# 文档说明

本文档适用的读者为具有基本的编程语言基础而缺乏实践经验，或者毫无编程经验，希望能够成为Java开发人员的读者。文档内容偏重实践，理论基础等涉及较少，在需要理论的地方推荐读者阅读相应的书籍。

本文档不适用于 Java高级以上的开发人员阅读。

本文档的读者需熟悉 Windows 或 Ubuntu 2种操作系统中的一种，无需其他基础。Unit11 之后为高阶内容，对于学有余力的读者可自行学习，相关内容较为简略，目的在于开阔视野，每一部分都有相关的书籍进行进一步的学习。

# 开发环境准备

## 2.1 配置开发环境

### 2.1.1 安装 JDK

**（1）Windows 系统**

下载JDK （Java Development Kit，即 Java 开发组件）， 地址详见甲骨文(Oracle)官网 <https://www.oracle.com/java/technologies/downloads/> ， 选择对应的操作系统（Windows、MacOS、Linux）和硬件架构（x86, aarch）的安装包。解压后安装。

1. **Ubuntu（乌班图） 系统**

以Ubuntu 22.04 LTS 为例，可直接在控制台终端（CTRL+ALT+T）中运行

|  |
| --- |
| *sudo apt-get install openjdk-17-jdk* |

输入管理员密码，即可。

### 2.1.2 配置环境变量

环境变量的作用，就是当用户在操作系统的控制台（Windows的cmd）中，输入一个命令时，操作系统会按照环境变量中设置的文件目录，逐个去查找。

**（1）Windows 系统**

1）点击 “我的电脑”->"属性"->高级系统设置->环境变量->系统变量->新建

JAVA\_HOME=java安装路径，例如

|  |
| --- |
| JAVA\_HOME=C:\Program files\jdk22 |

2）在“我的电脑”->"属性"->高级系统设置->环境变量->系统变量-> 选择PATH ->编辑 ， 在PATH环境变量中添加一个value, %JAVA\_HOME%\bin。

注意，环境变量PATH的值中有多个文件路径，不要覆盖已有的文件路径，在当前值下新增一个文件路径即可。

1. 验证。按下 Windows键+R，输入cmd，确定，输入 java -version，验证环境，若输出如下内容，则表示安装成功。

|  |
| --- |
| *java version 22\*\*\*\* Java (TM) SE Runtime Environment* |

**（2）Ubuntu 系统**

通过 sudo命令安装的JDK,无需添加环境变量。在控制台（CTRL+ALT+T）运行 java -version。若输出如下内容，则表示安装成功。

|  |
| --- |
| *openjdk version "17.0.10" 2024-01-16 OpenJDK Runtime Environment (build 17.0.10+7-Ubuntu-122.04.1) OpenJDK 64-Bit Server VM (build 17.0.10+7-Ubuntu-122.04.1, mixed mode, sharing)* |

至此，java运行时已安装完毕。

### 2.1.3 扩展阅读

为什么要安装JDK，不安装可以进行Java开发吗？ Java属于高级语言，需要编译后生成一种叫字节码（Byte-code）的中间二进制流，通过Java 虚拟机（JVM, java vritual machine）的解释后，才能成为机器可是别的机器指令。JDK提供以下功能。

**（1）源代码编译**（%JAVA\_HOME/bin/javac）。将Java源代码编译为Java 字节码，注意是字节码，一种字节流，这种字节流不是机器指令，机器无法识别。

**（2）java程序运行时**（%JAVA\_HOME/bin/java）。提供java虚拟机，即可以运行java字节码的环境，通过java虚拟机的翻译，将java字节码翻译为机器硬件可是别的机器指令。

**（3）函数库**。提供常用的一些算法、数据结构、工具类的函数包，方便开发人员开箱即用，无需自己写源代码。

综合以上分析， （1）和（2）是必须的， （3）是可选的（如果开发人员想自己实现的话），可见JDK是必须安装的，否则无法进行java相关的的开发工作。

**（4）字节码**。字节码是一种跟硬件无关的二进制字节流，可以被 java 虚拟机识别并转换为对应的机器可执行的机器指令。之所以这么做，是为了减少对不同硬件体系结构及其对应的硬件指令集的依赖。同一套源代码，生成同一套字节码，由JVM去适配不同的硬件指令集。

## 2.2 安装集成开发环境

### 2.2.1 GUI IDE 安装

安装集成开发环境（IDE， Integrated Development Environment ），能够为开发人员提供源代码输入，代码补全，代码编译，运行时调试等便利的功能。本节以开源的Eclipse 为例， 下载地址 <https://www.eclipse.org/downloads/packages/> 。

集成开发环境相当于一个干某个工种工人的工具箱，把特定职业所需的工具全部放一起，方便快速高效地完成工作。

**（1）Windows**

下载 Windows x86-64 版本的 Eclipse IDE for Enterprise Java and Web Developers， <https://www.eclipse.org/downloads/download.php?file=/technology/epp/downloads/release/2024-06/R/eclipse-jee-2024-06-R-win32-x86_64.zip> 。

下载完成后，解压缩，双击解压后文件夹中的 eclipse.exe 即可直接使用。

**（2）Ubuntu**

在 Ubuntu系统中，打开（CTRL+ALT+T）终端（Terminal）执行以下命令

|  |
| --- |
| *sudo apt-get install eclipse* |

安装成功后, 在终端中执行

|  |
| --- |
| *eclipse &* |

即可打开图形化用户界面（GUI， Graphical User Interface）。

### 2.2.2 扩展阅读

是否可以不安装IDE,直接通过文本文档进行开发工作？ 事实上，如果只是进行一些小型的demo实验，或者复杂度较低的java 程序编写，可以不使用 IDE，直接打开一个 文本文档编辑器，写好源代码后，直接另存为 test.java（注意：文件编码格式需使用UTF-8）即可， 然后通过java相关的命令工具进行编译，运行。但在实际的工程实践中，一般来说，都得使用一款自己熟悉的 IDE, IDE提供以下功能。

**（1）源代码着色。**通过源代码着色，提高代码的可读性。

**（2）工程管理。**对工程中所编写的源代码依赖的其他工具包进行自动管理，无需手动处理。

**（3）源代码debug GUI。**能够提供源代码 debug 的GUI，更加方便地进行代码的分析和问题查找。

目前市面上的IDE 有 Eclipse, Netbeans, Intelli IDEA，其中 Eclipse和netbeans 均为免费产品， Intelli IDEA 为收费产品。

**（4）代码打包及版本管理。**一般IDE 可以通过插件等其他形式提供代码打包及版本管理的图形化界面。

# 3. Hello world

## 3.1 使用 IDE 创建工程

以 eclipse-jee-2024-06-R-linux-gtk-x86\_64 的 eclipse为例进行说明。打开Eclipse, 选择 文件(File)-> 新建(New)->工程(Project ...) ->Java -> Java Project， 点击 下一步(Next), 输入工程名称“helloworld”， 点击下一步（Next），点击完成(Finish)， 如果弹出“是否打开 java 视图(Open java perspective)”，点击是即可。

双击IDE左侧区域 “package Explorer” 中的 “helloworld”， 出现2个下拉菜单： JRE System Library 和 src， 其中的 src就是源代码文件所在的目录。点击 src，右键点击新建(New) -> 包(package) ， 名称就是用默认的helloworld， 点击完成。此时在src下出现一个类似于小魔方的图标(田字形)，后面紧跟着helloworld,这就是刚创建的package。在田字形和helloworld上点击右键，点击新建(New)->类(class), 输入名称Hello,注意首字母大写，这是Java 类名称的规范写法，点击确认。此时出现了一个 Hello.java的打开文件，其中内容如代码段3-1所示。

|  |
| --- |
| *package helloworld;  public class Helllo {  }* |

**代码段 3-1 类框架示例**

在其中编写源代码， 如代码段 3-2 所示。

|  |
| --- |
| *package helloworld;  public class Helllo {  public static void main(String[] args) {  System.out.println("hello world, i am here");  } }* |

**代码段 3-2 helloworld代码示例**

点击菜单栏上的 运行(Run) -运行(Run CTRL+F11)，此时，会在IDE的最下方的 Console Tab 栏中，出现一行 “hello world, i am here”。

## 3.2 通过命令行执行代码

* **创建 java 源代码文件**

以下脚本默认运行在 linux terminal环境下， windows类同， 如代码段3-3所示。

|  |
| --- |
| *mkdir workspace # 创建自己的 java 工程目录， mkdir为linux命令， make a directory，创建文件夹 pwd # 查看当前目录, pwd 为linux 命令， print working directory, 打印当前的工作目录 touch hello.java # 创建一个文件，文件名成为 hello.java, touch 为linux命令， 若文件不存在则创建，若文件存 在则更新访问时间 vi hello.java # 打开 hello.java 文件， vi 为linux命令，类似于windows 下的文本编辑器，打开一个文件* |

**代码段 3-3 源代码创建示例**

Hello.java 内容如下所示。vi打开文件后，按键盘键“i”， 进入输入模式，输入内容如代码段3-4所示。

|  |
| --- |
| *class Hello {  public static void main(String[] args) {  System.out.println("hello world");  } }* |

**代码段 3-4 源代码内容示例**

注意：如果在Windows 下编辑文件， 则最好将 hell.java 保存为UTF-8 文件（Windows 默认为 ASCII 编码格式），保持良好的编程习惯。linux默认的文件编码格式为UTF-8，不存在这个问题。另外需要说明的是，linux下严格区分文件名称大小写，而windows中则不区分文件名称的大小写。

java 源代码文件的文件名（Hello.java）称跟其内容中的 class 名称（Hello）保持一致，这是默认规则。当然不一样可以吗？也可以，但初学者暂时按这个规则来，后续再讲不一样的处理方法。

* **编译为java 字节码文件**

运行代码段3-5所示的内容。

|  |
| --- |
| *javac Hello.java # 编译源代码 ls # 查看当前目录下的文件 Hello.java Hello.class # 可以看到编译生成的字节码文件 hello.class* |

**代码段 3-5 编译源代码示例**

javac 为 java 的编译器（即 java compiler），作用是将 java源代码文件 （hello.java） 编译为字节码文件（java规范定义的一种可以在任意平台上被 java 运行时翻译为机器二进制可执行文件的文件格式） Hello.class 。

* **运行字节码文件**

执行如代码段3-6所示的内容。

|  |
| --- |
| *java Hello # 运行字节码文件 hello world # 可以看到控制台的输出* |

**代码段 3-6 运行java字节码文件示例**

java 命令在执行后形成的进程在Java 生态中叫做 Java 虚拟机（Java Virtual Machine，简称 JVM），在代码段3-6中，字节码Hello.class是运行在Java虚拟机（java Hello 中的java命令）中的，只有Java虚拟机才能够解析Hello.class中的内容。之所以叫虚拟机， 是因为JVM中有一套虚拟的机器指令，就像是一台物理硬件一样，所以叫“虚拟机器”。Java之所以这么做，是因为不同的硬件平台，其机器语言存在差异，如果像C语言那样编译，则必须编译多套目标代码，来适配不同的硬件。Java为了实现跨平台，实现它自己立的flag“write once, run any where”, 通过JVM来映射适配不同的硬件平台，而开发人员只需要编译一次，将源代码编译为Java 规范定义的一套字节码即可。

注意，此处命令为java，后面的参数为 Hello，即 Hello.class的前缀， JVM 默认会查找当前目录下的 Hello.class 文件，读取并加载至内存，然后解析，找到这个类的入口 （main函数），然后执行。

# 几个基础概念

## 源代码文件（source code file）

java 中的源文件是以“.java”为后缀的编码为UTF-8的纯文本文档。源文件的名称一般以大写字母开头，例如Animal.java, Cat.java, Car.java。

源代码文件，就是编程人员手动编辑输出的文件，不管是“.java”（Java 源代码文件）, “.py”（Python源代码文件）,“.c”（C语言源代码文件）还是“.html”（HTML源代码文件），本质上都是纯文本文件（可以通过文本编辑器直接查看内容的文件）。只不过为了方便管理，改了个后缀名而已。

## 4.2 类（class）

Java 中的 类 （class）， 表示对物理世界中的一个名词（例如，MyCat）、动词（例如，DrinkWater）。类在源文件中体现为以class开头包含在{}中的部分源代码。通过java编译后，会生成一个与源文件同名但以.class为后缀名的类文件，例如 Animal.class, Cat.class, Car.class。.class文件一般称之为类文件，是一个类的字节码文件。

类可以理解为对自然界中某一种事物的描述。

## 4.3 对象（Object）

在面对对象的编程中，对象是经常会遇到的一个名词，而对象又是和类直接相关的，有时候又称之为类的实例（Instance of the class）。对象可以理解为具体的事物。一个类表示一种对事物的描述方法， 而类的对象（Object）/实例（instance）表示具体的某个类。举例说明，类Person 如代码段 4-1 所示，类 Person 表示对“人”这个事物的一种描述方法，这种描述方法叫“Person”。

|  |
| --- |
| *class Person {* // 姓名 *String name;* // 年龄 *Integer age;* // 地址 *String address;* // 身高 *double height;* // 体重 *double weight;  }* |

