**Java关键技术汇集**

**2014.10**

**修订记录**

|  |  |  |  |
| --- | --- | --- | --- |
| **版本** | **修订内容** | **作者** | **日期** |
| 1.0 | 初稿完成 | 袁欢 | 2014/10/5 |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |

**目 录**

[1 写在前面的话 8](#_Toc400460581)

[2 Java数据类型 8](#_Toc400460582)

[2.1 基本数据类型 8](#_Toc400460583)

[2.1.1 boolean 8](#_Toc400460584)

[2.1.2 char 9](#_Toc400460585)

[2.1.3 byte 9](#_Toc400460586)

[2.1.4 short 9](#_Toc400460587)

[2.1.5 int 9](#_Toc400460588)

[2.1.6 long 10](#_Toc400460589)

[2.1.7 float 10](#_Toc400460590)

[2.1.8 double 10](#_Toc400460591)

[2.2 数据类型转换 10](#_Toc400460592)

[2.2.1 自动类型转换 10](#_Toc400460593)

[2.2.2 强制类型转换 11](#_Toc400460594)

[2.2.3 包装类过渡类型转换 11](#_Toc400460595)

[2.2.4 字符串型与其他数据类型的转换 11](#_Toc400460596)

[2.2.5 将字符型直接作为数值转换为其他数据类型 11](#_Toc400460597)

[2.3 对象型数据类型 12](#_Toc400460598)

[2.3.1 ArrayList 12](#_Toc400460599)

[2.3.2 LinkedList 12](#_Toc400460600)

[2.3.3 HashMap 12](#_Toc400460601)

[2.3.4 HashTable 12](#_Toc400460602)

[2.3.5 HashSet 12](#_Toc400460603)

[2.3.6 Stack 12](#_Toc400460604)

[2.3.7 Queue 12](#_Toc400460605)

[2.3.8 Vector 12](#_Toc400460606)

[2.3.9 Iterator 13](#_Toc400460607)

[2.3.10 TreeSet 13](#_Toc400460608)

[2.3.11 Set 13](#_Toc400460609)

[3 多线程编程 13](#_Toc400460610)

[3.1 Java线程概念和原理 13](#_Toc400460611)

[3.2 Java线程创建和启动 13](#_Toc400460612)

[3.3 Java线程状态转换 13](#_Toc400460613)

[3.4 Java线程调度 14](#_Toc400460614)

[3.4.1 休眠 14](#_Toc400460615)

[3.4.2 优先级 14](#_Toc400460616)

[3.4.3 让步 14](#_Toc400460617)

[3.4.4 合并 14](#_Toc400460618)

[3.4.5 守护线程 14](#_Toc400460619)

[3.5 Java线程同步与锁 14](#_Toc400460620)

[3.5.1 同步方法 14](#_Toc400460621)

[3.5.2 同步块 15](#_Toc400460622)

[3.6 Java线程交互 15](#_Toc400460623)

[3.7 Java线程池 15](#_Toc400460624)

[4 设计模式 15](#_Toc400460625)

[4.1 设计模式的六大原则 15](#_Toc400460626)

[4.2 工厂方法模式 16](#_Toc400460627)

[4.3 抽象工厂模式 16](#_Toc400460628)

[4.4 单例模式 16](#_Toc400460629)

[4.5 建造者模式 16](#_Toc400460630)

[4.6 原型模式 16](#_Toc400460631)

[4.7 适配器模式 17](#_Toc400460632)

[4.8 装饰器模式 17](#_Toc400460633)

[4.9 代理模式 17](#_Toc400460634)

[4.10 外观模式 17](#_Toc400460635)

[4.11 桥接模式 17](#_Toc400460636)

[4.12 组合模式 17](#_Toc400460637)

[4.13 享元模式 17](#_Toc400460638)

[4.14 策略模式 17](#_Toc400460639)

[4.15 模板方法模式 17](#_Toc400460640)

[4.16 观察者模式 17](#_Toc400460641)

[4.17 迭代子模式 17](#_Toc400460642)

[4.18 责任链模式 18](#_Toc400460643)

[4.19 命令模式 18](#_Toc400460644)

[4.20 备忘录模式 18](#_Toc400460645)

[4.21 状态模式 18](#_Toc400460646)

[4.22 访问者模式 18](#_Toc400460647)

[4.23 中介者模式 18](#_Toc400460648)

[4.24 解释器模式 18](#_Toc400460649)

[5 JVM 18](#_Toc400460650)

[5.1 JVM体系结构与工作方式 18](#_Toc400460651)

[5.2 JVM内存管理 19](#_Toc400460652)

[5.2.1 内存泄露 19](#_Toc400460653)

[5.3 Class Loader工作机制 19](#_Toc400460654)

[5.4 JVM性能优化 19](#_Toc400460655)

[5.4.1 性能优化工具 19](#_Toc400460656)

[5.4.2 优化参数 19](#_Toc400460657)

[5.5 GC工作原理 19](#_Toc400460658)

[6 Spring框架 20](#_Toc400460659)

[6.1 Spring框架设计理念和设计模式 20](#_Toc400460660)

[6.1.1 核心组件 20](#_Toc400460661)

[6.1.2 DI 20](#_Toc400460662)

[6.1.3 Ioc 20](#_Toc400460663)

[6.1.4 AOP 20](#_Toc400460664)

[6.2 Spring MVC 20](#_Toc400460665)

[6.2.1 总体设计 20](#_Toc400460666)

[6.2.2 Control设计 21](#_Toc400460667)

