目 录

[第一章 java-gui 8](#_Toc518286056)

[第1节 Frame类 9](#_Toc518286057)

[第2节 Panel 9](#_Toc518286058)

[第3节 LayoutManager 10](#_Toc518286059)

[1. FlowLayout 10](#_Toc518286060)

[2. borderLayout 10](#_Toc518286061)

[3. GridLayout 12](#_Toc518286062)

[第4节 事件监听 13](#_Toc518286063)

[第5节 TextField类 14](#_Toc518286064)

[第6节 Graphics类 17](#_Toc518286065)

[第7节 WindowListener 17](#_Toc518286066)

[第8节 MouseAdapte类 18](#_Toc518286067)

[第9节 KeyListener 20](#_Toc518286068)

[第二章 Java-Thread 22](#_Toc518286069)

[第1节 Runtime与ProcessBuider 22](#_Toc518286070)

[第2节 Thread与Runnable 22](#_Toc518286071)

[第3节 线程安全 25](#_Toc518286072)

[第4节 线程的通信 26](#_Toc518286073)

[第5节 线程的阻塞 29](#_Toc518286074)

[1. 哲学家就餐 29](#_Toc518286075)

[第6节 Thread status 31](#_Toc518286076)

[1. 新建状态（new）： 31](#_Toc518286077)

[2. 可运行状态（RUNNABLE）： 32](#_Toc518286078)

[3. 阻塞状态（BLOCKED） 32](#_Toc518286079)

[4. 等待（WAITING） 32](#_Toc518286080)

[5. 计时等待（TIMED\_WAITING） 32](#_Toc518286081)

[6. 死亡状态（TERMINATED ） 32](#_Toc518286082)

[第7节 线程睡眠 33](#_Toc518286083)

[第8节 联合线程 34](#_Toc518286084)

[第9节 守护线程 34](#_Toc518286085)

[第10节 线程优先级 35](#_Toc518286086)

[第11节 线程礼让 36](#_Toc518286087)

[第12节 死锁 36](#_Toc518286088)

[第三章 Java反射 38](#_Toc518286089)

[第1节 Class类 38](#_Toc518286090)

[第2节 加载 38](#_Toc518286091)

[第3节 Method类 39](#_Toc518286092)

[第4节 Field类 39](#_Toc518286093)

[第四章 Java通信 42](#_Toc518286094)

[第1节 TCP/IP协议 42](#_Toc518286095)

[第2节 Inetaddress: 42](#_Toc518286096)

[第3节 URL 43](#_Toc518286097)

[第4节 Socket 44](#_Toc518286098)

[第5节 UDP 48](#_Toc518286099)

[第五章 Java注解 51](#_Toc518286100)

[第1节 JDK自带注解 51](#_Toc518286101)

[第2节 自定义注解 51](#_Toc518286102)

[第六章 IO 58](#_Toc518286103)

[第1节 FILE类 58](#_Toc518286104)

[第2节 FilenameFilter接口 60](#_Toc518286105)

[第3节 IO 60](#_Toc518286106)

[第4节 文件的字节流： 64](#_Toc518286107)

[第5节 包装流： 64](#_Toc518286108)

[第6节 缓冲流： 65](#_Toc518286109)

[第7节 转换流： 66](#_Toc518286110)

[第8节 内存流 66](#_Toc518286111)

[1. 字节内存流 66](#_Toc518286112)

[2. 字符内存流 67](#_Toc518286113)

[3. 字符串流 68](#_Toc518286114)

[第9节 合并流 68](#_Toc518286115)

[第10节 序列化与反序列化 69](#_Toc518286116)

[第11节 打印流 70](#_Toc518286117)

[第12节 标准输入输出流 71](#_Toc518286118)

[第13节 Scanner 72](#_Toc518286119)

[第14节 数据流 73](#_Toc518286120)

[第15节 RandomAccessFile 74](#_Toc518286121)

[第16节 管道流 75](#_Toc518286122)

[第17节 NIO 76](#_Toc518286123)

[第七章 数据结构与算法 78](#_Toc518286124)

[第1节 HelloWorld 78](#_Toc518286125)

[第2节 字符串 80](#_Toc518286126)

[1. 静态存储结构 81](#_Toc518286127)

[2. 动态存储结构 81](#_Toc518286128)

[第3节 数组 81](#_Toc518286129)

[1. 矩阵压缩 81](#_Toc518286130)

[第4节 ArrayList 81](#_Toc518286131)

[第5节 链表 84](#_Toc518286132)

[第6节 队列 87](#_Toc518286133)

[第7节 栈 88](#_Toc518286134)

[第8节 散列表 89](#_Toc518286135)

[第9节 树 90](#_Toc518286136)

[1. 二叉树 90](#_Toc518286137)

[2. 二叉树与普通树之间的转换 96](#_Toc518286138)

[3. 赫夫曼树与赫夫曼编码 97](#_Toc518286139)

[第10节 图 97](#_Toc518286140)

[1. 图的基本概念 97](#_Toc518286141)

[2. 图的存储结构： 100](#_Toc518286142)

[3. 马踏棋盘问题 102](#_Toc518286143)

[4. 图的最小生成树： 102](#_Toc518286144)

[5. 最短路径问题 107](#_Toc518286145)

[6. 拓扑排序 112](#_Toc518286146)

[第11节 查找 115](#_Toc518286147)

[1. 插值查找 115](#_Toc518286148)

[2. 斐波那契查找 115](#_Toc518286149)

[3. 线性索引 116](#_Toc518286150)

[4. 二叉排序树 116](#_Toc518286151)

[5. 二衡二叉排序树 121](#_Toc518286152)

[6. 2-3树 129](#_Toc518286153)

[7. 2-3-4树 136](#_Toc518286154)

[8. B树 136](#_Toc518286155)

[9. 散列表（哈希表）查找 136](#_Toc518286156)

[第12节 排序 139](#_Toc518286157)

[1. 冒泡排序 139](#_Toc518286158)

[2. 选择排序 141](#_Toc518286159)

[3. 简单插入排序 142](#_Toc518286160)

[4. 希尔排序 143](#_Toc518286161)

[5. 堆排序 144](#_Toc518286162)

[6. 归并排序 145](#_Toc518286163)

[7. 快速排序 147](#_Toc518286164)

[第八章 加密解密 149](#_Toc518286165)

[第1节 BASE64编码 149](#_Toc518286166)

[第2节 消息摘要 149](#_Toc518286167)

[1. 消息摘要 149](#_Toc518286168)

[2. 安全散列 150](#_Toc518286169)

[3. 消息认证码 150](#_Toc518286170)

[第3节 对称加密 151](#_Toc518286171)

[1. DES 151](#_Toc518286172)

[2. DESede 151](#_Toc518286173)

[3. AES 152](#_Toc518286174)

[第4节 非对称加密 152](#_Toc518286175)

[1. DH 152](#_Toc518286176)

[2. RSA 155](#_Toc518286177)

[第5节 数字签名 156](#_Toc518286178)

[1. IDSA 156](#_Toc518286179)

[2. DECDSA 157](#_Toc518286180)

[3. IRSA 158](#_Toc518286181)

[第九章 设计模式 160](#_Toc518286182)

[第1节 设计模式分类： 160](#_Toc518286183)

[1. 创建型模式 160](#_Toc518286184)

[2. 结构型模式 160](#_Toc518286185)

[3. 行为型模式 160](#_Toc518286186)

[4. J2EE 模式 160](#_Toc518286187)

[5. 设计模式的六大原则 161](#_Toc518286188)

[第2节 工厂模式 161](#_Toc518286189)

[第3节 抽象工厂模式 162](#_Toc518286190)

[第4节 单例模式 163](#_Toc518286191)

[1. 懒汉式，线程安全 163](#_Toc518286192)

[2. 饿汉式 163](#_Toc518286193)

[3. 双检锁/双重校验 164](#_Toc518286194)

[4. 登记式/静态内部类 164](#_Toc518286195)

[5. 枚举 164](#_Toc518286196)

[第5节 建造者模式 165](#_Toc518286197)

[第6节 原型模式 166](#_Toc518286198)

[第7节 适配器模式 167](#_Toc518286199)

[第8节 桥接模式 168](#_Toc518286200)

[第9节 命令模式 170](#_Toc518286201)

[第10节 静态代理 172](#_Toc518286202)

[第11节 常用的代理模式 172](#_Toc518286203)

[1. 继承方式实现代理 173](#_Toc518286204)

[2. 聚合的方式实现代理 173](#_Toc518286205)

[3. 两者比较 174](#_Toc518286206)

[第12节 动态代理 175](#_Toc518286207)

[第13节 责任链模式 178](#_Toc518286208)

[第14节 观察者模式 180](#_Toc518286209)

[第15节 空对象模式 182](#_Toc518286210)

[第十章 JVM原理之Call\_stub 184](#_Toc518286211)

[第1节 Castable\_address()定义： 184](#_Toc518286212)

[1. 连接器Link 185](#_Toc518286213)

[2. Method() 185](#_Toc518286214)

[3. Entry\_point 185](#_Toc518286215)

[4. Parameters() 186](#_Toc518286216)

[5. Size\_of\_parameters() 186](#_Toc518286217)

[第2节 \_call\_stub\_entry例程 186](#_Toc518286218)

[1. Pc() 186](#_Toc518286219)

[2. 定义入参 186](#_Toc518286220)

[3. CallStub保存调用者堆栈 186](#_Toc518286221)

[4. CallStub动态分配堆栈 186](#_Toc518286222)

[5. Callstub调用者保存 186](#_Toc518286223)

[6. CallStub 参数压栈 187](#_Toc518286224)

[7. 调用entry\_point例程 187](#_Toc518286225)

[第十一章 JVM原理之.class 188](#_Toc518286226)

[第1节 字节码结构 188](#_Toc518286227)

[1. 模数 188](#_Toc518286228)

[2. 版本 188](#_Toc518286229)

[3. 常量池 188](#_Toc518286230)

[4. 访问标志 190](#_Toc518286231)

[5. 本类索引 190](#_Toc518286232)

[6. 父类索引 190](#_Toc518286233)

[7. 接口索引 191](#_Toc518286234)

[8. 字段表 191](#_Toc518286235)

[9. 方法表 192](#_Toc518286236)

[10. 属性表 192](#_Toc518286237)

[第2节 解析示例 197](#_Toc518286238)

[第十二章 JVM原理之内存管理 205](#_Toc518286239)

[第1节 运行时数据区 205](#_Toc518286240)

[1. 方法区 205](#_Toc518286241)

[2. 运行时常量池 206](#_Toc518286242)

[3. 虚拟机栈 206](#_Toc518286243)

[4. 本地方法栈 206](#_Toc518286244)

[5. 堆 207](#_Toc518286245)

[6. 程序计数器 207](#_Toc518286246)

[7. 直接内存 208](#_Toc518286247)

[第2节 对象 208](#_Toc518286248)

[1. 创建对象 208](#_Toc518286249)

[2. 内存分配 209](#_Toc518286250)

[3. 担保 209](#_Toc518286251)

[4. 对象布局 210](#_Toc518286252)

[5. 对象访问 210](#_Toc518286253)

[第3节 GC 210](#_Toc518286254)

[1. 引用计数法 210](#_Toc518286255)

[2. 可达性分析法 211](#_Toc518286256)

[3. 再谈引用 211](#_Toc518286257)

[4. 生死 211](#_Toc518286258)

[5. 回收方法区 212](#_Toc518286259)

[6. 回收算法 213](#_Toc518286260)

[7. 枚举根节点 214](#_Toc518286261)

[8. 安全点 214](#_Toc518286262)

[9. 安全区域 214](#_Toc518286263)

[第4节 垃圾收集器 215](#_Toc518286264)

[1. Serial收集器 215](#_Toc518286265)

[2. ParNew收集器 215](#_Toc518286266)

[3. Parallel Scavenge收集器 216](#_Toc518286267)

[4. Serial Old收集器 216](#_Toc518286268)

[5. Parallel Old收集器 216](#_Toc518286269)

[6. CMS收集器 216](#_Toc518286270)

[7. G1收集器 217](#_Toc518286271)

[第5节 内存交互基础 219](#_Toc518286272)

[第6节 volatile 220](#_Toc518286273)

[第7节 Long和double 222](#_Toc518286274)

[第8节 原子性、可见性、有序性 222](#_Toc518286275)

[第十三章 JVM原理之工具 223](#_Toc518286276)

[第1节 JPS 223](#_Toc518286277)

[第2节 Jstat 223](#_Toc518286278)

[1. jstat –class<pid> 224](#_Toc518286279)

[2. jstat -compiler <pid> 224](#_Toc518286280)

[3. jstat -gc <pid> 224](#_Toc518286281)

[4. jstat -gccapacity <pid> 224](#_Toc518286282)

[5. jstat -gcutil <pid> 225](#_Toc518286283)

[6. jstat -gcnew <pid> 225](#_Toc518286284)

[7. jstat -gcnewcapacity<pid> 226](#_Toc518286285)

[8. jstat -gcold <pid> 226](#_Toc518286286)

[9. stat -gcoldcapacity <pid> 226](#_Toc518286287)

[10. jstat -gcpermcapacity<pid> 227](#_Toc518286288)

[11. jstat -printcompilation <pid> 227](#_Toc518286289)

[第3节 jinfo 227](#_Toc518286290)

[第4节 Jmap:java内存映像工具 227](#_Toc518286291)

[第5节 jconsole 228](#_Toc518286292)

[第十四章 JVM原理之类加载 229](#_Toc518286293)

[第1节 类的加载时机 229](#_Toc518286294)

[第2节 类的加载过程 229](#_Toc518286295)

[1. 加载 229](#_Toc518286296)

[2. 验证 230](#_Toc518286297)

[3. 准备 231](#_Toc518286298)

[4. 解析 231](#_Toc518286299)

[5. 初始化 231](#_Toc518286300)

[第3节 类与类加载器 232](#_Toc518286301)

[1. 类与类加载器 232](#_Toc518286302)

[2. 双亲委派模型 232](#_Toc518286303)

[第十五章 虚拟机字节码执行引擎 234](#_Toc518286304)

[第1节 运行时栈帧结构 234](#_Toc518286305)

[1. 局部变量表 234](#_Toc518286306)

[2. 操作数栈 234](#_Toc518286307)

[3. 动态连接 234](#_Toc518286308)

[4. 方法返回地址 235](#_Toc518286309)

[第2节 方法调用 235](#_Toc518286310)

[1. 解析 235](#_Toc518286311)

[2. 分派 235](#_Toc518286312)

[第3节 基于栈的指令集与解释器 241](#_Toc518286313)

[第4节 字节码生成与动态代表 241](#_Toc518286314)

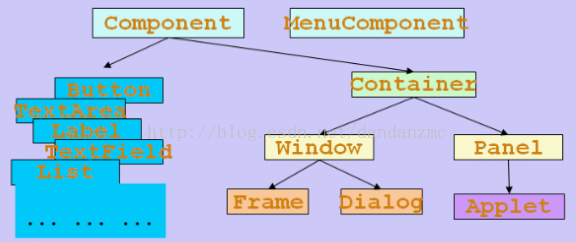
1. java-gui

主要内容：

* AWT
* 组件和容器
* 布局管理器
* 事件处理
* JAVA图形
* Window事件

GUI全称Graphical User Interfaces，意为图形用户户界面，又称为图形用户接口，GUI指的就是采用图形方式显示的计算机操作用户界面。

实现GUI编程的必不可少的三个条件是组件、事件。



**Component类：**

Java图形用户界面最基本组成部分是Component，Component类及其子类的对象用来描述以图形化的方式显示在屏幕上并能够与用户进行交互的GUI元素（标签、按钮）。[所有可以显示的元素都可以叫component]

**Container类：**

用来组织界面上的组件或者单元。有两种常用的Container（容器），一是Window，Window对象表示自由停泊的顶级窗口，另一个是Panel对象可作为容纳其他Component对象，但不能够独立存在，必须被添加到其他Container中，比如说Window或者Applet中。所认，Container是一个特殊的Component对象

Container是Comonent的子类，Container子类对象可以容纳别的Component对象。

Container对象可以使用add()向其中添加其它的Component对象。

Containter是Component的子类，因此Containter对象也可以被当作Component对象添加到其它的Container对象之中

有两种常用的Containter：

Window:其对象表示自由停泊的顶级窗口。

Panel：其对象可作为容纳其它Component对象，但不能独立存在显示，必须被添加到其它Container中

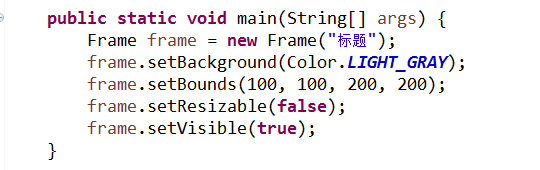
* 1. Frame类

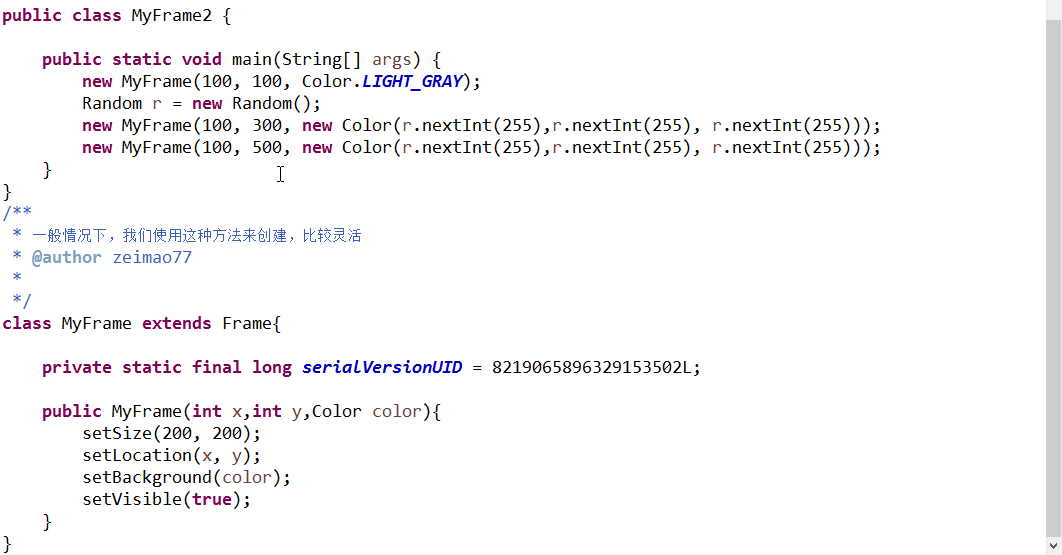
Frame是Window的一个子类，由Frame或其子类创建的对象为一个窗体。

Frame的常用构造方法

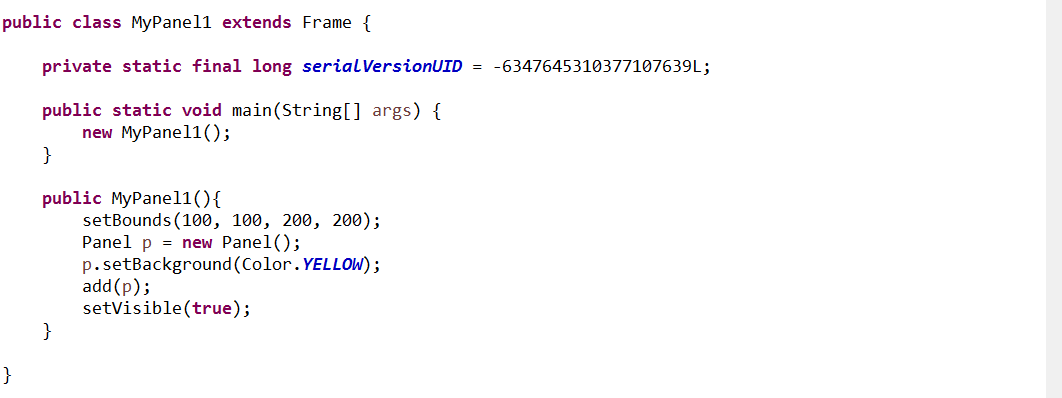
Frame()

Frame(String s)创建标题栏为s的窗口





* 1. Panel



* 1. LayoutManager
     1. FlowLayout

流式布局管理器

特点：逐行排满，是Panel的默认布局方式

构造方法：

FlowLayout()

构造一个新的 FlowLayout中心对齐和默认的5单位水平和垂直间隙。

FlowLayout(int align)

构造一个新的 FlowLayout具有指定的对齐和默认的5单位水平和垂直间隙。

FlowLayout(int align, int hgap, int vgap)

创建一个新的流程布局管理器，具有指示的对齐方式和指示的水平和垂直间距。

new FlowLayout(FlowLayout.RIGHT,20,40);

* + 1. borderLayout

**边界布局管理器：**

特点：将整个容器划分成五个区域：东（EAST）、西(WEST)、南(SOUTH)、北(NORTH)、中(CENTER)

如果不指定将默认添加到CENTER

每个区域只可以添加一个组件，如果加入多个，先加入的将被覆盖

是Frame默认的布局方式。

**缩放原则：**

北南两个区域在水平方向缩放

东西两个区域在垂直方向缩放

中部可以在两个方向缩放



package com.learn.awt.frame;

import java.awt.BorderLayout;

import java.awt.Color;

import java.awt.Frame;

import java.awt.Label;

import java.awt.Panel;

public class BorderLayoutDemo extends Frame{

private static final long serialVersionUID = 6283419465913104869L;

public static void main(String[] args) {

new BorderLayoutDemo();

}

public BorderLayoutDemo(){

setBounds(100, 100, 300,150);

add(new MyPanel(Color.YELLOW,"NORTH"),BorderLayout.NORTH);

add(new MyPanel(Color.CYAN,"CENTER"),BorderLayout.CENTER);

add(new MyPanel(Color.GRAY,"WEST"),BorderLayout.WEST);

add(new MyPanel(Color.PINK,"SOUTH"),BorderLayout.SOUTH);

add(new MyPanel(Color.ORANGE,"EAST"),BorderLayout.EAST);

setVisible(true);

}

}

class MyPanel extends Panel{

private static final long serialVersionUID = 8080383375776647577L;

public MyPanel(Color color,String str){

setLayout(new BorderLayout());

add(new Label(str));

setBackground(color);

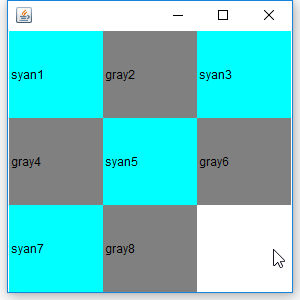
}

}

* + 1. GridLayout

**网格布局处理器**

特点：GridLayout型布局管理器将空间划分成规则的矩形风格，每个单元格区域大小相等，组件被添加到每个单元格中，从左到右依次填满。需要在构造方法中指定分割的行数与列数。



package com.learn.awt.frame;

import java.awt.BorderLayout;

import java.awt.Color;

import java.awt.Frame;

import java.awt.GridLayout;

import java.awt.Label;

import java.awt.Panel;

public class GridLayoutDemo extends Frame {

private static final long serialVersionUID = -1359868375805162302L;

public static void main(String[] args) {

new GridLayoutDemo();

}

public GridLayoutDemo(){

setSize(300, 300);

setLayout(new GridLayout(3, 3));

add(new MyPanel2(Color.cyan, "syan1"));

add(new MyPanel2(Color.gray, "gray2"));

add(new MyPanel2(Color.cyan, "syan3"));

add(new MyPanel2(Color.gray, "gray4"));

add(new MyPanel2(Color.cyan, "syan5"));

add(new MyPanel2(Color.gray, "gray6"));

add(new MyPanel2(Color.cyan, "syan7"));

add(new MyPanel2(Color.gray, "gray8"));

setVisible(true);

}

}

class MyPanel2 extends Panel{

private static final long serialVersionUID = 8080383375776647577L;

public MyPanel2(Color color,String str){

setLayout(new BorderLayout());

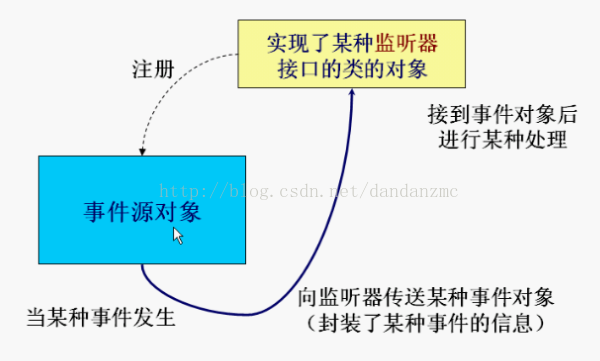
add(new Label(str));

setBackground(color);

}

}

* 1. 事件监听



package com.learn.awt.frame;

import java.awt.BorderLayout;

import java.awt.Button;

import java.awt.Frame;

import java.awt.event.ActionEvent;

import java.awt.event.ActionListener;

public class ActionEventDemo extends Frame {

private static final long serialVersionUID = -152366444189648932L;

public static void main(String[] args) {

new ActionEventDemo();

}

public ActionEventDemo(){

setSize(100, 100);

Button b1 = new Button("button1");

Button b2 = new Button("button2");

Monitor m = new Monitor();

b1.addActionListener(m);

b2.addActionListener(m);

add(b1,BorderLayout.WEST);

add(b2);

setVisible(true);

}

}

class Monitor implements ActionListener{

@Override

public void actionPerformed(ActionEvent e) {

System.out.println("a button has benn pressed!");

if(e.getActionCommand().equals("button1")){

System.out.println("button1 has benn pressed!");

}else if(e.getActionCommand().equals("button2")){

System.out.println("button2 has benn pressed!");

}

}

}

* 1. TextField类

用来创建文本框对象

直接对文本框的监听：

package com.learn.awt.frame;

import java.awt.Frame;

import java.awt.TextField;

import java.awt.event.ActionEvent;

import java.awt.event.ActionListener;

public class TextFieldDemo extends Frame {

private static final long serialVersionUID = 939666547456834156L;

public static void main(String[] args) {

new TextFieldDemo();

}

public TextFieldDemo(){

setSize(200, 200);

TextField t = new TextField();

t.setEchoChar('●');

t.addActionListener(new TfActionListener2());

add(t);

setVisible(true);

}

}

class TfActionListener2 implements ActionListener{

@Override

public void actionPerformed(ActionEvent e) {

TextField t = (TextField) e.getSource();

System.out.println(t.getText());

t.setText("");

