|  |
| --- |
| 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

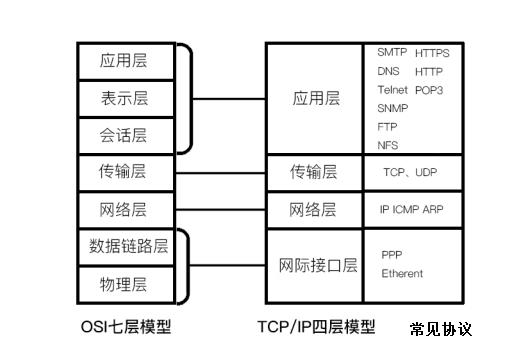
## 反射机制

## 注解

## Java常见命令

# 网络知识

## OSI七层网络模型



## TCP/IP协议

TCP/IP不是一个协议，而是一个协议族的统称。里面包括了IP协议，IMCP协议，TCP协议，以及我们更加熟悉的http、ftp、pop3协议等等。

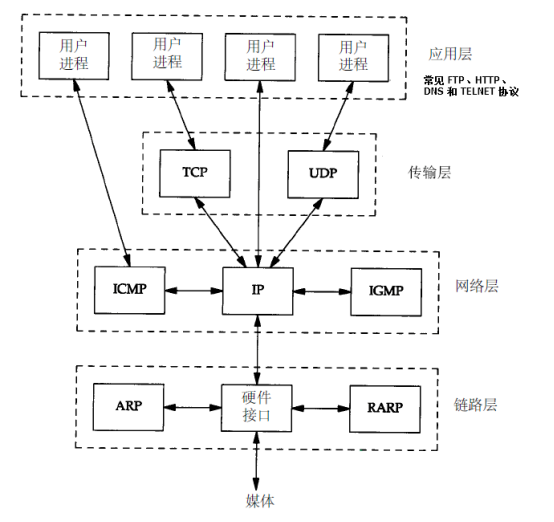
TCP/IP协议分为四层：

(1)应用层：应用程序通过这一层访问网络，常见 FTP、HTTP、DNS 和 TELNET 协议；

(2)传输层：TCP 协议和 UDP 协议；

(3)网络层：IP 协议，ARP、RARP 协议，ICMP 协议等；

(4)网络接口层：是 TCP/IP 协议的基层，负责数据帧的发送和接收。



### 基础知识

**1.IP 地址**

网络上每一个节点都必须有一个独立的 IP 地址，通常使用的 IP 地址是一个 32bit 的数字，被分成 4 组，例如，255.255.255.255 就是一个 IP 地址。IP地址就是计算机网络组成的最小单位。

在 Linux 系统中，可以用 ifconfig -a 命令查看自己的 IP 地址，windows的DOS中可以用ipconfing查看

**2.域名**

用 12 位数字组成的 IP 地址，在实际应用时，用户一般不需要记住 IP 地址，互联网给每个 IP 地址起了一个别名，习惯上称作域名。

可以使用命令 nslookup 或者 ping 在Linux中查看与域名相对应的 IP 地址。

**3.MAC 地址**

MAC（Media Access Control）地址，或称为物理地址、硬件地址，用来定义互联网中设备的位置。

在 TCP/IP 层次模型中，网络层管理 IP 地址，链路层则负责 MAC 地址。因此每个网络位置会有一个专属于它的 IP 地址，而每个主机会有一个专属于它 MAC 地址。

### 交互时数据处理方式

**封装**：当应用程序发送数据的时候，数据在协议层次当中从顶向下通过每一层，每一层都会对数据增加一些首部或尾部信息，这样的信息称之为协议数据单元（Protocol Data Unit，缩写为PDU），在分层协议系统里，在指定的协议层上传送的数据单元，包含了该层的协议控制信息和用户信息。如下图所示：

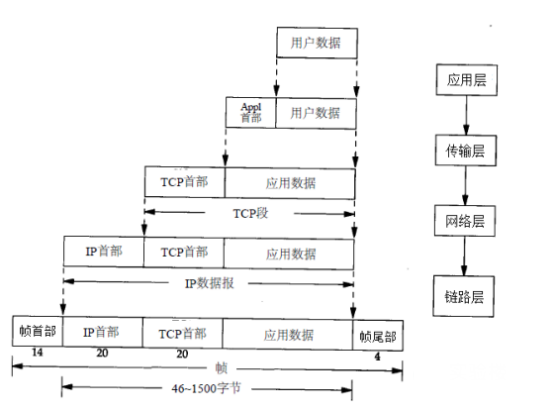
**· 物理层（一层）PDU指数据位（Bit）**

**· 数据链路层（二层）PDU指数据帧（Frame）**

**· 网络层（三层）PDU指数据包（Packet）**

**· 传输层（四层）PDU指数据段（Segment）**

**· 第五层以上为数据（data）**

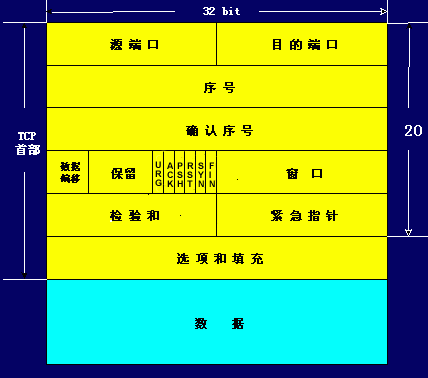


**分用**：当主机收到一个数据帧时，数据就从协议层底向上升，通过每一层时，检查并去掉对应层次的报文首部或尾部，与封装过程正好相反。

## TCP/UDP

### TCP

#### TCP报文格式



16位源端口号：16位的源端口中包含初始化通信的端口。源端口和源IP地址的作用是标识报文的返回地址。

16位目的端口号：16位的目的端口域定义传输的目的。这个端口指明报文接收计算机上的应用程序地址接口。

32位序号：32位的序列号由接收端计算机使用，重新分段的报文成最初形式。当SYN出现，序列码实际上是初始序列码（Initial Sequence Number，ISN），而第一个数据字节是ISN+1。这个序列号（序列码）可用来补偿传输中的不一致。

