**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**

[**Java关键技术汇集** 1](#_Toc400662509)

[1 写在前面的话 8](#_Toc400662510)

[2 Java三大特性 8](#_Toc400662511)

[2.1 封装 8](#_Toc400662512)

[2.2 继承 9](#_Toc400662513)

[2.3 多态 10](#_Toc400662514)

[3 Java数据类型 10](#_Toc400662515)

[3.1 基本数据类型 10](#_Toc400662516)

[3.1.1 boolean 10](#_Toc400662517)

[3.1.2 char 11](#_Toc400662518)

[3.1.3 byte 11](#_Toc400662519)

[3.1.4 short 11](#_Toc400662520)

[3.1.5 int 11](#_Toc400662521)

[3.1.6 long 12](#_Toc400662522)

[3.1.7 float 12](#_Toc400662523)

[3.1.8 double 12](#_Toc400662524)

[3.2 数据类型转换 13](#_Toc400662525)

[3.2.1 自动类型转换 13](#_Toc400662526)

[3.2.2 强制类型转换 13](#_Toc400662527)

[3.2.3 包装类过渡类型转换 13](#_Toc400662528)

[3.2.4 字符串型与其他数据类型的转换 13](#_Toc400662529)

[3.2.5 将字符型直接作为数值转换为其他数据类型 13](#_Toc400662530)

[3.3 对象型数据类型 14](#_Toc400662531)

[3.3.1 String 14](#_Toc400662532)

[3.3.2 ArrayList 16](#_Toc400662533)

[3.3.3 LinkedList 22](#_Toc400662534)

[3.3.4 HashMap 22](#_Toc400662535)

[3.3.5 HashTable 26](#_Toc400662536)

[3.3.6 HashSet 29](#_Toc400662537)

[3.3.7 Stack 29](#_Toc400662538)

[3.3.8 Queue 29](#_Toc400662539)

[3.3.9 Vector 29](#_Toc400662540)

[3.3.10 Iterator 29](#_Toc400662541)

[3.3.11 TreeSet 29](#_Toc400662542)

[3.3.12 Set 29](#_Toc400662543)

[4 多线程编程 29](#_Toc400662544)

[4.1 Java线程概念和原理 30](#_Toc400662545)

[4.2 Java线程创建和启动 30](#_Toc400662546)

[4.3 Java线程状态转换 30](#_Toc400662547)

[4.4 Java线程调度 30](#_Toc400662548)

[4.4.1 休眠 30](#_Toc400662549)

[4.4.2 优先级 30](#_Toc400662550)

[4.4.3 让步 30](#_Toc400662551)

[4.4.4 合并 30](#_Toc400662552)

[4.4.5 守护线程 31](#_Toc400662553)

[4.5 Java线程同步与锁 31](#_Toc400662554)

[4.5.1 同步 31](#_Toc400662555)

[4.5.2 监视器或内部锁 32](#_Toc400662556)

[4.5.3 原子变量 32](#_Toc400662557)

[4.5.4 volatile变量 33](#_Toc400662558)

[4.5.5 静态和volatile变量之间的差别 33](#_Toc400662559)

[4.5.6 volatile变量和同步之间的差别 34](#_Toc400662560)

[4.5.7 锁对象 34](#_Toc400662561)

[4.5.8 Java内存回收 34](#_Toc400662562)

[4.5.9 如何写一个死锁程序 34](#_Toc400662563)

[4.5.10 Java中的引用类型 36](#_Toc400662564)

[4.6 Java线程交互 36](#_Toc400662565)

[4.7 Java线程池 36](#_Toc400662566)

[5 设计模式 37](#_Toc400662567)

[5.1 设计模式的六大原则 37](#_Toc400662568)

[5.2 工厂方法模式 38](#_Toc400662569)

[5.3 抽象工厂模式 38](#_Toc400662570)

[5.4 单例模式 38](#_Toc400662571)

[5.5 建造者模式 38](#_Toc400662572)

[5.6 原型模式 38](#_Toc400662573)

[5.7 适配器模式 38](#_Toc400662574)

[5.8 装饰器模式 38](#_Toc400662575)

[5.9 代理模式 38](#_Toc400662576)

[5.10 外观模式 38](#_Toc400662577)

[5.11 桥接模式 38](#_Toc400662578)

[5.12 组合模式 39](#_Toc400662579)

[5.13 享元模式 39](#_Toc400662580)

[5.14 策略模式 39](#_Toc400662581)

[5.15 模板方法模式 39](#_Toc400662582)

[5.16 观察者模式 39](#_Toc400662583)

[5.17 迭代子模式 39](#_Toc400662584)

[5.18 责任链模式 39](#_Toc400662585)

[5.19 命令模式 39](#_Toc400662586)

[5.20 备忘录模式 39](#_Toc400662587)

[5.21 状态模式 39](#_Toc400662588)

[5.22 访问者模式 39](#_Toc400662589)

[5.23 中介者模式 40](#_Toc400662590)

[5.24 解释器模式 40](#_Toc400662591)

[6 JVM 40](#_Toc400662592)

[6.1 JVM体系结构与工作方式 40](#_Toc400662593)

[6.2 JVM内存管理 40](#_Toc400662594)

[6.2.1 内存泄露 40](#_Toc400662595)

[6.3 Class Loader工作机制 41](#_Toc400662596)

[6.4 JVM性能优化 41](#_Toc400662597)

[6.4.1 性能优化工具 41](#_Toc400662598)

[6.4.2 优化参数 41](#_Toc400662599)

[6.5 GC工作原理 41](#_Toc400662600)

[7 Spring框架 41](#_Toc400662601)

[7.1 Spring框架设计理念和设计模式 41](#_Toc400662602)

[7.1.1 核心组件 41](#_Toc400662603)

[7.1.2 DI 42](#_Toc400662604)

[7.1.3 Ioc 42](#_Toc400662605)

[7.1.4 AOP 42](#_Toc400662606)

[7.2 Spring MVC 42](#_Toc400662607)

[7.2.1 总体设计 42](#_Toc400662608)

[7.2.2 Control设计 42](#_Toc400662609)

[7.2.3 Model设计 42](#_Toc400662610)

[7.2.4 View设计 42](#_Toc400662611)