}

}

监听按钮对文本框取值

package com.learn.awt.frame;

import java.awt.Button;

import java.awt.FlowLayout;

import java.awt.Frame;

import java.awt.Label;

import java.awt.TextField;

import java.awt.event.ActionEvent;

import java.awt.event.ActionListener;

public class TrFrameDemo extends Frame {

private static final long serialVersionUID = 2970566227728657178L;

public TextField f1,f2,f3;

public static void main(String[] args) {

new TrFrameDemo();

}

public TrFrameDemo(){

f1=new TextField(10);

f2=new TextField(10);

f3=new TextField(12);

Label l = new Label("+");

Button b = new Button("=");

b.setActionCommand("eq");

setLayout(new FlowLayout());

b.addActionListener(new MyMonitor());

add(f1);

add(l);

add(f2);

add(b);

add(f3);

pack();

setVisible(true);

}

/\*\*

\* 内部类可以直接访问包装类的成员变量和方法

\* 内部类可以有效防止其它类引用此类

\* @author zeimao77

\*

\*/

private class MyMonitor implements ActionListener{

@Override

public void actionPerformed(ActionEvent e) {

if(e.getActionCommand().equals("eq")){

Integer n1 = Integer.parseInt(f1.getText());

Integer n2 = Integer.parseInt(f2.getText());

f3.setText(Integer.sum(n1, n2)+"");

}

}

}

}

* 1. Graphics类

所有Component都有一个paint(Graphics g)用于实现绘图目的，每次重画该Component时都会自动调用paint方法

package com.learn.awt.frame;

import java.awt.Color;

import java.awt.Frame;

import java.awt.Graphics;

public class GraphicsDemo extends Frame {

private static final long serialVersionUID = 2023941672200383085L;

public static void main(String[] args) {

new GraphicsDemo();

}

public GraphicsDemo(){

setBackground(Color.GREEN);

setSize(200, 200);

setVisible(true);

}

@Override

public void paint(Graphics g) {

Color c= g.getColor();

g.setColor(Color.CYAN);

g.fillOval(50, 50, 50, 100);

g.setColor(Color.RED);

g.fillArc(50, 50,6,6,0,200);

g.setColor(c);

}

}

* 1. WindowListener

|  |  |
| --- | --- |
| windowActivated(WindowEvent e) | 当窗口设置为活动窗口时调用。 |
| windowClosed(WindowEvent e) | 当窗口关闭时调用窗口调用处理结果时调用。 |
| windowClosing(WindowEvent e) | 当用户尝试从窗口的系统菜单中关闭窗口时调用。 |
| windowDeactivated(WindowEvent e) | 当窗口不再是活动窗口时调用。 |
| windowDeiconified(WindowEvent e) | 当窗口从最小化更改为正常状态时调用。 |
| windowIconified(WindowEvent e) | 当窗口从正常状态更改为最小化状态时调用。 |
| windowOpened(WindowEvent e) | 第一次调用窗口可见。 |

* 1. MouseAdapte类

鼠标事件适配器MouseAdapte实现了MouseListener接口，可以使用其子类作为MouseEvent的监听器，只要重写其相应的方法即可

其它的监听器，也有对应的适配器

使用适配器可以避免监听器类定义没有必要的空方法

package com.learn.awt.frame;

import java.awt.Color;

import java.awt.Frame;

import java.awt.Graphics;

import java.awt.event.MouseAdapter;

import java.awt.event.MouseEvent;

import java.awt.event.WindowAdapter;

import java.awt.event.WindowEvent;

import java.util.ArrayList;

import java.util.Random;

public class MyMouseAdapterDemo {

public static void main(String[] args) {

new MyFrame3("s");

}

}

class MyFrame3 extends Frame{

private static final long serialVersionUID = 1L;

ArrayList<Point> points = new ArrayList<>();

MyFrame3(String str){

super(str);

setLayout(null);

setBounds(300,300,400,300);

this.setBackground(new Color(200, 204, 255));

setVisible(true);

addMouseListener(new Monitor1());

addWindowListener(new WindowAdapter() {

@Override

public void windowClosing(WindowEvent e) {

System.exit(0);

}

});

}

public void paint(Graphics g){

Random r = new Random();

points.forEach(e->{

g.setColor(new Color(r.nextInt(255),r.nextInt(255),r.nextInt(255)));

g.fillOval(e.getX(), e.getY(),9,9);

});

}

void addPoint(Point point){

points.add(point);

}

}

class Point{

private Integer x;

private Integer y;

public Integer getX() {

return x;

}

public void setX(Integer x) {

this.x = x;

}

public Integer getY() {

return y;

}

public void setY(Integer y) {

this.y = y;

}

Point(int x,int y){

this.x = x;

this.y = y;

}

}

class Monitor1 extends MouseAdapter{

public void mousePressed(MouseEvent e){

MyFrame3 f = (MyFrame3) e.getSource();

f.addPoint(new Point(e.getX(), e.getY()));

f.repaint();

}

}

* 1. KeyListener

package com.learn.awt.frame;

import java.awt.BorderLayout;

import java.awt.Frame;

import java.awt.Label;

import java.awt.event.KeyEvent;

import java.awt.event.KeyListener;

import java.awt.event.WindowEvent;

import java.awt.event.WindowListener;

public class KeyListenerDemo extends Frame {

private static final long serialVersionUID = -192692922041562121L;

public static void main(String[] args) {

new KeyListenerDemo("keyCode");

}

KeyListenerDemo(String title){

setTitle(title);

Label l = new Label("code");

add(l, BorderLayout.NORTH);

setBounds(100, 100, 50, 100);

addWindowListener(new WindowListener() {

@Override

public void windowOpened(WindowEvent e) {}

@Override

public void windowIconified(WindowEvent e) {}

@Override

public void windowDeiconified(WindowEvent e) {}

@Override

public void windowDeactivated(WindowEvent e) {}

@Override

public void windowClosing(WindowEvent e) {

System.exit(0);

}

@Override

public void windowClosed(WindowEvent e) {}

@Override

public void windowActivated(WindowEvent e) {}

});

addKeyListener(new KeyListener() {

@Override

public void keyTyped(KeyEvent e) {

}

@Override

public void keyReleased(KeyEvent e) {

}

@Override

public void keyPressed(KeyEvent e) {

l.setText(e.getKeyCode()+"");

}

});

setVisible(true);

}

}

1. Java-Thread

概念之并行并发

并行：多个事件在同一时刻发生

并发：多个事件在同一时段发生

概念之进程和线程

进程：指一个内存中运行中的应用程序，每一个进程都有自己独立的内存空间，也就是说进程之间的通信是不方便的，因此我们引入了线程技术，一个应用程序可以同时启动多个进程。

线程：线程是比进程更轻量级的调试执行单位，指进程中的一个执行任务，一个进程可以同时并发支行多个线程，一个进程至少有一个线程。

线程调度：

计算机只有一个CPU时，在任意时刻都只能执行一条指令，每一个进程只有获得CPU的使用权时才可以执行指令，那么，在可运行池中，多个线程会处于就绪状态等待处理，JVM就负责了线程的调度。JVM又采用了抢占式调度，没有采用分时调度，所以多线程执行结果具有随机性。

JAVA程序的进程中至少包含两个主线程和垃圾回收线程。

* 1. Runtime与ProcessBuider

开始一个进程：两种方法

package com.learn.thread;

import java.io.IOException;

public class RuntimeDemo {

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

Runtime run = Runtime.getRuntime();

run.exec("notepad");

ProcessBuilder process = new ProcessBuilder("notepad");

process.start();

}

}

* 1. Thread与Runnable

开始线程：

Thread类

* + 1. 定义一个类实现java.lang.Thread类
    2. 在类中覆盖重写Thread类中的run方法
    3. 我们在main方法中创建线程对象，并启动线程 .start();

Runnable接口

* 1. 定义一个类实现java.lang.Runnable接口
  2. 在类中重写Runnable接口中的run方法
  3. 在main方法中创建线程对象，并启动线程

package com.learn.thread;

import java.util.Date;

public class ThreadDemo extends Thread{

public static void main(String[] args) {

new ThreadDemo().start();

new Thread(()->{

for(int i=0;i<500;i++){

System.out.println("抢占线程1。。。");

}

}).start();

(new Thread(){

@Override

public void run() {

for(int i=0;i<500;i++){

System.out.println("抢占线程2。。。");

}

}

}).start();

}

@Override

public void run() {

for(int i=0;i<2000;i++){

System.out.println(new Date());

}

}

}

两种实现方式的区别

继承方式：

Java中类是单继承的，如果继承Thread，就不可以继承其它类

从操作上分析，继承方式更简单，获得线程名字也简单

从多线程共享同一资源上分析，继承方式不可以做到

实现方式：

Java可以实现多个接口，可以继承或实现其它类或接口

操作上稍微复杂，得使用Thread.currentThread()来获取当前线程引用

从多线程共享同一个资源，实现方式可以做到

package com.learn.thread;

/\*\*

\* 实现方式可以实现数据共享

\* @author zeimao77

\*

\*/

public class EatApple2 implements Runnable {

private int count=50;

@Override

public void run() {

for(int i=0;i<50;i++){

if(count>0){

System.out.println(Thread.currentThread().getName()+"--:"+count--);

}

}

}

public static void main(String[] args) {

EatApple2 e = new EatApple2();

new Thread(e, "a").start();

new Thread(e, "b").start();

new Thread(e, "c").start();

}

}

package com.learn.thread;

/\*\*

\* 实现方式可以实现数据共享

\* @author zeimao77

\*

\*/

public class EatApple2 implements Runnable {

private int count=50;

@Override

public void run() {

for(int i=0;i<50;i++){

if(count>0){

System.out.println(Thread.currentThread().getName()+"--:"+count--);

}

}

}

public static void main(String[] args) {

EatApple2 e = new EatApple2();

new Thread(e, "a").start();

new Thread(e, "b").start();

new Thread(e, "c").start();

}

}

* 1. 线程安全

要解决上述多线程并发访问同一个资源的线程安全问题，我们必需保证保证线程中的方法必需同步完成，其它线程只有等到上一个线程把方法执行完毕后才可以进入方法体执行代码。因此提出了三种解决方法：

1. 同步代码块

public void run() {

for(int i=0;i<50;i++){

synchronized (this) {

if(count>0){

System.out.println(Thread.currentThread().getName()+"--:"+count--);

}

}

}

}

1. 同步方法

public synchronized void eat(){

if(count>0){

System.out.println(Thread.currentThread().getName()+"--:"+count--);

}

}

@Override

public void run() {

for(int i=0;i<50;i++){

eat();

}

}

1. 互斥锁

private ReentrantLock lock = new ReentrantLock();

@Override

public void run() {

for(int i=0;i<50;i++){

lock.lock();

if(count>0){

System.out.println(Thread.currentThread().getName()+"--:"+count--);

}

lock.unlock();

}

}

对于同步代码块，它的同步对象又是什么呢？

对于非静态方法，同步锁就是this

对于静态方法，我们使用当前方法所在的类的字节码对象

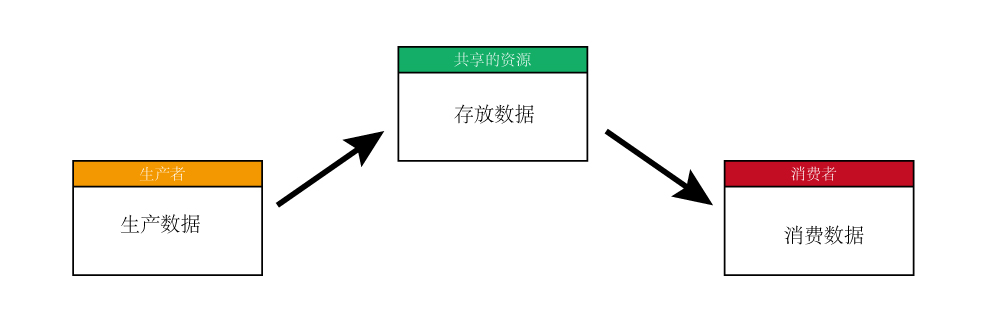
* 1. 线程的通信

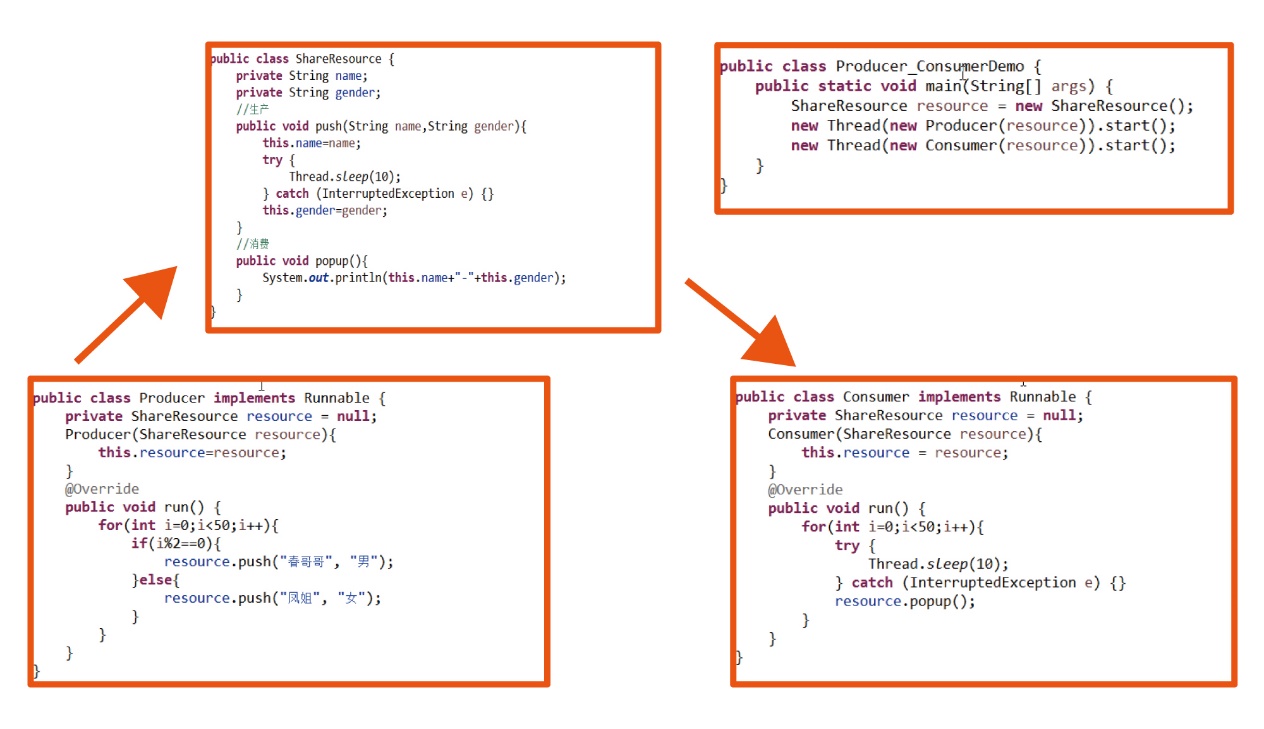
不同的线程执行不同的任务，如果这些任务有某种关系，线程之间必须能够通信，协调完成工作，经典的生产者和消费者模型。

模型分析： 应该存在一个或多个线程来表示生成者

应该存在一个或多个线程来表示消费者

生产者与消费者应该共享数据



在这里，我们为什么不把生产者生产的数据直接给消费者，而是先把数据存放到共享的资源对象中去，再由消费者取出数据消费呢？在这里主要是体现了面向对象的设计理念：低耦合。

分析生产者与消费模型存在的问题

* 建议在生产姓名和性别之间以及在打印之前使用Thread.sleep(10),使得效果更加明显。会出现性别紊乱问题：



出现原因分析：生产者先生产出春哥哥-男，此时消费者没有消费，生产者继续生产出姓名为凤姐时，此时开始消费，所以会产生性别紊乱问题。要解决这个问题，我们可以使用同步代码块或锁的机制来保持同步性

* 生产与消费本应该交替执行



出现的原因分析：当生产者在生产数据的时候，会首先拥有同步锁，其它线程在线程池中等待，只有等待同步锁被释放的时候，其它线程才开始抢占线程锁的使用权。而此时不一定可以抢到权限，所以我们可以使用唤醒机制来解决此问题。

线程通信之wait和notify方法

Java.lang.Object类提供了两类用于线程通信的方法

Wait()：执行该方法的线程对象释放同步锁，JVM把该线程存放到等待池中，等待其它线程唤醒该线程。

Notify()：执行该方法的线程唤醒在等待池中的任意一个线程，把此线程在存放到线程池中。

NotifyAll()：执行该方法的线程唤醒在等待池中等待的所有线程，把线程转到锁池中等待。

注意：以上方法只可以被同步监听锁对象来调用，否则会报错。

多个线程只有使用相同的对象的时候，多线程才会有互斥效果。我们把这个用来做互斥的对象称这为同步监听对象/同步锁

同步锁对象可以选择任意的JAVA对象，只需要保证多个线程之间使用的是相同的锁对象即可。

假设A线程和B线程共同操作一个对象x，A,B线程可以通过x对象的wait()和notify()方法来进行通信，流程如下：

1. 当A线程执行x对象的同步方法时，A线程持有x对象的锁，B线程在x对象的锁池中等待。
2. A线程在同步方法中执行x.wait()方法，A线程释放x对象的锁，进入x对象的等待池中
3. 在x对象的锁池中等待锁的B线程获取x对象的锁，执行x的另一个同步方法
4. B线程在同步方法中执行x.notify()方法时,JVM把线程A从x对象的等待池中移动到x对象的锁池中，等待获取锁，
5. B线程执行完同步方法，释放锁A线程获得锁，继续执行同步方法。

修改之后：

package com.learn.thread;

public class ShareResource {

private String name;

private String gender;

private boolean isEmpty = true;

//生产

public synchronized void push(String name,String gender){

if(!isEmpty){

try {

this.wait();

} catch (InterruptedException e) {}

}

this.name=name;

try {

Thread.sleep(10);

} catch (InterruptedException e) {}

this.gender=gender;

isEmpty = false;

this.notify();

}

//消费

public synchronized void popup(){

if(isEmpty){

try {

this.wait();

} catch (InterruptedException e) {}

}

System.out.println(this.name+"-"+this.gender);

isEmpty= true;

this.notify();

}

}

* 1. 线程的阻塞

在上述程序中，如果存在多个生产者或消费者会产生线程阻塞问题，我们可以使用notifyAll()方法解决。

* + 1. 哲学家就餐

哲学家就餐问题是在[计算机科学](http://baike.baidu.com/item/%E8%AE%A1%E7%AE%97%E6%9C%BA%E7%A7%91%E5%AD%A6/9132)中的一个经典问题，用来演示在[并行计算](http://baike.baidu.com/item/%E5%B9%B6%E8%A1%8C%E8%AE%A1%E7%AE%97)中多线程同步(Synchronization)时产生的问题。在1971年，著名的计算机科学家艾兹格·迪科斯彻提出了一个同步问题，即假设有五台计算机都试图访问五份共享的磁带驱动器。稍后，这个问题被托尼·霍尔重新表述为哲学家就餐问题。这个问题可以用来解释死锁和资源耗尽。

问题描述编辑

哲学家就餐问题可以这样表述，假设有五位哲学家围坐在一张圆形餐桌旁，做以下两件事情之一：吃饭，或者思考。吃东西的时候，他们就停止思考，思考的时候也停止吃东西。餐桌中间有一大碗意大利面，每两个哲学家之间有一只餐叉。因为用一只餐叉很难吃到意大利面，所以假设哲学家必须用两只餐叉吃东西。他们只能使用自己左右手边的那两只餐叉。哲学家就餐问题有时也用米饭和筷子而不是意大利面和餐叉来描述，因为很明显，吃米饭必须用两根筷子。

哲学家从来不交谈，这就很危险，可能产生死锁，每个哲学家都拿着左手的餐叉，永远都在等右边的餐叉（或者相反）。即使没有死锁，也有可能发生资源耗尽。例如，假设规定当哲学家等待另一只餐叉超过五分钟后就放下自己手里的那一只餐叉，并且再等五分钟后进行下一次尝试。这个策略消除了死锁（系统总会进入到下一个状态），但仍然有可能发生“活锁”。如果五位哲学家在完全相同的时刻进入餐厅，并同时拿起左边的餐叉，那么这些哲学家就会等待五分钟，同时放下手中的餐叉，再等五分钟，又同时拿起这些餐叉。

在实际的计算机问题中，缺乏餐叉可以类比为缺乏共享资源。一种常用的计算机技术是资源加锁，用来保证在某个时刻，资源只能被一个程序或一段代码访问。当一个程序想要使用的资源已经被另一个程序锁定，它就等待资源解锁。当多个程序涉及到加锁的资源时，在某些情况下就有可能发生死锁。例如，某个程序需要访问两个文件，当两个这样的程序各锁了一个文件，那它们都在等待对方解锁另一个文件，而这永远不会发生。

问题解法编辑

服务生解法

一个简单的解法是引入一个餐厅服务生，哲学家必须经过他的允许才能拿起餐叉。因为服务生知道哪只餐叉正在使用，所以他能够作出判断避免死锁。

为了演示这种解法，假设哲学家依次标号为A至E。如果A和C在吃东西，则有四只餐叉在使用中。B坐在A和C之间，所以两只餐叉都无法使用，而D和E之间有一只空余的餐叉。假设这时D想要吃东西。如果他拿起了第五只餐叉，就有可能发生死锁。相反，如果他征求服务生同意，服务生会让他等待。这样，我们就能保证下次当两把餐叉空余出来时，一定有一位哲学家可以成功的得到一对餐叉，从而避免了死锁。

资源分级解法

另一个简单的解法是为资源（这里是餐叉）分配一个偏序或者分级的关系，并约定所有资源都按照这种顺序获取，按相反顺序释放，而且保证不会有两个无关资源同时被同一项工作所需要。在哲学家就餐问题中，资源（餐叉）按照某种规则编号为1至5，每一个工作单元（哲学家）总是先拿起左右两边编号较低的餐叉，再拿编号较高的。用完餐叉后，他总是先放下编号较高的餐叉，再放下编号较低的。在这种情况下，当四位哲学家同时拿起他们手边编号较低的餐叉时，只有编号最高的餐叉留在桌上，从而第五位哲学家就不能使用任何一只餐叉了。而且，只有一位哲学家能使用最高编号的餐叉，所以他能使用两只餐叉用餐。当他吃完后，他会先放下编号最高的餐叉，再放下编号较低的餐叉，从而让另一位哲学家拿起后边的这只开始吃东西。

尽管资源分级能避免死锁，但这种策略并不总是实用的，特别是当所需资源的列表并不是事先知道的时候。例如，假设一个工作单元拿着资源3和5，并决定需要资源2，则必须先要释放5，之后释放3，才能得到2，之后必须重新按顺序获取3和5。对需要访问大量数据库记录的计算机程序来说，如果需要先释放高编号的记录才能访问新的记录，那么运行效率就不会高，因此这种方法在这里并不实用。

这种方法经常是实际计算机科学问题中最实用的解法，通过为分级锁指定常量，强制获得锁的顺序，就可以解决这个问题。

Chandy/Misra解法

1984年，K. Mani Chandy和J. Misra提出了哲学家就餐问题的另一个解法，允许任意的用户（编号P1, ..., Pn）争用任意数量的资源。与迪科斯彻的解法不同的是，这里编号可以是任意的。

1.对每一对竞争一个资源的哲学家，新拿一个餐叉，给编号较低的哲学家。每只餐叉都是“干净的”或者“脏的”。最初，所有的餐叉都是脏的。

2.当一位哲学家要使用资源（也就是要吃东西）时，他必须从与他竞争的邻居那里得到。对每只他当前没有的餐叉，他都发送一个请求。

3.当拥有餐叉的哲学家收到请求时，如果餐叉是干净的，那么他继续留着，否则就擦干净并交出餐叉。

4.当某个哲学家吃东西后，他的餐叉就变脏了。如果另一个哲学家之前请求过其中的餐叉，那他就擦干净并交出餐叉。

这个解法允许很大的并行性，适用于任意大的问题。

package com.learn.thread;

import java.util.concurrent.locks.Condition;

import java.util.concurrent.locks.ReentrantLock;

public class ShareResource2 {

private String name;

private String gender;

private boolean isEmpty;

private ReentrantLock lock = new ReentrantLock();

private Condition condition = lock.newCondition();

public void push(String name,String gender){

lock.lock();

try {

if(!isEmpty){

condition.await();

}

this.name=name;

Thread.sleep(10);

this.gender=gender;

isEmpty = false;

condition.signalAll();

} catch (InterruptedException e) {

e.printStackTrace();

}finally {

lock.unlock();

}

}

public void popup(){

lock.lock();

try {

if(isEmpty){

condition.await();

}

System.out.println(this.name+"-"+this.gender);

isEmpty= true;

condition.signalAll();

} catch (InterruptedException e) {

}finally{

lock.unlock();

}

}

}

多线程通信的时候很容易造成死锁，死锁无法解决，只能避免。

当A线程等待由B线程持有的锁，而B线程正在等待持有的锁时，会发生死锁现象，JVM不检测也不试图处理这种情况，所以程序员必须保证不导致死锁。避免死锁的法则：当多个线程都需要访问共享的资源A,B,C时，保证每一个线程都按照相同的顺序去访问他们，比如都先访问A,接着B，接着C。

* 1. Thread status

线程状态。 线程可以处于以下状态之一：

* + 1. 新建状态（new）：

尚未启动的线程处于此状态。 使用new创建一个线程对象，仅仅在堆中分配内存空间，在调用start方法之前，新建状态下，线程压根就没有启动，仅仅只是存在一个线程对象而已。

Thread thread = new Thread();此时的thread就属于新建状态。

* + 1. 可运行状态（RUNNABLE）：

分成两种状态，ready和running。分别表示就绪状态和支行状态。

就绪状态：线程对象调用start方法之后，等等JVM的调度。

运行状态：线程对象获得JVM调度，如果存在多个CPU，那么允许多个线程并行运行。

在Java虚拟机中执行的线程处于此状态。

* + 1. 阻塞状态（BLOCKED）

正在运行的线程因为某些原因放弃CPU，暂时停止运行就会进入阻塞状态，此时JVM不会给线程分配CPU，走到线程重新进入就绪状态，才有机会转到运行状态。

阻塞状态的两种情况

1.当A线程处于支行过程时，试图获取同步锁时，却被B线程获取，此时JVM当前A 线程存到对象的锁池中，A线程进入阻塞状态。

2.当现场处于支行过程时，发出了IO请求时，此时进入阻塞状态。

被阻塞等待监视器锁定的线程处于此状态。

* + 1. 等待（WAITING）

正在等待另一个线程执行特定动作的线程处于此状态。

1.当线程处于支行过程时，调用了wait()方法，此时JVM把当前线程存在的对象等待池中。

2.当线程执行了sleep()方法

* + 1. 计时等待（TIMED\_WAITING）

正在等待另一个线程执行动作达到指定等待时间的线程处于此状态。

1.当线程处于运行状态使用了wati(long time)方法，此时JVM把当前线程存在对象等待池中。

2.当前线程执行了sleep(long time)方法

* + 1. 死亡状态（TERMINATED ）

已退出的线程处于此状态。

1.正常执行完run方法而退出，

2.遇到异常而退出，

isAlive() ：测试这个线程是否活着，

一个线程可以在给定时间点处于一个状态。 这些状态是不反映任何操作系统线程状态的虚拟机状态。线程一旦死亡，不可以再启动。

package com.learn;

import java.io.BufferedReader;

import java.io.IOException;

import java.io.InputStreamReader;

public class ThreadTest {

//这种等待是非常耗性能的

public static void createBusyThread() {

Runnable runnable = ()->{

while (true){;}

};

new Thread(runnable,"testBusyThread").start();

}

//锁等待 这个线程在被唤醒前不会分配执行

public static void createLockThread(final Object lock){

Runnable runnable = ()->{

synchronized (lock){

try {

lock.wait();

} catch (InterruptedException e) {

e.printStackTrace();

}

}

};

new Thread(runnable,"testLockThread").start();

}

//readBytes没有检查到流更新时会马上归还执行令牌，所以这种等待只会消耗很小的CPU资源

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

BufferedReader br = new BufferedReader(new InputStreamReader(System.in));

br.readLine();

createBusyThread();

br.readLine();

Object obj = new Object();

createLockThread(obj);

}

}

* 1. 线程睡眠

让执行的线程暂停一段时间，进入计时等待状态。

方法：static void sleep(long millis)；使当前正在执行的线程以指定的毫秒数暂停（暂时停止执行），具体取决于系统定时器和调度程序的精度和准确性。调用 sleep后，当前线程放弃CPU，在指定时间段内，sleep所在线程不会获得执行的机会。此状态下的线程不会释放同步锁/同步监听器。

* 1. 联合线程

package com.learn.thread;

class JoinThread extends Thread {

@Override

public void run() {

for(int i=0;i<50;i++){

System.out.println("JoinThread:"+i);

}

}

}

public class JoinDemo{

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

JoinThread j=new JoinThread();

for(int i=0;i<50;i++){

System.out.println("main:"+i);

if(i==20){

j.start();

}

if(i==40){

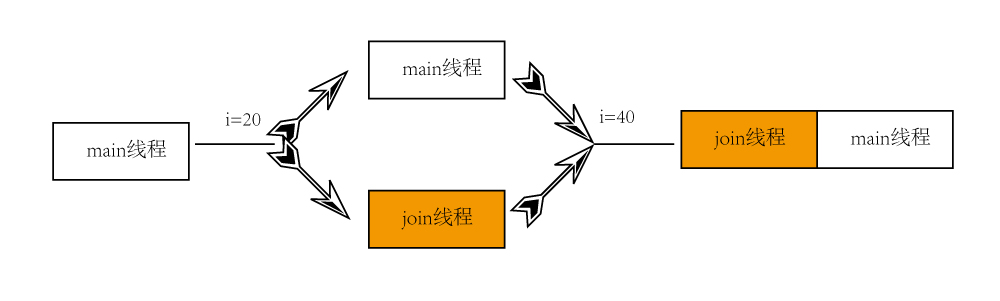
j.join();

}

}

}

}



* 1. 守护线程

在后台运行的线程，其目的是为其它线程提供服务，也称之为“守护线程”。JVM的垃圾回收线程就是典型的后台线程.