32位确认序号：32位的序列号由接收端计算机使用，重组分段的报文成最初形式。如果设置了ACK控制位，这个值表示一个准备接收的包的序列码。

4位首部长度：4位包括TCP头大小，指示何处数据开始。

保留（6位）：6位值域，这些位必须是0。为了将来定义新的用途而保留。

标志：6位标志域。表示为：紧急标志、有意义的应答标志、推、重置连接标志、同步序列号标志、完成发送数据标志。按照顺序排列是：URG、ACK、PSH、RST、SYN、FIN。

16位窗口大小：用来表示想收到的每个TCP数据段的大小。TCP的流量控制由连接的每一端通过声明的窗口大小来提供。窗口大小为字节数，起始于确认序号字段指明的值，这个值是接收端正期望接收的字节。窗口大小是一个16字节字段，因而窗口大小最大为65535字节。

16位校验和：16位TCP头。源机器基于数据内容计算一个数值，收信息机要与源机器数值 结果完全一样，从而证明数据的有效性。检验和覆盖了整个的TCP报文段：这是一个强制性的字段，一定是由发送端计算和存储，并由接收端进行验证的。

16位紧急指针：指向后面是优先数据的字节，在URG标志设置了时才有效。如果URG标志没有被设置，紧急域作为填充。加快处理标示为紧急的数据段。

选项：长度不定，但长度必须为1个字节。如果没有选项就表示这个1字节的域等于0。

数据：该TCP协议包负载的数据。

在上述字段中，6位标志域的各个选项功能如下：

URG：紧急标志。紧急标志为"1"表明该位有效。

ACK：确认标志。表明确认编号栏有效。大多数情况下该标志位是置位的。TCP报头内的确认编号栏内包含的确认编号（w+1）为下一个预期的序列编号，同时提示远端系统已经成功接收所有数据。

PSH：推标志。该标志置位时，接收端不将该数据进行队列处理，而是尽可能快地将数据转由应用处理。在处理Telnet或rlogin等交互模式的连接时，该标志总是置位的。

RST：复位标志。用于复位相应的TCP连接。

SYN：同步标志。表明同步序列编号栏有效。该标志仅在三次握手建立TCP连接时有效。它提示TCP连接的服务端检查序列编号，该序列编号为TCP连接初始端（一般是客户端）的初始序列编号。在这里，可以把TCP序列编号看作是一个范围从0到4，294，967，295的32位计数器。通过TCP连接交换的数据中每一个字节都经过序列编号。在TCP报头中的序列编号栏包括了TCP分段中第一个字节的序列编号。

FIN：结束标志。

#### TCP三次握手

所谓三次握手（Three-Way Handshake）即建立TCP连接，就是指建立一个TCP连接时，需要客户端和服务端总共发送3个包以确认连接的建立。在socket编程中，这一过程由客户端执行connect来触发，整个流程如下图所示：



（1）第一次握手：Client将标志位SYN置为1，随机产生一个值seq=J，并将该数据包发送给Server，Client进入SYN\_SENT状态，等待Server确认。

（2）第二次握手：Server收到数据包后由标志位SYN=1知道Client请求建立连接，Server将标志位SYN和ACK都置为1，ack=J+1，随机产生一个值seq=K，并将该数据包发送给Client以确认连接请求，Server进入SYN\_RCVD状态。

（3）第三次握手：Client收到确认后，检查ack是否为J+1，ACK是否为1，如果正确则将标志位ACK置为1，ack=K+1，并将该数据包发送给Server，Server检查ack是否为K+1，ACK是否为1，如果正确则连接建立成功，Client和Server进入ESTABLISHED状态，完成三次握手，随后Client与Server之间可以开始传输数据了。

简单来说，就是

1、建立连接时，客户端发送SYN包（SYN=i）到服务器，并进入到SYN-SEND状态，等待服务器确认

2、服务器收到SYN包，必须确认客户的SYN（ack=i+1）,同时自己也发送一个SYN包（SYN=k）,即SYN+ACK包，此时服务器进入SYN-RECV状态

3、客户端收到服务器的SYN+ACK包，向服务器发送确认报ACK（ack=k+1）,此包发送完毕，客户端和服务器进入ESTABLISHED状态，完成三次握手，客户端与服务器开始传送数据。

**SYN攻击**：

在三次握手过程中，Server发送SYN-ACK之后，收到Client的ACK之前的TCP连接称为半连接（half-open connect），此时Server处于SYN\_RCVD状态，当收到ACK后，Server转入ESTABLISHED状态。SYN攻击就是Client在短时间内伪造大量不存在的IP地址，并向Server不断地发送SYN包，Server回复确认包，并等待Client的确认，由于源地址是不存在的，因此，Server需要不断重发直至超时，这些伪造的SYN包将产时间占用未连接队列，导致正常的SYN请求因为队列满而被丢弃，从而引起网络堵塞甚至系统瘫痪。SYN攻击时一种典型的DDOS攻击，检测SYN攻击的方式非常简单，即当Server上有大量半连接状态且源IP地址是随机的，则可以断定遭到SYN攻击了，使用如下命令可以让之现行：

