|  |
| --- |
| 面试知识 |
| Java后台开发 |
| 本文档整理了java面试相关的知识点，以便自己在平时查漏补缺。同时方便以后再次面试时，可以随时查阅复习。 |

2018-10-14

[面试知识 0](#_Toc527321199)

[计算机基础 8](#_Toc527321200)

[十进制转二进制 8](#_Toc527321201)

[整数 8](#_Toc527321202)

[小数 8](#_Toc527321203)

[原码、反码、补码 8](#_Toc527321204)

[原码 8](#_Toc527321205)

[反码 9](#_Toc527321206)

[补码 9](#_Toc527321207)

[浮点数的二进制表示方法（IEEE754标准） 9](#_Toc527321208)

[补码运算 10](#_Toc527321209)

[加法运算 10](#_Toc527321210)

[减法运算 10](#_Toc527321211)

[乘法运算 11](#_Toc527321212)

[除法运算 11](#_Toc527321213)

[Java基础知识 12](#_Toc527321214)

[基本类型与对应封装类型 12](#_Toc527321215)

[基本类型 12](#_Toc527321216)

[封装类型 12](#_Toc527321217)

[装箱 12](#_Toc527321218)

[拆箱 12](#_Toc527321219)

[引用类型 12](#_Toc527321220)

[强引用 12](#_Toc527321221)

[软引用 12](#_Toc527321222)

[弱引用 12](#_Toc527321223)

[虚引用 12](#_Toc527321224)

[运算符 12](#_Toc527321225)

[算术运算符 12](#_Toc527321226)

[关系运算符 12](#_Toc527321227)

[位运算符 12](#_Toc527321228)

[逻辑运算符 13](#_Toc527321229)

[赋值运算符 13](#_Toc527321230)

[条件运算符 13](#_Toc527321231)

[Instanceof运算符 13](#_Toc527321232)

[数组 13](#_Toc527321233)

[关键字 13](#_Toc527321234)

[参数传递 13](#_Toc527321235)

[值传递 13](#_Toc527321236)

[引用传递 13](#_Toc527321237)

[IO流 13](#_Toc527321238)

[输出流、输入流 13](#_Toc527321239)

[Java IO 13](#_Toc527321240)

[Java NIO 13](#_Toc527321241)

[Java 8 Stream 13](#_Toc527321242)

[Java序列化 13](#_Toc527321243)

[多线程 13](#_Toc527321244)

[多进程 14](#_Toc527321245)

[多线程 14](#_Toc527321246)

[线程池 14](#_Toc527321247)

[对象池 14](#_Toc527321248)

[Java.util.Collection类 14](#_Toc527321249)

[List 14](#_Toc527321250)

[Set 14](#_Toc527321251)

[Queue 15](#_Toc527321252)

[Java.util.Map类型 15](#_Toc527321253)

[HashMap 15](#_Toc527321254)

[HashTable 15](#_Toc527321255)

[SoftHashMap 15](#_Toc527321256)

[WeakHashMap 15](#_Toc527321257)

[SortedMap 15](#_Toc527321258)

[Java.util.Date 15](#_Toc527321259)

[Java.util.concurrent包 15](#_Toc527321260)

[ConcurrentHashMap 15](#_Toc527321261)

[ConcurrentLinkedQueue 15](#_Toc527321262)

[ConcurrentLinkedDeque 16](#_Toc527321263)

[Jdk版本 16](#_Toc527321264)

[Java8 16](#_Toc527321265)

[Lambda 16](#_Toc527321266)

[Stream 16](#_Toc527321267)

[函数式接口 16](#_Toc527321268)

[反射机制 16](#_Toc527321269)

[注解 16](#_Toc527321270)

[Java常见命令 16](#_Toc527321271)

[网络知识 16](#_Toc527321272)

[OSI七层网络模型 16](#_Toc527321273)

[TCP/IP协议 16](#_Toc527321274)

[TCP/UDP 16](#_Toc527321275)

[Http/Https 16](#_Toc527321276)

[RPC协议 16](#_Toc527321277)

[短连接与长连接 17](#_Toc527321278)

[网络安全知识 17](#_Toc527321279)

[签名与验签 17](#_Toc527321280)

[对称加密 17](#_Toc527321281)