特点:所有的前台线程死亡,后台线程自动死亡,前台线程没有结束,后台线程是不会结束的.测试线程对像是否为后台线程:使用Tjread.isDaemon()。前台线程创建的线程默认是前台线程。可以通过setDaenon方法设置为后台线程，并且当且仅当后台线程创建的新线程时，新线程是后台线程。设置后台线程方法属性（setDaemon(true)）在start方法调用之前，否则出现异常。

package com.learn.thread;

public class DaemonThread {

public static void main(String[] args) {

Thread t = new Thread(()->{

for(int i=0;i<50;i++){

System.out.println("main:"+i);

}

});

t.setDaemon(true);

System.out.println(t.isDaemon());

try {

t.start();

Thread.sleep(100);

} catch (InterruptedException e) {

e.printStackTrace();

}

}

}

* 1. 线程优先级

每个线程都有优先级，优先级的高低只和线程获得执行机会的次数多少有关，并非线程优先级越高就一定优先执行，哪个优先取决于CPU的高度。

最高优先级：public final static int MAX\_PRIORITY = 10;

最低优先级：public final static int MIN\_PRIORITY = 1;

默认优先级：public final static int NORM\_PRIORITY = 5;

每个线程都有默认优先级，主线程默认优先级为5，如果A线程创建了B线程，那么B线程和A线程具有相同的优先级，不同的操作系统支持的线程优先级别不两只，建议使用上述三个优先级，尽量不要使用自定义。

package com.learn.thread;

public class PriorityDemo extends Thread {

public static void main(String[] args) {

Thread t1 = new Thread(new PriorityDemo(),"t1");

Thread t2 = new Thread(new PriorityDemo(),"t2");

t2.setPriority(MAX\_PRIORITY);

t1.start();

t2.start();

}

@Override

public void run() {

for(int i=0;i<50;i++){

System.out.println(Thread.currentThread().getName()+"--"+i);

}

}

}

* 1. 线程礼让

Yield方法：表示当前线程对象提示调度器自己愿意让出CPU资源，但是高度器可以自由的忽略该提示。

调用该方法后，线程对象进入就绪状态，所以完全有可能，某个线程调用了yield()之后，线程调度器又把它调度出来。从java7提供的文档上可以清楚看出，开发中很少会使用到该方法，该方法主要用于高度或测试，它可能有助于因多个线程竞争条件下的错误重现现象。用法和sleep方法用法一样。

Sleep方法和yield方法的区别：

1.都能使当前处于支行状态的线程放弃CPU，把运行的机会给其它线程。

2.sleep方法会给其他线程运行机会，但是不考虑其他线程的优先级，yield方法只会给相同优先级或者更高优先级的线程运行机会。

3.调用sleep方法后，线程进入计时等待状态，调用yeild方法后，线程进入就绪状态。

* 1. 死锁

package com.learn;

public class DLockTest implements Runnable {

int a,b;

DLockTest(int a,int b){

this.a = a;

this.b = b;

}

@Override

public void run() {

Integer ai = Integer.valueOf(a);

Integer bi = Integer.valueOf(b);

synchronized (ai){

synchronized (bi){

System.out.println(a+b);

}

}

}

public static void main(String[] args){

for(int i=0;i<100;i++){

new Thread(new DLockTest(2,3)).start();

new Thread(new DLockTest(3,2)).start();

}

}

}

这段代码开启了两百个线程去计算2+3及3+2的值，但我们很容易发现Integer.valueOf()返回的对象是同一个对象，这是因为JAVA虚拟机为了减少对象创建次数和减少内存压力，所以会将[-128-127]范围的数进行缓存，在调用valueOf方法时会返回缓存中的对象。所以会存在两个线程分别持有对方锁的情况，导致大家都进行不下去的情况。

1. Java反射
   1. Class类

package com.learn.clazz;

public class ClassDemo {

public static void main(String[] args) {

/\*Demo的实例对象\*/

Demo demo = new Demo();

/\*Demo这个类也是一个实例对象，Class类的实例 对象如何表示呢？

任何一个类都是Class的实例对象，这个实例对象有三种表示方式。

第一种表示方式->实际在告诉我们任何一个类都有一个隐含的静态成员\*/

Class c1 = Demo.class;

/\*第二种表示方式 已经知道该类的对象通过getclass方法\*/

Class c2 = demo.getClass();

/\*第三种表达方式\*/

try {

Class c3 = Class.forName("com.learn.clazz.Demo");

//我们完全可以通过类的类型创建该类的实例对象

Demo demo1 = (Demo) c1.newInstance();

} catch (InstantiationException e) {

// TODO Auto-generated catch block

e.printStackTrace();

} catch (IllegalAccessException e) {

// TODO Auto-generated catch block

e.printStackTrace();

} catch (ClassNotFoundException e) {

// TODO Auto-generated catch block

e.printStackTrace();

}

}

}

* 1. 加载

new创建对象，是静态加载，在编译时刻就需要加载所有可能使用到的类。因此，如果在编译时出现问题，那么整个程序都将不可运行。而可以通过动态加载类机制可以解决此问题。

package com.learn.clazz;

public class BeanFactory {

public static OfficeAble getBean(String beanName){

try {

return (OfficeAble) Class.forName(beanName).newInstance();

} catch (Exception e) {

e.printStackTrace();

}

return null;

}

}

* 1. Method类

每一个成员方法都是一个Method对象

getMethods()方法获取的是所有的public的函数，包括父类继承而来的。

getDeclaredMethods()获取的是所有该类自己声明的方法，不问访问权限。

package com.learn.clazz;

import java.lang.reflect.Method;

public class OfficeDemo {

public static void main(String[] args) {

OfficeAble off = BeanFactory.getBean("com.learn.clazz.Excel");

Method[] mes = BeanFactory.getMethod("com.learn.clazz.Excel");

for(Method me:mes){

System.out.println(me+",");

}

off.say();

}

}

* 1. Field类

成员变量也是对象（java.lang.reflect.Field）

Field类封装了关于成员变量的操作

GetFields()方法获取的是所有的public的成员变量信息

GetDeclaredFields()方法获取的是该类自己声明的成员变量信息。

package com.learn.clazz;

import java.lang.reflect.Field;

import java.lang.reflect.InvocationTargetException;

import java.lang.reflect.Method;

public class OfficeDemo {

public static void main(String[] args) throws NoSuchFieldException, SecurityException, IllegalArgumentException, IllegalAccessException, InvocationTargetException {

OfficeAble off = BeanFactory.getBean("com.learn.clazz.Excel");

Method[] mes = BeanFactory.getMethods("com.learn.clazz.Excel");

for(Method me:mes){

System.out.println(me.getName()+",");

}

Field[] fis = BeanFactory.getFields("com.learn.clazz.Excel");

for(Field fi:fis){

System.out.println(fi.getName()+","+Excel.class.getField(fi.getName()).get(off));//ABC,abc

}

mes[0].invoke(off, null);

}

}

Java中编译之后的集合是去泛型化的，是只为了防止输入错误，只在编译阶段有效，绕过编译就无效了，我们可以通过反射的方法来操作绕过编译。

package com.learn.clazz;

import java.lang.reflect.InvocationTargetException;

import java.lang.reflect.Method;

import java.util.ArrayList;

public class GenericParadigm {

public static void main(String[] args) {

try {

ArrayList<String> list = new ArrayList();

Class c = list.getClass();

Method m = c.getMethod("add", Object.class);

m.invoke(list, "abc");

//这里123可以添加进list集合中，但不可以遍历;

m.invoke(list, 123);

System.out.println(list.size());//2

list.forEach(System.out::println);//ClassCastException异常

} catch (Exception e) {

e.printStackTrace();

}

}

}

1. Java通信
   1. TCP/IP协议

TCP/IP协议是目前世界上应用最为广泛的协议，是以TCP和IP为基础的不同层次上多个协议的集合。也称TCP/IP协议族或TCP/IP协议栈。

TCP：Transmission Control Protocol 传输控制协议

IP：Internet Protocol 互联网协议

|  |  |
| --- | --- |
| 5.应用层 | HTTP协议、FTP文件传输协议、SMTP简单邮件传送协议、Telnet远程登录服务。 |
| 4.传输层 | TCP/IP协议 |
| 3.网络层 |  |
| 2.数据链路层 |  |
| 1.物理 | 硬件部分 |

IP

端口：

1. 用于区分不同应用程序
2. 端口号范围是0-65535，其是0-1023为系统所保留
3. HTTP:80 ;FTP:21 ;TELNET:23

IP地址和端口号组成了所谓的Socket，Socket是网络上支行的程序之间双向通信链路的终结点，是TCP和UDP的基础。

针对网络通信的不同层次，java提供的网络功能有四大类：

InetAddress:用于标识网络上的硬件资源。

URL：统一资源定位符，通过URL可以直接读取或写入网络上的数据

Sockets：使用TCP协议实现网络通信的Socket相关的类

Datagram：使用UDP协议，将数据保存在数据报中，通过网络进行通信

* 1. Inetaddress:

package com.learn.socket;

import java.net.InetAddress;

import java.net.UnknownHostException;

public class LocalInetAddress {

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

InetAddress localAddress = InetAddress.getLocalHost();

System.out.println(localAddress.getHostName());

System.out.println(localAddress.getHostAddress());

}

}

* 1. URL

1. URL(Uniform Resource Locator)统一资源定位符，表示Internet上某一资源的地址。
2. URL由两部分组成：协议名称和资源名称，中间用冒号隔开
3. 在java.net包中，提供了URL类来表示URL

package com.learn.socket;

import java.io.BufferedReader;

import java.io.File;

import java.io.FileWriter;

import java.io.InputStream;

import java.io.InputStreamReader;

import java.net.URL;

public class UrlDemo {

public static void main(String[] args) {

try {

URL baidu = new URL("https://www.baidu.com/");

File file = new File("C:\\Users\\Mao\\Desktop\\baidu.html");

if(file.exists()){

file.delete();

}

file.createNewFile();

FileWriter fw = new FileWriter(file);

InputStream is = baidu.openStream();

BufferedReader reader = new BufferedReader(new InputStreamReader(is, "utf-8"));

String data;

while((data=reader.readLine())!=null){

System.out.println(data);

fw.write(data);

}

fw.close();

is.close();

reader.close();

} catch (Exception e) {

e.printStackTrace();

}

}

}

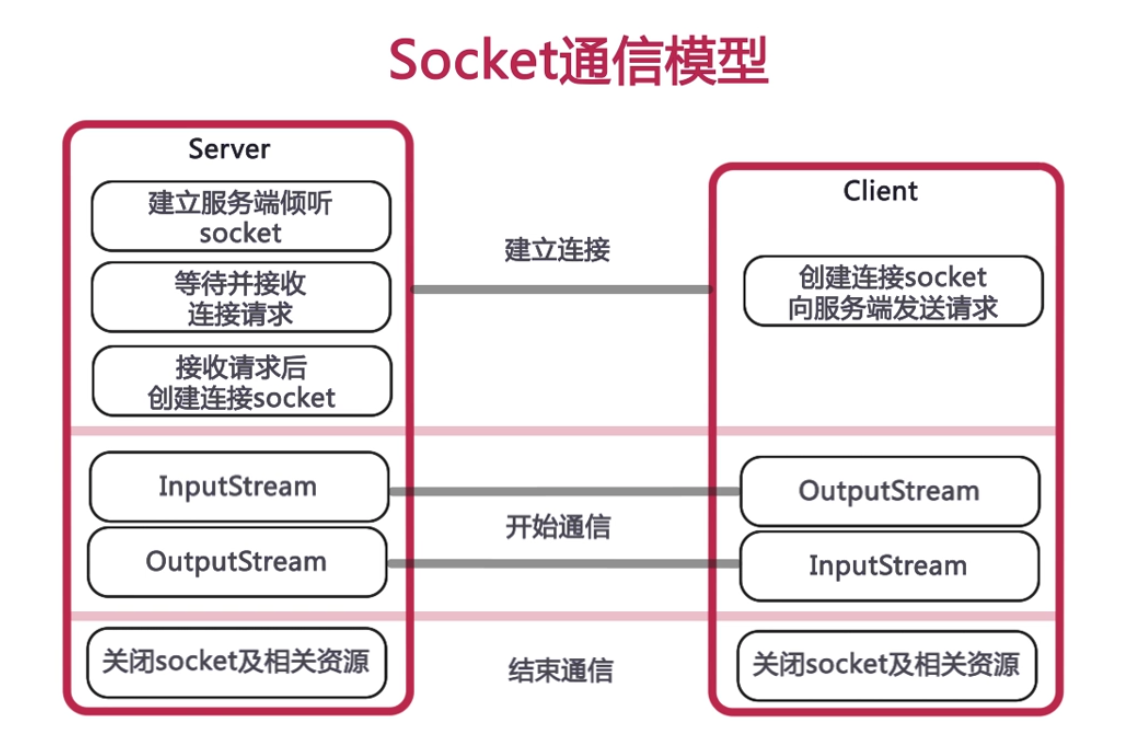
* 1. Socket

TCP协议是面向连接的、可靠的、有序的、以字节流的方式发送数据

基于TCP协议实现网络通信的类

客户端的Socket类

服务器端的ServerSocket类



Socket通信实现步骤

1. 创建ServerSocket和Socket
2. 打开连接到Socket的输入/输出流
3. 按照协议对Socket进行读/写操作
4. 关闭输入输出流，关闭Socket

服务器端

1. 创建ServerSocket对象，绑定监听端口
2. 通过accept()方法监听客户端请求
3. 连接建立后，通过输入流读取客户端面发送的请求信息
4. 通过输出流向客户端发送响应信息
5. 关闭相应资源

客户端

1. 创建socket对象，指明需要连接的服务器地址和端口号
2. 连接建立后，通过输入输出流向服务器端发送请求信息
3. 通过输入流获取服务器端响应的信息
4. 关闭相关的资源

package com.learn.socket;

import java.io.BufferedReader;

import java.io.IOException;

import java.io.InputStreamReader;

import java.io.PrintWriter;

import java.net.ServerSocket;

import java.net.Socket;

public class ServerMain {

public static void main(String[] args) {

start();

}

public static void start(){

ServerSocket serverSocket = null;

try {

serverSocket = new ServerSocket(8000);

System.out.println("服务器启动成功。。。。。。");

while(true){

Socket socket = serverSocket.accept();

new Thread(new ServerThread(socket)).start();

}

} catch (IOException e) {

e.printStackTrace();

}finally {

try {

serverSocket.close();

} catch (IOException e) {

e.printStackTrace();

}

}

}

}

class ServerThread implements Runnable{

Socket socket = null;

ServerThread(Socket socket){

this.socket = socket;

}

@Override

public void run() {

BufferedReader br = null;

PrintWriter pw = null;

try {

System.out.println(socket.getInetAddress().getHostAddress());

try {

br = new BufferedReader(new InputStreamReader(socket.getInputStream()));

} catch (IOException e) {

e.printStackTrace();

}

String info;

while((info=br.readLine())!=null){

System.out.println(info);

}

socket.shutdownInput();

pw = new PrintWriter(socket.getOutputStream());

pw.write("欢迎您，");

pw.flush();

socket.shutdownOutput();

} catch (IOException e) {

e.printStackTrace();

}finally{

try {

br.close();

pw.close();

} catch (IOException e) {

e.printStackTrace();

}

}

}

}

package com.learn.socket;

import java.io.BufferedReader;

import java.io.IOException;

import java.io.InputStreamReader;

import java.io.PrintWriter;

import java.net.Socket;

import java.net.UnknownHostException;

public class ClientMain {

public static void main(String[] args) {

start();

}

private static void start() {

Socket socket = null;

PrintWriter pw = null;

BufferedReader br = null;

try {

socket = new Socket("localhost", 8000);

socket.getOutputStream();

pw = new PrintWriter(socket.getOutputStream());

String info = "abc-125";

pw.write(info);

pw.flush();

socket.shutdownOutput();

br = new BufferedReader(new InputStreamReader(socket.getInputStream()));

String respinfo = null;

while((respinfo=br.readLine())!=null){

System.out.println(respinfo);

}

} catch (UnknownHostException e) {

e.printStackTrace();

} catch (IOException e) {

e.printStackTrace();

}finally {

try {

pw.close();

socket.close();

br.close();

} catch (IOException e) {

e.printStackTrace();

}

}

}

}

* 1. UDP

UDP协议（用户数据报协议）是无连接、不可靠、无序的。进行数据传输时，首先需要先将传输的数据定义成数据报（Datagram），在数据报中指明数据所要达到的Socket（主机地址和端口号），然后再将数据报发送出去。

服务器端实现步骤

1. 创建DatagramSocket，指定端口号
2. 创建DatagramPacket
3. 接收客户端发送的数据信息
4. 读取数据

客户端实现步骤

1. 定义发送信息
2. 创建DatagramPacket，包含将要发送的信息
3. 创建DatagramSocket
4. 发送数据

package com.learn.socket;

import java.io.IOException;

import java.net.DatagramPacket;

import java.net.DatagramSocket;

import java.net.InetAddress;

import java.net.SocketException;

import java.net.UnknownHostException;

public class UDPClient {

public static void main(String[] args) {

try {

InetAddress address = InetAddress.getLocalHost();

byte[] data = "用户名：admin;密码:123".getBytes();

DatagramPacket packet = new DatagramPacket(data, data.length,address,8000);

DatagramSocket socket = new DatagramSocket();

socket.send(packet);

byte[] data2 = new byte[1024];

DatagramPacket packet2 = new DatagramPacket(data2, data.length);

socket.receive(packet2);

String info = new String(data2,0,packet2.getLength());

System.out.println(info);

socket.close();

} catch (UnknownHostException e) {

e.printStackTrace();

} catch (SocketException e) {

e.printStackTrace();

} catch (IOException e) {

e.printStackTrace();

}

}

}

package com.learn.socket;

import java.io.IOException;

import java.net.DatagramPacket;

import java.net.DatagramSocket;

import java.net.InetAddress;

import java.net.SocketException;

public class UDPServer {

public static void main(String[] args) {

DatagramSocket socket = null;

try {

socket = new DatagramSocket(8000);

System.out.println("服务器启动成功！");

byte[] data = new byte[1024];

DatagramPacket packet = new DatagramPacket(data, data.length);

socket.receive(packet);

String info = new String(data,0,packet.getLength());

System.out.println(info);

byte[] data2 = "欢迎您,".getBytes();

DatagramPacket packet2 = new DatagramPacket(data2,data2.length,packet.getAddress(),packet.getPort());

socket.send(packet2);

socket.close();

} catch (SocketException e) {

e.printStackTrace();

} catch (IOException e) {

e.printStackTrace();

}finally{

socket.close();

}

}

}

1. Java注解

Java提供了一种原程序中的元素关联任何信息和任何元数据的途径和方法

* 1. JDK自带注解

@Override

@Deprecated

@Suppvisewarnings

注解分类：

按照运行机制来分：

* 源码注解

注解只在源代码存在，编译成字节码文件就不存在了。

* 编译注解

在原代码和字节码文件中都存在

* 运行注解

在运行阶段才起作用，甚至会影响支行逻辑的注解

* 1. 自定义注解

元注解：

* @Target({ElementType.METHOD,ElementType.TYPE}):

注解的作用域：

Constructor[构造方法声明]、field[字段声明]、local\_variable[局部变量声明]、method[方法声明]、package[包声明]、parameter[参数声明]、type[类、接口]

TYPE\_PARAMETER[类型注解]

* @Retention(RetentionPolicy.RUNTIME)

Source只在源码显示,在编译时将丢弃

Class 编译时会记录到字节码文件，运行时将忽略

Runtime运行时存在，可以通过反射读取。

* @Inherited

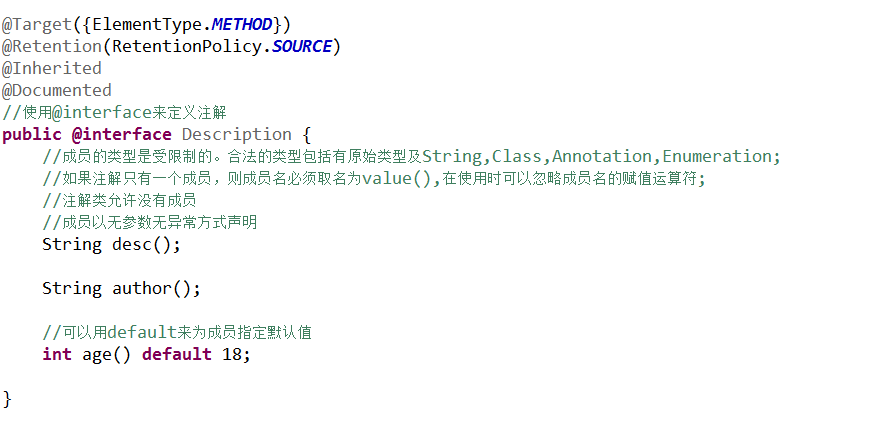
允许子类继承

* @Documented

生成javaDoc会包含注解信息

注解的使用语法：

@<注解名>(<成员名1>=<成员值1>,<成员名2>=<成员值2>……)



解析注解：

通过反射获取类、函数或成员上的运行时注解信息，从而实现动态控制程序运行的逻辑

实体类：user

package com.learn.ann;

@Table("users")

public class User {

@Id("id")

private Integer uid;

@Column(value = "name",length=10)

private String uname;

@Column("pass")

private String pass;

public User(Integer uid, String uname, String pass) {

this.uid = uid;

this.uname = uname;

this.pass = pass;

}

public Integer getUid() {

return uid;

}

public void setUid(Integer uid) {

this.uid = uid;

}

public String getUname() {

return uname;

}

public void setUname(String uname) {

this.uname = uname;

}

public String getPass() {

return pass;

}

public void setPass(String pass) {

this.pass = pass;

}

}

注解类：

package com.learn.ann;

import java.lang.annotation.ElementType;

import java.lang.annotation.Retention;

import java.lang.annotation.RetentionPolicy;

import java.lang.annotation.Target;

@Target(ElementType.FIELD)

@Retention(RetentionPolicy.RUNTIME)

public @interface Column {

String value();

int length() default 255;

}

package com.learn.ann;

import java.lang.annotation.ElementType;

import java.lang.annotation.Retention;

import java.lang.annotation.RetentionPolicy;

import java.lang.annotation.Target;

@Target(ElementType.FIELD)

@Retention(RetentionPolicy.RUNTIME)

public @interface Id {

String value();

}

package com.learn.ann;

import java.lang.annotation.ElementType;

import java.lang.annotation.Retention;

import java.lang.annotation.RetentionPolicy;

import java.lang.annotation.Target;

@Target(ElementType.TYPE)

@Retention(RetentionPolicy.RUNTIME)

public @interface Table {

String value();

}

package com.learn.ann;

import java.lang.reflect.Field;

import java.lang.reflect.Method;

import java.util.HashMap;

import java.util.Set;

public class Test {

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

User user = new User(100, "user", "asdf");

System.out.println(query(user));

System.out.println(insert(user));

System.out.println(update(user));

