目录

[问题： 7](#_Toc528681446)

[一、 Java面向对象概述和三大特性 7](#_Toc528681447)

[1、面向对象思想 7](#_Toc528681448)

[2、类和对象的关系 7](#_Toc528681449)

[3、封装思想 8](#_Toc528681450)

[4、继承和类实例化过程 10](#_Toc528681451)

[5、多态、反射和组件解耦 14](#_Toc528681452)

[6、重写和重载 15](#_Toc528681453)

[A. 重写 15](#_Toc528681454)

[方法的重写规则 15](#_Toc528681455)

[B. 重载 18](#_Toc528681456)

[重载规则: 18](#_Toc528681457)

[C. 重写与重载之间的区别 19](#_Toc528681458)

[二、 Java基础 20](#_Toc528681459)

[1. 变量的创建 20](#_Toc528681460)

[2. 访问控制 21](#_Toc528681461)

[3. 代码块 23](#_Toc528681462)

[A. 普通代码块 23](#_Toc528681463)

[B. 构造代码块 23](#_Toc528681464)

[C. 静态代码块: 24](#_Toc528681465)

[4. Static 26](#_Toc528681466)

[5. Final 26](#_Toc528681467)

[6. Transient 27](#_Toc528681468)

[7. Volatile 27](#_Toc528681469)

[8. 并发编程的三大特性 28](#_Toc528681470)

[9. Interface与abstract类 28](#_Toc528681471)

[10. 数据类型的默认值和长度 29](#_Toc528681472)

[A. 数据类型分类 29](#_Toc528681473)

[B. 默认值 30](#_Toc528681474)

[C. 长度 30](#_Toc528681475)

[D. 不同数据类型运算转换 31](#_Toc528681476)

[E. 包装器类型 32](#_Toc528681477)

[11. Java中创建对象的5种方式 32](#_Toc528681478)

[12. Java运算符优先级 33](#_Toc528681479)

[13. This、Super 34](#_Toc528681480)

[14. 强引用，软引用，弱引用，虚引用？ 35](#_Toc528681481)

[15. Java中final、finally和finalize的区别 36](#_Toc528681482)

[16. 进程间通信 37](#_Toc528681483)

[17. 补码 37](#_Toc528681484)

[三、 内部类 38](#_Toc528681485)

[1.为什么使用内部类? 38](#_Toc528681486)

[2.内部类分类: 38](#_Toc528681487)

[A. 成员内部类: 38](#_Toc528681488)

[B. 静态内部类: 是 static 修饰的内部类, 39](#_Toc528681489)

[C. 方法内部类：其作用域仅限于方法内，方法外部无法访问该内部类 40](#_Toc528681490)

[D. 匿名内部类: 41](#_Toc528681491)

[四、 多线程 42](#_Toc528681492)

[1. 状态 42](#_Toc528681493)

[2. 创建线程的方法 44](#_Toc528681494)

[A. 通过实现 Runnable 接口 44](#_Toc528681495)

[B. 通过继承 Thread 类本身 45](#_Toc528681496)

[C. 通过 Callable 和 Future 创建线程。 48](#_Toc528681497)

[D. 创建线程的三种方式的对比 49](#_Toc528681498)

[E. 线程池 49](#_Toc528681499)

[a) 线程池，其实就是一个容纳多个线程的容器，其中的线程可以反复使用，省去了频繁创建线程对象的操作，无需反复创建线程而消耗过多资源。（是什么） 49](#_Toc528681500)

[b) 那么，我们为什么需要用到线程池呢？每次用的时候手动创建不行吗？ 49](#_Toc528681501)

[c) 线程池都是通过线程池工厂创建，再调用线程池中的方法获取线程，再通过线程去执行任务方法。 50](#_Toc528681502)

[d) 这里介绍两种使用线程池创建线程的方法 50](#_Toc528681503)

[3. 线程间的通信方式 52](#_Toc528681504)

[A. synchronized同步 52](#_Toc528681505)

[B. while轮询的方式 53](#_Toc528681506)

[C. wait/notify机制 56](#_Toc528681507)

[D. 管道通信 58](#_Toc528681508)

[4. 产生死锁的四个必要条件 61](#_Toc528681509)

[5. 线程的优先级 62](#_Toc528681510)

[6. JAVA多线程和并发基础面试问答 62](#_Toc528681511)

[1) 进程和线程之间有什么不同？ 62](#_Toc528681512)

[2) 多线程编程的好处是什么？ 62](#_Toc528681513)

[3) 用户线程和守护线程有什么区别？ 63](#_Toc528681514)

[4) 我们如何创建一个线程？ 63](#_Toc528681515)

[5) 有哪些不同的线程生命周期？ 63](#_Toc528681516)

[6) 可以直接调用Thread类的run()方法么？ 63](#_Toc528681517)

[7) 如何让正在运行的线程暂停一段时间？ 64](#_Toc528681518)

[8) 你对线程优先级的理解是什么？ 64](#_Toc528681519)

[9) 什么是线程调度器(Thread Scheduler)和时间分片(Time Slicing)？ 64](#_Toc528681520)

[10) 在多线程中，什么是上下文切换(context-switching)？ 64](#_Toc528681521)

[11) 你如何确保main()方法所在的线程是Java程序最后结束的线程？ 65](#_Toc528681522)

[12) 线程之间是如何通信的？ 65](#_Toc528681523)

[13) 为什么线程通信的方法wait(), notify()和notifyAll()被定义在Object类里？ 65](#_Toc528681524)

[14) 为什么wait(), notify()和notifyAll()必须在同步方法或者同步块中被调用？ 65](#_Toc528681525)

[15) 为什么Thread类的sleep()和yield()方法是静态的？ 66](#_Toc528681526)

[16) 如何确保线程安全？ 66](#_Toc528681527)

[17) volatile关键字在Java中有什么作用？ 66](#_Toc528681528)

[18) 同步方法和同步块，哪个是更好的选择？ 66](#_Toc528681529)

[19) 如何创建守护线程？ 67](#_Toc528681530)

[20) 什么是ThreadLocal? 67](#_Toc528681531)

[21) 什么是Thread Group？为什么不建议使用它？ 67](#_Toc528681532)

[22) 什么是Java线程转储(Thread Dump)，如何得到它？ 68](#_Toc528681533)

[23) 什么是死锁(Deadlock)？如何分析和避免死锁？ 68](#_Toc528681534)

[24) 什么是Java Timer类？如何创建一个有特定时间间隔的任务？ 68](#_Toc528681535)

[25) 什么是线程池？如何创建一个Java线程池？ 69](#_Toc528681536)

[7. Java并发面试问题 69](#_Toc528681537)

[1) 什么是原子操作？在Java Concurrency API中有哪些原子类(atomic classes)？ 69](#_Toc528681538)

[2) Java Concurrency API中的Lock接口(Lock interface)是什么？对比同步它有什么优势？ 70](#_Toc528681539)

[3) 什么是Executors框架？ 70](#_Toc528681540)

[4) 什么是阻塞队列？如何使用阻塞队列来实现生产者-消费者模型？ 71](#_Toc528681541)

[5) 什么是Callable和Future? 71](#_Toc528681542)

[6) 什么是FutureTask? 72](#_Toc528681543)

[7) 什么是并发容器的实现？ 72](#_Toc528681544)

[8) Executors类是什么？ 72](#_Toc528681545)

[9) 为什么说ConcurrentHashMap是弱一致性的？以及为何多个线程并发修改ConcurrentHashMap时不会报ConcurrentModificationException？ 73](#_Toc528681546)

[五、 集合 73](#_Toc528681547)

[1. 概述 73](#_Toc528681548)

[2. List类集合 75](#_Toc528681549)

[A. ArrayList 76](#_Toc528681550)

[B. LinkedList 77](#_Toc528681551)

[C. ArrayList和LinkedList的大致区别 77](#_Toc528681552)

[D. Vector 78](#_Toc528681553)

[E. CopyOnWriteArrayList 78](#_Toc528681554)

[3. Map类集合 79](#_Toc528681555)

[A. HashMap 80](#_Toc528681556)

[HashMap的实现原理 81](#_Toc528681557)

[1. HashMap的数据结构 81](#_Toc528681558)

[2. HashMap的存取实现 83](#_Toc528681559)

[3. 解决hash冲突的办法 86](#_Toc528681560)

[4. 再散列rehash过程 87](#_Toc528681561)

[HashMap集合的四种遍历方式 88](#_Toc528681562)

[B. Hashtable 89](#_Toc528681563)

[C. ConcurrentHashMap 90](#_Toc528681564)

[D. LinkedHashMap 91](#_Toc528681565)

[E. TreeMap 92](#_Toc528681566)

[F. ConcurrentSkipListMap 92](#_Toc528681567)

[4. Set类集合 94](#_Toc528681568)

[5. Queue/Deque类集合 94](#_Toc528681569)

[A. ConcurrentLinkedQueue 96](#_Toc528681570)

[B. LinkedBlockingQueue 96](#_Toc528681571)

[C. ArrayBlockingQueue 97](#_Toc528681572)

[D. SynchronousQueue 97](#_Toc528681573)

[E. PriorityQueue & PriorityBlockingQueue 98](#_Toc528681574)

[六、 异常 99](#_Toc528681575)

[1. 异常类型 99](#_Toc528681576)

[2. finally关键字 100](#_Toc528681577)

[3. Java内置异常 100](#_Toc528681578)

[4. 异常方法 102](#_Toc528681579)

[5. 使用throws抛出异常 102](#_Toc528681580)

[6. 使用throw抛出异常 103](#_Toc528681581)

[Java7增强的throw语句： 104](#_Toc528681582)

[7. 自定义异常类 104](#_Toc528681583)

[8. catch和throw同时使用 105](#_Toc528681584)

[9. 异常链 106](#_Toc528681585)

[10. Java的异常跟踪栈 106](#_Toc528681586)

[11. 异常处理规则 106](#_Toc528681587)

[七、 设计模式 107](#_Toc528681588)

[六大原则 107](#_Toc528681589)

[三种类型 109](#_Toc528681590)

[1. 单例模式 Singleton 111](#_Toc528681591)

[2. 简单工厂模式 StaticFactory Method 111](#_Toc528681592)

[3. 工厂模式 Factory Method 111](#_Toc528681593)

[A. 介绍 111](#_Toc528681594)

[B. 实现 112](#_Toc528681595)

[4. 抽象工厂模式 Abstract Factory Method 115](#_Toc528681596)

[介绍 115](#_Toc528681597)

[实现 116](#_Toc528681598)

[5. 代理模式 Proxy Method 122](#_Toc528681599)

[6. 命令模式 Command Method 122](#_Toc528681600)

[7. 策略模式 Strategy Method 122](#_Toc528681601)

[8. 观察者模式 Observer Method 122](#_Toc528681602)

[介绍 122](#_Toc528681603)

[实现 127](#_Toc528681604)

[9. 生产者消费者模式 ProducerConsumer Method 127](#_Toc528681605)

[八、 输入输出流 127](#_Toc528681606)

[1. 概述 127](#_Toc528681607)

[2. InputStream类 128](#_Toc528681608)

[3. Reader类 129](#_Toc528681609)

[4. OutputStream类 129](#_Toc528681610)

[5. Writer类 129](#_Toc528681611)

[6. File类 130](#_Toc528681612)

[A:File类的概述 130](#_Toc528681613)

[B:构造方法 130](#_Toc528681614)

[创建功能 131](#_Toc528681615)

[判断功能 131](#_Toc528681616)

[获取功能 131](#_Toc528681617)

[文件名称过滤器的概述 132](#_Toc528681618)

[文件名称过滤器的概述 134](#_Toc528681619)

[7. 下面以一些字节输入输出流具体的案例操作（操作的时候认清自己使用的是字节流还是字符流）： 135](#_Toc528681620)

[8. IO流(BufferedInputStream和BufferOutputStream拷贝) 137](#_Toc528681621)

[9. 字符流FileReader和FileWriter 138](#_Toc528681622)

[九、 Java内存模型 140](#_Toc528681623)

[1. 简介 140](#_Toc528681624)

[2. 堆、栈、方法区 141](#_Toc528681625)

[堆区: 141](#_Toc528681626)

[栈区: 141](#_Toc528681627)

[堆和栈的联系 142](#_Toc528681628)

[方法区: 142](#_Toc528681629)

[3. 常量池 143](#_Toc528681630)

[1.字符串常量池(String Constant Pool): 144](#_Toc528681631)

[1.1:字符串常量池在Java内存区域的哪个位置？ 144](#_Toc528681632)

[1.2:字符串常量池是什么？ 144](#_Toc528681633)

[1.3:字符串常量池里放的是什么？ 144](#_Toc528681634)

[2.class常量池(Class Constant Pool)： 145](#_Toc528681635)

[2.1:class常量池简介： 145](#_Toc528681636)

[2.2:什么是字面量和符号引用： 145](#_Toc528681637)

[3.运行时常量池(Runtime Constant Pool)： 145](#_Toc528681638)

[4. 分代 145](#_Toc528681639)

[A. 年轻代（Young Generation） 145](#_Toc528681640)

[B. 年老代（Old Generation） 146](#_Toc528681641)

[C. 持久代（Permanent Generation） 146](#_Toc528681642)

[5. 垃圾回收 147](#_Toc528681643)

[A. Java垃圾回收概况 147](#_Toc528681644)

[B. 算法 148](#_Toc528681645)

[a) 引用计数法(Reference Counting Collector) 148](#_Toc528681646)

[b) 最基础的收集算法 — 标记/清除算法 149](#_Toc528681647)

[c) 复制算法 150](#_Toc528681648)

[d) 标记/整理算法 151](#_Toc528681649)

[e) GC Roots 的对象 152](#_Toc528681650)

[f) 七种垃圾回收器 152](#_Toc528681651)

[g) 对象A和B循环引用，最后会不会不被GC回收？ 153](#_Toc528681652)

[十、 JDBC 155](#_Toc528681653)

[1. JDBC是什么？ 155](#_Toc528681654)

[2. 数据库驱动与常用接口 155](#_Toc528681655)

[3. JDBC使用步骤 156](#_Toc528681656)

[4. SQL语法 160](#_Toc528681657)

[5. MySQL 事务 161](#_Toc528681658)

[事务控制语句： 162](#_Toc528681659)

[MYSQL 事务处理主要有两种方法： 163](#_Toc528681660)

[四种事务隔离级别 163](#_Toc528681661)

[6. MySql索引 165](#_Toc528681662)

[A. 索引介绍 165](#_Toc528681663)

[索引原理 166](#_Toc528681664)

[索引的类型 167](#_Toc528681665)

[索引的操作 168](#_Toc528681666)

[B. B+树索引原理 171](#_Toc528681667)

[b+树的查找过程 172](#_Toc528681668)

[b+树性质 172](#_Toc528681669)

[慢查询优化 173](#_Toc528681670)

[C. 建索引的几大原则 173](#_Toc528681671)

[D. 数据库索引和选择性的关系 174](#_Toc528681672)

[怎么计算索引的选择性？ 174](#_Toc528681673)

[选择性对数据库索引有什么影响？ 174](#_Toc528681674)

[什么时候不应该使用数据库索引？ 175](#_Toc528681675)

[“选择性”等于多少才会使用索引？ 175](#_Toc528681676)

[十一、 反射 175](#_Toc528681677)

[1. 反射的概述 175](#_Toc528681678)

[2. 查看Class类在java中的api详解（1.7的API） 176](#_Toc528681679)

[3. 反射的使用（这里使用Student类做演示） 177](#_Toc528681680)

[A. 获取Class对象的三种方式 177](#_Toc528681681)

[B. 通过反射获取构造方法并使用： 178](#_Toc528681682)

[C. 获取成员变量并调用 182](#_Toc528681683)

[D. 获取成员方法并调用 184](#_Toc528681684)

[E. 反射main方法 188](#_Toc528681685)

[F. 反射方法的其它使用之---通过反射运行配置文件内容 189](#_Toc528681686)

[G. 反射方法的其它使用之---通过反射越过泛型检查 190](#_Toc528681687)

[十二、 时间复杂度 191](#_Toc528681688)

[1. 概述 191](#_Toc528681689)

[时间频度 191](#_Toc528681690)

[2. 常见算法时间复杂度： 192](#_Toc528681691)

[3. 算法的时间复杂度（计算实例） 193](#_Toc528681692)

## 问题：

B\*tree创建的一般性原则，什么时候用全表查询，

数据的选择性

运算符的优先级

跟着视频做的也没关系，只要能把项目的细节、框架、技术，能挖多深就多深，项目的难点（比如高并发处理）  
另外Java基础（多线程+JVM）特别重要，MySQL的话优化也要了解

nio

## [Java面向对象概述和三大特性](https://www.cnblogs.com/xugf/p/8475931.html)

Java 是面向对象的高级编程语言，类和对象是 Java 程序的构成核心。围绕着 Java 类和 Java 对象，有三大基本特性：封装是 Java 类的编写规范、继承是类与类之间联系的一种形式、而多态为系统组件或模块之间解耦提供了解决方案。

### 1、面向对象思想

　　面向对象编程是当今主流的程序设计思想，已经取代了过程化程序开发技术，Java 是完全面向对象编程语言，所以必须熟悉面向对象才能够编写 Java 程序。

　　面向对象的程序核心是由对象组成的，每个对象包含着对用户公开的特定功能和隐藏的实现部分。程序中的很多对象来自 JDK 标准库，而更多的类需要我们程序员自定义。

　　从理论上讲，只要对象能够实现业务功能，其具体的实现细节不必特别关心。

　　面向对象有以下特点：

　　（1）面向对象是一种常见的思想，比较符合人们的思考习惯；

　　（2）面向对象可以将复杂的业务逻辑简单化，增强代码复用性；

　　（3）面向对象具有抽象、封装、继承、多态等特性。

　　面向对象的编程语言主要有：C++、Java、C#等。

### 2、类和对象的关系

**类：**

　　对某类事物的普遍一致性特征、功能的抽象、描述和封装，是构造对象的模版或蓝图，用 Java 编写的代码都会在某些类的内部。类之间主要有：依赖、聚合、继承等关系。

**对象：**

　　使用 new 关键字或反射技术创建的某个类的实例。同一个类的所有对象，都具有相似的数据（比如人的年龄、性别）和行为（比如人的吃饭、睡觉），但是每个对象都保存着自己独特的状态，对象状态会随着程序的运行而发生改变，需要注意状态的变化必须通过调用方法来改变，这就是封装的基本原则。

### 3、封装思想

　　核心思想就是“隐藏细节”、“数据安全”：将对象不需要让外界访问的成员变量和方法私有化，只提供符合开发者意愿的公有方法来访问这些数据和逻辑，保证了数据的安全和程序的稳定。

　　具体的实现方式就是：

　　使用 private 修饰符把成员变量设置为私有，防止外部程序直接随意调用或修改成员变量，然后对外提供 public 的 set 和 get 方法按照开发者的意愿（可以编写一些业务逻辑代码，虽然很少这样做）设置和获取成员变量的值。

　　也可以把只在本类内部使用的方法使用 private，这就是封装的思想，是面向对象最基本的开发规范之一。

　　在此，我们有必要说一下 Java 的访问权限修饰关键字。Java 中主要有 private、protected、public 和 默认访问权限 四种：

　　public 修饰符，具有最大的访问权限，可以访问任何一个在 CLASSPATH 下的类、接口、异常等。

　　protected 修饰符，主要作用就是用来保护子类，子类可以访问这些成员变量和方法，其余类不可以。

　　default 修饰符，主要是本包的类可以访问。

　　private 修饰符，访问权限仅限于本类内部，在实际开发过程中，大多数的成员变量和方法都是使用 private 修饰的。



　Java 的访问控制是停留在编译层的，只在编译时进行访问权限检查，不会在类文件中留下痕迹。

　　通过反射机制，还是可以访问类的私有成员的。

 　　我们举个小例子：

public class MobilePhone {

// 使用 private 关键字把成员变量私有化

private String os;

private String phoneNumber;

private String brand;

private double dumpEnergy;

// 对外提供访问、设置成员变量的 public 方法

// 这样就可以按照我们自己的意愿来访问、设置成员变量

// 而且也有助于在方法内部对数据有效性进行验证

public void setOs(String os){

this.os = os;

}

public String getOs(){

return this.os;

}

public void setPhoneNumber(String phoneNumber){

this.phoneNumber = phoneNumber;

}

public String getPhoneNumber(){

return this.phoneNumber;

}

public void setBrand(String brand){

this.brand = brand;

}

public String getBrand(){

return this.brand;

}

public void setDumpEnergy(double dumpEnergy){

this.dumpEnergy = dumpEnergy;

}

public double getDumpEnergy(){

return this.dumpEnergy;

}

// 发短信的方法，不需要做修改

public void sendMessage(String message, String targetPhoneNumber){

System.out.println("发给" + targetPhoneNumber + ", 内容是：" + message);

}

// 充电方法，不需要做修改

public double charge(){

System.out.println("正在充电, 剩余电量：" + dumpEnergy \* 100 + "%");

return dumpEnergy;

}

// 对外提供的开机方法

public void startup(){

System.out.println("正在开机......");

// 调用私有开机方法

startup2();

System.out.println("完成开机");

}

// 私有的开机方法，封装开机细节

private void startup2(){

System.out.println("启动操作系统......");

System.out.println("加载开机启动项......");

System.out.println("......");

}

}

在实际的开发过程中，这样的封装方式已经成了 Java Bean 代码编写的规范。现在主流的框架在使用反射技术为对象赋值、取值时使用的都是 set 和 get 方法，而不是直接操作字段的值。

### 4、继承和类实例化过程

　　（1）在多个不同的类中抽取出共性的数据和逻辑，对这些共性的内容进行封装一个新的类即父类（也叫做超类或基类），让之前的类来继承这个类，那些共性的内容在子类中就不必重复定义，比如 BaseDAO、BaseAction 等。

　　\* （2）Java 的继承机制是单继承，即一个类只能有一个直接父类。

　　\* （3）如果子类和父类有同名成员变量和方法，子类可以使用 super 关键字调用父类的成员变量和方法，上述使用方式前提是成员在子类可见。

　　\* （4）在调用子类构造方法时，会隐式的调用父类的构造方法 super()。如果父类没有无参构造方法，为了避免编译错误，需要在子类构造方法中显式的调用父类的含参构造方法。

　　（5）子类创建时调用父类构造方法：子类需要使用父类的成员变量和方法，所以就要调用父类构造方法来初始化，之后再进行子类成员变量和方法的初始化。因此，构造方法是无法覆盖的。

　　\* （6）当子类需要扩展父类的某个方法时，可以覆盖父类方法，但是子类方法访问权限必须大于或等于父类权限。

　　（7）继承提高了程序的复用性、扩展性，也是 Java 语言多态特征的前提。

　　（8）在实际开发、程序设计过程中，并非先有的父类，而是先有了子类中通用的数据和逻辑，然后再抽取封装出来的父类。

　　我们简单了解下类的实例化过程

　　（1）JVM 读取指定 classpath 路径下的 class 文件，加载到内存，如果有直接父类，也会加载父类；

　　（2）堆内存分配空间；

　　（3）执行父类、子类静态代码块；

　　（4）对象属性进行默认初始化；

　　（5）调用构造方法；

　　（6）在构造方法中，先调用父类构造方法初始化父类数据；

　　（7）初始化父类数据后，显示初始化，执行子类的构造代码块；

　　（8）再进行子类构造方法的特定初始化；

　　（9）初始化完毕后，将地址赋值给引用

　　为了说明上面的内容，我们来编写一个简单的例子，实际意义并不大，只是为了演示类继承实例化的过程。

/\*

父类

\*/

class Parent {

int num = 5;

static {

System.out.println("父类静态代码块");

System.out.println();

}

{

System.out.println("父类构造代码块1，" + num);

num = 1;

System.out.println("父类构造代码块2，" + num);

doSomething();

System.out.println();

}

Parent() {

System.out.println("父类构造方法1，" + num);

num = 2;

System.out.println("父类构造方法2，" + num);

doSomething();

System.out.println();

}

void doSomething() {

System.out.println("父类doSomething方法1，" + num);

num = 3;

System.out.println("父类doSomething方法2，" + num);

System.out.println();

}

}

/\*

子类

\*/

class Child extends Parent {

int num = 10;

/\*

静态代码块，在类加载时执行

\*/

static {

System.out.println("子类静态代码块");

System.out.println();

}

/\*

构造代码块

\*/

{

System.out.println("子类构造代码块1，" + num);

num = 11;

System.out.println("子类构造代码块2，" + num);

doSomething();

System.out.println();

}

Child() {

System.out.println("子类构造方法1，" + num);

num = 12;

System.out.println("子类构造方法2，" + num);

doSomething();

System.out.println();

}

void doSomething() {

System.out.println("子类doSomething方法1，" + num);

num = 13;

System.out.println("子类doSomething方法2，" + num);

System.out.println();

}

}

public class A {

public static void main(String[] args) {

Child child = new Child();

child.num = 20;

child.doSomething();

}

}

输出：

父类静态代码块

子类静态代码块

父类构造代码块1，5

父类构造代码块2，1

子类doSomething方法1，0

子类doSomething方法2，13

父类构造方法1，1

父类构造方法2，2

子类doSomething方法1，13

子类doSomething方法2，13

子类构造代码块1，10

子类构造代码块2，11

子类doSomething方法1，11

子类doSomething方法2，13

子类构造方法1，13

子类构造方法2，12

子类doSomething方法1，12

子类doSomething方法2，13

子类doSomething方法1，20

子类doSomething方法2，13

### 5、多态、反射和组件解耦

　　多态指允许不同类的对象对同一“消息”做出响应。即同一消息可以根据发送对象的不同而采用多种不同的行为方式。可以用于消除类型之间的耦合关系，Spring 的核心就是多态和面向接口编程。

　　（1）Java 中可以使用父类、接口变量引用子类、实现类对象；

　　（2）在这个过程中，会对子类、实现类对象做自动类型提升，其特有功能就无法访问了，如果需要使用，可以做强制类型转换。

　　Java 的反射技术和多态特性是框架开发、组件解耦的核心，在这方面，Spring 的 IOC 和 DI 为我们提供了一个极好的学习范例，Spring 的 IOC 使用反射技术创建、管理对象，DI 使用多态技术为组件注入依赖对象。

　　在没有学习 Spring 之前，简单的解决方案是使用一个 .properties 文件保存程序中使用的接口、实现类类型键值信息，然后在程序中使用一个全局 Properties 对象保存这些信息，并且使用反射技术把这些实现类初始化、提供一个静态的方法获取指定接口的实现类对象，在组件中就可以使用依赖对象的键获取需要的对象。

这样的方案带来的好处就是：当我们需要修改某个组件的实现方式时，比如把之前 JDBC 的 DAO 实现改为 Hibernate 实现，只要把这些新的实现类放到 classpath 下，把 .properties 文件对应接口的实现类类型改成新的 Hibernate 实现类，而不需要修改依赖组件的代码。

### 6、重写和重载

#### 重写

重写是子类对父类的允许访问的方法的实现过程进行重新编写, 返回值和形参都不能改变。即外壳不变，核心重写！

重写的好处在于子类可以根据需要，定义特定于自己的行为。 也就是说子类能够根据需要实现父类的方法。

重写方法不能抛出新的检查异常或者比被重写方法申明更加宽泛的异常。例如： 父类的一个方法申明了一个检查异常 IOException，但是在重写这个方法的时候不能抛出 Exception 异常，因为 Exception 是 IOException 的父类，只能抛出 IOException 的子类异常。

##### 方法的重写规则

* 参数列表必须完全与被重写方法的相同；
* 返回类型必须完全与被重写方法的返回类型相同；
* 访问权限不能比父类中被重写的方法的访问权限更低。例如：如果父类的一个方法被声明为public，那么在子类中重写该方法就不能声明为protected。
* 父类的成员方法只能被它的子类重写。
* 声明为final的方法不能被重写。
* 声明为static的方法不能被重写，但是能够被再次声明。
* 子类和父类在同一个包中，那么子类可以重写父类所有方法，除了声明为private和final的方法。
* 子类和父类不在同一个包中，那么子类只能够重写父类的声明为public和protected的非final方法。
* 重写的方法能够抛出任何非强制异常，无论被重写的方法是否抛出异常。但是，重写的方法不能抛出新的强制性异常，或者比被重写方法声明的更广泛的强制性异常，反之则可以。
* 构造方法不能被重写。
* 如果不能继承一个方法，则不能重写这个方法。

**当需要在子类中调用父类的被重写方法时，要使用super关键字。**

* class Animal{  
   public void move(){  
   System.*out*.println("动物可以移动");  
   }  
  }  
    
  class Dog extends Animal{  
   public void move(){  
   super.move(); // 应用super类的方法  
   System.*out*.println("狗可以跑和走");  
   }  
  }  
    
  public class TestDog{  
   public static void main(String args[]){  
    
   Animal b = new Dog(); // Dog 对象  
   b.move(); //执行 Dog类的方法  
    
   }  
  }

**在面向对象原则里，重写意味着可以重写任何现有方法。实例如下：**

**TestDog.java 文件代码：**

class Animal{  
 public void move(){  
 System.*out*.println("动物可以移动");  
 }  
}  
  
class Dog extends Animal{  
 public void move(){  
 System.*out*.println("狗可以跑和走");  
 }  
}  
  
public class TestDog{  
 public static void main(String args[]){  
 Animal a = new Animal(); // Animal 对象  
 Animal b = new Dog(); // Dog 对象  
  
 a.move();// 执行 Animal 类的方法  
  
 b.move();//执行 Dog 类的方法  
 }  
}

以上实例编译运行结果如下：

动物可以移动

狗可以跑和走

在上面的例子中可以看到，尽管b属于Animal类型，但是它运行的是Dog类的move方法。

这是由于在编译阶段，只是检查参数的引用类型。

然而在运行时，Java虚拟机(JVM)指定对象的类型并且运行该对象的方法。

因此在上面的例子中，之所以能编译成功，是因为Animal类中存在move方法，然而运行时，运行的是特定对象的方法。

思考以下例子：

**TestDog.java 文件代码：**

class Animal{  
 public void move(){  
 System.*out*.println("动物可以移动");  
 }  
}  
  
class Dog extends Animal{  
 public void move(){  
 System.*out*.println("狗可以跑和走");  
 }  
 public void bark(){  
 System.*out*.println("狗可以吠叫");  
 }  
}  
  
public class TestDog{  
 public static void main(String args[]){  
 Animal a = new Animal(); // Animal 对象  
 Animal b = new Dog(); // Dog 对象  
  
 a.move();// 执行 Animal 类的方法  
 b.move();//执行 Dog 类的方法  
 b.bark();  
 }  
}

以上实例编译运行结果如下：

TestDog.java:30: cannot find symbol

symbol : method bark()

location: class Animal

b.bark();

^

该程序将抛出一个编译错误，因为b的引用类型Animal没有bark方法。

#### 重载

重载(overloading) 是在一个类里面，方法名字相同，而参数不同。返回类型可以相同也可以不同。

每个重载的方法（或者构造函数）都必须有一个独一无二的参数类型列表。

最常用的地方就是构造器的重载。

##### 重载规则:

* 被重载的方法必须改变参数列表(参数个数或类型不一样)；
* 被重载的方法可以改变返回类型；
* 被重载的方法可以改变访问修饰符；
* 被重载的方法可以声明新的或更广的检查异常；
* 方法能够在同一个类中或者在一个子类中被重载。
* 无法以返回值类型作为重载函数的区分标准。
* public class Overloading {  
   public int test(){  
   System.*out*.println("test1");  
   return 1;  
   }  
    
   public void test(int a){  
   System.*out*.println("test2");  
   }  
    
   //以下两个参数类型顺序不同  
   public String test(int a,String s){  
   System.*out*.println("test3");  
   return "returntest3";  
   }  
    
   public String test(String s,int a){  
   System.*out*.println("test4");  
   return "returntest4";  
   }  
    
   public static void main(String[] args){  
   Overloading o = new Overloading();  
   System.*out*.println(o.test());  
   o.test(1);  
   System.*out*.println(o.test(1,"test3"));  
   System.*out*.println(o.test("test4",1));  
   }  
  }

#### 重写与重载之间的区别

|  |  |  |
| --- | --- | --- |
| **区别点** | **重载方法** | **重写方法** |
| 参数列表 | 必须修改 | 一定不能修改 |
| 返回类型 | 可以修改 | 一定不能修改 |
| 异常 | 可以修改 | 可以减少或删除，一定不能抛出新的或者更广的异常 |
| 访问 | 可以修改 | 一定不能做更严格的限制（可以降低限制） |

**总结**

方法的重写(Overriding)和重载(Overloading)是java多态性的不同表现，重写是父类与子类之间多态性的一种表现，重载可以理解成多态的具体表现形式。

* (1)方法重载是一个类中定义了多个方法名相同,而他们的参数的数量不同或数量相同而类型和次序不同,则称为方法的重载(Overloading)。
* (2)方法重写是在子类存在方法与父类的方法的名字相同,而且参数的个数与类型一样,返回值也一样的方法,就称为重写(Overriding)。
* (3)方法重载是一个类的多态性表现,而方法重写是子类与父类的一种多态性表现。



## Java基础

### 变量的创建

成员/实例变量在类实例化对象时候创建，在对象销毁的时候销毁。

局部变量在局部范围使用时创建，跳出局部范围销毁。

静态成员/类变量在类装载的时候就进行了创建，在整个程序结束时按序销毁。

1.局部变量：在方法、构造方法、语句块中定义的变量。其声明和初始化在方法中实现，在方法结束后自动销毁

public class ClassName{

    public void printNumber（）{

        int a;

    }

    // 其他代码

}

2.成员变量：定义在类中，方法体之外。变量在创建对象时实例化。成员变量可被类中的方法、构造方法以及特定类的语句块访问。

public class ClassName{

    int a;

    public void printNumber（）{

        // 其他代码

    }

}

3.类变量：定义在类中，方法体之外，但必须要有 static 来声明变量类型。静态成员属于整个类，可通过对象名或类名来调用。

public class ClassName{

    static int a;

    public void printNumber（）{

        // 其他代码

    }

}

### 访问控制

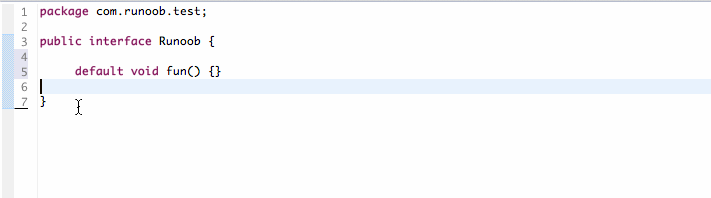


（1）：受保护的访问修饰符-protected

protected 需要从以下两个点来分析说明：

* **子类与基类在同一包中**：被声明为 protected 的变量、方法和构造器能被同一个包中的任何其他类访问；
* **子类与基类不在同一包中**：那么在子类中，子类实例可以访问其从基类继承而来的 protected 方法，而不能访问基类实例的protected方法。
* package p2;  
   class MyObject2 {  
   protected Object clone() throws CloneNotSupportedException{  
   return super.clone();  
   }  
   }  
     
  package p22;  
   public class Test2 extends MyObject2 {  
   public static void main(String args[]) {  
   MyObject2 obj = new MyObject2();  
   obj.clone(); // Compile Error ----（1）  
    
   Test2 tobj = new Test2();  
   tobj.clone(); // Complie OK ----（2）  
   }  
   }
* 对于(1)而言，clone()方法来自于类MyObject2本身，因此其可见性为包p2及MyObject2的子类，虽然Test2是MyObject2的子类，但在Test2中不能访问基类MyObject2的protected方法clone()，因此编译不通过;对于(2)而言，由于在Test2中访问的是其本身实例的从基类MyObject2继承来的的clone()，因此编译通过。

protected 访问修饰符可以修饰类及其方法和成员变量，但是接口及接口的成员变量和成员方法不能声明为 protected。 可以看看以下颜色图理解：



子类能访问 protected 修饰符声明的方法和变量，这样就能保护不相关的类使用这些方法和变量。

下面的父类使用了 protected 访问修饰符，子类重写了父类的 openSpeaker() 方法。

class AudioPlayer {  
 protected boolean openSpeaker(Speaker sp) {  
 // 实现细节  
 }  
}  
  
class StreamingAudioPlayer extends AudioPlayer {  
 protected boolean openSpeaker(Speaker sp) {  
 // 实现细节  
 }  
}

如果把 openSpeaker() 方法声明为 private，那么除了 AudioPlayer 之外的类将不能访问该方法。

如果把 openSpeaker() 声明为 public，那么所有的类都能够访问该方法。

如果我们只想让该方法对其所在类的子类可见，则将该方法声明为 protected。

### 代码块

#### 普通代码块

在方法或语句中出现的{}就称为普通代码块。普通代码块和一般语句的执行顺序由他们在代码中出现的次序决定，先出现先执行。

public class Test {  
 public static void main(String[] args) {  
 {  
 int x = 3;  
 System.*out*.println("普通代码块内的变量x=" + x);  
 }  
 int x = 1;  
 System.*out*.println("主方法内的变量x=" + x);  
 {  
 int y = 7;  
 System.*out*.println("普通代码块内的变量y=" + y);  
 }  
 }  
}  
/\*  
 \* 运行结果 普通代码块内的变量x=3  
 \* 主方法内的变量x=1  
 \* 普通代码块内的变量y=7  
 \*/

#### 构造代码块

直接在类中定义且没有加static关键字的代码块称为{}构造代码块。构造代码块在创建对象时被调用，每次创建对象都会被调用，并且构造代码块的执行次序优先于类构造函数。如果存在多个构造代码块，执行顺序由他们在代码中出现的次序决定，先出现先执行。

public class Test1 {  
 {  
 System.*out*.println("第一构造块");  
 }  
  
 public Test1(int i) {  
 System.*out*.println("第" + i + "次调用" + "构造方法");  
 }  
  
 {  
 System.*out*.println("第二构造块");  
 }  
  
 public static void main(String[] args) {  
 new Test1(0);  
 new Test1(1);  
 new Test1(2);  
  
 }  
}  
  
/\*  
 \* 执行结果 第一构造块  
 \* 第二构造块  
 \* 第0次调用构造方法  
 \* 第一构造块  
 \* 第二构造块  
 \* 第1次调用构造方法  
 \* 第一构造块  
 \* 第二构造块  
 \* 第2次调用构造方法  
 \*/

#### 静态代码块:

在java中使用static关键字声明的代码块。静态块用于初始化类，为类的属性初始化。每个静态代码块只会执行一次。由于JVM在加载类时会执行静态代码块，所以静态代码块先于主方法执行。  
如果类中包含多个静态代码块，那么将按照"先定义的代码先执行，后定义的代码后执行"。  
      注意：1 .静态代码块不能存在于任何方法体内。

2 .静态代码块不能直接访问实例变量和实例方法，需要通过类的实例对象来访问。

public class Test3 {  
 public static String *STATIC\_FIELD* = "静态属性";  
  
 // 静态块  
 static {  
 System.*out*.println(*STATIC\_FIELD*);  
 System.*out*.println("静态代码块1");  
 }  
  
 public String field = "非静态属性";  
  
 // 非静态块  
 {  
 System.*out*.println(field);  
 System.*out*.println("非静态代码块2");  
 }  
  
 public InitOderTest() {  
 System.*out*.println("无参构造函数");  
 }  
  
 public static void main(String[] args) {  
 InitOderTest test = new InitOderTest();  
 }  
  
 // 非静态块  
 {  
 System.*out*.println(field);  
 System.*out*.println("非静态代码块1");  
 }  
 // 静态块  
 static {  
 System.*out*.println(*STATIC\_FIELD*);  
 System.*out*.println("静态代码块2");  
 }  
  
}  
/\*  
 \* 运行结果 静态属性  
 \* 静态代码块1  
 \* 静态属性  
 \* 静态代码块2  
 \* 非静态属性  
 \* 非静态代码块2  
 \* 非静态属性  
 \* 非静态代码块1  
 \* 无参构造函数  
 \*/

--------------------- 本文来自 krisll 的CSDN 博客 ，全文地址请点击：https://blog.csdn.net/u012804721/article/details/52439311?utm\_source=copy

### Static

静态变量只能在类主体中定义，不能在方法中定义

成员方法又称为实例方法

静态方法又称为类方法

类方法是指类中被static修饰的方法，无this指针。

类方法是可以调用其他类的static方法的。

* 1. 方便在没有创建对象的情况下来进行调用（方法/变量）。
  2. 在静态方法中不能访问非静态成员方法和非静态成员变量，但是在非静态成员方法中是可以访问静态成员方法/变量的。
  3. 静态变量被所有的对象所共享，在内存中只有一个副本，它当且仅当在类初次加载时会被初始化。
  4. static块可以置于类中的任何地方，类中可以有多个static块。在类初次被加载的时候，会按照static块的顺序来执行每个static块，并且只会执行一次。
  5. 类属性中被static所引用的变量，会被作为GC的root根节点。作为根节点就意味着，这一类变量是基本上不会被回收的。因此，static很容易引入内存泄漏的风险。

### Final

1. **修饰类(class)。**     1、该类不能被继承。   
        2、类中的方法不会被覆盖，因此默认都是final的。   
        3、**用途：**设计类时，如果该类不需要有子类，不必要被扩展，类的实现细节不允许被改变，那么就设计成final类
2. **修饰方法(method)**   
        1、该方法可以被继承，但是不能被覆盖。   
        2、**用途：**一个类不允许子类覆盖该方法，则用final来修饰   
        3、**好处：**可以防止继承它的子类修改该方法的意义和实现；更为高效，编译器在遇到调用fianal方法转入内嵌机制，提高了执行效率。   
        4、**注意：**父类中的private成员方法不能被子类覆盖，因此，private方法默认是final型的(可以查看编译后的class文件)
3. **修饰变量(variable)**     1、用final修饰后变为常量。包括**静态变量、实例变量和局部变量**这三种。   
        2、**特点：**可以先声明，不给初值，这种叫做final空白。但是使用前必须被初始化。一旦被赋值，将不能再被改变。
4. **修饰参数(arguments)**     1、用final修饰参数时，可以读取该参数，但是不能对其作出修改

--------------------- 本文来自 ple婶 的CSDN 博客 ，全文地址请点击：https://blog.csdn.net/zhaotengfei36520/article/details/45098977?utm\_source=copy

1. **Java中被final修饰的内置类：**

public final class Byte

public final class Character

public static final class Character.UnicodeBlock

public final class Class<T>

public final class Compile

public final class Double

public final class Float

public final class Integer

public final class Long

public final class Math

public final class ProcessBuilder

public final class RuntimePermission

public final class Short

public final class StackTraceElement

public final class StrictMath

public final class String

public final class StringBuffer

public final class StringBuilder

public final class System

public final class Void

### Transient

1. 一旦变量被transient修饰，变量将不再是对象持久化的一部分，该变量内容在序列化后无法获得访问。
2. transient关键字只能修饰变量，而不能修饰方法和类。注意，本地变量是不能被transient关键字修饰的。变量如果是用户自定义类变量，则该类需要实现Serializable接口。
3. 被transient关键字修饰的变量不再能被序列化，一个静态变量不管是否被transient修饰，均不能被序列化。

### Volatile

一旦一个共享变量（类的成员变量、类的静态成员变量）被volatile修饰之后，那么就具备了两层语义：

* + 1. 保证了不同线程对这个变量进行操作时的可见性，即一个线程修改了某个变量的值，这新值对其他线程来说是立即可见的。
    2. 禁止进行指令重排序。
    3. 当程序执行到volatile变量的读操作或者写操作时，在其前面的操作的更改肯定全部已经进行，且结果已经对后面的操作可见；在其后面的操作肯定还没有进行；
    4. 在进行指令优化时，不能将在对volatile变量访问的语句放在其后面执行，也不能把volatile变量后面的语句放到其前面执行。

### 并发编程的三大特性

1. 原子性

在Java中，对基本数据类型的变量的读取和赋值操作是原子性操作，即这些操作是不可被中断的，要么执行，要么不执行。

1. 可见性

对于可见性，Java提供了volatile关键字来保证可见性。

当一个共享变量被volatile修饰时，它会保证修改的值会立即被更新到主存，当有其他线程需要读取时，它会去内存中读取新值。

1. 有序性

　　在Java内存模型中，允许编译器和处理器对指令进行重排序，但是重排序过程不会影响到单线程程序的执行，却会影响到多线程并发执行的正确性。

在Java里面，可以通过volatile关键字来保证一定的“有序性”（具体原理在下一节讲述）。另外可以通过synchronized和Lock来保证有序性.