[6.2.3 Model设计 21](#_Toc400460668)

[6.2.4 View设计 21](#_Toc400460669)

[6.3 Spring Security 21](#_Toc400460670)

[7 ORM框架 21](#_Toc400460671)

[7.1 iBatis 21](#_Toc400460672)

[7.2 Hibernate 21](#_Toc400460673)

[7.3 iBatis与Hibernate对比 21](#_Toc400460674)

[8 Java IO工作机制 22](#_Toc400460675)

[8.1 Java的IO类库的基本架构 22](#_Toc400460676)

[8.1.1 基于字节的IO操作接口 22](#_Toc400460677)

[8.1.2 基于字符的IO操作接口 22](#_Toc400460678)

[8.1.3 字节与字符的转化接口 22](#_Toc400460679)

[8.2 磁盘IO工作机制 22](#_Toc400460680)

[8.2.1 访问文件的方式 22](#_Toc400460681)

[8.2.2 Java访问磁盘文件 22](#_Toc400460682)

[8.2.3 Java序列化技术 22](#_Toc400460683)

[8.3 网络IO工作机制 22](#_Toc400460684)

[8.3.1 TCP状态变化 22](#_Toc400460685)

[8.3.2 影响网络传输的因素 23](#_Toc400460686)

[8.3.3 Java Socket的工作机制 23](#_Toc400460687)

[8.3.4 建立通信链路 23](#_Toc400460688)

[8.3.5 数据传输 23](#_Toc400460689)

[8.4 NIO的工作方式 23](#_Toc400460690)

[8.4.1 BIO带来的挑战 23](#_Toc400460691)

[8.4.2 NIO的工作机制 23](#_Toc400460692)

[8.4.3 Buffer的工作机制 23](#_Toc400460693)

[8.4.4 NIO的数据访问方式 23](#_Toc400460694)

[8.5 IO调优 24](#_Toc400460695)

[8.5.1 磁盘IO优化 24](#_Toc400460696)

[8.5.2 TCP网络参数优化 24](#_Toc400460697)

[8.5.3 网络IO优化 24](#_Toc400460698)

[9 Servlet工作原理 24](#_Toc400460699)

[9.1 Servlet容器 24](#_Toc400460700)

[9.2 创建Servlet实例 24](#_Toc400460701)

[9.3 Servlet体系结构 24](#_Toc400460702)

[9.4 Servlet如何工作 24](#_Toc400460703)

[9.5 Servlet中的Listener 24](#_Toc400460704)

[9.6 Filter如何工作 25](#_Toc400460705)

[9.7 Servlet中的url-patten 25](#_Toc400460706)

[10 Session和Cookie 25](#_Toc400460707)

[10.1 理解Cookie 25](#_Toc400460708)

[10.2 理解Session 25](#_Toc400460709)

[10.3 Cookie安全问题 25](#_Toc400460710)

[10.4 分布式Session框架 25](#_Toc400460711)

[10.5 Cookie压缩 25](#_Toc400460712)

[10.6 多终端Session统一 26](#_Toc400460713)

[11 其它技术点 26](#_Toc400460714)

[11.1 序列化和反序列化 26](#_Toc400460715)

[11.1.1 基本概念 26](#_Toc400460716)

[11.1.2 序列化 ID 问题 26](#_Toc400460717)

[11.1.3 父类的序列化与 Transient 关键字 27](#_Toc400460718)

[11.1.4 对敏感字段加密 27](#_Toc400460719)

[11.1.5 序列化存储规则 28](#_Toc400460720)

[11.2 浅拷贝和深拷贝 28](#_Toc400460721)

[11.2.1 clone方法 28](#_Toc400460722)

[11.2.2 什么是浅拷贝 28](#_Toc400460723)

[11.2.3 什么是深拷贝 28](#_Toc400460724)

[11.2.4 通过序列化实现深拷贝 28](#_Toc400460725)

[11.3 Date 29](#_Toc400460726)

[11.4 抽象类和接口 29](#_Toc400460727)

[11.5 参数传递 29](#_Toc400460728)

[11.6 静态变量和私有变量 31](#_Toc400460729)

[11.7 正则表达式 36](#_Toc400460730)

[11.8 36](#_Toc400460731)

[11.9 36](#_Toc400460732)

[11.10 36](#_Toc400460733)

[11.11 37](#_Toc400460734)

[11.12 37](#_Toc400460735)

[11.13 37](#_Toc400460736)

[11.14 37](#_Toc400460737)

[11.15 37](#_Toc400460738)

[12 相关开发工具和技能 38](#_Toc400460739)

[12.1 开源框架和技术 38](#_Toc400460740)

[12.2 版本管理工具 38](#_Toc400460741)

[12.2.1 Git 38](#_Toc400460742)

[12.2.2 SVN 38](#_Toc400460743)

[12.3 Java编程规范 38](#_Toc400460744)

[12.4 eclipse中Java代码格式化 38](#_Toc400460745)

[12.5 代码审查 39](#_Toc400460746)

[12.6 项目管理工具 39](#_Toc400460747)

[12.7 Wiki系统 39](#_Toc400460748)

[12.8 Bug追踪工具 39](#_Toc400460749)

[12.9 UML建模工具 39](#_Toc400460750)

[12.10 文档编写能力 40](#_Toc400460751)

[12.11 技术博客 40](#_Toc400460752)

[12.12 github 40](#_Toc400460753)