}

private static String query(User user) throws Exception{

StringBuilder sb = new StringBuilder();

Class clazz = user.getClass();

if(!clazz.isAnnotationPresent(Table.class)){

return "非table注解类";

}

Table t = (Table) clazz.getAnnotation(Table.class);

sb.append("select \* from `").append(t.value()).append("` ");

Field[] farray = clazz.getDeclaredFields();

for(Field field:farray){

if(!field.isAnnotationPresent(Id.class)){

continue;

}

Id c = field.getAnnotation(Id.class);

String columnName = c.value();

String fieldName = field.getName();

String getMethodName = "get"+ fieldName.substring(0,1).toUpperCase()+fieldName.substring(1);

Method getMethod = clazz.getMethod(getMethodName);

Integer fieldValue = (Integer) getMethod.invoke(user, null);

sb.append("where `").append(columnName).append("` = ").append(fieldValue+";");

}

return sb.toString();

}

private static String insert(User user) throws Exception{

StringBuilder sb = new StringBuilder();

sb.append("insert into `");

HashMap<String,Object> map = new HashMap<>();

Class clazz = user.getClass();

if(!clazz.isAnnotationPresent(Table.class)){

return "非table注解类";

}

Table t = (Table) clazz.getAnnotation(Table.class);

sb.append(t.value()+"`(");

Field[] farray = clazz.getDeclaredFields();

for(Field field:farray){

if(field.isAnnotationPresent(Column.class)){

Column c = field.getAnnotation(Column.class);

String colunmName = c.value();

int length = c.length();

String fieldName = field.getName();

String getMethodName = "get"+fieldName.substring(0,1).toUpperCase()+fieldName.substring(1);

Method getMethod = clazz.getMethod(getMethodName);

Object obj = getMethod.invoke(user);

if(length!=0){

if(obj.toString().length()>length){

return "字段太长";

}

}

map.put(colunmName,obj);

}

}

Set<String> keys = map.keySet();

for(String key:keys){

sb.append("`"+key+"`,");

}

sb.deleteCharAt(sb.length()-1);

sb.append(") values (");

for(String key:keys){

sb.append("\""+map.get(key)+"\",");

}

sb.deleteCharAt(sb.length()-1);

sb.append(");");

return sb.toString();

}

private static String update(User user) throws Exception{

StringBuilder sb = new StringBuilder();

sb.append("UPDATE ");

String wherestr = null;

Class clazz = user.getClass();

if(!clazz.isAnnotationPresent(Table.class)){

return "非table注解类";

}

Table t = (Table) clazz.getAnnotation(Table.class);

sb.append("`"+t.value()+"` SET ");

Field[] farray = clazz.getDeclaredFields();

for(Field field:farray){

String fieldName = field.getName();

String getMethodName = "get"+ fieldName.substring(0,1).toUpperCase()+fieldName.substring(1);

Method getMethod = clazz.getMethod(getMethodName);

if(field.isAnnotationPresent(Id.class)){

Id i = field.getAnnotation(Id.class);

String columnName = i.value();

Integer fieldValue = (Integer) getMethod.invoke(user, null);

wherestr = " where " +columnName+" = \""+fieldValue +"\";";

}else if(field.isAnnotationPresent(Column.class)){

Column c = field.getAnnotation(Column.class);

String columnName = c.value();

int length = c.length();

Object param = getMethod.invoke(user);

if(length!=0){

if(param.toString().length()>length){

return "字段太长";

}

}

sb.append("`"+columnName+"` = \""+param+"\",");

}

}

sb.deleteCharAt(sb.length()-1);

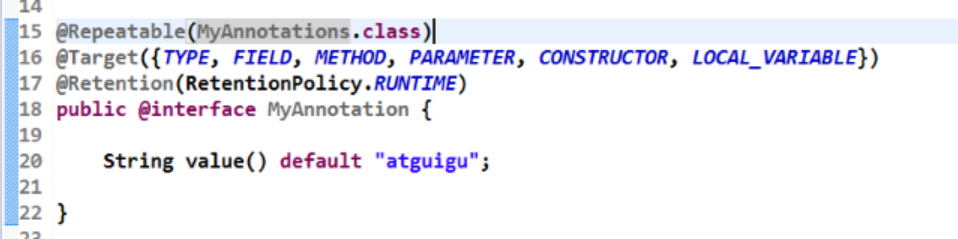
sb.append(wherestr);

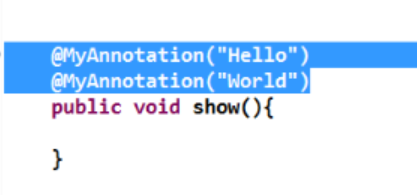
return sb.toString();

}

}

Java8还支持可重复注解，不过多个相同的注解需要有一个容器来管理，所以增加了一个元注解@Repealtable(容器注解类.class)来指定容器。





1. IO

文件流、数组流、转换流、缓冲流、打印流、扫描器、管道流、合并流、对象流、标准输入输出。

* 1. FILE类

目录和文件的抽象表现形式，File类包含了创建、删除、重命名文件、判断文件读写权限以及文件是否存在，查询等功能方法，它只可以设置和获取文件本身的信息，不可以操作文件的中的内容。

package com.learn.io;

import java.io.File;

public class FileClazz {

public static void main(String[] args) {

System.out.println("属性分隔符："+File.pathSeparator);

System.out.println("路径分隔符："+File.separator);

String path1 = "C:\\Users\\zeimao77\\Desktop\\abc.txt";

String path2 = "C:/Users/zeimao77/Desktop/abc.txt";

String path3 = "C:"+File.separator+"Users"+File.separator+"zeimao77"+File.separator+"Desktop"+File.separator+"abc.txt";

}

}

操作File路径和名称：

File getAbsoluteFile()：获取绝对路径

String getAbslutePath()：获取绝对路径

String getPath()：获取文件路径

String getName()：获取文件名称

File getParentFile()：获取上级目录文件

String getParent()：获取上级目录路径

检测File状态：

Boolean canExecute()：判断是否是可执行文件

Boolean canRead()：判断是否是可读

Boolean canWrite()：判断是否是可写

Boolean isHidden()：判断是否是隐藏文件

Long lastModified()：返回文件的最后修改时间

Long length()：获取该文件的长度大小（字节）

Boolean isFile()：是否是文件

boolean exists()：是否存在

操作File文件

Boolean createNewFile()：创建新文件

Boolean renameTo(File file)：重新命名文件

Static File createTempFile(String prefix,String suffix)：创建临时文件

Boolean delete()：删除文件

Boolean deleteOnExit()：在虚拟机停止时删除

操作File目录

Boolean isDirectory：判断是否是目录

Boolean mkdir()：创建当前目录

Boolean mkdirs()：创建当前目录及上级目录

String list()：列出所有的文件名

File[] listFiles()：列出所有文件对象

Static File[] listRoots()：列出系统盘符

package com.learn.io;

import java.io.File;

public class FileDemo1 {

public static void main(String[] args) {

File file = new File("E:/STS/java");

listAllFiles(file);

}

/\*\*

\* 列出文件夹及子文件下所有文件

\*/

public static void listAllFiles(File dir){

File[] files = dir.listFiles();

for(File file : files){

System.out.println(file);

if(file.isDirectory()){

listAllFiles(file);

}

}

}

}

* 1. FilenameFilter接口

package com.learn.io;

import java.io.File;

import java.io.FilenameFilter;

public class FileDemo2 {

public static void main(String[] args) {

File file = new File("E:/STS/java");

FilenameFilter filter = (d,n)->{

return n.endsWith(".java") || new File(d,n).isDirectory();

};

listAllFiles(file,filter);

}

/\*\*

\* 列出文件夹及子文件下所有文件

\*/

public static void listAllFiles(File dir,FilenameFilter filter){

File[] files = dir.listFiles(filter);

for(File file : files){

if(file.isFile()){

System.out.println(file);

}

if(file.isDirectory()){

listAllFiles(file,filter);

}

}

}

}

* 1. IO

流的分类和操作模板

按流向分：输入流、输出流

按数据单位划分：字节流、字符流

按功能划分：节点流、包装流

四大基流：字节输入流（InputStream）、字节输出流（OutputStream）、字符输入流（Reader）、字符输出流（Writer）

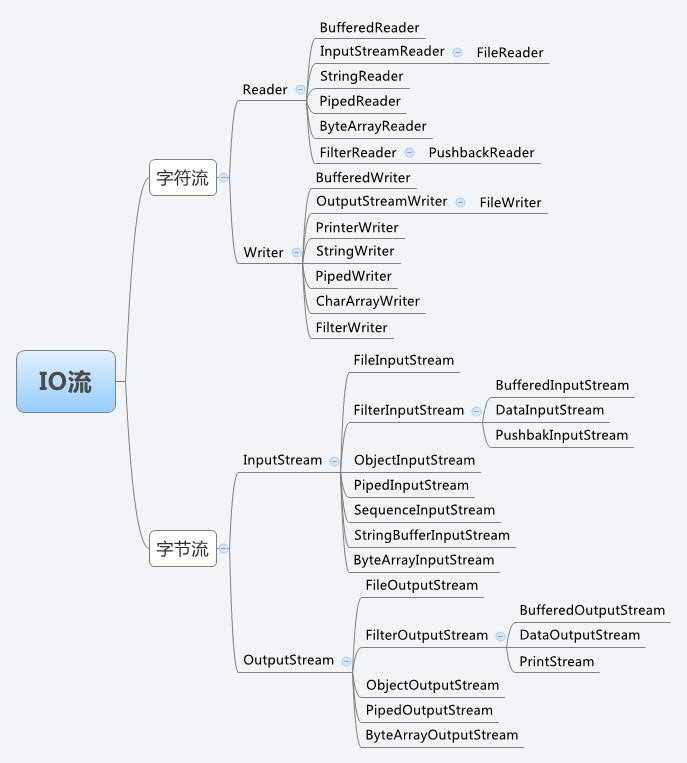
操作：

创建源或目标对象

创建IO流对象

具体的IO操作

关闭资源（很重要）



文件流：

FileInputStream：文件字节输入流

FileOutputStream：文件字节输出流

FileReader：文件字符输入流

FileWriter：文件字符输出流

package com.learn.io;

import java.util.Map.Entry;

import java.io.File;

import java.io.FileInputStream;

import java.io.FileNotFoundException;

import java.io.FileOutputStream;

import java.io.IOException;

import java.util.Properties;

import java.util.Set;

public class SystemPropertiesDemo {

static File file = new File("C:/Users/zeimao77/Desktop/abc.txt");

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

/\*inputDemo();\*/

outputDemo();

}

public static void inputDemo() throws IOException{

Properties pro = System.getProperties();

Set<Entry<Object,Object>> entrys = pro.entrySet();

FileOutputStream fos = new FileOutputStream(file,true);

for(Entry<Object,Object> entry :entrys){

String str = entry.getKey()+"--"+entry.getValue()+"\r\n";

fos.write(str.getBytes());

}

fos.close();

}

public static void outputDemo() throws IOException{

FileInputStream fis = new FileInputStream(file);

byte[] bytestr = new byte[1024];

int len = -1;

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

String str = new String(bytestr);

System.out.println(str);

}

fis.close();

}

}

文件的复制

package com.learn.io;

import java.io.File;

import java.io.FileInputStream;

import java.io.FileOutputStream;

import java.io.IOException;

public class CopyDemo {

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

File file = new File("C:/Users/Mao/Desktop/baidu.html");

File outfile = new File("C:/Users/Mao/Desktop/baidu1.html");

outfile.createNewFile();

FileInputStream fis = new FileInputStream(file);

FileOutputStream fos = new FileOutputStream(outfile);

byte[] bs = new byte[1024];

int len = -1;

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

fos.write(bs);

}

fos.flush();

fis.close();

fos.close();

}

}

另一种动态加载

import java.io.File;

import java.io.FileOutputStream;

import java.io.IOException;

import java.io.InputStream;

import java.io.OutputStream;

public class ProcessDemo {

public static void main(String[] args) {

try {

eval("System.out.print(\"Hello world!!!\");");

} catch (IOException e) {

e.printStackTrace();

}

}

private static void eval(String code) throws IOException{

StringBuilder sBuilder = new StringBuilder();

sBuilder.append("public class Hello{");

sBuilder.append("public static void main(String[] args){");

sBuilder.append(code);

sBuilder.append("}");

sBuilder.append("}");

OutputStream os = new FileOutputStream(new File("Hello.java"));

os.write(sBuilder.toString().getBytes());

os.flush();

os.close();

Process process = Runtime.getRuntime().exec("javac Hello.java");

InputStream is = process.getErrorStream();

byte[] bs = new byte[1024];

while((is.read(bs))!=-1){

System.out.println(new String(bs,0,bs.length));

}

is.close();

process = Runtime.getRuntime().exec("java Hello");

is = process.getInputStream();

while((is.read(bs))!=-1){

System.out.println(new String(bs,0,bs.length));

}

is.close();

}

}

* 1. 文件的字节流：

FileInputStream:

FileOutputStream:

使用字节流操作文本文件或特殊字符容易乱码，建议使用字符流

一般的操作二进制文件必须使用字节流

* 1. 包装流：

相对于节点流更加高级

隐藏了底层的节点流的差异，并对外提供了更加方便的输入输出功能，让我们只关心高级流的操作。

使用包装流包装了节点流程序直接操作处理流，让节点流与底层的设置做IO操作

只需要关闭处理流即可。

包装流在创建的时候，需要有另一个流对象

* 1. 缓冲流：

BufferedInputStream

BufferedOutputStream

操作流的时候，习惯定义一个byte/char数据

Int read():每次都从磁盘读取一个字节，直接操作磁盘文件性能极低

解决方案：定义一个缓冲区 Byte[] buffer = new byte[1024];

在操作字节和字符流时，都习惯使用缓冲流给包装起来，提升IO效率。

package com.learn.io;

import java.io.BufferedInputStream;

import java.io.BufferedOutputStream;

import java.io.FileInputStream;

import java.io.FileOutputStream;

import java.io.IOException;

public class BufferedInputStreamDemo {

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

BufferedOutputStream bos = new BufferedOutputStream(new FileOutputStream("C:/Users/zeimao77/Desktop/abc.txt",true));

bos.write("over".getBytes());

bos.close();

BufferedInputStream bis = new BufferedInputStream(new FileInputStream("C:/Users/zeimao77/Desktop/abc.txt"));

byte[] bs = new byte[1024];

int len =-1;

while((len=bis.read(bs))!=-1){

System.out.println(new String(bs,0,len));

}

bis.close();

}

}

package com.learn.io;

import java.io.BufferedReader;

import java.io.FileReader;

import java.io.IOException;

public class BufferedOutputStreamDemo {

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

BufferedReader reader = new BufferedReader(new FileReader("C:/Users/zeimao77/Desktop/abc.txt"));

String line = null;

while((line=reader.readLine())!=null){

System.out.println(line);

}

reader.close();

}

}

* 1. 转换流：

把字节注转换成字符流

InputStreamReader：把字节输入流转成字符输入流

OutputStreamWriter：把字节输出流转换成字符输出流

字节流可以操作一切文件，但字符流一般只用来操作文本文件，本身是对字节流的增加，可以有效防止乱码，所以字节流可以转字符流，但字符流不转字节流

有时候其它程序输入的是一个字节流，这里就需要进行转换成字符流

* 1. 内存流

内存流又叫数组流，就是先把数据临时存放在数组中，待会再从数组中获取出来。

分字节内存流、字符内存流、字符串流。

* + 1. 字节内存流

package com.learn.io;

import java.io.ByteArrayInputStream;

import java.io.ByteArrayOutputStream;

import java.io.IOException;

public class BayteArrayDemo {

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

//内存流的目标是内存，所以不需要输入目标

ByteArrayOutputStream bos = new ByteArrayOutputStream();

bos.write("abcdefg".getBytes());

//使用存储的临时数据

byte[] buffer = bos.toByteArray();

System.out.println(new String(buffer));

//内存输入流可以将数据由内存读取到程序

ByteArrayInputStream bis = new ByteArrayInputStream(buffer);

byte[] bys = new byte[5];

int len=-1;

while((len=bis.read(bys))!=-1){

System.out.println(new String(bys,0,len));

}

bis.close();

bis.close();

}

}

* + 1. 字符内存流

package com.learn.io;

import java.io.ByteArrayInputStream;

import java.io.ByteArrayOutputStream;

import java.io.IOException;

public class ByteArrayDemo {

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

//字节数组输出流：程序->内存

ByteArrayOutputStream bos = new ByteArrayOutputStream();

bos.write("abcdefg".getBytes());

byte[] buffer = bos.toByteArray();

ByteArrayInputStream bis = new ByteArrayInputStream(buffer);

byte[] bys = new byte[5];

int len = -1;

while((len=bis.read(bys))!=-1){

System.out.println(new String(bys,0,len));

}

bis.close();

bos.close();

}

}

* + 1. 字符串流

应用比较少，一般只有受特殊类型限定时使用

package com.learn.io;

import java.io.IOException;

import java.io.StringReader;

import java.io.StringWriter;

public class StringIODemo {

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

StringWriter sWriter = new StringWriter();

sWriter.write("龙哥今年几岁啊？");

sWriter.write("年年十七岁！");

System.out.println(sWriter.toString());

StringReader sReader = new StringReader(sWriter.toString());

char[] buffer = new char[1024];

int len=-1;

while((len=sReader.read(buffer))!=-1){

System.out.println(new String(buffer,0,len));

}

sWriter.close();

}

}

* 1. 合并流

package com.learn.io;

import java.io.File;

import java.io.FileInputStream;

import java.io.IOException;

import java.io.SequenceInputStream;

public class SequenceStreamDemo {

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

SequenceInputStream sis = new SequenceInputStream(

new FileInputStream(new File("C:/Users/zeimao77/Desktop/abc.txt")),

new FileInputStream(new File("C:/Users/zeimao77/Desktop/bcd.txt")));

byte[] buf = new byte[1024];

int len=-1;

while((len=sis.read(buf))!=-1){

System.out.println(new String(buf,0,len));

}

sis.close();

}

}

* 1. 序列化与反序列化

序列化：指把内存中的对象数据，存储到磁盘文件中或者传递给其他网络的节点（在网络上传输），我们把这个过程称之为序列化。

反序列化：把磁盘文件中的对象数据或者把网络节点上的对象数据，恢复成java对象的过程。

如果某些数据不需要做序列化，比如密码，此时我们可以将其字段设置成瞬态的。理论上来静态的字段和瞬态的字段是不可以做序列化的

序列化与反序列化的版本问题：

反序列化java对象时必须提供该对象的class文件，现在问题是随着项目的升级，class文件也会升级（增删改字段），如何保证两个class文件的兼容性呢？java通过serialVersionUID（序列化版本号）来判断字节码是否发生改变。如果不显示定义序列化版本号，那么该类的序列化版本号会由JVM根据类信息计算而来，而修改后的计算值和以前一般不同，众而导致对象反序列化因为版本不兼容而失败的问题。



package com.learn.io;

import java.io.Serializable;

public class User implements Serializable{

private static final long serialVersionUID = -6958483875070424723L;

private String name;

private Integer age;

private String sex;

transient private String password;

public User(String name, Integer age, String sex,String password) {

this.name = name;

this.age = age;

this.sex = sex;

this.password = password;

}

@Override

public String toString() {

return "User [name=" + name + ", age=" + age + ", sex=" + sex + "]";

}

}

package com.learn.io;

import java.io.File;

import java.io.FileInputStream;

import java.io.FileNotFoundException;

import java.io.FileOutputStream;

import java.io.IOException;

import java.io.ObjectInputStream;

import java.io.ObjectOutputStream;

public class ObjectStreamDemo {

public static void main(String[] args){

try {

File file = new File("file/obj.txt");

writeObj(file);

readObj(file);

} catch (Exception e) {

e.printStackTrace();

}

}

public static void writeObj(File file) throws FileNotFoundException, IOException{

ObjectOutputStream oos = new ObjectOutputStream(new FileOutputStream(file));

oos.writeObject(new User("张三", 30, "男","abcdefg"));

oos.close();

}

public static void readObj(File file) throws FileNotFoundException, IOException, ClassNotFoundException{

ObjectInputStream ois = new ObjectInputStream(new FileInputStream(file));

User user = (User) ois.readObject();

ois.close();

System.out.println(user);

}

}

* 1. 打印流

printStream：字节打印流

printWriter：字符打印流

对于printWriter来说，当启用字段刷新后，PrintWriter ps = new PrintWriter(new FleOutputStream(new File(“/file/out.txt”)),true);

调用println或print或format方法，便会立即刷新操作，如果没有开始自动刷新，则需要手动刷新或者当前缓冲区满的时候再自动刷新。

使用打印流作为输出流，此时的输出操作会特别简单，因为在打印流中

提供了print方法、println方法，可以支持打印各种数据类型。

package com.learn.io;

import java.io.File;

import java.io.FileNotFoundException;

import java.io.FileOutputStream;

import java.io.PrintStream;

import java.util.Date;

public class PrintStreamDemo {

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

PrintStream ps = new PrintStream(new FileOutputStream(new File("C:/Users/zeimao77/Desktop/abc.txt")));

ps.print("abc");

ps.println(123);

ps.println(new Date());

ps.close();

}

}

格式化输出

PrintStream p = System.out;

p.printf("姓名：%s,年龄%d","张三",22);

* 1. 标准输入输出流

System.in;标准输入流

System.out;标准输出流

System.err;标准错误输出流

标准的输入输出流的重定向操作：

标准的输入：通过键盘录入数据，重定向不再是键盘，而是文件。

public static void setIn(InputStream in)重新分配“标准”输入流。

public static void setOut(PrintStream out) 重新分配“标准”输出流。

public static void setErr(PrintStream err)重新分配“标准”错误输出流。

package com.learn.io;

import java.io.File;

import java.io.FileInputStream;

import java.io.FileOutputStream;

import java.io.IOException;

import java.io.PrintStream;

public class StandardIODemo {

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

System.setIn(new FileInputStream(new File("C:/Users/zeimao77/Desktop/abc.txt")));

System.setOut(new PrintStream(new FileOutputStream(new File("C:/Users/zeimao77/Desktop/bcd.txt"))));

byte[] bs = new byte[10];

int len = -1;

while((len=System.in.read(bs))!=-1){

System.out.println(new String(bs,0,len));

}

}

}

* 1. Scanner

package com.learn.io;

import java.io.File;

import java.io.FileNotFoundException;

import java.util.Scanner;

public class ScannerDemo {

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

Scanner scan = new Scanner(new File("C:/Users/zeimao77/Desktop/abc.txt"));

while(scan.hasNextLine()){

System.out.println(scan.nextLine());

}

scan.close();

}

}

* 1. 数据流

数据流提供了可以读/写任意数据类型的方法。

DataoutputStream：提供了wirtexxx(xxx value)方法

dataInputStream：提供了readxxx()方法

注意：

Writexxx和readxxx必需要对应起来，writeByte写出的数据，必需用readByte读回来。

package com.learn.io;

import java.io.DataInputStream;

import java.io.DataOutputStream;

import java.io.File;

import java.io.FileInputStream;

import java.io.FileOutputStream;

import java.io.IOException;

import java.io.PrintStream;

public class DataStream {

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

File file = new File("file/abc.tx");

write(file);

read(file);

}

public static void read(File file) throws IOException{

DataInputStream dis = new DataInputStream(new FileInputStream(file));

PrintStream p = System.out;

p.println(dis.readInt());

p.println(dis.readUTF());

p.println(dis.readByte());

dis.close();

}

public static void write(File file) throws IOException{

DataOutputStream dos = new DataOutputStream(new FileOutputStream(file));

dos.writeInt(35535);

dos.writeUTF("中国");

dos.writeByte(65);

dos.close();

}

}

* 1. RandomAccessFile

RandomAccessFile(File file, String mode) 创建从中读取和向其中写入（可选）的随机存取文件流，该文件由 File 参数指定。

RandomAccessFile(String name, String mode) 创建从中读取和向其中写入（可选）的随机存取文件流，该文件具有指定名称。

mode 参数指定用以打开文件的访问模式。允许的值及其含意为：

"r" 以只读方式打开。调用结果对象的任何 write 方法都将导致抛出 IOException。

"rw" 打开以便读取和写入。如果该文件尚不存在，则尝试创建该文件。

"rws" 打开以便读取和写入，对于 "rw"，还要求对文件的内容或元数据的每个更新都同步写入到基础存储设备。

"rwd" 打开以便读取和写入，对于 "rw"，还要求对文件内容的每个更新都同步写入到基础存储设备。

package com.learn.io;

import java.io.File;

import java.io.FileNotFoundException;

import java.io.IOException;

import java.io.PrintStream;

import java.io.RandomAccessFile;

public class AccessFileDemo {

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

File file = new File("file/bcd.txt");

write(file);

read(file);

}

public static void read(File file) throws IOException{

RandomAccessFile raf = new RandomAccessFile(file, "r");

System.out.println("当前指针："+raf.getFilePointer());//获取指针

System.out.println(raf.readByte());

System.out.println("当前指针："+raf.getFilePointer());

raf.seek(0);//可以设置指针

raf.skipBytes(1);//跳过指针

System.out.println(raf.readUTF());

raf.close();

}

public static void write(File file) throws IOException{

RandomAccessFile raf = new RandomAccessFile(file,"rw");

raf.writeByte(65);

raf.writeUTF("龙哥");

raf.close();

}

}

setLength(long newLength) 设置此文件的长度。

RandomAccessFile经常用来做多线程断点下载。

* 1. 管道流

管道流可以实现两个线程之间的数据交互。

package com.learn.io;

import java.io.IOException;

import java.io.PipedInputStream;

import java.io.PipedOutputStream;

public class A implements Runnable {

private PipedOutputStream pos = new PipedOutputStream();

public static void main(String[] args) {

A a0 = new A();

Thread a = new Thread(a0);

Thread b = new Thread(new B(a0.pos));

a.start();

b.start();

}

@Override

public void run() {

try {

for(int i=65;i<95;i++){

pos.write(i);

}

pos.close();

} catch (IOException e) {

e.printStackTrace();

}

}

}

class B implements Runnable{

PipedInputStream pis=null;

B(PipedOutputStream src){

try {

pis = new PipedInputStream(src);

} catch (IOException e) {

e.printStackTrace();

}

}

@Override

public void run() {

try {

int len=-1;

while((len=pis.read())!=-1){

System.out.println((char)len);

}

} catch (IOException e) {

e.printStackTrace();

}

}

}

* 1. NIO

package com.learn.io;

import java.io.IOException;

import java.nio.file.Files;

import java.nio.file.Paths;

import java.nio.file.StandardCopyOption;