**代码段 4-1 类 Person**

接着，创建一个类的对象 “person”，如代码段 4-2 所示。

|  |
| --- |
| *Person person = new Person(); person.name = "张三"; person.age=18; person.address="银河系火星"; person.height=175; person.weight=70;* |

**代码段 4-2 类 Person**

代码段4-2 中， 类“Person”的对象（实例）为“person”。在现实生活中，类“Person”是我们书本上写的“人”这个字，反正没啥感觉，就是个概念。而实例“person”则代表“你这个人”， “你这个家伙”，person是有血有肉的肉身人。

在软件世界中，类相当于是数据的定义，而实例是按照这种定义的实际的数据，以一个表格来说明，如表 4-1所示。

**表 4-1 类和类的对象（实例）类比**

| **Person（人）** | **name（姓名）** | **age（年龄）** | **address（地址）** | **height（身高:cm）** | **weight（体重:kg）** |
| --- | --- | --- | --- | --- | --- |
| person | 张三 | 18 | 银河系火星 | 175 | 70 |

如表4-1所示， 表头相当于是类， 而表格中的一行数据相当于类的一个实例（对象）。

## 4.4 包（package）

package 在Linux、Windows中体现为一堆嵌套的文件夹， 在 Java IDE 中以“.”连接的一串字符串，例如 com.alibaba.taobao.ui.user.login。那么从包名可以看出来这个是alibaba的淘宝这个应用的UI（用户界面）中的user（用户）的登录相关的而java类。

package 在 Java 中用于将某个功能模块涉及到的多个类文件放在一起，以方便管理。如果把一个源代码项目比喻成是一个衣柜，那么每个package可以理解为不同的衣橱，有的专门放外套，有的专门放鞋子，有的专门放裤子。

package的概念在很多编程语言中（C、python、C++）都有，主要为了方便源代码的管理，不然好几百个源代码文件放在一个文件夹下，放眼望去，太多了，无论是管理还是查找都不方便。

## 4.5 工程（project）

工程的概念，在很多编程语言中都有，其本质上是按照一定的结构，将一组源代码及其配置文件等形成一个完整的文件目录，最终能够完成某个特定的功能。

从IDE 的角度来说，工程是一个可以被 IDE 识别，被 IDE 独立管理、运行的一组文件。

## 4.6 静态语言

Java、C都属于静态语言，静态语言有个特征，都需要首先进行编译源文件/数据，生成一种格式的文件/数据，然后运行的时候需要运行生成的文件/数据。与此对应的有动态语言，例如Javascript、python、perl等都属于动态语言。动态语言的源代码文件无需编译，可直接加载至内存执行。

静态语言还有一个特征，所有的变量都需要预先定义，遵循先定义（声明）再使用的原则。例如 Integer a，告诉编译系统，变量 a 是一个整数，长度为4个字节。而像python这种动态语言，则可以给变量随意赋值，到底是什么类型，以及占用多少字节，都是在运行时（Runtime）决定的。用通俗的话来说，是“骡子是马，拉出来溜溜”，即通过程序的运行状态来决定到底是个什么变量。

总而言之，如果使用杯子来打比喻的话， 静态语言类似于先设定各种杯子的用途，玻璃杯子只能盛牛奶，陶瓷杯子只能盛茶，塑料杯子则只能盛水。而动态语言呢，则是先拿个杯子，在杯子里盛了啥饮料，那么这个杯子就是干什么的。

两种语言各有优缺点，静态语言在编译时会对一些基础错误进行检查，若存在潜在问题则可以事先发现（在编译时发现）。而且，各种变量是什么类型占用多少字节，都是预先规定好的，当程序运行起来（运行时，runtime）后，就只管执行就行，所以一般来说效率较高，但缺点是任何变量都需要实现进行约定，所以缺乏灵活性。动态语言的好处就是灵活性较强，一个变量，一会儿是个数字，一会儿可能就是个字符串了，使用较为方便，但缺点也很明显，就是在运行时，除了执行之外，还需要进行前置预处理，判断变量的类型从而决定内存分类，所以会有额外的开销，执行效率相较于静态语言来说要差一些。

## 4.7 Linux 命令

在Linux系统中，需要掌握一些基础的Linux命令，很多时候会起到事半功倍的效果， 如表 4-1 所示。

**表 4-1 常用 linux 命令清单**

| **No** | **命令（Name）** | **描述（DESCRIPTION）** |
| --- | --- | --- |
| 1 | pwd | print name of current/working directory,打印当前工作目录， linux中在没有图形化用户界面的条件下，通过pwd命令可以直到当前所在的文件目录中的位置 |
| 2 | mkdir | make directories， 创建目录（文件夹）。 |
| 3 | rm | remove files or directories， 删除一个文件或文件夹， 例如 rm file\_a rm directory\_b |
| 4 | ls | list directory contents， 现实指定目录下的文件和文件夹 例如 ls ./ （显示当前目录下的文件和文件夹） ， ls /usr/lib （显示/usr/lib 目录下的文件和文件夹） |
| 5 | ./ | . 或者 ./ 表示当前目录 |
| 6 | .. | .. 或者 ../ 表示上一级目录 |
| 7 | man | 查询系统命令的说明， 例如想知道pwd这个命令怎么用， 可以通过 man pwd 来查看 pwd 命令的用户手册，相当于windows下的help |
| 8 | ifconfig | 查看网络配置，例如IP, 网关，子网掩码等， 相当于windows下的ipconfig |
| 9 | telnet | 探测一个TCP 服务是否能够连接上，这个与windows相同， 例如 telnet 192.168.1.1 8080， 探测IP为192.168.1.1, 端口为8080 的服务是否能连接上 |
| 10 | top | 查看系统负载， 包括 CPU load, 内存利用率，是否有僵尸进程等 |
| 11 | ps -ef | 按照进程编号、进程启动命令及参数关键字查找进程信息， 例如 ps -ef | grep aaaa.bbbb.jar, 查看启动指令中包含“grep aaaa.bbbb.jar”指令的进程 |
| 12 | cat | cat - concatenate files and print on the standard output， 打印一个文件的内容至控制台，例如 cat test.txt将打印text.txt的内容至控制台 |
| 13 | grep | 查找一个文件中包含某个关键字的行的内容，例如 grep mykeywords test.txt 将打印包含关键字 mykeywords 的行至控制台 |
| 14 | where/which | 查找某个命令的绝对路径， 例如 执行which sh 可得到 /usr/bin/sh,即可得到sh命令的绝对路径为 /usr/bin/sh |
| 15 | tar -zxf | 解压缩并打开一个tar（发音：踏）包， 例如执行 tar -zxf abc.tar.gz ，即可得到一个abc的文件夹。tar用于对linux下的文件夹进行打包形成一个文件，方便在脚本等环境中进行拷贝等操作。 |
| 16 | tar -czf | 压缩并打包为一个tar包（发音：踏）包，例如执行 tar -czf abc.tar.gz abc, 即将当前目录下的abc文件夹及其内部的所有文件打包为一个压缩文件 abc.tar.gz。 |
| 17 | ssh | ssh user@IP, 通过SSH在客户端连接至网络上服务端的服务器，登录网络上的某个服务器，例如 ssh guest@192.168.1.1.1,接下来输入密码就登录到远程的服务器上了，类似于Windows下的远程桌面功能，只是ssh没有图形化界面。 |

# 包依赖管理

在程序运行的时候，总是需要一些第三方（其他开发者开发的）的工具， 例如JSON对象解析，ZIP文件解压缩，作为开发人员来说，不可能所有的工具都需要自己去做（重复发明轮子）。这就涉及到如何有效、便捷地使用第三方软件（包、类库等）的问题。在Java的生态中，有 ant、maven、gradle等构建工具（builld tool）可以使用。

打一个通俗的比喻，需要建设一栋房子，开发人员只需要涉及好图纸，打好地基，然后使用第三方开发人员开发的砖头、混凝土、屋顶预制板材、窗户、壁纸等，自己按照建筑设计规范组装起来，最终达到一定抗震、保暖、透光、防水要求即可。那么如何管理砖头供货商、混凝土供货商、窗户供货商的信息，以便于在必要的时间点，及时准确地让第三方的供货商参与进来，就是构建工具（build tool）要做的事情了。下面以maven为例，进行说明。

## 5.1 maven工程介绍

Java 中如果需要使用maven（发音：mān wen）来进行依赖包（一般称为jar包（发音：榨包），因为java中第三方打的包都是xxxx.jar格式的），则需要按照一定的文件目录格式放置相应的文件，maven才能正常工作。就是说maven比较挑起，必须按照她的要求把文件放好，它才好好干活儿。

maven是阿帕其基金会（Apache Fundation）管理的开源（open source）项目， 是当前主流的依赖包管理工具， ant历史更早，新兴的gradle也正在被使用。典型的maven工程目录如代码段5-1所示。

|  |
| --- |
| *. ├── pom.xml ├── src │   ├── main │   │   ├── java │   │   └── resources │   └── test │   ├── java │   └── resources └── target* |

**代码段 5-1 maven 项目文件目录结构**

pom.xml 是一个 XML 配置文件，内容涉及当前项目的名称、项目构建方法、依赖包的清单、依赖的父工程（maven工程可以嵌套）等信息。

src为源文件目录， target为编译生成的文件所在目录。src/main/java为java源代码文件（.java）所在目录，src/main/resources 为源代码所需要的配置文件、资源文件等其他不以“.java” 为后缀名的文件所在的目录。src/test/java 为java单元测试文（.java） 所在的目录， src/test/resources 为单元测试代码所需要的配置文件、资源文件等其他不以“.java” 为后缀名的文件所在的目录。

这里提到了“单元测试”的概念，还是以建设一栋房屋为例，来介绍单元测试的概念。建设一栋房屋，不能等房屋完全竣工再去查看地基是否达标、窗户是否达标、防水是否达标，而应该是在各个重要的阶段就对前一阶段的建设成果（一个单元）进行检验，这个就叫单元测试。虽然单元测试不能保证最终的房屋能够完全达标，但却可以事先发现某个单元存在的问题。或者说，当房屋最终质量出现问题时，通过单元测试能很快定位到是哪个部分（单元）出了问题。

这里还提到了 XML 文件，XML（可扩展标记语言 (Extensible Markup Language，简称 XML)）文件是一个文本文件，文件内容结构代码段 5-2所示。

|  |
| --- |
| *<?xml version="1.0" encoding="utf-8"?> <myfile>  <name>  <firstname>who</firstname>  <lastname>ami</lastname>  </name>  <time>2024-07-15</time>  <description>this is a demo xml file to illstrate the file structrue</description> </myfile>* |

**代码段 5-2 XML 文件内容结构**

相当于在文本文档的基础上，通过添加标签<tag></tag>，将文本进行分割，添加了说明，方便阅读，另外一个重要的作用，可以方便计算机进行处理。如果是文本文档，只能要求计算机读取第几行，但是XML 文档，就可以要求计算机读取代码段 4-2中的 myfile/name/firstname中的内容，这样是不是更加方便呢？这就是 XML 文件的作用了。这里的 “myfile/name/firstname” 被称之为 XPATH，有兴趣可以阅读 XML相关的书籍进行了解。

## 5.2 maven工程创建（create）

maven工程目录结构，可以手动进行创建，也可以通过IDE进行创建，一般IDE都提供有创建maven工程的图形化界面。

打开Eclipse（以 eclipse-jee-2024-06-R-linux-gtk-x86\_64 为例），点击 文件（File）->新建（New）->Maven Project（Maven 工程），在弹出的对话框中，选中“Create a simple project(skip archetype selection)”，选中“Use default workspace location”, 点击“下一步（Next）”， 在 group id 中填写 “cn.java.study”,， 在 artifact Id 中填写 “my.maven.demo”， 点击“完成（Finish）”。此时，在IDE 左侧的导航栏中能够看到如代码段 5-3 所示的目录结构。

|  |
| --- |
| *my.maven.demo  ├── pom.xml  ├── src  │   ├── main  │   │   ├── java  │   │   └── resources  │   └── test  │   ├── java  │   └── resources  └── target* |

**代码段 5-3 maven 工程结构示例**

在pom.xml 的图标上可能会出现红色的x符号，表示此文件存在错误，可以打开pom.xml文件，将 <project>所在的行替换为如下内容

|  |
| --- |
| *<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">* |

使得最终pom.xml的内容如代码段 5-4 所示。

|  |
| --- |
| *<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>cn.java.study</groupId>  <artifactId>my.maven.demo</artifactId>  <version>0.0.1-SNAPSHOT</version> </project>* |

**代码段 5-4 pom.xml 文件内容**

然后在 IDE左侧导航界面中选中“my.maven.demo”的图表， 右键菜单中点击“Maven”-> "Update Project"，在弹出的对话框中点击 其他错误，暂时忽略。