[7.3 Spring Security 42](#_Toc400662612)

[8 ORM框架 43](#_Toc400662613)

[8.1 iBatis 43](#_Toc400662614)

[8.2 Hibernate 43](#_Toc400662615)

[8.3 iBatis与Hibernate对比 43](#_Toc400662616)

[9 Java IO工作机制 43](#_Toc400662617)

[9.1 Java的IO类库的基本架构 43](#_Toc400662618)

[9.1.1 基于字节的IO操作接口 43](#_Toc400662619)

[9.1.2 基于字符的IO操作接口 43](#_Toc400662620)

[9.1.3 字节与字符的转化接口 43](#_Toc400662621)

[9.2 磁盘IO工作机制 44](#_Toc400662622)

[9.2.1 访问文件的方式 44](#_Toc400662623)

[9.2.2 Java访问磁盘文件 44](#_Toc400662624)

[9.2.3 Java序列化技术 44](#_Toc400662625)

[9.3 网络IO工作机制 44](#_Toc400662626)

[9.3.1 TCP状态变化 44](#_Toc400662627)

[9.3.2 影响网络传输的因素 44](#_Toc400662628)

[9.3.3 Java Socket的工作机制 44](#_Toc400662629)

[9.3.4 建立通信链路 44](#_Toc400662630)

[9.3.5 数据传输 44](#_Toc400662631)

[9.4 NIO的工作方式 45](#_Toc400662632)

[9.4.1 BIO带来的挑战 45](#_Toc400662633)

[9.4.2 NIO的工作机制 45](#_Toc400662634)

[9.4.3 Buffer的工作机制 45](#_Toc400662635)

[9.4.4 NIO的数据访问方式 45](#_Toc400662636)

[9.5 IO调优 45](#_Toc400662637)

[9.5.1 磁盘IO优化 45](#_Toc400662638)

[9.5.2 TCP网络参数优化 45](#_Toc400662639)

[9.5.3 网络IO优化 45](#_Toc400662640)

[10 Servlet工作原理 46](#_Toc400662641)

[10.1 Servlet容器 46](#_Toc400662642)

[10.2 创建Servlet实例 46](#_Toc400662643)

[10.3 Servlet体系结构 46](#_Toc400662644)

[10.4 Servlet如何工作 46](#_Toc400662645)

[10.5 Servlet中的Listener 46](#_Toc400662646)

[10.6 Filter如何工作 46](#_Toc400662647)

[10.7 Servlet中的url-patten 46](#_Toc400662648)

[11 Session和Cookie 47](#_Toc400662649)

[11.1 理解Cookie 47](#_Toc400662650)

[11.2 理解Session 47](#_Toc400662651)

[11.3 Cookie安全问题 47](#_Toc400662652)

[11.4 分布式Session框架 47](#_Toc400662653)

[11.5 Cookie压缩 47](#_Toc400662654)

[11.6 多终端Session统一 47](#_Toc400662655)

[12 其它技术点 47](#_Toc400662656)

[12.1 序列化和反序列化 47](#_Toc400662657)

[12.1.1 基本概念 47](#_Toc400662658)

[12.1.2 序列化 ID 问题 48](#_Toc400662659)

[12.1.3 父类的序列化与 Transient 关键字 48](#_Toc400662660)

[12.1.4 对敏感字段加密 49](#_Toc400662661)

[12.1.5 序列化存储规则 49](#_Toc400662662)

[12.2 浅拷贝和深拷贝 49](#_Toc400662663)

[12.2.1 clone方法 49](#_Toc400662664)

[12.2.2 什么是浅拷贝 50](#_Toc400662665)

[12.2.3 什么是深拷贝 50](#_Toc400662666)

[12.2.4 通过序列化实现深拷贝 50](#_Toc400662667)

[12.3 抽象类和接口 50](#_Toc400662668)

[12.4 参数传递 52](#_Toc400662669)

[12.5 静态变量和私有变量 54](#_Toc400662670)

[12.6 正则表达式 60](#_Toc400662671)

[12.7 重载和重写 60](#_Toc400662672)

[12.8 HashMap和HashTable 61](#_Toc400662673)

[12.9 构造函数和析构函数 61](#_Toc400662674)

[12.10 super关键字 61](#_Toc400662675)

[12.11 synchronized和java.util.concurrent.locks.Lock的异同 62](#_Toc400662676)

[13 相关开发工具和技能 62](#_Toc400662677)

[13.1 开源框架和技术 62](#_Toc400662678)

[13.2 版本管理工具 63](#_Toc400662679)

[13.2.1 Git 63](#_Toc400662680)

[13.2.2 SVN 63](#_Toc400662681)

[13.3 Java编程规范 63](#_Toc400662682)

[13.4 eclipse中Java代码格式化 63](#_Toc400662683)

[13.5 代码审查 64](#_Toc400662684)

[13.6 项目管理工具 64](#_Toc400662685)

[13.7 Wiki系统 64](#_Toc400662686)

[13.8 Bug追踪工具 64](#_Toc400662687)

[13.9 UML建模工具 64](#_Toc400662688)

[13.10 文档编写能力 65](#_Toc400662689)

[13.11 技术博客 65](#_Toc400662690)

[13.12 github 65](#_Toc400662691)

# 写在前面的话

一直以来，Java都是一种很应用普遍的开发语言，功能强大，并且入门相对比较简单。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

封装可以隐藏实现细节，使得代码模块化；继承可以扩展已存在的代码模块（类）。它们的目的都是为了代码重用。而多态则是为了实现接口重用。

## 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

### String

**问题1：**

String s1 = "abc";

String s2 = "abc";

s1==s2是否为true？"abc"是放在常量池的，所以s1和s2虽然都等于"abc"，但内存中只有一份副本，所以s1==s2。

**问题2：**

String s1 = new String("abc");

String s2 = new String("abc");

s1==s2是否为true？为false，因为new方法决定了两个不同的String "abc"被创建放在了内存的heap区。

原则：

1. “==”判断左右两个变量是否指向同一个内存地址；
2. “equals()”判断左右两个变量的值是否相同。

**问题3：**

String s = "a" + "b" + "c" + "d" + "e";