#netstat -nap | grep SYN\_RECV

#### TCP四次挥手

所谓四次挥手（Four-Way Wavehand）即终止TCP连接，就是指断开一个TCP连接时，需要客户端和服务端总共发送4个包以确认连接的断开。在socket编程中，这一过程由客户端或服务端任一方执行close来触发，整个流程如下图所示：



由于TCP连接时全双工的，因此，每个方向都必须要单独进行关闭，这一原则是当一方完成数据发送任务后，发送一个FIN来终止这一方向的连接，收到一个FIN只是意味着这一方向上没有数据流动了，即不会再收到数据了，但是在这个TCP连接上仍然能够发送数据，直到这一方向也发送了FIN。首先进行关闭的一方将执行主动关闭，而另一方则执行被动关闭，上图描述的即是如此。

（1）第一次挥手：Client发送一个FIN，用来关闭Client到Server的数据传送，Client进入FIN\_WAIT\_1状态。

（2）第二次挥手：Server收到FIN后，发送一个ACK给Client，确认序号为收到序号+1（与SYN相同，一个FIN占用一个序号），Server进入CLOSE\_WAIT状态。

（3）第三次挥手：Server发送一个FIN，用来关闭Server到Client的数据传送，Server进入LAST\_ACK状态。

（4）第四次挥手：Client收到FIN后，Client进入TIME\_WAIT状态，接着发送一个ACK给Server，确认序号为收到序号+1，Server进入CLOSED状态，完成四次挥手。

**为什么建立连接是三次握手，而关闭连接却是四次挥手呢？**

这是因为服务端在LISTEN状态下，收到建立连接请求的SYN报文后，把ACK和SYN放在一个报文里发送给客户端。而关闭连接时，当收到对方的FIN报文时，仅仅表示对方不再发送数据了但是还能接收数据，己方也未必全部数据都发送给对方了，所以己方可以立即close，也可以发送一些数据给对方后，再发送FIN报文给对方来表示同意现在关闭连接，因此，己方ACK和FIN一般都会分开发送。

**为什么TIME\_WAIT状态需要经过2MSL(最大报文段生存时间)才能返回到CLOSE状态？**

原因有二：

一、保证TCP协议的全双工连接能够可靠关闭

二、保证这次连接的重复数据段从网络中消失

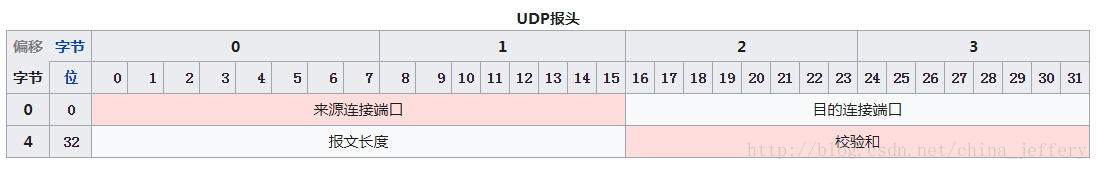
先说第一点，如果Client直接CLOSED了，那么由于IP协议的不可靠性或者是其它网络原因，导致Server没有收到Client最后回复的ACK。那么Server就会在超时之后继续发送FIN，此时由于Client已经CLOSED了，就找不到与重发的FIN对应的连接，最后Server就会收到RST而不是ACK，Server就会以为是连接错误把问题报告给高层。这样的情况虽然不会造成数据丢失，但是却导致TCP协议不符合可靠连接的要求。所以，Client不是直接进入CLOSED，而是要保持TIME\_WAIT，当再次收到FIN的时候，能够保证对方收到ACK，最后正确的关闭连接。

再说第二点，如果Client直接CLOSED，然后又再向Server发起一个新连接，我们不能保证这个新连接与刚关闭的连接的端口号是不同的。也就是说有可能新连接和老连接的端口号是相同的。一般来说不会发生什么问题，但是还是有特殊情况出现：假设新连接和已经关闭的老连接端口号是一样的，如果前一次连接的某些数据仍然滞留在网络中，这些延迟数据在建立新连接之后才到达Server，由于新连接和老连接的端口号是一样的，又因为TCP协议判断不同连接的依据是socket pair，于是，TCP协议就认为那个延迟的数据是属于新连接的，这样就和真正的新连接的数据包发生混淆了。所以TCP连接还要在TIME\_WAIT状态等待2倍MSL，这样可以保证本次连接的所有数据都从网络中消失。

### UDP

UDP是User Datagram Protocol的简称，中文名是用户数据报协议，是OSI参考模型中的传输层协议，它是一种无连接的传输层协议，提供面向事务的简单不可靠信息传送服务。

#### UDP报文格式



UDP报头由4个部分组成，其中两个是可选的（粉红背景标出部分）：

1、各16bit的来源端口和目的端口用来标记发送和接受的应用进程。因为UDP不需要应答，所以来源端口是可选的，如果来源端口不用，那么置为零。

2、在目的端口后面是长度固定的以字节为单位的报文长度域，用来指定UDP数据报包括数据部分的长度，长度最小值为8byte。

3、首部剩下地16bit是用来对首部和数据部分一起做校验和（Checksum）的，这部分是可选的，但在实际应用中一般都使用这一功能。

4、UDP和TCP的校验和都覆盖到了他们的首部和数据，而之前介绍的IP首部的校验和只覆盖了IP首部。

**应用场景：**

由于缺乏可靠性且属于非连接导向协议，UDP的应用一般必须允许一定量的丢包、出错和复制粘贴。但有些应用，比如TFTP，需要可靠性保证，则必须在应用层增加根本的可靠机制。但是绝大多数UDP应用都不需要可靠机制，甚至可能因为引入可靠机制而降低性能。流媒体、即时多媒体游戏和IP电话（VoIP）就是典型的UDP应用。如果某个应用需要很高的可靠性，那么可以用传输控制协议（即TCP协议）来代替UDP。