## 5.3 maven工程构建（build）

### 5.3.1 安装maven工具

此时，maven工程已经建好了，需要使用maven工具进行代码构建了。maven工具一般可以在IDE中安装插件，也可以从apache 官方网站下载，这里使用从apache官网下载的工具为例，降低复杂性。地址为 <https://maven.apache.org/download.cgi>。点击 “[apache-maven-3.9.8-bin.tar.gz](https://dlcdn.apache.org/maven/maven-3/3.9.8/binaries/apache-maven-3.9.8-bin.tar.gz)” 进行下载。

下载完成后，解压， 假定目录为/a/b/c/apache-maven-3.9.8/， 后续例子中可以替换为自己实际的文件目录, 需使用到的命令为 /a/b/c/apache-maven-3.9.8/bin/mvn，即mvn命令（Windows下为mvn.exe）。可以将这个路径添加到环境变量PATH中，后续就可以直接在控制台中输入“mvn”了，而无需使用绝对路径 “/a/b/c/apache-maven-3.9.8/bin/mvn” 才能使用mvn命令了。

这里推荐开发人员使用 Ubuntu linux系统，Ubuntu desktop版的系统提供 GUI 用户界面，对于开发人员来说，使用linux系统进行开发工作，利大于弊。当然，如果是 C#（一种微软研发的在windows生态中使用的对标 Java的开发语言）开发人员的话，那只能使用Windows系统了，C#不是开源的开发语言，也无法部署在linux中（当然什么都不是绝对的，或者叫部署难度较大，与linux的生态不兼容）。对于Java开发人员来说，使用Ubuntu 能够让开发人员很快熟悉起linux系统来，而且很多操作习惯，可以直接移植至 server端。

如果是ubuntu系统，可以直接在终端（CTRL + ALT+T）通过以下命令安装maven工具。

|  |
| --- |
| *sudo apt-get install maven* |

### 5.3.2 编译

进入“代码段 5-3 maven 工程结构”的工程根目录，Ubuntu 下的执行方法如代码段 5-5 所示。这里假定 my.maven.demo 处于目录 /a/b/c/d/e 下。

|  |
| --- |
| *# 进入 my.maven.demo 工程的根目录 cd /a/b/c/d/e/my.maven.demo # 使用maven 进行代码编译， 如果没有将maven的安装路径添加至系统变量 PATH中时， # 此处的mvn需要使用绝对路径 /\*\*\*/apache-maven-3.9.8/bin/mvn mvn compile* |

**代码段 5-5 mvn 编译方法示例**

此时，可看到如代码段 5-6 所示的输出。

|  |
| --- |
| *[INFO] --- resources:3.3.1:resources (default-resources) @ my.maven.demo --- [WARNING] Using platform encoding (UTF-8 actually) to copy filtered resources, i.e. build is platform dependent! [INFO] Copying 0 resource from src/main/resources to target/classes [INFO]  [INFO] --- compiler:3.13.0:compile (default-compile) @ my.maven.demo --- [INFO] Nothing to compile - all classes are up to date. [INFO] ------------------------------------------------------------------------ [INFO] BUILD SUCCESS [INFO] ------------------------------------------------------------------------ [INFO] Total time: 0.269 s [INFO] Finished at: 2024-07-15T10:26:27+08:00 [INFO] ------------------------------------------------------------------------* |

**代码段 5-6 mvn 编译输出示例**

注意，执行 mvn compile 命令时，maven会请求网络（需保证使用的计算机能够顺畅访问网络），从默认的中央仓下载一些文件至本地。本地文件的存储路径，以ubuntu为例，若 当前操作系统登录的用户名为 avata, 则maven本地的缓存目录为/home/avata/.m2/repository。若为Windows系统，则在 C:/User/用户名/Application data/\*\*\*/.m2/repository 下。至此，maven工程的编译完成了。

### 5.3.3 打包

上面的步骤只是将java源文件编译为 .class文件，作为一个工程来讲，有很多个.class文件，还有一些配置文件，作为可部署运行的程序，还需要通过打包（package）形成简单可部署、可执行的jar文件。

ubuntu下的执行方法如代码段 5-7 所示。

|  |
| --- |
| *# maven 首先执行clean，清空之前打包的文件，然后执行 package,打包 mvn clean package* |

**代码段 5-7 mvn 打包命令示例**

|  |
| --- |
| *[INFO] --- maven-jar-plugin:2.4:jar (default-jar) @ my.maven.demo --- [INFO] Building jar: /home/rd/eclipse-workspace-java/my.maven.demo/target/my.maven.demo-0.0.1-SNAPSHOT.jar [INFO] ------------------------------------------------------------------------ [INFO] BUILD SUCCESS [INFO] ------------------------------------------------------------------------ [INFO] Total time: 0.511 s [INFO] Finished at: 2024-07-15T10:27:11+08:00 [INFO] ------------------------------------------------------------------------* |

此时，可看到如代码段 5-8 所示的输出。

**代码段 5-8 mvn 打包输出内容示例**

此时，形成了可在JVM 中执行的文件 my.maven.demo-0.0.1-SNAPSHOT.jar， 执行如下命令

|  |
| --- |
| *java -jar target/my.maven.demo-0.0.1-SNAPSHOT.jar* |

输出

|  |
| --- |
| *no main manifest attribute, in target/my.maven.demo-0.0.1-SNAPSHOT.jar* |

此时，说明可部署的文件 my.maven.demo-0.0.1-SNAPSHOT.jar 是没问题的，由于前面的示例代码段 4-3 maven 工程结构示例中， 在src/main/java目录下，没有任何的java源代码文件，所以这里提示 “no main manifest attribute”。

## 5.4 maven demo 运行（run）

添加源代码， 在src/main/java中添加源代码文件。

（1）在IDE左侧导航栏中， 点击“src/main/java”图表，右键，新建(New) -> 文件(File), 在弹出的对话框中的 文件名（File name）一栏中，输入 “Test.java”, 点击 “完成(Finish)”。此时在 IDE 左侧导航栏中出现一个 “Test.java”的图标，同时在其上方出现一个“default package”的图标。可见当添加文件时，IDE默认只是添加一个普通文件，不一定是java 源代码文件。

（2）点击“src/main/java”图表，右键，新建(New) -> 类(Class), 在弹出的对话框中，在文件名（Name）一栏中，输入 “HelloMvn”, 确保其他几项正确 Source folder = my.maven.demo/src/main/java， Package=my.maven.demo， 其他保持默认项不变，点击 “完成(Finish)”。

此时在 IDE 左侧导航栏中出现一个 “HelloMvn.java”的图标，同时在其上方出现一个“my.maven.demo”的包(package)图标。

1. 双击打开 IDE 左侧导航栏中的 HelloMvn.java ，可见IDE 自动生成的源代码如代码段 5-9 所示。

|  |
| --- |
| *package my.maven.demo;  public class HelloMvn {  }* |

**代码段5-9 IDE 自动生成代码示例**

修改源代码，使其内容如代码段 5-10 所示。

|  |
| --- |
| *package my.maven.demo;  public class HelloMvn {* /\*\*\*主程序入口，指定了一批*class*文件，从哪个文件的哪个方法开始执行 \* @*param args* \*/ *public static void main(String[] args) {    System.out.println("Hello maven");  } }* |

**代码段 5-10 IDE 添加源代码示例**

在 pom.xml 中添加 打包配置 <build> 节点，pom.xml 文件内容如代码段 5-11 所示。

|  |
| --- |
| *<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>cn.java.study</groupId>  <artifactId>my.maven.demo</artifactId>  <version>0.0.1-SNAPSHOT</version>  <build>  <plugins>  <plugin>  <groupId>org.apache.maven.plugins</groupId>  <artifactId>maven-compiler-plugin</artifactId>  <version>3.6.0</version>  <configuration>  <source>1.8</source>  <target>1.8</target>  <encoding>UTF-8</encoding>  </configuration>  </plugin>  <plugin>  <groupId>org.apache.maven.plugins</groupId>  <artifactId>maven-assembly-plugin</artifactId>  <version>3.6.0</version>  <configuration>  <archive>  <manifest>  <mainClass>my.maven.demo.HelloMvn</mainClass>  </manifest>  </archive>  <descriptorRefs>  <descriptorRef>jar-with-dependencies</descriptorRef>  </descriptorRefs>  </configuration>  <executions>  <execution>  <id>make-assembly</id>  <phase>package</phase>  <goals>  <goal>single</goal>  </goals>  </execution>  </executions>  </plugin>  </plugins>  </build> </project>* |

**代码段 5-11 pom XML build节点配置示例**

开始编译打包，执行

|  |
| --- |
| # 首先clean,接着package, 跳过了compile.当package时，若没有compile生成的class文件 会自动执行compile ~/software/apache-maven-3.9.8/bin/mvn clean package |

此时，可以在target目录下看到生成的文件清单如代码段5-12所示。

|  |
| --- |
| *# 执行ls ls -al target/ # 输出的结果 total 44 drwxrwxr-x 9 rd rd 4096 7月 15 10:56 . drwxrwxr-x 5 rd rd 4096 7月 15 10:53 .. drwxrwxr-x 4 rd rd 4096 7月 15 10:58 classes drwxrwxr-x 3 rd rd 4096 7月 15 10:53 generated-sources drwxrwxr-x 3 rd rd 4096 7月 15 10:56 generated-test-sources drwxrwxr-x 2 rd rd 4096 7月 15 10:56 maven-archiver drwxrwxr-x 3 rd rd 4096 7月 15 10:53 maven-status -rw-rw-r-- 1 rd rd 2520 7月 15 10:59 my.maven.demo-0.0.1-SNAPSHOT.jar -rw-rw-r-- 1 rd rd 2547 7月 15 10:59 my.maven.demo-0.0.1-SNAPSHOT-jar-with-dependencies.jar drwxrwxr-x 2 rd rd 4096 7月 15 10:54 test-classes* |

**代码段 5-12 maven编译生成的文件清单**

可以看到， maven默认打包生成的文件名为 artifactId-version.jar, 这个文件包含了开发人员自行写的源代码对应的所有class文件。文件artifactId-version-jar-with-dependencies.jar中，除了开发人员自行写的源代码外，还有pom文件中配置的所有依赖的jar包中的class文件。

接下来，可以在JVM 中运行 jar包了，如代码段 5-13所示。

|  |
| --- |
| *java -jar target/my.maven.demo-0.0.1-SNAPSHOT-jar-with-dependencies.jar* *# 输出 Hello maven* |

**代码段 5-13 运行maven项目打包生成的文件**

从代码段5-13中可见，maven工程源文件编译成功，而且通过打包形成可运行的文件。

# 简单的Web 应用程序

通过这个单元的学习，能够进行简单的Web应用程序的构建， 程序启动后可以通过浏览器输入 URL 进行访问。

## 6.1 Hello Web

在 Unit 4 中， 学习了 maven 工程的基本构建方法，本节的 Java web 工程将采用 maven类进行构建，所以这个工程是一个 Java Web 工程，又是一个Java Maven 工程，这是从两个方面来说的，不产生冲突。

1. **添加pom依赖。** 首先在上一节的 pom.xml 中添加节点 dependencies， 添加跟Web相关的第三方组件dependency(依赖)，形成的pom.xml 文件 如代码段 6-1 所示。

|  |
| --- |
| *<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>cn.java.study</groupId>  <artifactId>my.maven.demo</artifactId>  <version>0.0.1-SNAPSHOT</version>  <dependencies>  <dependency>  <groupId>io.netty</groupId>  <artifactId>netty-all</artifactId>  <version>4.1.32.Final</version>  </dependency>  </dependencies>  <build>  <plugins>  <plugin>  <groupId>org.apache.maven.plugins</groupId>  <artifactId>maven-compiler-plugin</artifactId>  <version>3.6.0</version>  <configuration>  <source>1.8</source>  <target>1.8</target>  <encoding>UTF-8</encoding>  </configuration>  </plugin>  <plugin>  <groupId>org.apache.maven.plugins</groupId>  <artifactId>maven-assembly-plugin</artifactId>  <version>3.6.0</version>  <configuration>  <archive>  <manifest>  <mainClass>my.maven.demo.HelloMvn</mainClass>  </manifest>  </archive>   <descriptorRefs>  <descriptorRef>jar-with-dependencies</descriptorRef>  </descriptorRefs>  </configuration>  <executions>  <execution>  <id>make-assembly</id>  <phase>package</phase>  <goals>  <goal>single</goal>  </goals>  </execution>  </executions>  </plugin>  </plugins>  </build> </project>* |

**代码段 6-1 pom 添加 web组件依赖示例**

注意，节点“mainClass”中的内容，即jar 包中主程序启动的入口，后续需要修改。本节所写的代码依赖于如下所示的第三方组件

|  |
| --- |
| *<groupId>io.netty</groupId> <artifactId>netty-all</artifactId> <version>4.1.32.Final</version>* |