[非对称加密 17](#_Toc527321282)

[算法与数据结构 17](#_Toc527321283)

[数据结构 17](#_Toc527321284)

[表（列表） 17](#_Toc527321285)

[栈 17](#_Toc527321286)

[队列 17](#_Toc527321287)

[树 17](#_Toc527321288)

[堆 17](#_Toc527321289)

[图 17](#_Toc527321290)

[算法 18](#_Toc527321291)

[排序 18](#_Toc527321292)

[查找 18](#_Toc527321293)

[设计模式（23种） 18](#_Toc527321294)

[创建型模式 18](#_Toc527321295)

[工厂方法模式 18](#_Toc527321296)

[抽象工厂模式 18](#_Toc527321297)

[单例模式 18](#_Toc527321298)

[创建者模式 18](#_Toc527321299)

[原型模式 19](#_Toc527321300)

[结构型模式 19](#_Toc527321301)

[适配器模式 19](#_Toc527321302)

[装饰器模式 19](#_Toc527321303)

[代理模式 19](#_Toc527321304)

[外观模式 19](#_Toc527321305)

[桥接模式 19](#_Toc527321306)

[组合模式 19](#_Toc527321307)

[亨元模式 19](#_Toc527321308)

[行为型模式 19](#_Toc527321309)

[策略模式 19](#_Toc527321310)

[模板方法模式 19](#_Toc527321311)

[观察者模式 19](#_Toc527321312)

[迭代子模式 19](#_Toc527321313)

[责任链模式 19](#_Toc527321314)

[命令模式 19](#_Toc527321315)

[备忘录模式 20](#_Toc527321316)

[状态模式 20](#_Toc527321317)

[访问者模式 20](#_Toc527321318)

[终结者模式 20](#_Toc527321319)

[解释器模式 20](#_Toc527321320)

[常用框架 20](#_Toc527321321)

[Springboot 20](#_Toc527321322)

[Spring 20](#_Toc527321323)

[Spring mvc 20](#_Toc527321324)

[Mybatis 20](#_Toc527321325)

[Quartz 20](#_Toc527321326)

[Junit 20](#_Toc527321327)

[Dubbo 20](#_Toc527321328)

[FreeMarker 20](#_Toc527321329)

[Zookeeper 20](#_Toc527321330)

[Docket 21](#_Toc527321331)

[Netty 21](#_Toc527321332)

[Mina 21](#_Toc527321333)

[Hadoop 21](#_Toc527321334)

[分布式 21](#_Toc527321335)

[Rpc框架：dubbo 21](#_Toc527321336)

[注册中心：zookeeper 21](#_Toc527321337)

[配置中心：Apollo 21](#_Toc527321338)

[信息队列 21](#_Toc527321339)

[activeMq 21](#_Toc527321340)

[kafka 21](#_Toc527321341)

[rocketMq 21](#_Toc527321342)

[数据存储 21](#_Toc527321343)

[数据库 21](#_Toc527321344)

[Mysql 21](#_Toc527321345)

[Oracle 22](#_Toc527321346)

[缓存 22](#_Toc527321347)

[Memcached 22](#_Toc527321348)

[redis 22](#_Toc527321349)

[codis 22](#_Toc527321350)

[ftp 22](#_Toc527321351)

[ftp 22](#_Toc527321352)

[ftps 22](#_Toc527321353)

[工具和插件 22](#_Toc527321354)

[Eclipse 22](#_Toc527321355)

[Svn 22](#_Toc527321356)

[Git 22](#_Toc527321357)

[Maven 22](#_Toc527321358)

[服务器 22](#_Toc527321359)

[Tomcat 22](#_Toc527321360)

[Jetty 23](#_Toc527321361)

[Nginx 23](#_Toc527321362)

[Linux 23](#_Toc527321363)

[常见命令 23](#_Toc527321364)

[日志查找 23](#_Toc527321365)

[文件查找 23](#_Toc527321366)

[页面前端框架 23](#_Toc527321367)

[Bootstrap 23](#_Toc527321368)

[Jquery 23](#_Toc527321369)

[Highcharts 23](#_Toc527321370)

[Java虚拟机 23](#_Toc527321371)

[面试技巧 23](#_Toc527321372)