/\*\*

\* 复制文件，比较好用

\* @author zeimao77

\*

\*/

public class NIODemo {

public static void main(String[] args) {

try {

Files.copy(Paths.get("file/bcd.txt"),Paths.get("abc.txt"),StandardCopyOption.REPLACE\_EXISTING);

Files.move(Paths.get("file/cde.txt"),Paths.get("file/def.txt"),StandardCopyOption.ATOMIC\_MOVE);

} catch (IOException e) {

e.printStackTrace();

}

}

}

1. 数据结构与算法

数据结构是计算机存储、组织数据的方式，数据结构是指相互之间一种或多种特定关系的数据元素的集合，精心选择数据结构可以带来更高的支行或存储效率，数据结构往往同高效的检索算法和索引技术有关。数据结构在计算机中表示（映像）它包括数据元素的机内表示和关系的机内表示。

数据元素之间关系特征，一共有四种逻辑结构：

* 集合
* 线性结构
* 树型结构
* 图型结构

常见数据结构：数组【Array】、栈【stack】、链表【Linkded List】、哈希表【Hash】、队列【Queue】、堆【Heap】、图【Grap】、树【Tree】

JAVA中的集合框架其实就是数据结构的实现的封装。

不同的数据结构的操作性能是不同的。

最简单的数据结构是数组

* 1. HelloWorld

使用数组模拟球队上场安排，具体需求：

1. 初始化一个容量为5的线性列表，准备用来存储场上的五个球衣号码
2. 安排5个球员上场
3. 查询指定位置的球员的球衣号码是多少
4. 根据球衣号码查询球员位置，
5. 替换场上某位置的球员
6. 替换球衣号为22的球员
7. 把索引为2的球员罚下场，没有替补
8. 按照位置顺序打印球衣号码

package com.learn.datastructure;

import java.util.Arrays;

public class BallDemo {

private static Integer[] arr = null;

private static int size=0;

public static void main(String[] args) {

init(5);

add(11);add(22);add(33);add(44);add(55);

System.out.println(get(2));

System.out.println(getByPlayerNum(22));

repleat(22, 222);

delete(3);

print();

dilatation(10);

add(66);add(77);

print();

}

//1. 初始化一个容量为5的线性列表，准备用来存储场上的五个球衣号码

public static void init(int initialCapacity){

arr = new Integer[initialCapacity];

}

//2. 安排球员上场

public static void add(Integer playerNum){

arr[size] = playerNum;

size++;

}

//3.查询指定位置的球员的球衣号码是多少

public static Integer get(Integer index){

return arr[index];

}

//4.根据球衣号码查询球员位置

public static Integer getByPlayerNum(Integer palyerNum){

for(int i=0;i<arr.length;i++){

if(arr[i].equals(palyerNum)){

return i;

}

}

return -1;

}

//6.替换球衣号为22的球员

public static void repleat(Integer oldPlayer,Integer newPlayer){

for(int i=0;i<arr.length;i++){

if(arr[i].equals(oldPlayer)){

arr[i] = newPlayer;

return;

}

}

}

//7.把索引为2的球员罚下场，没有替补

public static void delete(int index){

for(int i=index;i<size-1;i++){

arr[i] = arr[1+i];

}

arr[size-1] = null;

size--;

}

//8.按照位置顺序打印球衣号码

public static void print(){

StringBuilder sb = new StringBuilder();

sb.append("[");

Arrays.stream(arr).forEach(e->sb.append(e+","));

sb.deleteCharAt(sb.length()-1);

sb.append("]");

System.out.println(sb.toString());

}

//数组扩容

public static void dilatation(int length){

try {

if(length<arr.length){

throw new Exception("长度不可以小于原数组长度");

}

} catch (Exception e) {

e.printStackTrace();

}

arr = Arrays.copyOf(arr, length);

}

}

* 1. 字符串
* 常见运算：
* 赋值运算
* 判断相等
* 求串长度
* 联接运算
* 求子串
* 定位index(str,i)
* 替换
* 插入
* 删除

串的存储方式一般取决于对串所进行的运算。如果串只是作为常量出现，则只需要作为一个字符的序列存储即可。但在多数非数值处理程序中，串也是操作对象，在程序执行的过程中，串的值可变。

* + 1. 静态存储结构

类似于线性表的顺序存储结构，用一组地址连接的存储单元存储串的字符序列。由于一个字符只占一个字节，而现在大多数计算机的存储器地址是采用的字编址，一个字占多个字节，因此顺序存储结构方式有两种：

非紧缩存储结构：每个存储单元只存放一个字符。这种方式空间利用率低下，但存储方式不需要分离，因而程序处理字符的速度高。

紧缩存储结构：一个字节存储一个字符，充分利用存储空间，但是在串的运算时，若要分离某一部分字符时，则非常麻烦。

* + 1. 动态存储结构

串的动态存储结构有两种存储方式：

块链结构：与线性表的链式结构类似，串值也可以采用链表方式存储。串的链式存储结构中每个结点包含字符域和结点链接指针域，字符域用于存放字符，指针域用于存入指向下一个结点的指针。用单链表存放串，每个结点仅存放一个字符，因此结点的指针域所占用的空间比字符域所占空间要大得多，为了提高空间利用率，可以使一个结点存放多个字符，称为块链结构。每个结点存放4个字符。当链表的最后一个结点不能被全占满时，通常补上“#”或其它非串值字符。块链结构虽然链表结构灵活，但因为要存放指针，会受存储密度限制。

堆结构

* 1. 数组

常见运算：

* + 给定下标，存取元素
  + 给定下标，修改对应元素及其数据项

LOC(ai) = LOC(a1) + (i-1)\*m

N维数组是一个线性表，数据元素是一个n-1维的数组。

* + 1. 矩阵压缩

在实际一些高阶矩阵中，同时在矩阵中存在许多零值或相同的值，为了节省存储空间，可以对这类矩阵进行压缩，一般思想是将相同元素分配到同一空间，零值不分配空间，一般将需要压缩的矩阵分为特殊矩阵和稀疏矩阵。

特殊矩阵又分对称矩阵和三角矩阵，对称矩阵是同对称相同的元素分配到同一空间，三角矩阵是将常数三角分配到同一空间，如果为零值，则一般不分配空间。

稀疏矩阵中多数元素都是零值，而且非零元素的分布没有规律时，该矩阵就是稀疏矩阵。它的压缩方法分三元组顺序表，行列逻辑链接顺序表，十字链表，可以参考图的存储。

* 1. ArrayList

线性表的常见运算：

* 初始化
* 判断是否为空
* 求长度
* 取第i个元素
* 查找满足条件的元素
* 在某个特定位置前插入元素
* 删除第i个元素
* 置空
* 查找第i个元素的前驱
* 查找第i个元素的后继
* 排序元素

保存操作：Arraylist(N);

如果把数据保存在数组的最后的位置，至少需要操作一次;

如果把数据保存在数组的第一个位置，至少需要操作N次;

平均操作（N+1）/2次

如果需要扩容，性能更低。

删除操作:

如果删除的是最后一个元素，操作一次

如果删除的是第一个元素，操作N次

修改操作：

操作一次

查询操作：

如果根据索引查询，操作一次

如果根据元素查询，此时使用线性搜索。平均操作(1+N)/2次；

我们发现：ArrayList做基于数组的结构做查询和修改是非常快的，但是做保存和删除操作比较慢。

package com.learn.datastructure;

import java.util.Arrays;

public class MyArrayList {

private Object[] elements = null;

private int size=0;

private static int DEFAULT\_SIZE = 10;

public static void main(String[] args) {

MyArrayList list = new MyArrayList();

list.add("abc");

list.add(123);

System.out.println(list);

list.delete(0);

System.out.println(list);

}

MyArrayList(){

elements = new Object[DEFAULT\_SIZE];

}

//1.初始化

MyArrayList(int initSize){

if(initSize<0){

throw new IllegalArgumentException("参数不可以为负数;");

}

DEFAULT\_SIZE = initSize;

elements = new Object[DEFAULT\_SIZE];

}

//2.添加元素

public void add(Object element){

if(elements.length==size){

this.dilatation(size+10);

}

elements[size++] = element;

}

//3.获取元素

public Object get(Integer index){

return elements[index];

}

//4.查询元素索引

public Integer getByelement(Object element){

for(int i=0;i<elements.length;i++){

if(elements[i].equals(element)){

return i;

}

}

return -1;

}

//6.替换元素

public void repleat(Integer index,Object element){

elements[index]=element;

}

//7.删除元素

public void delete(int index){

for(int i=index;i<size-1;i++){

elements[i] = elements[1+i];

}

elements[--size] = null;

}

//8.按照位置顺序打印球衣号码

@Override

public String toString(){

StringBuilder sb = new StringBuilder();

sb.append("[");

Arrays.stream(elements).forEach(e->sb.append(e+","));

sb.deleteCharAt(sb.length()-1);

sb.append("]");

return sb.toString();

}

//数组扩容

public void dilatation(int length){

try {

if(length<elements.length){

throw new IllegalArgumentException("扩容不可以超出当前数组长度");

}

} catch (Exception e) {

e.printStackTrace();

}

elements = Arrays.copyOf(elements, length);

}

public int size(){

return size;

}

public boolean isEmpty(){

return size==0;

}

}

* 1. 链表

单向链表：只可以从头遍历到尾或从尾遍历到头;

双向链表：既可以头头遍历到尾，也可以从尾遍历到头;

性能分析：

添加操作

双向链表可以直接获取自己的第一个或最后一个节点，如果添加的节点在第一或者最后只需要操作一次。不支持中间插入。

删除操作

如果删除自己第一个或最后一个只需要操作一次；

如果操作中间元素，平均操作（1+N）/2次，找到之后操作一次

查询操作

平均操作（1+N）/2次

修改操作

平均操作（1+N）/2次

我们发现：链表对

package com.learn.datastructure;

import java.util.function.Predicate;

public class MyLinkedList {

private Node first;

private Node last;

private int size;

public static void main(String[] args) {

MyLinkedList list = new MyLinkedList();

list.removeFirst();

System.out.println(list);

}

public boolean remove(Object obj){

if(size<=0){

return false;

}

Node current = this.first;

Node result = null;

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

if(current.ele.equals(obj)){

result = current;

break;

}

current = current.next;

}

if(result==first){

return removeFirst();

}else if(result == last){

return removeLast();

}else if(result == null){

return false;

}else{

current.prev.next = current.next;

current.next.prev = current.prev;

size--;

return true;

}

}

public boolean removeFirst(){

try {

first = first.next;

first.prev = null;

size--;

} catch (Exception e) {

return false;

}

return true;

}

public boolean removeLast(){

try {

last = last.prev;

last.next = null;

size--;

} catch (Exception e) {

return false;

}

return true;

}

void addLast(Object obj){

Node node = new Node(obj);

if(size == 0){

first = node;

last = node;

}else{

last.next = node;

node.prev = last;

last = node;

}

size++;

}

void addFirst(Object obj){

Node node = new Node(obj);

if(size==0){

first = node;

last = node;

}else{

node.next = first;

first.prev = node;

first = node;

}

size++;

}

@Override

public String toString() {

if(size==0){

return "[]";

}

StringBuilder sb = new StringBuilder();

Predicate<Node> pre =e->e.next==null;

Node current = this.first;

sb.append("[");

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

sb.append(current.ele);

if(pre.test(current)){

sb.append("]");

}else{

sb.append(",");

}

current = current.next;

}

return sb.toString();

}

class Node{

Node prev;

Node next;

Object ele;

Node(Object obj){

this.ele=obj;

}

}

}

* 1. 队列

常见运算：

* 初始化队列
* 判断空队
* 求队列长度
* 读队列头
* 入队
* 出队
* 清空队列

队列是一种特殊的线性表，特殊之处在于它只允许在表的前端进行删除操作，而在表的后端进行插入操作，和栈一样，队列一种操作受限制的线性表。进行插入操作的端称为队尾，进行删除操作的端称队头。

单向队列（Queue）：先进先出，只能从队列尾插入数据，只能从队头删除数据。

package com.learn.datastructure;

public class MyDeque extends MyLinkedList {

public static void main(String[] args) {

MyDeque deque = new MyDeque();

deque.addFirst("abc");

deque.addFirst(123);

System.out.println(deque.getFirst());

System.out.println(deque);

}

public Object getFirst(){

return first.ele;

}

public Object getLast(){

return last.ele;

}

}

* 1. 栈

常见运算：

* 初始化栈
* 判断空栈
* 求栈深
* 取栈顶
* 入栈
* 出栈
* 清空栈

栈是一种运算受限的线性表，

其限制是允许在表的一端进行插入、删除操作，这一端被称为栈顶，相对地，把另一端称为栈底。向一个栈插入新元素又称作进栈，它是把新元素放到栈顶元素的上面，使之成为新的栈顶元素;从一个栈删除元素又称作出栈或退栈，它是把栈顶元素删除掉，使其相信的元素成为新的顶元素。

package com.learn.datastructure;

public class MyStack extends MyArrayList {

public static void main(String[] args) {

MyStack stack = new MyStack();

stack.push("abc");

stack.push("bcd");

stack.push("cde");

System.out.println(stack.peek());

stack.pop();

System.out.println(stack);

}

public void push(Object obj){

add(obj);

}

public void pop(){

delete(size()-1);

}

public Object peek(){

return get(size()-1);

}

}

* 1. 散列表

在一般的数组中，元素在的索引位置是随机的，元素的取值和元素的位置之间不存在确定的关系，因此，在数组中查找特定的值时，需要把查找值和一系列的元素进行比较

此时的查询效率依赖于查找过程中所进行的比较次数。

如果元素的值（value）和在数组中的索引位置（Index）有一个确定的对应关系（hash）。

公式为：index=hash(value);

那么对于给定的值，只要调用上述的hash(value)方法，就可以找到数组中取值为value的元素的位置。

如果数组中元素的值和索引位置存在对应的关系，这样的数组就称为哈希表，可以看出哈希表最大的优点是提供查找数据的效率，一般情况下，我们是不会把哈希码作为元素在数组中的索引位置的，因为哈希码很大，数组长度有限，会造成索引越界问题。

这个时候，我们可以在哈希码和元素位置之间做某种映射关系。

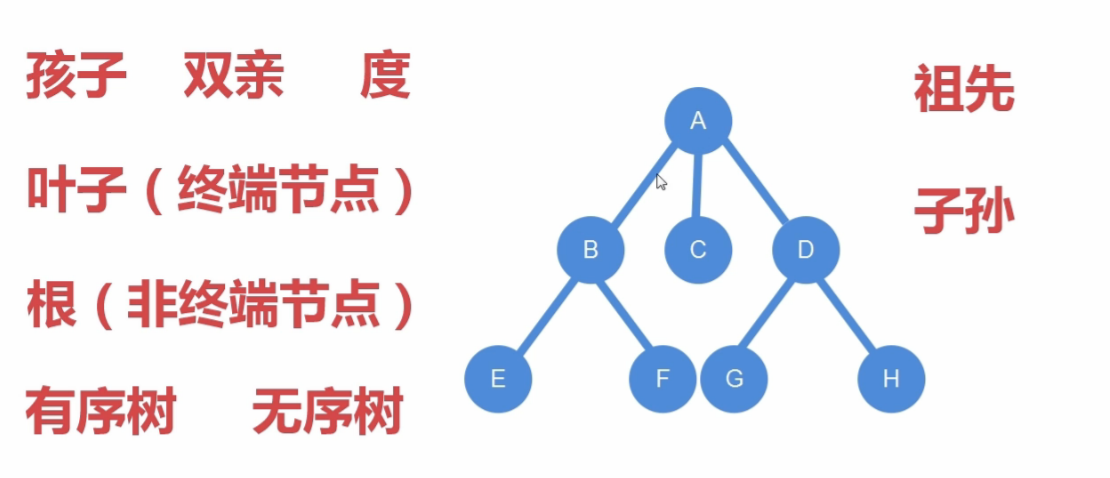
元素值—hash(value) —哈希码—元素存储索引

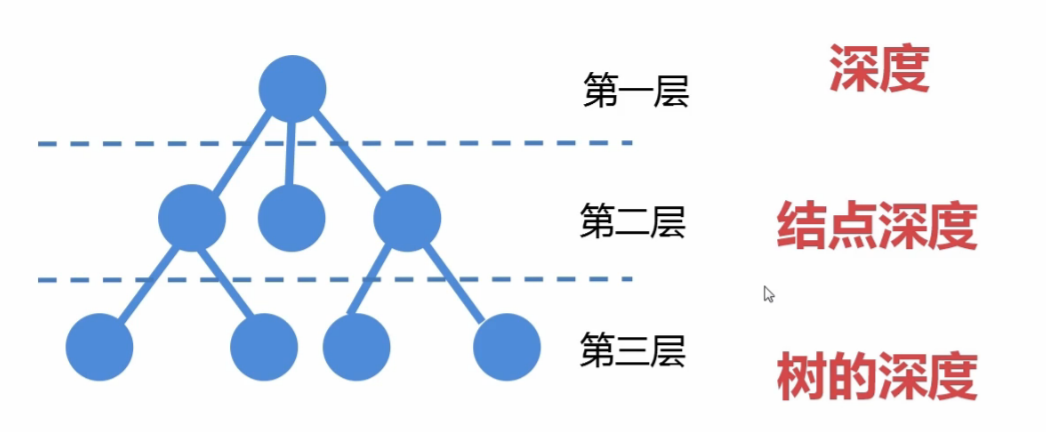
每一个对象的哈希码是不同的

哈希表的插入和查找是很优秀的，可是当哈希表接近装满时，因为数组的扩容问题，性能会比较低

* 1. 树

树是节点的有限集合





* + 1. 二叉树

二叉树结点最多有两棵子树，所以二叉树中不存在度大于2结点。

斜树：只能朝一边斜

满二叉树：

叶子只能出现在最下一层

非叶子结点的度一定是二

在同样深度的二叉树中，满二叉树的结点个数一定最多，同时叶子也是最多的。

完全二叉树：

叶子只出现在最下两层。

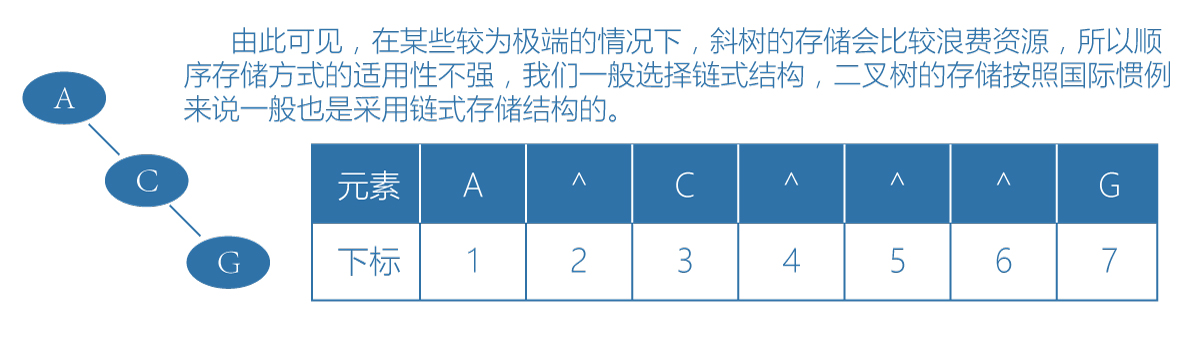
最下层的叶子一定集中在左部连续位置

倒数第二层，若有叶子结点，一定都在右部连续位置。

如果结点度为1，则该结点只能有左结节

同样结点的二叉树，完全二叉树的深度是最小的。

满二叉树一定是完全二叉树。



二叉树的遍历：

是指从根结点出发，按照某种次序依次访问二叉树中的所有结点，使得每一个结点被访问且仅被访问一次。

二叉树的遍历方式可以很多，但一般可以分为四类：

* 前序遍历

若二叉树为空，则空操作返回，否则先访问根结点，然后访问左子树，再访问右子树。

* 中序遍历

若树为空，则空操作返回，否则从根结点开始，再访问根节点，最后访问右结点

* 后序遍历

若树为空，则空操作返回，否则从左到右先叶子结点，最后访问根结点。

* 层序遍历

package com.learn.datastructure;

import java.util.ArrayList;

import java.util.LinkedList;

import java.util.Queue;

public class Node {

private Integer id;

private Object data;

private Node rchild,lchild;

ArrayList<Node> list = null;

Node(Integer id,Object data){

this.id = id;

this.data = data;

}

Node searchNode(Integer id){

if(this.id == id){

return this;

}

if(this.lchild != null && lchild.id == id){

return lchild;

}

if(this.lchild != null){

lchild.searchNode(id);

}

if(this.rchild != null && rchild.id ==id){

return rchild;

}

if(this.rchild != null){

rchild.searchNode(id);

}

return null;

}

boolean addNode(Integer id,Integer lor,Node node){

Node node1 = searchNode(id);

if(node1 != null){

switch (lor) {

case 0:

node1.lchild = node;

break;

case 1:

node1.rchild = node;

break;

default:

throw new IllegalArgumentException("左右节点参数异常");

}

}

return false;

}

/\*\*

\* 前序遍历;

\*/

void show1(){

System.out.println(this.data);

if(this.lchild!=null){

lchild.show1();

}

if(this.rchild != null){

rchild.show1();

}

}

/\*\*

\* 中序遍历

\*/

void show2(){

if(this.lchild!=null){

lchild.show2();

}

System.out.println(this.data);

if(this.rchild != null){

rchild.show2();

}

}

/\*\*

\* 后序遍历

\*/

void show3(){

if(this.lchild!=null){

lchild.show3();

}

if(this.rchild != null){

rchild.show3();

}

System.out.println(this.data);

}

/\*\*

\* 按层遍历

\*/

void show4(){

Queue<Node> queue = new LinkedList<>();

Node current = null;

queue.offer(this);

while(!queue.isEmpty()){

current = queue.remove();

System.out.println(current.data+"--->");

if(current.lchild!=null){

queue.offer(current.lchild);

}

if(current.rchild!=null){

queue.offer(current.rchild);

}

}

}

}

package com.learn.datastructure;

import java.util.ArrayList;

import java.util.Arrays;

import java.util.List;

import java.util.stream.Collectors;

import sun.font.CreatedFontTracker;

class DB{

Integer id;

Integer pid;

Integer lor;

Object data;

List<DB> list = null;

DB(){

list = Arrays.asList(

new DB(0, -1, null, "A"),

new DB(1, 0, 0, "B"),

new DB(2, 0, 1, "E"),

new DB(3, 1, 0, "C"),

new DB(4, 1, 1, "D"),

new DB(5, 2, 1, "F"),

new DB(6, 5, 0, "G")

);

}

public DB(Integer id, Integer pid, Integer lor, Object data) {

this.id = id;

this.pid = pid;

this.lor = lor;

this.data = data;

}

DB getById(Integer id){

for(DB d:list){

if(d.id==id){

return d;

}

}

return null;

}

List<DB> getByPid(Integer pid){

return list.stream()

.filter(e->e.pid==pid)

.collect(Collectors.toCollection(ArrayList::new));

}

}

public class NodeTest {

static DB db = new DB();

public static void main(String[] args) {

DB db1 = db.getById(0);

Node node = new Node(db1.id, db1.data);

node = createTree(node, 0);

System.out.println("\*\*\*\*\*\*前序遍历\*\*\*\*\*\*\*");

node.show1();

System.out.println("\*\*\*\*\*\*中序遍历\*\*\*\*\*\*\*");

node.show2();

System.out.println("\*\*\*\*\*\*后序遍历\*\*\*\*\*\*\*");

node.show3();

System.out.println("\*\*\*\*\*\*层序遍历\*\*\*\*\*\*\*");

node.show4();

}

public static Node createTree(Node node,Integer pid){

List<DB> list = db.getByPid(pid);

for(DB d:list){

Node node1 = new Node(d.id, d.data);

node.addNode(d.pid, d.lor, node1);

if(db.getByPid(d.id).size()>0){

createTree(node1, d.id);