添加完代码段 6-1 pom 中所示的依赖项后，在项目根目录中执行

|  |
| --- |
| # 强制更新 pom 依赖， 对项目进行编译 *mvn -U compile* |

此时， maven的mvn命令会从默认的maven中央仓（<https://repo.maven.apache.org/maven2>）中拉取（pull）相应的jar 包信息，存储至本地的目录（\*\*\*/.m2/repositoty）下，可以通过命令行（Windows中可以打开文件夹）查看， 如代码段 6-2 所示。

|  |
| --- |
| *# 进入 maven本地缓存文件默认存储目录 cd \*\*\*/.m2/repositoy # 进入 grouId/artifactId/version 目录下 cd io/netty/netty-all/4.1.32.Final # 查看文件 ls -al # 输出结果 total 6916 drwxrwxr-x 2 rd rd 4096 7月 15 14:09 . drwxrwxr-x 12 rd rd 4096 6月 6 16:40 .. -rw-rw-r-- 1 rd rd 3915287 4月 16 14:08 netty-all-4.1.32.Final.jar -rw-rw-r-- 1 rd rd 40 4月 16 14:08 netty-all-4.1.32.Final.jar.sha1 -rw-rw-r-- 1 rd rd 25566 4月 16 14:07 netty-all-4.1.32.Final.pom -rw-rw-r-- 1 rd rd 40 4月 16 14:07 netty-all-4.1.32.Final.pom.sha1 -rw-rw-r-- 1 rd rd 3105409 5月 30 09:50 netty-all-4.1.32.Final-sources.jar -rw-rw-r-- 1 rd rd 540 5月 30 09:50 netty-all-4.1.32.Final-sources.jar.lastUpdated -rw-rw-r-- 1 rd rd 40 5月 30 09:50 netty-all-4.1.32.Final-sources.jar.sha1 -rw-rw-r-- 1 rd rd 400 7月 15 14:09 \_remote.repositories* |

**代码段 6-2 maven下载到本地的 jar 包及相关文件信息示例**

可见本地缓存目录下已经下载了 第三方jar 包 netty-all-4.1.32.Final.jar，说明 pom.xml 中添加的依赖是成功的。

（2）提供Web 程序入口。在 IDE 左侧导航栏（Project Explorer）中选中 “src/main/java”图标，右键->新建(New) -> 包(package) , 在弹出的 “新java 包(New Java Package)”对话框中的 “名称(Name)”中输入 “my.web.demo”， 确保对话框中的 Source Folder=my.maven.demo/src/main/java， 点击“Finish(完成)”。

此时 IDE 左侧导航栏(Project Explorer) 中出现一个 my.web.demo的图标， 表示刚才新建的包(package)是成功的。左键点击该图表，右键->新建(New)->类(class)， 在“Name”输入框中输入 “Bootstrap”，点击确定。文件“Bootstrap.java" 中的代码内容如代码段 6-3 所示。

|  |
| --- |
| *package my.web.demo;* /\*\*\*程序启动（自举）入口\*/ *public class Bootstrap {* /\*\*\*定义了*Web*监听的端口号\*/ *private static Integer PORT = 8080;  public static void main(String[] args) throws Exception {* /\*\*\*实例化*Server* 对象\*/ *final Server srv = new Server();* /\*\*\*启动*Server* \*/ *srv.start(Bootstrap.PORT);  } }* |

**代码段 6-3 Web 程序启动(自举)入口**

在包(package) my.web.demo 中新建类(class) Server, Server.java中的代码内容如代码段 6-4 所示。

|  |
| --- |
| *package my.web.demo; import io.netty.bootstrap.ServerBootstrap; import io.netty.channel.ChannelFuture; import io.netty.channel.ChannelInitializer; import io.netty.channel.ChannelOption; import io.netty.channel.EventLoopGroup; import io.netty.channel.epoll.Epoll; import io.netty.channel.epoll.EpollServerSocketChannel; import io.netty.channel.nio.NioEventLoopGroup; import io.netty.channel.socket.SocketChannel; import io.netty.channel.socket.nio.NioServerSocketChannel; import io.netty.handler.codec.http.HttpObjectAggregator; import io.netty.handler.codec.http.HttpServerCodec;* /\*\*\*一个标准的 *HttpServer* \*/ *public class Server {    private static final int BS\_NUM = Runtime.getRuntime().availableProcessors();  private static final int WK\_NUM = BS\_NUM \* 2;  private static final EventLoopGroup BS\_GRP = new NioEventLoopGroup(BS\_NUM);  private static final EventLoopGroup WK\_GRP = new NioEventLoopGroup(WK\_NUM);    public void start(final int port) throws Exception {  try {  final ServerBootstrap boot = new ServerBootstrap();  boot.channel(NioServerSocketChannel.class)  .option(ChannelOption.SO\_BACKLOG, 1024)  .childOption(ChannelOption.SO\_KEEPALIVE, true)  .childOption(ChannelOption.TCP\_NODELAY, true)  .group(Server.BS\_GRP, Server.WK\_GRP)  .childHandler(this.buildInitializer());  ChannelFuture future = boot.bind(port).sync();  System.out.println("my\_web\_demo\_server\_started, listen " + port);  future.channel().closeFuture().sync();  } finally {  Server.BS\_GRP.shutdownGracefully();  Server.WK\_GRP.shutdownGracefully();  System.out.println("mt\_api\_srv\_shutdown");  }  }    private ChannelInitializer<SocketChannel> buildInitializer() {  return new ChannelInitializer<SocketChannel>() {  @Override  protected void initChannel(SocketChannel ch) throws Exception {  ch.pipeline()  .addLast("httpDec", new HttpServerCodec())  .addLast("httpAgr", new HttpObjectAggregator(65536))* /\*\*\*添加自定义的*Http*处理器\*/ *.addLast("htpHdl", new HttpHandler());  }  };  } }* |

**代码段 6-4 Web 服务器(Server)核心代码示例**

在包(package) my.web.demo 中新建类(class) HttpHandler, HttpHandler.java中的代码内容如代码段 6-5 所示。

|  |
| --- |
| *package my.web.demo;  import io.netty.channel.ChannelFutureListener; import io.netty.channel.ChannelHandlerContext; import io.netty.channel.SimpleChannelInboundHandler; import io.netty.handler.codec.http.\*; import io.netty.util.CharsetUtil;* /\*\*\*自定义的 *Http* 处理器\*/ *public class HttpHandler extends SimpleChannelInboundHandler<FullHttpRequest> {* /\*\*\*当服务器端收到客户端的请求数据，该做的处理\* *FullHttpRequest req* 代表来自客户端的请求数据\* *ChannelHandlerContext ctx* 为上下文信息\*/ *@Override  protected void channelRead0(ChannelHandlerContext ctx, FullHttpRequest req) {  final long id = System.currentTimeMillis();  System.out.println(id + ", rcv\_rq\_addr, " + ctx.channel().remoteAddress() + ", " + req.uri());  final String data = req.content().toString(CharsetUtil.UTF\_8);  boolean keepAlive = HttpUtil.isKeepAlive(req);  final String uri = req.uri();  final String txt = String.format("{\"status\":200, \"timestamp\":%s}", System.currentTimeMillis());  final FullHttpResponse response =  new DefaultFullHttpResponse(HttpVersion.HTTP\_1\_1, HttpResponseStatus.OK);  response.content().writeBytes(txt.getBytes());  response.headers().set(HttpHeaderNames.SERVER, "my\_web\_demo\_server");  response.headers().set(HttpHeaderNames.CONTENT\_TYPE, "application/json;charset=UTF-8");  response.headers().setInt(HttpHeaderNames.CONTENT\_LENGTH, response.content().readableBytes());  if (keepAlive) {  response.headers().set(HttpHeaderNames.CONNECTION, HttpHeaderValues.KEEP\_ALIVE);  ctx.writeAndFlush(response);  } else {  ctx.writeAndFlush(response).addListener(ChannelFutureListener.CLOSE);  }  }* /\*\*\*当发生异常后，需要做的处理\*/ *@Override  public void exceptionCaught(ChannelHandlerContext ctx, Throwable cause) {  if (cause.getMessage().contains("certificate\_unknown")) {  System.out.println("ssl\_error, " + cause.getMessage());  } else {  System.out.println("error, " + cause);  ctx.channel().close();  }  } }* |

**代码段 6-5 HTTP 请求核心代码示例**

至此，源代码已经完成了，需要修改 pom文件中的主程序入口，将pom.xml中的节点project/build/plugins下的 <artifactId>maven-assembly-plugin</artifactId>下的 configuration/archive/manifest/mainClass修改为 “my.web.demo.Bootstrap”，即<mainClass>my.web.demo.Bootstrap</mainClass>。

开始打包并运行，执行

|  |
| --- |
| *# 使用mvn安装的的绝对路径，进行 清理已打包的jar包(clean),重新打包(package) /xxxx/apache-maven-3.9.8/bin/mvn clean package # 运行打包形成的jar包， 启动 HTTP server java -jar target/my.maven.demo-0.0.1-SNAPSHOT-jar-with-dependencies.jar #输出结果 my\_web\_demo\_server\_started, listen 8080* |

至此， 一个简单的 HTTP Server 已经启动了。接下来验证下这个Server是否可以正常工作，打开浏览器，输入 <http://localhost:8080>, 回车，会看到以下结果

|  |
| --- |
| *{"status":200, "timestamp":1721027033026}* |

说明刚建立的 HTTP server已经在工作了，可以在启动 HTTP server执行java -jar target/my.maven.demo-0.0.1-SNAPSHOT-jar-with-dependencies.jar的控制台（Windows里的cmd, Ubuntu里的Terminal(CTRL+ALT+T)）界面下，看到输出的日志

|  |
| --- |
| *1721027313965, rcv\_rq\_addr, /[0:0:0:0:0:0:0:1]:49422, / 1721027313975, rcv\_rq\_addr, /[0:0:0:0:0:0:0:1]:49422, /favicon.ico 1721027314339, rcv\_rq\_addr, /[0:0:0:0:0:0:0:1]:49422, / 1721027314346, rcv\_rq\_addr, /[0:0:0:0:0:0:0:1]:49422, /favicon.ico* |

每请求一次，就会输出几行日志信息。

## 6.2 几个概念

**（1）HTTP请求。**HTTP 是 Hypertext Transfer Protocol（超文本传输协议）的简称。当用户打开浏览器，输入一个地址（例如， <https://www.baidu.com>）时，浏览器开始构建相关的 HTTP请求报文，相对应的服务器发起请求，服务器返回相关的数据信息（可能是一个html页面，一个电影、一个图片、一段文本），此时浏览器根据返回数据的类型，来决定如何展示给用户，如果是一个网页那么直接渲染（就像我们看到的新闻页面那样），如果是一个电影（例如某种流媒体格式）可能会在浏览器中直接播放。

**（2）HTTP服务器。**作为相应客户端请求的一个实时运行（监听）的程序，等待客户端发起的请求，并根据客户端的请求，返回相应的内容。打一个比喻，HTTP服务器类似于咖啡店的服务员，等待着喝咖啡顾客（HTTP请求）的到来，当客户点了某种咖啡（浏览器发起了某种HTTP请求）后，咖啡店的服务员开始制作，等待一段时间后将咖啡提供给顾客（从服务端返回给浏览器后被浏览器渲染出来的用户可见的内容）。

**（3）端口监听。**一台主机（即日常我们使用的台式电脑、笔记本电脑）上可以启动多个网络服务（Web服务），供用户通过浏览器访问。为了区分同一个地之下的多个网络服务，引入了端口的概念。打个比喻，一台主机相当于一栋大学宿舍楼，很多人都住在这栋楼里，那么这些同学的联系地址都是某大学某栋楼，那如何定位到某个宿舍呢，端口号类似于宿舍房间号。端口号的范围从1-65535共计6万多个，不过一般来说1-1024这1000多个端口号作为操作系统网络服务的默认端口号，例如默认HTTP 的端口号为80，HTTPS的端口号为443，SSH 端口号为22，ftp服务的端口号为21等。

在我们日常浏览网站的时候，似乎没用到端口号。比如 <http://www.baidu.com>,事实上，这个地址使用了默认端口号80，完整的地址为<http://www.baidu.com:80>。对于默认的端口号，可以省略。

# 源代码管理

## 7.1 基本概念

Source Code Management（源代码管理， 简称SCM），目前主流的源代码管理工具有 git、svn、TFS等。最受欢迎的当然是git了， svn在一些较早的开源项目中在使用，新项目一般都使用git。一些windows下的项目使用TFS,主要微软一般喜欢搞一些全家桶的东西，什么都用微软生态系统内的组件，跟开源精神背道而驰。