### Interface与abstract类

**4点关于JAVA中接口存在的意义：**

       1、**重要性**：在Java语言中， abstract class 和interface 是支持抽象类定义的两种机制。正是由于这两种机制的存在，才赋予了Java强大的 面向对象能力。

　　2、**简单、规范性**：如果一个项目比较庞大，那么就需要一个能理清所有业务的架构师来定义一些主要的接口，这些接口不仅告诉开发人员你需要实现那些业务，而且也将命名规范限制住了（防止一些开发人员随便命名导致别的程序员无法看明白）。

　　3、**维护、拓展性**：比如你要做一个画板程序，其中里面有一个面板类，主要负责绘画功能，然后你就这样定义了这个类。

         可是在不久将来，你突然发现这个类满足不了你了，然后你又要重新设计这个类，更糟糕是你可能要放弃这个类，那么其他地方可能有引用他，这样修改起来很麻烦。

         如果你一开始定义一个接口，把绘制功能放在接口里，然后定义类时实现这个接口，然后你只要用这个接口去引用实现它的类就行了，以后要换的话只不过是引用另一个类而已，这样就达到维护、拓展的方便性。

　　4、**安全、严密性**：接口是实现软件松耦合的重要手段，它描叙了系统对外的所有服务，而不涉及任何具体的实现细节。这样就比较安全、严密一些（一般软件服务商考虑的比较多）。

1. 含有abstract修饰符的class即为抽象类，abstract 类不能创建类的实例对象。含有abstract方法的类必须定义为abstract class，abstract class类中的方法不必是抽象的, 但是有抽象方法的类必定是抽象类。abstract class类中定义抽象方法必须在具体(Concrete)子类中实现，所以，不能有抽象构造方法或抽象静态方法**。如果的子类没有实现抽象父类中的所有抽象方法，那么子类也必须定义为abstract类型。**
2. 接口（interface）可以说成是抽象类的一种特例，接口中的所有方法都必须是抽象、**default或者static方法**。接口中的方法定义默认为public abstract类型，接口中的成员变量类型默认为public static final。**一个实现接口的类，必须实现接口内所描述的所有方法，否则就必须声明为抽象类。**另外，在 Java 中，接口类型可用来声明一个变量，他们可以成为一个空指针，或是被绑定在一个以此接口实现的对象。
3. **下面比较一下两者的语法区别：**
4. 抽象类可以有构造方法，接口中不能有构造方法。
5. 抽象类中可以有普通成员变量，接口中没有普通成员变量
6. 抽象类中可以包含非抽象的普通方法，接口中的所有方法必须都是抽象的，不能有非抽象的普通方法。
7. 抽象类中的抽象方法的访问类型可以是public，protected和（默认类型,虽然eclipse下不报错，但应该也不行），但接口中的抽象方法只能是public类型的，并且默认即为public abstract类型。
8. 抽象类中可以包含静态方法，接口中不能包含静态方法
9. 抽象类和接口中都可以包含静态成员变量，抽象类中的静态成员变量的访问类型可以任意，但接口中定义的变量只能是public static final类型，并且默认即为public static final类型。
10. 一个类可以实现多个接口，但只能继承一个抽象类。

### 数据类型的默认值和长度

#### 数据类型分类

基本数据类型（或叫做原生类、内置类型）8种：

整数：byte，short，int，long（默认是int类型）

浮点类型： float，double（默认是double类型， 在定义 float 类型时必须在数字后面跟上 F 或者 f，long同理）

字符类型：char

布尔类型：boolean

引用数据类型3种：数组，类，接口

Java中的byte，short，char进行计算时都会提升为int类型。

#### 默认值

1. 布尔型(boolean)变量默认值为false，byte、short、int、long为0，字符型为'\u0000'(空字符)，浮点型(float double)为0.0,引用类型(String)为null。
2. 注意：未初始化的局部变量是不可以使用的，在这里可以认为是因为局部变量没有默认值，所以不可以直接使用。空字符('\u0000')什么也不输出，不要认为输出是空格。

#### 长度

32位的系统，则一个字是4个字节，如果是64位，则是8个字节。

1字节（byte） = 8位（bit）

基本类型：byte 二进制位数：8

包装类：java.lang.Byte

最小值：Byte.MIN\_VALUE=-128

最大值：Byte.MAX\_VALUE=127

基本类型：short 二进制位数：16

包装类：java.lang.Short

最小值：Short.MIN\_VALUE=-32768

最大值：Short.MAX\_VALUE=32767

基本类型：int 二进制位数：32

包装类：java.lang.Integer

最小值：Integer.MIN\_VALUE=-2147483648

最大值：Integer.MAX\_VALUE=2147483647

基本类型：long 二进制位数：64

包装类：java.lang.Long

最小值：Long.MIN\_VALUE=-9223372036854775808

最大值：Long.MAX\_VALUE=9223372036854775807

基本类型：float 二进制位数：32

包装类：java.lang.Float

最小值：Float.MIN\_VALUE=1.4E-45

最大值：Float.MAX\_VALUE=3.4028235E38

基本类型：double 二进制位数：64

包装类：java.lang.Double

最小值：Double.MIN\_VALUE=4.9E-324

最大值：Double.MAX\_VALUE=1.7976931348623157E308

基本类型：char 二进制位数：16

包装类：java.lang.Character

最小值：Character.MIN\_VALUE=0

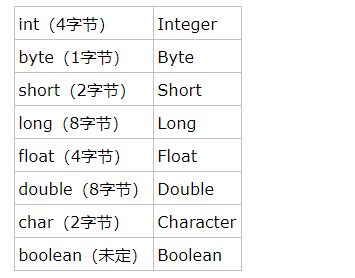
最大值：Character.MAX\_VALUE=65535

#### 不同数据类型运算转换

运算中，不同类型的数据先转化为同一类型，然后进行运算，转换规则如下：

|  |  |  |
| --- | --- | --- |
| **操作数 1 类型** | **操作数 2 类型** | **转换后的类型** |
| byte 、 short 、 char | int | int |
| byte 、 short 、 char 、 int | long | long |
| byte 、 short 、 char 、 int 、 long | float | float |
| byte 、 short 、 char 、 int 、 long 、 float | double | double |

#### 包装器类型

装箱过程是通过调用包装器的valueOf方法实现的，而拆箱过程是通过调用包装器的 xxxValue方法实现的。（xxx代表对应的基本数据类型）。

主动：Integer i = new Integer(10); //装箱

int i = a.intValue();//拆箱

自动：Integer i = 10; //装箱

int n = i; //拆箱

### [Java中创建对象的5种方式](https://www.cnblogs.com/wxd0108/p/5685817.html)

作为Java开发者，我们每天创建很多对象，但我们通常使用依赖管理系统，比如Spring去创建对象。然而这里有很多创建对象的方法，我们会在这篇文章中学到。

Java中有5种创建对象的方式，下面给出它们的例子还有它们的字节码

|  |  |
| --- | --- |
| 使用new关键字 | } → 调用了构造函数 |
| 使用Class类的newInstance方法 | } → 调用了构造函数 |
| 使用Constructor类的newInstance方法 | } → 调用了构造函数 |
| 使用clone方法 | } → 没有调用构造函数 |
| 使用反序列化 | } → 没有调用构造函数 |
| <https://www.cnblogs.com/wxd0108/p/5685817.html> |  |

### Java运算符优先级

当多个运算符出现在一个表达式中，谁先谁后呢？这就涉及到运算符的优先级别的问题。在一个多运算符的表达式中，运算符优先级不同会导致最后得出的结果差别甚大。

例如，（1+3）＋（3+2）\*2，这个表达式如果按加号最优先计算，答案就是 18，如果按照乘号最优先，答案则是 14。

再如，x = 7 + 3 \* 2;这里x得到13，而不是20，因为乘法运算符比加法运算符有较高的优先级，所以先计算3 \* 2得到6，然后再加7。

下表中具有最高优先级的运算符在的表的最上面，最低优先级的在表的底部。

|  |  |  |
| --- | --- | --- |
| **类别** | **操作符** | **关联性** |
| 后缀 | () [] . (点操作符) | 左到右 |
| 一元 | + + - ！〜 | 从右到左 |
| 乘性 | \* /％ | 左到右 |
| 加性 | + - | 左到右 |
| 移位 | >> >>>  << | 左到右 |
| 关系 | >> = << = | 左到右 |
| 相等 | ==  != | 左到右 |
| 按位与 | ＆ | 左到右 |
| 按位异或 | ^ | 左到右 |
| 按位或 | | | 左到右 |
| 逻辑与 | && | 左到右 |
| 逻辑或 | | | | 左到右 |
| 条件 | ？： | 从右到左 |
| 赋值 | = + = - = \* = / =％= >> = << =＆= ^ = | = | 从右到左 |
| 逗号 | ， | 左到右 |

### This、Super

this()和super()是你如果想用传入当前构造器中的参数或者构造器中的数据调用其他构造器或者控制父类构造器时使用的；

**在一个构造器中你只能使用this()或者super()之中的一个，而且调用的位置只能在构造器的第一行；**

在子类中如果你希望调用父类的构造器来初始化父类的部分，那就用合适的参数来调用super()，如果你用没有参数的super()来调用父类的构造器（同时也没有使用this()来调用其他构造器），父类缺省的构造器会被调用，如果父类没有缺省的构造器，那编译器就会报一个错误。

假如我们允许把this和super放置到任何位置。那么请看下面代码：

class A

{

A(){

System.out.println("You call super class non-args constructor!");

}

}

class B extends A

{

B(){

//这里，编译器将自动加上 super();

System.out.println("You call subclass constructor!");

}

B(String n){

super();

this();//实际就是调用了B(){...}，而在B(){...}中编译器自动加上了super();这样就相当于两次调用了super();

//也就是说对父类进行了两次初始化。而在实例化一个对象时，一个构造方法只能调用一次，这说明this和super不能同时存在一个构造方法中。

//同时因为系统没有在第一行发现this()或super()调用，就会自动加上super(),如果没有将this()和super()放在第一行就会产生矛盾。

//因为总有一个super()在第二句上。所以该程序不能通过编译！！！

}

也就是说你必须在构造器的第一行放置super或者this构造器，否则编译器会自动地放一个空参数的super构造器的，其他的构造器也可以调用super或者this，调用成一个递归构造链，最后的结果是父类的构造器（可能有多级父类构造器）始终在子类的构造器之前执行，递归的调用父类构造器。无法执行当前的类的构造器。也就不能实例化任何对象，这个类就成为一个无为类。

       从另外一面说，子类是从父类继承而来，继承了父类的属性和方法，如果在子类中先不完成父类的成员的初始化，则子类无法使用，应为在java中不允许调用没初始化的成员。在构造器中是顺序执行的，也就是说必须在第一行进行父类的初始化。而super能直接完成这个功能。This()通过调用本类中的其他构造器也能完成这个功能。

        因此，this()或者super()必须放在第一行。

  可以用在类方法和对象方法中。

    在对象方法中可以使用super关键字，调用当前对象从父类继承过来的对象方法。

在类方法中 super关键字可以调用当前类从父类继承过来的方法(类方法也能被子类继承，父类中的类方法可以使用父类名来调用，也可以使用子类名调用。再子类的类方法中，也可以用super来调用复用的类

原文：https://blog.csdn.net/qq\_36747738/article/details/71122150?utm\_source=copy

版权声明：本文为博主原创文章，转载请附上博文链接！

### 强引用，软引用，弱引用，虚引用？

⑴强引用（StrongReference）

强引用是使用最普遍的引用。如果一个对象具有强引用，那垃圾回收器绝不会回收它。当内存空间不足，Java虚拟机宁愿抛出OutOfMemoryError错误，使程序异常终止，也不会靠随意回收具有强引用的对象来解决内存不足的问题。  ps：强引用其实也就是我们平时A = new A()这个意思。

⑵软引用（SoftReference）

如果一个对象只具有软引用，则内存空间足够，垃圾回收器就不会回收它；如果内存空间不足了，就会回收这些对象的内存。只要垃圾回收器没有回收它，该对象就可以被程序使用。软引用可用来实现内存敏感的高速缓存（网页缓存，图片缓存）。

软引用可以和一个引用队列（ReferenceQueue）联合使用，如果软引用所引用的对象被垃圾回收器回收，Java虚拟机就会把这个软引用加入到与之关联的引用队列中。

SoftReference<String> sr = new SoftReference<String>(new String("hello"));

⑶弱引用（WeakReference）

弱引用与软引用的区别在于：只具有弱引用的对象拥有更短暂的生命周期。在垃圾回收器线程扫描它所管辖的内存区域的过程中，一旦发现了只具有弱引用的对象，不管当前内存空间足够与否，都会回收它的内存。不过，由于垃圾回收器是一个优先级很低的线程，因此不一定会很快发现那些只具有弱引用的对象。

弱引用可以和一个引用队列（ReferenceQueue）联合使用，如果弱引用所引用的对象被垃圾回收，Java虚拟机就会把这个弱引用加入到与之关联的引用队列中。

WeakReference<String> sr = new WeakReference<String>(new String("hello"));

⑷虚引用（PhantomReference）

“虚引用”顾名思义，就是形同虚设，与其他几种引用都不同，虚引用并不会决定对象的生命周期。如果一个对象仅持有虚引用，那么它就和没有任何引用一样，在任何时候都可能被垃圾回收器回收。

虚引用主要用来跟踪对象被垃圾回收器回收的活动。虚引用与软引用和弱引用的一个区别在于：虚引用必须和引用队列 （ReferenceQueue）联合使用。当垃圾回收器准备回收一个对象时，如果发现它还有虚引用，就会在回收对象的内存之前，把这个虚引用加入到与之 关联的引用队列中。

---------------------

作者：cyhaolihai

来源：CSDN

原文：https://blog.csdn.net/cyhaolihai/article/details/82902368?utm\_source=copy

版权声明：本文为博主原创文章，转载请附上博文链接！

### ****Java中final、finally和finalize的区别****

1. **final修饰符（关键字）。**

A. 被final修饰的类，不能被子类继承。【因此一个类不能既被abstract声明，又被final声明】

B. 被final修饰的方法，在使用的过程中不能被修改，只能使用，即不能方法重写

C. 被声明为final的变量必须在声明时给出变量的初始值，使用的过程中不能被修改，只能读取。

1. **finally**是在异常处理时提供finally块来执行任何清除操作。不管有没有异常被抛出、捕获，finally块都会被执行。try块中的内容是在无异常时执行到结束。catch块中的内容，是在try块内容发生catch所声明的异常时，跳转到catch块中执行。finally块则是无论异常是否发生，都会执行finally块的内容，所以在代码逻辑中有需要无论发生什么都必须执行的代码，就可以放在finally块中。
2. **finalize是方法名。**java技术允许使用finalize（）方法在垃圾收集器将对象从内存中清除出去之前做必要的清理工作。这个方法是由垃圾收集器在确定这个对象没有被引用时对这个对象调用的。它是在object类中定义的，因此所有的类都继承了它。子类覆盖finalize（）方法以整理系统资源或者被执行其他清理工作。finalize（）方法是在垃圾收集器删除对象之前对这个对象调用的。

---------------------

作者：cyhaolihai

来源：CSDN

原文：https://blog.csdn.net/cyhaolihai/article/details/82902368?utm\_source=copy

版权声明：本文为博主原创文章，转载请附上博文链接！

### 进程间通信

共享内存:将一块公共的物理内存映射到多个进程的私有地址空间。

socket通信

消息队列:对消息队列有写权限的进程可以向消息队列中添加新的消息，对消息队列有读权限的进程可以从消息队列中读取新的消息。

### 补码

1. 求**负整数的补码**，将其对应正数二进制表示所有位取反（包括符号位，0变1，1变0）后加1，-5对应正数5（00000101）→所有位取反（11111010）→加1(11111011)。

如果补码的符号位为“0”，表示是一个正数，其原码就是补码。

如果补码的符号位为“1”，表示是一个负数，那么求给定的这个补码的补码就是要求的原码。

小数补码求法，符号位保持1不变，数值位从右边数第一个1及其右边的0保持不变，左边按位取反。-0.0101为负数，补码为1.1011。

1. **补码加法**

[X+Y]补 = [X]补 + [Y]补

【例】X=+0110011,Y=-0101001，求[X+Y]补

[X]补=00110011 [Y]补=11010111

[X+Y]补 = [X]补 + [Y]补 = 00110011+11010111=00001010

注：因为计算机中运算器的位长是固定的（定长运算），上述运算中产生的最高位进位将丢掉，所以结果不是100001010，而是00001010。

1. **补码减法**

[X-Y]补 = [X]补 - [Y]补 = [X]补 + [-Y]补。

【例】1-1 [十进制]

1的原码00000001 转换成补码：00000001

-1的原码10000001 转换成补码：11111111

1+（-1)=0

00000001+11111111=00000000

00000000转换成十进制为0

0=0所以运算正确。

【例】-7-（-10) [十进制]

改为加法形式：-7-（-10）=-7+（-（-10））

-7的补码：11111001

-（-10）的补码：-10的原码为10001010，-（-10）的原码为00001010，

-（-10）的补码就是其原码，为00001010

-7 - （-10)= -7 + 10 = 3

11111001+00001010 = 00000011

转换成十进制为3

## 内部类

### 1.为什么使用内部类?

使用内部类最吸引人的原因是：每个内部类都能独立地继承一个（接口的）实现，所以无论外围类是否已经继承了某个（接口的）实现，

对于内部类都没有影响

1.1.使用内部类最大的优点就在于它能够非常好的解决多重继承的问题,使用内部类还能够为我们带来如下特性:

(1)、内部类可以用多个实例，每个实例都有自己的状态信息，并且与其他外围对象的信息相互独。

(2)、在单个外围类中，可以让多个内部类以不同的方式实现同一个接口，或者继承同一个类。

(3)、创建内部类对象的时刻并不依赖于外围类对象的创建。

(4)、内部类并没有令人迷惑的“is-a”关系，他就是一个独立的实体。

(5)、内部类提供了更好的封装，除了该外围类，其他类都不能访问。

### 2.内部类分类:

#### 成员内部类:

public class Outer{

private int age = 99;

String name = "Coco";

public class Inner{

String name = "Jayden";

public void show(){

System.out.println(Outer.this.name);

System.out.println(name);

System.out.println(age);

}

}

public Inner getInnerClass(){

return new Inner();

}

public static void main(String[] args){

Outer o = new Outer();

Inner in = o.new Inner();

in.show();

}

}

1.Inner 类定义在 Outer 类的内部，相当于 Outer 类的一个成员变量的位置，Inner 类可以使用任意访问控制符，

如 public 、 protected 、 private 等

2.Inner 类中定义的 show() 方法可以直接访问 Outer 类中的数据，而不受访问控制符的影响，

如直接访问 Outer 类中的私有属性age

3.定义了成员内部类后，必须使用外部类对象来创建内部类对象，而不能直接去 new 一个内部类对象，

即：内部类 对象名 = 外部类对象.new 内部类( );

4.编译上面的程序后，会发现产生了两个 .class 文件: Outer.class,Outer$Inner.class{}

5.成员内部类中不能存在任何 static 的变量和方法,可以定义常量:

(1).因为非静态内部类是要依赖于外部类的实例,而静态变量和方法是不依赖于对象的,仅与类相关,

简而言之:在加载静态域时,根本没有外部类,所在在非静态内部类中不能定义静态域或方法,编译不通过;

非静态内部类的作用域是实例级别

(2).常量是在编译器就确定的,放到所谓的常量池了

★★友情提示:

1.外部类是不能直接使用内部类的成员和方法的，可先创建内部类的对象，然后通过内部类的对象来访问其成员变量和方法;

2.如果外部类和内部类具有相同的成员变量或方法，内部类默认访问自己的成员变量或方法，如果要访问外部类的成员变量，

可以使用 this 关键字,如:Outer.this.name

#### 静态内部类: 是 static 修饰的内部类,

1.静态内部类不能直接访问外部类的非静态成员，但可以通过 new 外部类().成员 的方式访问

2.如果外部类的静态成员与内部类的成员名称相同，可通过“类名.静态成员”访问外部类的静态成员；

如果外部类的静态成员与内部类的成员名称不相同，则可通过“成员名”直接调用外部类的静态成员

3.创建静态内部类的对象时，不需要外部类的对象，可以直接创建 内部类 对象名 = new 内部类();

public class Outer{

private int age = 99;

static String name = "Coco";

public static class Inner{

String name = "Jayden";

public void show(){

System.out.println(Outer.name);

System.out.println(name);

}

}

public static void main(String[] args){

Inner i = new Inner();

i.show();

}

}

#### **方法内部类：**其作用域仅限于方法内，方法外部无法访问该内部类

(1).局部内部类就像是方法里面的一个局部变量一样，是不能有 public、protected、private 以及 static 修饰符的

(2).只能访问方法中定义的 final 类型的局部变量,因为:

当方法被调用运行完毕之后，局部变量就已消亡了。但内部类对象可能还存在,

直到没有被引用时才会消亡。此时就会出现一种情况，就是内部类要访问一个不存在的局部变量;

==>使用final修饰符不仅会保持对象的引用不会改变,而且编译器还会持续维护这个对象在回调方法中的生命周期.

局部内部类并不是直接调用方法传进来的参数，而是内部类将传进来的参数通过自己的构造器备份到了自己的内部，

自己内部的方法调用的实际是自己的属性而不是外部类方法的参数;

防止被篡改数据,而导致内部类得到的值不一致

/\*

使用的形参为何要为 final???

 在内部类中的属性和外部方法的参数两者从外表上看是同一个东西，但实际上却不是，所以他们两者是可以任意变化的，

 也就是说在内部类中我对属性的改变并不会影响到外部的形参，而然这从程序员的角度来看这是不可行的，

 毕竟站在程序的角度来看这两个根本就是同一个，如果内部类该变了，而外部方法的形参却没有改变这是难以理解

 和不可接受的，所以为了保持参数的一致性，就规定使用 final 来避免形参的不改变

 \*/

public class Outer{

public void Show(){

final int a = 25;

int b = 13;

class Inner{

int c = 2;

public void print(){

System.out.println("访问外部类:" + a);

System.out.println("访问内部类:" + c);

}

}

Inner i = new Inner();

i.print();

}

public static void main(String[] args){

Outer o = new Outer();

o.show();

}

}

(3).注意:在JDK8版本之中,方法内部类中调用方法中的局部变量,可以不需要修饰为 final,匿名内部类也是一样的，主要是JDK8之后增加了 Effectively final 功能

<http://docs.oracle.com/javase/tutorial/java/javaOO/localclasses.html>

反编译jdk8编译之后的class文件,发现内部类引用外部的局部变量都是 final 修饰的

#### 匿名内部类:

(1).匿名内部类是直接使用 new 来生成一个对象的引用;

(2).对于匿名内部类的使用它是存在一个缺陷的，就是它仅能被使用一次，创建匿名内部类时它会立即创建一个该类的实例，  
该类的定义会立即消失，所以匿名内部类是不能够被重复使用;  
(3).使用匿名内部类时，我们必须是继承一个类或者实现一个接口，但是两者不可兼得，同时也只能继承一个类或者实现一个接口;  
(4).匿名内部类中是不能定义构造函数的,匿名内部类中不能存在任何的静态成员变量和静态方法;  
(5).匿名内部类中不能存在任何的静态成员变量和静态方法,匿名内部类不能是抽象的,它必须要实现继承的类或者实现的接口的所有抽象方法  
(6).匿名内部类初始化:使用构造代码块！利用构造代码块能够达到为匿名内部类创建一个构造器的效果

public class OuterClass {

  public InnerClass getInnerClass(final int   num,String str2){

                return new InnerClass(){

                    int number = num + 3;

                    public int getNumber(){

                        return number;

                    }

                };        /\* 注意：分号不能省 \*/

            }

            public static void main(String[] args) {

                OuterClass out = new OuterClass();

            InnerClass inner = out.getInnerClass(2, "chenssy");

                System.out.println(inner.getNumber());

            }

        }

        interface InnerClass {

            int getNumber();

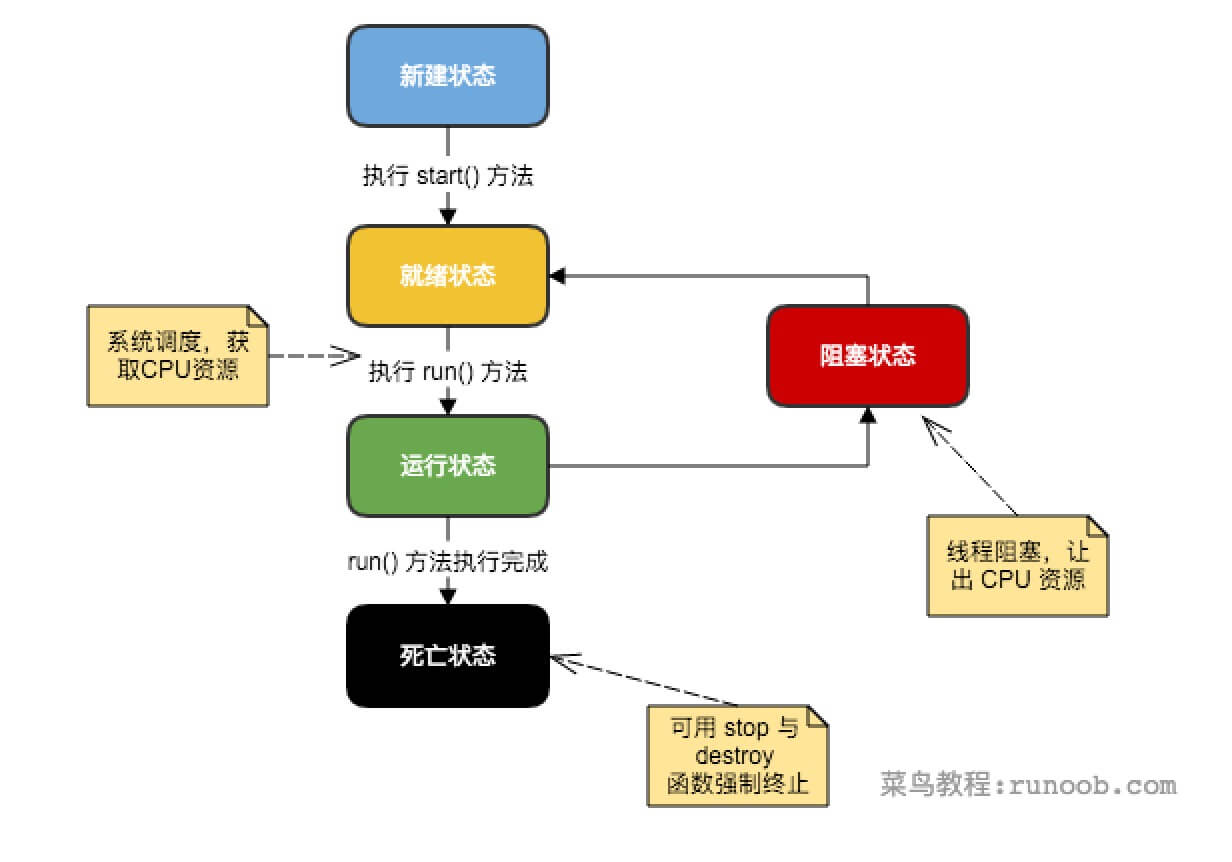
        }

## 多线程

### 状态

线程是一个动态执行的过程，它也有一个从产生到死亡的过程。

下图显示了一个线程完整的生命周期。



* **新建状态:**

使用 **new** 关键字和 **Thread** 类或其子类建立一个线程对象后，该线程对象就处于新建状态。它保持这个状态直到程序 **start()** 这个线程。

* **就绪状态:**

当线程对象调用了start()方法之后，该线程就进入就绪状态。就绪状态的线程处于就绪队列中，要等待JVM里线程调度器的调度。

* **运行状态:**

如果就绪状态的线程获取 CPU 资源，就可以执行 **run()**，此时线程便处于运行状态。处于运行状态的线程最为复杂，它可以变为阻塞状态、就绪状态和死亡状态。

* **阻塞状态:**

如果一个线程执行了sleep（睡眠）、suspend（挂起）等方法，失去所占用资源之后，该线程就从运行状态进入阻塞状态。在睡眠时间已到或获得设备资源后可以重新进入就绪状态。可以分为三种：

* + 等待阻塞：运行状态中的线程执行 wait() 方法，使线程进入到等待阻塞状态。
  + 同步阻塞：线程在获取 synchronized 同步锁失败(因为同步锁被其他线程占用)。
  + 其他阻塞：通过调用线程的 sleep() 或 join() 发出了 I/O 请求时，线程就会进入到阻塞状态。当sleep() 状态超时，join() 等待线程终止或超时，或者 I/O 处理完毕，线程重新转入就绪状态。
* **死亡状态:**

一个运行状态的线程完成任务或者其他终止条件发生时，该线程就切换到终止状态。

### 创建线程的方法

**Java 提供了三种创建线程的方法：**

#### 通过实现 Runnable 接口

创建一个线程，最简单的方法是创建一个实现 Runnable 接口的类。

为了实现 Runnable，一个类只需要执行一个方法调用 run()，声明如下：

public void run()

你可以重写该方法，重要的是理解的 run() 可以调用其他方法，使用其他类，并声明变量，就像主线程一样。

在创建一个实现 Runnable 接口的类之后，你可以在类中实例化一个线程对象。

Thread 定义了几个构造方法，下面的这个是我们经常使用的：

Thread(Runnable threadOb,String threadName);

新线程创建之后，你调用它的 start() 方法它才会运行。

void start();

class TestForRunnable implements Runnable {  
 private int k;  
 //多线程,implements Runnable  
  
 */\*\*  
 \*   
 \*/* public void run() {  
 for (; k < 100; k++) {  
 System.*out*.println(Thread.*currentThread*().getName()  
 + " " + k);  
 }  
 }  
 */\*\*  
 \*   
 \*/* public void test() {  
 //Runnable  
 for (int i = 0; i < 100; i++) {  
 System.*out*.println(Thread.*currentThread*().getName() + " " + i);  
 if (20 == i) {  
 try {  
 //共享k  
 TestForRunnable a = new TestForRunnable();  
// new Thread(a, "新线程1").start();  
// new Thread(a, "新线程2").start();  
 Thread t1 = new Thread(a, "新线程1");  
 Thread t2 = new Thread(a, "新线程2");  
 t2.start();  
 t1.start();  
 t1.join();  
 t2.*sleep*(1);  
 } catch (Exception e) {  
 e.printStackTrace();  
 }  
 }  
 }  
 }  
}

#### 通过继承 Thread 类本身

创建一个线程的第二种方法是创建一个新的类，该类继承 Thread 类，然后创建一个该类的实例。

继承类必须重写 run() 方法，该方法是新线程的入口点。它也必须调用 start() 方法才能执行。

该方法尽管被列为一种多线程实现方式，但是本质上也是实现了 Runnable 接口的一个实例。

class TestForThread extends Thread {  
 private int k;  
 */\*\*  
 \*  
 \** ***@param*** *name  
 \*/* public TestForThread(String name) {  
 super(name);  
 }  
  
 */\*\*  
 \*  
 \*/* public void run() {  
 try {  
 for (; k < 100; k++) {  
 System.*out*.println(getName() + " " + k);  
 //有sleep()抛出InterruptedException  
 *sleep*(10);  
 if (20 == k) {  
 //转入就绪状态，给同级或者更高等级线程机会  
 Thread.*yield*();  
 }  
 }  
  
 }catch (InterruptedException e) {  
 e.printStackTrace();  
 }  
 }

**下表列出了Thread类的一些重要方法：**

|  |  |
| --- | --- |
| **序号** | **方法描述** |
| 1 | **public void start()** 使该线程开始执行；**Java** 虚拟机调用该线程的 run 方法。 |
| 2 | **public void run()** 如果该线程是使用独立的 Runnable 运行对象构造的，则调用该 Runnable 对象的 run 方法；否则，该方法不执行任何操作并返回。 |
| 3 | **public final void setName(String name)** 改变线程名称，使之与参数 name 相同。 |
| 4 | **public final void setPriority(int priority)**  更改线程的优先级。 |
| 5 | **public final void setDaemon(boolean on)** 将该线程标记为守护线程或用户线程。 |
| 6 | **public final void join(long millisec)** 等待该线程终止的时间最长为 millis 毫秒。 |
| 7 | **public void interrupt()** 中断线程。 |
| 8 | **public final boolean isAlive()** 测试线程是否处于活动状态。 |

**测试线程是否处于活动状态。 上述方法是被Thread对象调用的。下面的方法是Thread类的静态方法。**

|  |  |
| --- | --- |
| **序号** | **方法描述** |
| 1 | **public static void yield()** 暂停当前正在执行的线程对象，并执行其他线程。 |
| 2 | **public static void sleep(long millisec)** 在指定的毫秒数内让当前正在执行的线程休眠（暂停执行），此操作受到系统计时器和调度程序精度和准确性的影响。 |
| 3 | **public static boolean holdsLock(Object x)** 当且仅当当前线程在指定的对象上保持监视器锁时，才返回 true。 |
| 4 | **public static Thread currentThread()** 返回对当前正在执行的线程对象的引用。 |
| 5 | **public static void dumpStack()** 将当前线程的堆栈跟踪打印至标准错误流。 |

#### 通过 Callable 和 Future 创建线程。

* 1. 创建 Callable 接口的实现类，并实现 call() 方法，该 call() 方法将作为线程执行体，并且有返回值。
* 2. 创建 Callable 实现类的实例，使用 FutureTask 类来包装 Callable 对象，该 FutureTask 对象封装了该 Callable 对象的 call() 方法的返回值。
* 3. 使用 FutureTask 对象作为 Thread 对象的 target 创建并启动新线程。
* 4. 调用 FutureTask 对象的 get() 方法来获得子线程执行结束后的返回值。
* class CallableAndFutureTask {  
   */\*\*  
   \*   
   \*/* public void test() {  
   //多线程Callable，FutureTask  
   FutureTask<Integer> task = new FutureTask<Integer>((Callable<Integer>)()-> {  
   //call()  
   int i = 0;  
   for (; i < 100; i++) {  
   System.*out*.println(Thread.*currentThread*().getName() + "的循环变量i的值：" + i);  
   }  
   return i;  
   });  
   for (int i = 0; i<100; i++) {  
   System.*out*.println(Thread.*currentThread*().getName() + "的循环变量i的值：" + i);  
   if (20 == i) {  
   new Thread(task,"有返回值的线程").start();  
   }  
   }  
   try {  
   System.*out*.println("子线程的返回值：" + task.get());  
   } catch (Exception e) {  
   e.printStackTrace();  
   }  
   }  
  }

#### 创建线程的三种方式的对比

* 1. 采用实现 Runnable、Callable 接口的方式创建多线程时，线程类只是实现了 Runnable 接口或 Callable 接口，还可以继承其他类。
* 2. 使用继承 Thread 类的方式创建多线程时，编写简单，如果需要访问当前线程，则无需使用 Thread.currentThread() 方法，直接使用 this 即可获得当前线程。
* Runnable的call()方法有返回值。

#### 线程池

##### 线程池，其实就是一个容纳多个线程的容器，其中的线程可以反复使用，省去了频繁创建线程对象的操作，无需反复创建线程而消耗过多资源。（是什么）

##### 那么，我们为什么需要用到线程池呢？每次用的时候手动创建不行吗？

在java中，如果每个请求到达就创建一个新线程，开销是相当大的。在实际使用中，创建和销毁线程花费的时间和消耗的系统资源都相当大，甚至可能要比在处理实际的用户请求的时间和资源要多的多。除了创建和销毁线程的开销之外，活动的线程也需要消耗系统资源。如果在一个jvm里创建太多的线程，可能会使系统由于过度消耗内存或“切换过度”而导致系统资源不足。为了防止资源不足，需要采取一些办法来限制任何给定时刻处理的请求数目，尽可能减少创建和销毁线程的次数，特别是一些资源耗费比较大的线程的创建和销毁，尽量利用已有对象来进行服务。（为什么）

线程池主要用来解决线程生命周期开销问题和资源不足问题。通过对多个任务重复使用线程，线程创建的开销就被分摊到了多个任务上了，而且由于在请求到达时线程已经存在，所以消除了线程创建所带来的延迟。这样，就可以立即为请求服务，使用应用程序响应更快；另外，通过适当的调整线程中的线程数目可以防止出现资源不足的情况。（什么用）

##### 线程池都是通过线程池工厂创建，再调用线程池中的方法获取线程，再通过线程去执行任务方法。

* **Executors**：线程池创建工厂类
* **public static ExecutorServicenewFixedThreadPool(int nThreads)**：返回线程池对象
* **ExecutorService**：线程池类
* **Future<?> submit(Runnable task)**：获取线程池中的某一个线程对象，并执行
* **Future 接口**：用来记录线程任务执行完毕后产生的结果。线程池创建与使用

##### 这里介绍两种使用线程池创建线程的方法

###### **使用Runnable接口创建线程池**

使用线程池中线程对象的步骤：

* 1、创建线程池对象
* 2、创建 Runnable 接口子类对象
* 3、提交 Runnable 接口子类对象
* 4、关闭线程池

**Test.java** 代码如下：

import java.util.concurrent.ExecutorService;

import java.util.concurrent.Executors;

public class Test {

public static void main(String[] args) {

//创建线程池对象 参数5，代表有5个线程的线程池

ExecutorService service = Executors.newFixedThreadPool(5);

//创建Runnable线程任务对象

TaskRunnable task = new TaskRunnable();

//从线程池中获取线程对象

service.submit(task);

System.out.println("----------------------");

//再获取一个线程对象

service.submit(task);

//关闭线程池

service.shutdown();

}

}

TaskRunnable.java 接口文件如下：

public class TaskRunnable implements Runnable{

@Override

public void run() {

for (int i = 0; i < 1000; i++) {

System.out.println("自定义线程任务在执行"+i);

}

}

}

###### 使用Callable接口创建线程池

Callable接口：与Runnable接口功能相似，用来指定线程的任务。其中的call()方法，用来返回线程任务执行完毕后的结果，call方法可抛出异常。

**ExecutorService**：线程池类

**<T> Future<T> submit(Callable<T> task)**：获取线程池中的某一个线程对象，并执行线程中的 call() 方法

**Future 接口**：用来记录线程任务执行完毕后产生的结果。线程池创建与使用

使用线程池中线程对象的步骤：

* 1、创建线程池对象
* 2、创建 Callable 接口子类对象
* 3、提交 Callable 接口子类对象
* 4、关闭线程池

Test.java 代码如下：

import java.util.concurrent.ExecutorService;

import java.util.concurrent.Executors;

public class Test{

public static void main(String[] args) {

ExecutorService service = Executors.newFixedThreadPool(3);

TaskCallable c = new TaskCallable();

//线程池中获取线程对象，调用run方法

service.submit(c);

//再获取一个

service.submit(c);

//关闭线程池

service.shutdown();

}

}

TaskCallable.java 接口文件如下：

import java.util.concurrent.Callable;

public class TaskCallable implements Callable<Object>{

@Override

public Object call() throws Exception {

for (int i = 0; i < 1000; i++) {

System.out.println("自定义线程任务在执行"+i);

}

return null;

}

}

### 线程间的通信方式

##### synchronized**同步**

这里讲的同步是指多个线程通过synchronized关键字这种方式来实现线程间的通信。

参考示例：

public class MyObject {

synchronized public void methodA() {

//do something....

