: " + arg);
    }

    /**
     * main app entry
     */
    public static void main(String args[]) throws Exception {
        SingletonProcess.getInstance().startupService(args[0]);
    }
}
```

2.3 Java 程序代码结构

1）Java 项目目录和包



2）关于 JAVA 的包访问权限

JAVA 提供了许多级别的访问权限，从而可以精确的控制类、超类以及包中变量和成员的可见性，从而更好的实现封装、继承，多态。因为类和包的相互影响，

对于 JAVA 中的类成员（变量及方法），主要表现为五种类型的可见性：

（1）类本身。

（2）相同包中的子类。

（3）相同包中的非子类。

（4）不同包中的子类。

（5）不同包中的非子类。

而对于访问权限，JAVA 又提供了三个关键字以及默认权限作出对应处理，分别为 private,default(默认访问权限)，protected，public。对其进行如下简化理解：

（1）所有 public 的成员可以在任何地方进行访问。

（2）所有 private 的成员在当前类外部都为不可见。

（3）如果是默认访问权限，只有当前包下的类可见，其他包中子类也无法访问。

（3）所有 protected 的成员，在默认权限的基础上，其他包中子类也可以访问。

	private	默认访问权限	protected	public
类本身	是	是	是	是
相同包中子类	否	是	是	是
相同包中的非子类	否	是	是	是
不同包中的子类	否	否	是	是
不同包中的非子类子类	否	否	否	是

3）使用 maven 构建工具

https://www.w3cschool.cn/maven/rm471htd.html

项目构建，管理，jar 包下载，下载 source 并关联 source。例如下面的命令：

mvn compile： 下载 jar 并编译项目

mvn package： 打包工程项目

mvn test： 执行 test

mvn dependency：sources 尝试下载 source 文件

mvn eclipse：eclipse 生成 eclipse 的工程配置文件

◆ 创建一个 java 工程：

(使用 maven-archetype-quickstart 插件创建一个简单的 Java 应用)

```
$ cd java-training/
$ ./mvn-create.sh
    mvn archetype:generate
    -DgroupId=com.pepstack.code
    -DartifactId=outputlib
    -DarchetypeArtifactId=maven-archetype-quickstart
    -DinteractiveMode=false
$ cd outputlib/
$ mvn compile
```

◆ mvn 项目的目录结构：

```
package com.pepstack.code;

/**
 * App.java
 */
public class App
{
    public static void main( String[] args )
    {
        System.out.println( "Hello World!" );
    }
}
```

(outputlib-1)

◆ pom.xml 文件结构：

```
<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>

    <groupId>com.pepstack.code</groupId>

    <artifactId>outputlib</artifactId>
    <packaging>jar</packaging>
    <version>1.0-SNAPSHOT</version>
    <name>outputlib</name>
    <url>http://maven.apache.org</url>

    <properties>
        <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
        <main.class>com.pepstack.code.App</main.class>
    </properties>

    <dependencies>
        <dependency>
            <groupId>junit</groupId>
            <artifactId>junit</artifactId>
            <version>3.8.1</version>
            <scope>test</scope>
        </dependency>
```

```
        <dependency>
            ...
        </dependency>

    </dependencies>

    <build>
        <plugins>
            <plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-compiler-plugin</artifactId>
                <version>2.3.2</version>
                <configuration>
                    <source>1.6</source>
                    <target>1.6</target>
                    <encoding>UTF8</encoding>
                </configuration>
            </plugin>

            <plugin>
                ...
            </plugin>
        </plugins>
    </build>
</project>
```

maven 常见操作：

（1）https://mvnrepository.com/搜索需要的 jar;

（2）添加到 pom.xml 合适的位置;

（3）编译;

（4）测试;

maven 自动下载的 jar 在哪里?

（1）Windows

C:\Users\$username\.m2\repository\

（2）Linux

/home/$username/.m2/repository/

本地 maven repository 安装自己创建的 jar 或下载的 jar?
（参考：mvn-install-jar.sh）

```
# mvn install:install-file \
    -Dfile='"$jarfile"' \
    -DgroupId='"$groupId"' \
    -DartifactId='"$artifactId"' \
    -Dversion='"$version"' \
    -Dpackaging=jar
```

pom.xml 引用本地 maven repository 的 jar：