# 写在前面的话

一直以来，Java都是一种很普遍的开发语言，功能强大，其入门相对比较简单，但是要想精通Java编程，还是需要对Java知识点有很深入的理解的，因此萌生了一个将Java相关知识点汇总的想法，以便便于自己更加理解Java开发。待本文完善之后，将会公开出来，希望能帮助到更多的Java爱好者。

本文的主要目的是梳理一遍Java开发技术，将其中非常重要的一些概念、框架、技巧、细节等汇总，便于查看。

本文内容来源于互联网Java技术文档、Java书籍以及本人经验汇总而成，主要是为了在工作、学习中便于查看，可能会有遗漏、或者错误之处，请多指正。

如果在文中发现有内容解释的不详细，可以对应在网上搜索相关知识点以便辅助理解，本文尽量做到面面俱到又保持精简。

本文面向的读者主要有：

* + 对Java有过一些基础开发经验，但是对很多细节不很了解的开发人员；
  + 即将参加Java相关职位的面试的程序员们，可以帮助大家对Java知识整体上有一个梳理；

# Java三大特性

## 封装

封装从字面上来理解就是包装的意思，专业点就是信息隐藏，是指利用抽象数据类型将数据和基于数据的操作封装在一起，使其构成一个不可分割的独立实体，数据被保护在抽象数据类型的内部，尽可能地隐藏内部的细节，只保留一些对外接口使之与外部发生联系。系统的其他对象只能通过包裹在数据外面的已经授权的操作来与这个封装的对象进行交流和交互。也就是说用户是无需知道对象内部的细节（当然也无从知道），但可以通过该对象对外的提供的接口来访问该对象。

对于封装而言，一个对象它所封装的是自己的属性和方法，所以它是不需要依赖其他对象就可以完成自己的操作。

使用封装有几大好处：

1、良好的封装能够减少耦合。

2、类内部的结构可以自由修改。

3、可以对成员进行更精确的控制。

4、隐藏信息，实现细节。

## 继承

继承是使用已存在的类的定义作为基础建立新类的技术，新类的定义可以增加新的数据或新的功能，也可以用父类的功能，但不能选择性地继承父类。通过使用继承我们能够非常方便地复用以前的代码，能够大大的提高开发的效率。

继承定义了类如何相互关联，共享特性。

对于若干个相同或者相识的类，我们可以抽象出他们共有的行为或者属相并将其定义成一个父类或者超类，然后用这些类继承该父类，他们不仅可以拥有父类的属性、方法还可以定义自己独有的属性或者方法。

同时在使用继承时需要记住三句话：

1、子类拥有父类非private的属性和方法。

2、子类可以拥有自己属性和方法，即子类可以对父类进行扩展。

3、子类可以用自己的方式实现父类的方法。

对于构造器而言，它只能够被调用，而不能被继承。

对于protected而言，它指明就类用户而言，他是private，但是对于任何继承与此类的子类而言或者其他任何位于同一个包的类而言，他却是可以访问的。

将子类转换成父类，在继承关系上面是向上移动的，所以一般称之为向上转型。由于向上转型是从一个叫专用类型向较通用类型转换，所以它总是安全的，唯一发生变化的可能就是属性和方法的丢失。这就是为什么编译器在“未曾明确表示转型”活“未曾指定特殊标记”的情况下，仍然允许向上转型的原因。

上面讲了继承所带来的诸多好处，那我们是不是就可以大肆地使用继承呢？送你一句话：慎用继承。

首先我们需要明确，继承存在如下缺陷：

1、父类变，子类就必须变。

2、继承破坏了封装，对于父类而言，它的实现细节对与子类来说都是透明的。

3、继承是一种强耦合关系。

所以说当我们使用继承的时候，我们需要确信使用继承确实是有效可行的办法。那么到底要不要使用继承呢？《Think in java》中提供了解决办法：问一问自己是否需要从子类向父类进行向上转型。如果必须向上转型，则继承是必要的，但是如果不需要，则应当好好考虑自己是否需要继承。

## 多态

多态是同一个行为具有多个不同表现形式或形态的能力。

多态性是对象多种表现形式的体现。

比如我们说"宠物"这个对象，它就有很多不同的表达或实现，比如有小猫、小狗、蜥蜴等等。那么我到宠物店说"请给我一只宠物"，服务员给我小猫、小狗或者蜥蜴都可以，我们就说"宠物"这个对象就具备多态性。

所谓多态就是指程序中定义的引用变量所指向的具体类型和通过该引用变量发出的方法调用在编程时并不确定，而是在程序运行期间才确定，即一个引用变量到底会指向哪个类的实例对象，该引用变量发出的方法调用到底是哪个类中实现的方法，必须在由程序运行期间才能决定。因为在程序运行时才确定具体的类，这样，不用修改源程序代码，就可以让引用变量绑定到各种不同的类实现上，从而导致该引用调用的具体方法随之改变，即不修改程序代码就可以改变程序运行时所绑定的具体代码，让程序可以选择多个运行状态，这就是多态性。