**（1）必要性。**为什么要使用源代码管理工具？是不是可以不用。答案是可以不用，但实际大型多人合作的开发项目，如果不使用源代码管理，那将是灾难性的。通过一个场景进行说明， 某大型开发项目，一共有开发人员50人，每人负责一个小模块，整个源代码所有人都可以修改（这也是开源linux的典型开发模式）。当A在修改某个文件abcde.java时，B也想修改这个文件abcde.java，怎么样让A和B同时修改，互相不干扰对方的工作，而且还不会造成混乱呢？ 还有一种场景，某大佬一个人在干某个开源项目（例如，开发curl的大佬），源代码文件有好几百个，断断续续改了2年的时间，突然有一天，他改来改去，发现其实6个月前的那一版源代码是最科学的，那怎么退回到6个月前的源代码版本呢？有没有一个工具可以一键实现这个功能？这就是SCM 工具发挥作用的时候了。

**（2）工作机制**

1）版本记录。git会在本地保存一份文件修改记录，当用户向远端（remote）的网络服务器（git server）提交时，这些修改记录会在git server端也保留下来，当发现同一个文件的同一行被多个人修改后，当开发人员提交修改时，会给出提醒，由开发人员决定如何合并修改（merge），然后再保存之远端的服务器。

2）多分支（branch）。当一份源代码需要被多个人修改时，可以生成多个不同的分支，不同分支上的源代码可以通过 git merge进行合并。

## 7.2 安装git

（1）安装git。windows 系统可通过 <https://git-scm.com/download/win> 下载安装，安装 Standalone Installer， 下载地址为 <https://github.com/git-for-windows/git/releases/download/v2.45.2.windows.1/Git-2.45.2-64-bit.exe>。

ubuntu系统通过如下命令进行安装

|  |
| --- |
| *sudo apt-get install git* |

最终，验证安装是否成功， 可通过执行一下命令进行验证。

|  |
| --- |
| *git help -a # 若输出如下内容，证明安装成功, 否则可通过设置环境变量等，查找问题 See 'git help <command>' to read about a specific subcommand  Main Porcelain Commands  add Add file contents to the index  am Apply a series of patches from a mailbox* |

## 7.3 代码管理

使用 git 进行源代码管理需要执行如代码段 7-1 所示的步骤。

|  |
| --- |
| *# [1] 进入工程根目录 cd project\_root\_dir # [2] 初始化git管理 git init # [3] 添加需要被管理的文件、文件夹 git add a\_file git add b\_directory # [4] 提交修改记录， -m 后面的内容是本次提交的内容提示 git commit -m 'this is my first modification of source code' # [5] 将修改推送至远端 git server，这里还需要尽心额外配置告诉git 远端服务器的地址，以及推送的分支，这里不做演示，只展示本地修改管理 git push # [6] 修改源代码 a/b/c/d.java, a/b/c/e.java # [7] 查看源代码状态，可以看到 git 记录了文件 a/b/c/d.java 和 a/b/c/e.java被修改了 git status # [8] 添加被修改的文件 git add a/b/c/d.java git add a/b/c/e.java # [9] 提交修改 git commit -m 'i modified 2 files, d.java and e.java, just for fun, enjoy life' # [10] 查看提交记录 git log* |

**代码段 7-1 git 源代码管理步骤示例**

以 “6.简单的Web应用程序” 小节中的代码为例，进行演示，实际操作如代码段 7-2 所示。

|  |
| --- |
| *# 进入工程根目录，即需要管理的代码的根目录, windows下请在cmd下执行相应的命令 cd my.maven.demo # 初始化 git相关的文件 git init # 查看目录下的文件， 其中的.git 就是执行完 git init 生成的文件夹 ls -al # 获得输出 total 36 drwxrwxr-x 6 rd rd 4096 7月 15 16:21 . drwxrwxr-x 5 rd rd 4096 7月 15 09:28 .. -rw-rw-r-- 1 rd rd 1569 7月 15 09:51 .classpath drwxrwxr-x 7 rd rd 4096 7月 15 16:21 .git -rw-rw-r-- 1 rd rd 1832 7月 15 14:38 pom.xml -rw-rw-r-- 1 rd rd 542 7月 15 09:51 .project drwxrwxr-x 2 rd rd 4096 7月 15 10:53 .settings drwxrwxr-x 4 rd rd 4096 7月 15 09:28 src drwxrwxr-x 8 rd rd 4096 7月 15 15:08 target # 查看当前源代码的管理状态 git status # 获得输出  # 告诉用户当前处于分支 master上 On branch master # 告诉用户尚未提交任何源代码 No commits yet # 告诉用户尚未被git 管理的源代码文件或目录 Untracked files:  (use "git add <file>..." to include in what will be committed)  .classpath  .project  .settings/  pom.xml  src/  target/ # 添加 需要被管理的文件或文件目录 git add pom.xml git add src # 查看状态 git status # 获得输出 # 告诉用户当前处于分支 master上 On branch master # 告诉用户尚未提交任何源代码 No commits yet # 告诉用户这些文件发生了变更，需要被提交。因为是第一次提交，所有凡是通过 git add 添加的文件均被标记为发生了变更 Changes to be committed:  (use "git rm --cached <file>..." to unstage)  new file: pom.xml  new file: src/main/java/my/maven/demo/HelloMvn.java  new file: src/main/java/my/web/demo/Bootstrap.java  new file: src/main/java/my/web/demo/HttpHandler.java  new file: src/main/java/my/web/demo/Server.java # 告诉用户这些文件尚未被git 跟踪。.classpath .project .settings 都是 IDE自动生成的，target是maven编译生成的 # 这些文件都不是源代码，无需被git管理 Untracked files:  (use "git add <file>..." to include in what will be committed)  .classpath  .project  .settings/  target/ # 提交 git commit -m 'init git' # 查看提交日志 git log # 获得输出 commit 722a4257d751e01e65e2876b6e333fd464223a56 (HEAD -> master) Author: aaaaa <demotest@qq.com> Date: Mon Jul 15 16:36:12 2024 +0800   init git* |

**代码段 7-2 实际工程源代码管理示例**

# HTML

HTML 是整个互联网的基础，这么说毫不夸张。

## 8.1 HTML 页面

HTML页面相关的内容，通常称之为前端，即用户可以直接看到的。 而前面几个单元所学的内容，称之为后端，即用户不能够直接看到的，但能够控制用户看到些什么。一般来说，前端的主要任务是提供优秀的用户体验，也就是说不管用户打开浏览器，还是打开一款APP,用起来感觉很舒服，看着舒心，操作不反人类，各种按钮都符合人体工程学（这方面比较优秀的是Apple公司）。而后端的任务是实现各种复杂的控制，在正确的时间给用户正确的内容。

作为 Java 开发人员，需要掌握必不可少的 HTML 基础知识，能够通过 txt 文本文档自己写一个简单的 HTML 页面，如果还能加入一些javascript 脚本来实现简单的页面内容控制，通过浏览器打开该文件，能够达到预期的效果。熟悉 form 的提交，通过javascript实现form表单的异步提交。 了解css,div等样式相关的内容。<https://www.w3cschool.cn/> 是一个不错的学习网站，可以自行学习。

## 8.2 JavaScript

JavaScript（简称JS），是HTML页面中非常流行的一种脚本语言，属于动态语言的范畴，是前端开发人员必须精通的一门编程语言。事实上， JavaScript 跟Java 一点关系也没有，虽然名称类似。

JavaScript 在 1995 年由Netscape（网景）公司的Brendan Eich，在（Netscape Navigator）网景导航者浏览器上首次设计实现而成。后来Netscape与Sun Microsystems（太阳微系统公司，目前已经被甲骨文（Oracle）公司收购）合作，Netscape管理层希望它外观看起来更像Java，因此取名为JavaScript。

# Java 基础

通过阅读Java基础的书籍（推荐《Core Java Volume I--Fundamentals(11th Edition)》，作者: [Cay S. Horstmann]， 出版社: Prentice Hall，初学者随便再找一本中文的，中英文同时看，毕竟这东西是老外发明的，中文翻译过来有些内涵就变了），通过一门编程语言，熟悉面对对象的编程思想。每种编程语言都有其特点，但面对对象的编程思想是相同的。

熟悉 Java 基本数据类型、类、抽象类、接口、类的对象、重载、重写、构造函数、方法、类的继承（扩展）、类的多态、反射、代理、堆、栈、集合（Map、List、Set、Arrays）等。熟悉多线程、临界区、同步、线程安全等。

# 10. Java生态

## 10.1 数据库（DB）

需数据库相关的基础知识和实践能力，可以选取MySQL（一款开源免费的数据库，目前由Oracle 公司拥有）进行练习，包含以下内容。

（1）通过 JDBC 连接数据库进行 CRUD（Create（新增）, Retrieve（查询）, Update（更新）, Delete（删除））。

（2）通过MyBatis进行数据库的CRUD，自动生成相应的mapper，以及对应的接口以及XML 文件。

（3）了解数据库的事务（原子操作）及其实现机制，数据库连接池的概念，2阶段提交实现跨数据库连接的事务。数据库的索引实现机制。

（4）了解关系型数据库（例如，MySQL、Oracle）和非关系型数据库（例如，MongoDB）的区别。

## 10.2 缓存（Cache）

至少掌握一种缓存的使用方法，例如Redis。了解缓存和数据库的相互配合，实现数据的持久存储和快速访问。

## 10.3 Spring

有关Spring 框架的内容，是一个庞大的体系，有专门的书籍讲解其思想，不过还得一点一点使用，得有自己的体会，光看书籍的话还是有些迷茫，一边看书籍边通过程序实践才是王道。下面内容只介绍其核心思想。

**（1）重要概念。**Spring框架（Spring Framework）是 Java生态中非常重要的一个开源框架。Spring中有2个重要的概念需要掌握，控制反转和依赖注入。

1）控制反转（IoC, Invert of Control）。对象的使用者只需要进行对象的声明，对象的创建由Spring框架完成。

下面以一个开车到某个地方的代码段来进行说明，如果不使用控制反转，那么程序实现方式如代码段 10-1所示。

|  |
| --- |
| *class DriveCar {* // 实例化 Driver 类的对象 driver *private Driver driver = new Driver();* // 实例化 Car 类的对象 car *private Car car = new Car();    public String driveCarToSomewhere(String location) {  this.driver.startCar(car);  this.driver.addGas(car);  this.driver.drive(car, location);  } }* |

**代码段 10-1 常规的类对象的实例化示例**

Spring 控制反转思想示范，如代码段 10-2 所示。

|  |
| --- |
| /\*\*\*[1]在程序启动的时候，Spring框架会扫描所有添加了注解（Anotation）标记“@Component” 的类，记录下来  \* [2]当程序需要运行到相应的逻辑时，Spring 会将对应类（DriveCar）中标记了注解“@Autowire”的属性进行实例化（new） \*/ *@Component class DriveCar {* /\*\*\*Spring框架根据“@Autowire”的标记，对driver进行实例化操作（driver = new Driver())\*/ *@Autowire  private Driver driver;    @Autowire  private Car car;    public String driveCarToSomewhere(String location) {* /\*\* \* 从代码逻辑来看， this.driver会报空指针错误（NullPorintError），因为没进行实例化  \* 但事实上不会，因为Spring框架通过注解（Anotation） @Autowire 把 “driver=new Driver()” 这个代码逻辑自己实现了。\*这种思想就叫控制反转\*/ *this.driver.startCar(car);  this.driver.addGas(car);  this.driver.drive(car, location);  } }* |

**代码段 10-2 Spring控制反转思想示例**

2）依赖注入。Spring通过容器(Spring container)管理 Spring bean（什么是Spring 豆子？）。实现对象实例的注入。Spring 控制反转的思想，能够让程序在运行时（Runtime）正常运行，是通过依赖注入的方式实现的。在代码段7-2中，类DriveCar 依赖于类 Driver 和类 Car。类 Driver的实例 driver、类 Car 的实例 car 都是通过依赖注入的方式实现。

Spring bean 是那些被Spring容器管理的类的对象（实例化的类），容器（container）管理的对象叫豆子（bean），老外搞得挺形象，就好像一个大篮子（Spring 容器）里装满了各种各样的豆子（Spring bean，类的对象）。

Spring container, 可以理解为一个Map<String, Object>，就像是一个班级里头班主任手里的花名册，里头有每个人的名字，只要点到哪个名字，那么叫这个名字的Human的实例（某同学肉身）就会被找到。Spring container 就好像是一个班主任，班级外面的人（自己开发的应用程序）需要排练一个节目，那么先设计队形的时候，在那个空缺位子上写上某同学的名字（只声明变量，而不用立即把这个同学肉身叫来站在队伍里（实例化））。当开始排练的时候（程序运行起来了），排练老师一看这个空位子需要某个同学，立即找他们班班主任（程序运行时调用Spring容器）， 班主任立即把那个同学肉身给拎过来（Spring容器立即返回这个同学名字代表的Human实例），填充到排练空位上（依赖注入），节目排练丝毫不受影响。这就是控制反转、依赖注入的思想。

那么如果不使用依赖注入、控制反转，是如何做的呢？还是以节目排练来做比喻，节目开始设计队形时，排练老师说需要A班B同学，这时候A班B同学的肉身必须站到队形相应位子上，然后干等着，B同学也很无聊，总体来说也很浪费人力。