这条语句创建了几个对象？

一共创建1个对象，因为"a"、"b"、"c"、"d"、"e"都是常量，对于常量，编译时就直接存储它们的子面值，而不是它们的引用，在编译时就直接将它们连接的结果提取出来变成了"abcde"，该语句在class文件中就相当于String s = "abcde";

**扩展知识1：常量池的概念**

在编译好的calss文件中，有一个区域称为Constant Pool，它是一个由数组组成的表，类型为cp\_info constant\_pool[]，用来存储程序中使用的各种常量，包括Class、String、Integer等各种基本的Java数据类型。

在程序执行的时候，Constant Pool会存储在Method Area中，而不是堆中。另外，对于“”内容为空的字符串常量，会创建一个长度为0、内容为空的字符串到Constant Pool中，而且Constant Pool在运行期是可以动态扩展的。

**扩展知识2：关于String类的说明**

String使用private final char value[]来实现字符串的存储，也就是说，String对象创建之后，就不能再修改此对象中存储的字符串内容，就是因为如此，才说String类型是不可变的。

String类有一个特殊的创建方法，就是使用””双引号来创建。例如：new String(“abc”)实际上创建了2个String对象，一个是”abc”通过双引号创建的，另一个是通过new创建的，只不过它们创建的时期不同，一个是在编译器，另一个是在运行期。

Java对String类型重载了加号操作符，可以直接使用“+”对两个字符串进行连接。

运行期调用String类的intern()方法可以向String Pool中动态添加对象。

String的创建方法一般有如下几种：

* 直接使用””引号创建；
* 使用new String()创建；
* 使用new String(“some String”)创建，以及其它一些重载构造函数创建；
* 使用重载的字符串连接操作符“+”创建。

**问题4：**

String str = “ABCDEFGH”;

String str1 = str.substring(3, 5);

那么，str1应为什么？

许多人以为是DEF，实际应为DE，这是因为substring()是这样定义的：public String substring(int beginIndex, int endIndex)，其中beginIndex the beginning index, inclusive. endIndex the ending index, exclusive。也就是说，substring是包前不包后的，其长度为endIndex- beginIndex。

**扩展知识3：String和StringBuffer的区别**

StringBuffer关键点：

* 简单的认为append()效率好于“+”是错误的；
* 不要使用new创建String；
* 注意intern()的使用；
* 在编译器能够确定字符串值的情况下，使用“+”效率最高；
* 避免使用“+=”来构造字符串；
* 在声明StringBuffer对象的时候，指定合适的capacity，不要使用默认值；
* 注意以下两者的区别，后者开辟了两个内存段：

String s= “a” + “b”;

String s = “a”;

s += “b”;

### ArrayList

ArrayList是实现List接口的动态数组，所谓动态就是它的大小是可变的。实现了所有可选列表操作，并允许包括 null 在内的所有元素。除了实现 List 接口外，此类还提供一些方法来操作内部用来存储列表的数组的大小。

每个ArrayList实例都有一个容量，该容量是指用来存储列表元素的数组的大小。默认初始容量为10。随着ArrayList中元素的增加，它的容量也会不断的自动增长。在每次添加新的元素时，ArrayList都会检查是否需要进行扩容操作，扩容操作带来数据向新数组的重新拷贝，所以如果我们知道具体业务数据量，在构造ArrayList时可以给ArrayList指定一个初始容量，这样就会减少扩容时数据的拷贝问题。当然在添加大量元素前，应用程序也可以使用ensureCapacity操作来增加ArrayList实例的容量，这可以减少递增式再分配的数量。

**注意，ArrayList实现不是同步的**。如果多个线程同时访问一个ArrayList实例，而其中至少一个线程从结构上修改了列表，那么它必须保持外部同步。所以为了保证同步，最好的办法是在创建时完成，以防止意外对列表进行不同步的访问：List list = Collections.synchronizedList(new ArrayList(...));

下面从源码分析ArrayList的数据结构。

ArrayList默认的容量大小为10：

|  |
| --- |
| **private** **static** **final** **int** *DEFAULT\_CAPACITY* = 10; |

ArrayList底层使用数组：

|  |
| --- |
| /\*\*  \* The array buffer into which the elements of the ArrayList are stored.  \* The capacity of the ArrayList is the length of this array buffer. Any  \* empty ArrayList with elementData == EMPTY\_ELEMENTDATA will be expanded to  \* DEFAULT\_CAPACITY when the first element is added.  \*/  **private** **transient** Object[] elementData; |

其中，transient为java关键字，为变量修饰符，如果用transient声明一个实例变量，当对象存储时，它的值不需要维持。Java的serialization提供了一种持久化对象实例的机制。当持久化对象时，可能有一个特殊的对象数据成员，我们不想用serialization机制来保存它。为了在一个特定对象的一个域上关闭serialization，可以在这个域前加上关键字transient。当一个对象被序列化的时候，transient型变量的值不包括在序列化的表示中，然而非transient型的变量是被包括进去的。

ArrayList的构造函数：

|  |
| --- |
| /\*\*  \* 默认构造函数，提供初始容量为10的空列表。  \*/  **public** ArrayList(**int** initialCapacity) {  **super**();  **if** (initialCapacity < 0)  **throw** **new** IllegalArgumentException("Illegal Capacity: "+ initialCapacity);  **this**.elementData = **new** Object[initialCapacity];  }  /\*\*  \* 构造一个具有指定初始容量的空列表。  \*/  **public** ArrayList() {  **super**();  **this**.elementData = *EMPTY\_ELEMENTDATA*;  }  /\*\*  \* 构造一个包含指定 collection 的元素的列表，这些元素是按照该 collection 的迭代器返回它们的顺序排列的。  \*/  **public** ArrayList(Collection<? **extends** E> c) {  elementData = c.toArray();  size = elementData.length;  // c.toArray might (incorrectly) not return Object[] (see 6260652)  **if** (elementData.getClass() != Object[].**class**)  elementData = Arrays.*copyOf*(elementData, size, Object[].**class**);  } |