}

synchronized public void methodB() {

//do some other thing

}

}

public class ThreadA extends Thread {

private MyObject object;

//省略构造方法

@Override

public void run() {

super.run();

object.methodA();

}

}

public class ThreadB extends Thread {

private MyObject object;

//省略构造方法

@Override

public void run() {

super.run();

object.methodB();

}

}

public class Run {

public static void main(String[] args) {

MyObject object = new MyObject();

//线程A与线程B 持有的是同一个对象:object

ThreadA a = new ThreadA(object);

ThreadB b = new ThreadB(object);

a.start();

b.start();

}

}

由于线程A和线程B持有同一个MyObject类的对象object，尽管这两个线程需要调用不同的方法，但是它们是同步执行的，比如：**线程B需要等待线程A执行完了methodA()方法之后，它才能执行methodB()方法。这样，线程A和线程B就实现了 通信。**

**这种方式，本质上就是“共享内存”式的通信。多个线程需要访问同一个共享变量，谁拿到了锁（获得了访问权限），谁就可以执行。**

##### **while轮询的方式**

代码如下：

import java.util.ArrayList;

import java.util.List;

public class MyList {

private List<String> list = new ArrayList<String>();

public void add() {

list.add("elements");

}

public int size() {

return list.size();

}

}

import mylist.MyList;

public class ThreadA extends Thread {

private MyList list;

public ThreadA(MyList list) {

super();

this.list = list;

}

@Override

public void run() {

try {

for (int i = 0; i < 10; i++) {

list.add();

System.out.println("添加了" + (i + 1) + "个元素");

Thread.sleep(1000);

}

} catch (InterruptedException e) {

e.printStackTrace();

}

}

}

import mylist.MyList;

public class ThreadB extends Thread {

private MyList list;

public ThreadB(MyList list) {

super();

this.list = list;

}

@Override

public void run() {

try {

while (true) {

if (list.size() == 5) {

System.out.println("==5, 线程b准备退出了");

throw new InterruptedException();

}

}

} catch (InterruptedException e) {

e.printStackTrace();

}

}

}

import mylist.MyList;

import extthread.ThreadA;

import extthread.ThreadB;

public class Test {

public static void main(String[] args) {

MyList service = new MyList();

ThreadA a = new ThreadA(service);

a.setName("A");

a.start();

ThreadB b = new ThreadB(service);

b.setName("B");

b.start();

}

}

在这种方式下，线程A不断地改变条件，线程ThreadB不停地通过while语句检测这个条件(list.size()==5)是否成立 ，从而实现了线程间的通信。但是**这种方式会浪费CPU资源**。之所以说它浪费资源，是因为JVM调度器将CPU交给线程B执行时，它没做啥“有用”的工作，只是在不断地测试 某个条件是否成立。就类似于现实生活中，某个人一直看着手机屏幕是否有电话来了，而不是： 在干别的事情，当有电话来时，响铃通知TA电话来了。关于线程的轮询的影响，[可参考：](http://www.cnblogs.com/hapjin/p/5467984.html)[JAVA多线程之当一个线程在执行死循环时会影响另外一个线程吗？](http://www.cnblogs.com/hapjin/p/5467984.html)

这种方式还存在另外一个问题：

轮询的条件的可见性问题，关于内存可见性问题，可参考：[JAVA多线程之volatile 与 synchronized 的比较](http://www.cnblogs.com/hapjin/p/5492880.html)中的第一点“**一，volatile关键字的可见性**”

线程都是先把变量读取到本地线程栈空间，然后再去再去修改的本地变量。因此，如果线程B每次都在取本地的 条件变量，那么尽管另外一个线程已经改变了轮询的条件，它也察觉不到，这样也会造成死循环。

##### **wait/notify机制**

代码如下：

import java.util.ArrayList;

import java.util.List;

public class MyList {

private static List<String> list = new ArrayList<String>();

public static void add() {

list.add("anyString");

}

public static int size() {

return list.size();

}

}

public class ThreadA extends Thread {

private Object lock;

public ThreadA(Object lock) {

super();

this.lock = lock;

}

@Override

public void run() {

try {

synchronized (lock) {

if (MyList.size() != 5) {

System.out.println("wait begin "

+ System.currentTimeMillis());

lock.wait();

System.out.println("wait end "

+ System.currentTimeMillis());

}

}

} catch (InterruptedException e) {

e.printStackTrace();

}

}

}

public class ThreadB extends Thread {

private Object lock;

public ThreadB(Object lock) {

super();

this.lock = lock;

}

@Override

public void run() {

try {

synchronized (lock) {

for (int i = 0; i < 10; i++) {

MyList.add();

if (MyList.size() == 5) {

lock.notify();

System.out.println("已经发出了通知");

}

System.out.println("添加了" + (i + 1) + "个元素!");

Thread.sleep(1000);

}

}

} catch (InterruptedException e) {

e.printStackTrace();

}

}

}

public class Run {

public static void main(String[] args) {

try {

Object lock = new Object();

ThreadA a = new ThreadA(lock);

a.start();

Thread.sleep(50);

ThreadB b = new ThreadB(lock);

b.start();

} catch (InterruptedException e) {

e.printStackTrace();

}

}

}

线程A要等待某个条件满足时(list.size()==5)，才执行操作。线程B则向list中添加元素，改变list 的size。

A,B之间如何通信的呢？也就是说，线程A如何知道 list.size() 已经为5了呢？

这里用到了Object类的 wait() 和 notify() 方法。

当条件未满足时(list.size() !=5)，线程A调用wait() 放弃CPU，并进入阻塞状态。---不像②while轮询那样占用CPU

当条件满足时，线程B调用 notify()通知 线程A，所谓通知线程A，就是唤醒线程A，并让它进入可运行状态。

这种方式的一个好处就是CPU的利用率提高了。

但是也有一些缺点：比如，线程B先执行，一下子添加了5个元素并调用了notify()发送了通知，而此时线程A还执行；当线程A执行并调用wait()时，那它永远就不可能被唤醒了。因为，线程B已经发了通知了，以后不再发通知了。这说明：**通知过早，会打乱程序的执行逻辑。**

##### **管道通信**

就是使用java.io.PipedInputStream 和 java.io.PipedOutputStream进行通信

具体就不介绍了。分布式系统中说的两种通信机制：共享内存机制和消息通信机制。感觉前面的①中的synchronized关键字和②中的while轮询 “属于” 共享内存机制，由于是轮询的条件使用了volatile关键字修饰时，这就表示它们通过判断这个“共享的条件变量“是否改变了，来实现进程间的交流。

而管道通信，更像消息传递机制，也就是说：通过管道，将一个线程中的消息发送给另一个。

建立管道输入端和输出端的连接

首先为了创建一个管道流，我们必须首先创建一个PipedOutputStream对象，然后创建一个PipedInputStream对象。如下：

PipedOutputStream out = null;

PipedInputStream in = null;

对象建立好以后使用connect()方法将二者建立连接

out.connect(in);

该方法在PipedOutputStream 、PipedInputStream当中都有，随便调用那个来建立连接都可以，但注意智能建立一次连接，重复建立会抛异常。

不使用connect()方法也是可以建立连接的，方法如下：

out = new PipedOutputStream(in );

一旦建立了管道，就可以像操作文件一样对管道进行数据读写。

开始通信

首先有一点特别注意，不能在同一个线程当中既写入又读取，这样会造成死锁，因为管道会有阻塞的时候(当管道当中没有数据，进行读操作时，读操作的线程会阻塞，直到有线程来写数据；当管道当中满数据，进行写操作时，写操作的线程阻塞，直到有线程来读数据)，有时需要写和读的两端同时都在工作，只有一个线程去完成读和写，显然无法保证能够同时读写，所以读写最好放在单独的线程去完成。

建立的管道是一个包含1024字节大小的循环缓冲数组，从管道当中读取的数据，会被清除出管道，即是读取以后就相当于把该数据从管道当中拿走了，所以是循环缓冲数组。

下面演示代码：

package cn.zhoucy.pipe;

import java.io.IOException;

import java.io.PipedInputStream;

import java.io.PipedOutputStream;

public class TestPiped {

public static void main(String[] args) {

Sender sender = new Sender();

Recive recive = new Recive();

PipedInputStream pi = recive.getPipedInputputStream();

PipedOutputStream po = sender.getPipedOutputStream();

try {

pi.connect(po);

} catch (IOException e) {

System.out.println(e.getMessage());

}

new Thread(sender).start();

new Thread(recive).start();

}

}

class Sender implements Runnable {

PipedOutputStream out = null;

public PipedOutputStream getPipedOutputStream() {

out = new PipedOutputStream();

return out;

}

@Override

public void run() {

try {

out.write("Hello , Reciver!".getBytes());

} catch (IOException e) {

System.out.println(e.getMessage());

}

try {

out.close();

} catch (IOException e) {

System.out.println(e.getMessage());

}

}

}

class Recive implements Runnable {

PipedInputStream in = null;

public PipedInputStream getPipedInputputStream() {

in = new PipedInputStream();

return in;

}

@Override

public void run() {

byte[] bys = new byte[1024];

try {

in.read(bys);

System.out.println("读取到的信息：" + new String(bys).trim());

in.close();

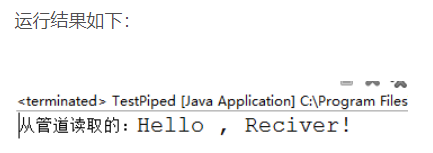
} catch (IOException e) {

System.out.println(e.getMessage());

}

}

}



---------------------

作者：无扬人生

来源：CSDN

原文：https://blog.csdn.net/theblackbeard/article/details/54880180

版权声明：本文为博主原创文章，转载请附上博文链接！

### 产生死锁的四个必要条件

(1)   互斥条件：使用的资源是不能共享的。

（2） 请求与保持条件(hold and wait)：线程持有一个资源并等待获取一个被其他线程持有的资源。

（3） 不剥夺条件:线程正在持有的资源不能被其他线程强行剥夺。

（4） 循环等待条件:线程之间形成一种首尾相连的等待资源的关系。

---------------------

作者：cyhaolihai

来源：CSDN

原文：https://blog.csdn.net/cyhaolihai/article/details/82902368?utm\_source=copy

版权声明：本文为博主原创文章，转载请附上博文链接！

**如何解决死锁？**

有三个方法:

1.进程在申请资源时，一次性得请求他所需要的所有资源。若无法满足则不能执行。

2.进程在申请新的资源时，释放已占有的资源。后面若还需要它们，则需要重新申请。

3.将系统中的资源顺序编号，规定进程只能依次申请资源。

### 线程的优先级

每一个 Java 线程都有一个优先级，这样有助于操作系统确定线程的调度顺序。

Java 线程的优先级是一个整数，其取值范围是 1 （Thread.MIN\_PRIORITY ） - 10 （Thread.MAX\_PRIORITY ）。

默认情况下，每一个线程都会分配一个优先级 NORM\_PRIORITY（5）。

具有较高优先级的线程对程序更重要，并且应该在低优先级的线程之前分配处理器资源。但是，线程优先级不能保证线程执行的顺序，而且非常依赖于平台。

### JAVA多线程和并发基础面试问答

https://www.cnblogs.com/dolphin0520/p/3932934.html

#### 进程和线程之间有什么不同？

一个进程是一个独立(self contained)的运行环境，它可以被看作一个程序或者一个应用。而线程是在进程中执行的一个任务。Java运行环境是一个包含了不同的类和程序的单一进程。线程可以被称为轻量级进程。线程需要较少的资源来创建和驻留在进程中，并且可以共享进程中的资源。

#### 多线程编程的好处是什么？

在多线程程序中，多个线程被并发的执行以提高程序的效率，CPU不会因为某个线程需要等待资源而进入空闲状态。多个线程共享堆内存(heap memory)，因此创建多个线程去执行一些任务会比创建多个进程更好。举个例子，Servlets比CGI更好，是因为Servlets支持多线程而CGI不支持。

#### 用户线程和守护线程有什么区别？

当我们在Java程序中创建一个线程，它就被称为用户线程。一个守护线程是在后台执行并且不会阻止JVM终止的线程。当没有用户线程在运行的时候，JVM关闭程序并且退出。一个守护线程创建的子线程依然是守护线程。

#### 我们如何创建一个线程？

有两种创建线程的方法：一是实现Runnable接口，然后将它传递给Thread的构造函数，创建一个Thread对象；二是直接继承Thread类。若想了解更多可以阅读这篇关于如何在[Java中创建线程](http://www.journaldev.com/1016/java-thread-example-extending-thread-class-and-implementing-runnable-interface)的文章。

#### 有哪些不同的线程生命周期？

当我们在Java程序中新建一个线程时，它的状态是New。当我们调用线程的start()方法时，状态被改变为Runnable。线程调度器会为Runnable线程池中的线程分配CPU时间并且讲它们的状态改变为Running。其他的线程状态还有Waiting，Blocked 和Dead。读这篇文章可以了解更多关于[线程生命周期](http://www.journaldev.com/1044/life-cycle-of-thread-understanding-thread-states-in-java)的知识。

#### 可以直接调用Thread类的run()方法么？

当然可以，但是如果我们调用了Thread的run()方法，它的行为就会和普通的方法一样，为了在新的线程中执行我们的代码，必须使用Thread.start()方法。

#### 如何让正在运行的线程暂停一段时间？

我们可以使用Thread类的Sleep()方法让线程暂停一段时间。需要注意的是，这并不会让线程终止，一旦从休眠中唤醒线程，线程的状态将会被改变为Runnable，并且根据线程调度，它将得到执行。

#### 你对线程优先级的理解是什么？

每一个线程都是有优先级的，一般来说，高优先级的线程在运行时会具有优先权，但这依赖于线程调度的实现，这个实现是和操作系统相关的(OS dependent)。我们可以定义线程的优先级，但是这并不能保证高优先级的线程会在低优先级的线程前执行。线程优先级是一个int变量(从1-10)，1代表最低优先级，10代表最高优先级。

#### 什么是线程调度器(Thread Scheduler)和时间分片(Time Slicing)？

线程调度器是一个操作系统服务，它负责为Runnable状态的线程分配CPU时间。一旦我们创建一个线程并启动它，它的执行便依赖于线程调度器的实现。时间分片是指将可用的CPU时间分配给可用的Runnable线程的过程。分配CPU时间可以基于线程优先级或者线程等待的时间。线程调度并不受到Java虚拟机控制，所以由应用程序来控制它是更好的选择（也就是说不要让你的程序依赖于线程的优先级）。

#### 在多线程中，什么是上下文切换(context-switching)？

上下文切换是存储和恢复CPU状态的过程，它使得线程执行能够从中断点恢复执行。上下文切换是多任务操作系统和多线程环境的基本特征。

#### 你如何确保main()方法所在的线程是Java程序最后结束的线程？

我们可以使用Thread类的joint()方法来确保所有程序创建的线程在main()方法退出前结束。这里有一篇文章关于[Thread类的joint()方法](http://www.journaldev.com/1024/java-thread-join-example-with-explanation)。

#### 线程之间是如何通信的？

当线程间是可以共享资源时，线程间通信是协调它们的重要的手段。Object类中wait()\notify()\notifyAll()方法可以用于线程间通信关于资源的锁的状态。点击[这里](http://www.journaldev.com/1037/java-thread-wait-notify-and-notifyall-example)有更多关于线程wait, notify和notifyAll.

#### 为什么线程通信的方法wait(), notify()和notifyAll()被定义在Object类里？

Java的每个对象中都有一个锁(monitor，也可以成为监视器) 并且wait()，notify()等方法用于等待对象的锁或者通知其他线程对象的监视器可用。在Java的线程中并没有可供任何对象使用的锁和同步器。这就是为什么这些方法是Object类的一部分，这样Java的每一个类都有用于线程间通信的基本方法

#### 为什么wait(), notify()和notifyAll()必须在同步方法或者同步块中被调用？

当一个线程需要调用对象的wait()方法的时候，这个线程必须拥有该对象的锁，接着它就会释放这个对象锁并进入等待状态直到其他线程调用这个对象上的notify()方法。同样的，当一个线程需要调用对象的notify()方法时，它会释放这个对象的锁，以便其他在等待的线程就可以得到这个对象锁。由于所有的这些方法都需要线程持有对象的锁，这样就只能通过同步来实现，所以他们只能在同步方法或者同步块中被调用。

#### 为什么Thread类的sleep()和yield()方法是静态的？

Thread类的sleep()和yield()方法将在当前正在执行的线程上运行。所以在其他处于等待状态的线程上调用这些方法是没有意义的。这就是为什么这些方法是静态的。它们可以在当前正在执行的线程中工作，并避免程序员错误的认为可以在其他非运行线程调用这些方法。

#### 如何确保线程安全？

在Java中可以有很多方法来保证线程安全——同步，使用原子类(atomic concurrent classes)，实现并发锁，使用volatile关键字，使用不变类和线程安全类。在[线程安全教程](http://www.journaldev.com/1061/java-synchronization-and-thread-safety-tutorial-with-examples)中，你可以学到更多。

#### volatile关键字在Java中有什么作用？

当我们使用volatile关键字去修饰变量的时候，所以线程都会直接读取该变量并且不缓存它。这就确保了线程读取到的变量是同内存中是一致的。

#### 同步方法和同步块，哪个是更好的选择？

同步块是更好的选择，因为它不会锁住整个对象（当然你也可以让它锁住整个对象）。同步方法会锁住整个对象，哪怕这个类中有多个不相关联的同步块，这通常会导致他们停止执行并需要等待获得这个对象上的锁。

#### 如何创建守护线程？

使用Thread类的setDaemon(true)方法可以将线程设置为守护线程，需要注意的是，需要在调用start()方法前调用这个方法，否则会抛出IllegalThreadStateException异常。

#### 什么是ThreadLocal?

ThreadLocal用于创建线程的本地变量，我们知道一个对象的所有线程会共享它的全局变量，所以这些变量不是线程安全的，我们可以使用同步技术。但是当我们不想使用同步的时候，我们可以选择ThreadLocal变量。

每个线程都会拥有他们自己的Thread变量，它们可以使用get()\set()方法去获取他们的默认值或者在线程内部改变他们的值。ThreadLocal实例通常是希望它们同线程状态关联起来是private static属性。在[ThreadLocal例子](http://www.journaldev.com/1076/java-threadlocal-example-to-create-thread-local-variables)这篇文章中你可以看到一个关于ThreadLocal的小程序。

#### 什么是Thread Group？为什么不建议使用它？

ThreadGroup是一个类，它的目的是提供关于线程组的信息。

ThreadGroup API比较薄弱，它并没有比Thread提供了更多的功能。它有两个主要的功能：一是获取线程组中处于活跃状态线程的列表；二是设置为线程设置未捕获异常处理器(ncaught exception handler)。但在Java 1.5中Thread类也添加了setUncaughtExceptionHandler(UncaughtExceptionHandler eh) 方法，所以ThreadGroup是已经过时的，不建议继续使用。

#### 什么是Java线程转储(Thread Dump)，如何得到它？

线程转储是一个JVM活动线程的列表，它对于分析系统瓶颈和死锁非常有用。有很多方法可以获取线程转储——使用Profiler，Kill -3命令，jstack工具等等。我更喜欢jstack工具，因为它容易使用并且是JDK自带的。由于它是一个基于终端的工具，所以我们可以编写一些脚本去定时的产生线程转储以待分析。读这篇文档可以了解更多关于[产生线程转储](http://www.journaldev.com/1053/how-to-generate-thread-dump-in-java)的知识。

#### 什么是死锁(Deadlock)？如何分析和避免死锁？

死锁是指两个以上的线程永远阻塞的情况，这种情况产生至少需要两个以上的线程和两个以上的资源。

分析死锁，我们需要查看Java应用程序的线程转储。我们需要找出那些状态为BLOCKED的线程和他们等待的资源。每个资源都有一个唯一的id，用这个id我们可以找出哪些线程已经拥有了它的对象锁。

避免嵌套锁，只在需要的地方使用锁和避免无限期等待是避免死锁的通常办法，阅读这篇文章去学习[如何分析死锁](http://www.journaldev.com/1058/java-deadlock-example-and-how-to-analyze-deadlock-situation)。

#### 什么是Java Timer类？如何创建一个有特定时间间隔的任务？

java.util.Timer是一个工具类，可以用于安排一个线程在未来的某个特定时间执行。Timer类可以用安排一次性任务或者周期任务。

java.util.TimerTask是一个实现了Runnable接口的抽象类，我们需要去继承这个类来创建我们自己的定时任务并使用Timer去安排它的执行。

这里有关于[java Timer的例子](http://www.journaldev.com/1050/java-timer-and-timertask-example-tutorial)。

#### 什么是线程池？如何创建一个Java线程池？

一个线程池管理了一组工作线程，同时它还包括了一个用于放置等待执行的任务的队列。

java.util.concurrent.Executors提供了一个 java.util.concurrent.Executor接口的实现用于创建线程池。[线程池例子](http://www.journaldev.com/1069/java-thread-pool-example-using-executors-and-threadpoolexecutor)展现了如何创建和使用线程池，或者阅读[ScheduledThreadPoolExecutor](http://www.journaldev.com/2340/java-scheduledthreadpoolexecutor-example-to-schedule-tasks-after-delay-and-execute-periodically)例子，了解如何创建一个周期任务。

### Java并发面试问题

#### 什么是原子操作？在Java Concurrency API中有哪些原子类(atomic classes)？

原子操作是指一个不受其他操作影响的操作任务单元。原子操作是在多线程环境下避免数据不一致必须的手段。

int++并不是一个原子操作，所以当一个线程读取它的值并加1时，另外一个线程有可能会读到之前的值，这就会引发错误。

为了解决这个问题，必须保证增加操作是原子的，在JDK1.5之前我们可以使用同步技术来做到这一点。到JDK1.5，java.util.concurrent.atomic包提供了int和long类型的装类，它们可以自动的保证对于他们的操作是原子的并且不需要使用同步。可以阅读这篇文章来了解[Java的atomic类](http://www.journaldev.com/1095/java-atomic-operations-atomicinteger-example)。

#### Java Concurrency API中的Lock接口(Lock interface)是什么？对比同步它有什么优势？

Lock接口比同步方法和同步块提供了更具扩展性的锁操作。他们允许更灵活的结构，可以具有完全不同的性质，并且可以支持多个相关类的条件对象。

它的优势有：

* 可以使锁更公平
* 可以使线程在等待锁的时候响应中断
* 可以让线程尝试获取锁，并在无法获取锁的时候立即返回或者等待一段时间
* 可以在不同的范围，以不同的顺序获取和释放锁

阅读更多[关于锁的例子](http://www.journaldev.com/2377/java-lock-example-and-concurrency-lock-vs-synchronized)

#### 什么是Executors框架？

Executor框架同java.util.concurrent.Executor 接口在Java 5中被引入。Executor框架是一个根据一组执行策略调用，调度，执行和控制的异步任务的框架。

无限制的创建线程会引起应用程序内存溢出。所以创建一个线程池是个更好的的解决方案，因为可以限制线程的数量并且可以回收再利用这些线程。利用Executors框架可以非常方便的创建一个线程池，阅读这篇文章可以了解[如何使用Executor框架创建一个线程池](http://www.journaldev.com/1069/java-thread-pool-example-using-executors-and-threadpoolexecutor)。

#### 什么是阻塞队列？如何使用阻塞队列来实现生产者-消费者模型？

java.util.concurrent.BlockingQueue的特性是：当队列是空的时，从队列中获取或删除元素的操作将会被阻塞，或者当队列是满时，往队列里添加元素的操作会被阻塞。

阻塞队列不接受空值，当你尝试向队列中添加空值的时候，它会抛出NullPointerException。

阻塞队列的实现都是线程安全的，所有的查询方法都是原子的并且使用了内部锁或者其他形式的并发控制。

BlockingQueue 接口是java collections框架的一部分，它主要用于实现生产者-消费者问题。

阅读这篇文章了解[如何使用阻塞队列实现生产者-消费者问题。](http://www.journaldev.com/1034/java-blockingqueue-example-implementing-producer-consumer-problem)

#### 什么是Callable和Future?

Java 5在concurrency包中引入了java.util.concurrent.Callable 接口，它和Runnable接口很相似，但它可以返回一个对象或者抛出一个异常。

Callable接口使用泛型去定义它的返回类型。Executors类提供了一些有用的方法去在线程池中执行Callable内的任务。由于Callable任务是并行的，我们必须等待它返回的结果。java.util.concurrent.Future对象为我们解决了这个问题。在线程池提交Callable任务后返回了一个Future对象，使用它我们可以知道Callable任务的状态和得到Callable返回的执行结果。Future提供了get()方法让我们可以等待Callable结束并获取它的执行结果。

阅读这篇文章了解更多[关于Callable，Future的例子](http://www.journaldev.com/1090/java-callable-future-example)。

#### 什么是FutureTask?

FutureTask是Future的一个基础实现，我们可以将它同Executors使用处理异步任务。通常我们不需要使用FutureTask类，单当我们打算重写Future接口的一些方法并保持原来基础的实现是，它就变得非常有用。我们可以仅仅继承于它并重写我们需要的方法。阅读[Java FutureTask例子](http://www.journaldev.com/1650/java-futuretask-example-program)，学习如何使用它。

#### 什么是并发容器的实现？

Java集合类都是快速失败的，这就意味着当集合被改变且一个线程在使用迭代器遍历集合的时候，迭代器的next()方法将抛出ConcurrentModificationException异常。

并发容器支持并发的遍历和并发的更新。

主要的类有ConcurrentHashMap, CopyOnWriteArrayList 和CopyOnWriteArraySet，阅读这篇文章了解[如何避免ConcurrentModificationException](http://www.journaldev.com/378/how-to-avoid-concurrentmodificationexception-when-using-an-iterator)。

#### Executors类是什么？

Executors为Executor，ExecutorService，ScheduledExecutorService，ThreadFactory和Callable类提供了一些工具方法。

Executors可以用于方便的创建线程池

#### 为什么说ConcurrentHashMap是弱一致性的？以及为何多个线程并发修改ConcurrentHashMap时不会报ConcurrentModificationException？

　　参考：<http://ifeve.com/java-concurrent-hashmap-2/>

<http://ifeve.com/concurrenthashmap-weakly-consistent/>

<http://blog.csdn.net/liuzhengkang/article/details/2916620>

## 集合

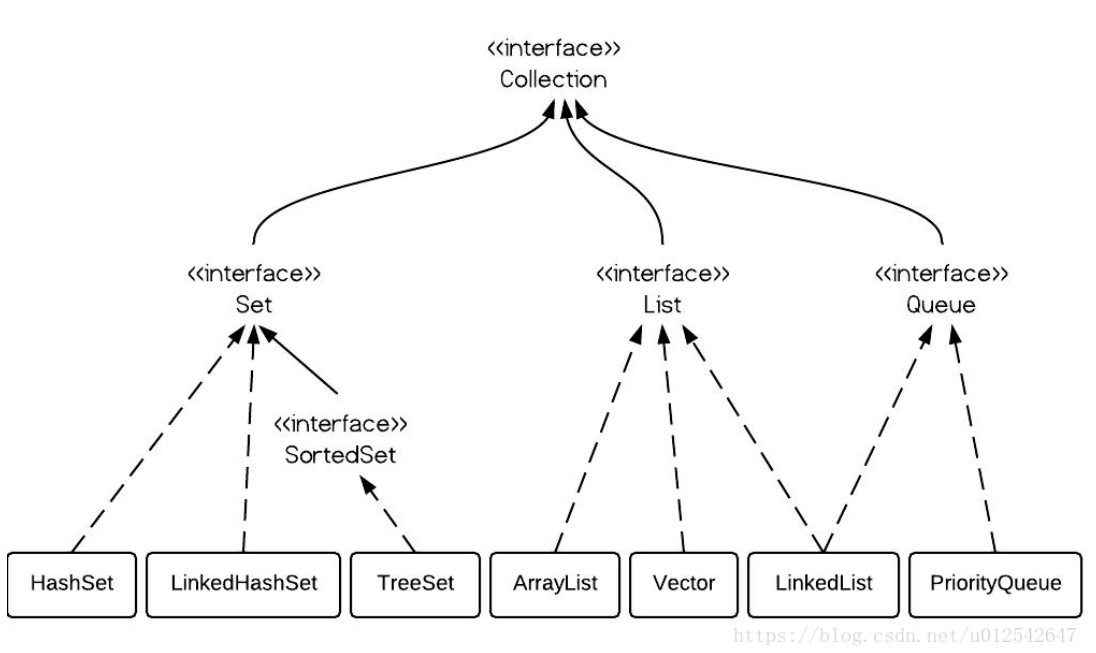
### 概述

Java提供的众多集合类由两大接口衍生而来：Collection接口和Map接口

**Collection接口**

Collection接口定义了一个包含一批对象的集合。接口的主要方法包括：

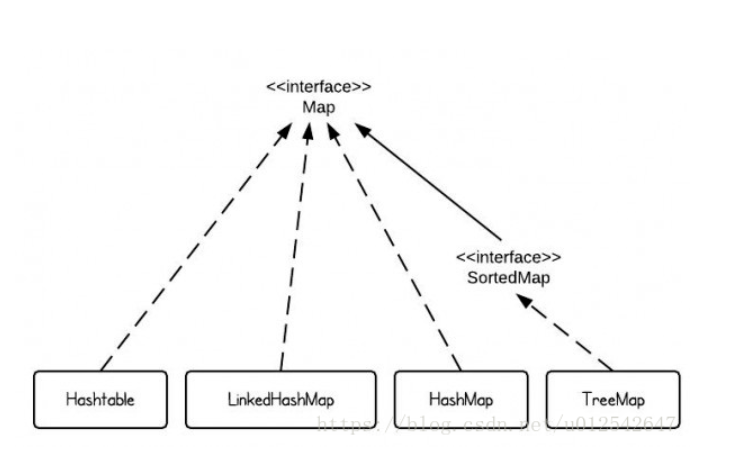
* size() - 集合内的对象数量
* add(E)/addAll(Collection) - 向集合内添加单个/批量对象
* remove(Object)/removeAll(Collection) - 从集合内删除单个/批量对象
* contains(Object)/containsAll(Collection) - 判断集合中是否存在某个/某些对象
* toArray() - 返回包含集合内所有对象的数组等



**Map接口**

Map接口在Collection的基础上，为其中的每个对象指定了一个key，并使用Entry保存每个key-value对，以实现通过key快速定位到对象(value)。Map接口的主要方法包括：

* size() - 集合内的对象数量
* put(K,V)/putAll(Map) - 向Map内添加单个/批量对象
* get(K) - 返回Key对应的对象
* remove(K) - 删除Key对应的对象
* keySet() - 返回包含Map中所有key的Set
* values() - 返回包含Map中所有value的Collection
* entrySet() - 返回包含Map中所有key-value对的EntrySet
* containsKey(K)/containsValue(V) - 判断Map中是否存在指定key/value等



在了解了Collection和Map两大接口之后，我们再来看一下这两个接口衍生出来的常用集合类：

### ****List类集合****

List接口继承自Collection，用于定义以列表形式存储的集合，List接口为集合中的每个对象分配了一个索引(index)，标记该对象在List中的位置，并可以通过index定位到指定位置的对象。

List在Collection基础上增加的主要方法包括：

* get(int) - 返回指定index位置上的对象
* add(E)/add(int, E) - 在List末尾/指定index位置上插入一个对象
* set(int, E) - 替换置于List指定index位置上的对象
* indexOf(Object) - 返回指定对象在List中的index位置
* subList(int,int) - 返回指定起始index到终止index的子List对象等

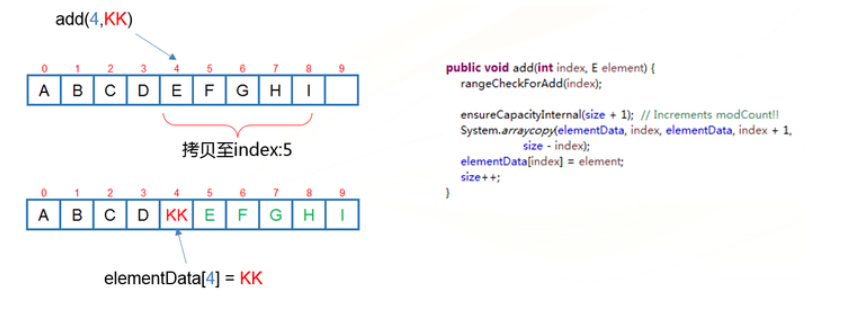
**List接口的常用实现类：**

#### ****ArrayList****

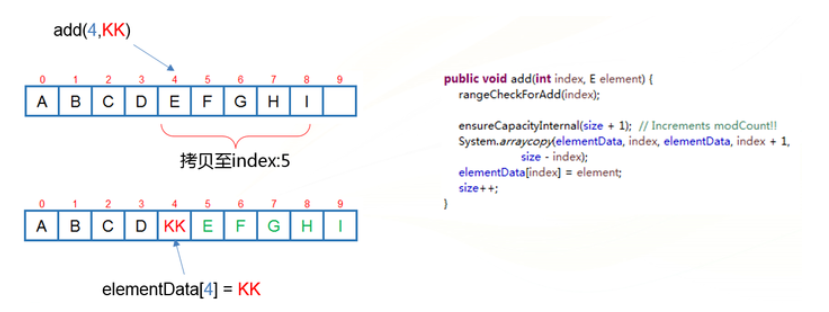
ArrayList基于数组来实现集合的功能，其内部维护了一个可变长的对象数组，数据结构是**线性表**，集合内所有对象存储于这个数组中，并实现该数组长度的动态伸缩；

ArrayList使用数组拷贝来实现指定位置的插入和删除：

插入：



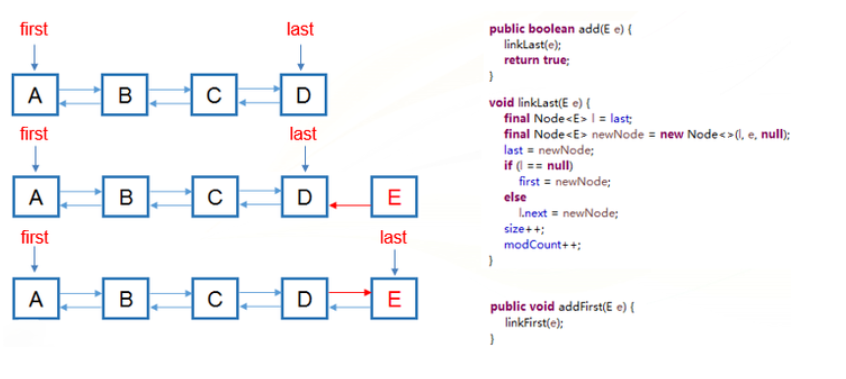
删除：



#### ****LinkedList****

LinkedList基于链表来实现集合的功能，其实现了静态类Node，集合中的每个对象都由一个Node保存，每个Node都拥有到自己的前一个和后一个Node的引用

LinkedList追加元素的过程示例：



**ArrayList vs LinkedList**

* ArrayList的随机访问更高，基于数组实现的ArrayList可直接定位到目标对象，而LinkedList需要从头Node或尾Node开始向后/向前遍历若干次才能定位到目标对象
* **LinkedList**在**头/尾**节点执行插入/删除操作的效率比**ArrayList**要高
* 由于ArrayList每次扩容的容量是当前的1.5倍，所以LinkedList所占的内存空间要更小一些
* 二者的遍历效率接近，但需要注意，遍历LinkedList时应用iterator方式，不要用get(int)方式，否则效率会很低