1. **单例（Singleton）**。在一定范围内（例如，Spring 容器）获取到的对象实例，都是同一个对象。下面通过一段代码段 10-3 来说明单例。

|  |
| --- |
| // 类 Car 的实例 car1 是通过 new Car() 的方式创建的 *Car car1 = new Car();* // 类 Car 的实例 car2 是通过 new Car() 的方式创建的 *Car car2 = new Car();* // 在运行时 debug的过程中，可以看到 car1和car2的内存地址是不一样的，// 也就是说 car1 和 car2 是两个实例， car1 == car2 为 false// car1 和 car2 属于同一个类 Car 的两个实例 |

**代码段 10-3 直接创建类的对象示例**

代码段 10-3 中的获取 Car 实例的方法，如果通过单例的方式，如代码段 10-4 所示。

|  |
| --- |
| *class CarFactory {* // 通过 private修饰符(modifier)，限制 CarFactory 外部无法直接访问 realCar *private static Car realCar;* // 通过public 修饰符, 作为 CarFactory类的唯一外部可访问的方法 *public Car getInstance() {  if (null == CarFactory.realCar) {* /\*\*\*通过 *synchronized*（同步）修饰符的限制，使得多线程变成单线程\*可以这样理解 *synchronized*，等同于“一夫当关，万夫莫开”。\*没有使用 *synchronized*，一个门开得很大，随便三四个人并排都能同时出入\*使用 *synchronized* 后，门被关得很严，每次只能出入一个人\*/ *synchronized (CarFactory.class) {  if (null == CarFactory.realCar) {  CarFactory.realCar = new Car();  }  }  }  return CarFactory.realCar; } }* |

**代码段 10-4 通过单例创建对象示例**

调用代码段 10-4 创建Car的代码如下所示。

|  |
| --- |
| *Car singleCar = CarFactory.getInstance();* |

通过代码段 10-4 的逻辑可以看出，调用 CarFactory.getInstance() ，无论调用多少次，最终返回的都是同一个类的实例，如代码段 10-5 所示。

|  |
| --- |
| *Car car1 = CarFactory.getInstance(); Car car2 = CarFactory.getInstance(); if (car1 == car2) {* // 程序将运行到这里， car1 和 car2 是同一个类的实例 *System.out.println("we are the same car."); }* |

**代码段 10-5 调用单例创建方法创建对象示例**

如果不做特殊配置（Spring容器可以通过配置使用多例模式）， 从 Spring 容器中获取到的Bean 都是单例模式。

（3）Spring家族。Spring 家族目前有 Spring-Core、Spring-MVC、Spring-boot、Spring-Cloud等多个框架，形成了一个框架家族。core主要提供Spring 容器，来管理Spring bean。mvc 主要提供进行web开发相关的 mvc功能。boot提供了一个servlet容器插件，解决了无需再打war包再由servlet容器（例如Tomcat）启动程序的问题。cloud则提供了一系列跟微服务相关的组件，如负载均衡、熔断、限流、降级等相关的组件。

## 10.4 MVC的概念

MVC（Model（模型）、View（视图）、Controller（控制器））。Spring框架中通过Spring MVC 模块，来实现Web 页面逻辑的MVC分离的概念。

先通过一个HTML页面来解释下以上3个基本概念， 页面源代码如代码段 10-6 所示，假定这个页面的名称为“客户订单页面”，文件名称为 order.html。

|  |
| --- |
| *<html>  <header>  <meta charset="utf-8">  <title>客户订单页面</title>  <script src="https://www.clarity.ms/s/0.7.32/clarity.js"></script>  <link rel="stylesheet" type="text/css" href="https://csdnimg.cn/public/sandalstrap/1.4/css/sandalstrap.min.css">  <style>  div, span, input, button {font:18px Verdana}  div, button {font-weight:bold;text-align: center;}  span {text-align: center;}  input {width: 300px;}  select {width: 308px;}  </style>  </header>  <body>  <div style="text-align: center;margin-top: 15%;" >  <p align="center" style="font:22px Verdana; font-weight:bold;">${user\_name}的订单</p>  <form id = "my\_form" style="margin-top: 2%" method="post" action="/adm">  <div align="center" style="margin-bottom: 0.5%">  <span>商品名称:</span>  <input id = "order\_product" name = "order\_product" type="text" value="一本学MVC的书">  </div>  <input id = "t" name = "t" type="hidden">  </form>  <div align="center"style="margin-bottom: 0.5%">  <span>商品价格:</span>  <input id = "order\_price" name="order\_price" type="text" value="28.80元">  </div>  <div align="center">  <span style="vertical-align: top;">购买时间:</span>  <input id = "order\_time" name="order\_time" type="text" value=" 2024-03-15 13：57">  </div>  <div align="center" style="margin-bottom: 0.5%">  <span>支付金额:</span>  <input id = "order\_pay" name="order\_pay" type="text" value="25.40元">  </div>  <div align="center" style="margin-bottom: 0.5%">  <span>联系人:</span>  <input id = "order\_customer" name="order\_customer" type="text" value="张先生">  </div>  <div align="center" style="margin-bottom: 0.5%">  <span>联系电话:</span>  <input id = "order\_contact" name="order\_contact" type="text" value="158\*\*\*\*8803">  </div>  <div align="center" style="margin-bottom: 0.5%">  <span>配送地址:</span>  <input id = "order\_address" name="order\_address" type="text" value="银河系火星某基地6338号实验室">  </div>  </div>  <script>  var order\_price = document.getElementById("order\_price");  order\_price.onchange = function () {  document.getElementById("t").value = CryptoJS.MD5(psw\_input.value).toString();  };  </script>  </body> </html>* |

**代码段 10-6 HTML页面源码示例**

浏览器渲染后的效果如图 10-1 所示。



**图 10-1 HTML页面源码渲染效果示意图**

控制这个页面的逻辑为，当用户输入 <http://abc.def/user_id/order> 时，即 URL 以“/用户ID/login”结尾时，服务器返回给浏览器的内容如代码段 10-6 所示，其中的${user\_name}替换成用户的名称，经浏览器渲染后用户看到的内容如图10-1所示。

MVC 的提出是有历史背景的，最初的Web页面类似于JSP（Java Server Pages， 即Java 服务端页面， 微软的C#（读作 C sharp）有类似的 asp页面）的方式，在服务端Web页面的展示方式（样式，展示的字体、字号、颜色、展示的位置等）、数据（可以想象为table中的数据）、以及生成数据的逻辑（在什么情况下展示什么数据，可以想象为switch case）是混杂在一个源代码文件里的，修改起来很容易出现错误（只要是人，都会犯错误），这样导致软件的可维护性、稳定性都比较差。

MVC 概念的提出，是为了将不同种类的内容，分割至不同的源代码文件中，当用户发起请求后，将各种文件内容按照一定的而规则进行融合，形成一个用户可见的页面。这样用户看到的是一个综合体，而软件人员维护的是一个一个独立、清晰的源代码（数据）文件，提升了软件整体的可维护性和系统的稳定性。

1. **Model，即数据模型。**与代码段10-6的页面对应的数据模型，即用户订单的相关信息，可以拆分为几个Java类，代码段 10-7 为订单信息数据模型。

|  |
| --- |
| *class OrderInfo {* // 订单ID *private String orderId;* // 用户ID，指哪个用户下的订单 *private String userId;* //商品Id *private String skuId;* // 支付途径 *private String payType;* // 支付金额 *private String payCash;* // 订单创建时间 *private Date createTime;* // 支付时间 *private Date payTime; }* |

**代码段 10-7 订单信息 Model**

代码段 10-8 为商品信息数据模型。

|  |
| --- |
| *class SkuInfo {* // 商品Id *private String skuId;* //商品名称 *private String skuName;* // 商品价格 *private String price;* // 商品重量（kg） *private Ingeter weight;* // 库存数量 *private Integer stockNum; }* |

**代码段 10-8 商品信息 Model**

代码段 10-9 为用户信息数据模型。

|  |
| --- |
| *class UserInfo {* // 用户ID *private String id;* // 用户姓名 *private String name;* // 用户地址，进行订单配送 *private String address;* // 用户手机号，配送订单联系用户 *private String mobile; }* |

**代码段 10-9 用户信息 Model**

在关系型数据库中，一般通过数据建模形成一个表的表结构，这个表结构也相当于是一个数据模型。在XML 文档中，对数据的定义，即DTD，也是一种数据模型。注意，数据模型与数据本身的区别， 以地址信息为例，如表10-1所示。

**表 10-1 数据模型和数据本身的区别**

| **地址信息数据模型** | **地址数据** |
| --- | --- |
| 国别-省-市-区-县-街道-道路号-小区名称-楼栋号-单元号-房间号 | 中国广东省广州市花西区莲花街道光明路华侨小区2号楼3单元405房间 |

**（2）View，即视图**。用户可见的内容，或者说将多个Model的数据组合成一个用户需要的页面。可以认为代码段 10-6 的页面（图10-1）就是一个视图，体现为一个 HTML页面模板（即order.html），其中的几个变量商品名称（order\_product）、商品价格（order\_price）、支付金额（order\_pay）、联系人（order\_customer）、配送地址（order\_address）等， 需要根据用户的ID、商品ID、订单ID等从几个数据模型（OrderInfo、SkuInfo、UserInfo）中查询后获取，然后填充在order.html相应的位置，展示给用户。

在关系型数据库中，也存在视图的概念， 视图不存储数据，而是在几个数据表（模型）中部分字段（模型属性）进行组合后，形成的一种数据展示形式，即视图。

1. **Controller，即控制器。**控制器的功能就是实现当用户输入以“/用户id/order”结尾的URL时，返回如图 10-1 所示的页面。代码逻辑如代码段 10-10 所示，实现方式以spring MVC模块为例。

|  |
| --- |
| /\*\*\*告诉spring 框架，这是个控制器，需要将相关的HTTP请求映射至这个class  \* 以便这个class 处理相应的请求逻辑\*\*/ *@Controller class OrderController {* // 告诉spring 框架，这块代码处理以 “/用户id/order” 结尾的 HTTP 请求 *@RequestMapping("/{userId}/order")  public ModelView getOrder(String userId) {* // 对应 order.html的模板，将在模板中填充相应的内容，形成客户的订单信息 *ModelView modelView = new ModelView("order");* // 根据用户 ID,查询订单 ID, // 根据订单 ID 中的商品 ID 查询商品信息// 根据订单 ID 中的用户ID 查询用户信息 *return modelView;  } }* |

**代码段 10-10 订单信息处理 controller**

至此，对于MVC的概念应该有一些理解了吧。MVC是一种设计模式。什么是设计模式？可以理解为套公式，有一套方法论（思想），对于不同的问题采用不同的程序实现方法，这个跟编程语言无关。这种方法论，被称之为设计模式，推荐看这本书 《Design Patterns: Elements of Reusable Object-Oriented Software》，作者 Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides， ISBN 978-0201633610， 出版社 Addison-Wesley Professional，网上有PDF版的，可以打印出来多看几遍。

## 10.5 服务调用

**（1）RESTful 。**RESTful（REST，Representational State Transfer，表述性状态转变）是一种 HTTP 接口的封装风格，本质上还是 HTTP请求，不过对于什么样的业务逻辑（获取信息、提交数据、修改数据等）该用哪种HTTP Method，以及对于 HTTP 接口的地址规则进行了规范，这样方便大家交流，形成一些约定俗成的最佳实践，能给跨团队合作带来很多好处。

**（2）RPC。**RPC，即Remote Procedure Call（远程程序调用）。在了解RPC之前，在操作系统中，有一个IPC（Interprocess communication，进程间通信）的概念，即单个操作系统中的多个进程的通信问题，这个可以看讲操作系统方面的书籍进一步了解。

这里提到IPC，IPC解决的是单个操作系统上多个进程通信的问题。那么，如果有2个进程，分布在2个不同的操作系统中（两台主机，网络是连接在一起的），这两个进程需要进行通信，如何解决？这就是RPC 需要解决的问题。当然了，当前很多微服务之间，都通过RESTful接口（HTTP请求）进行通信，不也实现了RPC？确实这么说没有问题。

RPC 是让开发人员像调用本地方法（本地类库）一样的体验，来发起远程服务调用的，即隐式地进行远程网络请求， 而HTTP对于开发人员来说，是显示地发起远程网络请求。RPC本质上是将HTTP、UDP、TCP等远程网络请求进行一定的封装，使得开发人员感觉在调用本地类库，而无需去了解 HTTP协议、UDP、TCP等soket通信的一种简便的方法，而且这种封装稳定可靠，有利于对服务进行大规模部署、扩容，是互联网服务不可多得的一种好的服务调用方法。