增加元素：

ArrayList提供了add(E e)、add(int index, E element)、addAll(Collection<? extends E> c)、addAll(int index, Collection<? extends E> c)、set(int index, E element)这个五个方法来实现ArrayList增加。

|  |
| --- |
| **public** **boolean** add(E e) {  ensureCapacityInternal(size + 1); // Increments modCount!!  elementData[size++] = e;  **return** **true**;  }  **public** **void** add(**int** index, E element) {  rangeCheckForAdd(index);  ensureCapacityInternal(size + 1); // Increments modCount!!  System.*arraycopy*(elementData, index, elementData, index + 1,  size - index);  elementData[index] = element;  size++;  }  **public** **boolean** addAll(**int** index, Collection<? **extends** E> c) {  rangeCheckForAdd(index);  Object[] a = c.toArray();  **int** numNew = a.length;  ensureCapacityInternal(size + numNew); // Increments modCount  **int** numMoved = size - index;  **if** (numMoved > 0)  System.*arraycopy*(elementData, index, elementData, index + numNew,  numMoved);  System.*arraycopy*(a, 0, elementData, index, numNew);  size += numNew;  **return** numNew != 0;  }  **public** **boolean** addAll(Collection<? **extends** E> c) {  Object[] a = c.toArray();  **int** numNew = a.length;  ensureCapacityInternal(size + numNew); // Increments modCount  System.*arraycopy*(a, 0, elementData, size, numNew);  size += numNew;  **return** numNew != 0;  } |

删除元素：

ArrayList提供了remove(int index)、remove(Object o)、removeRange(int fromIndex, int toIndex)、removeAll()四个方法进行元素的删除。

|  |
| --- |
| **public** E remove(**int** index) {  rangeCheck(index);  modCount++;  E oldValue = elementData(index);  **int** numMoved = size - index - 1;  **if** (numMoved > 0)  System.*arraycopy*(elementData, index+1, elementData, index,  numMoved);  elementData[--size] = **null**; // clear to let GC do its work  **return** oldValue;  }  **public** **boolean** remove(Object o) {  **if** (o == **null**) {  **for** (**int** index = 0; index < size; index++)  **if** (elementData[index] == **null**) {  fastRemove(index);  **return** **true**;  }  } **else** {  **for** (**int** index = 0; index < size; index++)  **if** (o.equals(elementData[index])) {  fastRemove(index);  **return** **true**;  }  }  **return** **false**;  }  **public** **boolean** removeAll(Collection<?> c) {  **return** batchRemove(c, **false**);  }  **protected** **void** removeRange(**int** fromIndex, **int** toIndex) {  modCount++;  **int** numMoved = size - toIndex;  System.*arraycopy*(elementData, toIndex, elementData, fromIndex,  numMoved);  // clear to let GC do its work  **int** newSize = size - (toIndex-fromIndex);  **for** (**int** i = newSize; i < size; i++) {  elementData[i] = **null**;  }  size = newSize;  } |

扩容：

在上面的新增方法的源码中我们发现每个方法中都存在这个方法：ensureCapacity()，该方法就是ArrayList的扩容方法。在前面就提过ArrayList每次新增元素时都会需要进行容量检测判断，若新增元素后元素的个数会超过ArrayList的容量，就会进行扩容操作来满足新增元素的需求。所以当我们清楚知道业务数据量或者需要插入大量元素前，我可以使用ensureCapacity来手动增加ArrayList实例的容量，以减少递增式再分配的数量。

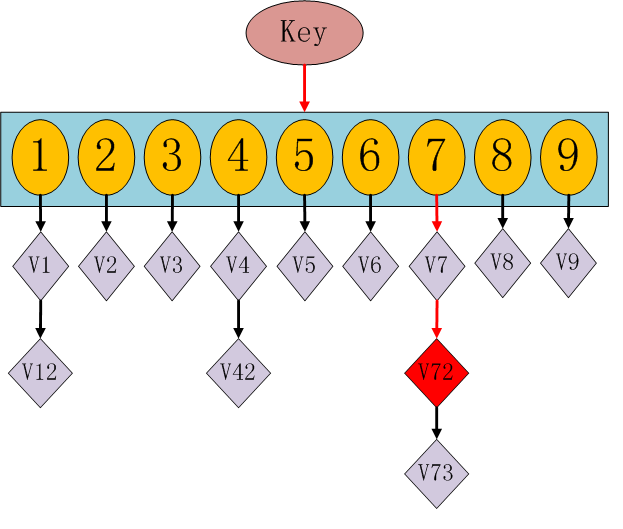
|  |
| --- |
| **public** **void** ensureCapacity(**int** minCapacity) {  **int** minExpand = (elementData != *EMPTY\_ELEMENTDATA*)  // any size if real element table  ? 0  // larger than default for empty table. It's already supposed to be  // at default size.  : *DEFAULT\_CAPACITY*;  **if** (minCapacity > minExpand) {  ensureExplicitCapacity(minCapacity);  }  }  **private** **void** ensureCapacityInternal(**int** minCapacity) {  **if** (elementData == *EMPTY\_ELEMENTDATA*) {  minCapacity = Math.*max*(*DEFAULT\_CAPACITY*, minCapacity);  }  ensureExplicitCapacity(minCapacity);  }  **private** **void** ensureExplicitCapacity(**int** minCapacity) {  modCount++;  // overflow-conscious code  **if** (minCapacity - elementData.length > 0)  grow(minCapacity);  }  **private** **void** grow(**int** minCapacity) {  // overflow-conscious code  **int** oldCapacity = elementData.length;  **int** newCapacity = oldCapacity + (oldCapacity >> 1);  **if** (newCapacity - minCapacity < 0)  newCapacity = minCapacity;  **if** (newCapacity - *MAX\_ARRAY\_SIZE* > 0)  newCapacity = *hugeCapacity*(minCapacity);  // minCapacity is usually close to size, so this is a win:  elementData = Arrays.*copyOf*(elementData, newCapacity);  } |

从“int newCapacity = oldCapacity + (oldCapacity >> 1);”可以看出，每次新的容量大小为原来容量大小的1.5倍。在这里有一个疑问，为什么每次扩容处理会是1.5倍，而不是2.5、3、4倍呢？因为一次性扩容太大(例如2.5倍)可能会浪费更多的内存(1.5倍最多浪费33%，而2.5被最多会浪费60%，3.5倍则会浪费71%……)。但是一次性扩容太小，需要多次对数组重新分配内存，对性能消耗比较严重。所以1.5倍刚刚好，既能满足性能需求，也不会造成很大的内存消耗。