# Java数据类型

## 基本数据类型

Java语言提供了八种基本类型。六种数字类型（四个整数型，两个浮点型），一种字符类型，还有一种布尔型。

### boolean

boolean数据类型表示一位的信息；

只有两个取值：true和false；

这种类型只作为一种标志来记录true/false情况；

默认值是false；

例子：boolean one = true。

### char

char类型是一个单一的16位Unicode字符；

最小值是’\u0000’（即为0）；

最大值是’\uffff’（即为65,535）；

char数据类型可以储存任何字符；

例子：char letter = ‘A’。

### byte

byte数据类型是8位、有符号的，以二进制补码表示的整数；

最小值是-128（-2^7）；

最大值是127（2^7-1）；

默认值是0；

byte类型用在大型数组中节约空间，主要代替整数，因为byte变量占用的空间只有int类型的四分之一；

例子：byte a = 100，byte b = -50。

### short

short数据类型是16位、有符号的以二进制补码表示的整数

最小值是-32768（-2^15）；

最大值是32767（2^15 - 1）；

Short数据类型也可以像byte那样节省空间。一个short变量是int型变量所占空间的二分之一；

默认值是0；

例子：short s = 1000，short r = -20000。

### int

int数据类型是32位、有符号的以二进制补码表示的整数；

最小值是-2,147,483,648（-2^31）；

最大值是2,147,485,647（2^31 - 1）；

一般地整型变量默认为int类型；

默认值是0；

例子：int a = 100000, int b = -200000。

### long

long数据类型是64位、有符号的以二进制补码表示的整数；

最小值是-9,223,372,036,854,775,808（-2^63）；

最大值是9,223,372,036,854,775,807（2^63 -1）；

这种类型主要使用在需要比较大整数的系统上；

默认值是0L；

例子： long a = 100000L，int b = -200000L。

### float

float数据类型是单精度、32位、符合IEEE 754标准的浮点数；

float在储存大型浮点数组的时候可节省内存空间；

默认值是0.0f；

浮点数不能用来表示精确的值，如货币；

例子：float f1 = 234.5f。

### double

double数据类型是双精度、64位、符合IEEE 754标准的浮点数；

浮点数的默认类型为double类型；

double类型同样不能表示精确的值，如货币；

默认值是0.0f；

例子：double d1 = 123.4。

## 数据类型转换

### 自动类型转换

在Java中，整型、实型、字符型被视为简单数据类型，这些类型由低到高分别为（byte，short，char）🡪 int 🡪long 🡪 float 🡪 double。简单数据类型之间的转换又可以分为：低级到高级的自动类型转换，高级到低级的强制类型转换、包装类过渡类型能够转换。

例如：

**byte** b = 0;

**int** i = b;

**long** l = i;

l = b;

**double** d = l;

### 强制类型转换

将高级变量转换为低级变量的时候，情况会复杂一点，可以使用强制类型转换。

例如：int i=99; byte b = (byte) i; char c= (char) i;

### 包装类过渡类型转换

Java中共有六个包装类，分别是Boolean、Character、Integer、Long、Float和Double，分别对应于boolean、char、int、long、float、double。而String和Data本身就是类，不存在包装类的概念。

例如：

double d = 100.0; Double D = new Double(d); int i = D.intValue();

### 字符串型与其他数据类型的转换

Object类派生的所有类都提供了toString()方法，即将该类转换为字符串。

例如：Integer i = new Integer(3); String s = i.toString();

### 将字符型直接作为数值转换为其他数据类型

将字符型变量转换为数值型变量实际上有两种对应关系：一种是将其转换成对应的ASCII码，另一种是转换关系，例如，’1’就是指数值1，而不是ASCII码，对于这种转换，可以使用Character的getNumericValue(char ch)方法。

## 对象型数据类型

### ArrayList

### LinkedList

### HashMap

### HashTable

### HashSet

### Stack

### Queue

### Vector

### Iterator

### TreeSet

### Set

# 多线程编程

参考链接：http://lavasoft.blog.51cto.com/62575/27069/

## Java线程概念和原理

## Java线程创建和启动

## Java线程状态转换

## Java线程调度

### 休眠

### 优先级

### 让步

### 合并

### 守护线程

## Java线程同步与锁

### 同步方法

### 同步块

## Java线程交互

## Java线程池

# 设计模式

参考链接：http://www.cnblogs.com/maowang1991/archive/2013/04/15/3023236.html

## 设计模式的六大原则

**1、开闭原则（Open Close Principle）**

开闭原则就是说**对扩展开放，对修改关闭**。在程序需要进行拓展的时候，不能去修改原有的代码，实现一个热插拔的效果。所以一句话概括就是：为了使程序的扩展性好，易于维护和升级。想要达到这样的效果，我们需要使用接口和抽象类，后面的具体设计中我们会提到这点。

**2、里氏代换原则（Liskov Substitution Principle）**

里氏代换原则(Liskov Substitution Principle LSP)面向对象设计的基本原则之一。 里氏代换原则中说，任何基类可以出现的地方，子类一定可以出现。 LSP是继承复用的基石，只有当衍生类可以替换掉基类，软件单位的功能不受到影响时，基类才能真正被复用，而衍生类也能够在基类的基础上增加新的行为。里氏代换原则是对“开-闭”原则的补充。实现“开-闭”原则的关键步骤就是抽象化。而基类与子类的继承关系就是抽象化的具体实现，所以里氏代换原则是对实现抽象化的具体步骤的规范。—— From Baidu 百科

**3、依赖倒转原则（Dependence Inversion Principle）**

这个是开闭原则的基础，具体内容：真对接口编程，依赖于抽象而不依赖于具体。

**4、接口隔离原则（Interface Segregation Principle）**

这个原则的意思是：使用多个隔离的接口，比使用单个接口要好。还是一个降低类之间的耦合度的意思，从这儿我们看出，其实设计模式就是一个软件的设计思想，从大型软件架构出发，为了升级和维护方便。所以上文中多次出现：降低依赖，降低耦合。