#### ****ArrayList和LinkedList的大致区别****

**LinkedList简介**

LinkedList 是一个继承于AbstractSequentialList的双向链表。它也可以被当作堆栈、队列或双端队列进行操作。  
LinkedList 实现 List 接口，能对它进行队列操作。  
LinkedList 实现 Deque 接口，即能将LinkedList当作双端队列使用。  
LinkedList 实现了Cloneable接口，即覆盖了函数clone()，能克隆。  
LinkedList 实现java.io.Serializable接口，这意味着LinkedList支持序列化，能通过序列化去传输。  
LinkedList 是非同步的。

1.ArrayList是实现了基于动态数组的数据结构，LinkedList是基于链表结构。

2.对于随机访问的get和set方法，ArrayList要优于LinkedList，因为LinkedList要移动指针。

3.对于新增和删除操作add和remove，LinkedList比较占优势，因为ArrayList要移动数据。

所以在我们进行对元素的增删查操作的时候，进行查操作时用ArrayList，进行增删操作的时候最好用LinkedList。

#### ****Vector****

Vector和ArrayList很像，都是基于数组实现的集合，它和ArrayList的主要区别在于（数据结构同样是**线性表**）

* Vector是线程安全的，而ArrayList不是
* 由于Vector中的方法基本都是synchronized的，其性能低于ArrayList
* Vector可以定义数组长度扩容的因子，ArrayList不能

#### ****CopyOnWriteArrayList****

与 Vector一样，CopyOnWriteArrayList也可以认为是ArrayList的线程安全版，不同之处在于 CopyOnWriteArrayList在写操作时会先复制出一个副本，在新副本上执行写操作，然后再修改引用。这种机制让 CopyOnWriteArrayList可以对读操作不加锁，这就使CopyOnWriteArrayList的读效率远高于Vector。 CopyOnWriteArrayList的理念比较类似读写分离，适合读多写少的多线程场景。但要注意，CopyOnWriteArrayList只能保证数据的最终一致性，并不能保证数据的实时一致性，如果一个写操作正在进行中且并未完成，此时的读操作无法保证能读到这个写操作的结果。

**Vector vs CopyOnWriteArrayList**

* 二者均是线程安全的、基于数组实现的List
* Vector是【绝对】线程安全的，CopyOnWriteArrayList只能保证读线程会读到【已完成】的写结果，但无法像Vector一样实现读操作的【等待写操作完成后再读最新值】的能力
* CopyOnWriteArrayList读性能远高于Vector，并发线程越多优势越明显
* CopyOnWriteArrayList占用更多的内存空间

### ****Map类集合****

Map与Set

Java先实现了Map，然后通过包装一个value都为null的Map就实现了Set

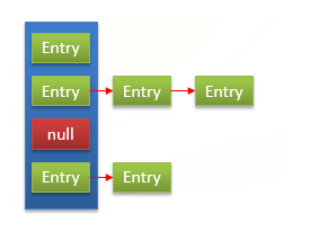
Map将key和value封装至一个叫做Entry的对象中，Map中存储的元素实际是Entry。只有在keySet()和values()方法被调用时，Map才会将keySet和values对象实例化。

每一个Map根据其自身特点，都有不同的Entry实现，以对应Map的内部类形式出现。

前文已经对Map接口的基本特点进行过描述，我们直接来看一下Map接口的常用实现类

#### ****HashMap****

HashMap的数据结构是**数组&链表**；HashMap将Entry对象存储在一个数组中，并通过哈希表来实现对Entry的快速访问：

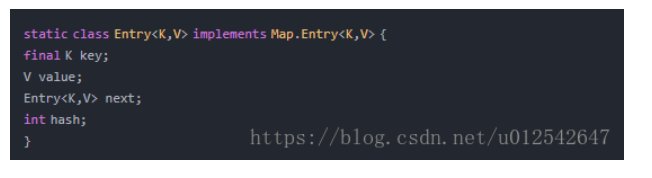


由每个Entry中的key的哈希值决定该Entry在数组中的位置。以这种特性能够实现通过key快速查找到Entry，从而获得该key对应的value。在不发生哈希冲突的前提下，查找的时间复杂度是O(1)。

如果两个不同的key计算出的index是一样的，就会发生两个不同的key都对应到数组中同一个位置的情况，也就是所谓的哈希冲突。HashMap处理哈希冲突的方法是拉链法，也就是说数组中每个位置保存的实际是一个Entry链表，链表中每个Entry都拥有指向链表中后一个Entry的引用。在发生哈希冲突时，将冲突的Entry追加至链表的头部。当HashMap在寻址时发现某个key对应的数组index上有多个Entry，便会遍历该位置上的 Entry链表，直到找到目标的Entry。



HashMap的Entry类：



HashMap由于其快速寻址的特点，可以说是最经常被使用的Map实现类

##### HashMap的实现原理

##### 1. HashMap的数据结构

数据结构中有数组和链表来实现对数据的存储，但这两者基本上是两个极端。

###### 数组

数组存储区间是连续的，占用内存严重，故空间复杂的很大。但数组的二分查找时间复杂度小，为O(1)；数组的特点是：寻址容易，插入和删除困难；

###### 链表

链表存储区间离散，占用内存比较宽松，故空间复杂度很小，但时间复杂度很大，达O（N）。**链表**的特点是：寻址困难，插入和删除容易。

###### 哈希表

那么我们能不能综合两者的特性，做出一种寻址容易，插入删除也容易的数据结构？答案是肯定的，这就是我们要提起的哈希表。哈希表（(Hash table）既满足了数据的查找方便，同时不占用太多的内容空间，使用也十分方便。

　　哈希表有多种不同的实现方法，我接下来解释的是最常用的一种方法—— 拉链法，我们可以理解为“**链表的数组**” ，如图：

　　从上图我们可以发现哈希表是由**数组+链表**组成的，一个长度为16的数组中，每个元素存储的是一个链表的头结点。那么这些元素是按照什么样的规则存储到数组中呢。一般情况是通过hash(key)%len获得，也就是元素的key的哈希值对数组长度取模得到。比如上述哈希表中，12%16=12,28%16=12,108%16=12,140%16=12。所以12、28、108以及140都存储在数组下标为12的位置。

　　HashMap其实也是一个线性的数组实现的,所以可以理解为其存储数据的容器就是一个线性数组。这可能让我们很不解，一个线性的数组怎么实现按键值对来存取数据呢？这里HashMap有做一些处理。

　　首先HashMap里面实现一个静态内部类Entry，其重要的属性有 key , value, next，从属性key,value我们就能很明显的看出来Entry就是HashMap键值对实现的一个基础bean，我们上面说到HashMap的基础就是一个线性数组，这个数组就是Entry[]，Map里面的内容都保存在Entry[]里面。

    /\*\*

     \* The table, resized as necessary. Length MUST Always be a power of two.

     \*/

**transient** Entry[] table;

##### 2. HashMap的存取实现

  既然是线性数组，为什么能随机存取？这里HashMap用了一个小算法，大致是这样实现：

// 存储时:  
**int** hash = key.hashCode(); // 这个hashCode方法这里不详述,只要理解每个key的hash是一个固定的int值  
**int** index = hash % Entry[].length;  
Entry[index] = value;  
  
// 取值时:  
**int** hash = key.hashCode();  
**int** index = hash % Entry[].length;  
**return** Entry[index];

###### 1）put

疑问：如果两个key通过hash%Entry[].length得到的index相同，会不会有覆盖的危险？

　　这里HashMap里面用到链式数据结构的一个概念。上面我们提到过Entry类里面有一个next属性，作用是指向下一个Entry。打个比方， 第一个键值对A进来，通过计算其key的hash得到的index=0，记做:Entry[0] = A。一会后又进来一个键值对B，通过计算其index也等于0，现在怎么办？HashMap会这样做:B.next = A,Entry[0] = B,如果又进来C,index也等于0,那么C.next = B,Entry[0] = C；这样我们发现index=0的地方其实存取了A,B,C三个键值对,他们通过next这个属性链接在一起。所以疑问不用担心。**也就是说数组中存储的是最后插入的元素。**到这里为止，HashMap的大致实现，我们应该已经清楚了。

**public** V put(K key, V value) {

**if** (key == **null**)

**return** putForNullKey(value); //null总是放在数组的第一个链表中

**int** hash = *hash*(key.hashCode());

**int** i = *indexFor*(hash, table.length);

        //遍历链表

**for** (Entry<K,V> e = table[i]; e != **null**; e = e.next) {

            Object k;

            //如果key在链表中已存在，则替换为新value

**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**;

    }

**void** addEntry(**int** hash, K key, V value, **int** bucketIndex) {

    Entry<K,V> e = table[bucketIndex];

    table[bucketIndex] = **new** Entry<K,V>(hash, key, value, e); //参数e, 是Entry.next

    //如果size超过threshold，则扩充table大小。再散列

**if** (size++ >= threshold)

            resize(2 \* table.length);

}

　　当然HashMap里面也包含一些优化方面的实现，这里也说一下。比如：Entry[]的长度一定后，随着map里面数据的越来越长，这样同一个index的链就会很长，会不会影响性能？HashMap里面设置一个因子，随着map的size越来越大，Entry[]会以一定的规则加长长度。

###### 2）get

**public** V get(Object key) {

**if** (key == **null**)

**return** getForNullKey();

**int** hash = *hash*(key.hashCode());

        //先定位到数组元素，再遍历该元素处的链表

**for** (Entry<K,V> e = table[*indexFor*(hash, table.length)];

             e != **null**;

             e = e.next) {

            Object k;

**if** (e.hash == hash && ((k = e.key) == key || key.equals(k)))

**return** e.value;

        }

**return** **null**;

}

###### 3）null key的存取

null key总是存放在Entry[]数组的第一个元素。

**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**;

    }

**private** V getForNullKey() {

**for** (Entry<K,V> e = table[0]; e != **null**; e = e.next) {

**if** (e.key == **null**)

**return** e.value;

        }

**return** **null**;

    }

###### 4）确定数组index：hashcode % table.length取模

HashMap存取时，都需要计算当前key应该对应Entry[]数组哪个元素，即计算数组下标；算法如下：

   /\*\*

     \* Returns index for hash code h.

     \*/

**static** **int** indexFor(**int** h, **int** length) {

**return** h & (length-1);

    }

按位取并，作用上相当于取模mod或者取余%。

这意味着数组下标相同，并不表示hashCode相同。

###### 5）table初始大小

**public** HashMap(**int** initialCapacity, **float** loadFactor) {

        .....

        // Find a power of 2 >= initialCapacity

**int** capacity = 1;

**while** (capacity < initialCapacity)

            capacity <<= 1;

**this**.loadFactor = loadFactor;

        threshold = (**int**)(capacity \* loadFactor);

        table = **new** Entry[capacity];

        init();

    }

**注意table初始大小并不是构造函数中的initialCapacity！！**

**而是 >= initialCapacity的2的n次幂！！！！**

————为什么这么设计呢？——

##### 3. 解决hash冲突的办法

1. 开放定址法（线性探测再散列，二次探测再散列，伪随机探测再散列）
2. 再哈希法
3. **链地址法**
4. 建立一个公共溢出区

Java中hashmap的解决办法就是采用的链地址法。

##### 4. 再散列rehash过程

当哈希表的容量超过默认容量时，必须调整table的大小。当容量已经达到最大可能值时，那么该方法就将容量调整到Integer.MAX\_VALUE返回，这时，需要创建一张新表，将原表的映射到新表中。

   /\*\*

     \* Rehashes the contents of this map into a new array with a

     \* larger capacity.  This method is called automatically when the

     \* number of keys in this map reaches its threshold.

     \*

     \* If current capacity is MAXIMUM\_CAPACITY, this method does not

     \* resize the map, but sets threshold to Integer.MAX\_VALUE.

     \* This has the effect of preventing future calls.

     \*

     \* **@param** newCapacity the new capacity, MUST be a power of two;

     \*        must be greater than current capacity unless current

     \*        capacity is MAXIMUM\_CAPACITY (in which case value

     \*        is irrelevant).

     \*/

**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);

        table = newTable;

        threshold = (**int**)(newCapacity \* loadFactor);

    }

    /\*\*

     \* Transfers all entries from current table to newTable.

     \*/

**void** transfer(Entry[] newTable) {

        Entry[] src = table;

**int** newCapacity = newTable.length;

**for** (**int** j = 0; j < src.length; j++) {

            Entry<K,V> e = src[j];

**if** (e != **null**) {

                src[j] = **null**;

**do** {

                    Entry<K,V> next = e.next;

                    //重新计算index

**int** i = *indexFor*(e.hash, newCapacity);

                    e.next = newTable[i];

                    newTable[i] = e;

                    e = next;

                } **while** (e != **null**);

            }

        }

    }

--------------------- 作者：AlphaWang 来源：CSDN 原文：https://blog.csdn.net/vking\_wang/article/details/14166593?utm\_source=copy 版权声明：本文为博主原创文章，转载请附上博文链接！

##### HashMap集合的四种遍历方式

import java.util.HashMap;

import java.util.Iterator;

import java.util.Map;

public class TestMap {

public static void main(String[] args) {

Map<Integer, String> map = new HashMap<Integer, String>();

map.put(1, "a");

map.put(2, "b");

map.put(3, "ab");

map.put(4, "ab");

map.put(4, "ab");// 和上面相同 ， 会自己筛选

System.out.println(map.size());

// 第一种：

/\*

\* Set<Integer> set = map.keySet(); //得到所有key的集合

\*

\* for (Integer in : set) { String str = map.get(in);

\* System.out.println(in + " " + str); }

\*/

System.out.println("第一种：通过Map.keySet遍历key和value：");

for (Integer in : map.keySet()) {

//map.keySet()返回的是所有key的值

String str = map.get(in);//得到每个key多对用value的值

System.out.println(in + " " + str);

}

// 第二种：

System.out.println("第二种：通过Map.entrySet使用iterator遍历key和value：");

Iterator<Map.Entry<Integer, String>> it = map.entrySet().iterator();

while (it.hasNext()) {

Map.Entry<Integer, String> entry = it.next();

System.out.println("key= " + entry.getKey() + " and value= " + entry.getValue());

}

// 第三种：推荐，尤其是容量大时

System.out.println("第三种：通过Map.entrySet遍历key和value");

for (Map.Entry<Integer, String> entry : map.entrySet()) {

//Map.entry<Integer,String> 映射项（键-值对） 有几个方法：用上面的名字entry

//entry.getKey() ;entry.getValue(); entry.setValue();

//map.entrySet() 返回此映射中包含的映射关系的 Set视图。

System.out.println("key= " + entry.getKey() + " and value= "

+ entry.getValue());

}

// 第四种：

System.out.println("第四种：通过Map.values()遍历所有的value，但不能遍历key");

for (String v : map.values()) {

System.out.println("value= " + v);

}

}

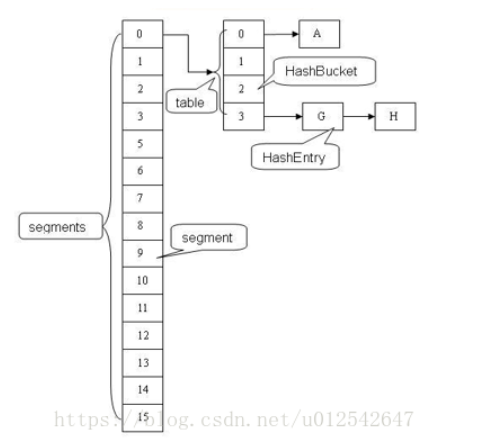
}

#### ****Hashtable****

Hashtable 可以说是HashMap的前身（Hashtable自JDK1.0就存在，而HashMap乃至整个Map接口都是JDK1.2引入的新特性），其实现思 路与HashMap几乎完全一样，都是通过数组存储Entry，以key的哈希值计算Entry在数组中的index，用拉链法解决哈希冲突。二者最大的不同在于，Hashtable是线程安全的，其提供的方法几乎都是同步的。

#### ****ConcurrentHashMap****

ConcurrentHashMap是HashMap的线程安全版（自JDK1.5引入），提供比Hashtable更高效的并发性能。



Hashtable 在进行读写操作时会锁住整个Entry数组，这就导致数据越多性能越差。而ConcurrentHashMap使用分离锁的思路解决并发性能，其将 Entry数组拆分至16个Segment中，以哈希算法决定Entry应该存储在哪个Segment。这样就可以实现在写操作时只对一个Segment 加锁，大幅提升了并发写的性能。

在进行读操作时，ConcurrentHashMap在绝大部分情况下都不需要加锁，其Entry中的value是volatile的，这保证了value被修改时的线程可见性，无需加锁便能实现线程安全的读操作。

ConcurrentHashMap的HashEntry类：



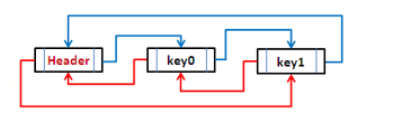
但是鱼与熊掌不可兼得，ConcurrentHashMap的高性能是有代价的（否则Hashtable就没有存在价值了），那就是它不能保证读操作的绝对 一致性。ConcurrentHashMap保证读操作能获取到已存在Entry的value的最新值，同时也能保证读操作可获取到已完成的写操作的内容，但如果写操作是在创建一个新的Entry，那么在写操作没有完成时，读操作是有可能获取不到这个Entry的。

**HashMap vs Hashtable vs ConcurrentHashMap**

* 三者在数据存储层面的机制原理基本一致
* HashMap不是线程安全的，多线程环境下除了不能保证数据一致性之外，还有可能在rehash阶段引发Entry链表成环，导致死循环
* Hashtable是线程安全的，能保证绝对的数据一致性，但性能是问题，并发线程越多，性能越差
* ConcurrentHashMap 也是线程安全的，使用分离锁和volatile等方法极大地提升了读写性能，同时也能保证在绝大部分情况下的数据一致性。但其不能保证绝对的数据一致性， 在一个线程向Map中加入Entry的操作没有完全完成之前，其他线程有可能读不到新加入的Entry

#### ****LinkedHashMap****

LinkedHashMap与HashMap非常类似，唯一的不同在于前者的Entry在HashMap.Entry的基础上增加了到前一个插入和后一个插入的Entry的引用，以实现能够按Entry的插入顺序进行遍历。



#### ****TreeMap****

TreeMap是基于红黑树实现的Map结构，其Entry类拥有到左/右叶子节点和父节点的引用，同时还记录了自己的颜色：



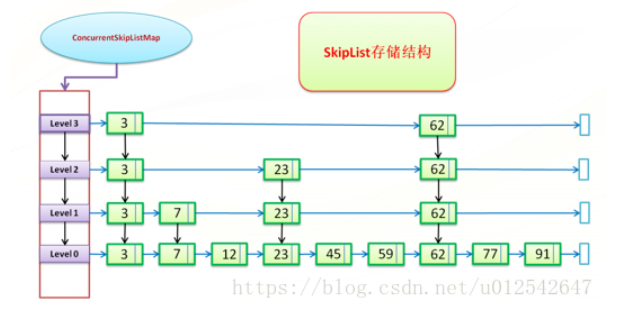
红黑树实际是一种算法复杂但高效的平衡二叉树，具备二叉树的基本性质，即任何节点的值大于其左叶子节点，小于其右叶子节点，利用这种特性，TreeMap能够实现Entry的排序和快速查找。

关于红黑树的具体介绍，可以参考这篇文章，非常详细：[http://blog.csdn.net/chenssy/article/details/26668941](https://link.jianshu.com/?t=http://blog.csdn.net/chenssy/article/details/26668941)

TreeMap的Entry是有序的，所以提供了一系列方便的功能，比如获取以升序或降序排列的KeySet(EntrySet)、获取在指定key(Entry)之前/之后的key(Entry)等等。适合需要对key进行有序操作的场景。

#### ****ConcurrentSkipListMap****

ConcurrentSkipListMap同样能够提供有序的Entry排列，但其实现原理与TreeMap不同，是基于跳表(SkipList)的：



如上图所示，ConcurrentSkipListMap由一个多级链表实现，底层链上拥有所有元素，逐级上升的过程中每个链的元素数递减。在查找时从顶层链出发，按先右后下的优先级进行查找，从而实现快速寻址。



与TreeMap不同，ConcurrentSkipListMap在进行插入、删除等操作时，只需要修改影响到的节点的右引用，而右引用又是volatile的，所以ConcurrentSkipListMap是线程安全的。但ConcurrentSkipListMap与ConcurrentHashMap一样，不能保证数据的绝对一致性，在某些情况下有可能无法读到正在被插入的数据。

与TreeMap不同，ConcurrentSkipListMap在进行插入、删除等操作时，只需要修改影响到的节点的右引用，而右引用又是volatile的，所以ConcurrentSkipListMap是线程安全的。但ConcurrentSkipListMap与ConcurrentHashMap一样，不能保证数据的绝对一致性，在某些情况下有可能无法读到正在被插入的数据。

**TreeMap vs ConcurrentSkipListMap**

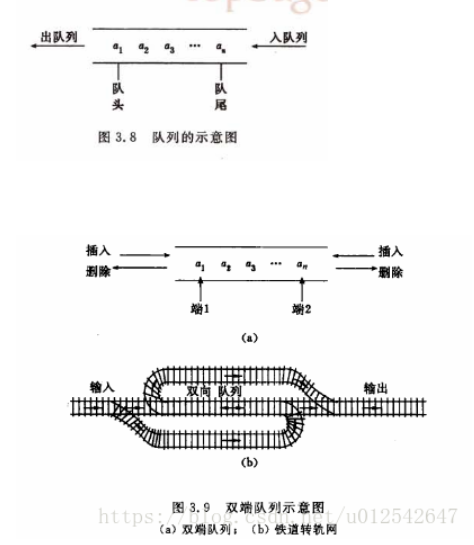
* 二者都能够提供有序的Entry集合
* 二者的性能相近，查找时间复杂度都是O(logN)
* ConcurrentSkipListMap会占用更多的内存空间
* ConcurrentSkipListMap是线程安全的，TreeMap不是

### ****Set类集合****

Set 接口继承Collection，用于存储不含重复元素的集合。几乎所有的Set实现都是基于同类型Map的，简单地说，Set是阉割版的Map。每一个Set内都有一个同类型的Map实例（CopyOnWriteArraySet除外，它内置的是CopyOnWriteArrayList实例），Set把元素作为key存储在自己的Map实例中，value则是一个空的Object。Set的常用实现也包括 HashSet、TreeSet、ConcurrentSkipListSet等，原理和对应的Map实现完全一致，此处不再赘述。

### ****Queue/Deque类集合****

Queue和Deque接口继承Collection接口，实现FIFO（先进先出）的集合。二者的区别在于，Queue只能在队尾入队，队头出队（**队列**），而Deque接口则在队头和队尾都可以执行出/入队操作（**双端队列**）



Queue接口常用方法：

* add(E)/offer(E)：入队，即向队尾追加元素，二者的区别在于如果队列是有界的，add方法在队列已满的情况下会抛出IllegalStateException，而offer方法只会返回false
* remove()/poll()：出队，即从队头移除1个元素，二者的区别在于如果队列是空的，remove方法会抛出NoSuchElementException，而poll只会返回null
* element()/peek()：查看队头元素，二者的区别在于如果队列是空的，element方法会抛出NoSuchElementException，而peek只会返回null

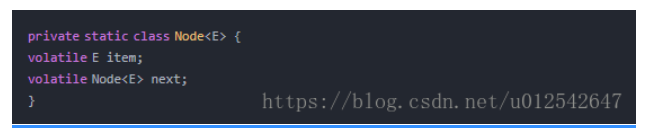
Deque接口常用方法：

* addFirst(E) / addLast(E) / offerFirst(E) / offerLast(E)
* removeFirst() / removeLast() / pollFirst() / pollLast()
* getFirst() / getLast() / peekFirst() / peekLast()
* removeFirstOccurrence(Object) / removeLastOccurrence(Object)

Queue接口的常用实现类：

#### ****ConcurrentLinkedQueue****

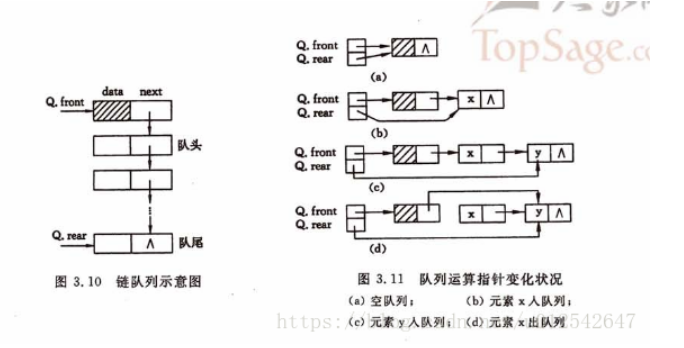
ConcurrentLinkedQueue是**基于链表实现的队列**，队列中每个Node拥有到下一个Node的引用：



由于Node类的成员都是volatile的，所以ConcurrentLinkedQueue自然是线程安全的。能够保证入队和出队操作的原子性和一致性，但在遍历和size()操作时只能保证数据的弱一致性。

#### ****LinkedBlockingQueue****

与ConcurrentLinkedQueue不同，LinkedBlocklingQueue是一种无界的阻塞队列。所谓阻塞队列，就是在入队时如果队列已满，线程会被阻塞，直到队列有空间供入队再返回；同时在出队时，如果队列已空，线程也会被阻塞，直到队列中有元素供出队时再返回。LinkedBlocklingQueue同样基于链表实现，其出队和入队操作都会使用ReentrantLock进行加锁。所以本身是线程安全的，但同样的，只能保证入队和出队操作的原子性和一致性，在遍历时只能保证数据的弱一致性。



#### ****ArrayBlockingQueue****

ArrayBlockingQueue是一种有界的阻塞队列，基于数组实现。其同步阻塞机制的实现与LinkedBlocklingQueue基本一致，区别仅在于前者的生产和消费使用同一个锁，后者的生产和消费使用分离的两个锁。

**ConcurrentLinkedQueue vsLinkedBlocklingQueue vs ArrayBlockingQueue**

* ConcurrentLinkedQueue是非阻塞队列，其他两者为阻塞队列
* 三者都是线程安全的
* LinkedBlocklingQueue是无界的，适合实现不限长度的队列， ArrayBlockingQueue适合实现定长的队列

#### ****SynchronousQueue****

SynchronousQueue算是JDK实现的队列中比较奇葩的一个，它不能保存任何元素，size永远是0，peek()永远返回null。向其中插入元素的线程会阻塞，直到有另一个线程将这个元素取走，反之从其中取元素的线程也会阻塞，直到有另一个线程插入元素。

这种实现机制非常适合传递性的场景。也就是说如果生产者线程需要及时确认到自己生产的任务已经被消费者线程取走后才能执行后续逻辑的场景下，适合使用SynchronousQueue。

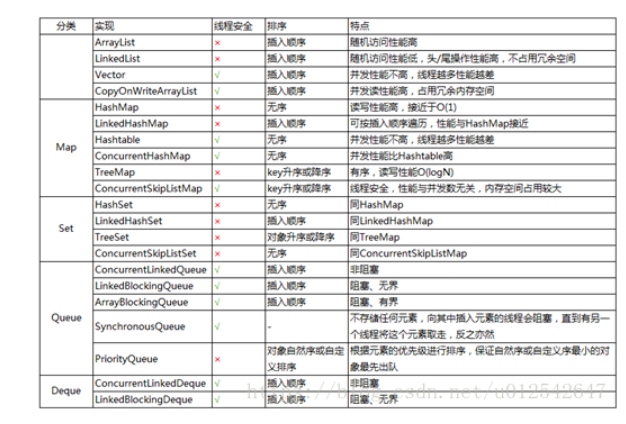
#### ****PriorityQueue & PriorityBlockingQueue****

这两种Queue并不是FIFO队列，而是根据元素的优先级进行排序，保证最小的元素最先出队，也可以在构造队列时传入Comparator实例，这样PriorityQueue就会按照Comparator实例的要求对元素进行排序。

PriorityQueue是非阻塞队列，也不是线程安全的，PriorityBlockingQueue是阻塞队列，同时也是线程安全的。

Deque 的实现类包括LinkedList（前文已描述过）、ConcurrentLinkedDeque、LinkedBlockingDeque，其实现机制与前文所述的ConcurrentLinkedQueue和LinkedBlockingQueue非常类似，此处不再赘述

最后，对本文中描述的常用集合实现类做一个简单总结：



简而言之（常用的）：

* 关于数据结构：
  + 大部分的实现List的集合框架都结合了数组和线性表，如ArrayList、Vector，部分为链表，如LinkedList等；
  + 大部分实现Map的集合框架结合了数组和链表，如HashMap、HashTable，或纯链表，LinkedHashMap等.
* 关于线程安全：
  + ArrayList线程不安全，Vector、CopyOnWriteArrayList（升级版ArrayList，可保证结果一致性）线程安全；
  + HashMap线程不安全，HashTable、LinkedHashMap线程安全；
* 关于性能与特点：
  + ArrayList随机访问性能高，而线程安全的LinkedList和Vector，性能会较低；
  + HashMap读写性能高，LinkedHashMap接近，HashTable并发性能不高；

## 异常

### 异常类型

1. **unchecked exceptions（运行时异常）：**都是RuntimeException类及其子类异常，就是我们在开发中测试功能时程序终止，控制台出现的异常，比如：   
   NullPointerException(空指针异常)、   
   IndexOutOfBoundsException(下标越界异常)   
   ClassCastException(类转换异常)   
   ArrayStoreException(数据存储异常，操作数组时类型不一致)   
   IO操作的BufferOverflowException异常
2. **checked exceptions，非运行时异常 （编译异常）**：是RuntimeException以外的异常，类型上都属于Exception类及其子类。从程序语法角度讲是必须进行处理的异常，如果不处理，程序就不能编译通过。如IOException、SQLException等以及用户自定义的Exception异常，一般情况下不自定义检查异常。   
   通俗的话说就是在写代码时出现红线，需要try catch或者throws时出现的异常。

### finally关键字

finally 关键字用来创建在 try 代码块后面执行的代码块。

无论是否发生异常，finally 代码块中的代码总会被执行。

在 finally 代码块中，可以运行清理类型等收尾善后性质的语句

### Java内置异常

Java 根据各个类库也定义了一些其他的异常，下面的表中列出了 Java 的非检查性异常。

|  |  |
| --- | --- |
| **异常** | **描述** |
| ArithmeticException | 当出现异常的运算条件时，抛出此异常。例如，一个整数"除以零"时，抛出此类的一个实例。 |
| ArrayIndexOutOfBoundsException | 用非法索引访问数组时抛出的异常。如果索引为负或大于等于数组大小，则该索引为非法索引。 |
| ArrayStoreException | 试图将错误类型的对象存储到一个对象数组时抛出的异常。 |
| ClassCastException | 当试图将对象强制转换为不是实例的子类时，抛出该异常。 |
| IllegalArgumentException | 抛出的异常表明向方法传递了一个不合法或不正确的参数。 |
| IllegalMonitorStateException | 抛出的异常表明某一线程已经试图等待对象的监视器，或者试图通知其他正在等待对象的监视器而本身没有指定监视器的线程。 |
| IllegalStateException | 在非法或不适当的时间调用方法时产生的信号。换句话说，即 Java 环境或 Java 应用程序没有处于请求操作所要求的适当状态下。 |
| IllegalThreadStateException | 线程没有处于请求操作所要求的适当状态时抛出的异常。 |
| IndexOutOfBoundsException | 指示某排序索引（例如对数组、字符串或向量的排序）超出范围时抛出。 |
| NegativeArraySizeException | 如果应用程序试图创建大小为负的数组，则抛出该异常。 |
| NullPointerException | 当应用程序试图在需要对象的地方使用 null 时，抛出该异常 |
| NumberFormatException | 当应用程序试图将字符串转换成一种数值类型，但该字符串不能转换为适当格式时，抛出该异常。 |
| SecurityException | 由安全管理器抛出的异常，指示存在安全侵犯。 |
| StringIndexOutOfBoundsException | 此异常由 String 方法抛出，指示索引或者为负，或者超出字符串的大小。 |
| UnsupportedOperationException | 当不支持请求的操作时，抛出该异常。 |

下面的表中列出了 Java 定义在 java.lang 包中的检查性异常类。

|  |  |
| --- | --- |
| **异常** | **描述** |
| ClassNotFoundException | 应用程序试图加载类时，找不到相应的类，抛出该异常。 |
| CloneNotSupportedException | 当调用 Object 类中的 clone 方法克隆对象，但该对象的类无法实现 Cloneable 接口时，抛出该异常。 |
| IllegalAccessException | 拒绝访问一个类的时候，抛出该异常。 |
| InstantiationException | 当试图使用 Class 类中的 newInstance 方法创建一个类的实例，而指定的类对象因为是一个接口或是一个抽象类而无法实例化时，抛出该异常。 |
| InterruptedException | 一个线程被另一个线程中断，抛出该异常。 |
| NoSuchFieldException | 请求的变量不存在 |
| NoSuchMethodException | 请求的方法不存在 |

### 异常方法

下面的列表是 Throwable 类的主要方法:

|  |  |
| --- | --- |
| **序号** | **方法及说明** |
| 1 | **public String getMessage()** 返回关于发生的异常的详细信息。这个消息在Throwable 类的构造函数中初始化了。 |
| 2 | **public Throwable getCause()** 返回一个Throwable 对象代表异常原因。 |
| 3 | **public String toString()** 使用getMessage()的结果返回类的串级名字。 |
| 4 | **public void printStackTrace()** 打印toString()结果和栈层次到System.err，即错误输出流。 |
| 5 | **public StackTraceElement [] getStackTrace()** 返回一个包含堆栈层次的数组。下标为0的元素代表栈顶，最后一个元素代表方法调用堆栈的栈底。 |
| 6 | **public Throwable fillInStackTrace()** 用当前的调用栈层次填充Throwable 对象栈层次，添加到栈层次任何先前信息中。 |

### 使用throws抛出异常

如果在当前方法不知道该如何处理该异常时，则可以使用throws对异常进行抛出给调用者处理或者交给JVM。JVM对异常的处理方式是：打印异常的跟踪栈信息并终止程序运行。   
throws在使用时应处于方法签名之后使用，可以抛出多种异常并用英文字符逗号’,’隔开   
e.g.1   
public void throwsTest() throws ExceptionClass1, ExceptionClass2 {...}   
如果抛出给调用者的异常是Checked异常，这种异常是我们需要处理以来提高程序健壮性的，一般抛出则要调用者做相应处理，要么调用者对该异常进行try…catch处理，要么再次throws交给上一层。这其中需要注意一点：子类方法声明抛出的异常类型应是父类方法声明抛出的异常类型的子类或相同，子类方法声明跑出的异常不允许比父类方法声明抛出的异常多。   
e.g.2

public class ThrowsTest2{

//因为test();会抛出IOException，main方法是调用者

//则需要使用try...catch进行捕获或者抛给JVM

//抛出时要遵循父类声明抛出“大于”子类声明抛出的原则

public static void main(String[] args) throws Exception {

test();

}

public static void test() throws IOException() {

//因为FileInputStream构造器会抛出IOException

//所以需要使用try...catch块进行处理或使用throws抛给调用者

FileInputStream fis = new FileInputStream("a.txt");

}

}

### 使用throw抛出异常

如果需要程序在程序中自行抛出异常，应该使用throw语句抛出，抛出的不是一个类而是一个对象且只能抛出一个对象。它可以单独使用，也可以结合catch块捕获使用。如果抛出的异常对象时Checked异常则处于try块里被catch捕获或者放在一个带throws的方法里；如果抛出的是RuntimeException则既可以显示使用try…catch捕获也可以不理会它   
e.g.3

public class ThrowTest {

public static void main(String[] args) {

try{

throwChecked(3);

}catch(Exception e) {

System.out.println(e.getMessage());

}

throwRuntime(-3);

}

//该方法内抛出一个Exception异常对象，必须捕获或抛给调用者

public static void throwChecked(int a) throws Exception {

if(a < 0) {

throw new Exception("a的值应大于0，不符合要求")

}

}

//该方法内抛出一个RuntimeException对象，可以不理会直接交给JVM处理

public static void throwRuntime(int a) {

if(a < 0) {

throw new RuntimeException("a的值应大于0，不符合要求")

}

}

}

Java7增强的throw语句：   
在Java7之前，父类和子类在声明抛出异常时应符合父类包含的异常“大于等于”子类包含的异常的规则；从Java7开始，Java编译器会检查throw语句抛出的异常的实际类型，如下代码中编译器知道throw e只能抛出FileNotFoundException 所以在方法签名上可以直接写该异常：   
e.g.4

public class ThrowTest2 {

public static void main(String[] args) throws FileNotFoundException {

try {

new FileOutputStream("a.txt");

}catch(Exception e) {

e.printStackTrace();

throw e;

}

}

}

### 自定义异常类

在抛出异常时，异常类名往往包含有用的信息，所以在选择抛出异常时需要选择适合的类，从而可以明确的描述该异常情况。这时候就需要我们自己定义异常，自定义异常一定是Throwable的子类，若是检查异常就要继承自Exception，若是运行时异常就要继承自RuntimeException   
e.g.5

public class AuctionException extends Exception {

//无参构造

public AuctionException() {}

//含参构造

//通过调用父类的构造器将字符串msg传给异常对象的massage属性，

//massage属性就是对异常的描述

public AuctionException(String msg) {

super(msg);

}

}

关于自定义异常深入学习可以参考以下链接：   
<http://www.cnblogs.com/AlanLee/p/6104492.html>

### catch和throw同时使用

在实际应用中，在异常出现的当前方法中，程序只能对异常做部分处理，还有些处理需要在该方法的调用者才能够完成，使用需要再次抛出异常。这时，就需要将catch和throw结合使用。   
e.g.6

public class AuctionTest {

private double initPrice = 30.0;

public void bid(String bidPrice) throws AuctionException {

double d = 0.0;

try{

d = Double.parseDouble(bidPrice);

}catch(Exception e) {

e.printStackTrace();

throw new AuctionException("竞拍价必须为数值，不能包含其它数值");

}

if(initPrice > d) {

throw new AuctionException("竞拍价应比起拍价高");

}

initPrice = d;

}

public static void main(String[] args) {

AuctionTest at = new AuctionTest();

try{

at.bid("..");

}catch(AuctionException ae) {

System.out.println(ae.getMessage());

}

}

}

以上代码的运行结果为：

竞拍价必须为数值，不能包含其它数值

java.lang.NumberFormatException: multiple points

at sun.misc.FloatingDecimal.readJavaFormatString(Unknown Source)

at sun.misc.FloatingDecimal.parseDouble(Unknown Source)

at java.lang.Double.parseDouble(Unknown Source)

at ExceptionTest.AuctionTest.bid(AuctionTest.java:15)

at ExceptionTest.AuctionTest.main(AuctionTest.java:28)

### 异常链

把捕获一个异常然后接着抛出另一个异常，并把原始异常信息保存下来是一种典型的链式处理，也被称作异常链。

### Java的异常跟踪栈

如e.g.6的运行结果   
除去第一行打印的自定义的提示信息之外的第一行是异常的类型和详细消息，之后的提示信息中含有确切代码行数的信息记录了所有的异常发生点，各行显示被调用方法中执行的停止位置。跟踪栈总是最内部的被调用方法逐渐上传知道外部业务操作的起点，通常就是main方法或者Thread的run方法。