目前，服务之间的调用，除了RESTful（HTTP请求）外， RPC是另一种方式，一般来说，若涉及跨越数据中心（例如需要走公共互联网），一般采用RESTful的方式。如果是一个大型产品内部不同模块之间（多个服务都部署在一个逻辑内网中）的调用， 则采用RPC较为合适。因为RPC 一般是基于 TCP socket通信的，从调用开销（开销，可以理解为买东西时外面的包装材料，包装材料越多则买东西的开销越大）上来说要比RESTful要小一些，从而性能要比RESTful要高一些。

目前主流的RPC框架有Dubbo、Thrift、gRPC，这三种都有团队在使用，具体使用哪种取决于团队的技术路线和个人的爱好。

1）Dubbo。dubbo（发音：搭宝）最初由alibaba开发，2018年阿里巴巴已将其开源贡献给了阿帕奇基金会，可通过<https://dubbo.apache.org>了解。优点，使用简单方便，容易上手。

2）Thrift。 Thrift（发音：[θrɪft]），Thrift最初由Facebook公司在2007年开发，目前也将其开源贡献给了阿帕奇基金会，可通过 <https://thrift.apache.org/> 了解。优点为跨语言，支持C++、Java、Python、PHP、Ruby、Erlang、Perl、Haskell、C#、Cocoa、JavaScript、Node.js、Smalltalk、OCaml、Delphi等之间互相调用。

3）gRPC。 gRPC是由Google基于HTTP/2协议开发的一种RPC框架。

# 11. Netty

Netty 作为Java 生态中一个重量级的NIO 开源框架，初学者无需掌握，但对于做网络相关的高级开发人员来说是必须掌握的内容。下面提几个基本概念。

**（1）NIO**。NIO 叫做 NewIO，或者叫Noblock IO，即非阻塞式IO。提到非阻塞式IO，必然有阻塞式 IO，即Blocking IO。给一个场景，写一段程序，将10GB的数据写到本地磁盘，代码段如 11-1 所示，发起请求后，操作系统说你等着，于是你的程序就卡顿在那个wirte的位置。

举个例子进行类比，客户在人特别多的小饭馆点了个餐，特别饿，但餐馆老板太忙，管理也不规范，反正也没小票，过一阵子客户询问老板餐是否做好，老板回答说我催催后厨。这样一来，客户一直在为吃饭操心，也没心思玩手机或干别的（相当于餐厅客户的心思被做饭大厨给阻塞了）。

|  |
| --- |
| *class BlockIO {  Data bigData= new Data(10GB);  File file = new File("/tmp/test.dat");  file.open();* // 由于数据较大，程序会在这个位置卡顿一段时间，需要等待 *file.writeBIO(bigData);  file.close(); }* |

**代码段 11-1 BIO 示例**

非阻塞式IO，即不管任务有多繁重，程序运行总是立即返回，然后给你一个ticket，你可以随时查询， 代码段如 11-2 所示。类似你在五星级大饭店下单了一顿大餐之后，服务生优雅地给你打印一个小票，让你坐那儿等着，还给你糖果零食小吃随便吃，你可以拿小票催服务员查询大餐啥时候做好（或者在线App里随时看，类似于麦当劳App在线单，会显示制作中、已完成、请去餐之类的），这时候你可以放心地玩手机，或者跟朋友闲聊（相当于你自己的心思并没有被做饭大厨阻塞）。

|  |
| --- |
| *class NoBlockIO {  Data bigData= new Data(10GB);  File file = new File("/tmp/test.dat");  file.open();* // 立即返回， 不卡顿，不等待 *Promise promise = file.writeNIO(bigData);* // 会告诉你，正在处理中、已完成、已失败 *promise.get();  file.close(); }* |

**代码段 11-2 NIO 示例**

说到NIO，就涉及到操作系统对IO任务队列的处理方式，涉及到2个概念， select, epoll。Ubuntu系统可以通过以下命令简单学习，深入学习还需要阅读操作系统的书籍。

|  |
| --- |
| *man select man epoll* |

**（2）TCP 协议。** Netty底层有对 TCP、UDP等 TCP/IP 协议族 socket通信的封装。 什么是TCP，好像跟我日常的编程关系不大？TCP 即 Transmission Control Protocol， 传输控制协议，是当今整个网络通信的基础，日常我们浏览网页使用的 HTTP协议就是基于 TCP协议实现的。整个TCP 协议族，可以阅读《*TCP*/*IP* *Illustrated*, Volume 1: The Protocols, Second Edition 》作者Gary R. Wright，W. Richard Stevens。如果做网络变成，这本书是必读的，不但读，还得能讲得头头是道。知其然，知其所以然。知识是一片海洋，越学会越觉得自己渺小。

# 12. Java 内存模型

Java 的底层是C语言， C语言的底层是操作系统， Java的内存分配、管理模型，不能说跟 C 没有关系。这里涉及到对 Java 中 volatile 修饰符的理解， C 语言中也有 volatile 关键字， 理解其含义，从 C 语言出发也是一个不错的选项。进一步理解 Java 中的 synchronized 修饰符的真正含义和底层实现，还有internal 关键字。推荐阅读《Thinking in Java》作者 Bruce Eckel。

# 13. 操作系统

学习操作系统的知识可以阅读《Computer System: A Programmer's Perspective》。这里的每一个点都是重点，对于开发人员理解软件运行内部机制来说很重要。

操作系统对 IO 的处理（ 磁盘 IO，网络IO， NIO, BIO, epoll）、多线程处理（Thread、CPU时间片），线程同步（JDK中有一个 ThreadLocal类、synchronized、Lock、Monitor），进程的 fork，临界区，PV操作、寄存器（java 中有 register）、用户态和内核态（零拷贝、内存映射、数据区、代码区）、堆栈。

# 14. 算法

算法（Algorithm）是比较基础的内容，也是软件的核心，决定着软件的性能（单位时间内对同一个任务的处理能力），可以看看基础的书籍，同时可以在刷题网站（<https://leetcode.cn/>）做做算法题，能够提高自己的思维能力以及实践能力。算法是软件开发人员很重要的一种基础能力，需要得到重视。

# 15. 软件架构

## 15.1 架构设计基本原则

软件架构（software architecture），即如何设计软件系统，使得在满足产品（用户）需求的基础上，使软件系统便于维护（后期代码修改）、横向扩展（当用户访问量增加时，如何提升系统处理能力）。

**（1）抽象能力。**一个软件系统，抽象能力发挥得越好，那么同样的功能所需的代码量（行数）将会越少。在进行接口设计的时候，一般依赖于抽象的接口（例如List），而不依赖于具体的接口实现（ArrayList）。

**（2）开闭原则**（Open-Close Principle，简称OCP）。开闭原则是指一个软件实体（类、模块、方法等）应该对扩展（扩展一个类、接口）开放，对修改（直接修改当前的类、接口）关闭。因为直接修改当前的类、接口，满足了新的需求，但可能会影响到已经在运行的历史系统，导致某些地方出现问题。而扩展一个类和方法，因为原来的类、方法未做改动，所以对历史业务不会造成影响，同时扩展的类又可以满足新的业务需求。

**（3）模块化。**模块化，就是把不同的功能放在不同的模块中，相同的功能放在同一模块中，便于后期维护。同时，部分模块可以被其他系统复用，减少重复开发的工作量。

**（4）微服务化。**由多个服务（进程）协同完成一项任务，这样需要升级维护某一个服务时，其他服务不受影响（系统影响面较小），而不是搞一个超级无敌大的服务（单体服务，Monolithic Service）。在单体服务模式下，当需要更改一个很小的模块时，整个服务都要停止重新启动，系统影响面较大。当然，这也不是绝对的，很多传统企业因为人力不足，采用单体服务可以节约运维成本。

**（5）分层设计。**TCP/IP协议参考模型就是这种思想的一种很好的实践。一般有网络层（公网、内网、VPN网等）、容器层（OS、JDK、Docker）、数据层（MySQL、Redis）、中间服务层（数据访问、缓存控制、权限控制服务、客户管理服务、互联互通服务等）、应用层（桌面Web端、IOS App、Android App、H5、Websocket、第三方开放API）。

**（6）高可用原则。**任何一个节点都不能发生单点失败，即任何一个物理服务节点发生故障，都有与其功能相同的节点迅速接管这部分任务。从系统外部来看，似乎没有发生任何故障。

**（7）容灾。**一般在部署时需要进行多数据中心备份，还需要考虑数据的备份和恢复。

## 15.2 软件全生命周期管理

本节内容涉及到软件工程方面的知识。一个软件从设计开始直到后期的迭代更新的整个生命周期，是一个长期的过程，下面进行详细介绍。

**（1）确定需求和目标。**需要确定软件系统所需要解决的问题是什么，划清问题的边界，哪些问题不是所开发软件系统需要去解决的。需求有功能需求（例如能够查看订单）、性能需求（例如，点击某链接响应时间不应该超过2秒，系统可支持1000人同时在线使用）、软件需求（例如，支持在ARM Linux系统上运行、支持权限的在线更改和配置）。

**（2）概要设计。**在确定了基本需求之后，就可以开始软件系统的概要设计了，概要设计完成软件系统的大致功能模块，每个模块的功能边界即可，不用过多考虑内部细节。类似于画素描的时候，先画出一个大概的轮廓出来。

**（3）详细设计。**在概要设计的基础上，对每个功能细节进行丰富，需要考虑实际开发的工程实现，每个细节都是能够落地的，除了写代码之外的所有细节，都是需要在详细设计文档中体现的，例如采用哪种数据库，采用哪些开源组件，服务交互是用HTTP还是TCP，数据库访问到底有哪些接口。在这个过程中，可能会回退到需求确定阶段，因为发现某项需求不明确，无法做出详细设计。

**（4）软件开发。**这个阶段需要按照前期的详细设计输出的文档，进行软件开发工作，一般分为以下几个步骤。

1）UI 原型设计。即用户可见的部分，是什么页面、什么按钮、怎么交互，有没有跟第三方的交互，一般由负责产品功能的人员来做，需要用到一些类似于Axure的原型设计工具。

2）UI 静态页面（美术、网页渲染）设计。按照功能要求，设计出 div、css、图片等静态页面，这部分对HTML 的要求较高，需要较为专业的前端人员来做，而且需要具备一定的美术功底，具备一定的审美能力。

3）Web前端代码和后端代码开发。如果采用前后端分离的设计，则前端人员在静态页面设计的基础上，进行前端代码开发，同时调用后端人员提供的接口，实现数据的填充。后端开发人员需要参考详细设计文档以及UI静态页面的设计，通过接口（一般为RESTful API）提供方便前端人员使用的数据。

前后端开发人员需要密切配合，后端人员提供的接口需要有明确的接口文档。

**（5）单元测试。**不管是前端人员还是后端人员，都需要对自己负责的代码各个模块提供单元测试，保证代码功能正常。一般来说，单元测试与软件开发是同时进行的，开发完一个小模块，即可进行单元测试代码，maven工程中 src/test目录下的代码就是用来进行单元测试的。

**（6）系统测试。**测试部分一般有有系统功能测试、接口功能测试和性能测试。

1）系统功能测试。功能测试人员按照详细设计中的功能描述，列举功能测试点及测试方法。测试开发人员提供的系统的功能是否如设计文档描述的内容达到预期效果。例如，点击登录，可正常登录。点击订单提交，可提交订单等。测试要点基本上围绕用户可使用、可见的部分，一般包括 web 页面的使用，App 界面的使用等。可以人工手动进行测试，也可以借助自动化测试工具（如 Selenium 等）。

2）接口功能测试。接口测试人员按照后端人员提供的接口文档，通过接口测试工具（例如PostMan, JMeter等）构造接口入口参数，查看输出是否正常。通过接口测试人员的测试，判定软件接口功能与接口文档描述的一致性。

3）性能测试。性能测试用于判定系统的性能指标是否能够达到预期的设计指标（例如，可保证1000人同时在线使用，某功能的响应时间不超过1秒）。对于大型的软件系统，可先进行各个分系统的性能测试，再进行整个系统的性能测试。由于最终用户使用的系统一般为集群（多台服务器），在进行性能测试时，一般会先进行单台服务器的性能，再测试集群的性能。性能测试一般会使用到 JMeter（Apache开源免费产品）、LoadRunner（HP公司旗下商业产品）之类的工具，通过脚本来模拟一定的并发用户量（例如 1000 个人同时使用系统），来看整个系统的吞吐量、响应时间以及并发量。

以上测试过程中，存在的问题，一般记录在专门的测试系统里（例如 TestDirector、 Jira等），每个记录为1个bug，需要对存在的问题进行详细描述，包括时间、位置、发生的前置条件、产生的问题、导致的错误的数据等，必要时辅助以图片、视频进行说明，开发人员按照测试系统中记录的bug，逐个进行调试并解决。

**（7）系统上线。**系统测试完成后，整个系统需要部署在生产环境（用户真正使用的环境为生产环境，测试时使用的环境为测试环境，一般来说测试环境和生产环境是隔离开的）。部署完成后，一般来说由测试人员模拟真实的用户，进行“冒烟测试”（即保证系统基本功能可用），然后向真正的用户开放。