**5、迪米特法则（最少知道原则）（Demeter Principle）**

为什么叫最少知道原则，就是说：一个实体应当尽量少的与其他实体之间发生相互作用，使得系统功能模块相对独立。

**6、合成复用原则（Composite Reuse Principle）**

原则是尽量使用合成/聚合的方式，而不是使用继承。

## 工厂方法模式

## 抽象工厂模式

## 单例模式

## 建造者模式

## 原型模式

## 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

## 享元模式

## 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

## 中介者模式

## 解释器模式

# JVM

## JVM体系结构与工作方式

## JVM内存管理

Java的内存管理就是对象的分配和释放问题。在Java中，程序员需要通过关键字new为每个对象申请内存空间（基本类型除外），所有的对象都在堆（Heap）中分配空间。另外，对象的释放是由GC决定和执行的。在Java中，内存的分配是由程序完成的，而内存的释放是由GC完成的这种收支两条线的方法确实简化了程序员的工作，但同时也加重了JVM的工作，这样是Java程序运行速度慢的一个重要原因。

### 内存泄露

在Java中内存泄露就是存在一些被分配的对象，这些对象有下面两个特点：1）对象时可达的，即在有向图中，存在通路可以与其相连；2）对象是无用的，即程序以后不会再使用这些对象。如果满足这两个条件，这些对象就可以判定为Java中的内存泄露，这些对象不会被GC所回收，然而它们却占用内存。

## Class Loader工作机制

## JVM性能优化

### 性能优化工具

### 优化参数

## GC工作原理

在Java语言中，判断一块内存空间是否符合GC收集标准只有以下两个条件：

1. 给对象赋予了空值null，以后再也没有调用过；
2. 给对象赋予了新值，即重新分配了内存空间。

一块内存空间符合了GC的收集标准，并不意味着这块内存一定会被GC收集。并且显式调用System.gc()可以运行GC，但是不能保证立即回收指定的对象。GC发生的时间也是未知的，不要假定这个时间。

# Spring框架

## Spring框架设计理念和设计模式

### 核心组件

### DI

### Ioc

### AOP

## Spring MVC

### 总体设计

### Control设计

### Model设计

### View设计

## Spring Security

http://www.2cto.com/kf/201408/329607.html

# ORM框架

## iBatis

## Hibernate

## iBatis与Hibernate对比

# Java IO工作机制

## Java的IO类库的基本架构

### 基于字节的IO操作接口

### 基于字符的IO操作接口

### 字节与字符的转化接口

## 磁盘IO工作机制

### 访问文件的方式

### Java访问磁盘文件

### Java序列化技术

## 网络IO工作机制

### TCP状态变化

### 影响网络传输的因素

### Java Socket的工作机制

### 建立通信链路

### 数据传输

## NIO的工作方式

### BIO带来的挑战

### NIO的工作机制

### Buffer的工作机制

### NIO的数据访问方式

## IO调优

### 磁盘IO优化

### TCP网络参数优化

### 网络IO优化

# Servlet工作原理

## Servlet容器

## 创建Servlet实例

## Servlet体系结构

## Servlet如何工作

## Servlet中的Listener

## Filter如何工作

## Servlet中的url-patten

# Session和Cookie

## 理解Cookie

## 理解Session

## Cookie安全问题

## 分布式Session框架

## Cookie压缩

## 多终端Session统一

# 其它技术点

## 序列化和反序列化

### 基本概念

Java平台允许我们在内存中创建可复用的Java对象，但一般情况下，只有当JVM处于运行时，这些对象才可能存在，即，这些对象的生命周期不会比JVM的生命周期更长。但在现实应用中，就可能要求在JVM停止运行之后能够保存(持久化)指定的对象，并在将来重新读取被保存的对象。Java对象序列化就能够帮助我们实现该功能。

使用Java对象序列化，在保存对象时，会把其状态保存为一组字节，在未来，再将这些字节组装成对象。必须注意地是，对象序列化保存的是对象的"状态"，即它的成员变量。由此可知，对象序列化不会关注类中的静态变量。

除了在持久化对象时会用到对象序列化之外，当使用RMI(远程方法调用)，或在网络中传递对象时，都会用到对象序列化。Java序列化API为处理对象序列化提供了一个标准机制，该API简单易用。

如果仅仅只是让某个类实现Serializable接口，而没有其它任何处理的话，则就是使用默认序列化机制。使用默认机制，在序列化对象时，不仅会序列化当前对象本身，还会对该对象引用的其它对象也进行序列化，同样地，这些其它对象引用的另外对象也将被序列化，以此类推。所以，如果一个对象包含的成员变量是容器类对象，而这些容器所含有的元素也是容器类对象，那么这个序列化的过程就会较复杂，开销也较大。

当某个字段被声明为transient后，默认序列化机制就会忽略该字段。

### 序列化 ID 问题

**情境**：两个客户端 A 和 B 试图通过网络传递对象数据，A 端将对象 C 序列化为二进制数据再传给 B，B 反序列化得到 C。

**问题**：C 对象的全类路径假设为 com.inout.Test，在 A 和 B 端都有这么一个类文件，功能代码完全一致。也都实现了 Serializable 接口，但是反序列化时总是提示不成功。

**解决**：虚拟机是否允许反序列化，不仅取决于类路径和功能代码是否一致，一个非常重要的一点是两个类的序列化 ID 是否一致（就是 private static final long serialVersionUID = 1L）。

### 父类的序列化与 Transient 关键字

**情境：**一个子类实现了 Serializable 接口，它的父类都没有实现 Serializable 接口，序列化该子类对象，然后反序列化后输出父类定义的某变量的数值，该变量数值与序列化时的数值不同。

**解决：**要想将父类对象也序列化，就需要让父类也实现Serializable 接口。如果父类不实现的话的，就 需要有默认的无参的构造函数。在父类没有实现 Serializable 接口时，虚拟机是不会序列化父对象的，而一个 Java 对象的构造必须先有父对象，才有子对象，反序列化也不例外。所以反序列化时，为了构造父对象，只能调用父类的无参构造函数作为默认的父对象。因此当我们取父对象的变量值时，它的值是调用父类无参构造函数后的值。如果你考虑到这种序列化的情况，在父类无参构造函数中对变量进行初始化，否则的话，父类变量值都是默认声明的值，如 int 型的默认是 0，string 型的默认是 null。