使用UDP协议的应用有：域名系统（DNS）、简单网络管理协议（SNMP）、动态主机配置协议（DHCP）、路由信息协议（RIP）等等。因为UDP不属于连接型协议，因而具有资源消耗小，处理速度快的优点，所以通常音频、视频和普通数据在传送时使用UDP较多，因为它们即使偶尔丢失几个数据包，也不会对接收结果产生太大影响。

### TCP与UDP区别

|  |  |  |
| --- | --- | --- |
| 特征点 | TCP | UDP |
| 是否连接 | 面向连接 | 面向非连接 |
| 传输可靠性 | 可靠 | 会丢包、不可靠 |
| 应用场景 | 传输数据量大 | 数据量小 |
| 速度 | 慢 | 块 |

UDP(用户数据报协议)是一个简单的面向数据报的运输层协议。UDP不提供可靠性，它只是把应用程序传给IP层的数据报发送出去，但是并不能保证它们能到达目的地。由于UDP在传输数据报前不用在客户和服务器之间建立一个连接，且没有超时重发等机制，故而传输速度很快。

由于UDP缺乏拥塞控制（congestion control），需要基于网络的机制来减少因失控和高速UDP流量负荷而导致的拥塞崩溃效应。换句话说，因为UDP发送者不能够检测拥塞，所以像使用包队列和丢弃技术的路由器这样的网络基本设备往往就成为降低UDP过大通信量的有效工具。数据报拥塞控制协议（DCCP）设计成通过在诸如流媒体类型的高速率UDP流中，增加主机拥塞控制，来减小这个潜在的问题。

## Socket

Socket是应用层与TCP/IP协议族通信的中间软件抽象层，它是一组接口。在设计模式中，Socket其实就是一个门面模式，它把复杂的TCP/IP协议族隐藏在Socket接口后面，对用户来说，一组简单的接口就是全部，让Socket去组织数据，以符合指定的协议。



## Http/Https

### Http

Hyper Text Transfer Protocol，超文本传输协议，是一种建立在TCP上的无状态连接，整个基本的工作流程是客户端发送一个HTTP请求，说明客户端想要访问的资源和请求的动作，服务端收到请求之后，服务端开始处理请求，并根据请求做出相应的动作访问服务器资源，最后通过发送HTTP响应把结果返回给客户端。其中一个请求的开始到一个响应的结束称为事务，当一个事物结束后还会在服务端添加一条日志条目。

#### 请求

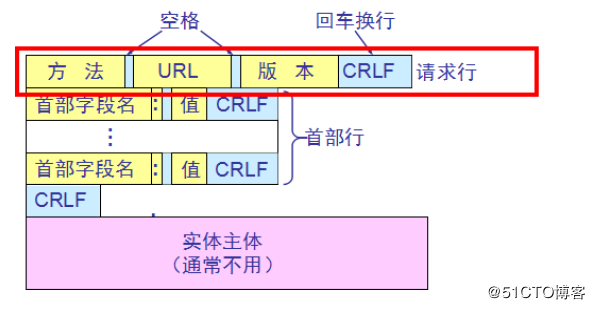
HTTP请求是客户端往服务端发送请求动作，告知服务器自己的要求。

HTTP请求由状态行、请求头、请求正文三部分组成：

状态行：包括请求方式Method、资源路径URL、协议版本Version；

请求头：包括一些访问的域名、用户代理、Cookie等信息；

请求正文：就是HTTP请求的数据。



### Https

### Http与HTTPS的区别

## RPC协议

## 短连接与长连接

# 网络安全知识

## 摘要算法

常用的有MD5、SHA1，摘要算法是一个不可逆过程，就是无论多大数据，经过算法运算后都是生成固定长度的数据,一般结果使用16进制进行显示。不需要密钥。

MD5和SHA1的区别：MD5结果是128位摘要，SHa1是160位摘要。那么MD5的速度更快，而SHA1的强度更高。

1、MD5：与MD5同一个家族还有MD2、MD4，都是产生128位摘要。主要用途有：验证消息完整性，安全访问认证，数据签名。

消息完整性：由于每一份数据生成的MD5值不一样，因此发送数据时可以将数据和其MD5值一起发送，然后就可以用MD5验证数据是否丢失、修改。

安全访问认证：这是使用了算法的不可逆性质，（就是无法从MD5值中恢复原数据）对账号登陆的密码进行MD5运算然后保存，这样可以保证除了用户之外，即使数据库管理人员都无法得知用户的密码。

数字签名：这是结合非对称加密算法和CA证书的一种使用场景。

加密采用Apache提供commons-codec的jar包，改jar包在springboot中dependencyManagement管理了

|  |
| --- |
| /\*\*  \* 用“jdk的实现”生成MD5摘要，在rt.jar中java.security.MessageDigest  \* 使用Apache提供commons-codec.jar的Hex做编码转换  \* MD5生成128位摘要，用32位的十六进制表示  \*  \* **@param** message  \* **@return**  \* **@throws** NoSuchAlgorithmException  \* **@throws** UnsupportedEncodingException  \*/  **public** **static** String md5Sum(String message) **throws** NoSuchAlgorithmException, UnsupportedEncodingException{  MessageDigest digest = MessageDigest.*getInstance*("MD5");  **byte**[] bytes = digest.digest(message.getBytes("UTF-8"));  String md5 = Hex.*encodeHexString*(bytes);  **return** md5;  } |

|  |
| --- |
| /\*\*  \* 使用Apache提供commons-codec.jar做md5摘要  \*  \* **@param** message  \* **@return**  \*/  **public** **static** String md5Sum2(String message){  **return** org.apache.commons.codec.digest.DigestUtils.*md5Hex*(message);  } |