### LinkedList

### HashMap

HashMap是基于数组+链表的一种数据结构，它的内部结构如下图所示：



一个新的<key, value>键值对加入到HashMap的过程为：

* 首先，判断key是否为null，如果为null，则直接调用putForNullKey方法；
* 如果key不为null，则计算key的hash值；
* 然后根据hash值搜索在table数组中的索引位置，如果table数组在该位置处有元素，则通过比较是否存在相同的key，若存在则覆盖原来key的value，否则将该元素保存在链头（最先保存的元素放在链尾）。若table在该处没有元素，则直接保存即可。

在HashMap中查找key的过程为：

* 首先，判断key是否为null，若为null，调用getForNullKey方法返回相对应的value；
* 如果key不为null，则计算key的hash值；
* 然后根据hash值搜索在table数组中的索引位置，如果table数组在该位置处有元素，则通过比较是否存在相同的key，存在则直接返回对应的value，不存在则返回null。若table在该处没有元素，则直接返回null。

通过对HashMap的数据结构和存取过程进行分析，可以知道HashMap之所以查找速度很快的关键原因就在于对key进行了一次hash计算，然后对每个不同的hash值建立了一个链表，因此查询的时候只需要在某一个链表分支上顺序查找即可，将一个海量数据查找问题转化为一个小样本库查找，查找次数大幅减少，效率也就极大提高了。还有一个关键问题，如何保证HashMap中每个table数组中的数据是均匀分布的？不能太紧也不能太松，太紧会导致查询速度慢，太松则浪费空间，HashMap是通过调用indexFor方法来进行优化的。

HashMap有两个参数影响其性能：初始容量和加载因子。默认初始容量是16，加载因子是0.75。容量是哈希表中桶(Entry数组)的数量，初始容量只是哈希表在创建时的容量。加载因子是哈希表在其容量自动增加之前可以达到多满的一种尺度。当哈希表中的条目数超出了加载因子与当前容量的乘积时，通过调用 rehash 方法将容量翻倍。

HashMap定义：

|  |
| --- |
| **public** **class** HashMap<K,V>  **extends** AbstractMap<K,V>  **implements** Map<K,V>, Cloneable, Serializable |

参见源码中下面的参数：

|  |
| --- |
| **static** **final** **int** *DEFAULT\_INITIAL\_CAPACITY* = 1 << 4; // aka 16  **static** **final** **int** *MAXIMUM\_CAPACITY* = 1 << 30;  **static** **final** **float** *DEFAULT\_LOAD\_FACTOR* = 0.75f; |

HashMap构造器源码如下：

|  |
| --- |
| **public** HashMap(**int** initialCapacity, **float** loadFactor) {  **if** (initialCapacity < 0)  **throw** **new** IllegalArgumentException("Illegal initial capacity: " + initialCapacity);  **if** (initialCapacity > *MAXIMUM\_CAPACITY*)  initialCapacity = *MAXIMUM\_CAPACITY*;  **if** (loadFactor <= 0 || Float.*isNaN*(loadFactor))  **throw** **new** IllegalArgumentException("Illegal load factor: " + loadFactor);  **this**.loadFactor = loadFactor;  threshold = initialCapacity;  init();  } |

HashMap的put方法源码：

|  |
| --- |
| **public** V put(K key, V value) {  **if** (table == *EMPTY\_TABLE*) {  inflateTable(threshold);  }  **if** (key == **null**)  **return** putForNullKey(value);  **int** hash = hash(key);  **int** i = *indexFor*(hash, table.length);  **for** (Entry<K,V> e = table[i]; e != **null**; e = e.next) {  Object k;  **if** (e.hash == hash && ((k = e.key) == key || key.equals(k))) {  V oldValue = e.value;  e.value = value;  e.recordAccess(**this**);  **return** oldValue;  }  }  modCount++;  addEntry(hash, key, value, i);  **return** **null**;  }  **private** V putForNullKey(V value) {  **for** (Entry<K,V> e = table[0]; e != **null**; e = e.next) {  **if** (e.key == **null**) {  V oldValue = e.value;  e.value = value;  e.recordAccess(**this**);  **return** oldValue;  }  }  modCount++;  addEntry(0, **null**, value, 0);  **return** **null**;  } |

HashMap的get方法源码：

|  |
| --- |
| **public** V get(Object key) {  **if** (key == **null**)  **return** getForNullKey();  Entry<K,V> entry = getEntry(key);  **return** **null** == entry ? **null** : entry.getValue();  }  **private** V getForNullKey() {  **if** (size == 0) {  **return** **null**;  }  **for** (Entry<K,V> e = table[0]; e != **null**; e = e.next) {  **if** (e.key == **null**)  **return** e.value;  }  **return** **null**;  } |

HashMap扩容的源码：

|  |
| --- |
| **void** resize(**int** newCapacity) {  Entry[] oldTable = table;  **int** oldCapacity = oldTable.length;  **if** (oldCapacity == *MAXIMUM\_CAPACITY*) {  threshold = Integer.*MAX\_VALUE*;  **return**;  }  Entry[] newTable = **new** Entry[newCapacity];  transfer(newTable, initHashSeedAsNeeded(newCapacity));  table = newTable;  threshold = (**int**)Math.*min*(newCapacity \* loadFactor, *MAXIMUM\_CAPACITY* + 1);  } |

### HashTable

HashTable与HashMap设计思想基本类似，同样采用hashcode(key)+数组+链表的形式来实现数据的快速存取，但是与HashMap稍有不同。

HashTable定义：

|  |
| --- |
| **public** **class** Hashtable<K,V>  **extends** Dictionary<K,V>  **implements** Map<K,V>, Cloneable, java.io.Serializable |

HashTable构造器源码如下：

|  |
| --- |
| **public** Hashtable() {  **this**(11, 0.75f); // 与HashMap初始容量和加载因子为16/0.75不同，HashTable是11/0.75  }  **public** Hashtable(**int** initialCapacity, **float** loadFactor) {  **if** (initialCapacity < 0)  **throw** **new** IllegalArgumentException("Illegal Capacity: "+ initialCapacity);  **if** (loadFactor <= 0 || Float.*isNaN*(loadFactor))  **throw** **new** IllegalArgumentException("Illegal Load: "+loadFactor);  **if** (initialCapacity==0)  initialCapacity = 1;  **this**.loadFactor = loadFactor;  table = **new** Entry[initialCapacity];  threshold = (**int**)Math.*min*(initialCapacity \* loadFactor, *MAX\_ARRAY\_SIZE* + 1);  initHashSeedAsNeeded(initialCapacity);  } |