Transient 关键字的作用是控制变量的序列化，在变量声明前加上该关键字，可以阻止该变量被序列化到文件中，在被反序列化后，transient 变量的值被设为初始值，如 int 型的是 0，对象型的是 null。

### 对敏感字段加密

**情境：**服务器端给客户端发送序列化对象数据，对象中有一些数据是敏感的，比如密码字符串等，希望对该密码字段在序列化时，进行加密，而客户端如果拥有解密的密钥，只有在客户端进行反序列化时，才可以对密码进行读取，这样可以一定程度保证序列化对象的数据安全。

**解决：**在序列化过程中，虚拟机会试图调用对象类里的 writeObject 和 readObject 方法，进行用户自定义的序列化和反序列化，如果没有这样的方法，则默认调用是 ObjectOutputStream 的 defaultWriteObject 方法以及 ObjectInputStream 的 defaultReadObject 方法。用户自定义的 writeObject 和 readObject 方法可以允许用户控制序列化的过程，比如可以在序列化的过程中动态改变序列化的数值。

### 序列化存储规则

Java 序列化机制为了节省磁盘空间，具有特定的存储规则，当写入文件的为同一对象时，并不会再将对象的内容进行存储，而只是再次存储一份引用。

## 浅拷贝和深拷贝

### clone方法

clone方法是在Object中定义的，而且是protected类型的，只有实现了这个接口，才可以在该类的实例上调用clone方法，否则会抛出CloneNotSupportedException。

Object中默认的实现是一个浅拷贝，也就是表面复制，如果需要实现深层次复制，必须对类中可变域生成新的实例。

### 什么是浅拷贝

浅拷贝是按位拷贝对象，它会创建一个新对象，这个对象有着原始对象属性值的一份精确拷贝。如果属性是基本类型，拷贝的就是基本类型的值；如果属性是内存地址（引用类型），拷贝的就是内存地址 ，因此如果其中一个对象改变了这个地址，就会影响到另一个对象。

### 什么是深拷贝

深拷贝会拷贝所有的属性，并拷贝属性指向的动态分配的内存。当对象和它所引用的对象一起拷贝时即发生深拷贝。深拷贝相比于浅拷贝速度较慢并且花销较大。

### 通过序列化实现深拷贝

也可以通过序列化来实现深拷贝。序列化是干什么的?它将整个对象图写入到一个持久化存储文件中并且当需要的时候把它读取回来, 这意味着当你需要把它读取回来时你需要整个对象图的一个拷贝。这就是当你深拷贝一个对象时真正需要的东西。请注意，当你通过序列化进行深拷贝时，必须确保对象图中所有类都是可序列化的。

## Date

## 抽象类和接口

## 参数传递

经典问题：Java到底是值传递还是引用传递？

Think in Java中的一个经典描述：如果Java中传值，那么传递的是值的副本；如果Java中传引用，那么传递的是引用的副本。

在Java中，变量分为两种：

1）对于基本数据类型 ，Java传递的是值；

2）对于一切对象型变量，Java都是传递的是引用的副本。其实传递引用副本的实质就是复制指向地址的指针，只不过Java不像C++那样有显著的\*和&符号。

需要注意的是，String类型也是对象型变量，所以它必然也是传引用副本。不要因为String类无需new，就蒙蔽的将其当做基本变量类型。只不过String是一个非可变类，使得其传值还是传引用显得没什么区别。

对于基本类型而言，传值就是把自己复制一份传递，即使自己的副本变了，自己也不变。而对于对象类型而言，它传递的是引用副本指向自己的地址，而不是自己实际值的副本。这是因为对象类型是存放在堆里面的，一方便，速度相对于基本类型比较慢，另一方面，对象类型本身一般比较大，如果采用重新复制值得方法，浪费内存并且速度又慢。

代码样例：

|  |
| --- |
| **public** **class** Value  {  **public** **int** i = 15;  }  **public** **class** Test  {  **public** **void** first()  {  **int** i = 5;  Value v = **new** Value();  v.i = 25;  second(v, i);  System.*out*.println(v.i + " " + i);  }  **public** **void** second(Value v, **int** i)  {  i = 0;  v.i = 20;  Value val = **new** Value();  v = val;  System.*out*.println(v.i + " " + i);  }  **public** **static** **void** main(String[] args)  {  Test t = **new** Test();  t.first();  }  } |

输出结果为：

15 0

20 5

分析：

按照上面的解释，对于second()方法中的两个参数，Value v是引用传递（传的是引用的副本），Int i是值传递（传的是值的副本）。在second()方法中，v.i改成了20，i改成了0，同时v指向了一个新Value对象的引用（该对象的默认i=15），所以在second()方法中输出的结果就是：15 0。接着分析first()方法，在first()中，i初始化为5，在执行second()方法的时候i被改成了0，但是只是其副本被改为了0，i本身并未改变，所以i值仍为5；Value对象v对象创建的时候v.i默认为15，后被改为25，然后作为引用传递给了second()方法，在second()中v.i改为了20，同时v指向了一个全新的Value对象（v.i=15），但是需要注意的是，在second()中v只是first()中v的一个引用副本，也就是说在second方法中只是其引用副本重新指向了一个新的Value对象，原始的对象v的引用未发生改变，只是v.i在second方法中被改为了20，所以first方法中输出20 5。