}

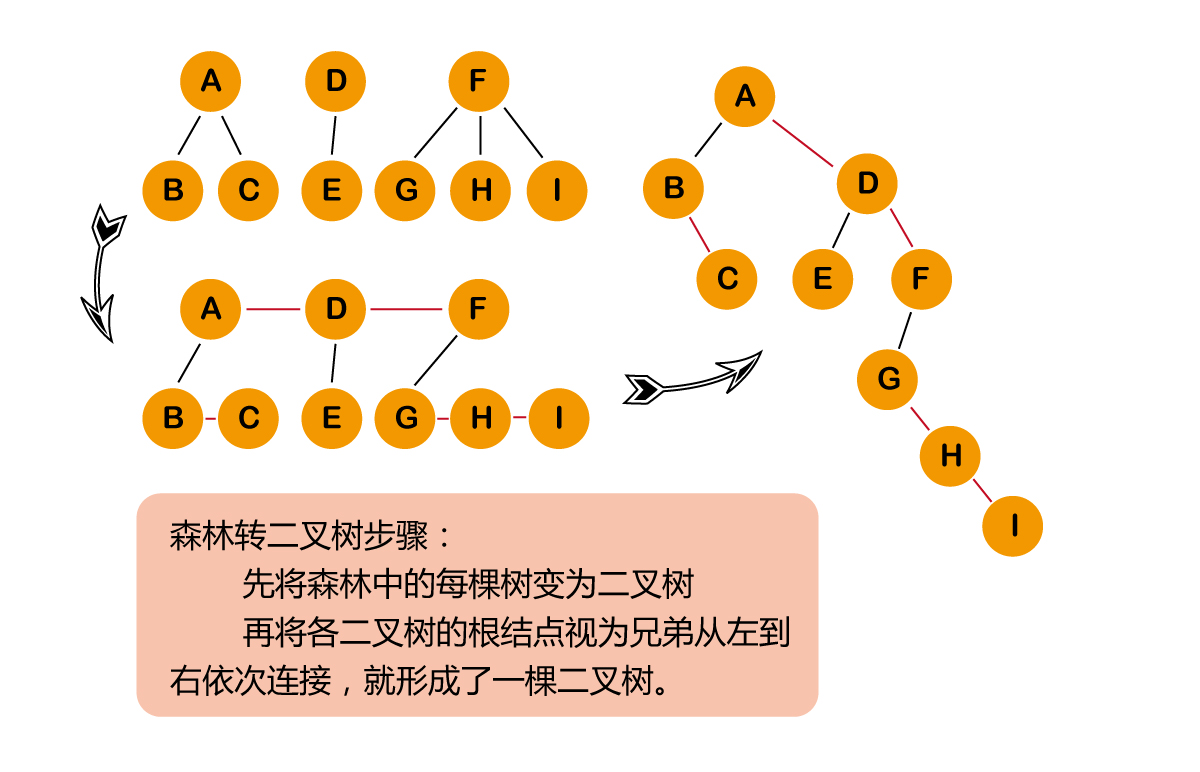
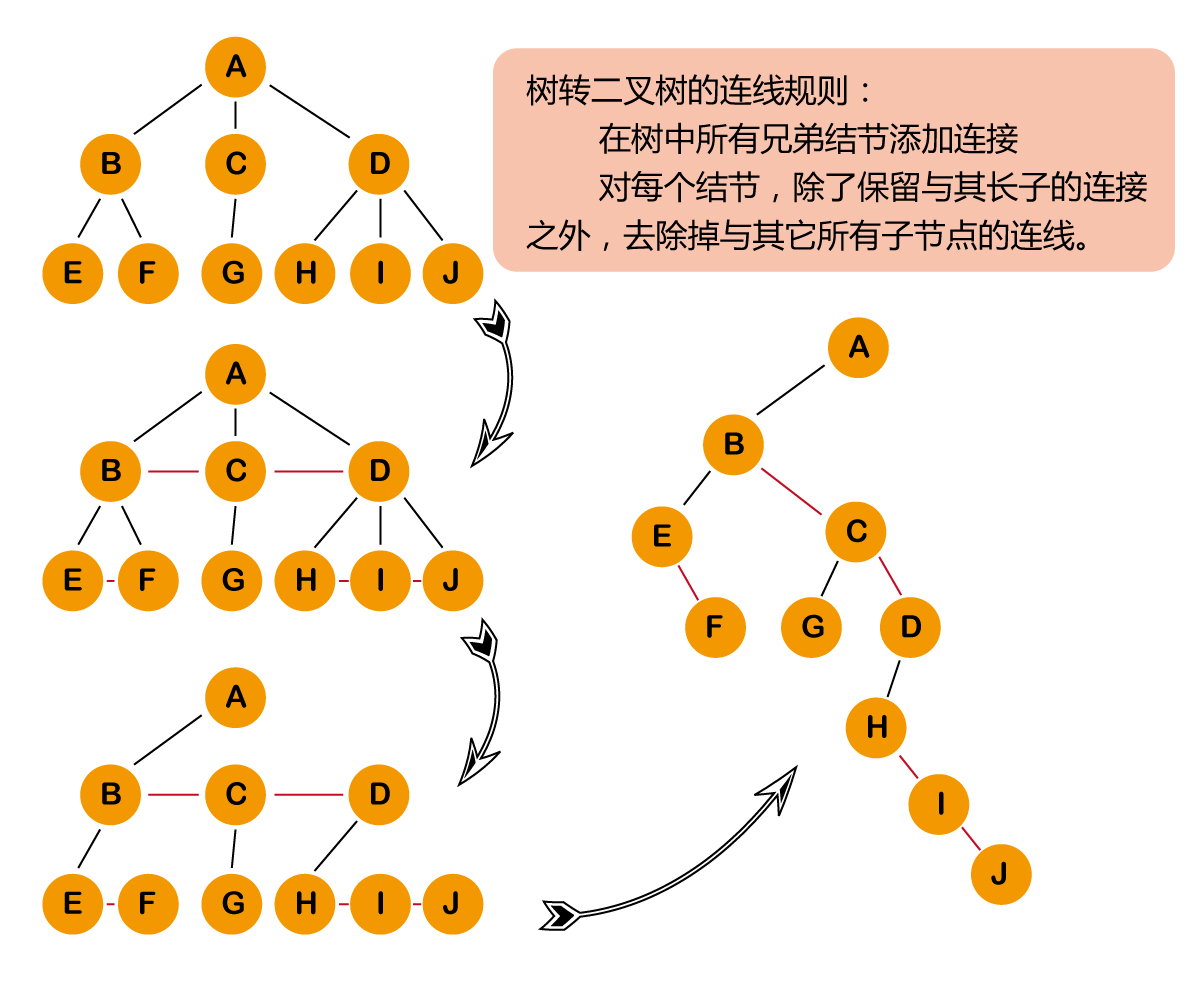
}

return node;

}

}

* + 1. 二叉树与普通树之间的转换



由图我们可以发现，由树转化成的二叉树仅有左子树，而由森林转化而成的二叉树不仅有左子树，而且还有右子树。

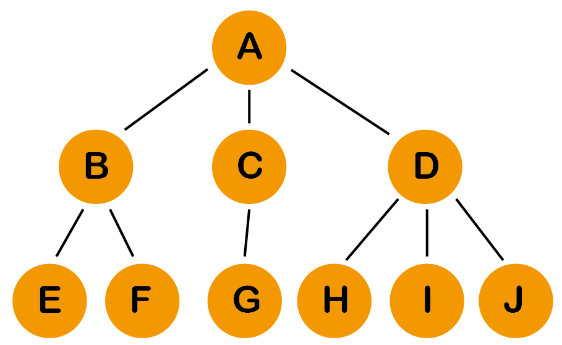
二叉树转回树或者森林的过程正好相反

第一步:或结点x是其双亲y的左孩子，则把x的右孩子都与y用连线连接起来。

第二步:去掉所有双亲到右孩子之间的连线。

树的遍历：

树的遍历为分两种方式：先根遍历：先访问树的根结点，然后依次先根遍历根的每一棵子树。后根遍历：先依次遍历每棵子树，然后再访问根节点



先根遍历的访问结果：

ABEFCGDHIJ

后根遍历的访问结果：

EFBGCHIJDA

林林的遍历也分为前序遍历和后序遍历，其实就是按照树的先根遍历和后根遍历依次访问森林的每一棵树

我们发现：树、森林的先根遍历和二叉树的前序（序）遍历结果是相同的，森林、树的后根（序）遍历和二叉树的中序遍历结果相同！

* + 1. 赫夫曼树与赫夫曼编码
  1. 图
     1. 图的基本概念

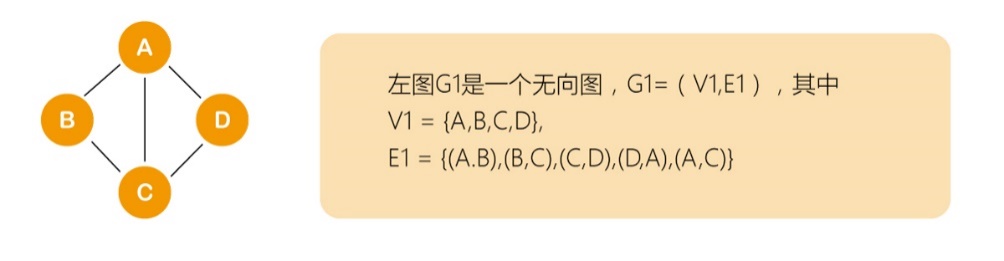
图是由顶点的有空非空集合和顶点之间的边的集合组成，通常表示为G（V,E）,其中G表示一个图，V是图G中顶点的集合，E是图G中边的集合。

在线性表中我们把数据元素叫元素，树中叫结点，在图中数据元素我们则称之为顶点。

线性表可以没有数据元素，称为空表。树中可以没有结点，叫做空树，而图结构在中国国内的教材中强调顶点集合V要有穷而非空。

线性表中，相邻的数据元素之间具有线性关系，树结构中，相邻两层的结点具有层次关系，而图结构中，任意两个顶点之间都可能有关系，顶点之间的逻辑关系用边来表示，边集可以是空的。

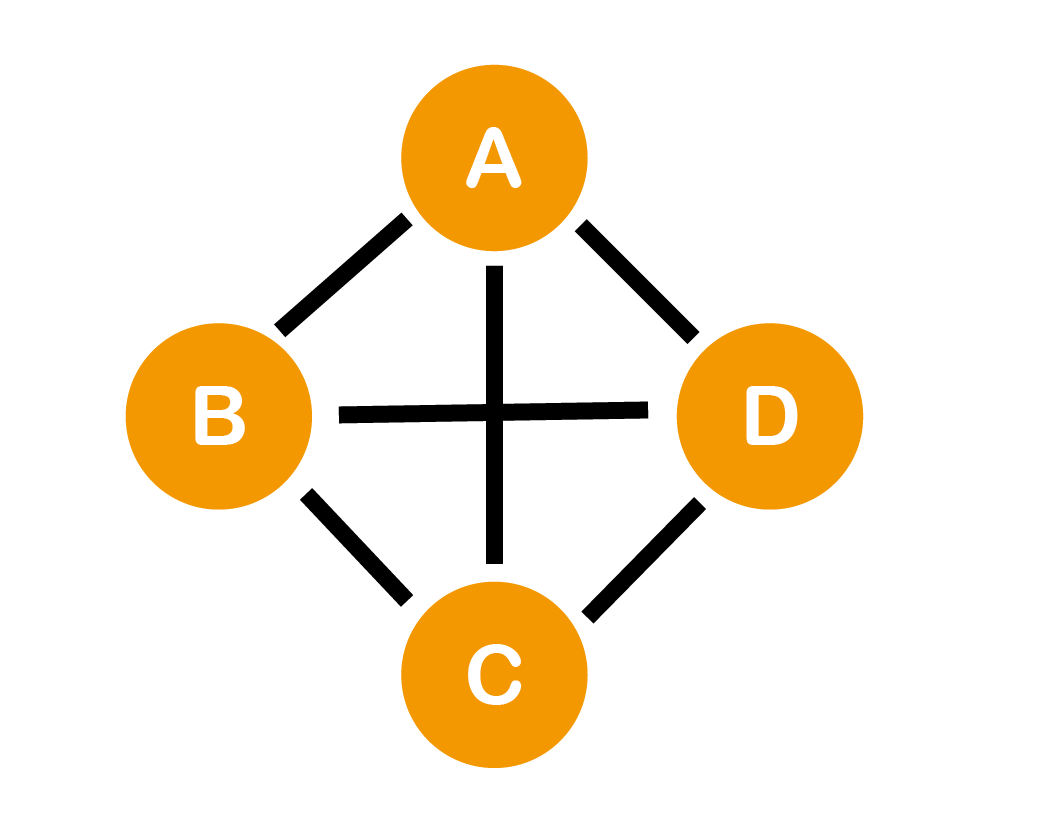
无向边：若顶点Vi到Vj之间的边没有方向，则称这条边为无向边（Edge），用无序偶（Vi,vj）来表示



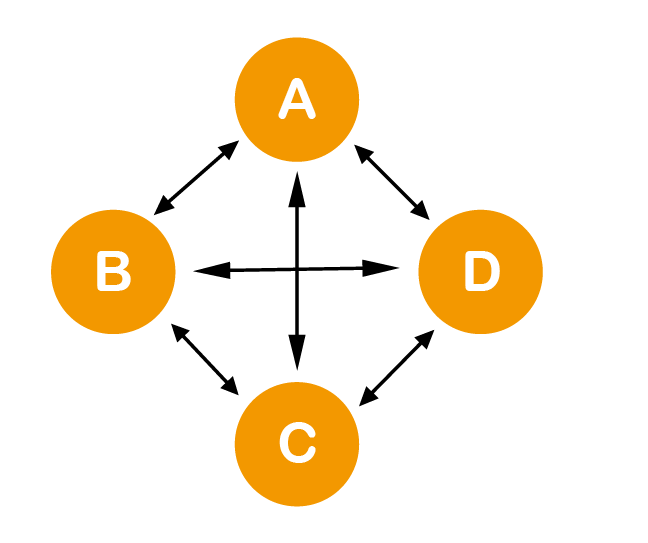
有向边：若从顶点Vi到Vj的边有方向，则称这条边为有向边，也称为弧，用有序偶<Vi,Vj>来表示，Vi称为弧尾，Vj表示弧头

简单图：在图结构中，若不存在顶点到其自身的边，且同一条边不重复出现，则称这样的图为简单图。

无向完全图：在无向图中，如果任意两个顶点之间都存在边，则称该图为无向完全图，含有n个顶点的无向完全图有（n-1）\*n/2条边。



有向完全图：在有向图中，如果任意两个顶点之间都存在互为相反的两条弧，则称该图为有向完全图，含有N个顶点的有向完全图有n\*(n-1)条边。

大

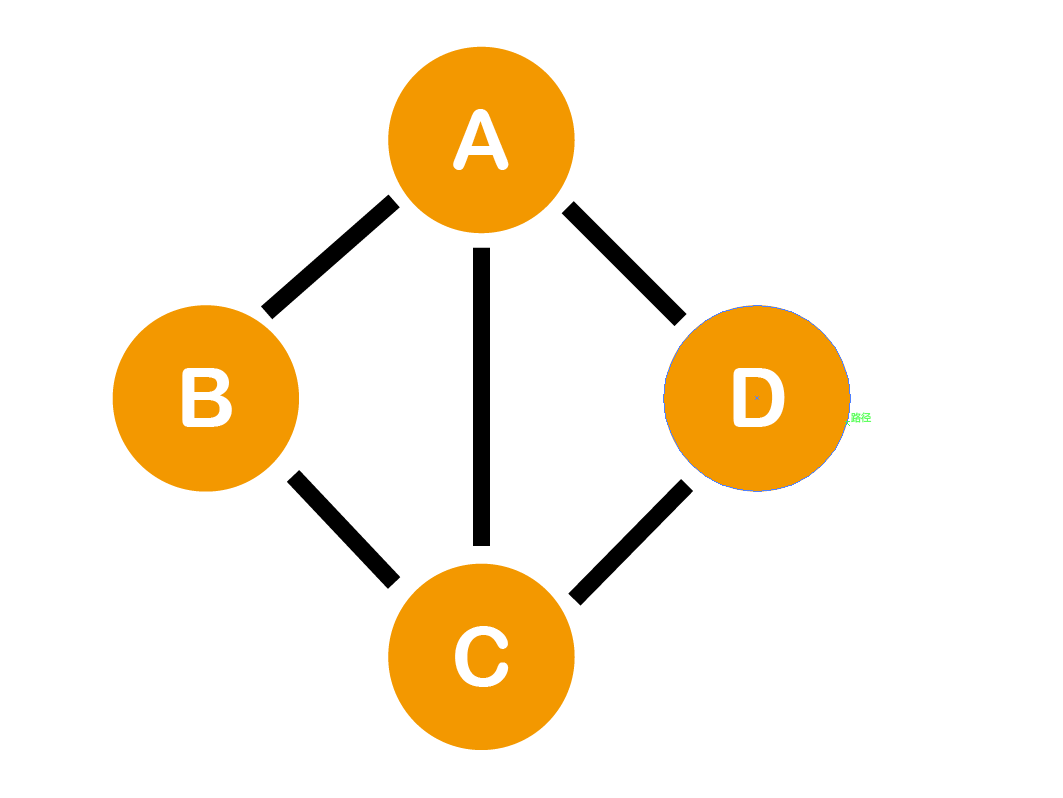
稀疏图和稠密图：这里的稀疏和稠密是模糊的概念，都是相对而言的，通常认为边或弧数小于n\*logn(n是顶点个数)的图称为稀疏图，反这则为稠密图。

有些图的边或弧带有与它相关的数字，这种与图的边或弧相关的数叫权(Weight)，带权的图通常称为网（Network）

顶点与边

对于无向图G=（V,E），如果边（V1,V2）∈E,则称顶点V1和V2互为邻接点（Adjacent），即V1和V2相邻接，边V1,V2依附于顶点V1和V2，或者说边(V1,V2)与顶点V1和V2相关联。

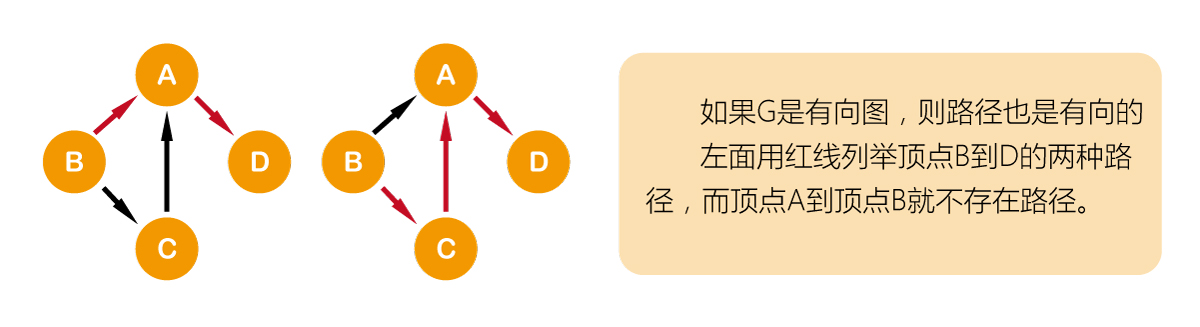
顶点V的度（Degree）是和V相关联的边的数目，记为TD(V),如下图，顶点A为B互为邻接点，边（A,B）依附于顶点A与B上，顶点A的度为3



对于有向图G=（G,E）,如果有<V1,V2>∈E,则称顶点V1邻接到顶点V2，顶点V2邻接自顶点V1.

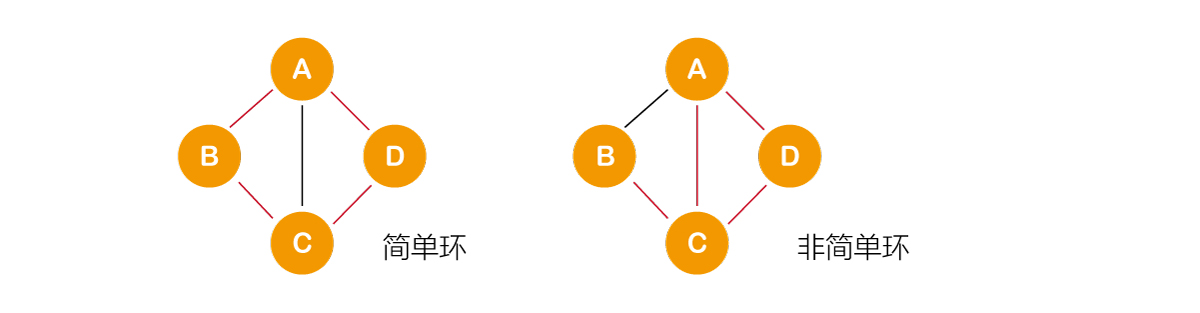
以顶点V为头的弧的的数目称为V的入度（InDegree）,记为ID(V),以V为尾的弧的数目称为V的出度（Outdegree），记为OD(V),因此顶点V的度为TD(V)=ID(V)+OD(V).

A顶点的入度是2，出度为1，所以A的度为3



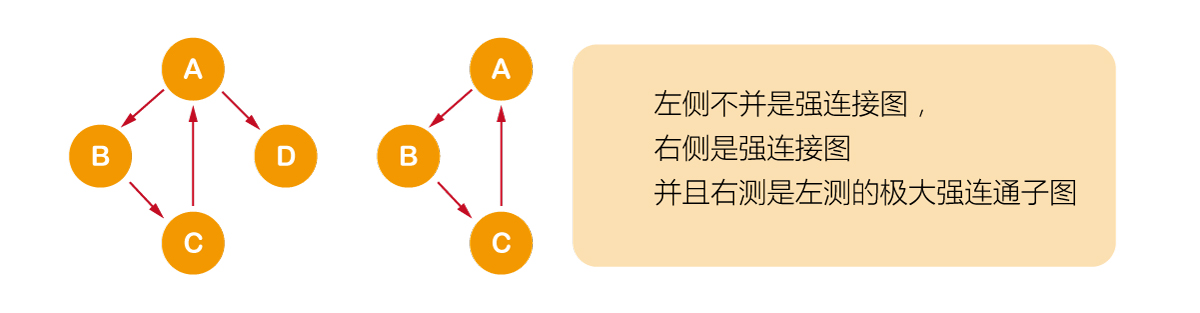
从第一个顶点到第最后一个顶点（自己）的路径称为回路或环（Cycle）。

序列中顶点不重复出现的路径称为简单路径，除了第一个顶点和最后一下顶点之外，其余顶点不重复出现的回路,称作简单回路或者简单环。



连通图：在无向图中，如果从顶点V1到顶点V2有路径，则称V1和V2都是连接图，如果对于图中任意两个项点Vi和Vj都是连通的，则称G是连通图（ConnectedGraph）。

右下图中的极大强连通子图称为有向图的强连接分量



所谓连通图的生成树是一个极小的连通子图，它含有图中全部的N个顶点，但只有足以构成一棵树的N-1条边。

如果有一有向图恰有一个顶点入度为0，其余顶点的入度均为1，则是一棵有向树。

* + 1. 图的存储结构：

邻接矩阵

邻接表

十字链表

邻接多重表

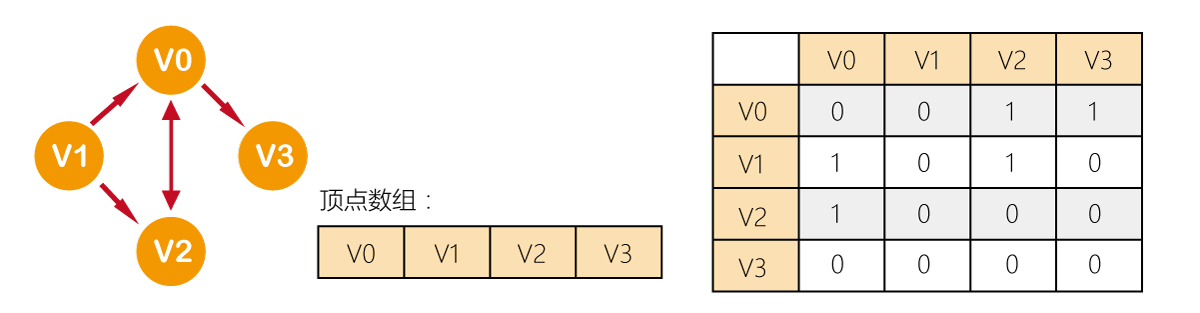
邻接矩阵可以在计算机中定义一个二维数组来存储数据。

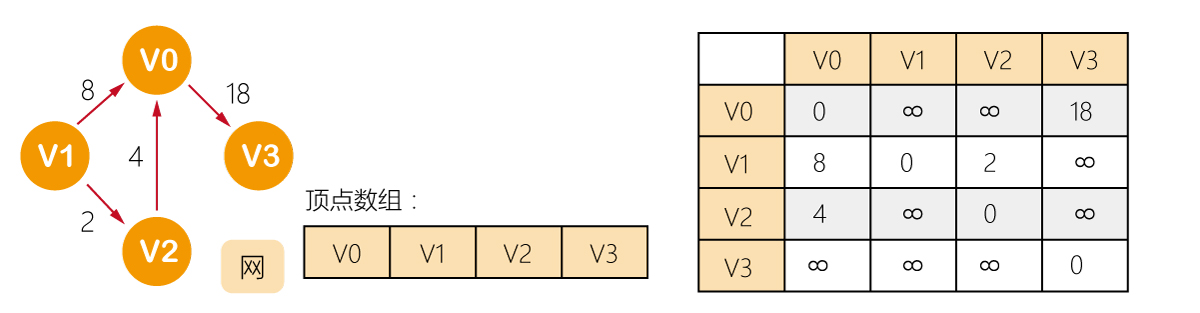
Class Map{

顶点数组;

邻接矩阵;

}

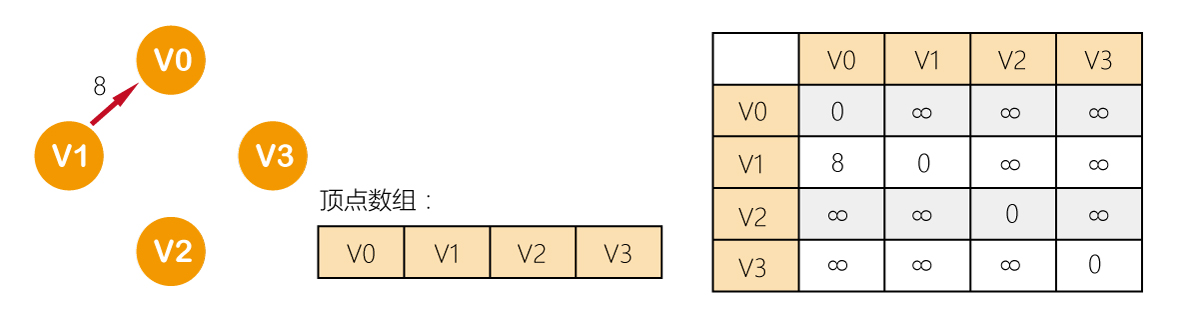




对于有向图是有讲究的，要考虑入度和出度，顶点V1的毛主席以为1，正好是第V1列的各数之和，顶点V1的出度为2，正好是第V1行的各数之和。

邻接表

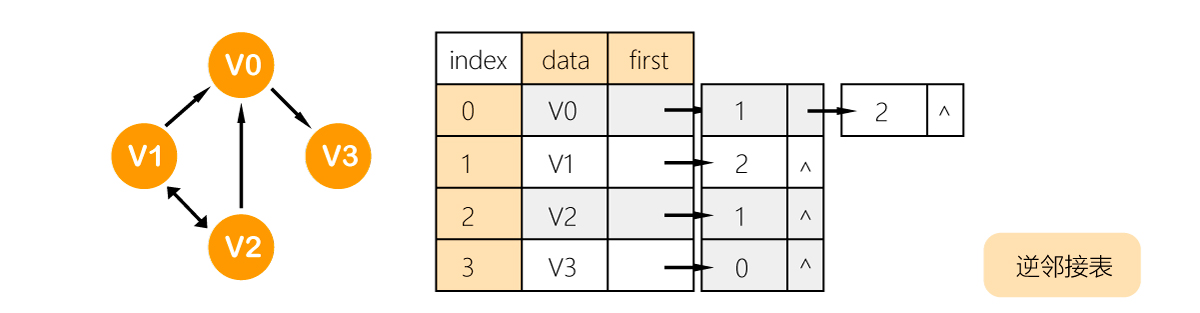
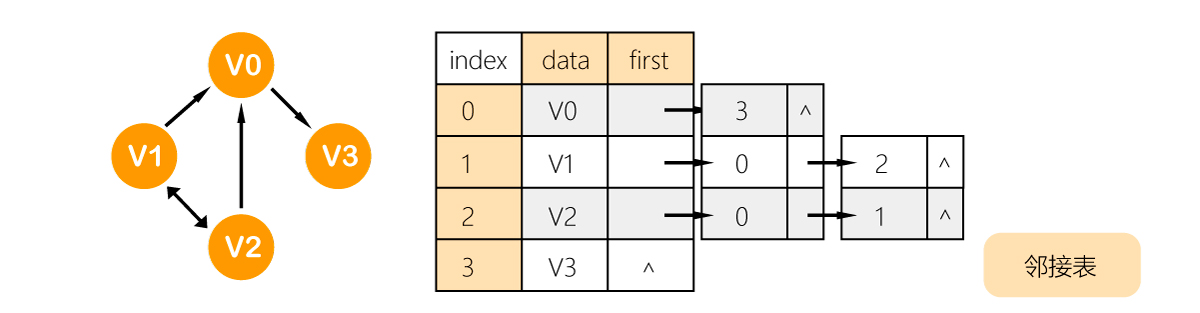
邻接矩阵在许多图的存储上是一个不错的选择，比较容易理解，但是我们也发现对于一些顶点较少的图， 这种结构会造成极大的浪费。