HashTable的put方法源码如下，与HashMap可以有一个null键多个null值不同的是，HashTable中不允许有null键和null值：

|  |
| --- |
| **public** **synchronized** V put(K key, V value) {  // Make sure the value is not null  **if** (value == **null**) {  **throw** **new** NullPointerException();  }  // Makes sure the key is not already in the hashtable.  Entry tab[] = table;  **int** hash = hash(key);  **int** index = (hash & 0x7FFFFFFF) % tab.length;  **for** (Entry<K,V> e = tab[index] ; e != **null** ; e = e.next) {  **if** ((e.hash == hash) && e.key.equals(key)) {  V old = e.value;  e.value = value;  **return** old;  }  }  modCount++;  **if** (count >= threshold) {  // Rehash the table if the threshold is exceeded  rehash();  tab = table;  hash = hash(key);  index = (hash & 0x7FFFFFFF) % tab.length;  }  // Creates the new entry.  Entry<K,V> e = tab[index];  tab[index] = **new** Entry<>(hash, key, value, e);  count++;  **return** **null**;  } |

HashTable的get方法如下：

|  |
| --- |
| **public** **synchronized** V get(Object key) {  Entry tab[] = table;  **int** hash = hash(key);  **int** index = (hash & 0x7FFFFFFF) % tab.length;  **for** (Entry<K,V> e = tab[index] ; e != **null** ; e = e.next) {  **if** ((e.hash == hash) && e.key.equals(key)) {  **return** e.value;  }  }  **return** **null**;  } |

HashTable扩容的源码：

|  |
| --- |
| /\*\*  \* Increases the capacity of and internally reorganizes this  \* hashtable, in order to accommodate and access its entries more  \* efficiently. This method is called automatically when the  \* number of keys in the hashtable exceeds this hashtable's capacity  \* and load factor.  \*/  **protected** **void** rehash() {  **int** oldCapacity = table.length;  Entry<K,V>[] oldMap = table;  // overflow-conscious code  **int** newCapacity = (oldCapacity << 1) + 1;  **if** (newCapacity - *MAX\_ARRAY\_SIZE* > 0) {  **if** (oldCapacity == *MAX\_ARRAY\_SIZE*)  // Keep running with MAX\_ARRAY\_SIZE buckets  **return**;  newCapacity = *MAX\_ARRAY\_SIZE*;  }  Entry<K,V>[] newMap = **new** Entry[newCapacity];  modCount++;  threshold = (**int**)Math.*min*(newCapacity \* loadFactor, *MAX\_ARRAY\_SIZE* + 1);  **boolean** rehash = initHashSeedAsNeeded(newCapacity);  table = newMap;  **for** (**int** i = oldCapacity ; i-- > 0 ;) {  **for** (Entry<K,V> old = oldMap[i] ; old != **null** ; ) {  Entry<K,V> e = old;  old = old.next;  **if** (rehash) {  e.hash = hash(e.key);  }  **int** index = (e.hash & 0x7FFFFFFF) % newCapacity;  e.next = newMap[index];  newMap[index] = e;  }  }  } |

### 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

在多线程程序中，同步修饰符用来控制对临界区代码的访问。其中一种方式是用synchronized关键字来保证代码的线程安全性。在Java中，synchronized修饰的代码块或方法不会被多个线程并发访问。它强制要求线程在进入一个方法之前获得一个锁，在离开方法时释放该锁。它保证了在同一时刻只有一个线程能执行被其修饰的方法。

如果我们把一个方法或代码块定义为同步的，就意味着在同一个对象中，只会有一个对同步方法的调用。如果在一个线程内部调用了一个同步方法，则其他线程会一直阻塞，直到第一个线程完成方法调用。

在进入一个对象的同步方法之前，需要申请对该对象上锁，完成方法调用后释放锁供其他线程申请。同步方法遵循happens-before机制，它保证了对象状态的改变在其他线程中都是可见的。

当标记一个代码块为同步时，需要用一个对象作为参数。当一个运行线程执行到该代码块时，要等到其他运行线程退出这个对象的同步代码区。然而，一个线程可以进入另一个对象的同步代码区。但是同一个对象的非同步方法可以不用申请锁。

如果定义一个静态方法为同步，则是在类上同步，而不是在对象上同步。也即如果一个静态同步方法在执行时，整个类被锁住，对该类中的其他静态方法调用会阻塞。

1）当一个线程进入了一个实例的同步方法，则其他任何线程都不能进入该实例的任何一个同步方法。

2）当一个线程进入了一个类的静态同步方法，则其他任何线程都不能进入该类的任何一个静态同步方法。

**注意：**

同步的静态方法和非静态方法之间没有关系。也即静态同步方法和非静态同步方法可以同时执行，除非非静态同步方法显式在该类上同步（例如，synchronized(MyClass.class){…}）

类的构造函数不能定义成同步的。

### 监视器或内部锁

锁限制了对某个对象状态的访问，同时保证了happens-before关系。

每个对象都有一个锁对象，一个线程在访问对象之前必须申请锁，完成以后释放锁。其他线程不能访问对象，知道获得该对象的锁。这保证了一个线程改变了对象的状态后，新的状态对其他在同一个监视器上线程可见。

当线程释放锁时，会将cache中的内容更新到主内存，这也就使得该对象的状态变化对其他线程是可见的——这就是happens-before关系。

synchronized和volatile，包括Thread.start()和Thread.join()方法，都能保证happens-before关系。

同步语句和同步方法获取的锁相同，某个线程可以请求同一个锁多次。

一个线程获得了对象锁后，不会影响其他线程访问对象的字段或调用对象的非同步方法。

同步语句首先尝试获取对象的锁，获取成功后立即开始执行同步代码块，执行完后释放锁。

如果方法是对象成员或对象实例，线程将锁住该实例。如果方法是静态的，线程锁住的是该类对应的Class对象。同步方法用SYNCHRONIZED标记，该标记被方法调用指令识别。