# 计算机基础

## 十进制转二进制

### 整数

采用除以2取余的方式

Eg:13=1101[二进制]

13/2=6 ---- 1

6/2=3 ---- 0

3/2=1 ---- 1

1/2=0 ---- 1

所以从下往上去余数为1101

### 小数

采用乘以2取整数部分

Eg：0.5=0.1[二进制]

0.5\*2=1.0 -------- 1

所以1

0.4=0.0110 0110 0110 0110 …

0.4\*2=0.8 -------- 0

0.8\*2=1.6 -------- 1

0.6\*2=1.2 -------- 1

0.2\*2=0.4 -------- 0

…

**所以浮点数在计算机存储可能会出现精度问题。**

## 原码、反码、补码

### 原码

第一位表示符号（0表示正数，1表示负数），后面其余表示值

取值返回-127~127

### 反码

正数的反码是其本身，负数的反码是在**原码**的基础上，符号位不变，其余各位取反

取值返回-127~127

### 补码

正数的补码是其本身，负数的补码是在**反码**的基础上+1

取值返回-128~127，比原码、反码多一位1000 0000表示-128

(-1) + (-127) = [1000 0001]原 + [1111 1111]原 = [1111 1111]补 + [1000 0001]补 = [1000 0000]补

**特殊**：0用原码或者反码都有两种标识，补码只有一种，补码解决了0有两种编码的问题。降级计算机处理复杂性，也就是计算机采用补码作为机器码原因，采用了补码，计算机只有加法运算，且符号位参加运算，降低运算的复杂度

原码：0 = 0000 0000 或者 0 = 1000 0000

反码：0 = 0000 0000 或者 0 = 1111 1111

补码：0 = 0000 0000

|  |  |  |  |
| --- | --- | --- | --- |
| 十进制数 | 原码 | 反码 | 补码 |
| 85 | 0101 0101 | 0101 0101 | 0101 0101 |
| -85 | 1101 0101 | 1010 1010 | 1010 1011 |
| 9 | 0000 1001 | 0000 1001 | 0000 1001 |
| -9 | 1000 1001 | 1111 0110 | 1111 0111 |

## 浮点数的二进制表示方法（IEEE754标准）

现代计算机中，以IEEE 754标准存储浮点数，这个标准在内存中存储的格式：

|  |  |  |
| --- | --- | --- |
| 数符 | 阶码（含阶符） | 尾数 |
| sign | exponent | fraction |

float与double的存储标准

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 类型 | 数符 | 阶码 | 尾数 | 总长度 |
| float | 1 | 8 | 23 | 32 |
| double | 1 | 11 | 52 | 64 |

eg1:

float f = 13.4的二进制

1、数符为0（整数位0，负数为1）

2、整数部分转二进制 13 = 1101

3、小数部分转二进制 0.4 = 0110 0110 0110 0110 0110...，是无穷尽，取小数跟整数加起来24位即可（**高位1不存入内存**）

4、13.4的二进制表示 13.4 = 1101.0110 0110 0110 0110 0110

5、尾数1.101 0110 0110 0110 0110 0110，左移3，阶符=0

6、阶码=0111 1111（偏移量）+11（左移3） = 1000 0010

7、偏移量作用是把负数转化为整数，这样阶码取值返回就是0~255

|  |  |  |
| --- | --- | --- |
| 0 | 1000 0010 | 101 0110 0110 0110 0110 0110 |

**注意：整数位1不存入内容，固定为1，尾数储存小数部分，所以能多一位来存放小数**

eg2:

double d = 13.4的二级制

过程与float一致，只是精度不一致

|  |  |  |
| --- | --- | --- |
| 0 | 100 0000 0010 | 101 0110 0110 0110 0110 0110 0110 0110 0110 0110 0110 0110 0110 |

## 补码运算

**在进行补码运算时，符号位也参与运算，且不考虑符号位的进位问题，直接忽略掉进位。**

### 加法运算

**按补码做加法运算**

1)、计算13+7的值

2)、操作数的补码，13=0000 1101， 7=0000 0111

3)、13+7 = 0000 1101 + 0000 0111 = 0001 0100 = 20

### 减法运算

**转换成加法运算。减去一个数，相当于加上一个负数**

1)、计算13-7的值

2)、13-7 = 13+(-7)