### 异常处理规则

1. 不要过度使用异常：对于完全已知的错误应编写处理这种错误代码从而提高代码的健壮性，只有外部的、不能确定的和不可预知的运行时错误使是用异常，并且异常机制的效率低于正常的流程控制。
2. 不要使用过于庞大的try块：过于庞大的try块业务也相对更复杂，会导致try块中异常的可能性大大增加，在分析发生异常的原因时难度增加。
3. 避免使用Catch All语句：Catch All是catch(Throwable t)，也会在发生异常是分析原因的复杂度增加。
4. 不要忽略已捕获到的异常：对于捕获到的异常应该对其进行处理从而提高代码健壮性，而不是什么都不做或者只是打印跟踪栈信息。

--------------------- 本文来自 sdr\_zd 的CSDN 博客 ，全文地址请点击：https://blog.csdn.net/sdr\_zd/article/details/75675004?utm\_source=copy

## 设计模式

### 六大原则

**1.单一职责原则（Single Responsibility Principle）**

定义：就一个类而言，应该仅有一个引起它变化的原因；

    如果一个类承担的职责过多，就等于把这些职责耦合在一起，一个职责变化可能会消弱或者抑制这个类完成其他职责的能力。这种耦合会导致脆弱的设计，当变化发生时，设计会遭受到意想不到的破坏；

    T负责两个不同的职责：职责P1，职责P2。当由于职责P1需求发生改变而需要修改类T时，有可能会导致原本运行正常的职责P2功能发生故障。也就是说职责P1和P2被耦合在了一起。

    单一职责比较容易理解，但是在实际设计过程中容易发生职责扩散：因为某种原因，某一职责被分化为颗粒度更细的多个职责了。

    解决办法：遵守单一职责原则，将不同的职责封装到不同的类或模块中。

**2.里氏替换原则（LiskovSubstitution Principle）**

定义：子类型必须能够替换掉它们的父类型；

    一个软件实体如果使用的是一个父类的话，那么一定适用于其子类，而且它察觉不出父类对象和子类对象的区别。也就是说，在软件里面，把父类都替换成它的子类，程序的行为没有变化；

    只有子类可以替换父类，软件单位功能不受到影响时，父类才能真正被复用，而子类也能够在父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。

    在进行设计的时候，我们尽量从抽象类继承，而不是从具体类继承。如果从继承等级树来看，所有叶子节点应当是具体类，而所有的树枝节点应当是抽象类或者接口。

**3.依赖倒置原则（DependenceInversion Principle）**

定义：

          A.高层模块不应该依赖低层模块。两个都应该依赖抽象；

          B.抽象不应该依赖细节，细节应该依赖抽象；（针对接口编程，而不是针对实现；）

面向过程的开发，上层调用下层，上层依赖于下层，当下层剧烈变动时上层也要跟着变动，这就会导致模块的复用性降低而且大大提高了开发的成本。依赖倒转很好的解决了这个问题；

**4.合成/聚合原则（Composite/Aggregate Reuse Principle）**

定义：尽量使用合成/聚合，尽量不要使用类继承；

    优先使用对象的合成/聚合将有助于你保持每个类被封装，并被集中在单个任务上。这样类和类继承层次会保持较小规模，并且不太可能增长为不可控制的庞然大物；

为什么尽量不要使用类继承而使用合成/聚合？

    对象的继承关系在编译时就定义好了，所以无法在运行时改变从父类继承的子类的实现

    子类的实现和它的父类有非常紧密的依赖关系，以至于父类实现中的任何变化必然会导致子类发生变化

    当你复用子类的时候，如果继承下来的实现不适合解决新的问题，则父类必须重写或者被其它更适合的类所替换

    这种依赖关系限制了灵活性，并最终限制了复用性

**5.迪米特法则（Law Of Demeter）**

定义：如果两个类不必彼此直接通信，那么这两个类就不应当发生直接的相互作用，如果其中一个类需要调用另一个类的某一个方法的话，可以通过第三者转发这个调用；

迪米特根本思想是：类之间的松耦合；

    类之间的耦合越弱，越有利于复用，一个处于弱耦合的类被修改，不会对有关系的类造成波及。信息的隐藏促进了软件的复用；

广义的迪米特法则在类的设计上的体现：

    优先考虑将一个类设置成不变类。

    尽量降低一个类的访问权限。

    谨慎使用Serializable。

    尽量降低成员的访问权限。

**6.开放-封闭原则（Open Closed Principle）**

定义：软件实体（类，模块，函数等等）应该可以扩展，但是不可以修改；

    对扩展开放，意味着有新的需求或变化时，可以对现有代码进行扩展，以适应新的情况。

    对修改封闭，意味着类一旦设计完成，就可以独立完成其工作，而不要对类进行任何修改。

    这样的设计，能够面对需求改变却可以保持相对稳定，从而使系统在第一个版本以后不断推出新的版本；面对需求，对程序的改动是通过增加新的代码进行的，而不是更改现有的代码；

    开放封闭原则，是最为重要的设计原则，Liskov替换原则和合成/聚合复用原则为开放封闭原则的实现提供保证。

### 三种类型

设计模式分为那几类，它们是怎么区分的，每一种模式类型的特点，包含具体模式呢？

设计模式按照目的来分，可以分为创建型模式、结构型模式和行为型模式。

**1.创建型**

创建型模式用来处理对象的创建过程，主要包含以下5种设计模式：

工厂方法模式（Factory Method Pattern）

抽象工厂模式（Abstract Factory Pattern）

建造者模式（Builder Pattern）

原型模式（Prototype Pattern）

单例模式（Singleton Pattern）

**2.结构型**

结构型模式用来处理类或者对象的组合，主要包含以下7种设计模式：

适配器模式（Adapter Pattern）

桥接模式（Bridge Pattern）

组合模式（Composite Pattern）

装饰者模式（Decorator Pattern）

外观模式（Facade Pattern）

享元模式（Flyweight Pattern）

代理模式（Proxy Pattern）

**3.行为型**

行为型模式用来对类或对象怎样交互和怎样分配职责进行描述，主要包含以下11种设计模式：

责任链模式（Chain of Responsibility Pattern）

命令模式（Command Pattern）

解释器模式（Interpreter Pattern）

迭代器模式（Iterator Pattern）

中介者模式（Mediator Pattern）

备忘录模式（Memento Pattern）

观察者模式（Observer Pattern）

状态模式（State Pattern）

策略模式（Strategy Pattern）

模板方法模式（Template Method Pattern）

访问者模式（Visitor Pattern）

---------------------

作者：小姐不得无礼

来源：CSDN

原文：https://blog.csdn.net/rqf520/article/details/79071871

版权声明：本文为博主原创文章，转载请附上博文链接！

**【单例模式 工厂模式   策略模式  适配器模式  生产者消费者模式 观察者模式】**

### ****单例模式 Singleton****

**如果一个类始终只能创建一个实例，则这个类被称为单例类，这种模式就被称为单例模式。**

**一般建议单例模式的方法命名为：getInstance()，可以有参数，一般无参数。**

**优点：**

1)减少创建[Java](http://lib.csdn.net/base/javase)实例所带来的系统开销

2)便于系统跟踪单个Java实例的生命周期、实例状态等。

### ****简单工厂模式 StaticFactory Method****

**由一个工厂对象决定创建出哪一种产品类的实例。**

**优点：**

**让对象的调用者和对象创建过程分离，当对象调用者需要对象时，直接向工厂请求即可。**

### ****工厂模式 Factory Method****

工厂模式（Factory Pattern）是 Java 中最常用的设计模式之一。这种类型的设计模式属于创建型模式，它提供了一种创建对象的最佳方式。

在工厂模式中，我们在创建对象时不会对客户端暴露创建逻辑，并且是通过使用一个共同的接口来指向新创建的对象。

#### 介绍

**意图：**定义一个创建对象的接口，让其子类自己决定实例化哪一个工厂类，工厂模式使其创建过程延迟到子类进行。

**主要解决：**主要解决接口选择的问题。

**何时使用：**我们明确地计划不同条件下创建不同实例时。

**如何解决：**让其子类实现工厂接口，返回的也是一个抽象的产品。

**关键代码：**创建过程在其子类执行。

**应用实例：** 1、您需要一辆汽车，可以直接从工厂里面提货，而不用去管这辆汽车是怎么做出来的，以及这个汽车里面的具体实现。 2、Hibernate 换数据库只需换方言和驱动就可以。

**优点：** 1、一个调用者想创建一个对象，只要知道其名称就可以了。 2、扩展性高，如果想增加一个产品，只要扩展一个工厂类就可以。 3、屏蔽产品的具体实现，调用者只关心产品的接口。

**缺点：**每次增加一个产品时，都需要增加一个具体类和对象实现工厂，使得系统中类的个数成倍增加，在一定程度上增加了系统的复杂度，同时也增加了系统具体类的依赖。这并不是什么好事。

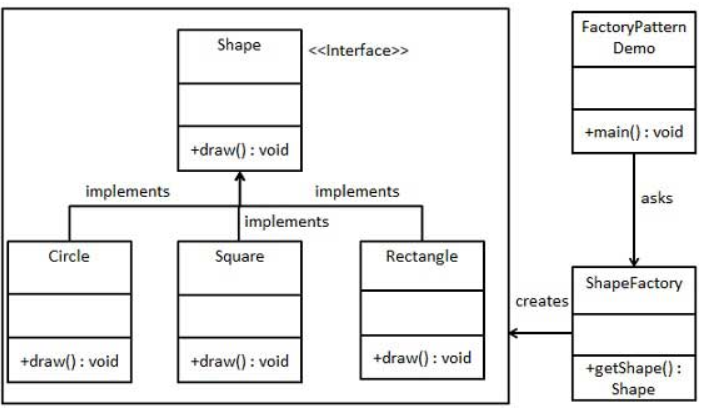
**使用场景：** 1、日志记录器：记录可能记录到本地硬盘、系统事件、远程服务器等，用户可以选择记录日志到什么地方。 2、数据库访问，当用户不知道最后系统采用哪一类数据库，以及数据库可能有变化时。 3、设计一个连接服务器的框架，需要三个协议，"POP3"、"IMAP"、"HTTP"，可以把这三个作为产品类，共同实现一个接口。

**注意事项：**作为一种创建类模式，在任何需要生成复杂对象的地方，都可以使用工厂方法模式。有一点需要注意的地方就是复杂对象适合使用工厂模式，而简单对象，特别是只需要通过 new 就可以完成创建的对象，无需使用工厂模式。如果使用工厂模式，就需要引入一个工厂类，会增加系统的复杂度。

#### 实现

我们将创建一个 *Shape* 接口和实现 *Shape* 接口的实体类。下一步是定义工厂类 *ShapeFactory*。

*FactoryPatternDemo*，我们的演示类使用 *ShapeFactory* 来获取 *Shape* 对象。它将向 *ShapeFactory* 传递信息（*CIRCLE / RECTANGLE / SQUARE*），以便获取它所需对象的类型。



**步骤 1**

创建一个接口:

public interface Shape { void draw(); }

**步骤 2**

创建实现接口的实体类。

Rectangle.java

public class Rectangle implements Shape {  
  
 @Override  
 public void draw() {  
 System.*out*.println("Inside Rectangle::draw() method.");  
 }  
}

Square.java

public class Square implements Shape {  
  
 @Override  
 public void draw() {  
 System.*out*.println("Inside Square::draw() method.");  
 }  
}

Circle.java

public class Circle implements Shape {  
  
 @Override  
 public void draw() {  
 System.*out*.println("Inside Circle::draw() method.");  
 }  
}

**步骤 3**

创建一个工厂，生成基于给定信息的实体类的对象。

ShapeFactory.java

public class ShapeFactory {  
  
 //使用 getShape 方法获取形状类型的对象  
 public Shape getShape(String shapeType){  
 if(shapeType == null){  
 return null;  
 }  
 if(shapeType.equalsIgnoreCase("CIRCLE")){  
 return new Circle();  
 } else if(shapeType.equalsIgnoreCase("RECTANGLE")){  
 return new Rectangle();  
 } else if(shapeType.equalsIgnoreCase("SQUARE")){  
 return new Square();  
 }  
 return null;  
 }  
}

**步骤 4**

使用该工厂，通过传递类型信息来获取实体类的对象。

FactoryPatternDemo.java

public class FactoryPatternDemo {  
  
 public static void main(String[] args) {  
 ShapeFactory shapeFactory = new ShapeFactory();  
  
 //获取 Circle 的对象，并调用它的 draw 方法  
 Shape shape1 = shapeFactory.getShape("CIRCLE");  
  
 //调用 Circle 的 draw 方法  
 shape1.draw();  
  
 //获取 Rectangle 的对象，并调用它的 draw 方法  
 Shape shape2 = shapeFactory.getShape("RECTANGLE");  
  
 //调用 Rectangle 的 draw 方法  
 shape2.draw();  
  
 //获取 Square 的对象，并调用它的 draw 方法  
 Shape shape3 = shapeFactory.getShape("SQUARE");  
  
 //调用 Square 的 draw 方法  
 shape3.draw();  
 }  
}

**步骤 5**

执行程序，输出结果：

Inside Circle::draw() method.

Inside Rectangle::draw() method.

Inside Square::draw() method.

### ****抽象工厂模式 Abstract Factory Method****

**为了解决客户端代码与不同工厂类耦合的问题。在工厂类的基础上再增加一个工厂类，该工厂类不制造具体的被调用对象，而是制造不同工厂对象。（建立一个生产工厂的工厂，把原来工厂模式的工厂作为方法）**

抽象工厂模式（Abstract Factory Pattern）是围绕一个超级工厂创建其他工厂。该超级工厂又称为其他工厂的工厂。这种类型的设计模式属于创建型模式，它提供了一种创建对象的最佳方式。

在抽象工厂模式中，接口是负责创建一个相关对象的工厂，不需要显式指定它们的类。每个生成的工厂都能按照工厂模式提供对象。

#### 介绍

**意图：**提供一个创建一系列相关或相互依赖对象的接口，而无需指定它们具体的类。

**主要解决：**主要解决接口选择的问题。

**何时使用：**系统的产品有多于一个的产品族，而系统只消费其中某一族的产品。

**如何解决：**在一个产品族里面，定义多个产品。

**关键代码：**在一个工厂里聚合多个同类产品。

**应用实例：**工作了，为了参加一些聚会，肯定有两套或多套衣服吧，比如说有商务装（成套，一系列具体产品）、时尚装（成套，一系列具体产品），甚至对于一个家庭来说，可能有商务女装、商务男装、时尚女装、时尚男装，这些也都是成套的，即一系列具体产品。假设一种情况（现实中是不存在的，要不然，没法进入共产主义了，但有利于说明抽象工厂模式），在您的家中，某一个衣柜（具体工厂）只能存放某一种这样的衣服（成套，一系列具体产品），每次拿这种成套的衣服时也自然要从这个衣柜中取出了。用 OO 的思想去理解，所有的衣柜（具体工厂）都是衣柜类的（抽象工厂）某一个，而每一件成套的衣服又包括具体的上衣（某一具体产品），裤子（某一具体产品），这些具体的上衣其实也都是上衣（抽象产品），具体的裤子也都是裤子（另一个抽象产品）。

**优点：**当一个产品族中的多个对象被设计成一起工作时，它能保证客户端始终只使用同一个产品族中的对象。

**缺点：**产品族扩展非常困难，要增加一个系列的某一产品，既要在抽象的 Creator 里加代码，又要在具体的里面加代码。

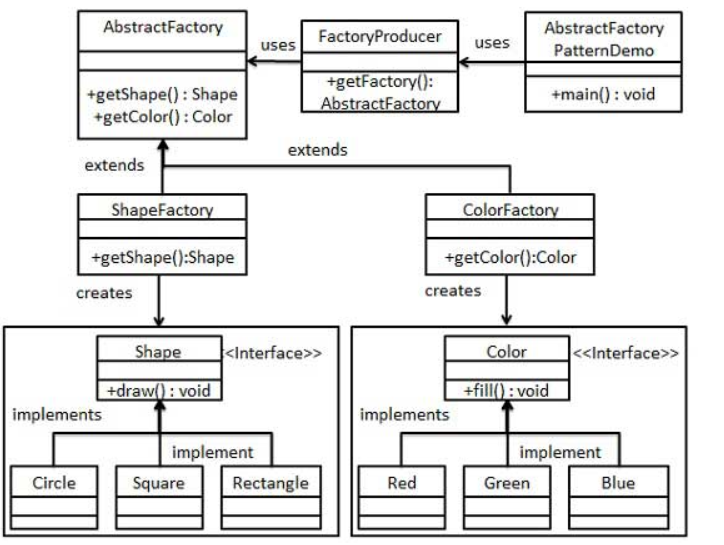
**使用场景：** 1、QQ 换皮肤，一整套一起换。 2、生成不同操作系统的程序。

**注意事项：**产品族难扩展，产品等级易扩展。

#### 实现

我们将创建 *Shape* 和 *Color* 接口和实现这些接口的实体类。下一步是创建抽象工厂类 *AbstractFactory*。接着定义工厂类 *ShapeFactory* 和 *ColorFactory*，这两个工厂类都是扩展了 *AbstractFactory*。然后创建一个工厂创造器/生成器类 *FactoryProducer*。

*AbstractFactoryPatternDemo*，我们的演示类使用 *FactoryProducer* 来获取 *AbstractFactory* 对象。它将向 *AbstractFactory* 传递形状信息 *Shape*（*CIRCLE / RECTANGLE / SQUARE*），以便获取它所需对象的类型。同时它还向 *AbstractFactory* 传递颜色信息 *Color*（*RED / GREEN / BLUE*），以便获取它所需对象的类型。



**步骤 1**

为形状创建一个接口。

Shape.java

public interface Shape { void draw(); }

**步骤 2**

创建实现接口的实体类。

Rectangle.java

public class Rectangle implements Shape {  
  
 @Override  
 public void draw() {  
 System.*out*.println("Inside Rectangle::draw() method.");  
 }  
}

Square.java

public class Square implements Shape {  
  
 @Override  
 public void draw() {  
 System.*out*.println("Inside Square::draw() method.");  
 }  
}

Circle.java

public class Circle implements Shape {  
  
 @Override  
 public void draw() {  
 System.*out*.println("Inside Circle::draw() method.");  
 }  
}  
步骤 3  
为颜色创建一个接口。  
  
Color.java  
public interface Color {  
 void fill();  
}  
步骤4  
创建实现接口的实体类。  
  
Red.java  
public class Red implements Color {  
  
 @Override  
 public void fill() {  
 System.*out*.println("Inside Red::fill() method.");  
 }  
}  
Green.java  
public class Green implements Color {  
  
 @Override  
 public void fill() {  
 System.*out*.println("Inside Green::fill() method.");  
 }  
}  
Blue.java  
public class Blue implements Color {  
  
 @Override  
 public void fill() {  
 System.*out*.println("Inside Blue::fill() method.");  
 }  
}  
步骤 5  
为 Color 和 Shape 对象创建抽象类来获取工厂。  
  
AbstractFactory.java  
public abstract class AbstractFactory {  
 public abstract Color getColor(String color);  
 public abstract Shape getShape(String shape) ;  
}  
步骤 6  
创建扩展了 AbstractFactory 的工厂类，基于给定的信息生成实体类的对象。  
  
ShapeFactory.java  
public class ShapeFactory extends AbstractFactory {  
  
 @Override  
 public Shape getShape(String shapeType){  
 if(shapeType == null){  
 return null;  
 }  
 if(shapeType.equalsIgnoreCase("CIRCLE")){  
 return new Circle();  
 } else if(shapeType.equalsIgnoreCase("RECTANGLE")){  
 return new Rectangle();  
 } else if(shapeType.equalsIgnoreCase("SQUARE")){  
 return new Square();  
 }  
 return null;  
 }  
  
 @Override  
 public Color getColor(String color) {  
 return null;  
 }  
}  
ColorFactory.java  
public class ColorFactory extends AbstractFactory {  
  
 @Override  
 public Shape getShape(String shapeType){  
 return null;  
 }  
  
 @Override  
 Color getColor(String color) {  
 if(color == null){  
 return null;  
 }  
 if(color.equalsIgnoreCase("RED")){  
 return new Red();  
 } else if(color.equalsIgnoreCase("GREEN")){  
 return new Green();  
 } else if(color.equalsIgnoreCase("BLUE")){  
 return new Blue();  
 }  
 return null;  
 }  
}  
步骤 7  
创建一个工厂创造器/生成器类，通过传递形状或颜色信息来获取工厂。  
  
FactoryProducer.java  
public class FactoryProducer {  
 public static AbstractFactory getFactory(String choice){  
 if(choice.equalsIgnoreCase("SHAPE")){  
 return new ShapeFactory();  
 } else if(choice.equalsIgnoreCase("COLOR")){  
 return new ColorFactory();  
 }  
 return null;  
 }  
}  
步骤 8  
使用 FactoryProducer 来获取 AbstractFactory，通过传递类型信息来获取实体类的对象。  
  
AbstractFactoryPatternDemo.java  
public class AbstractFactoryPatternDemo {  
 public static void main(String[] args) {  
  
 //获取形状工厂  
 AbstractFactory shapeFactory = FactoryProducer.getFactory("SHAPE");  
  
 //获取形状为 Circle 的对象  
 Shape shape1 = shapeFactory.getShape("CIRCLE");  
  
 //调用 Circle 的 draw 方法  
 shape1.draw();  
  
 //获取形状为 Rectangle 的对象  
 Shape shape2 = shapeFactory.getShape("RECTANGLE");  
  
 //调用 Rectangle 的 draw 方法  
 shape2.draw();  
  
 //获取形状为 Square 的对象  
 Shape shape3 = shapeFactory.getShape("SQUARE");  
  
 //调用 Square 的 draw 方法  
 shape3.draw();  
  
 //获取颜色工厂  
 AbstractFactory colorFactory = FactoryProducer.getFactory("COLOR");  
  
 //获取颜色为 Red 的对象  
 Color color1 = colorFactory.getColor("RED");  
  
 //调用 Red 的 fill 方法  
 color1.fill();  
  
 //获取颜色为 Green 的对象  
 Color color2 = colorFactory.getColor("Green");  
  
 //调用 Green 的 fill 方法  
 color2.fill();  
  
 //获取颜色为 Blue 的对象  
 Color color3 = colorFactory.getColor("BLUE");  
  
 //调用 Blue 的 fill 方法  
 color3.fill();  
 }  
}  
步骤 9  
执行程序，输出结果：  
  
Inside Circle::draw() method.  
Inside Rectangle::draw() method.  
Inside Square::draw() method.  
Inside Red::fill() method.  
Inside Green::fill() method.  
Inside Blue::fill() method.

### ****代理模式 Proxy Method****

**当客户端代码需要调用某个对象时，客户端实际上不关心是否准确得到该对象，它只要一个能提供该功能的对象即可，此时我们就可返回该对象的代理（Proxy）。**

### ****命令模式 Command Method****

**某个方法需要完成某一个功能，完成这个功能的大部分步骤已经确定了，但可能有少量具体步骤无法确定，必须等到执行该方法时才可以确定。在Java中，传入该方法的是一个对象，该对象通常是某个接口的匿名实现类的实例，该接口通常被称为命令接口，这种设计方式也被称为命令模式。**

### ****策略模式 Strategy Method****

**策略模式用于封装系列的**[算法](http://lib.csdn.net/base/datastructure)**，这些算法通常被封装在一个被称为Context的类中，客户端程序可以自由选择其中一种算法，或让Context为客户端选择一种最佳算法——使用策略模式的优势是为了支持算法的自由切换。**

**可以考虑使用配置文件来指定用户使用哪种具体算法——这就彻底分离客户端代码和具体算法策略。**

### ****观察者模式 Observer Method****

**观察者模式定义了对象间的一对多依赖关系，让一个或多个观察者对象观察一个主题对象。当主题对象的状态发生变化时，系统能通知所有的依赖于此对象的观察者对象，从而使得观察者对象能够自动更新。**

**在观察者模式中，被观察的对象常常也被称为目标或主题（Subject），依赖的对象被称为观察者（Observer）。**

当对象间存在一对多关系时，则使用观察者模式（Observer Pattern）。比如，当一个对象被修改时，则会自动通知它的依赖对象。观察者模式属于行为型模式。

#### 介绍

**意图：**定义对象间的一种一对多的依赖关系，当一个对象的状态发生改变时，所有依赖于它的对象都得到通知并被自动更新。

**主要解决：**一个对象状态改变给其他对象通知的问题，而且要考虑到易用和低耦合，保证高度的协作。

**何时使用：**一个对象（目标对象）的状态发生改变，所有的依赖对象（观察者对象）都将得到通知，进行广播通知。

**如何解决：**使用面向对象技术，可以将这种依赖关系弱化。

**关键代码：**在抽象类里有一个 ArrayList 存放观察者们。

**应用实例：** 1、拍卖的时候，拍卖师观察最高标价，然后通知给其他竞价者竞价。 2、西游记里面悟空请求菩萨降服红孩儿，菩萨洒了一地水招来一个老乌龟，这个乌龟就是观察者，他观察菩萨洒水这个动作。

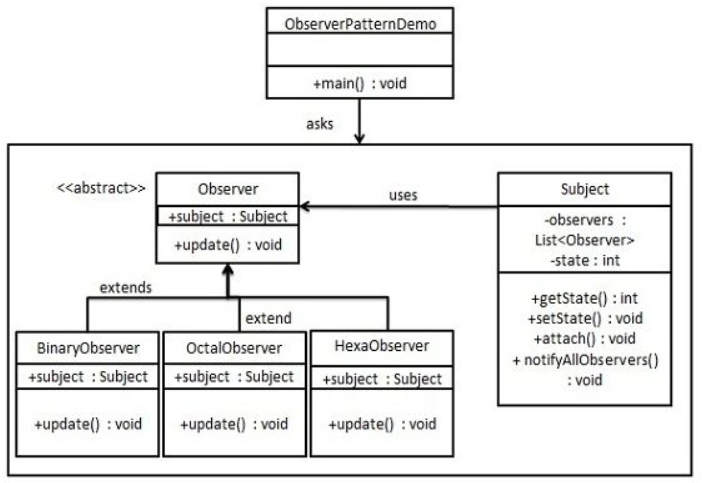
**优点：** 1、观察者和被观察者是抽象耦合的。 2、建立一套触发机制。

**缺点：** 1、如果一个被观察者对象有很多的直接和间接的观察者的话，将所有的观察者都通知到会花费很多时间。 2、如果在观察者和观察目标之间有循环依赖的话，观察目标会触发它们之间进行循环调用，可能导致系统崩溃。 3、观察者模式没有相应的机制让观察者知道所观察的目标对象是怎么发生变化的，而仅仅只是知道观察目标发生了变化。

**使用场景：**

* 一个抽象模型有两个方面，其中一个方面依赖于另一个方面。将这些方面封装在独立的对象中使它们可以各自独立地改变和复用。
* 一个对象的改变将导致其他一个或多个对象也发生改变，而不知道具体有多少对象将发生改变，可以降低对象之间的耦合度。
* 一个对象必须通知其他对象，而并不知道这些对象是谁。
* 需要在系统中创建一个触发链，A对象的行为将影响B对象，B对象的行为将影响C对象……，可以使用观察者模式创建一种链式触发机制。

**注意事项：** 1、JAVA 中已经有了对观察者模式的支持类。 2、避免循环引用。 3、如果顺序执行，某一观察者错误会导致系统卡壳，一般采用异步方式。



步骤 1  
 创建 Subject 类。  
  
 Subject.java  
import java.util.ArrayList;  
import java.util.List;  
  
 public class Subject {  
  
 private List<Observer> observers  
 = new ArrayList<Observer>();  
 private int state;  
  
 public int getState() {  
 return state;  
 }  
  
 public void setState(int state) {  
 this.state = state;  
 notifyAllObservers();  
 }  
  
 public void attach(Observer observer){  
 observers.add(observer);  
 }  
  
 public void notifyAllObservers(){  
 for (Observer observer : observers) {  
 observer.update();  
 }  
 }  
 }  
 步骤 2  
 创建 Observer 类。  
  
 Observer.java  
 public abstract class Observer {  
 protected Subject subject;  
 public abstract void update();  
 }  
 步骤 3  
 创建实体观察者类。  
  
 BinaryObserver.java  
 public class BinaryObserver extends Observer{  
  
 public BinaryObserver(Subject subject){  
 this.subject = subject;  
 this.subject.attach(this);  
 }  
  
 @Override  
 public void update() {  
 System.*out*.println( "Binary String: "  
 + Integer.*toBinaryString*( subject.getState() ) );  
 }  
 }  
 OctalObserver.java  
 public class OctalObserver extends Observer{  
  
 public OctalObserver(Subject subject){  
 this.subject = subject;  
 this.subject.attach(this);  
 }  
  
 @Override  
 public void update() {  
 System.*out*.println( "Octal String: "  
 + Integer.*toOctalString*( subject.getState() ) );  
 }  
 }  
 HexaObserver.java  
 public class HexaObserver extends Observer{  
  
 public HexaObserver(Subject subject){  
 this.subject = subject;  
 this.subject.attach(this);  
 }  
  
 @Override  
 public void update() {  
 System.*out*.println( "Hex String: "  
 + Integer.*toHexString*( subject.getState() ).toUpperCase() );  
 }  
 }  
 步骤 4  
 使用 Subject 和实体观察者对象。  
  
 ObserverPatternDemo.java  
 public class ObserverPatternDemo {  
 public static void main(String[] args) {  
 Subject subject = new Subject();  
  
 new HexaObserver(subject);  
 new OctalObserver(subject);  
 new BinaryObserver(subject);  
  
 System.*out*.println("First state change: 15");  
 subject.setState(15);  
 System.*out*.println("Second state change: 10");  
 subject.setState(10);  
 }  
 }  
 步骤 5  
 执行程序，输出结果：  
  
 First state change: 15  
 Hex String: F  
 Octal String: 17  
 Binary String: 1111  
 Second state change: 10  
 Hex String: A  
 Octal String: 12  
 Binary String: 1010

#### 实现

观察者模式使用三个类 Subject、Observer 和 Client。Subject 对象带有绑定观察者到 Client 对象和从 Client 对象解绑观察者的方法。我们创建 *Subject* 类、*Observer* 抽象类和扩展了抽象类 *Observer* 的实体类。

*ObserverPatternDemo*，我们的演示类使用 *Subject* 和实体类对象来演示观察者模式。

### ****生产者消费者模式 ProducerConsumer Method****

**生产者和消费者在同一时间段内共用同一存储空间，生产者向空间（仓库/缓冲区）里生产数据，而消费者取走数据。**

**优点：**

1)解耦

2)支持并发

生产者与消费者是两个独立的并发体，他们之间是用缓冲区作为桥梁连接。

3)支持忙闲不均

当数据制造快的时候，消费者来不及处理，未处理的数据可以暂时存在缓冲区中。 等生产者的制造速度慢下来，消费者再慢慢处理掉。

--------------------- 本文来自 cyhaolihai 的CSDN 博客 ，全文地址请点击：<https://blog.csdn.net/cyhaolihai/article/details/82902368?utm_source=copy>

## 输入输出流

### 概述

Java语言定义了许多类专门负责各种方式的输入或者输出，这些类都被放在java.io包中。其中，

所有输入流类都是抽象类InputStream(字节输入流)，或者抽象类Reader(字符输入流)的子类；

而所有输出流都是抽象类OutputStream(字节输出流)或者Writer(字符输出流)的子类。

【首先需要明白的是：**流是干什么的？？？**（为了永久性的保存数据）

根据数据流向的不同分为输入流和输出流；

根据处理数据类型的不同分为字符流和字节流；】

【然后需要明白的是**输入模式和输出模式是谁流向谁：**

InputStream(字节输入流)和Reader(字符输入流)通俗的理解都是读（read）的。

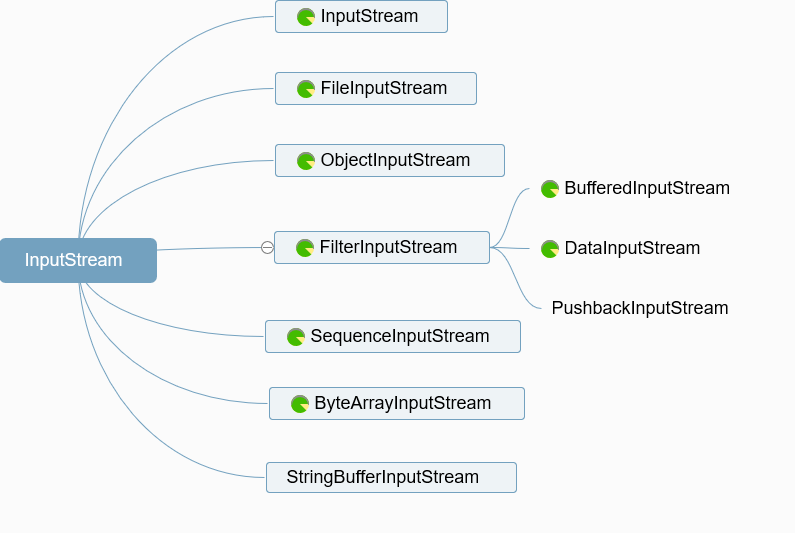
OutputStream(字节输出流)和Writer(字符输出流)通俗的理解都是写(writer)的。

】

最后下面搞清楚**各种流的类型的该怎么用，谁包含谁，理清思路。**

### InputStream类

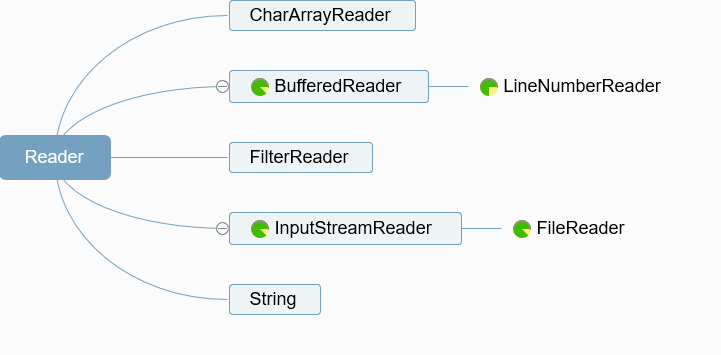
字节输入流的抽象类，是所有字节输入流的父类，InputStream类具有层次结构如下图所示；



java中的字符是Unicode编码的，是双字节的。InputStream是用来处理字节的，在处理字符文本时很不方便。Java为字符文本的输入提供了专门的一套类Reader。

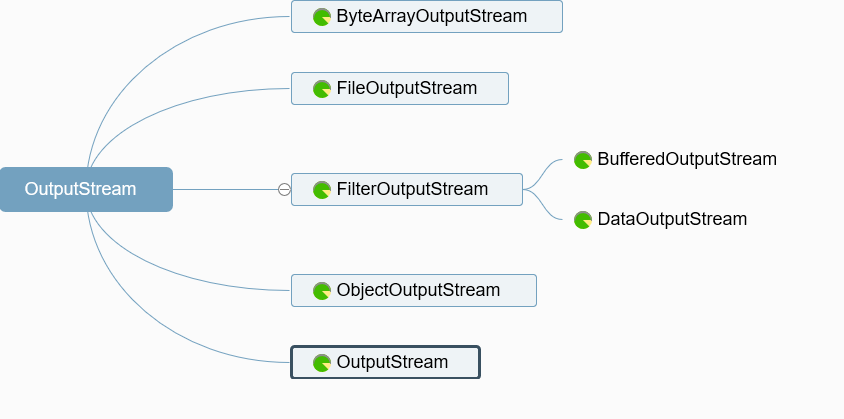
### Reader类

是字符输入流的抽象类，所有字符输入流的实现都是它的子类。



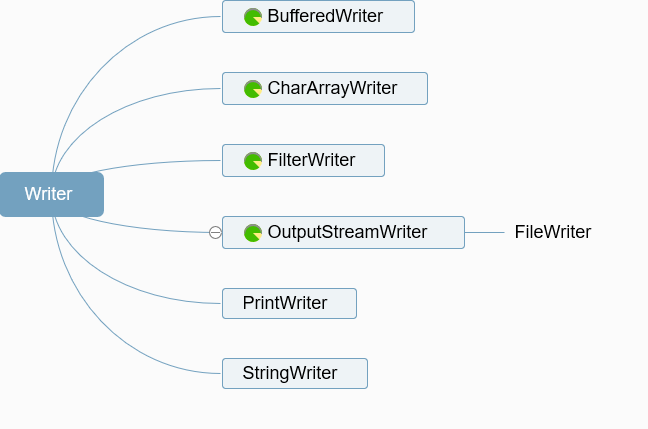
### OutputStream类

输出流OutputStream类是字节输入流的抽象类，此抽象类表示输出字节流的所有类的超类



### Writer类

是字符输出流的抽象类，所有字符输出类的实现都是它的子类。



### File类

是IO包中唯一代表磁盘文件本身的对象。通过File来创建，删除，重命名文件。File类对象的主要作用就是用来获取文本本身的一些信息。如文本的所在的目录，文件的长度，读写权限等等。（有的需要记忆，比如isFile(),isDirectory(),exits();有的了解即可。使用的时候查看API）

详细如下：

File类(File类的概述和构造方法)

#### A:File类的概述

　　File更应该叫做一个路径

　　文件路径或者文件夹路径

　　路径分为绝对路径和相对路径

　　　　绝对路径是一个固定的路径,从盘符开始

　　　　相对路径相对于某个位置,在eclipse下是指当前项目下,在dos下

　　查看API指的是当前路径

　　文件和目录路径名的抽象表示形式

#### B:构造方法

　　File(String pathname)：根据一个路径得到File对象

　　File(String parent, String child):根据一个目录和一个子文件/目录得到File对象

　　File(File parent, String child):根据一个父File对象和一个子文件/目录得到File对象

File类(File类的创建功能)

#### 创建功能

　　　　public boolean createNewFile():创建文件 如果存在这样的文件，就不创建了

　　　　public boolean mkdir():创建文件夹 如果存在这样的文件夹，就不创建了

　　　　public boolean mkdirs():创建文件夹,如果父文件夹不存在，会帮你创建出来

（使用createNewFile()文件创建的时候不加.txt或者其他后缀也是文件，不是文件夹；使用mkdir()创建文件夹的时候，如果起的名字是比如aaa.txt也是文件夹不是文件；）

注意事项：

如果你创建文件或者文件夹忘了写盘符路径，那么，默认在项目路径下。

File类(File类的重命名和删除功能)

　　A:重命名和删除功能

　　　　public boolean renameTo(File dest):把文件重命名为指定的文件路径

　　　　public boolean delete():删除文件或者文件夹

　　B:重命名注意事项

　　　　如果路径名相同，就是改名。

　　　　如果路径名不同，就是改名并剪切。

　　C:删除注意事项：

　　　　Java中的删除不走回收站。

　　　　要删除一个文件夹，请注意该文件夹内不能包含文件或者文件夹

File类(File类的判断功能)

#### 判断功能

　　　　public boolean isDirectory():判断是否是目录

　　　　public boolean isFile():判断是否是文件

　　　　public boolean exists():判断是否存在

　　　　public boolean canRead():判断是否可读

　　　　public boolean canWrite():判断是否可写

　　　　public boolean isHidden():判断是否隐藏

File类(File类的获取功能)

#### 获取功能

　　　　public String getAbsolutePath()：获取绝对路径

　　　　public String getPath():获取路径

　　　　public String getName():获取名称

　　　　public long length():获取长度。字节数

　　　　public long lastModified():获取最后一次的修改时间，毫秒值

　　　　public String[] list():获取指定目录下的所有文件或者文件夹的名称数组

　　　　public File[] listFiles():获取指定目录下的所有文件或者文件夹的File数组

File类(文件名称过滤器的概述及使用)