一般破解方法：字典法，就是将常用密码生成MD5值字典，然后反向查找达到破解目的，因此建议使用强密码。

2、SHA1：SHA1生产160位摘要，以16进制显示40位。与SHA1同一个家族还有SHA224、SHA256、SHA384、SHA512，产生的位数不一样。

|  |
| --- |
| /\*\*  \* 使用jdk提供生成SHA1摘要。  \* 使用Apache提供commons-codec.jar的Hex做编码转换  \* SHA1生成160位摘要，SHA256生成256位摘要，SHA384生成384位摘要，SHA512生成512位摘要  \*  \* **@param** message  \* **@return**  \* **@throws** NoSuchAlgorithmException  \*/  **public** **static** String sha1Sum(String message) **throws** NoSuchAlgorithmException{  MessageDigest digest = MessageDigest.*getInstance*("SHA-1");  **byte**[] bytes = digest.digest(message.getBytes());  String sha1 = Hex.*encodeHexString*(bytes);  **return** sha1;  } |

|  |
| --- |
| /\*\*  \* 使用Apache提供commons-codec.jar做sha1摘要  \*  \* **@param** message  \* **@return**  \*/  **public** **static** String sha1Sum2(String message){  **return** org.apache.commons.codec.digest.DigestUtils.*sha1Hex*(message);  } |

## 签名与验签

SHA1withRSA做签名，先进行SHA1签名，再对签名结果在做RSA加密。

类似还有MD5withRSA.

|  |
| --- |
| /\*\*  \* SHA1withRSA做签名，先进行SHA1签名，再对签名结果在做RSA加密。  \* 类似还有MD5withRSA.  \*  \*/  **public** **class** SHA1withRSAUtils {  **private** **static** **final** String ***KEY\_ALGORITHM*** = "SHA1withRSA";  //private static final String KEY\_ALGORITHM = "MD5withRSA";    **private** **static** **final** String ***ENCODING*** = "UTF-8";    /\*\*  \* 签名  \*  \* **@param** data  \* **@param** rsaPrivateKey  \* **@return**  \* **@throws** Exception  \*/  **public** **static** **byte**[] sign(String data, RSAPrivateKey rsaPrivateKey) **throws** Exception{  //获取Signature对象  Signature sign = Signature.*getInstance*(***KEY\_ALGORITHM***);  //初始化sign对象  sign.initSign(rsaPrivateKey);  //签名  sign.update(data.getBytes(***ENCODING***));  **byte**[] bytes = sign.sign();    **return** bytes;  }    /\*\*  \* 验签  \*  \* **@param** bytes  \* **@param** data  \* **@param** rsaPublicKey  \* **@return**  \* **@throws** Exception  \*/  **public** **static** **boolean** verify(**byte**[] bytes, String data, RSAPublicKey rsaPublicKey) **throws** Exception{  //获取Signature对象  Signature sign = Signature.*getInstance*(***KEY\_ALGORITHM***);  //初始化sign对象  sign.initVerify(rsaPublicKey);  //验签  sign.update(data.getBytes(***ENCODING***));  **return** sign.verify(bytes);  }    **public** **static** **void** main(String[] args) **throws** Exception {  //密钥对生产：http://web.chacuo.net/netrsakeypair  String priKey = "";  String pubKey = "MIGfMA0GCSqGSIb3DQEBAQUAA4GNADCBiQKBgQDWlJp3G5MOxfBYh7X+2RjpdQkT5D6OFKapCsxwVYmLOtYgE79cbsoS6HaQMURzLcJaosjfNY9OGE7kJZKf3Lezv7dVO/0hE9BCEJIeIgrwo6yW+KymHN0caV7NSwJRw1o73LrLGpBYyr44sWpFA6VuZDcEhrAsIndDLiotkniF7wIDAQAB";    String message = "rsa";    RSAPrivateKey rsaPrivateKey = RSAUtils.*getRSAPrivateKey*(priKey);  RSAPublicKey rsaPublicKey = RSAUtils.*getRSAPublicKey*(pubKey);    **byte**[] bytes = *sign*(message, rsaPrivateKey);  **boolean** verify = *verify*(bytes, message, rsaPublicKey);  System.***out***.println(verify);  }  } |

## 对称加密

对称加密是加密与解密使用相同的密钥。常见算法有AES、DES

对称加密与非对称加密区别：对称加密算法只是为了区分非对称加密算法。其中鲜明的特点是对称加密是加密解密使用相同的密钥，而非对称加密加密和解密时使用的密钥不一样。对于大部分情况我们都使用对称加密，而对称加密的密钥交换时使用非对称加密，这有效保护密钥的安全。非对称加密加密和解密密钥不同，那么它的安全性是无疑最高的，但是它加密解密的速度很慢，不适合对大数据加密。而对称加密加密速度快，因此混合使用最好。

**对称加密是最快速、最简单的一种加密方式，加密（encryption）与解密（decryption）用的是同样的密钥（secret key）。对称加密有很多种算法，由于它效率很高，所以被广泛使用在很多加密协议的核心当中。**

**对称加密通常使用的是相对较小的密钥，一般小于256 bit。因为密钥越大，加密越强，但加密与解密的过程越慢。如果你只用1 bit来做这个密钥，那黑客们可以先试着用0来解密，不行的话就再用1解；但如果你的密钥有1 MB大，黑客们可能永远也无法破解，但加密和解密的过程要花费很长的时间。密钥的大小既要照顾到安全性，也要照顾到效率，是一个trade-off**。

DES：比较老的算法，一共有三个参数入口（原文，密钥，加密模式）。而3DES只是DES的一种模式，是以DES为基础更安全的变形，对数据进行了三次加密，也是被指定为AES的过渡算法。