3)、操作数补码，13=0000 1101， -7=1111 1001

4)、13 - 7 = 13+(-7) = 0000 1101 + 1111 1001 = 0000 0110 = 5

### 乘法运算

**补码运算都是通过加法和位移来实现，二进制的乘法运算其实就是被乘数的移位与相加**

补码一位乘法是把乘数Y补码的符号位设成0，当做一个正数，与被乘数补码相乘。乘法运算 的步骤与原码乘法相同。乘积出来后，如果Y是负数，则加上[-X]补得到[X\*Y]补；否则乘积就直接等于[X\*Y]补。这样的补码一位乘法也称校正法。

Eg:12\*11

1100\*1011=1100 + 1 1000 + 00 0000 + 110 0000 =1000 0100=132

Eg:2.5\*0.5

0010.1\*0.1 = 0001.01=1.25

### 除法运算

**二进制的除法运算实际是除数的移位与相减**

**计算机内部是通过计数器记录被除数-除数的次数来求值，**

**Eg：12/6=2**

**1100-0110 = 0110 ---0001**

**0110-0110 = 0000 ---0001**

**所以0001+0001 = 0010 = 2**

# Java基础知识

## 基本类型与对应封装类型

### 基本类型

java基本类型有3类共8种。6个数字类型，1个字符类型，1个布尔类型。**注意String不是基本类型**

**1、byte：8位**有符号以二进制**补码**表示的整数。

最大值：0111 1111 = 2^7-1 = 127

最小值：1111 1111【原码】=1000 0000【反码】=1000 0001【补码】= -(2^7-1)=-127

由于使用补码没有-0=1000 0000，使用-0来表示-128

所以最小值-128

默认值：0

**2、short：16位**有符号以二进制**补码**表示的整数

最大值：0111 1111 1111 1111 = 2^15-1 = 32767

最小值：-32768。原理同byte，负数比整数多一位

默认值：0

**3、int：32位**有符号以二进制**补码**表示的整数

最大值：2^31-1 = 2147483648-1=2147483647

最小值：-2147483648。原理见byte

默认值：0

**4、long：64位**有符号以二进制**补码**表示的整数

最大值：2^63-1

最小值：2^63

默认值：0L

"L"理论上不分大小写，但是若写成"l"容易与数字"1"混淆，不容易分辩。所以最好大写。

**5、float：**float 数据类型是**单精度、32位**、符合IEEE 754标准的浮点数

默认值：0.0f

float a = 5.0f。必须以f结尾，不然会报错

**6、double：**double 数据类型是**双精度、64 位**、符合IEEE 754标准的浮点数

默认值：0.0d

double a = 5.0d或者double a = 5.0。5.0默认是double类型

**用浮点数做计算，可能会损失精度**。如果需要精确计算使用**BigInteger或者BegDecimal**

**7、char：**char类型是一个**单一的 16 位 Unicode 字符**

最大值：\uffff = 2^16-1=65535

最小值：\u0000=0

char 数据类型可以储存任何字符，以Ascii数值来表示字符

char a = 'a'; //字符

int n = 'a'; //ascii

System.out.println(a + ":"+ n);//a:97

**8、boolean：**boolean数据类型表示**一位**的信息

只有两个取值：true 和 false

### 封装类型

基本类型都有对应封装类型

|  |  |  |
| --- | --- | --- |
| 基本类型 | 封装类型 |  |
| int | Integer | Integer.value(int i) |
| long | Long |  |
| byte | Byte |  |
| short | Short |  |
| float | Float |  |
| double | Double |  |
| boolean | Boolean |  |
| char | Character |  |