## 静态变量和私有变量

static表示“全局”或者“静态”的意思，用来修饰成员变量和成员方法，也可以形成静态static代码块，但是Java语言中没有全局变量的概念。

被static修饰的成员变量和成员方法独立于该类的任何对象。也就是说，它不依赖类特定的实例，被类的所有实例共享。

只要这个类被加载，Java虚拟机就能根据类名在运行时数据区的方法区内定找到他们。因此，static对象可以在它的任何对象创建之前访问，无需引用任何对象。

用public修饰的static成员变量和成员方法本质是全局变量和全局方法，当声明它类的对象市，不生成static变量的副本，而是类的所有实例共享同一个static变量。

static变量前可以有private修饰，表示这个变量可以在类的静态代码块中，或者类的其他静态成员方法中使用（当然也可以在非静态成员方法中使用--废话），但是不能在其他类中通过类名来直接引用，这一点很重要。实际上你需要搞明白，private是访问权限限定，static表示不要实例化就可以使用，这样就容易理解多了。static前面加上其它访问权限关键字的效果也以此类推。

static修饰的成员变量和成员方法习惯上称为静态变量和静态方法，可以直接通过类名来访问，访问语法为：

类名.静态方法名(参数列表...)

**类名.静态变量名**

用static修饰的代码块表示静态代码块，当Java虚拟机（JVM）加载类时，就会执行该代码块。

**1、static变量**

按照是否静态的对类成员变量进行分类可分两种：一种是被static修饰的变量，叫静态变量或类变量；另一种是没有被static修饰的变量，叫实例变量。

两者的区别是：

对于静态变量在内存中只有一个拷贝（节省内存），JVM只为静态分配一次内存，在加载类的过程中完成静态变量的内存分配，可用类名直接访问（方便），当然也可以通过对象来访问（但是这是不推荐的）。

对于实例变量，没创建一个实例，就会为实例变量分配一次内存，实例变量可以在内存中有多个拷贝，互不影响（灵活）。

所以一般在需要实现以下两个功能时使用静态变量：

* 在对象之间共享值时
* 方便访问变量时

**2、静态方法**

静态方法可以直接通过类名调用，任何的实例也都可以调用，

因此静态方法中不能用this和super关键字，不能直接访问所属类的实例变量和实例方法(就是不带static的成员变量和成员成员方法)，只能访问所属类的静态成员变量和成员方法。

因为实例成员与特定的对象关联！这个需要去理解，想明白其中的道理，不是记忆！！！

因为static方法独立于任何实例，因此static方法必须被实现，而不能是抽象的abstract。

例如为了方便方法的调用，Java API中的Math类中所有的方法都是静态的，而一般类内部的static方法也是方便其它类对该方法的调用。

静态方法是类内部的一类特殊方法，只有在需要时才将对应的方法声明成静态的，一个类内部的方法一般都是非静态的

**3、static代码块**

static代码块也叫静态代码块，是在类中独立于类成员的static语句块，可以有多个，位置可以随便放，它不在任何的方法体内，JVM加载类时会执行这些静态的代码块，如果static代码块有多个，JVM将按照它们在类中出现的先后顺序依次执行它们，每个代码块只会被执行一次。例如：

1. **public** **class** Test5 {
2. **private** **static** **int** a;
3. **private** **int** b;
5. **static**{
6. Test5.a=3;
7. System.out.println(a);
8. Test5 t=**new** Test5();
9. t.f();
10. t.b=1000;
11. System.out.println(t.b);
12. }
13. **static**{
14. Test5.a=4;
15. System.out.println(a);
16. }
17. **public** **static** **void** main(String[] args) {
18. // TODO 自动生成方法存根
19. }
20. **static**{
21. Test5.a=5;
22. System.out.println(a);
23. }
24. **public** **void** f(){
25. System.out.println("hhahhahah");
26. }
27. }

运行结果：

3

hhahhahah

1000

4

5

利用静态代码块可以对一些static变量进行赋值，最后再看一眼这些例子，都一个static的main方法，这样JVM在运行main方法的时候可以直接调用而不用创建实例。

**4、static和final一块用表示什么**

static final用来修饰成员变量和成员方法，可简单理解为“全局常量”！

对于变量，表示一旦给值就不可修改，并且通过类名可以访问。

对于方法，表示不可覆盖，并且可以通过类名直接访问。

有时你希望定义一个类成员，使它的使用完全独立于该类的任何对象。通常情况下，类成员必须通过它的类的对象访问，但是可以创建这样一个成员，它能够被它自己使用，而不必引用特定的实例。在成员的声明前面加上关键字static(静态的)就能创建这样的成员。如果一个成员被声明为static，它就能够在它的类的任何对象创建之前被访问，而不必引用任何对象。你可以将方法和变量都声明为static。static 成员的最常见的例子是main( ) 。因为在程序开始执行时必须调用main() ，所以它被声明为static。

声明为static的变量实质上就是全局变量。当声明一个对象时，并不产生static变量的拷贝，而是该类所有的实例变量共用同一个static变量。声明为static的方法有以下几条限制：