#### 文件名称过滤器的概述

　　　　public String[] list(FilenameFilter filter)

　　　　public File[] listFiles(FileFilter filter)

package com.ningmeng;

import java.io.File;

public class Test {

public static void main(String[] args) throws Exception{

// TODO Auto-generated method stub

File file=new File("aa.txt");//文件默认就创建在你创建的项目下面，刷新即可看到

System.out.println(file.exists());//判断文件是否存在

file.createNewFile();//创建文件，不是文件夹

System.out.println(file.exists());//再次判断是否存在

System.out.println(file.getName());//获取文件的名字

System.out.println(file.getAbsolutePath());//获取文件的绝对路径

System.out.println(file.getPath());//获取文件的相对路径

System.out.println(file.getParent());//获取文件的父路径

System.out.println(file.canRead());//文件是否可读

System.out.println(file.canWrite());//文件是否可写

System.out.println(file.length());//文件的长度

System.out.println(file.lastModified());//文件最后一次修改的时间

System.out.println(file.isDirectory());//判断文件是否是一个目录

System.out.println(file.isHidden());//文件是否隐藏

System.out.println(file.isFile());//判断文件是否存在

}

}

public String[] list():获取指定目录下的所有文件或者文件夹的名称数组

　　public File[] listFiles():获取指定目录下的所有文件或者文件夹的File数组

list()获取某个目录下所有的文件或者文件夹：

package com.ningmeng;

import java.io.File;

public class FileTest {

public static void main(String[] args){

File file=new File("D:/");//指定文件目录

String[] str=file.list();//获取指定目录下的所有文件或者文件夹的名称数组

for(String s : str){//加强for循环遍历输出

System.out.println(s);

}

}

}

package com.ningmeng;

import java.io.File;

public class FileTest {

public static void main(String[] args){

File file=new File("D:/");//指定文件路径

File[] f=file.listFiles();//获取指定目录下的所有文件或者文件夹的File数组

for(File fi : f){//加强for循环遍历输出

System.out.println(fi);

}

}

}

案例演示：

获取某种格式的文件比如获取某种后缀的图片，并输出文件名：

package com.ningmeng;

import java.io.File;

public class FileTest {

public static void main(String[] args){

File file=new File("C:\\Users\\biehongli\\Pictures\\xuniji");

String[] str=file.list();

for(String s : str){

if(s.endsWith(".jpg") || s.endsWith(".png")){//如果后缀是这种格式的就输出

System.out.println(s);

}

}

}

}

下面演示获取文件夹下面子目录里面的文件获取（并没有完全获取子目录的子目录等等，仅仅获取了子一级目录）：

package com.ningmeng;

import java.io.File;

public class FileTest {

public static void main(String[] args){

File file=new File("C:\\Users\\biehongli\\Pictures\\Camera Roll");

File[] f=file.listFiles();

for(File fi : f){

if(fi.isDirectory()){//判断如果是一个目录

String[] s=fi.list();

for(String str : s){

if(str.endsWith(".jpg")){

System.out.println(str);

}

}

}

}

}

}

#### 文件名称过滤器的概述

　　　　public String[] list(FilenameFilter filter)

　　　　public File[] listFiles(FileFilter filter)

package com.ningmeng;

import java.io.File;

import java.io.FilenameFilter;

public class FileTest {

public static void main(String[] args){

File file=new File("C:\\Users\\biehongli\\Pictures\\Camera Roll");

String[] str=file.list(new FilenameFilter() {//过滤器，匿名内部类

@Override

public boolean accept(File dir, String name) {

// TODO Auto-generated method stub

//System.out.println(dir);//获取文件的路径

//System.out.println(name);//获取文件的名字

File f=new File(dir,name);

return f.isFile() && f.getName().endsWith(".jpg");

}

});

for(String s : str){

System.out.println(s);

}

}

}

### 下面以一些字节输入输出流具体的案例操作（操作的时候认清自己使用的是字节流还是字符流）：

注意：read()方法读取的是一个字节,为什么返回是int,而不是byte

字节输入流可以操作任意类型的文件,比如图片音频等,这些文件底层都是以二进制形式的存储的,如果每次读取都返回byte,有可能在读到中间的时候遇到111111111；那么这11111111是byte类型的-1,我们的程序是遇到-1就会停止不读了,后面的数据就读不到了,所以在读取的时候用int类型接收,如果11111111会在其前面补上；24个0凑足4个字节,那么byte类型的-1就变成int类型的255了这样可以保证整个数据读完,而结束标记的-1就是int类型

FileInputStream的单个字节读取：

FileOutputStream的单个字节写入：

package com.ningmeng;

import java.io.FileInputStream;

import java.io.FileOutputStream;

public class FileTest {

public static void main(String[] args) throws Exception{

FileInputStream fis=new FileInputStream("aaa.txt");

FileOutputStream fos=new FileOutputStream("bbb.txt",true);

//FileOutputStream()后面加true指文件后面可追加

int a=fis.read();//read()一次读取一个字节

System.out.println(a);//读取的一个字节输出

fos.write(101);//write()一次写一个字节

fis.close();//一定记得关闭流，养成好习惯

fos.close();

}

}

FileInputStream和FileOutputStream进行拷贝文本或者图片或者歌曲：

package com.ningmeng;

import java.io.FileInputStream;

import java.io.FileOutputStream;

public class FileTest {

public static void main(String[] args) throws Exception{

FileInputStream fis=new FileInputStream("aaa.txt");

FileOutputStream fos=new FileOutputStream("bbb.txt");

//如果没有bbb.txt,会创建出一个

int b;

while((b=fis.read())!=-1){

fos.write(b);

}

fis.close();

fos.close();

}

}

FileInputStream和FileOutputStream定义小数组进行读写操作：

package com.ningmeng;

import java.io.FileInputStream;

import java.io.FileOutputStream;

public class FileTest {

public static void main(String[] args) throws Exception{

FileInputStream fis = new FileInputStream("aaa.txt");

FileOutputStream fos = new FileOutputStream("bbb.txt");

int len;

byte[] arr = new byte[1024 \* 8];//自定义字节数组

while((len = fis.read(arr)) != -1) {

//fos.write(arr);

fos.write(arr, 0, len);//写出字节数组写出有效个字节个数

}

//IO流(定义小数组)

//write(byte[] b)

//write(byte[] b, int off, int len)写出有效的字节个数

fis.close();

fos.close();

}

}

### IO流(BufferedInputStream和BufferOutputStream拷贝)

\* A:缓冲思想

　　\* 字节流一次读写一个数组的速度明显比一次读写一个字节的速度快很多，

　　\* 这是加入了数组这样的缓冲区效果，java本身在设计的时候，

　　\* 也考虑到了这样的设计思想，所以提供了字节缓冲区流

\* B.BufferedInputStream

　　\* BufferedInputStream内置了一个缓冲区(数组)

　　\* 从BufferedInputStream中读取一个字节时

　　\* BufferedInputStream会一次性从文件中读取8192个, 存在缓冲区中, 返回给程序一个

　　\* 程序再次读取时, 就不用找文件了, 直接从缓冲区中获取

　　\* 直到缓冲区中所有的都被使用过, 才重新从文件中读取8192个

\* C.BufferedOutputStream

　　\* BufferedOutputStream也内置了一个缓冲区(数组)

　　\* 程序向流中写出字节时, 不会直接写到文件, 先写到缓冲区中

\* 直到缓冲区写满, BufferedOutputStream才会把缓冲区中的数据一次性写到文件里

package com.ningmeng;

import java.io.BufferedInputStream;

import java.io.BufferedOutputStream;

import java.io.FileInputStream;

import java.io.FileOutputStream;

public class FileTest {

public static void main(String[] args) throws Exception{

FileInputStream fis = new FileInputStream("aaa.txt");

FileOutputStream fos = new FileOutputStream("bbb.txt");

BufferedInputStream bis=new BufferedInputStream(fis);

//使用装饰模式，把fis装饰进去bis中。使用缓冲读取速度变快

BufferedOutputStream bos=new BufferedOutputStream(fos);

int b;

while((b=bis.read())!=-1){

bos.write(b);

}

bis.close();

bos.close();

}

}

面试题：小数组的读写和带Buffered的读取哪个更快?

　　\* 定义小数组如果是8192个字节大小和Buffered比较的话

　　\* 定义小数组会略胜一筹,因为读和写操作的是同一个数组

　　\* 而Buffered操作的是两个数组

IO流(flush和close方法的区别)

flush()方法： 用来刷新缓冲区的,刷新后可以再次写出（字节缓冲流内置缓冲区，如果没有读取出来，可以使用flush()刷新来）

close()方法：用来关闭流释放资源的的,如果是带缓冲区的流对象的close()方法,不但会关闭流,还会再关闭流之前刷新缓冲区,关闭后不能再写出

### 字符流FileReader和FileWriter

字符流是什么

　　\* 字符流是可以直接读写字符的IO流

　　\* 字符流读取字符, 就要先读取到字节数据, 然后转为字符. 如果要写出字符, 需要把字符转为字节再写出.

IO流(什么情况下使用字符流)

\* 字符流也可以拷贝文本文件, 但不推荐使用. 因为读取时会把字节转为字符, 写出时还要把字符转回字节.

\* 程序需要读取一段文本, 或者需要写出一段文本的时候可以使用字符流

\* 读取的时候是按照字符的大小读取的,不会出现半个中文

\* 写出的时候可以直接将字符串写出,不用转换为字节数组

IO流(字符流是否可以拷贝非纯文本的文件)

\* 不可以拷贝非纯文本的文件

\* 因为在读的时候会将字节转换为字符,在转换过程中,可能找不到对应的字符,就会用?代替,写出的时候会将字符转换成字节写出去

\* 如果是?,直接写出,这样写出之后的文件就乱了,看不了了

package com.ningmeng;

import java.io.FileReader;

import java.io.FileWriter;

public class FileTest {

public static void main(String[] args) throws Exception{

//FileReader类的read()方法可以按照字符大小读取

FileReader fr=new FileReader("aaa.txt");

int b;

while((b=fr.read())!=-1){

System.out.println((char)b);//int类型转为字符型

}

fr.close();

//FileWriter类的write()方法可以自动把字符转为字节写出

FileWriter fw = new FileWriter("aaa.txt",true);

fw.write("aaa");

fw.close();

//字符流的拷贝

FileReader fr2 = new FileReader("aaa.txt");

FileWriter fw2 = new FileWriter("bbb.txt");

int ch;

while((ch = fr2.read()) != -1) {

fw2.write(ch);

}

fr2.close();

fw2.close();

}

}

package com.ningmeng;

import java.io.BufferedReader;

import java.io.BufferedWriter;

import java.io.FileReader;

import java.io.FileWriter;

public class FileTest {

public static void main(String[] args) throws Exception{

BufferedReader br=new BufferedReader(new FileReader("aaa.txt"));

BufferedWriter bw=new BufferedWriter(new FileWriter("bbb.txt"));

//BufferedReader和BufferedWriter的使用：

int b;

while((b=br.read())!=-1){

bw.write((char)b);

}

br.close();

bw.close();

}

}

## Java内存模型

<https://www.cnblogs.com/andy-zcx/p/5522836.html>

### 简介

1. 堆

堆是所有线程共享的，主要用来存储对象。其中，堆可分为：年轻代和老年代两块区域。使用NewRatio参数来设定比例。对于年轻代，一个Eden区和两个Suvivor区，使用参数SuvivorRatio来设定大小；

1. Java虚拟机栈/本地方法栈

线程私有的，主要存放局部变量表，操作数栈，动态链接和方法出口等；

1. 程序计数器

同样是线程私有的，记录当前线程的行号指示器，为线程的切换提供保障；

1. 方法区

线程共享的，主要存储类信息、常量池、静态变量、JIT编译后的代码等数据。方法区理论上来说是堆的逻辑组成部分；

1. 运行时常量池

是方法区的一部分，用于存放编译期生成的各种字面量和符号引用；

### 堆、栈、方法区

JAVA的JVM的内存可分为3个区：堆(heap)、堆栈(stack)和方法区(method)

#### 堆区:

1. 提供所有类实例和数组对象存储区域。
2. jvm只有一个堆区(heap)被所有线程共享，堆中不存放基本类型和对象引用，只存放对象本身
3. 堆内存用于存放由new创建的对象和数组。在堆中分配的内存，由java虚拟机自动垃圾回收器来管理。在堆中产生了一个数组或者对象后，还可以在栈中定义一个特殊的变量，这个变量的取值等于数组或者对象在堆内存中的首地址，在栈中的这个特殊的变量就变成了数组或者对象的引用变量，以后就可以在程序中使用栈内存中的引用变量来访问堆中的数组或者对象，引用变量相当于为数组或者对象起的一个别名，或者代号。
4. 全局变量i，他是存放在java堆中。因为它不是静态的变量，不会独立于类的实例而存在，而该类实例化之后，放在堆中，当然也包含了它的属性i。

#### 栈区:

1. 每个线程包含一个栈区，栈中只保存基础数据类型的对象和自定义对象的引用(不是对象)，对象都存放在堆区中
2. 每个栈中的数据(原始类型和对象引用)都是私有的，其他栈不能访问。
3. 在函数中定义的一些基本类型的变量和对象的引用变量都是在函数的栈内存中分配。当在一段代码块中定义一个变量时，java就在栈中为这个变量分配内存空间，当超过变量的作用域后，java会自动释放掉为该变量分配的内存空间，该内存空间可以立刻被另作他用。
4. **虚拟机栈（JVM Stack）：**

一个线程的每个方法在执行的同时，都会创建一个栈帧（Statck Frame），栈帧中存储的有局部变量表、操作站、动态链接、方法出口等，当方法被调用时，栈帧在JVM栈中入栈，当方法执行完成时，栈帧出栈。

　　局部变量表中存储着方法的相关局部变量，包括各种基本数据类型，对象的引用，返回地址等。在局部变量表中，只有long和double类型会占用2个局部变量空间（Slot，对于32位机器，一个Slot就是32个bit），其它都是1个Slot。需要注意的是，局部变量表是在编译时就已经确定好的，方法运行所需要分配的空间在栈帧中是完全确定的，在方法的生命周期内都不会改变。

　　虚拟机栈中定义了两种异常，如果线程调用的栈深度大于虚拟机允许的最大深度，则抛出StatckOverFlowError（栈溢出）；不过多数Java虚拟机都允许动态扩展虚拟机栈的大小(有少部分是固定长度的)，所以线程可以一直申请栈，直到内存不足，此时，会抛出OutOfMemoryError（内存溢出）。

　　每个线程对应着一个虚拟机栈，因此虚拟机栈也是线程私有的。

1. **本地方法栈（Native Method Statck）：**

本地方法栈在作用，运行机制，异常类型等方面都与虚拟机栈相同，唯一的区别是：虚拟机栈是执行Java方法的，而本地方法栈是用来执行**native方法**的，在很多虚拟机中（如Sun的JDK默认的HotSpot虚拟机），会将本地方法栈与虚拟机栈放在一起使用。

　　本地方法栈也是线程私有的。

#### 堆和栈的联系

1. 引用变量是普通变量，定义时在栈中分配内存，引用变量在程序运行到作用域外释放。而数组＆对象本身在堆中分配，即使程序运行到使用new产生数组和对象的语句所在地代码块之外，数组和对象本身占用的堆内存也不会被释放，数组和对象在没有引用变量指向它的时候（比如先前的引用变量x=null时），才变成垃圾，不能再被使用，但是仍然占着内存，在随后的一个不确定的时间被垃圾回收器释放掉。这个也是java比较占内存的主要原因。

2. 总结起来就是对象存储在堆内存，引用变量存储在栈内存。栈内存指向堆内存。

#### 方法区:

1. 又叫静态区，跟堆一样，被所有的线程共享。方法区包含所有的class和static变量。
2. 方法区中包含的都是在整个程序中永远唯一的元素，如class，static变量。
3. 运行时常量池都分配在 Java 虚拟机的方法区之中
4. 更加详细一点的说法是方法区里存放着类的版本，字段，方法，接口和常量池。[常量池](#_常量池)里存储着字面量和符号引用。



--------------------- 本文来自 zhuliyuan丶 的CSDN 博客 ，全文地址请点击：<https://blog.csdn.net/zly921112/article/details/61192747?utm_source=copy>

**永久代和方法区的关系**

  涉及到内存模型时，往往会提到永久代，那么它和方法区又是什么关系呢？《Java虚拟机规范》只是规定了有方法区这么个概念和它的作用，并没有规定如何去实现它。那么，在不同的 JVM 上方法区的实现肯定是不同的了。 同时大多数用的JVM都是Sun公司的HotSpot。在HotSpot上把GC分代收集扩展至方法区，或者说**使用永久代来实现方法区**。因此，我们得到了结论，永久代是HotSpot的概念，方法区是Java虚拟机规范中的定义，是一种规范，而永久代是一种实现，一个是标准一个是实现。其他的虚拟机实现并没有永久带这一说法。在1.7之前在(JDK1.2 ~ JDK6)的实现中，HotSpot 使用永久代实现方法区，HotSpot 使用 GC分代来实现方法区内存回收，可以使用如下参数来调节方法区的大小:

-XX:PermSize

方法区初始大小

-XX:MaxPermSize

方法区最大大小

超过这个值将会抛出OutOfMemoryError异常:java.lang.OutOfMemoryError: PermGen

**元空间**

  对于Java8， HotSpots取消了永久代，那么是不是也就没有方法区了呢？当然不是，方法区是一个规范，规范没变，它就一直在。那么取代永久代的就是元空间。它可永久代有什么不同的？存储位置不同，永久代物理是是堆的一部分，和新生代，老年代地址是连续的，而元空间属于**本地内存，不在java虚拟机中的**；存储内容不同，元空间存储类的元信息，静态变量和常量池等并入堆中。相当于永久代的数据被分到了堆和元空间中。

--------------------- 本文来自 抽离的心 的CSDN 博客 ，全文地址请点击：https://blog.csdn.net/u011635492/article/details/81046174?utm\_source=copy

### 常量池

在Java的内存分配中，总共3种常量池：

#### ****1.字符串常量池(String Constant Pool):****

##### ****1.1:字符串常量池在Java内存区域的哪个位置？****

* 在JDK6.0及之前版本，字符串常量池是放在Perm Gen区(也就是方法区)中；
* 在JDK7.0版本，字符串常量池被移到了堆中了。至于为什么移到堆内，大概是由于方法区的内存空间太小了。

##### ****1.2:字符串常量池是什么？****

* 在HotSpot VM里实现的string pool功能的是一个StringTable类，它是一个Hash表，默认值大小长度是1009；这个StringTable在每个HotSpot VM的实例只有一份，被所有的类共享。字符串常量由一个一个字符组成，放在了StringTable上。
* 在JDK6.0中，StringTable的长度是固定的，长度就是1009，因此如果放入String Pool中的String非常多，就会造成hash冲突，导致链表过长，当调用String#intern()时会需要到链表上一个一个找，从而导致性能大幅度下降；
* 在JDK7.0中，StringTable的长度可以通过参数指定：

-XX:StringTableSize=66666

* 1

##### ****1.3:字符串常量池里放的是什么？****

* 在JDK6.0及之前版本中，String Pool里放的都是字符串常量；
* 在JDK7.0中，由于String#intern()发生了改变，因此String Pool中也可以存放放于堆内的字符串对象的引用。关于String在内存中的存储和String#intern()方法的说明，可以参考我的另外一篇博客：

需要说明的是：字符串常量池中的字符串只存在一份！   
如：

String s1 = "hello,world!";

String s2 = "hello,world!";

即执行完第一行代码后，常量池中已存在 “hello,world!”，那么 s2不会在常量池中申请新的空间，而是直接把已存在的字符串内存地址返回给s2。(这里具体的字符串如何分配就不细说了，可以看我的另一篇博客)

字符串常量池也是运行时常量池一部分

#### ****2.class常量池(Class Constant Pool)：****

##### ****2.1:class常量池简介：****

* 我们写的每一个Java类被编译后，就会形成一份class文件；class文件中除了包含类的版本、字段、方法、接口等描述信息外，还有一项信息就是常量池(constant pool table)，用于存放编译器生成的各种字面量(Literal)和符号引用(Symbolic References)；
* 每个class文件都有一个class常量池。

##### ****2.2:什么是字面量和符号引用：****

* 字面量包括：1.文本字符串 2.八种基本类型的值 3.被声明为final的常量等;
* 符号引用包括：1.类和方法的全限定名 2.字段的名称和描述符 3.方法的名称和描述符。

#### ****3.运行时常量池(Runtime Constant Pool)：****

* 运行时常量池存在于内存中，**是方法区的一部分**，也就是class常量池被加载到内存之后的版本，不同之处是：它的字面量可以动态的添加(String#intern()),符号引用可以被解析为直接引用
* JVM在执行某个类的时候，必须经过加载、连接、初始化，而连接又包括验证、准备、解析三个阶段。而当类加载到内存中后，jvm就会将class常量池中的内容存放到运行时常量池中，由此可知，运行时常量池也是每个类都有一个。在解析阶段，会把符号引用替换为直接引用，解析的过程会去查询字符串常量池，也就是我们上面所说的StringTable，以保证运行时常量池所引用的字符串与字符串常量池中是一致的。

--------------------- 本文来自 zhuminChosen 的CSDN 博客 ，全文地址请点击：https://blog.csdn.net/zm13007310400/article/details/77534349?utm\_source=copy

### 分代

#### **年轻代（Young Generation）**

1.所有新生成的对象首先都是放在年轻代的。年轻代的目标就是尽可能快速的收集掉那些生命周期短的对象。

2.新生代内存按照8:1:1的比例分为一个eden区和两个survivor(survivor0,survivor1)区。一个Eden区，两个 Survivor区(一般而言)。大部分对象在eden区中生成。回收时先将eden区存活对象复制到一个survivor0区，然后清空eden区，当这个survivor0区也存放满了时，则将eden区和survivor0区存活对象复制到另一个survivor1区，然后清空eden和这个survivor0区，此时survivor0区是空的，然后将survivor0区和survivor1区交换，即保持survivor1区为空， 如此往复。

3.当survivor1区不足以存放 eden和survivor0的存活对象时，就将存活对象直接存放到老年代。若是老年代也满了就会触发一次Full GC，也就是新生代、老年代都进行回收

4.新生代发生的GC也叫做Minor GC，MinorGC发生频率比较高(不一定等Eden区满了才触发)

#### **年老代（Old Generation）**

1.在年轻代中经历了N次垃圾回收后仍然存活的对象，就会被放到年老代中。因此，可以认为年老代中存放的都是一些生命周期较长的对象。

2.内存比新生代也大很多(大概比例是1:2)，当老年代内存满时触发Major GC即Full GC，Full GC发生频率比较低，老年代对象存活时间比较长，存活率标记高。

#### **持久代（Permanent Generation）**

用于存放静态文件，如Java类、方法等。持久代对垃圾回收没有显著影响，但是有些应用可能动态生成或者调用一些class，例如Hibernate 等，在这种时候需要设置一个比较大的持久代空间来存放这些运行过程中新增的类。

**注：**jdk1.8移除了永久代, 链接<http://openjdk.java.net/projects/jdk8/milestones>. 其中 “[JEP 122: Remove the Permanent Generation](http://openjdk.java.net/jeps/122)“说的就是移除永久代.

说明：类的元数据, 字符串池, 类的静态变量将会从永久代移除, 放入Java heap或者native memory. 其中建议JVM的实现中将类的元数据放入 native memory, 将字符串池和类的静态变量放入java堆中. 这样可以加载多少类的元数据就不在由MaxPermSize控制, 而由系统的实际可用空间来控制.

为什么这么做呢? 减少OOM只是表因, 更深层的原因还是要合并HotSpot和JRockit的代码, JRockit从来没有一个叫永久代的东西, 但是运行良好, 也不需要开发运维人员设置这么一个永久代的大小.

当然不用担心运行性能问题了, 在覆盖到的测试中, 程序启动和运行速度降低不超过1%, 但是这一点性能损失换来了更大的安全保障.

--------------------- 本文来自 老伯⭐️ 的CSDN 博客 ，全文地址请点击：<https://blog.csdn.net/afterlife_qiye/article/details/52506976?utm_source=copy>

### 垃圾回收

#### Java垃圾回收概况

　　Java GC（Garbage Collection，垃圾回收）机制，是Java与C++/C的主要区别之一，作为Java开发者，一般不需要专门编写内存回收和垃圾清理代码，对内存泄露和溢出的问题，也不需要像C程序员那样战战兢兢。这是因为在Java虚拟机中，存在自动内存管理和垃圾清扫机制。概括地说，该机制对JVM中的内存进行标记，并确定哪些内存需要回收，根据一定的回收策略，自动的回收内存，永不停息的保证JVM中的内存空间，防止出现内存泄露和溢出问题。

　　关于JVM，需要说明一下的是，目前使用最多的Sun公司的JDK中，自从1999年的JDK1.2开始直至现在仍在广泛使用的JDK6，其中默认的虚拟机都是HotSpot。2009年，Oracle收购Sun，加上之前收购的EBA公司，Oracle拥有3大虚拟机中的两个：JRockit和HotSpot，Oracle也表明了想要整合两大虚拟机的意图，但是目前在新发布的JDK8中，默认的虚拟机仍然是HotSpot，因此本文中默认介绍的虚拟机都是HotSpot，相关机制也主要是指HotSpot的GC机制。

Java GC机制主要完成3件事：

确定哪些内存需要回收  
确定什么时候需要执行GC  
如何执行GC

经过这么长时间的发展，Java GC机制已经日臻完善，几乎可以自动的为我们做绝大多数的事情。然而，如果我们从事较大型的应用软件开发，曾经出现过内存优化的需求，就必定要研究Java GC机制。

　　学习Java GC机制，可以帮助我们在日常工作中排查各种内存溢出或泄露问题，解决性能瓶颈，达到更高的并发量，写出更高效的程序。

--------------------- 本文来自 叩丁狼教育 的CSDN 博客 ，全文地址请点击：https://blog.csdn.net/wolfcode\_cn/article/details/81626108?utm\_source=copy

#### 算法

Java语言规范没有明确地说明JVM使用哪种垃圾回收算法，但是任何一种垃圾回收算法一般要做2件基本的事情：（1）发现无用信息对象；（2）回收被无用对象占用的内存空间，使该空间可被程序再次使用。

##### ****引用计数法(Reference Counting Collector)****

1.1算法分析

引用计数是垃圾收集器中的早期策略。在这种方法中，堆中每个对象实例都有一个引用计数。当一个对象被创建时，且将该对象实例分配给一个变量，该变量计数设置为1。当任何其它变量被赋值为这个对象的引用时，计数加1（a = b,则b引用的对象实例的计数器+1），但当一个对象实例的某个引用超过了生命周期或者被设置为一个新值时，对象实例的引用计数器减1。任何引用计数器为0的对象实例可以被当作垃圾收集。当一个对象实例被垃圾收集时，它引用的任何对象实例的引用计数器减1。

1.2优缺点

优点：

引用计数收集器可以很快的执行，交织在程序运行中。对程序需要不被长时间打断的实时环境比较有利。

缺点：

无法检测出循环引用。如父对象有一个对子对象的引用，子对象反过来引用父对象。这样，他们的引用计数永远不可能为0.

1.3引用计数算法无法解决循环引用问题，例如：

public class Main {

    public static void main(String[] args) {

        MyObject object1 = new MyObject();

        MyObject object2 = new MyObject();

        object1.object = object2;

        object2.object = object1;

        object1 = null;

        object2 = null;

    }

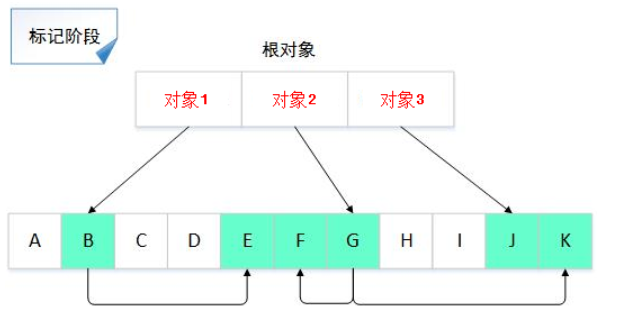
}

最后面两句将object1和object2赋值为null，也就是说object1和object2指向的对象已经不可能再被访问，但是由于它们互相引用对方，导致它们的引用计数器都不为0，那么垃圾收集器就永远不会回收它们。

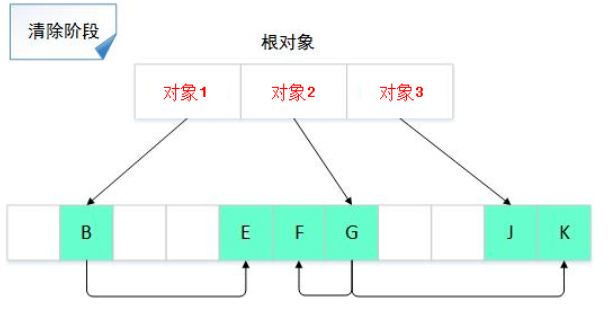
##### 最基础的收集算法 — 标记/清除算法

标记/清除算法的基本思想就跟它的名字一样，分为“标记”和“清除”两个阶段：首先标记出所有需要回收的对象，在标记完成后统一回收所有被标记的对象。

标记阶段：标记的过程其实就是检查哪些对象能被外界访问的可达性分析算法的过程，遍历所有的GC Roots对象，对从GC Roots对象可达的对象都打上一个标识，一般是在对象的header中，将其记录为可达对象。



清除阶段：清除的过程是对堆内存进行遍历，如果发现某个对象没有被标记为可达对象（通过读取对象header信息），则将其回收。



上面两张图是标记/清除算法的示意图，在标记阶段，从对象1可以访问到B对象，从B对象又可以访问到E对象，因此从GC 对象1到B、E都是可达的，同理，对象F、G、J、K都是可达对象；到了清除阶段，所有不可达对象都会被回收。

在垃圾收集器进行GC时，必须停止所有Java执行线程（也称”Stop The World”），原因是在标记阶段进行可达性分析时，不可以出现分析过程中对象引用关系还在不断变化的情况，否则的话可达性分析结果的准确性就无法得到保证。在等待标记清除结束后，应用线程才会恢复运行。

标记/清除算法存在问题：

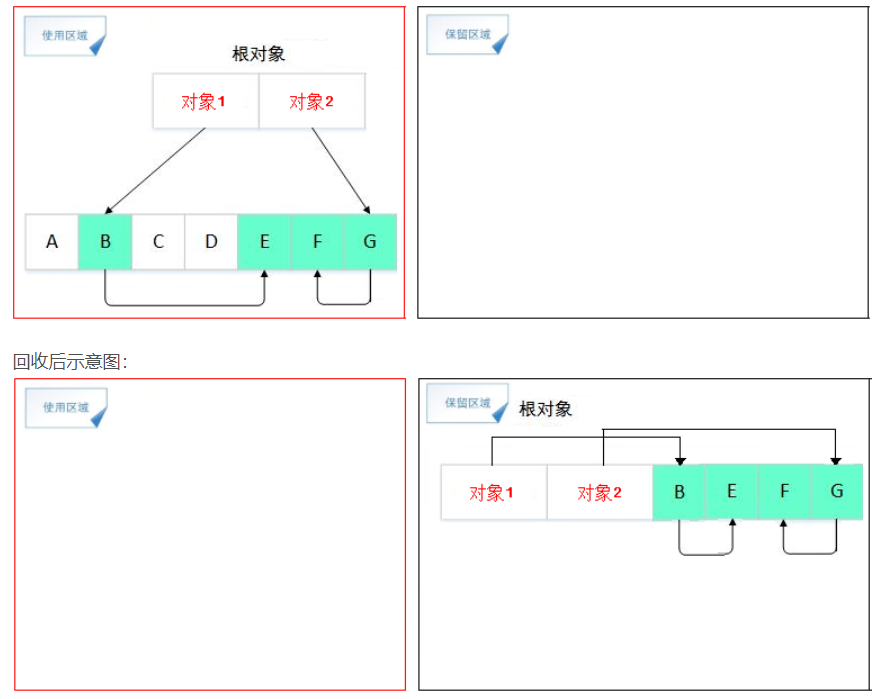
1、效率问题。标记和清除两个阶段的效率都不高，因为这两个阶段都需要遍历内存中的对象，很多时候内存中的对象实例数量是非常庞大的，这无疑很耗费时间，而且GC时需要停止应用程序，这会导致非常差的用户体验。

2、空间问题。标记清除之后会产生大量不连续的内存碎片（从上图可以看出），内存空间碎片太多可能会导致以后在程序运行过程中需要分配较大对象时，无法找到足够的连续内存而不得不提前触发另一次垃圾回收动作。

既然标记/清除算法有这么多的缺点，那它还有存在的意义吗？别急，一个算法有缺陷，人们肯定会想办法去完善它，接下来的两个算法就是在标记/清除算法的基础上完善而来的。

##### 复制算法

为了解决效率问题，复制算法出现了。复制算法的原理是：将可用内存按容量划分为大小相等的两块，每次使用其中的一块。当这一块的内存用完了，就将还存活的对象复制到另一块内存上，然后把这一块内存所有的对象一次性清理掉。  
回收前示意图：



复制算法存在问题：

复制算法每次都是对整个半区进行内存回收，这样就减少了标记对象遍历的时间，在清除使用区域对象时，不用进行遍历，直接清空整个区域内存，而且在将存活对象复制到保留区域时也是按地址顺序存储的，这样就解决了内存碎片的问题，在分配对象内存时不用考虑内存碎片等复杂问题，只需要按顺序分配内存即可。

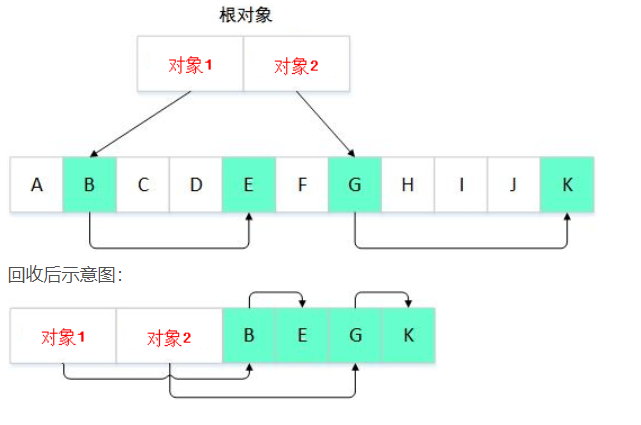
复制算法简单高效，优化了标记/清除算法的效率低、内存碎片多的问题。但是它的缺点也很明显：

1、将内存缩小为原来的一半，浪费了一半的内存空间，代价太高；

2、如果对象的存活率很高，极端一点的情况假设对象存活率为100%，那么我们需要将所有存活的对象复制一遍，耗费的时间代价也是不可忽视的。

##### 标记/整理算法

复制算法在对象存活率较高时要进行较多的复制操作，效率会变得很低，更关键的是，如果不想浪费50%的内存空间，就需要有额外的内存空间进行分配担保，以应对内存中对象100%存活的极端情况，因此，在老年代中由于对象的存活率非常高，复制算法就不合适了。根据老年代的特点，高人们提出了另一种算法：标记/整理算法。从名字上看，这种算法与标记/清除算法很像，事实上，标记/整理算法的标记过程任然与标记/清除算法一样，但后续步骤不是直接对可回收对象进行回收，而是让所有存活的对象都向一端移动，然后直接清理掉端边线以外的内存。  
回收前示意图：



--------------------- 本文来自 叩丁狼教育 的CSDN 博客 ，全文地址请点击：<https://blog.csdn.net/wolfcode_cn/article/details/81626128?utm_source=copy>

##### GC Roots 的对象

虚拟机栈(栈帧中的局部变量表)中引用的对象；

方法区中类静态属性引用的对象；

方法区中常量引用的对象；

本地方法栈中Native方法引用的对象；

##### **七种垃圾回收器**

1. **Serial收集器（复制算法):**新生代单线程收集器，标记和清理都是单线程，优点是简单高效；
2. **Serial Old收集器 (标记-整理算法):**老年代单线程收集器，Serial收集器的老年代版本；
3. **ParNew收集器 (复制算法):**新生代收并行集器，实际上是Serial收集器的多线程版本，在多核CPU环境下有着比Serial更好的表现；
4. **Parallel Scavenge收集器 (复制算法):**新生代并行收集器，追求高吞吐量，高效利用 CPU。吞吐量 = 用户线程时间/(用户线程时间+GC线程时间)，高吞吐量可以高效率的利用CPU时间，尽快完成程序的运算任务，适合后台应用等对交互相应要求不高的场景；
5. **Parallel Old收集器 (标记-整理算法)：** 老年代并行收集器，吞吐量优先，Parallel Scavenge收集器的老年代版本；
6. **CMS(Concurrent Mark Sweep)收集器（标记-清除算法）：** 老年代并行收集器，以获取最短回收停顿时间为目标的收集器，具有高并发、低停顿的特点，追求最短GC回收停顿时间。
7. **G1(Garbage First)收集器 (标记-整理算法)：** Java堆并行收集器，G1收集器是JDK1.7提供的一个新收集器，G1收集器基于“标记-整理”算法实现，也就是说不会产生内存碎片。此外，G1收集器不同于之前的收集器的一个重要特点是：G1回收的范围是整个Java堆(包括新生代，老年代)，而前六种收集器回收的范围仅限于新生代或老年代。

---------------------

作者：cyhaolihai

来源：CSDN

原文：https://blog.csdn.net/cyhaolihai/article/details/82902368?utm\_source=copy

版权声明：本文为博主原创文章，转载请附上博文链接！

##### 对象A和B循环引用，最后会不会不被GC回收？

①首先说一下，GC里边在JVM当中是使用的ROOT算法，ROOT算法，什么称作为ROOT呢，就是说类的静态成员，静态成员就是static修饰的那种，是“根”的一个，根还包括方法中的成员变量，只有成员或对象不挂在根上，GC的时候就可能把他们搞掉，这里提到的循环引用，就看这个循环引用是否挂在根上，如果挂在根上，如果这个根还被JVM的Java代码所执行的话，就不会GC掉，如果说这个根已经被释放掉了，这个对象不挂在跟上了，那个这个对象就会被GC掉。

②说一下根搜索算法，ROOTS，这个算法，哪些在Java里会被认为是根呢，在我印象里一般是static修饰的类成员，比如说静态字段，这种字段引用的对象被称为根，只要类在POOL区里不被卸载，一直在堆里，类对象只要没被回收掉，他引用的对象就不会被GC。

③再说另一种情况，方法中的栈，栈中有他的栈成员 Integer a = XXX，当方法没有被释放，没有出栈的时候，方法没有被弹出的时候，那Integer a 所引用的对象也是不会被回收的，在什么情况下回收呢，就是这个对象没有挂在根上，就会被回收。

④我们回到标题的问题，这个循环引用是否被回收，就看这个循环引用是否挂在根上，A引用B，B引用A，A和Ｂ并没有挂在某个内存元和根上，当他们的生命周期结束的时候，这两个对象都有可能被回收。

⑥具体回收的机制，就比较复杂了，每次GC的时候，对要被回收的对象标记一次，比如说会有个计数器每次+1，+1，+1，每次GC的时候就+1一次，当对象达到默认值了，比如说好像15次吧，在新生带创建的对象达到15次了就会被达到老年带里去，而老年代对象的回收的频率和新生带回收的频率是不一样的，可以仔细看下图中pool里的分区，了解他们的运行机制。

注：JVM heap分区块

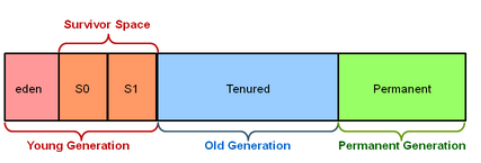
Generation代

* YongGeneration/NewGeneration：新生代，在Eden/S0/S1的存活的对象。
* OldGeneration：老年代，在Tenured区存活的对象。
* PermanentGeneration：永久代。