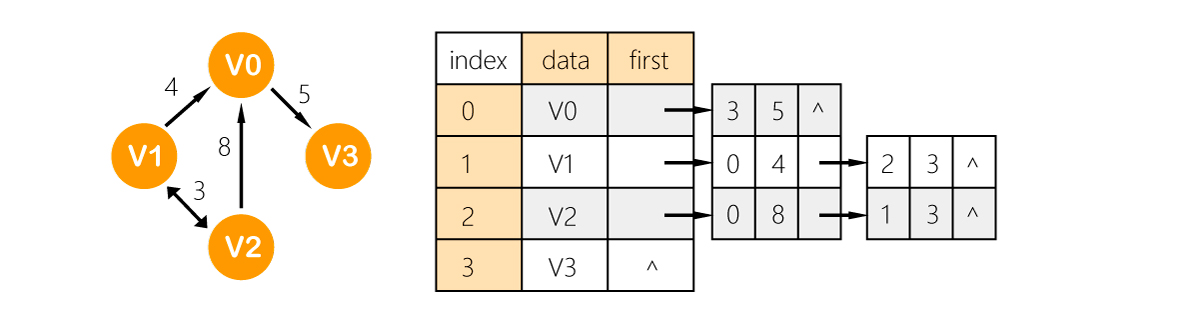
因此我们考虑另一种存储方式，把数组与链表结合起来存储，这种方式在图结构中也适用，我们把这种存储方式称为邻接表



如果是有向图，邻接表结构也是类似的，把顶点当前弧尾来建立的连接表，这样容易得到每到每个顶点的出度。有时也为了便于确定顶点的入度或心顶点为弧头的弧，我们可以建立一个有向图的逆邻接表。



对于网图，我们可以在边表结点定义中再增加一个数据域来存储权值即可。



十字链表

邻接表固然优秀，但也有不足不处，例如对有向图的处理上，有时候需要再建立一个逆邻接表。而十字链表就是把邻接表和逆邻接表结合起来而产生的。

|  |  |  |
| --- | --- | --- |
| Data | firstIn | firstOut |
| 数据 | 入边表的指针 | 出边表的指针 |

边表的结点结构

|  |  |  |  |
| --- | --- | --- | --- |
| TailVex | headVex | headLink | tailLink |
| 弧尾索引 | 弧头索引 | 头指针 | 尾指针 |

十字链表的好处就是因为把邻接表和逆邻接表整合在了一起，这样既容易找到以Vi为尾的弧，也容易找到以Vi为头的弧，因而容易求得顶点的出度和入度。

十字链表除了结构复杂一点以外，其实创建图算法的时间复杂度是和邻接表相同的，因此。在有向图的就用中，十字链表也是非常好的数据结构模型。

树的遍历我们谈到了四种方式，因为树根结点只有一个，并且所有的结点都只有一个双亲，所以不是很难以理解，但是谈到图的遍历，那复杂多了，因为它的任一顶点都可以和其余的所有顶点都邻接，因此极有可能存在重复走过某个基点或漏过了某个顶点的遍历过程，如果要避免以上的情况，就需要科学地设计遍历方案，通常有两种图的遍历方案：它们是深度优先和广度优先遍历。

有视频附件

* + 1. 马踏棋盘问题

马踏棋盘问题（又称骑士周游问题）是算法设计的经典问题之一。

题目要求：

国际象棋的棋盘为8\*8的方格棋盘，现将“马”入在任意指定的方格中，按照“马”走棋的规则将“马”进行移动。要求每个方格只能走进入一次，最终使得“马”走遍64个方格。

相关知识点：

回溯法：

指导思想很简单，就是一条路走到黑，碰壁了再回来，一般和递归可以很好的搭配使用，还有深度优先搜索（DFS）

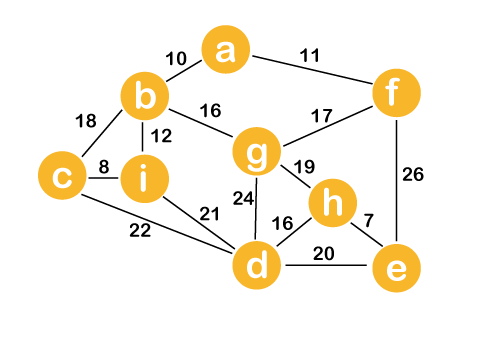
哈密尔顿路径：

图G中的哈密尔顿路径指的是经过图G中每个顶点，且只经过一次的一条轨迹。如果这条轨迹是一条闭合的路径（从起点出发不重复地遍历所有点后仍然回到起始点），那么这条路径称为哈密尔顿回路。

* + 1. 图的最小生成树：

普里姆算法：

普里姆算法的每一步都会为一棵生长中的树添加一条边， 一开始这棵树只有一个顶点，然后会向它添加V-1条边，每次总是将下一条连接树中的顶点与不在树中的顶点且权重最小的边加入树中。普里姆算法可以得到任意加权无向图的最小生成树。



步骤：

* + - 1. 把出发点变成黑色
      2. 把黑点与所有相邻结点之间的连线涂成紫色
      3. 在紫边中选择一条权值最小的边，将之与相连接的点改成黑色，将连线改成线色。
      4. 重复步骤2，并将会够成环路的紫边去除
      5. 重复步骤3……

package com.learn.tu;

import java.util.\*;

import java.util.function.BiPredicate;

import java.util.function.Function;

import java.util.function.Predicate;

import java.util.function.Supplier;

/\*\*

\* 普里姆算法

\*/

public class Prim {

private static final Integer MAX = Integer.MAX\_VALUE;

//定义点

private static final String[] value = {"a","b","c","d","e","f","g","h","i"};

//定义一个方法用来返回数组下标

private static Function<String,Integer> fun = o ->{

for(int i=0;i<value.length;i++){

if(value[i].equals(o)){return i;}

}

return -1;

};

//定义一个Map用于存储可能的路径

private static HashMap<String,Integer> map = new HashMap<>(16);

private BiPredicate<String,Integer> biPredicate = (o1, o2) -> {

String temp = new StringBuffer(o1).reverse().toString();

if(result.containsKey(o1)||result.containsKey(temp)){

return false;

}

if(delete.containsKey(o1)||delete.containsKey(temp)){

return false;

}

if(map.containsKey(o1)||map.containsKey(temp)){

return false;

}

map.put(o1,o2);

return true;

};

private Supplier<Map.Entry<String,Integer>> supplie = () ->{

Map.Entry<String,Integer> temp = null;

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

if (temp == null) {

temp = entry;

}

if (temp.getValue() > entry.getValue()) {

temp = entry;

}

}

map.remove(temp.getKey(),temp.getValue());

result.put(temp.getKey(),temp.getValue());

return temp;

};

//定义一个Map用来存储结果

private static Map<String,Integer> result = new HashMap<>(8);

//定义一个Map用来存储已经访问过但不可用的路径

private static Map<String,Integer> delete = new HashMap<>(16);

//定义一个数组用于存放黑点

private static Set<Integer> tempdata = new LinkedHashSet<>(9);

private static Predicate<Integer> addBlackPoint = (Integer o) ->{

tempdata.add(o);

for (Integer aTempdata : tempdata) {

String removekey = value[aTempdata] + "-" + value[o];

if (map.containsKey(removekey)) {

System.out.println("移出不可能路径：" + removekey);

delete.put(removekey, map.get(removekey));

map.remove(removekey);

}

}

return true;

};

//建立一个数组，定义图的数组

private int[][] data = {

{0,10,MAX,MAX,MAX,11,MAX,MAX,MAX},

{10,0,18,MAX,MAX,MAX,16,MAX,12},

{MAX,18,0,22,MAX,MAX,MAX,MAX,8},

{MAX,MAX,22,0,20,MAX,24,16,21},

{MAX,MAX,MAX,20,0,26,MAX,7,MAX},

{11,MAX,MAX,MAX,26,0,17,MAX,MAX},

{MAX,16,MAX,24,MAX,17,0,19,MAX},

{MAX,MAX,MAX,16,7,MAX,19,0,MAX},

{MAX,12,8,21,MAX,MAX,MAX,MAX,0}

};

public static void main(String[] args) {

System.out.println();

Prim prim = new Prim();

prim.start(0);

}

private void start(int start){

addBlackPoint.test(start);

if(tempdata.size() >=9){

System.out.println("结果输出：");

result.forEach((o1,o2)-> System.out.println(o1+"==>"+o2));

return;

}

int[] temp = data[start];

for(int i=0;i<temp.length;i++){

if(temp[i] != MAX && temp[i] != 0){

biPredicate.test(value[start] + "-" +value[i],temp[i]);

System.out.println("添加可能路径："+value[start] + "-" +value[i]+"权值："+temp[i]);

}

}

Map.Entry<String,Integer> entry = supplie.get();

String key = entry.getKey();

String[] data = key.split("-");

Integer index = fun.apply(data[1]);

System.out.println(index);

start(index);

}

}

克鲁斯卡尔算法

Kruskal算法能够计算任意加权无向图的最小生成树。

Kruskal算法构造最小生树的时候，也是一条边一条边的构造，但不同的是它寻找的边会连接成一片森林中的两棵树。我们从一片由V棵单顶点的树构成的森林开始并不断将两棵树合并，直到只剩下一棵树，它就是最小生成树

package com.learn.tu;

import java.util.ArrayList;

import java.util.List;

import java.util.function.IntFunction;

public class Kruskal {

private static final String[] value = {"a","b","c","d","e","f","g","h","i"};

private static final Integer MAX = Integer.MAX\_VALUE;

//设置一个表用维护路径

List<int[]> list = new ArrayList<>();

//设置一个数组用于避免回路

int[] huilu = new int[9];

IntFunction<Integer> ifun = o->{

while(huilu[o] >0){

o = huilu[o];

}

return o;

};

private int[][] data = {

{0,10,MAX,MAX,MAX,11,MAX,MAX,MAX},

{10,0,18,MAX,MAX,MAX,16,MAX,12},

{MAX,18,0,22,MAX,MAX,MAX,MAX,8},

{MAX,MAX,22,0,20,MAX,24,16,21},

{MAX,MAX,MAX,20,0,26,MAX,7,MAX},

{11,MAX,MAX,MAX,26,0,17,MAX,MAX},

{MAX,16,MAX,24,MAX,17,0,19,MAX},

{MAX,MAX,MAX,16,7,MAX,19,0,MAX},

{MAX,12,8,21,MAX,MAX,MAX,MAX,0}

};

public static void main(String[] args) {

new Kruskal().start();

}

private Kruskal(){

for(int i=0;i<data.length;i++){

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

if(data[i][j] != MAX && data[i][j] !=0){

int[] arr = new int[3];

arr[1] = i;

arr[0] = j;

arr[2] = data[i][j];

list.add(arr);

}

}

}

list.sort((int[] o1,int[] o2)->{

if(o1[2] > o2[2]){

return 1;

}

if(o1[2] < o2[2]){

return -1;

}

return 0;

});

for(int[] in : list){

System.out.println("起："+in[0]+"终："+in[1]+"权："+in[2]);

}

}

private void start(){

for(int[] ar : list){

int start = ar[0];

int end = ar[1];

System.out.print(value[start] +"-"+value[end] +"正在测试：");

start = ifun.apply(start);

end = ifun.apply(end);

if(start != end){

System.out.println("测试结果：可用");

huilu[start] = end;

}else{

System.out.println("测试结果：不可用");

}

}

}

}

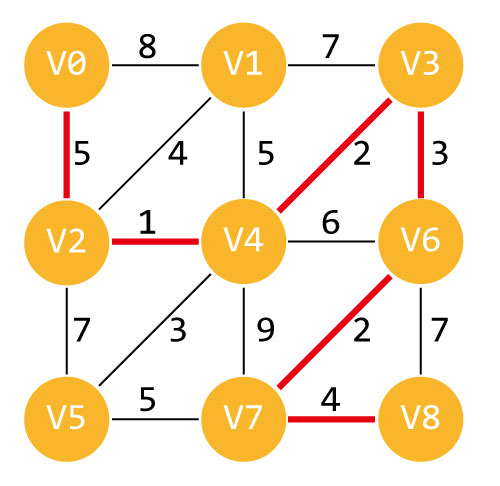
* + 1. 最短路径问题

在网图和非网图中，最短路径的含义是不同的。

网图是两顶点经过的边上权值之和最少的路径

非网图是丙顶点之间经过的边数最少的路径，非网图的权值都为1

迪杰斯特拉算法：



package com.learn.tu;

public class Dijkstra {

private static final Integer MAX = 50000;

//图

private static final int[][] graph = {

{0,8,5,MAX,MAX,MAX,MAX,MAX,MAX},

{8,0,3,7,5,MAX,MAX,MAX,MAX},

{5,3,0,MAX,1,7,MAX,MAX,MAX},

{MAX,7,MAX,0,2,MAX,3,MAX,MAX},

{MAX,5,1,2,0,3,6,9,MAX},

{MAX,MAX,7,MAX,3,0,MAX,5,MAX},

{MAX,MAX,MAX,3,6,MAX,0,2,7},

{MAX,MAX,MAX,MAX,9,5,2,0,4},

{MAX,MAX,MAX,MAX,MAX,MAX,7,4,0},

};

//定义一个数组用来存放查找标志位

private int[] flag;

//定义一个数组用来存放最短路径的前驱

private int[] P;

//定义一个数组用来存放路径的权值和

private int[] D;

public static void main(String[] args) {

new Dijkstra(0).start();

}

//初始化各个数据

private Dijkstra(int start){

flag = new int[graph.length];

P = new int[graph.length];

D = new int[graph.length];

for(int i=0;i<graph.length;i++){

flag[i] = 0;

D[i] = graph[start][i];

P[i] = 0;

}

flag[start] = 1;

printarr("D",D);

printarr("P",P);

printarr("falg",flag);

}

private static void printarr(String name,int[] arr){

System.out.print(name+":=");

for(int a:arr){

System.out.print(a+",");

}

System.out.println();

}

private void start(){

int main,k=0;

//将所有的点拿出来遍历，第i点

for(int i= 0;i<graph.length;i++){

main = MAX;

//从起始点向周围查找一条最短的路径，记录权值到main,记录前驱到k

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

if(flag[j]==0 && D[j] < main){

k = j;

main = D[j];

}

}

//System.out.println("k:"+k + "main:"+main);

//标记从起始点到k点是可达的，并且已经找到一路路径

flag[k] = 1;

//遍历所有点，第j点，向周围点摸索

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

//如果从此点到周围的点比之前确定的方案优秀，那么更改以前的方案

if(flag[j]==0 && main +graph[k][j]< D[j]){

D[j] = main + graph[k][j];

P[j] = k;

}

}

}

printarr("D",D);

printarr("P",P);

printarr("falg",flag);

}

}

弗洛伊德算法

弗洛伊德算法的时间复杂度O(N^3)，而迪杰斯特拉算法的时间复杂度为O(N^2),那么我们还有必要学习吗？

因为迪杰斯特拉算法求的是一个顶点到所有顶点的最短路径，而弗洛伊德算法求的是所有顶点到所有顶点的最短路径。

弗洛伊德算法非常简洁

package com.learn.tu;

public class Floyd {

private static final Integer MAX = 50000;

//图

private static final int[][] GRAPH = {

{0, 8, 5, MAX, MAX, MAX, MAX, MAX, MAX},

{8, 0, 3, 7, 5, MAX, MAX, MAX, MAX},

{5, 3, 0, MAX, 1, 7, MAX, MAX, MAX},

{MAX, 7, MAX, 0, 2, MAX, 3, MAX, MAX},

{MAX, 5, 1, 2, 0, 3, 6, 9, MAX},

{MAX, MAX, 7, MAX, 3, 0, MAX, 5, MAX},

{MAX, MAX, MAX, 3, 6, MAX, 0, 2, 7},

{MAX, MAX, MAX, MAX, 9, 5, 2, 0, 4},

{MAX, MAX, MAX, MAX, MAX, MAX, 7, 4, 0},

};

//定义一个数组用来存放权值和

private static int[][] D;

//定义一个数组用来存在前驱

private static int[][] P;

public static void main(String[] args){

new Floyd().start();

}

Floyd(){

this.init();

}

private void init() {

int length = GRAPH.length;

D = new int[length][length];

P = new int[length][length];

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

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

D[i][j] = GRAPH[i][j];

P[i][j] = j;

}

}

print(D,"D");

print(P,"P");

}

private void print(int[][] arr ,String name){

int i=0;

for (int[] ints : arr) {

StringBuilder sbuilder = new StringBuilder(name +"["+(++i)+"] ");

for (int anInt : ints) {

sbuilder.append(anInt).append(",");

}

System.out.println(sbuilder);

}

}

public void start(){

int length = GRAPH.length;

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

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

for (int k = 0; k < length; k++) {

if(D[j][k] > D[j][i] + D[i][k]){

D[j][k] = D[j][i] + D[i][k];

P[j][k] = P[j][i];

}

}

}

}

print(D,"D");

print(P,"P");

jie(0,8);

}

//解析起始点start，解析终点end

private void jie(int start,int end){

System.out.format("解析从%d到%d点\n",start,end);

System.out.format("最短权%d\n",D[start][end]);

StringBuilder builder = new StringBuilder("最短路径：");

int temp = start;

while (temp != end) {

temp = P[temp][end];

builder.append(temp+"->");

}

System.out.println(builder.delete(builder.length()-2,builder.length()));

}

}

* + 1. 拓扑排序

一个无环的有向图称为无环图（Directed Acyclic Grap）,DAG图

所有工程或者某种流程都可以分为若干个小的工程或者阶段，我们称这些小的工程或者阶段为“活动”。在一个表示工程的有向图中，用顶点表示活动，用弧表示活动之间的优先关系，这样的有向图为顶点表示活动的网，我们称之为AOV图（Active On Vertex Network）.AOV网中的弧表示活动之间存在的某种制约关系，所以AOV网不能存在回路。

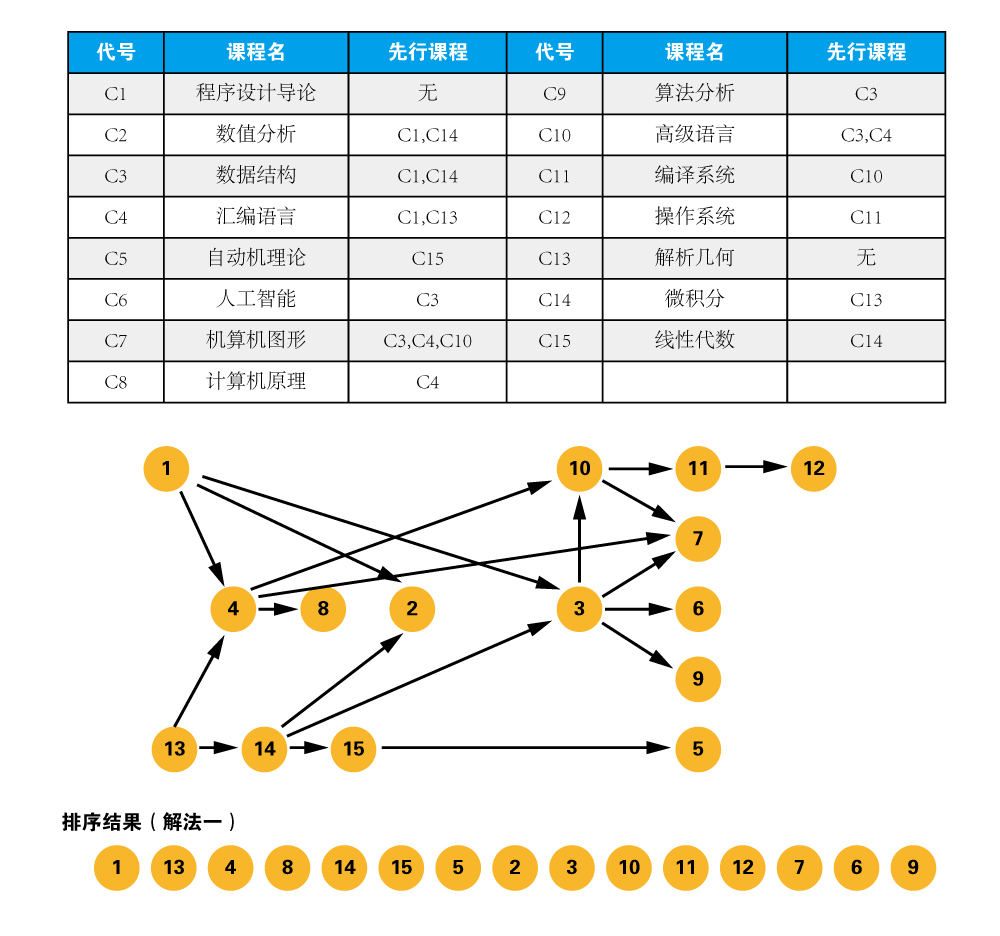
拓扑序列：设G=(V,E)是一个具有n个顶点的有向图，V中的顶点序列V1,V2,……满足若从顶点Vi到Vj有一条路径，则在顶点序列中顶点Vi必在顶点Vj之前。我们称这样的顶点序列为一个拓扑序列。

对AOV网进行拓扑排序的方法和步骤如下：

从AOV网中选择一个没有前驱的顶点（顶点的入度为0）并且输出它

从网中删除该顶点，并且删除去从该顶点出发的全部有向边

重复上述两步，直到不再存在没有前驱的顶点为止。



package com.learn.tu;

import java.util.Arrays;

import java.util.function.IntPredicate;

//拓扑排序

public class TP {

public static final int WEI = 1;

public static final int MAX = 0;

private static final int[][] GRAPH = {

{MAX,WEI,WEI,WEI,MAX,MAX,MAX,MAX,MAX,MAX,MAX,MAX,MAX,MAX,MAX},

{MAX,MAX,MAX,MAX,MAX,MAX,MAX,MAX,MAX,MAX,MAX,MAX,MAX,MAX,MAX},

{MAX,MAX,MAX,MAX,MAX,WEI,WEI,MAX,WEI,WEI,MAX,MAX,MAX,MAX,MAX},

{MAX,MAX,MAX,MAX,MAX,MAX,WEI,WEI,MAX,WEI,MAX,MAX,MAX,MAX,MAX},

{MAX,MAX,MAX,MAX,MAX,MAX,MAX,MAX,MAX,MAX,MAX,MAX,MAX,MAX,MAX},

{MAX,MAX,MAX,MAX,MAX,MAX,MAX,MAX,MAX,MAX,MAX,MAX,MAX,MAX,MAX},

{MAX,MAX,MAX,MAX,MAX,MAX,MAX,MAX,MAX,MAX,MAX,MAX,MAX,MAX,MAX},

{MAX,MAX,MAX,MAX,MAX,MAX,MAX,MAX,MAX,MAX,MAX,MAX,MAX,MAX,MAX},

{MAX,MAX,MAX,MAX,MAX,MAX,MAX,MAX,MAX,MAX,MAX,MAX,MAX,MAX,MAX},

{MAX,MAX,MAX,MAX,MAX,MAX,WEI,MAX,MAX,MAX,WEI,MAX,MAX,MAX,MAX},

{MAX,MAX,MAX,MAX,MAX,MAX,MAX,MAX,MAX,MAX,MAX,WEI,MAX,MAX,MAX},

{MAX,MAX,MAX,MAX,MAX,MAX,MAX,MAX,MAX,MAX,MAX,MAX,MAX,MAX,MAX},

{MAX,MAX,MAX,MAX,MAX,MAX,MAX,MAX,MAX,MAX,MAX,MAX,MAX,WEI,MAX},

{MAX,WEI,WEI,MAX,MAX,MAX,MAX,MAX,MAX,MAX,MAX,MAX,MAX,MAX,WEI},

{MAX,MAX,MAX,MAX,WEI,MAX,MAX,MAX,MAX,MAX,MAX,MAX,MAX,MAX,MAX},

};

private static int[] result;

IntPredicate ipre = o->{

for(int j:result){

if(j==o){return true;}

}

return false;

};

private static int[] temp;

public static void main(String[] args){

new TP().start();

}

TP(){

int length = GRAPH.length;

result = new int[length];

temp = new int[length];

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

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

temp[j] += GRAPH[i][j];

}

result[i] = -1;

}

}

void start(){

int k = 0,ind = 0;

int length = GRAPH.length;

while(ind<length){

//从AOV网中选择一个没有前驱的顶点（顶点的入度为0）并且输出它

if(temp[k]==0 && !ipre.test(k)){

result[ind++] = k;

//从网中删除该顶点，并且删除去从该顶点出发的全部有向边

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

temp[j] -= GRAPH[k][j];

}

k=0;

}else {

k++;

}

}

print(result,"RESULT");

}

private void print(int[] arr,String name){

System.out.print(name+":{");

for (int i : arr) {

System.out.print(i+1+",");

}

System.out.println("}");

}

}

* 1. 查找

对于静态查找来说，我们可以用线表结构组织数据，这样可以使用顺序查找算法，如果我们再对关键字进行排序，则可以使用折半查找算法或斐波那契算法等来提高查找效率。

对于动态查找来说，我们可以考虑使用二叉排序树的查找技术，另外我们还可以使用散列表结构来解决这些查找问题。

* + 1. 插值查找

插值查找法是由折半查找法优化而来，它适合于数据分布均匀的查找，它的核心思想是不于以折半查找盲目的0.5去查找，而是优化比例。就比如让我们在字典中找以a开头的显然是中间折半是不合适的。

* + 1. 斐波那契查找



* + 1. 线性索引

折半查找必需是建立在数据排序的基础之上，而经常在数据量较大或者更新量大的情况下，排序是不可能实时实现的，从而就必须使用索引查找。

稠密索引

适用于数据量不大的索引，一对一提取数据再对索引表进行排序，非常耗存储空间

分块索引

分块索引要求块之间有序，块内不要求。类似于数据库大表的分表

倒排索引

适用于根据值来查找记录，所以称倒排索引

A:I love l

B:I am lerning in l

|  |  |
| --- | --- |
| 关键字 | 记录 |
| I | A,B |
| Love | A |
| Am | B |
| Learning | B |
| …… | |