DES：已经被破解，已经不用了。

AES:高级加密标准，新一代标准，加密速度更快，安全性更高（不用说优先选择）

参数：”AES/ECB/PKCS5Padding”在加密和解密时必须相同，可以直接写”AES”,这样就是使用默认模式（C#和java默认的模式不一样，C#中默认的是这种,java的默认待研究）。分别的意思为：AES是加密算法，ECB是工作模式，PKCS5Padding是填充方式。

AES是分组加密算法，也称块加密。每一组16字节。这样明文就会分成多块。当有一块不足16字节时就会进行填充。

一共有四种工作模式：

ECB 电子密码本模式：相同的明文块产生相同的密文块，容易并行运算，但也可能对明文进行攻击。

CBC 加密分组链接模式：一块明文加密后和上一块密文进行链接，不利于并行，但安全性比ECB好，是SSL,IPSec的标准。

CFB 加密反馈模式：将上一次密文与密钥运算，再加密。隐藏明文模式，不利于并行，误差传递。

OFB 输出反馈模式：将上一次处理过的密钥与密钥运算，再加密。隐藏明文模式，不利于并行，有可能明文攻击，误差传递。

PKCS5Padding的填充方式是差多少字节就填数字多少；刚好每一不足16字节时，那么就会加一组填充为16.还有其他填充模式【Nopadding,ISO10126Padding】（不影响算法，加密解密时一致就行）。

|  |
| --- |
| /\*\*  \* 对称加密，已经被破解，不再使用。  \* 3DES是DES升级版  \*  \*/  **public** **class** DESUtils {    **private** **static** **final** String ***DEFAULT\_CIPHER\_ALGORITHM*** = "DES/ECB/PKCS5Padding";  **private** **static** **final** String ***KEY\_ALGORITHM*** = "DES";    **private** **static** **final** String ***ENCODING*** = "UTF-8";    **public** **static** **byte**[] aesEncrypt(**final** String message, **final** String key) **throws** Exception{  //获取cipher对象  Cipher cipher = Cipher.*getInstance*(***DEFAULT\_CIPHER\_ALGORITHM***);  //初始化cipher对象  cipher.init(Cipher.***ENCRYPT\_MODE***, *getSecretKey*(key), **new** SecureRandom());  //执行操作  **byte**[] bytes = cipher.doFinal(message.getBytes(***ENCODING***));  **return** bytes;  }    **public** **static** String aesDecrypt(**final** **byte**[] bytes, **final** String key) **throws** Exception{  //获取cipher对象  Cipher cipher = Cipher.*getInstance*(***DEFAULT\_CIPHER\_ALGORITHM***);  //初始化cipher对象  cipher.init(Cipher.***DECRYPT\_MODE***, *getSecretKey*(key), **new** SecureRandom());  //执行操作  **byte**[] results = cipher.doFinal(bytes);  **return** **new** String(results, ***ENCODING***);  }    **private** **static** SecretKey getSecretKey(**final** String key) **throws** Exception{  // 从原始密钥数据创建DESKeySpec对象  DESKeySpec dks = **new** DESKeySpec(key.getBytes(***ENCODING***));  // 创建一个密钥工厂，然后用它把DESKeySpec转换成SecretKey对象  SecretKeyFactory factory = SecretKeyFactory.*getInstance*(***KEY\_ALGORITHM***);  SecretKey secretKey = factory.generateSecret(dks);  **return** secretKey;  }    **public** **static** **void** main(String[] args) **throws** Exception {  String message = "des";  String key = "12345678";//密码一定要是8的倍数  **byte**[] bytes = *aesEncrypt*(message, key);  System.***out***.println(**new** String(bytes, ***ENCODING***));    String result = *aesDecrypt*(bytes, key);  System.***out***.println(result);  }  } |

|  |
| --- |
| /\*\*  \* 最常用的对称加密算法  \*  \*/  **public** **class** AESUtils {    /\*\*  \* 加密和解密必须用相同  \*  \* AES：加密算法  \* ECB：工作模式  \* PKCS5Padding：填充方式  \*/  **private** **static** **final** String ***DEFAULT\_CIPHER\_ALGORITHM*** = "AES/ECB/PKCS5Padding";    **private** **static** **final** String ***KEY\_ALGORITHM*** = "AES";    **private** **static** **final** String ***ENCODING*** = "UTF-8";  /\*\*  \* **@param** message  \* **@param** key  \* **@return**  \* **@throws** Exception  \*/  **public** **static** **byte**[] aesEncrypt(**final** String message, **final** String key) **throws** Exception{  //获取Cipher对象  Cipher cipher = Cipher.*getInstance*(***DEFAULT\_CIPHER\_ALGORITHM***);  //使用密钥初始化Cipher，设置opmode=1(解密)  cipher.init(Cipher.***ENCRYPT\_MODE***, *getSecretKey*(key));  //执行操作  **byte**[] bytes = cipher.doFinal(message.getBytes("UTF-8"));    **return** bytes;  }    /\*\*  \* **@param** bytes  \* **@param** key  \* **@return**  \* **@throws** Exception  \*/  **public** **static** String aesDecrypt(**final** **byte**[] bytes, **final** String key) **throws** Exception{  //获取cipher对象  Cipher cipher = Cipher.*getInstance*(***DEFAULT\_CIPHER\_ALGORITHM***);  //使用密钥初始化cipher，设置opmode=2(解密)  cipher.init(Cipher.***DECRYPT\_MODE***, *getSecretKey*(key));  //执行操作  **byte**[] results = cipher.doFinal(bytes);  //byte数组转字符串  String message = **new** String(results, ***ENCODING***);  **return** message;  }    **private** **static** SecretKeySpec getSecretKey(**final** String key) **throws** Exception {  //返回生成指定算法密钥生成器的 KeyGenerator 对象  KeyGenerator keyGenerator = KeyGenerator.*getInstance*(***KEY\_ALGORITHM***);  //AES 要求密钥长度为 128  keyGenerator.init(128, **new** SecureRandom(key.getBytes(***ENCODING***)));  //生成一个密钥  SecretKey secretKey = keyGenerator.generateKey();    **return** **new** SecretKeySpec(secretKey.getEncoded(), ***KEY\_ALGORITHM***);  }    **public** **static** **void** main(String[] args) **throws** Exception {  String message = "aes";  String key = "1";  **byte**[] bytes = *aesEncrypt*(message, key);  System.***out***.println(**new** String(bytes, "UTF-8"));    String result = *aesDecrypt*(bytes, key);  System.***out***.println(result);  }  } |