### 装箱

java从java se5开始提供了自动装箱的机制，从基本类型自动封装成对象类型。

### 拆箱

java从java se5开始提供了自动拆箱的机制，从对象类型自动转成成基本类型

## 引用类型

### 强引用

### 软引用

### 弱引用

### 虚引用

## 运算符

### 算术运算符

### 关系运算符

### 位运算符

### 逻辑运算符

### 赋值运算符

### 条件运算符

### Instanceof运算符

## 数组

## 关键字

## 参数传递

### 值传递

### 引用传递

## IO流

### 输出流、输入流

### Java IO

### Java NIO

### Java 8 Stream

## Java序列化

## 多线程

### 多进程

### 多线程

### 线程池

## 对象池

## Java.util.Collection类

### List

#### ArrayList

#### LinkedList

#### Vector

#### SynchronizedList

### Set

#### HashSet

#### TreeSet

#### LinkedHashSet

#### PersistentSet

#### PredicatedSet

#### SortedSet

### Queue

#### ConcurrentLinkedQueue

#### BlockingQueue

#### Deque

## Java.util.Map类型

### HashMap

### HashTable

### SoftHashMap

### WeakHashMap

### SortedMap

## Java.util.Date

## Java.util.concurrent包

### ConcurrentHashMap

### ConcurrentLinkedQueue

### ConcurrentLinkedDeque

## Jdk版本

## Java8

### Lambda

### Stream

### 函数式接口

## 反射机制

## 注解

## Java常见命令

# 网络知识

## OSI七层网络模型

## TCP/IP协议

## TCP/UDP

## Http/Https

## RPC协议

## 短连接与长连接

# 网络安全知识

## 签名与验签

## 对称加密

## 非对称加密

# 算法与数据结构

## 数据结构

### 表（列表）

### 栈

### 队列

### 树

#### 二叉树

#### 红黑树

### 堆

### 图

## 算法

### 排序

#### 选择排序

选择排序可以说是最简单的一种排序方法：

1.找到数组中最小的那个元素

2.将最小的这个元素和数组中第一个元素交换位置

3.在剩下的元素中找到最小的的元素，与数组第二个元素交换位置

重复以上步骤，即可以得到有序数组。

|  |
| --- |
| **public** **static** **int**[] sort(**int**[] src){  **if**(src == **null** || src.length == 0){  **return** src;  }    **int** min = 0;  **for**(**int** i=0; i<src.length; i++){  min = i;  **for**(**int** j = i+1; j<src.length; j++){  //每次遍历比较，找出最小（或者最大）下标  **if**(src[min] > src[j]){  min = j;  }  }    //如果遍历找到比第一位更小的数，调换位置  **if**(i != min){  **int** tem = src[i];  src[i] = src[min];  src[min] = tem;  }  }    **return** src;  } |

效率：对于长度为N的数组，选择排序需要大约N²/2次比较和N次交换。也即最好、最差、平均时间效率均为O（n²），只需要一个辅助变量帮助交换元素。

选择排序可以看成是冒泡排序的扩展，一个是把最小或最大的选出来，再交换，一个是一直交换直到最大最小的出现在正确的位置上，选择排序相对于冒泡排序，比较次数是一样的，但是交换次数要少很多。

#### 插入排序

插入排序类似整理扑克牌，将每一张牌插到其他已经有序的牌中适当的位置。

插入排序由N-1趟排序组成，对于P=1到N-1趟，插入排序保证从位置0到位置P上的元素为已排序状态。

简单的说，就是插入排序总共需要排序N-1趟，从index为1开始，讲该位置上的元素与之前的元素比较，放入合适的位置，这样循环下来之后，即为有序数组。

|  |
| --- |
| **public** **static** **int**[] sort(**int**[] src){  **if**(src == **null** || src.length == 0){  **return** src;  }    **for**(**int** i=1; i<src.length; i++){  **for**(**int** j=0; j<i; j++){  **if**(src[i] < src[j]){  **int** temp = src[i];  **for**(**int** k=i; k>j; ){  src[k] = src[--k];  }  src[j] = temp;  }  }  }    **return** src;  } |

效率：如果目标是把n个元素的序列升序排列，那么采用插入排序存在最好情况和最坏情况。最好情况就是，序列已经是升序排列了，在这种情况下，需要进行的比较操作需（n-1）次即可。最坏情况就是，序列是降序排列，那么此时需要进行的比较共有n(n-1)/2次。插入排序的赋值操作是比较操作的次数加上 (n-1）次。平均来说插入排序算法的时间复杂度为O(n^2）