```
<!-- xstream:
    http://xstream.codehaus.org/
    dependencies:
        xmlpull-1.1.3.1.jar
        xpp3_min-1.1.4c.jar
-->
<dependency>
    <groupId>com.thoughtworks.xstream</groupId>
    <artifactId>xstream</artifactId>
    <version>1.4.7</version>
</dependency>

<dependency>
    <groupId>org.xmlpull</groupId>
    <artifactId>xmlpull</artifactId>
    <version>1.1.3.1</version>
</dependency>

<dependency>
    <groupId>org.xmlpull</groupId>
    <artifactId>xpp3_min</artifactId>
    <version>1.1.4c</version>
</dependency>
```

2.4 实验

实验 1

利用最简单的编辑器如：npp，将本章中的代码集成到一个可执行程序 computer.jar 中，并用 javac 命令编译，然后使用 java 命令运行。

实验 2

用 maven 创建 computer 工程，然后用 maven 命令编译和运行。

实验 3

将 Printer 类和 AdvancePrinter 类从主程序 computer.jar 中分离出来,创建一个自己的类库（output.jar），将 Printer 类集成到 output 类库中。实现 computer 打印功能。然后再实现升级版的 output.jar,实现升级的 computer.jar 打印功能。

通过本章实验使学生完全掌握简单的 java 程序和类库的创建方法，理解面向接口编程的思维方法，为以后熟练运用打下基础。了解 maven 基本命令，能创建可执行 jar，类库 jar，能安装类库 jar 到本地 maven 仓库。

参考

- Java 基础知识总结：
  https://www.cnblogs.com/BYRans/p/Java.html

- OOP——UML 六种关系：
  https://blog.csdn.net/weixin_34174132/article/details/85584600

3 SpringBoot 入门