## 非对称加密

加密跟解密使用不同的密钥。常见算法：RSA

|  |
| --- |
| /\*\*  \* 非对称加密：RSA  \*  \*/  **public** **class** RSAUtils {    **private** **static** **final** String ***DEFAULT\_CIPHER\_ALGORITHM*** = "RSA/ECB/PKCS1Padding";    **private** **static** **final** String ***KEY\_ALGORITHM*** = "RSA";    **private** **static** **final** String ***ENCODING*** = "UTF-8";    /\*\*  \* 根据公钥字符串生产公钥  \*  \* **@param** publicKey  \* **@return**  \* **@throws** Exception  \*/  **public** **static** RSAPublicKey getRSAPublicKey(String publicKey) **throws** Exception{  KeyFactory factory = KeyFactory.*getInstance*(***KEY\_ALGORITHM***);  X509EncodedKeySpec spec = **new** X509EncodedKeySpec(Base64.*decode*(publicKey));  RSAPublicKey rsaPublicKey = (RSAPublicKey) factory.generatePublic(spec);  **return** rsaPublicKey;  }    /\*\*  \* 根据私钥字符串生产私钥  \*  \* **@param** privateKey  \* **@return**  \* **@throws** Exception  \*/  **public** **static** RSAPrivateKey getRSAPrivateKey(String privateKey) **throws** Exception{  KeyFactory factory = KeyFactory.*getInstance*(***KEY\_ALGORITHM***);  PKCS8EncodedKeySpec keySpec = **new** PKCS8EncodedKeySpec(Base64.*decode*(privateKey));  RSAPrivateKey rsaPrivateKey = (RSAPrivateKey) factory.generatePrivate(keySpec);  **return** rsaPrivateKey;  }    /\*\*  \* 公钥加密  \*  \* **@param** data  \* **@param** rsaPublicKey  \* **@return**  \* **@throws** Exception  \*/  **public** **static** **byte**[] publicEncrypt(String data, RSAPublicKey rsaPublicKey) **throws** Exception{  Cipher cipher = Cipher.*getInstance*(***DEFAULT\_CIPHER\_ALGORITHM***);  cipher.init(Cipher.***ENCRYPT\_MODE***, rsaPublicKey);    **int** maxBlock = rsaPublicKey.getModulus().bitLength()/8 -11;  ByteArrayOutputStream byteArrayOutputStream = **new** ByteArrayOutputStream();  **int** offset = 0;  **byte**[] buff;  **int** i = 0;  **byte**[] datas = data.getBytes(***ENCODING***);  **while**(datas.length > offset){  **if**(datas.length-offset > maxBlock){  buff = cipher.doFinal(datas, offset, maxBlock);  }**else**{  buff = cipher.doFinal(datas, offset, datas.length-offset);  }  byteArrayOutputStream.write(buff, 0, buff.length);  i++;  offset = i \* maxBlock;  }  **byte**[] resultDatas = byteArrayOutputStream.toByteArray();  **return** resultDatas;  }    /\*\*  \* 私钥解密  \*  \* **@param** datas  \* **@param** rsaPrivateKey  \* **@return**  \*/  **public** **static** **byte**[] privateDecrypt(**byte**[] datas, RSAPrivateKey rsaPrivateKey) **throws** Exception{  Cipher cipher = Cipher.*getInstance*(***DEFAULT\_CIPHER\_ALGORITHM***);  cipher.init(Cipher.***DECRYPT\_MODE***, rsaPrivateKey);    **int** maxBlock = rsaPrivateKey.getModulus().bitLength()/8;  ByteArrayOutputStream byteArrayOutputStream = **new** ByteArrayOutputStream();  **int** offset = 0;  **byte**[] buff;  **int** i = 0;  **while**(datas.length > offset){  **if**(datas.length-offset > maxBlock){  buff = cipher.doFinal(datas, offset, maxBlock);  }**else**{  buff = cipher.doFinal(datas, offset, datas.length-offset);  }  byteArrayOutputStream.write(buff, 0, buff.length);  i++;  offset = i \* maxBlock;  }  **byte**[] resultDatas = byteArrayOutputStream.toByteArray();  **return** resultDatas;  }    //------------------------------------------------------------------------------    /\*\*  \* 私钥加密  \*  \* **@param** data  \* **@param** rsaPrivateKey  \* **@return**  \*/  **public** **static** **byte**[] privateEncrypt(String data, RSAPrivateKey rsaPrivateKey) **throws** Exception{  Cipher cipher = Cipher.*getInstance*(***DEFAULT\_CIPHER\_ALGORITHM***);  cipher.init(Cipher.***ENCRYPT\_MODE***, rsaPrivateKey);    **int** maxBlock = rsaPrivateKey.getModulus().bitLength()/8 -11;  ByteArrayOutputStream byteArrayOutputStream = **new** ByteArrayOutputStream();  **int** offset = 0;  **byte**[] buff;  **int** i = 0;  **byte**[] datas = data.getBytes(***ENCODING***);  **while**(datas.length > offset){  **if**(datas.length-offset > maxBlock){  buff = cipher.doFinal(datas, offset, maxBlock);  }**else**{  buff = cipher.doFinal(datas, offset, datas.length-offset);  }  byteArrayOutputStream.write(buff, 0, buff.length);  i++;  offset = i \* maxBlock;  }  **byte**[] resultDatas = byteArrayOutputStream.toByteArray();  **return** resultDatas;  }    /\*\*  \* 公钥解密  \*  \* **@param** datas  \* **@param** rsaPublicKey  \* **@return**  \* **@throws** Exception  \*/  **public** **static** **byte**[] publicDecrypt(**byte**[] datas, RSAPublicKey rsaPublicKey) **throws** Exception{  Cipher cipher = Cipher.*getInstance*(***DEFAULT\_CIPHER\_ALGORITHM***);  cipher.init(Cipher.***DECRYPT\_MODE***, rsaPublicKey);    **int** maxBlock = rsaPublicKey.getModulus().bitLength()/8;  ByteArrayOutputStream byteArrayOutputStream = **new** ByteArrayOutputStream();  **int** offset = 0;  **byte**[] buff;  **int** i = 0;  **while**(datas.length > offset){  **if**(datas.length-offset > maxBlock){  buff = cipher.doFinal(datas, offset, maxBlock);  }**else**{  buff = cipher.doFinal(datas, offset, datas.length-offset);  }  byteArrayOutputStream.write(buff, 0, buff.length);  i++;  offset = i \* maxBlock;  }  **byte**[] resultDatas = byteArrayOutputStream.toByteArray();  **return** resultDatas;  }    **public** **static** **void** main(String[] args) **throws** Exception {  //密钥对生产：http://web.chacuo.net/netrsakeypair  String priKey = "";  String pubKey = "MIGfMA0GCSqGSIb3DQEBAQUAA4GNADCBiQKBgQDWlJp3G5MOxfBYh7X+2RjpdQkT5D6OFKapCsxwVYmLOtYgE79cbsoS6HaQMURzLcJaosjfNY9OGE7kJZKf3Lezv7dVO/0hE9BCEJIeIgrwo6yW+KymHN0caV7NSwJRw1o73LrLGpBYyr44sWpFA6VuZDcEhrAsIndDLiotkniF7wIDAQAB";    String message = "rsa";    RSAPrivateKey rsaPrivateKey = *getRSAPrivateKey*(priKey);  RSAPublicKey rsaPublicKey = *getRSAPublicKey*(pubKey);    **byte**[] bytes1 = *publicEncrypt*(message, rsaPublicKey);  **byte**[] datas1 = *privateDecrypt*(bytes1, rsaPrivateKey);  System.***out***.println(**new** String(datas1, ***ENCODING***));    **byte**[] bytes2 = *privateEncrypt*(message, rsaPrivateKey);  **byte**[] datas2 = *publicDecrypt*(bytes2, rsaPublicKey);  System.***out***.println(**new** String(datas2, ***ENCODING***));  } |