#### 希尔排序

把记录按步长 gap 分组，对每组记录采用直接插入排序方法进行排序。

随着步长逐渐减小，所分成的组包含的记录越来越多，当步长的值减小到 1 时，整个数据合成为一组，构成一组有序记录，则完成排序。

|  |
| --- |
| **public** **static** **int**[] sort(**int**[] a){  Integer h = a.length;  Integer temp = 0;  **while**(h >= 1) {  **for**(**int** i=h;i<a.length;i++) {  **for**(**int** j=i;j>=h && a[j] < a[j-h];j -= h) {  temp = a[j];  a[j] = a[j-h];  a[j-h] = temp;    }  }  h /= 9;  }  **return** a;  } |

#### 堆排序

#### 归并排序

#### 快速排序

#### 桶式排序

### 查找

# 设计模式（23种）

## 创建型模式

### 工厂方法模式

### 抽象工厂模式

### 单例模式

### 创建者模式

### 原型模式

## 结构型模式

### 适配器模式

### 装饰器模式

### 代理模式

### 外观模式

### 桥接模式

### 组合模式

### 亨元模式

## 行为型模式

### 策略模式

### 模板方法模式

### 观察者模式

### 迭代子模式

### 责任链模式

### 命令模式

### 备忘录模式

### 状态模式

### 访问者模式

### 终结者模式

### 解释器模式

# 常用框架

## Springboot

1、属于spring旗下的一个项目，其设计目的是用来简化新Spring应用的初始搭建以及开发过程。该框架使用了特定的方式来进行配置，它使用“习惯优于配置”的理念，从而使开发人员不再需要定义样板化的配置。使用springboot很容易创建一个独立运行的spring项目，并且可以几乎不使用spring配置或者使用很少的配置。

2、结构

1）、springboot是一个maven项目，只需要在pom.xml添加依赖就可以。

2）、在pom.xml添加一个parent依赖，控制整个springboot项目的版本

<parent>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-parent</artifactId>

<version>1.5.6.RELEASE</version>

<relativePath/> <!-- lookup parent from repository -->

</parent>

3）、spring提供了很多spring-boot-starter-XXX的maven依赖，来简化maven依赖配置

比如：web相关spring-boot-starter-web，会自动关联一些常见web依赖，比如springMVC， servlet、tomcat容器。。。因此无需在tomcat服务器上部署。

比如：spring-boot-starter-thymeleaf，会自动关联一些与thymeleaf相关的依赖

4）、编写springboot的入口类，一般命名是xxxApplication.java，其他的类都在它所在的目录或者子目录中，这样spring容器会自动扫描有相应注解的类。入口类上的注解是SpringBootApplicaton,类中有个Main方法作为程序入口，springboot框架提供一个方法启动项目Application.run(本类名.class，args)。

5）、springboot启动时会自动扫描src/main/resources中配置application.properties

6）、入口注解@SpringBootApplication注解是一个组合注解，它包含@configuration、@EnableAutoConfiguration和@componentScan三个注解组成。它的核心是@EnableAutoConfiguration注解。这个注解中有一个@Import注解，它导入配置功能EnableAutoConfigurationImportSelector，这个类有个方法扫描具有/META-INF/spring.factories文件的jar。spring.factories中声明了哪些自动配置。然后我们在application.properties文件中进行相应的配置，比如数据库连接，使springboot中的自动配置中的参数是我们需要的配置。

7）如果项目最后要以jar形式运行，必须加入spring-boot-maven-plugin插件

<plugin>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-maven-plugin</artifactId>

<configuration>

<fork>true</fork>

</configuration>

</plugin>

8）@SpringBootApplication标签

@ EnableAutoConfiguration：启用Spring Boot的自动配置机制

@ ComponentScan：在应用程序所在的包上启用@Component 扫描

@Configuration：允许在上下文中注册额外的bean或导入额外的配置类

## Spring

## Spring mvc

## Mybatis

## Quartz

## Junit

## Dubbo

## FreeMarker

## Zookeeper

## Docket

## Netty

## Mina

## Hadoop

# 分布式

## Rpc框架：dubbo

## 注册中心：zookeeper

## 配置中心：Apollo

# 信息队列

## activeMq

## kafka

## rocketMq

# 数据存储

## 数据库

### Mysql

### Oracle

## 缓存

### Memcached

### redis

### codis

## ftp

### ftp

### ftps

# 工具和插件

## Eclipse

## Svn

## Git

## Maven

### Maven生命周期

一个完整的项目构建过程通常包括清理、编译、测试、打包、集成测试、验证、部署等步骤，Maven从中抽取了一套完善的、易扩展的生命周期。Maven的生命周期是抽象的，其中的具体任务都交由插件来完成。Maven为大多数构建任务编写并绑定了默认的插件，如针对编译的插件：maven-compiler-plugin。用户也可自行配置或编写插件。