**（8）系统更新。**生产环境下，如果有部分功能需要更新，则由开发人员提供部署安装包，由运维人员进行部署，大型系统（例如微信）一般采用灰度发布的模式。所谓的灰度发布，即新老系统并存，新功能只选择性地开放给一小部分用户（拿用户当小白鼠），这样万一新功能有问题，影响的只是一小部分用户。若经过一段时间的运行，这一小部分用户没出什么大问题，则逐步开放给所有用户。

**（9）系统运维。**系统在交付使用（或交付给用户）后，需要专门的运维人员进行系统运维。软件系统是一个动态系统，需要随时关注其运行状态，例如磁盘是否被占满，CPU利用率是否过高（系统计算任务负载高或程序bug导致）、内存利用率（有些应用比较消耗内存，比如将一个2GB的大文件全部加载至内存），磁盘IO（是否有磁盘读写的任务处于等待状态）、网络IO（网络带宽是否被消耗殆尽）、集群负载是否均衡（部分节点负载高、部分节点闲置）。

所有这些都需要通过监控系统来完成，监控系统具备以下能力。

1）收集数据。监控系统收集业务软件系统运行的各种数据，并将一定时间内的历史数据存储下来，方便进行分析。

2）数据可视化。监控系统将近期的数据进行数据可视化，展现形式有饼图、二维曲线等。

3）数据预警。监控系统中可设置各种阈值，当某个指标达到设定的阈值时，系统即刻以声光电等形式发出警告，也可以发送mail、短信，拨打制定电话等。

4）集群管理。监控系统提供一定的集群管理功能，方便运维人员批量操作几十至几百台服务器。

5）关键数据备份。通过定时任务（例如crontab）等，定期备份业务系统关键数据，一旦发生灾难（例如地震、火灾、网络黑客攻击、数据库系统崩溃、停电、停网等）后，方便异地恢复数据，保证为用户提供的服务不发生变化。例如，谷歌在全球的多个数据中心部署有其服务，当某个数据中心发生灾难后，可迅速将关键数据通过异地容灾恢复至其他数据中心，同时将用户的请求转发至相应的数据中心上，这样用户基本上感觉不到数据中心服务器宕机造成的影响。

**（10）配置管理。**配置管理（Software configuration management ，简称SCM）是软件开发过程中通过版本控制和变更控制的一套程序，以保证交付给客户的系统能够稳定运行。

举个例子，某关键系统运行了20年，交付了多个客户，客户使用中的系统存在使用5年前、10年前以及最新版本的多个软件版本。当使用10年前软件版本系统的客户需要升级一个子模块时，这时候该怎么做？这就是SCM 发挥作用的时候了。从SCM系统中可以获取10年前指定软件版本（V3.2.1193）对应的各个子系统的软件版本号以及部署安装包、安装手册、用户手册、运维手册等一系列软件和文档全部精确无误地获取到，然后形成一套交付方案，由实施人员在客户的硬件环境中进行实施。

# 16. 系统架构

系统架构（Architecture），或者叫做企业架构，主要是设计思想，是方法论层面的东西，通过一套比较科学的方法论来指导系统的设计，尽可能地降低风险，减少损失。需要设计人员从软件中脱离出来，站在更高的视角（宇宙视角）来看整个系统，从整个系统来考虑问题，自上而下地思考现实中的问题。软件是用来实现价值的，不是独立存在的，软件需要部署在硬件上，硬件又被分割在很多物理位置。很多现实中出现的问题不是单单某个软件系统的问题，举个例子，软件系统A单独运行没有问题，软件系统B单独运行也没有问题，但如果部署在一个网络内（或一台机器上），就会出现一些莫名其妙的问题。

系统架构设计思想，被称之为Enterprise Architecture，简称EA。可以看看 TOGAF（The Open Group Architecture Framework）和 DoDAF（Department of Defense Architecture Framework， （美国）国防部体系结构框架）相关的内容。

系统架构设计，不只局限于软件系统，还会关注跟软件系统进行交互的周围系统、环境、人等因素。

客观世界中的系统，是一个物理的客体，是不变的，但对于不同的人来讲，因为关注点不同，所以对与同一个系统的理解会有不同，这样就衍生除了数据架构、业务架构、功能架构、软件架构等不同的视角。对于一个大型系统来讲，需要满足不同的利益相关者（stockholder）的要求，例如负责管理数据的希望看到整个系统的数据流是如何流动的，负责管理业务功能的希望看到整个系统的业务是如何运作的，在什么节点有哪些参与方需要参与；负责软件系统的希望看到系统都是用了哪些开源组件，对应的开源协议是否合规，是什么操作系统，什么数据库，是不是微服务架构，上层应用和中间层的划分是否合理，运行是否稳定，软件维护是否方便快捷；负责财务审计的管理人员希望看到整个项目在什么里程碑节点给谁支付什么款项，各种财务开支是否符合财务管理规定。

## 16.1 DoDAF 视图

本文选取DoDAF 的视图，来说明不同的利益相关者从不同的角度对同一套系统进行分析、理解及其关注点。在DoDAF中描述了8 种视图，‌基本上代表了不同的利益相关者。

**（1）全局视图（All Viewpoint，AV）。**全局视图提供对整个架构描述的顶层概貌，包括架构描述的范围和背景。

全局视图，是一种战略性的愿景视图，描述整个系统的战略规划，将在什么样的背景下，在什么样的范围内应用，将达到什么样的战略目标，抽象度较高。类似于我们写论文里的摘要部分，高度概括整个系统。

全局视图是负责制定公司/企业/团体发展战略的管理人员所关注的信息，例如投资人、董事会成员等。举个例子，某创业公司，就以做出ChatGPT 的 OpenAI为例，潜力巨大，这时候企业资金消耗量巨大（买高端GPU，招揽高级人才进行人工智能模型的训练），公司准备向某投资机构进行宣讲，看看是否能够拿到风险投资。那么这个时候，就需要一个 ChatGPT 系统的全局视图，告诉投资人这个系统的整体架构，能干什么，能解决什么大的问题，有什么先进性等。

‌ **（2）能力视图（Capability Viewpoint，CV）。**能力视图描述实现组织愿景所需的能力。例如，企业/团体/组织的领袖制订了组织的2个愿景，第一个：能上天，第二个：能入地。这个时候某个下属部门做了一套系统，告诉领导，这个系统具备“能入地”的能力。再举个例子，阿里巴巴的淘宝和速卖通实现了“让天下没有难做的生意”的愿景，具备使买卖双方在线上进行安全交易的能力。

能力视图类似于我们做软件系统的功能视图，不过需要高度概括，“能力”比“功能”维度更高，更抽象。例如软件功能里有一项“支持用户登录”， 在能力层面可以描述为系统具备对用户使用权限进行控制的能力。软件系统中的第三方开放接口功能，在能力层面可以描述为系统具备与第三方系统互联互通的能力。

能力视图是公司/企业/团体的高级管理人员所关注的信息，例如CEO，总经理等。以ChatGPT为例，该系统具备与人类进行日常对话的能力。

**‌ （3）数据与信息视图（Data and Information Viewpoint，DIV）。**数据与信息视图关注业务信息需求和结构化业务流程规则。

数据与信息视图，主要关注系统所使用和产生的数据的存储、流向、管理等问题。例如，在什么阶段输入什么数据，输出什么数据，数据如何安全管理，符合保密合规要求等。

数据与信息视图是管理数据的利益相关方所关注的信息，例如一个团体中管理数据存储、备份、合规的管理人员。

以ChatGPT为例，需要存储海量的来自全球互联网的文本、视频、图片数据，部分数据还需要进行人工标定（标定数据），方便机器学习。还需要存储大模型数据。

‌ **（4）运行（操作）视图（Operational Viewpoint，OV）。**运行视图描述组织、任务或活动及其间的信息交换。

运行（操作）视图，即在什么样的业务场景中，该系统运行起来后与周边的人、系统如何交互，会对实际的业务有什么样的贡献和支持。什么样的角色（可以是一个人、机器、其他系统）需要参与到该系统中，最终会有什么效果。

运行（操作）视图一般是公司/企业/团体中负责业务的管理人员所关注的信息。例如，为一个生产制造企业的库房管理人员提供了一个物料管理系统，那么库房管理人员需要弄清楚怎么用这个系统，如何和现有的库房中的物料交互，出库、入库登级等，定期盘查，人员24h换岗、消防管理，温度湿度控制等。

以ChatGPT为例，需要有安装有GPU的服务器机房，部署已经训练好的模型，接收线上用户在线输入数据，将模型输出的结果反馈给用户，同时还要在内容输出方面符合各个国家的法律，同时还需要通过监控系统及时处理不合规信息，提供给模型训练人员，优化线上模型。

**‌ （5）项目视图（Project Viewpoint，PV）。**项目视图用于管理项目计划如何组合成具有前后承接关系的投资组合计划。

使用/开发该系统的项目节点（甘特图、里程碑、成果物等），以及人力、资金的投入，前置资源的协调安排、风险点及其控制等。在什么时间节点交付那部分系统，用户使用是否满意，是否可以进行下一个环节的判定、评审等。

项目视图是进行项目管理的人员所关注的。以ChatGPT为例， 项目里程碑如下所示。

1）在2017年基于google的Transformer模型进行探索，能够给出探索实践研究报告，为后续发展指明方向，预计投入XXX亿美元，研发工程师XXX名；

2）2018年6月前发布GPT-1 模型 ，能够进行生成式训练，进行自然语言的生成（通俗理解，预测一句话接下来该说什么），预计投入XXX亿美元，研发工程师XXX名， XXX日向投资人董事会汇报；

3）2019年2月前发布 GPT-2 模型 ，使用大规模网页数据集WebText进行预训练，能够进行无监督多任务学习（无监督学习，通俗理解，无需实现用一张图片告诉AI 这个是鲸鱼，再让AI看下一张图片是什么动物。而是给AI大量的图片，让AI自己学习各种图片是什么内容），预计投入XXX亿美元，研发工程师XXX名，XXX日向投资人董事会汇报；

4）2020年5月前发布 GPT-3 模型，能够进行上下文学习（通俗理解，说话的上下文，通过之前说的话，对当前说的话能够更加准确），预计投入XXX亿美元，研发工程师XXX名，XXX日向投资人董事会汇报；

5）2022年3月前发布GPT-4模型，能够进行推理，支持根据输入的文本和图片进行推理，预计投入XXX亿美元，研发工程师XXX名，XXX日向投资人董事会汇报。

‌ **（6）服务视图（Services Viewpoint，SvcV）。**服务视图描述为作战活动提供支持的系统、服务和互连功能。

服务视图，这个跟软件的各个服务部署方案类似，即整个系统都有那些子系统（可以理解为一个或一组进程），这些服务之间是如何互相依赖互相交互的，谁调用谁，哪个服务先启动哪个服务后启动。服务之间调用是什么协议，需要什么样的网络策略。各个服务部署的物理/逻辑位置，一般容易出现的问题及恢复时间、手段等。服务如何升级、扩容、缩容，灾备等。

以 ChatGPT 为例，服务包括大模型调用服务，人机交互Web服务、负载均衡服务、用户认证授权服务、用户计费费服务，用户充值服务。

**‌ （7）标准视图（Standards Viewpoint，StdV）。**标准视图定义控制系统各部分或元素间组合、交互和互依赖性的规则。

标准视图关注整个系统的一些关键的数据标准、交互规则标准等，即该系统的研制/运行，都需要哪些标准和规范（企业标准、行业标准、国家标准、国际标准）。在大型企业中，一般都有专门负责进行标准制订的部门，同时作为一套系统来讲，创建新的标准也是一种创新成果。在整个系统中遵循同一套标准，这样能够使不同的子系统、模块在进行互联互通时较为顺畅。数据交换和共享有统一的标准，服务交互也有统一的标准。

以 ChatGPT 为例，标准包含《文本生成正确率评判标准》，《通用人工智能智能成熟度评价标准》，《人机对话上下文关联成熟度评价标准》。

‌ **（8）系统视图（Systems Viewpoint，SV）。**系统视图关注自动化系统、互连通性和系统功能方面的信息。

系统视图，关注的是整个系统都包含哪些子系统，整个系统都和哪些其他（第三方）系统进行交互。子系统之间的关系是什么，子系统之间是如何交互和关联的。

以 ChatGPT 为例，其子系统有全球主要语言分类子系统、模型训练子系统、在线模型子系统、人工智能合规监督子系统、运行监控子系统等， 在线支付子系统，用户管理子系统。

## 16.2 系统架构工具

企业架构设计的工具，目前有 Enterprise Architect （EA），详见 <https://sparxsystems.com/enterprise-architect/> 。该软件遵循 OMG（Object Management Group， 对象管理组织）组织制订的 BPMN （Business Process Modeling Notation）标准。这套标准规范了企业架构建模过程中各种数据的形式化表达方法。BPMN本质上是一种 UML 建模标准。