Space 区

* Eden：伊甸园区，是新生代的一个区。- Survivor：幸存区，属于新生代，为了复制算法的需要。一般分成大小相等的两个区（S0/S1或者From/To）。
* Tenured：存放老年代的区域。
* Permanent：终身区。

下图：Hotspot 的 Heap 分区



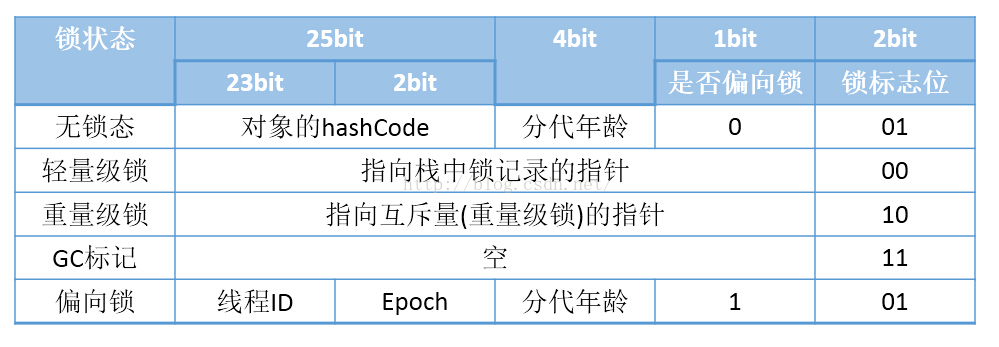
--------------------- 本文来自 徐刘根 的CSDN 博客 ，全文地址请点击：https://blog.csdn.net/xlgen157387/article/details/44832049?utm\_source=copy

### 6.对象头

HotSpot虚拟机中，对象在内存中存储的布局可以分为三块区域：对象头（Header）、实例数据（Instance Data）和对齐填充（Padding）。

HotSpot虚拟机的对象头(Object Header)包括两部分信息，第一部分用于存储对象自身的运行时数据， 如哈希码（HashCode）、GC分代年龄、锁状态标志、线程持有的锁、偏向线程ID、偏向时间戳等等，这部分数据的长度在32位和64位的虚拟机（暂 不考虑开启压缩指针的场景）中分别为32个和64个Bits，官方称它为“Mark Word”。

对象需要存储的运行时数据很多，其实已经超出了32、64位Bitmap结构所能记录的限度，但是对象头信息是与对象自身定义的数据无关的额 外存储成本，考虑到虚拟机的空间效率，Mark Word被设计成一个非固定的数据结构以便在极小的空间内存储尽量多的信息，它会根据对象的状态复用自己的存储空间。例如在32位的HotSpot虚拟机 中对象未被锁定的状态下，Mark Word的32个Bits空间中的25Bits用于存储对象哈希码（HashCode），4Bits用于存储对象分代年龄，2Bits用于存储锁标志 位，1Bit固定为0，在其他状态（轻量级锁定、重量级锁定、GC标记、可偏向）下对象的存储内容如下表所示。



对象头的另外一部分是类型指针，即是对象指向它的类的元数据的指针，虚拟机通过这个指针来确定这个对象是哪个类的实例。并不是所有的虚拟机实现都必须在对象数据上保留类型指针，换句话说查找对象的元数据信息并不一定要经过对象本身。另外，如果对象是一个Java数组，那在对象头中还必须有一块用于记录数组长度的数据，因为虚拟机可以通过普通Java对象的元数据信息确定Java对象的大小，但是从数组的元数据中无法确定数组的大小。

这里要特别关注的是锁标志位：锁标志位与是否偏向锁对应到唯一的锁状态。

所以锁的状态保存在对象头中，所以再理解Synchronized锁的到底是什么, 锁住的是代码还是对象)（答案锁的是对象）？

java中锁，锁的是对象，它是怎么实现的？

这两个问题，就好懂了！

### 7.锁的状态

锁的状态总共有四种：无锁状态、偏向锁、轻量级锁和重量级锁。随着锁的竞争，锁可以从偏向锁升级到轻量级锁，再升级的重量级锁（但是锁的升级是单向的，也就是说只能从低到高升级，不会出现锁的降级）。JDK 1.6中默认是开启偏向锁和轻量级锁的，我们也可以通过-XX:-UseBiasedLocking来禁用偏向锁。

1、轻量级锁的加锁过程

（1）在代码进入同步块的时候，如果同步对象锁状态为无锁状态（锁标志位为“01”状态，是否为偏向锁为“0”），虚拟机首先将在当前线程的栈帧中建立一个名为锁记录（Lock Record）的空间，用于存储锁对象目前的Mark Word的拷贝，官方称之为 Displaced Mark Word。这时候线程堆栈与对象头的状态如图2.1所示。

（2）拷贝对象头中的Mark Word复制到锁记录中。

（3）拷贝成功后，虚拟机将使用CAS操作尝试将对象的Mark Word更新为指向Lock Record的指针，并将Lock record里的owner指针指向object mark word。如果更新成功，则执行步骤（4），否则执行步骤（5）。

（4）如果这个更新动作成功了，那么这个线程就拥有了该对象的锁，并且对象Mark Word的锁标志位设置为“00”，即表示此对象处于轻量级锁定状态，这时候线程堆栈与对象头的状态如图2.2所示。

（5）如果这个更新操作失败了，虚拟机首先会检查对象的Mark Word是否指向当前线程的栈帧，如果是就说明当前线程已经拥有了这个对象的锁，那就可以直接进入同步块继续执行。否则说明多个线程竞争锁，轻量级锁就要膨胀为重量级锁，锁标志的状态值变为“10”，Mark Word中存储的就是指向重量级锁（互斥量）的指针，后面等待锁的线程也要进入阻塞状态。 而当前线程便尝试使用自旋来获取锁，自旋就是为了不让线程阻塞，而采用循环去获取锁的过程。

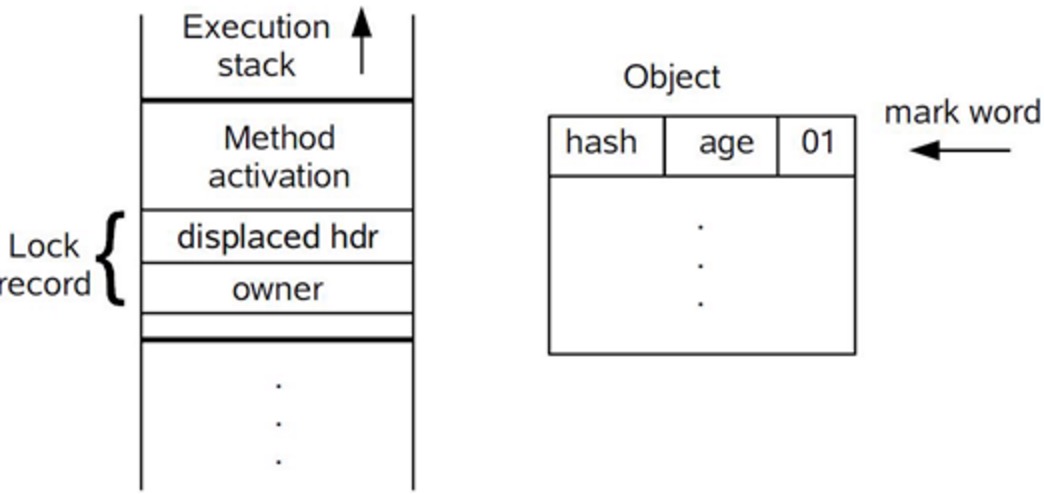


图2.1 轻量级锁CAS操作之前堆栈与对象的状态

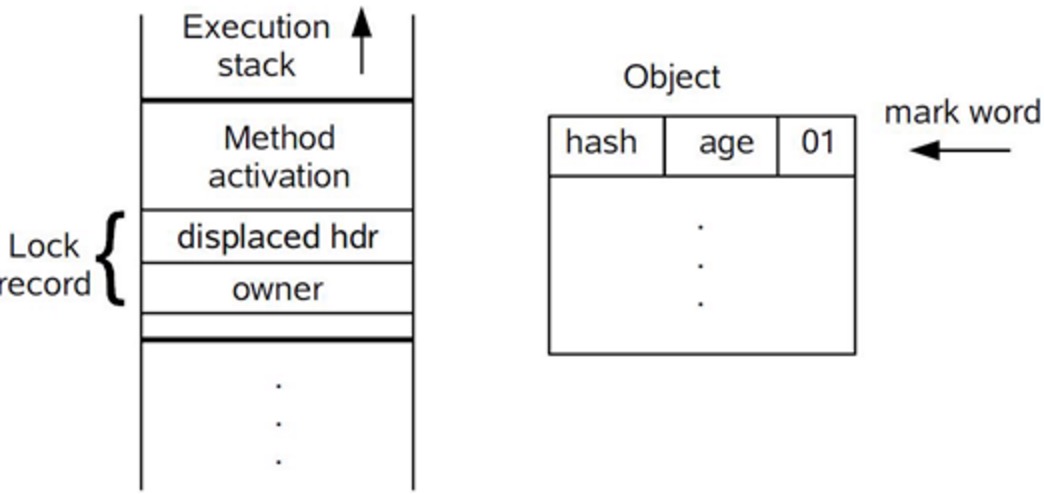


图2.2 轻量级锁CAS操作之后堆栈与对象的状态

**偏向锁**

引入偏向锁是为了在无多线程竞争的情况下尽量减少不必要的轻量级锁执行路径，因为轻量级锁的获取及释放依赖多次CAS原子指令，而偏向锁只需要在置换ThreadID的时候依赖一次CAS原子指令（由于一旦出现多线程竞争的情况就必须撤销偏向锁，所以偏向锁的撤销操作的性能损耗必须小于节省下来的CAS原子指令的性能消耗）。上面说过，轻量级锁是为了在线程交替执行同步块时提高性能，而偏向锁则是在只有一个线程执行同步块时进一步提高性能。

1、偏向锁获取过程：

（1）访问Mark Word中偏向锁的标识是否设置成1，锁标志位是否为01——确认为可偏向状态。

（2）如果为可偏向状态，则测试线程ID是否指向当前线程，如果是，进入步骤（5），否则进入步骤（3）。

（3）如果线程ID并未指向当前线程，则通过CAS操作竞争锁。如果竞争成功，则将Mark Word中线程ID设置为当前线程ID，然后执行（5）；如果竞争失败，执行（4）。

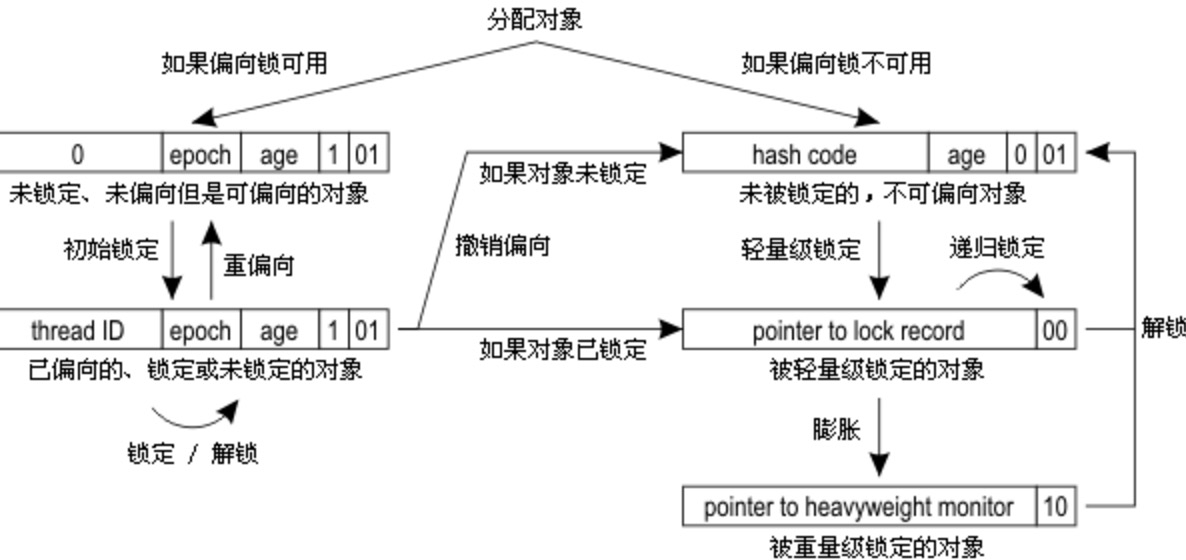
（4）如果CAS获取偏向锁失败，则表示有竞争。当到达全局安全点（safepoint）时获得偏向锁的线程被挂起，偏向锁升级为轻量级锁，然后被阻塞在安全点的线程继续往下执行同步代码。

（5）执行同步代码。

2、偏向锁的释放：

　　偏向锁的撤销在上述第四步骤中有提到。偏向锁只有遇到其他线程尝试竞争偏向锁时，持有偏向锁的线程才会释放锁，线程不会主动去释放偏向锁。偏向锁的撤销，需要等待全局安全点（在这个时间点上没有字节码正在执行），它会首先暂停拥有偏向锁的线程，判断锁对象是否处于被锁定状态，撤销偏向锁后恢复到未锁定（标志位为“01”）或轻量级锁（标志位为“00”）的状态。

3、重量级锁、轻量级锁和偏向锁之间转换



参考资料：

http://blog.csdn.net/u010723709/article/details/50341631

http://www.cnblogs.com/paddix/p/5405678.html

http://www.cnblogs.com/lingepeiyong/archive/2012/10/30/2745973.html

## JDBC

### JDBC是什么？

JDBC（JAVA DataBase Connection）即JAVA数据库连接技术，JDBC API是一个Java API，可以访问任何类型表列数据，特别是存储在关系数据库中的数据。   
JDBC库中所包含的API通常与数据库使用于：

* 连接到数据库
* 创建SQL语句
* 在数据库中执行SQL查询
* 查看和修改数据库中的数据记录

### 数据库驱动与常用接口

JDBC API使用数据库驱动连接操作数据库，其实也就是数据库厂商的JDBC接口实现，即对Connection等接口的实现类的jar文件。   
我使用的是MySql数据库，所以使用MySql提供的数据库驱动包mysql-connector-java-5.1.40-bin.jar。   
驱动包一般会提供很多操作数据库的接口，常用的有：

1. **Driver接口：**

在编程中要连接数据库，必须先装载特定厂商的数据库驱动程序，不同的数据库有不同的装载方法，MySql驱动装载方法为：Class.forName(“com.mysql.jdbc.Driver”);

1. **Connection接口：**

Connection与特定数据库的连接（会话），在连接上下文中执行sql语句并返回结果。DriverManager.getConnection(url, user, password)方法可以获取相应的数据库连接。MySql数据库连接获取方法为：Connection conn = DriverManager.getConnection(“jdbc:mysql://host:port/database”, “user”, “password”)。该接口包含了几个常用的方法：  
createStatement():创建向数据库发送sql语句的statement对象；   
prepareStatement(sql) ：创建向数据库发送预编译sql的PrepareSatement对象；   
setAutoCommit(boolean)：设置数据库事务是否自动提交；   
commit()：提交事务；   
rollback()：事务回滚。

1. **Statement接口：**

用于执行静态SQL语句并返回它所生成结果的对象。一般有三种Statement对象：   
1、Statement：

由createStatement创建，用于发送简单的SQL语句（不带参数）；   
2、PreparedStatement ：

继承自Statement接口，由preparedStatement创建，用于发送含有一个或多个参数的SQL语句。PreparedStatement对象比Statement对象的效率更高，并且可以防止SQL注入，所以我们一般都使用PreparedStatement；   
3、CallableStatement：

继承自PreparedStatement接口，由方法prepareCall创建，用于调用存储过程。   
以下是几个常用的Statement方法：   
1、execute(String sql):运行语句，返回是否有结果集；   
2、executeQuery(String sql)：运行select语句，返回ResultSet结果集；   
3、executeUpdate(String sql)：运行insert/update/delete操作，返回更新的行数；   
4、addBatch(String sql) ：把多条sql语句放到一个批处理中；   
5、executeBatch()：向数据库发送一批sql语句执行。

1. **ResultSet接口：ResultSet提供检索不同类型字段的方法，常用的有：**   
   1、getString(int index)、getString(String columnName)：获得在数据库里是varchar、char等类型的数据对象；   
   2、getFloat(int index)、getFloat(String columnName)：获得在数据库里是Float类型的数据对象；   
   3、getDate(int index)、getDate(String columnName)：获得在数据库里是Date类型的数据；   
   4、getBoolean(int index)、getBoolean(String columnName)：获得在数据库里是Boolean类型的数据；   
   5、getObject(int index)、getObject(String columnName)：获取在数据库里任意类型的数据。
2. **相比于以上方法，ResultSet接口提供了更常用的对于结果集的处理方法：**   
   1、next()：移动到下一行；   
   2、Previous()：移动到前一行；   
   3、absolute(int row)：移动到指定行；   
   4、beforeFirst()：移动resultSet的最前面；   
   5、afterLast() ：移动到resultSet的最后面。

### JDBC使用步骤

* JDBC标准流程为：加载JDBC驱动程序→建立数据库连接Connection→创建执行SQL的语句Statement→处理执行结果ResultSet→释放资源，其中释放资源顺序为：ResultSet→Statement→Connection。   
  以下是一个基本例子：   
  首先创建要是用的数据库表：
* create database db\_test;
* use db\_test;
* create table person(
* id int primary key,
* name varchar(20),
* job varchar(20)
* );
* insert into person values(1001,'zhangsan','teacher');
* */\*\*  
   \* 获取数据库连接方法  
   \** ***@return*** *\*/* private static Connection getConn(){  
   String driver = "com.mysql.jdbc.Driver";  
   //后面的一串表示关闭ssl协议  
   String url = "jdbc:mysql://192.168.1.110:3306/db\_test?useUnicode=true&characterEncoding=utf-8&useSSL=false";  
   String username = "root";  
   String password = "123456";  
   Connection connection = null;  
    
   try {  
   //加载驱动  
   Class.*forName*(driver);  
   //获取数据库连接  
   connection = (Connection) DriverManager.*getConnection*(url, username, password);  
   }catch (SQLException e){  
   e.printStackTrace();  
   }catch (ClassNotFoundException e){  
   e.printStackTrace();  
   }  
   return connection;  
   }  
   */\*\*  
   \* 以下基于Statement  
   \*/* //执行查询类的SQL语句，有返回集  
   public ResultSet executeQuery(String sql)  
   {  
   try  
   {  
   stmt=conn.createStatement(ResultSet.*TYPE\_SCROLL\_SENSITIVE*,ResultSet.*CONCUR\_UPDATABLE*);  
   rs=stmt.executeQuery(sql);  
   }catch(SQLException e){  
   System.*err*.println("Data.executeQuery: " + e.getMessage());  
   }  
   return rs;  
   }  
    
   */\*\*  
   \* insert  
   \*/* public int executeUpdate(String sql) {  
   int rows = 0;  
   try  
   {  
   stmt=conn.createStatement();  
   rows = stmt.executeUpdate(sql);  
    
   }catch(SQLException e){  
   System.*err*.println("Data.executeUpdate: " + e.getMessage());  
   }  
   return rows;  
   }  
     
   */\*\*  
   \* 以下基于PreparedStatement  
   \*/  
   /\*\*  
   \* 插入方法  
   \** ***@param*** *person  
   \** ***@return*** *\*/* private static int insert(Person person){  
   Connection connection = getConn();  
   int i = 0;  
   String sql = "insert into person (id,name,job) values(?,?,?);";  
   PreparedStatement pstmt;  
   try {  
   //从连接获取PreparedStatement对象  
   pstmt = (PreparedStatement) connection.prepareStatement(sql);  
   //组装sql  
   pstmt.setInt(1, person.getId());  
   pstmt.setString(2, person.getName());  
   pstmt.setString(3, person.getJob());  
   //执行sql  
   i = pstmt.executeUpdate();  
   //关闭连接  
   pstmt.close();  
   connection.close();  
   }catch (SQLException e){  
   e.printStackTrace();  
   }  
   return i;  
   }  
    
   */\*\*  
   \* 更新方法  
   \** ***@param*** *person  
   \** ***@return*** *\*/* private static int update(Person person) {  
   Connection conn = getConn();  
   int i = 0;  
   String sql = "update person set name='" + person.getName() + "' where id='" + person.getId() + "'";  
   PreparedStatement pstmt;  
   try {  
   pstmt = (PreparedStatement) conn.prepareStatement(sql);  
   i = pstmt.executeUpdate();  
   System.*out*.println("resutl: " + i);  
   pstmt.close();  
   conn.close();  
   } catch (SQLException e) {  
   e.printStackTrace();  
   }  
   return i;  
   }  
    
   private static Integer getAll() {  
   Connection conn = getConn();  
   String sql = "select \* from person";  
   PreparedStatement pstmt;  
   try {  
   pstmt = (PreparedStatement)conn.prepareStatement(sql);  
   ResultSet rs = pstmt.executeQuery();  
   int col = rs.getMetaData().getColumnCount();  
   System.*out*.println("============================");  
   while (rs.next()) {  
   for (int i = 1; i <= col; i++) {  
   System.*out*.print(rs.getString(i) + "\t");  
   if ((i == 2) && (rs.getString(i).length() < 8)) {  
   System.*out*.print("\t");  
   }  
   }  
   System.*out*.println("");  
   }  
   System.*out*.println("============================");  
   } catch (SQLException e) {  
   e.printStackTrace();  
   }  
   return null;  
   }  
    
   private static int delete(Integer id) {  
   Connection conn = getConn();  
   int i = 0;  
   String sql = "delete from person where id='" + id + "'";  
   PreparedStatement pstmt;  
   try {  
   pstmt = (PreparedStatement) conn.prepareStatement(sql);  
   i = pstmt.executeUpdate();  
   System.*out*.println("resutl: " + i);  
   pstmt.close();  
   conn.close();  
   } catch (SQLException e) {  
   e.printStackTrace();  
   }  
   return i;  
   }  
    
   }  
    
  ---------------------  
    
   本文来自 smilematch 的CSDN 博客 ，全文地址请点击：https://blog.csdn.net/smilematch/article/details/78473835?utm\_source=copy

### SQL语法

1. Group by

“Group By”从字面意义上理解就是根据“By”指定的规则对数据进行分组，所谓的分组就是将一个“数据集”划分成若干个“小区域”，然后针对若干个“小区域”进行数据处理。



select 类别, sum(数量) as 数量之和

from A

group by 类别



1. **WHERE和HAVING的区别在于：**

where 子句的作用是对查询结果进行分组前，将不符合where条件的行去掉，即在分组之前过滤数据，where条件中不能包含聚组函数，使用where条件过滤出特定的行。

having 子句的作用是筛选满足条件的组，即在分组之后过滤数据，条件中经常包含聚组函数，使用having 条件过滤出特定的组，也可以使用多个分组标准进行分组。

### MySQL 事务

MySQL 事务主要用于处理操作量大，复杂度高的数据。比如说，在人员管理系统中，你删除一个人员，你即需要删除人员的基本资料，也要删除和该人员相关的信息，如信箱，文章等等，这样，这些数据库操作语句就构成一个事务！可**以保证数据的一致性和完整性(避免异常和错误等导致的数据信息异常)**

* 在 MySQL 中只有使用了 Innodb 数据库引擎的数据库或表才支持事务。
* 事务处理可以用来维护数据库的完整性，保证成批的 SQL 语句要么全部执行，要么全部不执行。
* 事务用来管理 insert,update,delete 语句

一般来说，事务是必须满足4个条件（ACID）：：原子性（**A**tomicity，或称不可分割性）、一致性（**C**onsistency）、隔离性（**I**solation，又称独立性）、持久性（**D**urability）。

* **原子性：**一个事务（transaction）中的所有操作，要么全部完成，要么全部不完成，不会结束在中间某个环节。事务在执行过程中发生错误，会被回滚（Rollback）到事务开始前的状态，就像这个事务从来没有执行过一样。
* **一致性：**在事务开始之前和事务结束以后，数据库的完整性没有被破坏。这表示写入的资料必须完全符合所有的预设规则，这包含资料的精确度、串联性以及后续数据库可以自发性地完成预定的工作。
* **隔离性：**数据库允许多个并发事务同时对其数据进行读写和修改的能力，隔离性可以防止多个事务并发执行时由于交叉执行而导致数据的不一致。事务隔离分为不同级别，包括读未提交（Read uncommitted）、读提交（read committed）、可重复读（repeatable read）和串行化（Serializable）。
* **持久性：**事务处理结束后，对数据的修改就是永久的，即便系统故障也不会丢失。

#### 事务控制语句：

* BEGIN或START TRANSACTION；显式地开启一个事务；
* COMMIT；也可以使用COMMIT WORK，不过二者是等价的。COMMIT会提交事务，并使已对数据库进行的所有修改成为永久性的；
* ROLLBACK；有可以使用ROLLBACK WORK，不过二者是等价的。回滚会结束用户的事务，并撤销正在进行的所有未提交的修改；
* SAVEPOINT identifier；SAVEPOINT允许在事务中创建一个保存点，一个事务中可以有多个SAVEPOINT；
* RELEASE SAVEPOINT identifier；删除一个事务的保存点，当没有指定的保存点时，执行该语句会抛出一个异常；
* ROLLBACK TO identifier；把事务回滚到标记点；
* SET TRANSACTION；用来设置事务的隔离级别。InnoDB存储引擎提供事务的隔离级别有READ UNCOMMITTED、READ COMMITTED、REPEATABLE READ和SERIALIZABLE。

#### MYSQL 事务处理主要有两种方法：

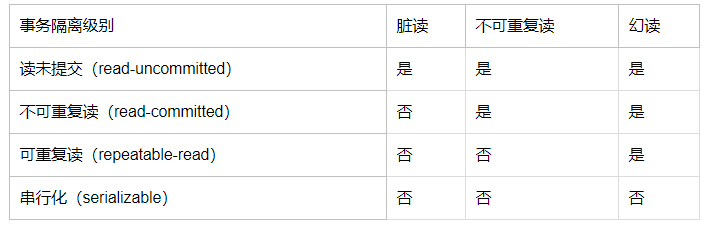
1、用 BEGIN, ROLLBACK, COMMIT来实现

* **BEGIN** 开始一个事务
* **ROLLBACK** 事务回滚
* **COMMIT** 事务确认

2、直接用 SET 来改变 MySQL 的自动提交模式:

* **SET AUTOCOMMIT=0** 禁止自动提交
* **SET AUTOCOMMIT=1** 开启自动提交

#### 四种事务隔离级别



**1.脏读**

A事务读取B事务尚未提交的更改数据，并在这个数据的基础上进行操作，这时候如果事务B回滚，那么A事务读到的数据是不被承认的。例如常见的取款事务和转账事务：



**2.不可重复读(update/delete)**

不可重复读是指A事务读取了B事务已经提交的更改数据。假如A在取款事务的过程中，B往该账户转账100，A两次读取的余额发生不一致。

**3.幻读（insert）**

A事务读取B事务提交的新增数据,会引发幻读问题。幻读一般发生在计算统计数据的事务中，例如银行系统在同一个事务中两次统计存款账户的总金额，在两次统计中，刚好新增了一个存款账户，存入了100，这时候两次统计的总金额不一致。



注意：不可重复读和幻读的区别是：前者是指读到了已经提交的事务的更改数据（修改或删除），后者是指读到了其他已经提交事务的新增数据。对于这两种问题解决采用不同的办法，防止读到更改数据，只需对操作的数据添加行级锁，防止操作中的数据发生变化；二防止读到新增数据，往往需要添加表级锁，将整张表锁定，防止新增数据（oracle采用多版本数据的方式实现）。

**4.第一类丢失更新**

A事务撤销时，把已经提交的B事务的更新数据覆盖了。例如：



这时候取款事务A撤销事务，余额恢复为1000，这就丢失了更新。

**5.第二类丢失更新**

A事务覆盖B事务已经提交的数据，造成B事务所做的操作丢失



---------------------

作者：小小程序汪

来源：CSDN

原文：https://blog.csdn.net/starlh35/article/details/76445267?utm\_source=copy

版权声明：本文为博主原创文章，转载请附上博文链接！

* mysql中默认事务隔离级别是可重复读，不会锁住读取到的行。
* 读取已提交内容  和  可重复读 两种隔离级别通过MCVV(多版本并发控制)实现.
* 读取已提交内容:MVCC  读不加锁 写加锁
* (读取历史版本)相对于可重复读，是最近的历史版本。
* 可重复读:MVCC+Read View +Gap锁 读不加锁写加锁(读取历史版本)
* Read View保证事务过程中读取的一致性，即可重复读
* Gap锁(解决当前读)+Read View(解决快照读) 解决幻读问题
* MySQL的可重复读模式下，不会出现幻读。
* 在读取提交内容 和 可重复读 模式下，读操作都不加锁， 在串行化模式下读加锁
* ---------------------
* 作者：cyhaolihai
* 来源：CSDN
* 原文：https://blog.csdn.net/cyhaolihai/article/details/82902368?utm\_source=copy
* 版权声明：本文为博主原创文章，转载请附上博文链接！

### MySql索引

#### 索引介绍

对于任何DBMS，索引都是进行优化的最主要的因素。对于少量的数据，没有合适的索引影响不是很大，但是，当随着数据量的增加，性能会急剧下降。

2.5万数据单表中，无索引：200ms-700ms，添加索引后10ms-15ms，使用redis缓存1ms-7ms，如果数据量更大的时候，索引效果将会更加明显。更甚者，多表查询。

##### 索引原理

1、除了词典，生活中随处可见索引的例子，如火车站的车次表、图书的目录等。它们的原理都是一样的，通过不断的缩小想要获得数据的范围来筛选出最终想要的结果，同时把随机的事件变成顺序的事件，也就是我们总是通过同一种查找方式来锁定数据。

数据库也是一样，但显然要复杂许多，因为不仅面临着等值查询，还有范围查询(>、<、between、in)、模糊查询(like)、并集查询(or)等等。数据库应该选择怎么样的方式来应对所有的问题呢？我们回想字典的例子，能不能把数据分成段，然后分段查询呢？最简单的如果1000条数据，1到100分成第一段，101到200分成第二段，201到300分成第三段……这样查第250条数据，只要找第三段就可以了，一下子去除了90%的无效数据。但如果是1千万的记录呢，分成几段比较好？稍有算法基础的同学会想到搜索树，其平均复杂度是lgN，具有不错的查询性能。但这里我们忽略了一个关键的问题，复杂度模型是基于每次相同的操作成本来考虑的，数据库实现比较复杂，数据保存在磁盘上，而为了提高性能，每次又可以把部分数据读入内存来计算，因为我们知道访问磁盘的成本大概是访问内存的十万倍左右，所以简单的搜索树难以满足复杂的应用场景。

2、另外，比如学生信息表，添加学生姓名索引，索引是在name上排序的。现在，当查找某个学生信息时，就不需要逐行搜索全表，可以利用索引进行有序查找（如二分查找法），并快速定位到匹配的值，以节省大量搜索时间。

3、是当数据量非常大，查询涉及多个表时，使用索引往往能使查询速度加快成千上万倍。

例如，有3个未索引的表t1、t2、t3，分别只包含列c1、c2、c3，每个表分别含有1000行数据组成，指为1～1000的数值，查找对应值相等行的查询如下所示。

SELECT c1,c2,c3 FROM t1,t2,t3 WHERE c1=c2 AND c1=c3

1

此查询结果应该为1000行，每行包含3个相等的值。在无索引的情况下处理此查询，必须寻找3个表所有的组合，以便得出与WHERE子句相配的那些行。而可能的组合数目为1000×1000×1000（十亿），显然查询将会非常慢。

如果对每个表进行索引，就能极大地加速查询进程。利用索引的查询处理如下。

（1）从表t1中选择第一行，查看此行所包含的数据。

（2）使用表t2上的索引，直接定位t2中与t1的值匹配的行。类似，利用表t3上的索引，直接定位t3中与来自t1的值匹配的行。

（3）扫描表t1的下一行并重复前面的过程，直到遍历t1中所有的行。

在此情形下，仍然对表t1执行了一个完全扫描，但能够在表t2和t3上进行索引查找直接取出这些表中的行，比未用索引时要快一百万倍。

利用索引，MySQL加速了WHERE子句满足条件行的搜索，而在多表连接查询时，在执行连接时加快了与其他表中的行匹配的速度。

##### 索引的类型

MySQL的索引包括普通索引、唯一性索引、全文索引、单列索引、多列索引和空间索引等。

1．普通索引

在创建普通索引时，不附加任何限制条件。这类索引可以创建在任何数据类型中，其值是否唯一和非空由字段本身的完整性约束条件决定。建立索引以后，查询时可以通过索引进行查询。例如，在student表的stu\_id字段上建立一个普通索引。查询记录时，就可以根据该索引进行查询。

2．唯一性索引

使用UNIQUE参数可以设置索引为唯一性索引。在创建唯一性索引时，限制该索引的值必须是唯一的。例如，在student表的stu\_name字段中创建唯一性索引，那么stu\_name字段的值就必需是唯一的。通过唯一性索引，可以更快速地确定某条记录。主键就是一种特殊唯一性索引。

3．全文索引

使用FULLTEXT参数可以设置索引为全文索引。全文索引只能创建在CHAR、VARCHAR或TEXT类型的字段上。查询数据量较大的字符串类型的字段时，使用全文索引可以提高查询速度。例如，student表的information字段是TEXT类型，该字段包含了很多的文字信息。在information字段上建立全文索引后，可以提高查询information字段的速度。MySQL数据库从3.23.23版开始支持全文索引，但只有MyISAM存储引擎支持全文检索。在默认情况下，全文索引的搜索执行方式不区分大小写。但索引的列使用二进制排序后，可以执行区分大小写的全文索引。

4．单列索引

在表中的单个字段上创建索引。单列索引只根据该字段进行索引。单列索引可以是普通索引，也可以是唯一性索引，还可以是全文索引。只要保证该索引只对应一个字段 即可。

5．多列索引

多列索引是在表的多个字段上创建一个索引。该索引指向创建时对应的多个字段，可以通过这几个字段进行查询。但是，只有查询条件中使用了这些字段中第一个字段时，索引才会被使用。例如，在表中的id、name和sex字段上建立一个多列索引，那么，只有查询条件使用了id字段时该索引才会被使用。

6．空间索引

使用SPATIAL参数可以设置索引为空间索引。空间索引只能建立在空间数据类型上，这样可以提高系统获取空间数据的效率。MySQL中的空间数据类型包括GEOMETRY和POINT、LINESTRING和POLYGON等。目前只有MyISAM存储引擎支持空间检索，而且索引的字段不能为空值。对于初学者来说，这类索引很少会用到。

##### 索引的操作

1.添加PRIMARY KEY（主键索引）

mysql>ALTER TABLE `table\_name` ADD PRIMARY KEY ( `column` )

1

2.添加UNIQUE(唯一索引)

mysql>ALTER TABLE `table\_name` ADD UNIQUE ( `column` )

1

3.添加INDEX(普通索引)

mysql>ALTER TABLE `table\_name` ADD INDEX index\_name ( `column` )

1

4.添加FULLTEXT(全文索引)

mysql>ALTER TABLE `table\_name` ADD FULLTEXT ( `column`)

1

5.添加多列索引

mysql>ALTER TABLE `table\_name` ADD INDEX index\_name ( `column1`, `column2`, `column3` )

1

创建索引

在执行CREATE TABLE语句时可以创建索引，也可以单独用CREATE INDEX或ALTER TABLE来为表增加索引。

1．ALTER TABLE

ALTER TABLE用来创建普通索引、UNIQUE索引或PRIMARY KEY索引。

ALTER TABLE table\_name ADD INDEX index\_name (column\_list)

ALTER TABLE table\_name ADD UNIQUE (column\_list)

ALTER TABLE table\_name ADD PRIMARY KEY (column\_list)

1

2

3

其中table\_name是要增加索引的表名，column\_list指出对哪些列进行索引，多列时各列之间用逗号分隔。索引名index\_name可选，缺省时，MySQL将根据第一个索引列赋一个名称。另外，ALTER TABLE允许在单个语句中更改多个表，因此可以在同时创建多个索引。

2．CREATE INDEX

CREATE INDEX可对表增加普通索引或UNIQUE索引。

CREATE INDEX index\_name ON table\_name (column\_list)

CREATE UNIQUE INDEX index\_name ON table\_name (column\_list)

1

2

table\_name、index\_name和column\_list具有与ALTER TABLE语句中相同的含义，索引名不可选。另外，不能用CREATE INDEX语句创建PRIMARY KEY索引。

3．索引类型

在创建索引时，可以规定索引能否包含重复值。如果不包含，则索引应该创建为PRIMARY KEY或UNIQUE索引。对于单列惟一性索引，这保证单列不包含重复的值。对于多列惟一性索引，保证多个值的组合不重复。

PRIMARY KEY索引和UNIQUE索引非常类似。事实上，PRIMARY KEY索引仅是一个具有名称PRIMARY的UNIQUE索引。这表示一个表只能包含一个PRIMARY KEY，因为一个表中不可能具有两个同名的索引。

下面的SQL语句对students表在sid上添加PRIMARY KEY索引。

ALTER TABLE students ADD PRIMARY KEY (sid)

1

删除索引

可利用ALTER TABLE或DROP INDEX语句来删除索引。类似于CREATE INDEX语句，DROP INDEX可以在ALTER TABLE内部作为一条语句处理，语法如下。

DROP INDEX index\_name ON talbe\_name

ALTER TABLE table\_name DROP INDEX index\_name

ALTER TABLE table\_name DROP PRIMARY KEY

1

2

3

其中，前两条语句是等价的，删除掉table\_name中的索引index\_name。

第3条语句只在删除PRIMARY KEY索引时使用，因为一个表只可能有一个PRIMARY KEY索引，因此不需要指定索引名。如果没有创建PRIMARY KEY索引，但表具有一个或多个UNIQUE索引，则MySQL将删除第一个UNIQUE索引。

如果从表中删除了某列，则索引会受到影响。对于多列组合的索引，如果删除其中的某列，则该列也会从索引中删除。如果删除组成索引的所有列，则整个索引将被删除。

注：一般数据库默认都会为主键生成索引

---------------------

作者：小宝鸽

来源：CSDN

原文：https://blog.csdn.net/u013142781/article/details/51424174?utm\_source=copy

版权声明：本文为博主原创文章，转载请附上博文链接！

#### B+树索引原理

**B-树**

是一种多路搜索树（并不是二叉的）：

1.定义任意非叶子结点最多只有M个儿子；且M>2；

2.根结点的儿子数为[2, M]；

3.除根结点以外的非叶子结点的儿子数为[M/2, M]；

4.每个结点存放至少M/2-1（取上整）和至多M-1个关键字；（至少2个关键字）

5.非叶子结点的关键字个数=指向儿子的指针个数-1；

6.非叶子结点的关键字：K[1], K[2], …, K[M-1]；且K[i] < K[i+1]；

7.非叶子结点的指针：P[1], P[2], …, P[M]；其中P[1]指向关键字小于K[1]的子树，P[M]指向关键字大于K[M-1]的子树，其它P[i]指向关键字属于(K[i-1], K[i])的子树；