maven包括三套生命周期：clean、default、site，每个生命周期都包含了一些阶段（phase）。三套生命周期相互独立，但各个生命周期中的phase却是有顺序的，且后面的phase依赖于前面的phase。执行某个phase时，其前面的phase会依顺序执行，但不会触发另外两套生命周期中的任何phase。

#### Clean：清除编译生成的文件，有三个阶段

1）pre-clean:执行清除前的工作

2）clean:清理上一次构建生成的所有文件

3）post-clean:执行清除后的工作

#### Default： default生命周期是最核心的，它包含了构建项目时真正需要执行的所有步骤。

1）validate

2）initialize

3）generate-sources

4）process-sources

5）generate-resources

6）process-resources    ：复制和处理资源文件到target目录，准备打包；

7）compile    ：编译项目的源代码；

8）process-classes

9）generate-test-sources

10）process-test-sources

11）generate-test-resources

12）process-test-resources

13）test-compile    ：编译测试源代码；

14）process-test-classes

15）test    ：运行测试代码；

16）prepare-package

17）package    ：打包成jar或者war或者其他格式的分发包；

18）pre-integration-test

19）integration-test

20）post-integration-test

21）verify

22）install    ：将打好的包安装到本地仓库，供其他项目使用；

23）deploy    ：将打好的包安装到远程仓库，供其他项目使用；

#### Site：生成项目的站点文档

1、pre-site:

2、site: 生成项目的站点文档；

3、post-site:

4、site-deploy:发布生成的站点文档

### Maven插件

Maven所有工作都是由插件来完成，插件分为两类：

1、Build plugins 在构建项目的时候执行，应该被配置在 <build> 元素中

2、Reporting plugins 在生成站点的时候执行，应该被配置在 <reporting> 元素中

所有的插件配置要求有三个信息：groupId ， artifactId ，version ，这跟依赖的配置相似。类似 <dependencyManagement> ， 插件配置也有 <pluginManagement> ，用法也一样，参考依赖管理。

一个普通的配置看起来如下：

|  |
| --- |
| <project>  ...  <build>  <plugins>  <plugin>  <artifactId>maven-myquery-plugin</artifactId>  <version>1.0</version>  <configuration>  <url>http://www.foobar.com/query</url>  <timeout>10</timeout>  <options>  <option>one</option>  <option>two</option>  <option>three</option>  </options>  </configuration>  </plugin>  </plugins>  </build>  ...  </project> |

<configuration> 里面的元素对应插件目标的参数，如果想知道某个插件目标的可用参数，通常可以通过下面的命令查询

mvn <pluginName>:help -Ddetail -Dgoal=<goalName>

比如想知道 install 插件的 install 目标的参数，则可执行命令

mvn install:help -Ddetail -Dgoal=install

**插件目标的配置**

通常我们需要配置插件目标执行时的参数，看下面的例子

|  |
| --- |
| <project>  ...  <build>  <plugins>  <plugin>  <artifactId>maven-myquery-plugin</artifactId>  <version>1.0</version>  <executions>  <execution>  <id>execution1</id>  <phase>test</phase>  <configuration>  <url>http://www.foo.com/query</url>  <timeout>10</timeout>  <options>  <option>one</option>  <option>two</option>  <option>three</option>  </options>  </configuration>  <goals>  <goal>query</goal>  </goals>  </execution>  <execution>  <id>execution2</id>  <configuration>  <url>http://www.bar.com/query</url>  <timeout>15</timeout>  <options>  <option>four</option>  <option>five</option>  <option>six</option>  </options>  </configuration>  <goals>  <goal>query</goal>  </goals>  </execution>  </executions>  </plugin>  </plugins>  </build>  ...  </project> |