### 原子变量

来看语句 **int c++**，它包含多个操作，e.g. 从内存读取c的值，将c的值加1，然后写回内存。这个操作对单线程来说是正确的，但是在多线程环境却可能出错。它存在竞态条件，在多线程环境中可能多个线程同时读取c的值

原子访问保证所有操作作为一个整体一次完成。一个原子操作要么完全执行要么完全不执行。

以下这些操作能认为是原子操作：

对引用类型和大部分基本数据类型（long和double类型除外）的读和写操作。

声明为volatile类型变量的读和写操作（包括long和double变量）。

Java并发包[java.util.concurrent.atomic](http://docs.oracle.com/javase/8/docs/api/java/util/concurrent/atomic/package-summary.html)定 义了对单个变量进行原子操作的类。所有类都有get和set方法，就像对volatile变量的读写一样。这就意味着，一个写操作happens- before其他任何对该变量的读操作。原子方法compareAndSet同样有这些特性，就像对整型变量做原子的算术运算一样。

在Java 5.0的并发包中，定义了支持原子操作的类。Java虚拟机编译这些类时利用硬件提供的CAS（Compare and set）来实现。

AtomicInteger

AtomicLong

AtomicBoolean

AtomicReference

### volatile变量

volatile只能用来修饰变量。用volatile修饰的变量可能被异步地修改，所以编译器会对它们特殊处理。

volatile修饰符保证读取某个字段的任何线程都能看到该变量最近被写入的值。

使用volatile修饰的变量降低了内存一致性的风险，因为任何对volatile变量的写操作都能被其他线程可见。另外，当一个线程访问volatile变量时，不止能看到对该变量最近的修改，还能修改该变量的代码所带来的其他影响。

在多线程环境中，对象在不同线程中都保存有副本。但是volatile变量却没有，它们在堆中只有一个实例。这样对volatile变量的修改就能立即对其他线程可见。另外，本地线程缓存没有完成后刷新的工作。

volatile能够保证可见性，但是也带来了竞态条件。它不会锁定等待完成某个操作。例如：

|  |  |
| --- | --- |
| 1 | volatile int i=0; |

两 个线程同时执行 **i +=5** 时，会得到5-10之间的某个值（译者注：原文为i +=5 invoking by two simultaneously thread give result 5 or 10 but it guarantee to see immediate changes 感觉有问题）

**使用场景：**用一个volatile布尔变量作为一个线程终止的标志。

### 静态和volatile变量之间的差别

声明一个静态变量，意味着该类的多个实例将共享该变量，静态变量与类关联而不是与对象关联。线程可能会有静态变量的本地缓存值。

当两个线程同时更新静态（非volatile）变量的值时，可能有一个线程的缓存中是一个过期的值。虽然多线程能够访问的是同一个静态变量，每个线程还是可能会保存自己的缓存副本。

一个volatile变量则在内存中只保留一个副本，该副本在多个线程中共享。

### volatile变量和同步之间的差别

在线程内存和主内存之间，volatile只是同步了一个变量的值，synchronized则同步了（synchronized块中）所有变量的值，并且会锁住和释放一个监视器。所以，synchronized比volatile会有更多的开销。

volatile变量不允许有一个本地副本与主内存中的值不同。一个声明为volatile的变量必须保证所有线程中的副本同步，不管哪个线程修改了变量的值，另外其他线程都能立即看到该值。

### 锁对象

锁对象的作用像synchronized代码使用的隐式锁一样。像隐式锁一样，同时只能有一个线程持有锁。锁还支持wait/notify机制，通过他们之间的condition对象。

锁对象相对于隐式锁最大的优点是，他们能从尝试获得锁的状态返回。如果锁当前不可用或者在一个超时时间之前，tryLock()方法能够返回。在获得锁之前，如果其他线程发送了一个中断，lockInterruptibly()方法能返回。

### [Java内存回收](http://www.codeceo.com/article/java-gc-with-image.html)

在 Java中，创建的对象存放在堆中。Java堆被称为内存回收堆。内存收集不能强制执行。当内存收集器运行时，它释放掉那些不可达对象占用的内存。垃圾收集线程作为一个优先级较低的守护线程运行。你能通过System.gc()提示虚拟机进行垃圾回收，但是不能强迫其执行。

### 如何写一个死锁程序

在多线程环境中，死锁意味着两个或多个线程一直阻塞，等待其他线程释放锁。下面是死锁的一个示例：

public class DeadlockSample {

private final Object obj1 = new Object();

private final Object obj2 = new Object();

public static void main(String[] args) {

DeadlockSample test = new DeadlockSample();

test.testDeadlock();

}

private void testDeadlock() {

Thread t1 = new Thread(new Runnable() {

public void run() {

calLock12();

}

});

Thread t2 = new Thread(new Runnable() {

public void run() {

calLock21();

}

});

t1.start();

t2.start();

}

private void calLock12() {

synchronized (obj1) {

sleep();

synchronized (obj2) {

sleep();

}

}

}

private void calLock21() {

synchronized (obj2) {

sleep();

synchronized (obj1) {

sleep();

}

}

}

private void sleep() {

try {

Thread.sleep(100);

} catch (InterruptedException e) {

e.printStackTrace();

}

}

}

### Java中的引用类型

java.lang.ref包能用来声明软引用（soft reference），弱引用（weak reference）和虚引用（phantom reference）。

垃圾收集器不会回收强引用。

在内存不足时才会回收软引用，所以用它实现缓存可以避免内存不足。

垃圾收集器将会在下一次垃圾收集时回收弱引用。弱引用能被用来实现特殊的map。java.util.WeakHashMap中的key就是弱引用。

虚引用会被立即回收。能被用来跟踪对象被垃圾回收的活动。

## 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。

## 抽象类和接口

抽象类和接口的主要区别如下：

1.abstract class 在 Java 语言中表示的是一种继承关系，一个类只能使用一次继承关系。但是，一个类却可以实现多个interface。

2.在abstract class 中可以有自己的数据成员，也可以有非abstarct的成员方法，而在interface中，只能够有静态的不能被修改的数据成员（也就是必须是static final的，不过在 interface中一般不定义数据成员），所有的成员方法都是abstract的。