* 它们仅能调用其他的static 方法。
* 它们只能访问static数据。
* 它们不能以任何方式引用this 或super（关键字super 与继承有关，在下一章中描述）。

如果你需要通过计算来初始化你的static变量，你可以声明一个static块，Static 块仅在该类被加载时执行一次。下面的例子显示的类有一个static方法，一些static变量，以及一个static 初始化块：

1. // Demonstrate static variables，methods，and blocks.
3. **class** UseStatic {
4. **static** **int** a = 3;
5. **static** **int** b;
7. **static** **void** meth(**int** x) {
8. System.out.println("x = " + x);
9. System.out.println("a = " + a);
10. System.out.println("b = " + b);
11. }
13. **static** {
14. System.out.println("Static block initialized.");
15. b = a \* 4;
16. }
18. **public** **static** **void** main(String args[]) {
19. meth(42);
20. }
21. }

一旦UseStatic 类被装载，所有的static语句被运行。首先，a被设置为3，接着static 块执行(打印一条消息)，最后，b被初始化为a\*4 或12。然后调用main()，main() 调用meth() ，把值42传递给x。3个println ( ) 语句引用两个static变量a和b，以及局部变量x 。

注意：在一个static 方法中引用任何实例变量都是非法的。

下面是该程序的输出：

Static block initialized.

x = 42

a = 3

b = 12

在定义它们的类的外面，static 方法和变量能独立于任何对象而被使用。这样，你只要在类的名字后面加点号运算符即可。例如，如果你希望从类外面调用一个static方法，你可以使用下面通用的格式：

classname.method( )

这里，classname 是类的名字，在该类中定义static方法。可以看到，这种格式与通过对象引用变量调用非static方法的格式类似。一个static变量可以以同样的格式来访问——类名加点号运算符。这就是Java 如何实现全局功能和全局变量的一个控制版本。

下面是一个例子。在main() 中，static方法callme() 和static 变量b在它们的类之外被访问。

1. **class** StaticDemo {
2. **static** **int** a = 42;
3. **static** **int** b = 99;
4. **static** **void** callme() {
6. System.out.println("a = " + a);
7. }
8. }
10. **class** StaticByName {
12. **public** **static** **void** main(String args[]) {
13. StaticDemo.callme();
14. System.out.println("b = " + StaticDemo.b);
15. }
16. }

下面是该程序的输出：

a = 42

b = 99

static成员是不能被其所在class创建的实例访问的。

如果不加static修饰的成员是对象成员，也就是归每个对象所有的。

加static修饰的成员是类成员，就是可以由一个类直接调用，为所有对象共有的。

## 正则表达式

## 重载和重写

重载和重写都是针对方法的概念，在弄清楚这两个概念之前，我们先来了解一下什么叫方法的型构（英文名是signature，有的译作“签名”，虽然它被使用的较为广泛，但是这个翻译不准确的）。型构就是指方法的组成结构，具体包括方法的名称和参数，涵盖参数的数量、类型以及出现的顺序，但是不包括方法的返回值类型，访问权限修饰符，以及abstract、static、final等修饰符。比如下面两个就是具有相同型构的方法：

Java代码：

public void method(int i, String s) {

    // do something

}

public String method(int i, String s) {

    // do something

}

而这两个就是具有不同型构的方法：

Java代码

public void method(int i, String s) {

    // do something

}

public void method(String s, int i) {

    // do something

}

了解完型构的概念后我们再来看看重载和重写，请看它们的定义：

重写，英文名是overriding，是指在继承情况下，子类中定义了与其基类中方法具有相同型构的新方法，就叫做子类把基类的方法重写了。这是实现多态必须的步骤。

重载，英文名是overloading，是指在同一个类中定义了一个以上具有相同名称，但是型构不同的方法。在同一个类中，是不允许定义多于一个的具有相同型构的方法的。

## 

## 

## 

## 

## 

## 

## 

# 相关开发工具和技能

## 开源框架和技术

一般情况下，如果阅读过主流的一些开源框架（例如：Spring、iBatis、Hibernate等待），甚至自己实际参与过一些开源项目的开发或提交过bug/patch等，都会为自己加分。

## 版本管理工具

### Git

Git是一个分布式的版本管理工具。

一般可以通过下面两种方式使用Git来管理代码。

* 自行搭建git server；
* 采用git公共托管服务：
  + 国外的Github：<https://github.com>
  + 国内的CSDN：http://code.csdn.net/

### SVN

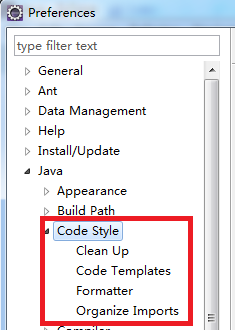
SVN是一种集中式的版本管理工具。

## Java编程规范

参考：《Google Java编程规范》

## eclipse中Java代码格式化

在eclipse中可以设置Java代码的Code Style，一般通过下图中的配置来完成，并可以通过xml文件导入导出，可以解决大部分的check style问题，使得编程过程中无需关注代码格式问题，项目组内成员采用统一的Java风格也有利于代码的维护。



## 代码审查

## 项目管理工具

Jira等。

## Wiki系统

MediaWiki、Confluence系统等

## Bug追踪工具

BugZilla、Confluence、github等

## UML建模工具

Visio、Rational Rose、PowerDesign等。

## 文档编写能力

doc、ppt等。

## 技术博客

工作之余将学习心得等总结出来，放到个人技术博客上去。

## github

可以在github上编写自己的开源项目，也可以参与到其他的开源项目中去。