即将 <configuration> 置于 <execution> 标签中，<execution> 标签中通过配置 <phase> 、<goal> 分别指定了配置应用的阶段和目标，如例子中的 id 为 execution1 的配置会应用在 test 阶段中的 query 目标中。我们可以看到 id 为 execution2 的 <execution> 中没有 phase 标签，那么它会在什么时候应用呢？

如果该目标默认绑定了一个阶段，则在这个阶段应用。

如果该目标没有默认的绑定，则不会应用。

这里的 <id>execution1</id> 有什么用呢？其实当我们执行一条命令时，像

mvn maven-myquery-plugin:query

这时它会应用什么配置呢？如果在 <executions> 外有配置，则会应用，如果没有，则上面配置的 <execution> 并不会应用上，那么如果我们希望执行上面配置好参数的目标，那么可以加上 id 执行，如

mvn maven-myquery-plugin:query@execution1

执行时就会应用上 execution1 的配置。

**忽略继承**

默认情况下，子工程会继承父工程的插件配置，如果不希望继承，则可配置 <inherited> 标签

<project>

...

<build>

<plugins>

<plugin>

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

<artifactId>maven-antrun-plugin</artifactId>

<version>1.2</version>

<inherited>false</inherited>

...

</plugin>

</plugins>

</build>

...

</project>

### Maven常用命令

1、mvn clean:清理项目生产的临时文件,一般是模块下的target目录

2、mvn package:项目打包工具,会在模块下的target目录生成jar或war等文件，如下运行结果

3、mvn test:测试命令,或执行src/test/java/下junit的测试用例.

4、mvn install:模块安装命令 将打包的的jar/war文件复制到你的本地仓库中,供其他模块使用

5、mvn deploy:发布命令 将打包的文件发布到远程参考,提供其他人员进行下载依赖 ,一般是发布到公司的私服

6、mvn compile:编译项目源代码

7、mvn test-compile:编译测试源代码

### Pom.xml文件详解

实际上pom之间存在三种关系：继承、依赖、聚合。

#### 基本信息

|  |  |
| --- | --- |
| modelVersion | Maven模块版本，目前我们一般都取值4.0.0 |
| groupId | 整个系统的名称。 |
| artifactId | 子模块名称。 |
| packaging | 打包类型，可取值：jar,war等等，这个配置用于package的phase，具体可以参见package运行的时候启动的plugin，后面有机会我们会讲述如何配置打包的插件。 |

#### Dependence：依赖

|  |  |
| --- | --- |
| groupId | 依赖项的groupId |
| artifactId | 依赖项的artifactId |
| version | 依赖项的版本 |
| scope | 依赖项的适用范围：   * compile，缺省值，适用于所有阶段，会随着项目一起发布。 * provided，类似compile，期望JDK、容器或使用者会提供这个依赖。如servlet.jar。 * runtime，只在运行时使用，如JDBC驱动，适用运行和测试阶段。 * test，只在测试时使用，用于编译和运行测试代码。不会随项目发布。 * system，类似provided，需要显式提供包含依赖的jar，Maven不会在Repository中查找它。   之前例子里的junit就只用在了test中。 |
| exclusions | 排除项目中的依赖冲突时使用。 |
| type | 如果该依赖不是 jar 类型，则需要指定 type ，默认是 jar。如果是 war ，则需指定 <type>war<type> |

#### Parent：继承

<parent>

<groupId>com.thoughtworks.xstream</groupId>

<artifactId>xstream-parent</artifactId>

<version>1.4.3</version>

</parent>

#### Module：聚合

<modules>

<module>module1</module>

<module>module2</module>

</modules>

#### Properties：属性

定义一些常量

#### dependencyManagement：依赖管理

依赖管理只是管理依赖，并没有真正引入，一般在父pom中管理依赖，在子pom引入依赖。

<dependencyManagement>

<dependencies>

<dependency>

<groupId>junit</groupId>

<artifactId>junit</artifactId>

<version>4.12</version>

<scope>test</scope>

</dependency>

</dependencies>

</dependencyManagement>

#### PluginManagement：插件管理

插件管理只是管理插件，并没有真正引入。用法跟dependencyManagement类似

# 服务器

## Tomcat

## Jetty

## Nginx

# Linux

## 常见命令

## 日志查找

## 文件查找

# 页面前端框架

## Bootstrap

## Jquery

## Highcharts

# Java虚拟机

# 面试技巧