3.abstract class和interface所反映出的设计理念不同。其实abstract class表示的是"is-a"关系，interface表示的是"like-a"关系。

4.实现抽象类和接口的类必须实现其中的所有方法。抽象类中可以有非抽象方法。接口中则不能有实现方法。

5.接口中定义的变量默认是public static final 型，且必须给其初值，所以实现类中不能重新定义，也不能改变其值。

6.抽象类中的变量默认是 friendly 型，其值可以在子类中重新定义，也可以重新赋值。

7.接口中的方法默认都是 public,abstract 类型的。

抽象类和接口的对比表

|  |  |  |
| --- | --- | --- |
| **参数** | **抽象类** | **接口** |
| 默认的方法实现 | 它可以有默认的方法实现 | 接口完全是抽象的。它根本不存在方法的实现 |
| 实现 | 子类使用**extends**关键字来继承抽象类。如果子类不是抽象类的话，它需要提供抽象类中所有声明的方法的实现。 | 子类使用关键字**implements**来实现接口。它需要提供接口中所有声明的方法的实现 |
| 构造器 | 抽象类可以有构造器 | 接口不能有构造器 |
| 与正常Java类的区别 | 除了你不能实例化抽象类之外，它和普通Java类没有任何区别 | 接口是完全不同的类型 |
| 访问修饰符 | 抽象方法可以有**public**、**protected**和**default**这些修饰符 | 接口方法默认修饰符是**public**。你不可以使用其它修饰符。 |
| main方法 | 抽象方法可以有main方法并且我们可以运行它 | 接口没有main方法，因此我们不能运行它。 |
| 多继承 | 抽象方法可以继承一个类和实现多个接口 | 接口只可以继承一个或多个其它接口 |
| 速度 | 它比接口速度要快 | 接口是稍微有点慢的，因为它需要时间去寻找在类中实现的方法。 |
| 添加新方法 | 如果你往抽象类中添加新的方法，你可以给它提供默认的实现。因此你不需要改变你现在的代码。 | 如果你往接口中添加方法，那么你必须改变实现该接口的类。 |

**什么时候使用抽象类和接口？**

如果你拥有一些方法并且想让它们中的一些有默认实现，那么使用抽象类吧。

如果你想实现多重继承，那么你必须使用接口。由于**Java不支持多继承**，子类不能够继承多个类，但可以实现多个接口。因此你就可以使用接口来解决它。

如果基本功能在不断改变，那么就需要使用抽象类。如果不断改变基本功能并且使用接口，那么就需要改变所有实现了该接口的类。

## 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：

|  |
| --- |
| **public** **class** Test5 {  **private** **static** **int** a;  **private** **int** b;    **static**{  Test5.a=3;  System.out.println(a);  Test5 t=**new** Test5();  t.f();  t.b=1000;  System.out.println(t.b);  }  **static**{  Test5.a=4;  System.out.println(a);  }  **public** **static** **void** main(String[] args) {  // TODO 自动生成方法存根  }  **static**{  Test5.a=5;  System.out.println(a);  }  **public** **void** f(){  System.out.println("hhahhahah");  }  } |

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：

|  |
| --- |
| **class** UseStatic {  **static** **int** a = 3;  **static** **int** b;    **static** **void** meth(**int** x) {  System.out.println("x = " + x);  System.out.println("a = " + a);  System.out.println("b = " + b);  }    **static** {  System.out.println("Static block initialized.");  b = a \* 4;  }    **public** **static** **void** main(String args[]) {  meth(42);  }  } |

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。

|  |
| --- |
| **class** StaticDemo {  **static** **int** a = 42;  **static** **int** b = 99;  **static** **void** callme() {    System.out.println("a = " + a);  }  }    **class** StaticByName {    **public** **static** **void** main(String args[]) {  StaticDemo.callme();  System.out.println("b = " + StaticDemo.b);  }  } |

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。

## HashMap和HashTable

他们都属于Map接口的类，实现了将唯一键映射到特定的值上。

HashMap默认的初始容量和加载因子分别为16和0.75，而HashTable默认分别为11和0.75。

HashMap类没有分类或者排序。它允许一个null键和多个null值。

HashTable类似于HashMap，但是不允许null键和null值，它也比HashMap慢，因为它是同步的。

HashTable继承自Dictionary类，而HashMap是Java1.2引进的Map接口的一个实现。

两者最大的不同是，HashTable的方法是Synchronize的，而HashMap不是，在多个线程访问HashTable的时候，不需要自己为它的方法实现同步，而HashMap就必须为之提供外同步。

HashTable和HashMap采用的hash/rehash算法类似，所以性能不会有很大的差异。

## 构造函数和析构函数

构造函数可以被重载，但不能被覆盖。

派生类必须通过super调用父类的含有参数的构造函数。

## super关键字

Java里在类中用super调用父类构造函数的时候，为什么调用语句必须是子类的第一条语句？

这是因为如果不放在第一条的话，super之前的语句就起不到作用，也就是说子类所自定义的一些行为就会无效。例如下面一段代码：

|  |
| --- |
| **class** Father  {  **public** Fatcher()  {  String name = **null**;  }  }  **class** Son **extends** Father  {  **public** Son()  {  String name = "Son";  **super**();  }  } |

super()之前的语句（String name = "Son";）实际上是无效的，因此要将super()放到第一句。

## synchronized和java.util.concurrent.locks.Lock的异同

主要相同点是Lock能完成synchronized所实现的所有功能。

主要不同点事Lock有比synchronized更精确的线程语义和更好的性能。synchronized会自动释放锁，而Lock一定要求程序员手动释放，并且必须在finally从句中释放。

# 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

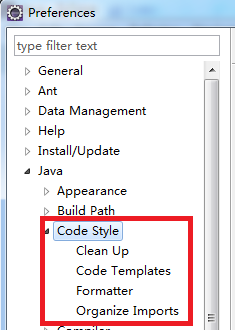
SVN是一种集中式的版本管理工具。

## 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

人工审查：项目组成员之间相互审查代码；

代码提交时审查：只有审查通过的代码才可以加入到git主版本分支中。

## 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。