Spring 提出了依赖注入的思想，即依赖类不由程序员实例化，而是通过 Spring 容器帮我们 new 指定实例并且将实例注入到需要该对象的类中。依赖注入的另一种说法是"控制反转"。通俗的理解是：平常我们 new 一个实例，这个实例的控制权是我们程序员。而控制反转是指 new 实例工作不由我们程序员来做而是交给 Spring 容器来做。控制反转的过程由 Spring 读取配置文件来自动完成。

3.1 技术背景

从 Servlet 技术到 Spring 和 Spring MVC，Java 开发 Web 应用变得越来越简捷。但是 Spring 和 Spring MVC 的众多配置有时即使是开发一个简单的 Web 应用，都需要我们在 pom 文件中导入各种依赖，编写 web.xml、spring.xml、springmvc.xml 配置文件等。特别是需要导入大量的 jar 包依赖时，需要在网上查找各种 jar 包资源，各个 jar 间可能存在着各种依赖关系，这时候又得下载其依赖的 jar 包，有时候 jar 包间还存在着严格的版本要求。所以常常开发一个简单的 Web 应用时，却把大部分的时间花在了编写配置文件和导入 jar 包依赖上。

为了简化 Spring 繁杂的配置，Spring Boot 应运而生。正如 Spring Boot 的名称一样，一键启动：Spring Boot 提供了开箱即用的自动配置功能，使我们将重心放在业务逻辑的开发上。

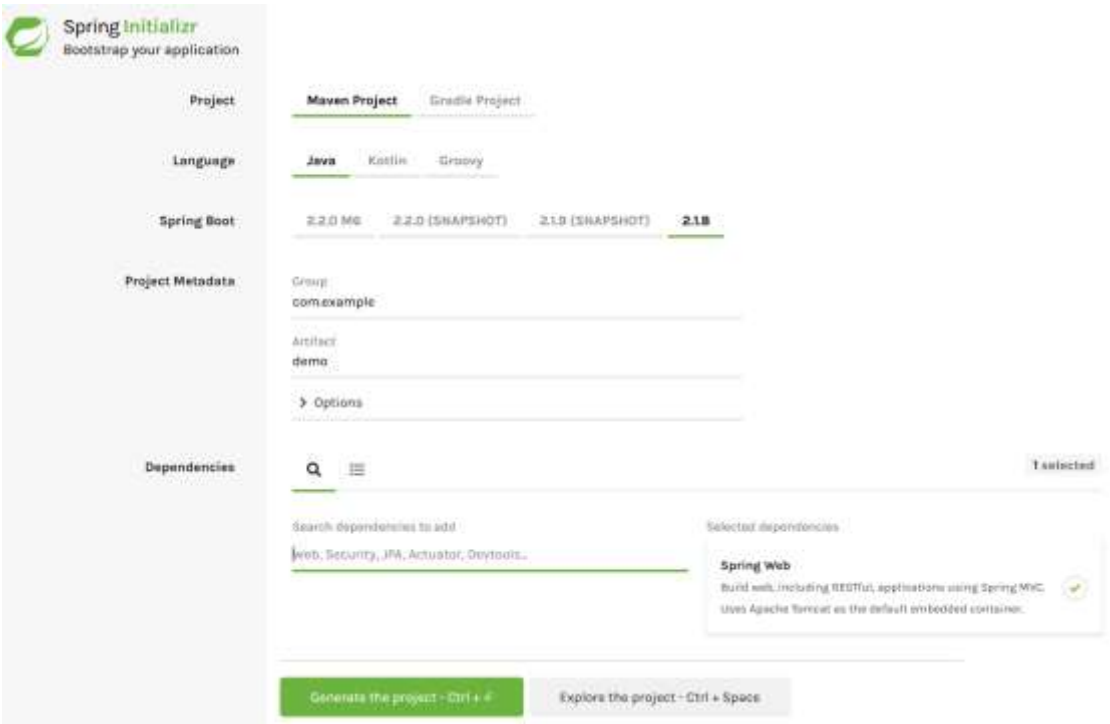
3.2 Spring Boot 特性

- 使用 Spring 项目引导页面可以在几秒构建一个项目（https://start.spring.io/）。
- 方便对外输出各种形式的服务，如 REST API、WebSocket、Web、Streaming、Tasks。
- 非常简洁的安全策略集成。
- 支持关系数据库和非关系数据库。
- 支持运行期内嵌容器，如 Tomcat、Jetty。

- 强大的开发包，支持热启动。
- 自动管理依赖。
- 自带应用监控。

3.3 创建一个简单的 Web 站点

1）进入：https://start.spring.io/，根据需要进行项目配置（配置就是自动生成 pom.xml 的过程，可以后期手工修改）

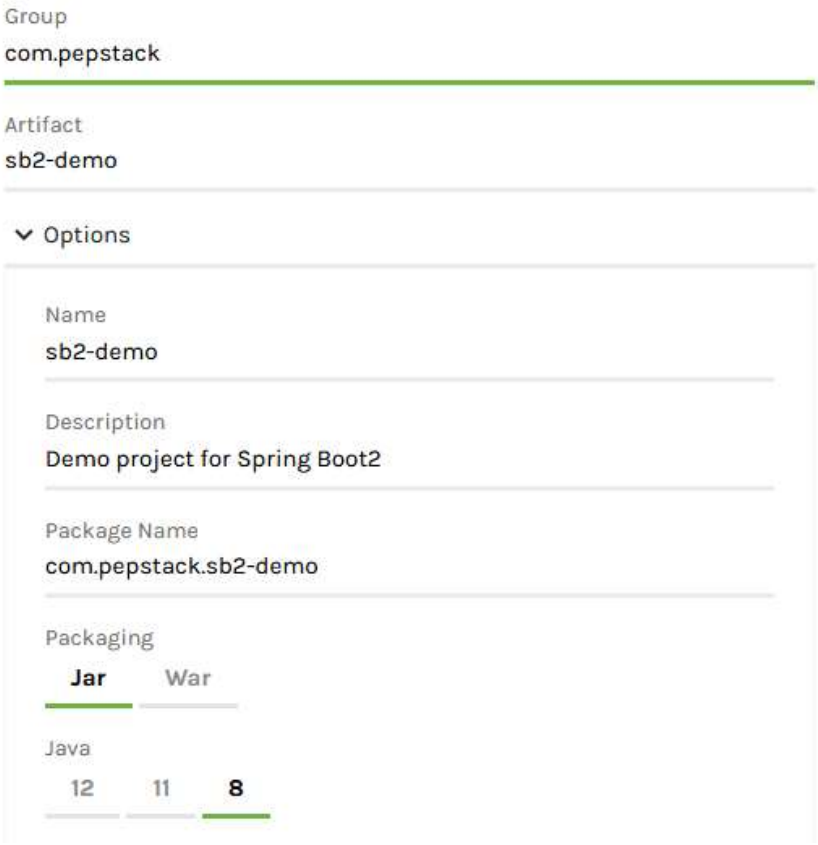


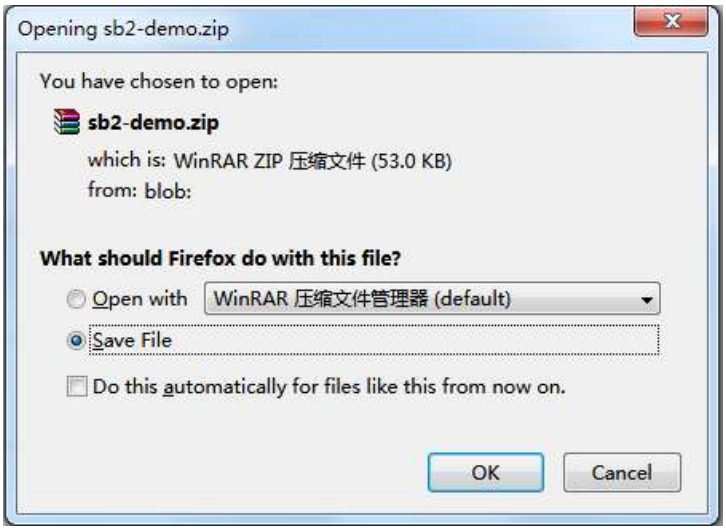
2）Generate the project，保存到本地：java-training/

注意：

Maven Project

Packaging: Jar
Java: 8





3）编译