## 编码算法

常见的编码有Base64,HEX和对URL的编码。这都是为了实际需要才进行的编码。HEX是编码成16进制字符，MD5一般就是以HEX进行编码，这不说了。

|  |
| --- |
| /\*\*  \* base64编码解码。  \* 在jdk1.8之前，用sun.misc.Base64Encoder和sun.misc.Base64Decoder，能力较差。  \* 在jdk1.8之前，通常用Apache的commons-codec.jar中Base64来做编码。  \*  \* 在jdk1.8，jdk提供了java.util.Base64.Decoder和java.util.Base64.Encoder做编码  \* 结论：如果是jdk1.8之前，用Apache来做编码；  \* 如果是jdk1.8，用jdk提供包来做编码  \*  \*/  **public** **class** Base64Utils {    **public** **static** String encode(**byte**[] bytes){  Encoder encoder = Base64.*getEncoder*();  **return** encoder.encodeToString(bytes);  }    **public** **static** **byte**[] decode(String data){  Decoder decoder = Base64.*getDecoder*();  **return** decoder.decode(data);  }    **public** **static** **void** main(String[] args) **throws** UnsupportedEncodingException {  String data = "base64";    String base = *encode*(data.getBytes("UTF-8"));  System.***out***.println(base);    System.***out***.println(**new** String(*decode*(base), "UTF-8"));  }  } |

|  |
| --- |
| /\*\*  \* URL编码解码  \*  \*/  **public** **class** URLUtils {    **private** **static** **final** String ***ENCODING*** = "UTF-8";  /\*\*  \* 编码  \*  \* **@param** src  \* **@return**  \* **@throws** UnsupportedEncodingException  \*/  **public** **static** String encoder(String src) **throws** UnsupportedEncodingException{  String des = URLEncoder.*encode*(src, ***ENCODING***);  **return** des;  }    /\*\*  \* 解码  \*  \* **@param** src  \* **@return**  \* **@throws** UnsupportedEncodingException  \*/  **public** **static** String decoder(String src) **throws** UnsupportedEncodingException{  String des = URLDecoder.*decode*(src, ***ENCODING***);  **return** des;  }    **public** **static** **void** main(String[] args) **throws** UnsupportedEncodingException {  String url = "http://baidu.com/对的";  String encoder = *encoder*(url);  System.***out***.println(encoder);    String decoder = *decoder*(encoder);  System.***out***.println(decoder);  }  } |

# 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