8.所有叶子结点位于同一层；

**B+树**

B+树是B-树的变体，也是一种多路搜索树：5

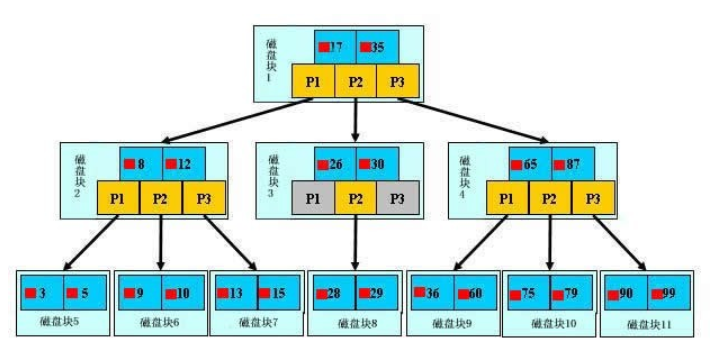
1.其定义基本与B-树同，除了：

2.非叶子结点的子树指针与关键字个数相同；

3.非叶子结点的子树指针P[i]，指向关键字值属于[K[i], K[i+1])的子树（B-树是开区间）；

5.为所有叶子结点增加一个链指针；

6.所有关键字都在叶子结点出现；



如上图，是一颗b+树，关于b+树的定义可以参见B+树，这里只说一些重点，浅蓝色的块我们称之为一个磁盘块，可以看到每个磁盘块包含几个数据项（深蓝色所示）和指针（黄色所示），如磁盘块1包含数据项17和35，包含指针P1、P2、P3，P1表示小于17的磁盘块，P2表示在17和35之间的磁盘块，P3表示大于35的磁盘块。真实的数据存在于叶子节点即3、5、9、10、13、15、28、29、36、60、75、79、90、99。非叶子节点只不存储真实的数据，只存储指引搜索方向的数据项，如17、35并不真实存在于数据表中。

##### b+树的查找过程

如图所示，如果要查找数据项29，那么首先会把磁盘块1由磁盘加载到内存，此时发生一次IO，在内存中用二分查找确定29在17和35之间，锁定磁盘块1的P2指针，内存时间因为非常短（相比磁盘的IO）可以忽略不计，通过磁盘块1的P2指针的磁盘地址把磁盘块3由磁盘加载到内存，发生第二次IO，29在26和30之间，锁定磁盘块3的P2指针，通过指针加载磁盘块8到内存，发生第三次IO，同时内存中做二分查找找到29，结束查询，总计三次IO。真实的情况是，3层的b+树可以表示上百万的数据，如果上百万的数据查找只需要三次IO，性能提高将是巨大的，如果没有索引，每个数据项都要发生一次IO，那么总共需要百万次的IO，显然成本非常非常高。

##### b+树性质

1.通过上面的分析，我们知道IO次数取决于b+数的高度h，假设当前数据表的数据为N，每个磁盘块的数据项的数量是m，则有h=㏒(m+1)N，当数据量N一定的情况下，m越大，h越小；而m = 磁盘块的大小 / 数据项的大小，磁盘块的大小也就是一个数据页的大小，是固定的，如果数据项占的空间越小，数据项的数量越多，树的高度越低。这就是为什么每个数据项，即索引字段要尽量的小，比如int占4字节，要比bigint8字节少一半。这也是为什么b+树要求把真实的数据放到叶子节点而不是内层节点，一旦放到内层节点，磁盘块的数据项会大幅度下降，导致树增高。当数据项等于1时将会退化成线性表。

2.当b+树的数据项是复合的数据结构，比如(name,age,sex)的时候，b+数是按照从左到右的顺序来建立搜索树的，比如当(张三,20,F)这样的数据来检索的时候，b+树会优先比较name来确定下一步的所搜方向，如果name相同再依次比较age和sex，最后得到检索的数据；但当(20,F)这样的没有name的数据来的时候，b+树就不知道下一步该查哪个节点，因为建立搜索树的时候name就是第一个比较因子，必须要先根据name来搜索才能知道下一步去哪里查询。比如当(张三,F)这样的数据来检索时，b+树可以用name来指定搜索方向，但下一个字段age的缺失，所以只能把名字等于张三的数据都找到，然后再匹配性别是F的数据了， 这个是非常重要的性质，即索引的最左匹配特性。

##### 慢查询优化

关于MySQL索引原理是比较枯燥的东西，大家只需要有一个感性的认识，并不需要理解得非常透彻和深入。我们回头来看看一开始我们说的慢查询，了解完索引原理之后，大家是不是有什么想法呢？先总结一下索引的几大基本原则

**MySQL会记录下查询超过指定时间的语句，我们将超过指定时间的SQL语句查询称为慢查询，都记在慢查询日志里.**

#### 建索引的几大原则

1.最左前缀匹配原则，非常重要的原则，mysql会一直向右匹配直到遇到范围查询(>、<、between、like)就停止匹配，比如a = 1 and b = 2 and c > 3 and d = 4 如果建立(a,b,c,d)顺序的索引，d是用不到索引的，如果建立(a,b,d,c)的索引则都可以用到，a,b,d的顺序可以任意调整。

2.=和in可以乱序，比如a = 1 and b = 2 and c = 3 建立(a,b,c)索引可以任意顺序，mysql的查询优化器会帮你优化成索引可以识别的形式

3.尽量选择区分度高的列作为索引,区分度的公式是count(distinct col)/count(\*)，表示字段不重复的比例，比例越大我们扫描的记录数越少，唯一键的区分度是1，而一些状态、性别字段可能在大数据面前区分度就是0，那可能有人会问，这个比例有什么经验值吗？使用场景不同，这个值也很难确定，一般需要join的字段我们都要求是0.1以上，即平均1条扫描10条记录

4.索引列不能参与计算，保持列“干净”，比如from\_unixtime(create\_time) = ’2014-05-29’就不能使用到索引，原因很简单，b+树中存的都是数据表中的字段值，但进行检索时，需要把所有元素都应用函数才能比较，显然成本太大。所以语句应该写成create\_time = unix\_timestamp(’2014-05-29’);

5.尽量的扩展索引，不要新建索引。比如表中已经有a的索引，现在要加(a,b)的索引，那么只需要修改原来的索引即可

#### 数据库索引和选择性的关系

在讨论数据库索引的时候，经常会提到“选择性”(selectivity)这个概念。“选择性”是描述列值数据分布情况的一个重要属性。“选择性”和“基数”(cardinality)是两个密不可分的概念。“基数”就是一列中唯一值的数量，对于有唯一约束的列，“基数”等于表的总行数。

##### ****怎么计算索引的选择性？****

选择性  =  基数/总行数 \* 100%

**选择性是什么意思？**

上面的公式应该怎么理解？假设表中有Sex(性别)这一列，列值只有两种可能—`Male`(男性)和`Female`(女性)，那么，Sex列的基数(cardinality)就是2。如果这张表中有10000条行记录，那么Sex列的索引的选择性就是2/10000 \* 100% = 0.02%。

为什么“选择性”对索引很重要，数据库怎么利用“选择性”的？

“选择性”的高低可以衡量列值的可能性，换句话说，在给定的样本集里有多少个不同的值。

我们思考一下，低选择性意味着什么？低选择性意味着列值没有太多变化（或没有太多的可能性）。例如，Sex列的选择性，只有非常低的0.02%，这就意味着，Sex列的列值有很少的不同值。

##### ****选择性对数据库索引有什么影响？****

数据库的查询优化器会根据索引的“选择性”来判断是否使用索引执行查询。也就是说，你在某列上创建了索引，这不意味着数据库就一定会使用这个索引，因为有时全表扫描是更高效的选择。

##### ****什么时候不应该使用数据库索引？****

什么时候不应该使用数据库索引？当“选择性”比较低的时候！为什么低选择性的时候不适合使用索引呢？设想一下，现在我们要查询所有女性的名字，由于性别只有男性和女性两种情况，所以女性占比是50%的可能性很大。那我们就假设确实有50%(5000)是女性。如果查找索引的话，数据库为了查找出所有女性就需要访问索引5000次。要知道访问索引也是需要消耗时间和资源的。所以这种情况，直接去做全表扫描可能会更快一些。可以看到，数据库的查询优化器会根据“选择性”的高低来决定使用索引还是直接全表扫描。

##### ****“选择性”等于多少才会使用索引？****

这个问题很难回答，它因数据库而异。

当然，“选择性”很高时，应该使用索引。例如，我们要查询的某列，其“选择性”是100%，说明该列的列值都是唯一的。这时，如果只查询其中的一行，使用索引是最高效的。同时，这也是执行全表扫描最坏的情形。

## 反射

### 反射的概述

JAVA反射机制是在运行状态中，对于任意一个类，都能够知道这个类的所有属性和方法；对于任意一个对象，都能够调用它的任意一个方法和属性；这种动态获取的信息以及动态调用对象的方法的功能称为java语言的反射机制。

要想解剖一个类,必须先要获取到该类的字节码文件对象。而解剖使用的就是Class类中的方法.所以先要获取到每一个字节码文件对应的Class类型的对象.

**以上的总结就是什么是反射**

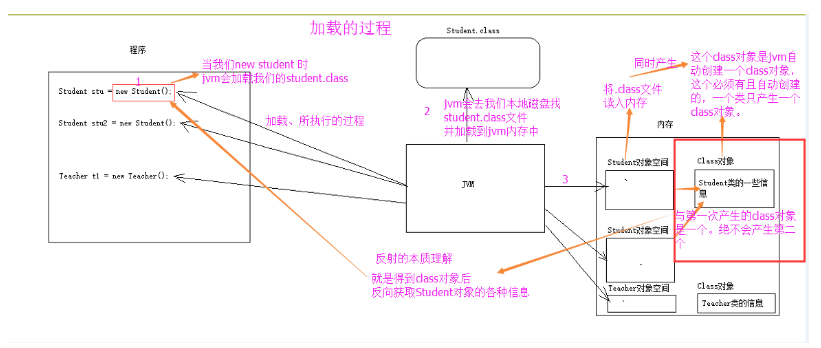
反射就是把java类中的各种成分映射成一个个的Java对象

例如：一个类有：成员变量、方法、构造方法、包等等信息，利用反射技术可以对一个类进行解剖，把个个组成部分映射成一个个对象。

     （其实：一个类中这些成员方法、构造方法、在加入类中都有一个类来描述）

如图是类的正常加载过程：反射的原理在与class对象。

熟悉一下加载的时候：Class对象的由来是将class文件读入内存，并为之创建一个Class对象。



其中这个Class对象很特殊。我们先了解一下这个Class类

### 查看Class类在java中的api详解（1.7的API）

如何阅读java中的api详见java基础之——String字符串处理



Class 类的实例表示正在运行的 Java 应用程序中的类和接口。也就是jvm中有N多的实例每个类都有该Class对象。（包括基本数据类型）

Class 没有公共构造方法。Class 对象是在加载类时由 Java 虚拟机以及通过调用类加载器中的defineClass 方法自动构造的。也就是这不需要我们自己去处理创建，JVM已经帮我们创建好了。

没有公共的构造方法，方法共有64个太多了。下面用到哪个就详解哪个吧



### 反射的使用（这里使用Student类做演示）

#### 获取Class对象的三种方式

1.1 Object ——> getClass();  
1.2 任何数据类型（包括基本数据类型）都有一个“静态”的class属性  
1.3 通过Class类的静态方法：forName（String  className）(常用)

其中1.1是因为Object类中的getClass方法、因为所有类都继承Object类。从而调用Object类来获取



package fanshe;  
*/\*\*  
 \* 获取Class对象的三种方式  
 \* 1 Object ——> getClass();  
 \* 2 任何数据类型（包括基本数据类型）都有一个“静态”的class属性  
 \* 3 通过Class类的静态方法：forName（String className）(常用)  
 \*  
 \*/* public class Fanshe {  
 public static void main(String[] args) {  
 //第一种方式获取Class对象   
 Student stu1 = new Student();//这一new 产生一个Student对象，一个Class对象。  
 Class stuClass = stu1.getClass();//获取Class对象  
 System.*out*.println(stuClass.getName());  
  
 //第二种方式获取Class对象  
 Class stuClass2 = Student.class;  
 System.*out*.println(stuClass == stuClass2);//判断第一种方式获取的Class对象和第二种方式获取的是否是同一个  
  
 //第三种方式获取Class对象  
 try {  
 Class stuClass3 = Class.*forName*("fanshe.Student");//注意此字符串必须是真实路径，就是带包名的类路径，包名.类名  
 System.*out*.println(stuClass3 == stuClass2);//判断三种方式是否获取的是同一个Class对象  
 } catch (ClassNotFoundException e) {  
 e.printStackTrace();  
 }  
  
 }  
 }

**注意：在运行期间，一个类，只有一个Class对象产生。**

三种方式常用第三种，第一种对象都有了还要反射干什么。第二种需要导入类的包，依赖太强，不导包就抛编译错误。一般都第三种，一个字符串可以传入也可写在配置文件中等多种方法。

#### 通过反射获取构造方法并使用：

student类：

package fanshe;  
  
 public class Student {  
  
 //---------------构造方法-------------------  
 //（默认的构造方法）  
 Student(String str){  
 System.*out*.println("(默认)的构造方法 s = " + str);  
 }  
  
 //无参构造方法  
 public Student(){  
 System.*out*.println("调用了公有、无参构造方法执行了。。。");  
 }  
  
 //有一个参数的构造方法  
 public Student(char name){  
 System.*out*.println("姓名：" + name);  
 }  
  
 //有多个参数的构造方法  
 public Student(String name ,int age){  
 System.*out*.println("姓名："+name+"年龄："+ age);//这的执行效率有问题，以后解决。  
 }  
  
 //受保护的构造方法  
 protected Student(boolean n){  
 System.*out*.println("受保护的构造方法 n = " + n);  
 }  
  
 //私有构造方法  
 private Student(int age){  
 System.*out*.println("私有的构造方法 年龄："+ age);  
 }  
  
 }

共有6个构造方法；

测试类：

package fanshe;  
   
import java.lang.reflect.Constructor;  
   
   
/\*  
 \* 通过Class对象可以获取某个类中的：构造方法、成员变量、成员方法；并访问成员；  
 \*   
 \* 1.获取构造方法：  
 \* 1).批量的方法：  
 \* public Constructor[] getConstructors()：所有"公有的"构造方法  
 public Constructor[] getDeclaredConstructors()：获取所有的构造方法(包括私有、受保护、默认、公有)  
   
 \* 2).获取单个的方法，并调用：  
 \* public Constructor getConstructor(Class... parameterTypes):获取单个的"公有的"构造方法：  
 \* public Constructor getDeclaredConstructor(Class... parameterTypes):获取"某个构造方法"可以是私有的，或受保护、默认、公有；  
 \*   
 \* 调用构造方法：  
 \* Constructor-->newInstance(Object... initargs)  
 \*/  
 public class Constructors {  
  
 public static void main(String[] args) throws Exception {  
 //1.加载Class对象  
 Class clazz = Class.*forName*("fanshe.Student");  
  
  
 //2.获取所有公有构造方法  
 System.*out*.println("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*所有公有构造方法\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*");  
 Constructor[] conArray = clazz.getConstructors();  
 for(Constructor c : conArray){  
 System.*out*.println(c);  
 }  
  
  
 System.*out*.println("\*\*\*\*\*\*\*\*\*\*\*\*所有的构造方法(包括：私有、受保护、默认、公有)\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*");  
 conArray = clazz.getDeclaredConstructors();  
 for(Constructor c : conArray){  
 System.*out*.println(c);  
 }  
  
 System.*out*.println("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*获取公有、无参的构造方法\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*");  
 Constructor con = clazz.getConstructor(null);  
 //1>、因为是无参的构造方法所以类型是一个null,不写也可以：这里需要的是一个参数的类型，切记是类型  
 //2>、返回的是描述这个无参构造函数的类对象。  
  
 System.*out*.println("con = " + con);  
 //调用构造方法  
 Object obj = con.newInstance();  
 // System.out.println("obj = " + obj);  
 // Student stu = (Student)obj;  
  
 System.*out*.println("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*获取私有构造方法，并调用\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*");  
 con = clazz.getDeclaredConstructor(char.class);  
 System.*out*.println(con);  
 //调用构造方法  
 con.setAccessible(true);//暴力访问(忽略掉访问修饰符)  
 obj = con.newInstance('男');  
 }  
  
 }

后台输出：

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*所有公有构造方法\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

public fanshe.Student(java.lang.String,int)

public fanshe.Student(char)

public fanshe.Student()

\*\*\*\*\*\*\*\*\*\*\*\*所有的构造方法(包括：私有、受保护、默认、公有)\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

private fanshe.Student(int)

protected fanshe.Student(boolean)

public fanshe.Student(java.lang.String,int)

public fanshe.Student(char)

public fanshe.Student()

fanshe.Student(java.lang.String)

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*获取公有、无参的构造方法\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

con = public fanshe.Student()

调用了公有、无参构造方法执行了。。。

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*获取私有构造方法，并调用\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

public fanshe.Student(char)

姓名：男

**调用方法：**

1.获取构造方法：

  1).批量的方法：  
public Constructor[] getConstructors()：所有"公有的"构造方法  
            public Constructor[] getDeclaredConstructors()：获取所有的构造方法(包括私有、受保护、默认、公有)  
       
  2).获取单个的方法，并调用：  
public Constructor getConstructor(Class... parameterTypes):获取单个的"公有的"构造方法：  
public Constructor getDeclaredConstructor(Class... parameterTypes):获取"某个构造方法"可以是私有的，或受保护、默认、公有；  
  
  调用构造方法：

Constructor-->newInstance(Object... initargs)

2、newInstance是 Constructor类的方法（管理构造函数的类）

api的解释为：

newInstance(Object... initargs)  
           使用此 Constructor 对象表示的构造方法来创建该构造方法的声明类的新实例，并用指定的初始化参数初始化该实例。

它的返回值是T类型，所以newInstance是创建了一个构造方法的声明类的新实例对象。并为之调用

#### 获取成员变量并调用

student类：

package fanshe.field;  
  
 public class Student {  
 public Student(){  
  
 }  
 //\*\*\*\*\*\*\*\*\*\*字段\*\*\*\*\*\*\*\*\*\*\*\*\*//  
 public String name;  
 protected int age;  
 char sex;  
 private String phoneNum;  
  
 @Override  
 public String toString() {  
 return "Student [name=" + name + ", age=" + age + ", sex=" + sex  
 + ", phoneNum=" + phoneNum + "]";  
 }  
  
  
 }

测试类：

package fanshe.field;  
import java.lang.reflect.Field;  
 /\*  
 \* 获取成员变量并调用：  
 \*  
 \* 1.批量的  
 \* 1).Field[] getFields():获取所有的"公有字段"  
 \* 2).Field[] getDeclaredFields():获取所有字段，包括：私有、受保护、默认、公有；  
 \* 2.获取单个的：  
 \* 1).public Field getField(String fieldName):获取某个"公有的"字段；  
 \* 2).public Field getDeclaredField(String fieldName):获取某个字段(可以是私有的)  
 \*  
 \* 设置字段的值：  
 \* Field --> public void set(Object obj,Object value):  
 \* 参数说明：  
 \* 1.obj:要设置的字段所在的对象；  
 \* 2.value:要为字段设置的值；  
 \*  
 \*/  
 public class Fields {  
  
 public static void main(String[] args) throws Exception {  
 //1.获取Class对象  
 Class stuClass = Class.*forName*("fanshe.field.Student");  
 //2.获取字段  
 System.*out*.println("\*\*\*\*\*\*\*\*\*\*\*\*获取所有公有的字段\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*");  
 Field[] fieldArray = stuClass.getFields();  
 for(Field f : fieldArray){  
 System.*out*.println(f);  
 }  
 System.*out*.println("\*\*\*\*\*\*\*\*\*\*\*\*获取所有的字段(包括私有、受保护、默认的)\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*");  
 fieldArray = stuClass.getDeclaredFields();  
 for(Field f : fieldArray){  
 System.*out*.println(f);  
 }  
 System.*out*.println("\*\*\*\*\*\*\*\*\*\*\*\*\*获取公有字段\*\*并调用\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*");  
 Field f = stuClass.getField("name");  
 System.*out*.println(f);  
 //获取一个对象  
 Object obj = stuClass.getConstructor().newInstance();//产生Student对象--》Student stu = new Student();  
 //为字段设置值  
 f.set(obj, "刘德华");//为Student对象中的name属性赋值--》stu.name = "刘德华"  
 //验证  
 Student stu = (Student)obj;  
 System.*out*.println("验证姓名：" + stu.name);  
  
  
 System.*out*.println("\*\*\*\*\*\*\*\*\*\*\*\*\*\*获取私有字段\*\*\*\*并调用\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*");  
 f = stuClass.getDeclaredField("phoneNum");  
 System.*out*.println(f);  
 f.setAccessible(true);//暴力反射，解除私有限定  
 f.set(obj, "18888889999");  
 System.*out*.println("验证电话：" + stu);  
  
 }  
 }

后台输出：

\*\*\*\*\*\*\*\*\*\*\*\*获取所有公有的字段\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

public java.lang.String fanshe.field.Student.name

\*\*\*\*\*\*\*\*\*\*\*\*获取所有的字段(包括私有、受保护、默认的)\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

public java.lang.String fanshe.field.Student.name

protected int fanshe.field.Student.age

char fanshe.field.Student.sex

private java.lang.String fanshe.field.Student.phoneNum

\*\*\*\*\*\*\*\*\*\*\*\*\*获取公有字段\*\*并调用\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

public java.lang.String fanshe.field.Student.name

验证姓名：刘德华

\*\*\*\*\*\*\*\*\*\*\*\*\*\*获取私有字段\*\*\*\*并调用\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

private java.lang.String fanshe.field.Student.phoneNum

验证电话：Student [name=刘德华, age=0, sex=

由此可见

调用字段时：需要传递两个参数：

Object obj = stuClass.getConstructor().newInstance();//产生Student对象--》Student stu = new Student();  
//为字段设置值  
f.set(obj, "刘德华");//为Student对象中的name属性赋值--》stu.name = "刘德华"

第一个参数：要传入设置的对象，第二个参数：要传入实参

#### 获取成员方法并调用

student类：

package fanshe.method;  
  
 public class Student {  
 //\*\*\*\*\*\*\*\*\*\*\*\*\*\*成员方法\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*//  
 public void show1(String s){  
 System.*out*.println("调用了：公有的，String参数的show1(): s = " + s);  
 }  
 protected void show2(){  
 System.*out*.println("调用了：受保护的，无参的show2()");  
 }  
 void show3(){  
 System.*out*.println("调用了：默认的，无参的show3()");  
 }  
 private String show4(int age){  
 System.*out*.println("调用了，私有的，并且有返回值的，int参数的show4(): age = " + age);  
 return "abcd";  
 }  
 }

测试类：

package fanshe.method;  
   
import java.lang.reflect.Method;  
   
/\*  
 \* 获取成员方法并调用：  
 \*   
 \* 1.批量的：  
 \* public Method[] getMethods():获取所有"公有方法"；（包含了父类的方法也包含Object类）  
 \* public Method[] getDeclaredMethods():获取所有的成员方法，包括私有的(不包括继承的)  
 \* 2.获取单个的：  
 \* public Method getMethod(String name,Class<?>... parameterTypes):  
 \* 参数：  
 \* name : 方法名；  
 \* Class ... : 形参的Class类型对象  
 \* public Method getDeclaredMethod(String name,Class<?>... parameterTypes)  
 \*   
 \* 调用方法：  
 \* Method --> public Object invoke(Object obj,Object... args):  
 \* 参数说明：  
 \* obj : 要调用方法的对象；  
 \* args:调用方式时所传递的实参；  
):  
 \*/  
 public class MethodClass {  
  
 public static void main(String[] args) throws Exception {  
 //1.获取Class对象  
 Class stuClass = Class.*forName*("fanshe.method.Student");  
 //2.获取所有公有方法  
 System.*out*.println("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*获取所有的”公有“方法\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*");  
 stuClass.getMethods();  
 Method[] methodArray = stuClass.getMethods();  
 for(Method m : methodArray){  
 System.*out*.println(m);  
 }  
 System.*out*.println("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*获取所有的方法，包括私有的\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*");  
 methodArray = stuClass.getDeclaredMethods();  
 for(Method m : methodArray){  
 System.*out*.println(m);  
 }  
 System.*out*.println("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*获取公有的show1()方法\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*");  
 Method m = stuClass.getMethod("show1", String.class);  
 System.*out*.println(m);  
 //实例化一个Student对象  
 Object obj = stuClass.getConstructor().newInstance();  
 m.invoke(obj, "刘德华");  
  
 System.*out*.println("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*获取私有的show4()方法\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*");  
 m = stuClass.getDeclaredMethod("show4", int.class);  
 System.*out*.println(m);  
 m.setAccessible(true);//解除私有限定  
 Object result = m.invoke(obj, 20);//需要两个参数，一个是要调用的对象（获取有反射），一个是实参  
 System.*out*.println("返回值：" + result);  
  
  
 }  
 }

控制台输出：

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*获取所有的”公有“方法\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

public void fanshe.method.Student.show1(java.lang.String)

public final void java.lang.Object.wait(long,int) throws java.lang.InterruptedException

public final native void java.lang.Object.wait(long) throws java.lang.InterruptedException

public final void java.lang.Object.wait() throws java.lang.InterruptedException

public boolean java.lang.Object.equals(java.lang.Object)

public java.lang.String java.lang.Object.toString()

public native int java.lang.Object.hashCode()

public final native java.lang.Class java.lang.Object.getClass()

public final native void java.lang.Object.notify()

public final native void java.lang.Object.notifyAll()

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*获取所有的方法，包括私有的\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

public void fanshe.method.Student.show1(java.lang.String)

private java.lang.String fanshe.method.Student.show4(int)

protected void fanshe.method.Student.show2()

void fanshe.method.Student.show3()

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*获取公有的show1()方法\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

public void fanshe.method.Student.show1(java.lang.String)

调用了：公有的，String参数的show1(): s = 刘德华

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*获取私有的show4()方法\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

private java.lang.String fanshe.method.Student.show4(int)

调用了，私有的，并且有返回值的，int参数的show4(): age = 20

返回值：abcd

由此可见：

m = stuClass.getDeclaredMethod("show4", int.class);//调用制定方法（所有包括私有的），需要传入两个参数，第一个是调用的方法名称，第二个是方法的形参类型，切记是类型。  
System.out.println(m);  
m.setAccessible(true);//解除私有限定  
Object result = m.invoke(obj, 20);//需要两个参数，一个是要调用的对象（获取有反射），一个是实参  
System.out.println("返回值：" + result);//

控制台输出：

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*获取所有的”公有“方法\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

public void fanshe.method.Student.show1(java.lang.String)

public final void java.lang.Object.wait(long,int) throws java.lang.InterruptedException

public final native void java.lang.Object.wait(long) throws java.lang.InterruptedException

public final void java.lang.Object.wait() throws java.lang.InterruptedException

public boolean java.lang.Object.equals(java.lang.Object)

public java.lang.String java.lang.Object.toString()

public native int java.lang.Object.hashCode()

public final native java.lang.Class java.lang.Object.getClass()

public final native void java.lang.Object.notify()

public final native void java.lang.Object.notifyAll()

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*获取所有的方法，包括私有的\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

public void fanshe.method.Student.show1(java.lang.String)

private java.lang.String fanshe.method.Student.show4(int)

protected void fanshe.method.Student.show2()

void fanshe.method.Student.show3()

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*获取公有的show1()方法\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

public void fanshe.method.Student.show1(java.lang.String)

调用了：公有的，String参数的show1(): s = 刘德华

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*获取私有的show4()方法\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

private java.lang.String fanshe.method.Student.show4(int)

调用了，私有的，并且有返回值的，int参数的show4(): age = 20

返回值：abcd

其实这里的成员方法：在模型中有属性一词，就是那些setter（）方法和getter()方法。还有字段组成，这些内容在内省中详解

#### 反射main方法

student类：

package fanshe.main;  
  
 public class Student {  
  
 public static void main(String[] args) {  
 System.*out*.println("main方法执行了。。。");  
 }  
 }

测试类：

package fanshe.main;

import java.lang.reflect.Method;

/\*\*

\* 获取Student类的main方法、不要与当前的main方法搞混了

\*/

public class Main {

public static void main(String[] args) {

try {

//1、获取Student对象的字节码

Class clazz = Class.forName("fanshe.main.Student");

//2、获取main方法

Method methodMain = clazz.getMethod("main", String[].class);//第一个参数：方法名称，第二个参数：方法形参的类型，

//3、调用main方法

// methodMain.invoke(null, new String[]{"a","b","c"});

//第一个参数，对象类型，因为方法是static静态的，所以为null可以，第二个参数是String数组，这里要注意在jdk1.4时是数组，jdk1.5之后是可变参数

//这里拆的时候将 new String[]{"a","b","c"} 拆成3个对象。。。所以需要将它强转。

methodMain.invoke(null, (Object)new String[]{"a","b","c"});//方式一

// methodMain.invoke(null, new Object[]{new String[]{"a","b","c"}});//方式二

} catch (Exception e) {

e.printStackTrace();

}

}

}

控制台输出：

main方法执行了。。。

#### 反射方法的其它使用之---通过反射运行配置文件内容

student类：

public class Student {

public void show(){

System.out.println("is show()");

}

}

配置文件以txt文件为例子（pro.txt）：

className = cn.fanshe.Student

methodName = show

测试类：

import java.io.FileNotFoundException;  
import java.io.FileReader;  
import java.io.IOException;  
import java.lang.reflect.Method;  
import java.util.Properties;  
  
 /\*  
 \* 我们利用反射和配置文件，可以使：应用程序更新时，对源码无需进行任何修改  
 \* 我们只需要将新类发送给客户端，并修改配置文件即可  
 \*/  
 public class Demo {  
 public static void main(String[] args) throws Exception {  
 //通过反射获取Class对象  
 Class stuClass = Class.*forName*(*getValue*("className"));//"cn.fanshe.Student"  
 //2获取show()方法  
 Method m = stuClass.getMethod(*getValue*("methodName"));//show  
 //3.调用show()方法  
 m.invoke(stuClass.getConstructor().newInstance());  
  
 }  
  
 //此方法接收一个key，在配置文件中获取相应的value  
 public static String getValue(String key) throws IOException{  
 Properties pro = new Properties();//获取配置文件的对象  
 FileReader in = new FileReader("pro.txt");//获取输入流  
 pro.load(in);//将流加载到配置文件对象中  
 in.close();  
 return pro.getProperty(key);//返回根据key获取的value值  
 }  
 }

控制台输出：

is show()

**需求：**  
当我们升级这个系统时，不要Student类，而需要新写一个Student2的类时，这时只需要更改pro.txt的文件内容就可以了。代码就一点不用改动

要替换的student2类：

public class Student2 {

public void show2(){

System.out.println("is show2()");

}

}

配置文件更改为：

className = cn.fanshe.Student2

methodName = show2

控制台输出：

is show2();

#### 反射方法的其它使用之---通过反射越过泛型检查

泛型用在编译期，编译过后泛型擦除（消失掉）。所以是可以通过反射越过泛型检查的

测试类：

import java.lang.reflect.Method;  
import java.util.ArrayList;  
  
 /\*  
 \* 通过反射越过泛型检查  
 \*  
 \* 例如：有一个String泛型的集合，怎样能向这个集合中添加一个Integer类型的值？  
 \*/  
 public class Demo {  
 public static void main(String[] args) throws Exception{  
 ArrayList<String> strList = new ArrayList<>();  
 strList.add("aaa");  
 strList.add("bbb");  
  
 // strList.add(100);  
 //获取ArrayList的Class对象，反向的调用add()方法，添加数据  
 Class listClass = strList.getClass(); //得到 strList 对象的字节码 对象  
 //获取add()方法  
 Method m = listClass.getMethod("add", Object.class);  
 //调用add()方法  
 m.invoke(strList, 100);  
  
 //遍历集合  
 for(Object obj : strList){  
 System.*out*.println(obj);  
 }  
 }  
 }

控制台输出：

aaa  
bbb  
100

--------------------- 本文来自 敬业的小码哥 的CSDN 博客 ，全文地址请点击：<https://blog.csdn.net/sinat_38259539/article/details/71799078?utm_source=copy>

## 时间复杂度

### 概述

时间频度  
一个算法执行所耗费的时间，从理论上是不能算出来的，必须上机运行[测试](http://lib.csdn.net/base/softwaretest)才能知道。但我们不可能也没有必要对每个算法都上机测试，只需知道哪个算法花费的时间多，哪个算法花费的时间少就可以了。并且一个算法花费的时间与算法中语句的执行次数成正比例，哪个算法中语句执行次数多，它花费时间就多。一个算法中的语句执行次数称为语句频度或时间频度。记为T(n)。   
  
时间复杂度   
  
在刚才提到的时间频度中，n称为问题的规模，当n不断变化时，时间频度T(n)也会不断变化。但有时我们想知道它变化时呈现什么规律。为此，我们引入时间复杂度概念。   
  
一般情况下，算法中基本操作重复执行的次数是问题规模n的某个函数，用T(n)表示，若有某个辅助函数f(n),使得当n趋近于无穷大时，T（n)/f(n)的极限值为不等于零的常数，则称f(n)是T(n)的同数量级函数。记作T(n)=O(f(n)),称O(f(n)) 为算法的渐进时间复杂度，简称时间复杂度。   
  
在各种不同算法中，若算法中语句执行次数为一个常数，则时间复杂度为O(1),另外，在时间频度不相同时，时间复杂度有可能相同，如T(n)=n2+3n+4与T(n)=4n2+2n+1它们的频度不同，但时间复杂度相同，都为O(n2)。   
  
按数量级递增排列，常见的时间复杂度有：   
  
常数阶O(1),对数阶O(log2n),线性阶O(n),   
  
线性对数阶O(nlog2n),平方阶O(n2)，立方阶O(n3),...，   
  
k次方阶O(nk),指数阶O(2n)。随着问题规模n的不断增大，上述时间复杂度不断增大，算法的执行效率越低。   
  
2、空间复杂度   
  
与时间复杂度类似，空间复杂度是指算法在计算机内执行时所需存储空间的度量。记作:   
  
S(n)=O(f(n))   
  
我们一般所讨论的是除正常占用内存开销外的辅助存储单元规模

### 常见算法时间复杂度：

O(1): 表示算法的运行时间为常量

O(n): 表示该算法是线性算法

O(㏒2n): 二分查找算法

O(n2): 对数组进行排序的各种简单算法，例如直接插入排序的算法。

O(n3): 做两个n阶矩阵的乘法运算

O(2n): 求具有n个元素集合的所有子集的算法

O(n!): 求具有N个元素的全排列的算法

优<---------------------------<劣

O(1)<O(㏒2n)<O(n)<O(n2)<O(2n)

时间复杂度按数量级递增排列依次为：常数阶O(1)、对数阶O(log2n)、线性阶O(n)、线性对数阶O(nlog2n)、平方阶O(n2)、立方阶O(n3)、……k次方阶O(nk)、指数阶O(2n)。

### 算法的时间复杂度（计算实例）

定义：如果一个问题的规模是n，解这一问题的某一算法所需要的时间为T(n)，它是n的某一函数 T(n)称为这一算法的“时间复杂性”。

当输入量n逐渐加大时，时间复杂性的极限情形称为算法的“渐近时间复杂性”。

我们常用大O表示法表示时间复杂性，注意它是某一个算法的时间复杂性。大O表示只是说有上界，由定义如果f(n)=O(n)，那显然成立f(n)=O(n^2)，它给你一个上界，但并不是上确界，但人们在表示的时候一般都习惯表示前者。

此外，一个问题本身也有它的复杂性，如果某个算法的复杂性到达了这个问题复杂性的下界，那就称这样的算法是最佳算法。

“大O记法”：在这种描述中使用的基本参数是 n，即问题实例的规模，把复杂性或运行时间表达为n的函数。这里的“O”表示量级 (order)，比如说“二分检索是 O(logn)的”,也就是说它需要“通过logn量级的步骤去检索一个规模为n的数组”记法 O ( f(n) )表示当 n增大时，运行时间至多将以正比于 f(n)的速度增长。

这种渐进估计对算法的理论分析和大致比较是非常有价值的，但在实践中细节也可能造成差异。例如，一个低附加代价的O(n2)算法在n较小的情况下可能比一个高附加代价的 O(nlogn)算法运行得更快。当然，随着n足够大以后，具有较慢上升函数的算法必然工作得更快。

O(1)

Temp=i;i=j;j=temp;

以上三条单个语句的频度均为1，该程序段的执行时间是一个与问题规模n无关的常数。算法的时间复杂度为常数阶，记作T(n)=O(1)。如果算法的执行时 间不随着问题规模n的增加而增长，即使算法中有上千条语句，其执行时间也不过是一个较大的常数。此类算法的时间复杂度是O(1)。

O(n^2)

2.1. 交换i和j的内容

     sum=0；                 （一次）

     for(i=1;i<=n;i++)       （n次 ）

        for(j=1;j<=n;j++) （n^2次 ）

         sum++；       （n^2次 ）

解：T(n)=2n^2+n+1 =O(n^2)

2.2.

    for (i=1;i<n;i++)

    {

        y=y+1;         ①

        for (j=0;j<=(2\*n);j++)

           x++;        ②

    }

解： 语句1的频度是n-1

          语句2的频度是(n-1)\*(2n+1)=2n^2-n-1

          f(n)=2n^2-n-1+(n-1)=2n^2-2

          该程序的时间复杂度T(n)=O(n^2).

O(n)

2.3.//存疑

    a=0;

    b=1;                      ①

    for (i=1;i<=n;i++) ②

    {

       s=a+b;　　　　③

       b=a;　　　　　④

       a=s;　　　　　⑤

    }

解： 语句1的频度：2,

           语句2的频度： n,

          语句3的频度： n-1,

          语句4的频度：n-1,

          语句5的频度：n-1,

          T(n)=2+n+3(n-1)=4n-1=O(n).

O(log2n )

2.4.

     i=1;       ①

    while (i<=n)

       i=i\*2; ②

解： 语句1的频度是1,

          设语句2的频度是f(n),   则：2^f(n)<=n;f(n)<=log2n

          取最大值f(n)= log2n,

          T(n)=O(log2n )

O(n^3)

2.5.

    for(i=0;i<n;i++)

    {

       for(j=0;j<i;j++)

       {

          for(k=0;k<j;k++)

             x=x+2;

       }

    }

解：当i=m, j=k的时候,内层循环的次数为k当i=m时, j 可以取 0,1,...,m-1 , 所以这里最内循环共进行了0+1+...+m-1=(m-1)m/2次所以,i从0取到n, 则循环共进行了: 0+(1-1)\*1/2+...+(n-1)n/2=n(n+1)(n-1)/6所以时间复杂度为O(n^3).

我们还应该区分算法的最坏情况的行为和期望行为。如快速排序的最 坏情况运行时间是 O(n^2)，但期望时间是 O(nlogn)。通过每次都仔细 地选择基准值，我们有可能把平方情况 (即O(n^2)情况)的概率减小到几乎等于 0。在实际中，精心实现的快速排序一般都能以 (O(nlogn)时间运行。

下面是一些常用的记法：

访问数组中的元素是常数时间操作，或说O(1)操作。一个算法如果能在每个步骤去掉一半数据元素，如二分检索，通常它就取 O(logn)时间。用strcmp比较两个具有n个字符的串需要O(n)时间 。常规的矩阵乘算法是O(n^3)，因为算出每个元素都需要将n对 元素相乘并加到一起，所有元素的个数是n^2。

指数时间算法通常来源于需要求出所有可能结果。例如，n个元 素的集合共有2n个子集,所以要求出所有子集的算法将是O(2n)的 。指数算法一般说来是太复杂了，除非n的值非常小，因为，在 这个问题中增加一个元素就导致运行时间加倍。不幸的是，确实有许多问题 (如著名 的“巡回售货员问题” )，到目前为止找到的算法都是指数的。如果我们真的遇到这种情况， 通常应该用寻找近似最佳结果的算法替代之。