```
$ cd sb2-demo/
$ mvn compile
```

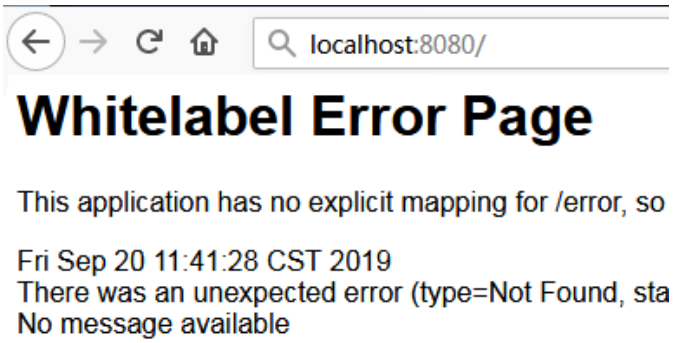
直接运行：

```
$ mvn spring-boot:run
```

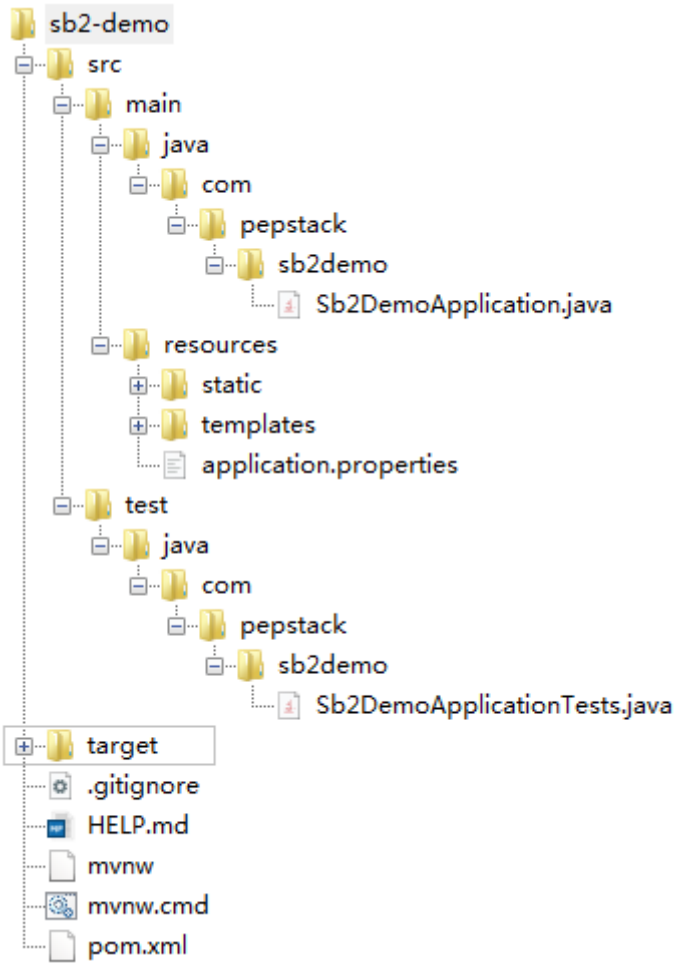
或者：

```
$ mvn package
$ java -jar target/sb2-demo-0.0.1-SNAPSHOT.jar
```

浏览器：http://localhost:8080/



4）浏览一下项目结构



以上我们并没做任何事情，但是已经具备了一个 Web 服务的外壳，接下来在此基础上真正要建立一个 Web 网站，或者 webapi。

3.4 真正开始创建一个 Web 网站

（参考：springboot2.0 使用 Thymeleaf 开发 web 项目简单示例）

Spring boot 开发 web 项目，通常打成 jar 包，使用内置的 web 服务器 Tomcat、Jetty、undertow 来运行。静态资源（css、js、图片等）默认放在 resources/static 下面。如果要修改默认存放目录，可以通过设置属性 spring.mvc.static-path-pattern 来实现。

模板文件默认放在 templates 目录下。Spring boot 支持使用模板来开发 web 应用，支持的模板类型包括：

- ✓ FreeMarker
- ✓ Groovy
- ✓ Thymeleaf
- ✓ Mustache

Spring boot 不建议使用 jsp 开发 web。本文使用 Thymeleaf 来作为模板引擎开发 web 项目。

Thymeleaf 是一个 Java 模板引擎开发库，可以处理和生成 HTML、XML、JavaScript、CSS 和文本，在 Web 和非 Web 环境下都可以正常工作。Thymeleaf 可以跟 Spring boot 很好地集成。

获取本节代码：

```
# git clone git@github.com:pepstack/java-training.git

# cd java-training/springboot2-webapp-sample01/
```

直接在 mvn 中运行：

```
# mvn spring-boot:run
```

浏览器访问：

```
http://localhost:8088/webapp/user/list
http://localhost:8088/webapp/user/2
```

3.5 SpringBoot Web 和自动生成脚本

按照约定大于配置的原则，制定基于 springboot2 的 web 项目的目录结构，可以使用自动化脚本自动创建 sb2web 项目的全部子目录。sb2web 项目依据的准则是：

- ✓ spring boot2
- ✓ maven 3.3+
- ✓ java 8
- ✓ thymeleaf 作为模板引擎
- ✓ restful 服务接口

https://blog.csdn.net/wangjianno2/article/details/51044780

https://blog.csdn.net/ubuntu64fan/article/details/80555372

https://blog.csdn.net/ubuntu64fan/article/details/80555915

```
$artifactRootDir/
       └─┬─ mvnw
         ├─ mvnw.cmd
         ├─ pom.xml
         └─ src /
             ├─ main/
             │    ├─ java/
             │    │    └─ com/
             │    │         └─ pepstack/
             │    │              └─ webapp/
             │    │                   ├─ controller/
             │    │                   │    └─ UserController.java
             │    │                   ├─ model/
             │    │                   │    └─ User.java
             │    │                   └─ WebappApplication.java
             │    └─ resources /
             │         ├─ application.properties
             │         ├─ static /
             │         └─ templates/
             │              └─ user/
             │                   ├─ detail.html
             │                   └─ list.html
             └─ test /
                  └─ java /
                       └─ com /
                            └─ pepstack/
                                 └─ webapp /
                                      └─ WebappApplicationTests.java
```