* + 1. 二叉排序树

对于普通的线性数据结构，在不排序的情况下插入和删除元素时虽然效率较高，但查找必须全部遍历，而排序时，插入和删除的效率又非常的低，有什么数据结构可以解决这个问题呢？

二叉排序数又数又称为二叉查找树，它或者是一棵空树，或者是具有下列性质的二叉数：

若它的左子树不为空，则左子树上所有结点的值均小于它的根结构的值;

若它的右子树不为空，则右子树上所有结点的值均大小它的根结构的值;

它的左、右树也分别为二叉排序树（递归）

二叉排序树的删除是比较麻烦的，可以分为三种情况考虑：

* + - 1. 如果是叶子节点
      2. 如果只有左子树或右子树，直接用子树顶替要删除的节点即可
      3. 如果既有左子树，又有右子树，需要选择直接前驱或直接后继来顶替。

package com.learn.shu;

import java.security.InvalidParameterException;

public class BinaryTree {

BinaryTree parent,lnode,rnode;

int index;

Object obj;

BinaryTree(int index,Object obj){

this.index = index;

this.obj = obj;

this.rnode = null;

this.lnode = null;

this.parent = null;

}

boolean insertNode(int index,Object obj){

if(this.index ==index){

System.out.println("索引已经存在，插入失败！");

return false;

}else if(this.index > index){

if(this.lnode == null){

BinaryTree binaryTree = new BinaryTree(index,obj);

binaryTree.parent = this;

this.lnode = binaryTree;

System.out.println("索引["+index+"]添加到["+this.index+"]的lnode成功！");

return true;

}else{

return this.lnode.insertNode(index,obj);

}

}else if(this.index < index){

if(this.rnode == null){

BinaryTree binaryTree = new BinaryTree(index,obj);

binaryTree.parent = this;

this.rnode = binaryTree;

System.out.println("索引["+index+"]添加到["+this.index+"]的rnode成功！");

return true;

}else{

return this.rnode.insertNode(index,obj);

}

}

return false;

}

public boolean deleteNode(int index){

BinaryTree deleteNode = selectNode(index);

//如果结点不存在

if(deleteNode == null){

return false;

//结点存在

}else{

//如果是叶子节点

if(deleteNode.rnode == null && deleteNode.lnode == null){

if(deleteNode.parent.lnode == deleteNode){

deleteNode.parent.lnode = null;

}else if(deleteNode.parent.rnode == deleteNode){

deleteNode.parent.rnode = null;

}

}

//如果只有左子树

else if(deleteNode.rnode == null && deleteNode.lnode != null){

deleteNode.lnode.parent = deleteNode.parent;

deleteNode.parent.lnode = deleteNode.lnode;

}

//如果只有右子树

else if(deleteNode.lnode == null && deleteNode.rnode != null){

deleteNode.rnode.parent = deleteNode.parent;

deleteNode.parent.rnode = deleteNode.rnode;

}

//如果既有左子树，又有右子树

else if(deleteNode.lnode != null && deleteNode.rnode != null){

BinaryTree temp = deleteNode.rnode;

while(temp.lnode != null){

temp = temp.lnode;

}

deleteNode.index = temp.index;

deleteNode.obj = temp.obj;

if(temp == deleteNode.rnode){

deleteNode.rnode = temp.rnode;

}else{

temp.parent.lnode = temp.rnode;

}

}

}

System.out.println("删除结点["+index+"]成功！");

return true;

}

BinaryTree selectNode(int index){

if(index == this.index){

return this;

}

if(index > this.index && this.rnode != null){

return this.rnode.selectNode(index);

}

if(index < this.index && this.lnode != null){

return this.lnode.selectNode(index);

}

System.out.println("没有找到索引结点["+index+"]");

return null;

}

/\*\*

\* 遍历二叉树

\* @param fun

\* 参数1 为前序遍历

\* 参数2 为中序遍历

\* 参数3 为后序遍历

\*/

void printNode(int fun){

switch (fun){

case 1:

printQian();

System.out.println();

break;

case 2:

printZhong();

System.out.println();

break;

case 3:

printHou();

System.out.println();

break;

default:

throw new InvalidParameterException("参数异常，[1,2,3]");

}

}

private void printHou() {

if(this.rnode != null){

this.rnode.printHou();

}

System.out.print(this.index+",");

if(this.lnode != null){

this.lnode.printHou();;

}

}

private void printZhong() {

System.out.print(this.index+",");

if(this.lnode != null){

this.lnode.printZhong();

}

if(this.rnode!= null){

this.rnode.printZhong();

}

}

void printQian(){

if(this.lnode != null){

this.lnode.printQian();

}

System.out.print(this.index+",");

if(this.rnode != null){

this.rnode.printQian();

}

}

public static void main(String[] args){

BinaryTree root = new BinaryTree(3,3);

root.insertNode(4,4);

root.insertNode(6,6);

root.insertNode(5,5);

root.insertNode(11,11);

root.insertNode(8,8);

root.insertNode(10,10);

root.insertNode(9,9);

root.insertNode(100,100);

root.insertNode(99,99);

root.printNode(1);

root.deleteNode(6);

root.printNode(1);

}

}

* + 1. 二衡二叉排序树

我们将二叉树上的结点上的左子树的深度的值减去右子树的深度的值称为平衡因子FB,平衡二叉树就是一棵二叉树上所有结点的平衡因子的绝对值小于等于1的树。

package com.learn.shu;

import java.security.InvalidParameterException;

public class AVL {

int index;

Object obj;

AVL parent,lnode,rnode;

public AVL(int index,Object obj){

this.index = index;

this.obj = obj;

}

public static int getHeight(AVL avl){

if(avl == null){

return 0;

}

int left = getHeight(avl.lnode);

int right = getHeight(avl.rnode);

return 1+Math.max(left,right);

}

//进行单右旋操作

private void ll\_rotate(){

if(this.parent != null && this.parent.lnode == this){

this.parent.lnode = this.lnode;

}else if(this.parent != null && this.parent.rnode == this){

this.parent.rnode = this.lnode;

}

this.lnode.parent = this.parent;

AVL temp = this.lnode.rnode;

if(temp != null){

temp.parent = this;

}

this.lnode.rnode = this;

this.parent = this.lnode;

this.lnode = temp;

}

//进行单左旋操作

private void rr\_rotate(){

//插入的是this右子节点的右子树

if (this.parent != null && this.parent.lnode == this) {

this.parent.lnode = this.rnode;

} else if (this.parent != null && this.parent.rnode == this) {

this.parent.rnode = this.rnode;

}

this.rnode.parent = this.parent;

AVL temp = this.rnode.lnode;

if(temp != null){

temp.parent = this;

}

this.rnode.lnode = this;

this.parent = this.rnode;

this.rnode = temp;

}

public void show(){

show(0);

}

private void show(int num){

StringBuilder sBuilder = new StringBuilder();

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

sBuilder.append("|--");

}

if(this.parent != null && this.parent.lnode == this){

sBuilder.append("l");

}

if(this.parent != null && this.parent.rnode == this){

sBuilder.append("r");

}

sBuilder.append(this.index);

System.out.println(sBuilder.toString());

if(this.lnode != null){

this.lnode.show(1+num);

}

if(this.rnode!= null){

this.rnode.show(num+1);

}

}

public void printNode(int fun){

if (fun == 1) {

printQian();

System.out.println();

} else if (fun == 2) {

printZhong();

System.out.println();

} else if (fun == 3) {

printHou();

System.out.println();

} else {

throw new InvalidParameterException("参数异常，[1,2,3]");

}

}

private void printHou() {

if(this.rnode != null){

this.rnode.printHou();

}

System.out.print(this.index+",");

if(this.lnode != null){

this.lnode.printHou();;

}

}

private void printZhong() {

System.out.print(this.index+",");

if(this.lnode != null){

this.lnode.printZhong();

}

if(this.rnode!= null){

this.rnode.printZhong();

}

}

void printQian(){

if(this.lnode != null){

this.lnode.printQian();

}

System.out.print(this.index+",");

if(this.rnode != null){

this.rnode.printQian();

}

}

public AVL getRoot(){

if(this.parent != null){

return this.parent.getRoot();

}else{

return this;

}

}

public boolean insertNode(int index,Object obj) {

if (this.index == index) {

System.out.println("[索引" + index + "]已经存在，无法插入！");

return false;

}

if (index < this.index) {

if (this.lnode == null) {

AVL avl = new AVL(index, obj);

avl.parent = this;

this.lnode = avl;

System.out.println("索引[" + index + "]添加到[" + this.index + "]的lnode成功！");

return true;

} else {

//如果插入成功，检查树是否平衡

if (this.lnode.insertNode(index, obj)) {

//如果等于2,则表示失去了平衡

if (AVL.getHeight(lnode) - AVL.getHeight(rnode) >= 2) {

System.out.println("树[" + this.index + "]已经失去了平衡");

//插入的是this左子节点的左子树

if (index < this.lnode.index) {

ll\_rotate();

//插入的是this左子节点的右子树

} else {

//先对左右子树进行单左旋操作

lnode.rr\_rotate();

//对自身结点进行单右旋操作

ll\_rotate();

}

}

return true;

} else {

return false;

}

}

}

if (index > this.index) {

if (this.rnode == null) {

AVL avl = new AVL(index, obj);

avl.parent = this;

this.rnode = avl;

System.out.println("索引[" + index + "]添加到[" + this.index + "]的rnode成功！");

return true;

} else {

if (this.rnode.insertNode(index, obj)) {

if (AVL.getHeight(rnode) - AVL.getHeight(lnode) >= 2) {

System.out.println("树[" + this.index + "]已经失去了平衡");

//插入的是this右子节点的左子树

if (index < this.rnode.index) {

rnode.ll\_rotate();

rr\_rotate();

} else {

rr\_rotate();

}

}

return true;

} else {

return false;

}

}

}

return false;

}

public boolean deleteNode(int index){

if(index == this.index){

//找到了

//如果是叶子节点

if(rnode == null && lnode == null){

if(parent.lnode == this){

parent.lnode = null;

}else if(parent.rnode == this){

parent.rnode = null;

}

}

//如果只有左子树

else if(rnode == null && lnode != null){

lnode.parent = this.parent;

parent.lnode = lnode;

}

//如果只有右子树

else if(lnode == null && rnode != null){

rnode.parent = this.parent;

parent.rnode = rnode;

}

//如果既有左子树，又有右子树

else if(lnode != null && rnode != null){

//左子树高度大于右子树高度，以前驱顶点代替删除顶点

if(getHeight(lnode)>getHeight(rnode)){

AVL temp = lnode;

while(temp.rnode != null){

temp = temp.rnode;

}

this.index = temp.index;

this.obj = temp.obj;

if(temp == lnode){

lnode = temp.lnode;

}else{

temp.parent.rnode = temp.lnode;

}

}

//左子树高度小于右子树高度，以后继顶点代替删除顶点

else{

AVL temp = rnode;

while(temp.lnode != null){

temp = temp.lnode;

}

this.index = temp.index;

obj = temp.obj;

if(temp == rnode){

rnode = temp.rnode;

}else{

temp.parent.lnode = temp.rnode;

}

}

}

return true;

}

if(index > this.index && this.rnode != null){

//在右子树上删除

if(this.rnode.deleteNode(index)){

//如果左子树高于右子树

if(getHeight(lnode)-getHeight(rnode)>=2){

System.out.println("树["+this.index+"]失去了平衡");

if(getHeight(lnode.rnode)>getHeight(lnode.lnode)){

//需要进行LR操作

lnode.rr\_rotate();

ll\_rotate();

}else if(getHeight(lnode.lnode)>getHeight(lnode.rnode)){

//需要进行单右旋调整

ll\_rotate();

}

}

return true;

}else{

return false;

}

}

if(index < this.index && this.lnode != null){

if(this.lnode.deleteNode(index)){

if (getHeight(rnode) - getHeight(lnode) >= 2) {

System.out.println("树["+this.index+"]失去了平衡");

if(getHeight(rnode.rnode)>getHeight(rnode.lnode)){

//需要进行单左旋调整

rr\_rotate();

}else if(getHeight(rnode.lnode)>getHeight(rnode.rnode)){

//需要进行RL调整

rnode.ll\_rotate();

rr\_rotate();

}

}

return true;

}else{

return false;

}

}

System.out.println("没有找到索引结点["+index+"]");

return false;

}

public AVL selectNode(int index){

if(index == this.index){

return this;

}

if(index > this.index && this.rnode != null){

return this.rnode.selectNode(index);

}

if(index < this.index && this.lnode != null){

return this.lnode.selectNode(index);

}

System.out.println("没有找到索引结点["+index+"]");

return null;

}

public static void main(String[] args){

testdeleteNode();

}

private static void testdeleteNode(){

AVL root = new AVL(20,20);

root.insertNode(10,10);

root.insertNode(30,30);

root.insertNode(25,25);

root.insertNode(15,15);

root.insertNode(40,40);

root.insertNode(24,24);

//root.insertNode(27,27);

root.deleteNode(15);

root = root.getRoot();

root.show();

}

private static void testinsertNode() {

AVL root = new AVL(3,3);

root.getRoot().insertNode(2,2);

root.getRoot().insertNode(1,1);

root.getRoot().insertNode(4,4);

root.getRoot().insertNode(7,7);

root.getRoot().insertNode(8,8);

root = root.getRoot();

root.getRoot().insertNode(9,9);

root.getRoot().insertNode(12,12);

root.getRoot().insertNode(11,11);

root.getRoot().insertNode(10,10);

root.getRoot().insertNode(10,10);

root.getRoot().insertNode(5,5);

root.getRoot().insertNode(13,13);

root = root.getRoot();

root.printNode(1);

root.show(0);

}

}

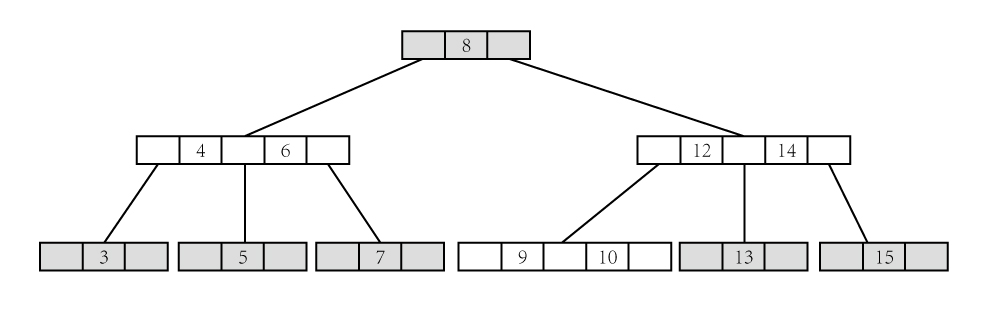
* + 1. 2-3树

二叉树的每个结点只能存储单个数据，朝下只有两个分支，这样在数据量大的时候会导致层的增加，从而导致查询效率降低。B树就可以解决这引问题，2-3树就是B树的一种。

一棵2-3查找树或为一棵空树，或由以下结点组成：

2-结点：含有一个键（及其对应的值）和两条链接，左链接指向2-3树中的键都小于该结点，右键链接指向的2-3树中的键都大于该结点。

3-结点：含有两个键和三条链接。左链接指向的2-3树中的键都小于该结点，中链接指向的2-3树中的键都位于该结点的两个键之间在，右链接指向的2-3树中的键都大于该结点。



package com.learn.shu;

public class TTTree {

TTTree parent;

Integer key1;

Integer key2;

TTTree left;

TTTree cent;

TTTree righ;

//判断是二结点

private boolean testTT(){

return this.key1 != null && this.key2 == null;

}

//判断是三结点

private boolean testTTT(){

return this.key1 != null && this.key2 != null && key1 < key2;

}

private TTTree(Integer key1,TTTree parent){

this.key1 = key1;

this.key2 = null;

this.left = null;

this.cent = null;

this.righ = null;

this.parent = parent;

}

private TTTree(Integer key1,Integer key2,TTTree parent){

if(key1 < key2){

this.key1 = key1;

this.key2 = key2;

this.left = null;

this.cent = null;

this.righ = null;

this.parent = parent;

}else{

System.out.println("构造参数key1必须小于key2");

}

}

private TTTree getRoot(){

if(this.parent != null){

return this.parent.getRoot();

}else{

return this;

}

}

private TTTree selectNode(Integer key) {

if(this.testTT()){

if(this.key1 == key){

return this;

}else if(this.key1 > key){

return this.left.selectNode(key);

}else{

return this.righ.selectNode(key);

}

}else if(this.testTTT()){

if(this.key1 == key || this.key2 == key){

return this;

}else if(this.key1 > key){

return this.left.selectNode(key);

}else if(this.key2 < key){

return this.righ.selectNode(key);

}else{

return this.cent.selectNode(key);

}

}else {

System.out.println("结点异常，并非合法结点");

return null;

}

}

private boolean insertNode(Integer key){

if(key == key1 || key == key2){

System.out.println("已经存在，无法插入");

return false;

}

if(this.testTT()){

if(key < key1){

if(left != null){

return left.insertNode(key);

}else{

//就是这个点插入，将二结点变为三结点

key2 = this.key1;

key1 = key;

return true;

}

}else{

if(righ != null){

return righ.insertNode(key);

}else{

key2 = key;

return true;

}

}

}else if(this.testTTT() && this.left != null){

if(key < key1){

return left.insertNode(key);

}else if(key > key2){

return righ.insertNode(key);

}else{

return cent.insertNode(key);

}

}else if(this.testTTT() && this.left == null){

//向上查找，找到tnode的父结点为二结点或者根结点

TTTree tnode = this;

//将三结点转化为临时四结点临时存放中间key

Integer tempkey = key;

if(key < key1){

tempkey = key1;

key1 = key;

}else if(key > key2){

tempkey = key2;

key2 = key;

}

//将2-3-4转化为一棵2-3树

TTTree temptree = new TTTree(tnode.key2,null);

tnode.key2 = null;

if(tnode.parent != null && tnode.parent.cent != null && temptree.key1 < tnode.parent.cent.key1){

TTTree t = tnode.parent.righ;

tnode.parent.righ = tnode.parent.cent;

tnode.parent.cent = temptree;

temptree = t;

}else if(tnode.parent != null && tnode.parent.cent != null && temptree.key1 < tnode.parent.righ.key1){

TTTree t = tnode.parent.righ;

tnode.parent.righ = temptree;

temptree = t;

}

while(tnode.parent != null && tnode.parent.testTTT()){

//将tnode这个三结点变为二结点

if(tempkey < tnode.parent.key1){

tempkey = tempkey ^ tnode.parent.key1;

tnode.parent.key1 = tempkey ^ tnode.parent.key1;

tempkey = tempkey ^ tnode.parent.key1;

}else if(tempkey > tnode.parent.key2){

tempkey = tempkey ^ tnode.parent.key2;

tnode.parent.key2 = tempkey ^ tnode.parent.key2;

tempkey = tempkey ^ tnode.parent.key2;

}

TTTree t1 = new TTTree(tnode.parent.key2,null);

tnode.parent.key2 = null;

t1.left = tnode.parent.righ;

t1.righ = temptree;

tnode.parent.righ = tnode.parent.cent;

tnode.parent.cent = null;

tnode = tnode.parent;

t1.left.parent = t1;

t1.righ.parent = t1;

temptree = t1;

}

if(tnode.parent != null && tnode.parent.testTT()){

//将tnode二结点转化为三结点，将临时树合并到tnode结点

if(tempkey < tnode.parent.key1){

tnode.parent.key2 = tnode.parent.key1;

tnode.parent.key1 = tempkey;

tnode.parent.cent = temptree;

}else{

tnode.parent.key2 = tempkey;

tnode.parent.cent = tnode.parent.righ;

tnode.parent.righ = temptree;

}

temptree.parent = tnode.parent;

}else if(tnode.parent == null){

TTTree t1 = new TTTree(tempkey,null);

t1.left = tnode;

t1.righ = temptree;

temptree.parent = t1;

tnode.parent = t1;

//t1.left.cent = null;

}

return true;

}else{

System.out.println("不是标准二结点，或三结点，无法插入");

return false;

}

}

private void show(){

System.out.format("key1=[%d],key2=[%d],下级：%s;",key1,key2,this.left==null?"无":"有");

if(this.left != null){

this.left.show();

}

if(this.cent != null){

this.cent.show();

}

if(this.righ != null){

this.righ.show();

}

}

public static void test1(){

TTTree root = new TTTree(8,null);

root.getRoot().insertNode(2);

root = root.getRoot();

root.getRoot().insertNode(4);

root = root.getRoot();

root.getRoot().insertNode(11);

root = root.getRoot();

root.getRoot().insertNode(1);

root = root.getRoot();

root.getRoot().insertNode(3);

root = root.getRoot();

root.getRoot().insertNode( 5);

root = root.getRoot();

root.getRoot().insertNode(6);

root = root.getRoot();

root.getRoot().insertNode(9);

root = root.getRoot();

root.getRoot().insertNode(13);

root = root.getRoot();

root.getRoot().insertNode(7);

root = root.getRoot();

root.show();

}

public static void test2(){

TTTree root = new TTTree(4,null);

root = root.getRoot();

root.getRoot().insertNode(21);

root = root.getRoot();

root.getRoot().insertNode(2);

root = root.getRoot();

root.getRoot().insertNode(8);

root = root.getRoot();

root.getRoot().insertNode(15);

root = root.getRoot();

root.getRoot().insertNode(26);

root = root.getRoot();

root.getRoot().insertNode(1);

root = root.getRoot();

root.getRoot().insertNode(3);

root = root.getRoot();

root.getRoot().insertNode(5);

root = root.getRoot();

root.getRoot().insertNode(6);

root = root.getRoot();

root.getRoot().insertNode(10);

root = root.getRoot();

root.getRoot().insertNode(11);

root = root.getRoot();

root.getRoot().insertNode(19);

root = root.getRoot();

root.getRoot().insertNode(17);

root = root.getRoot();

root.getRoot().insertNode(22);

root = root.getRoot();

root.getRoot().insertNode(23);

root = root.getRoot();

root.getRoot().insertNode(30);

root = root.getRoot();

root.getRoot().insertNode(12);

root = root.getRoot();

root.getRoot().insertNode(13);

root = root.getRoot();

root.getRoot().insertNode(14);

root = root.getRoot();

root.getRoot().insertNode(18);

root = root.getRoot();

root.getRoot().insertNode(20);

root = root.getRoot();

root.show();

}

public static void main(String[] args){

test2();

}

}

输出结果：

key1=[10],key2=[null],下级：有;key1=[4],key2=[null],下级：有;key1=[2],key2=[null],下级：有;key1=[1],key2=[null],下级：无;key1=[3],key2=[null],下级：无;key1=[6],key2=[null],下级：有;key1=[5],key2=[null],下级：无;key1=[8],key2=[null],下级：无;key1=[15],key2=[21],下级：有;key1=[12],key2=[null],下级：有;key1=[11],key2=[null],下级：无;key1=[13],key2=[14],下级：无;key1=[18],key2=[null],下级：有;key1=[17],key2=[null],下级：无;key1=[19],key2=[20],下级：无;key1=[23],key2=[null],下级：有;key1=[22],key2=[null],下级：无;key1=[26],key2=[30],下级：无;

* + 1. 2-3-4树
    2. B树

B树是一种平衡的多路查找树，2-3树和2-3-4树都是B树的特例。

我们把结点最大的孩子树数目称为B树的阶，因此，2-3树是3阶的B树，2-3-4树是4阶的B树。

一个m阶的B树具有如下属性：

如果根结点不是叶子结点，则其至少有两棵子树。

每一个非根的分支结点都有k-1个元素（关键字）和k个孩子，其中k满足（m/2）的向上取整 <= k <= m。

所有叶子结点都位于同一层次

每一个分支结点包含下列信息数据：

N,a0,k1,a1,k2,a2mk3……

且其中k为关键字，kn<kn+1

A为指向子树根结点的指针

* + 1. 散列表（哈希表）查找

如果我们要在a[]中查找key关键字：

顺序表查找：挨个逐一比较

有序表查找：插值查找

散列表查找：存储位置=f(关键字)

散列技术是在记录的存储位置和它的关键字之间建立一个对应关系f，使得每个关键字对应一个存储位置f(key);q 我们把这种对应关系f称为散列函数，又称为哈希函数。采用散列技术将记录存储在一块连续的存储空间中，这块连续存储空间成为散列表或哈希表。

好的构造散列函数有两个基本原则：

* + - 1. 计算简单
      2. 分布均匀

构造1:直接定值法即f(key) =a\*key+b;ab为常数

构造2:数字分析法通常适合处理关键字位数比较大的情况，一般通过抽取实现

例如：在同一区域存储身份证信息前6位相同的概率比较大，通过分析可以抽取后几位作为hash值

构造3：平方取中法

平方取中法是将关键字平方之后取中间若干位数字作为散列地址，因此适合于不知道关键字的分布并且位数不大情况。大数取平方效率不高

构造4：折叠法

折叠法是将关键字从左到右分割成位数相等的几部分，不足补零，然后将这几部分叠加求和，并按散列表表长取后几位作为散列地址。

构造5：除留余数法

此方法为最常用的构造散列函数方法，对于散列表长为M的散列函数计算公式为

-f(key) = key mod p(p<M)

事实上，这个方法不仅可以对关键字直接取模，也可以通过折叠、平方以中后再取模。P的选择是关键。应尽可能减少冲突。P小于或等于表长接近n的最小质数或不包含小于20质因子的和数。

构造6：随机数法

选择一个随机数，取关键字的随机函数值为它的散列地址。F(key) = random();

适合于关键字长度不相等的情况。

现实中，我们应该视不同的情况采用不同的散列函数“

-计算散列地址所需要的时间

-关键字的长度

-散列表的大小

-关键字的布情况

-记录查找的频率

散列值冲突的解决办法

方法1：开放定值法

所谓开放定址法就是一旦发生了冲突，就去寻找下一个散列地址，只要散列表足够大，空的散列地址总能找到，并将记录存入。

线性探测法

Fi(key) = f(f(key)+di) mod m(di=1,2,……,m-1)

平方探测法

Fi(key) = f(f(key)+di) mod m(di=1^2,-1^2,2^2,-2^2……)

随机探测法

Fi(key) = f(f(key)+di) mod m(di是由一个随机获取是到的数列)

方法2：再散列函数法

准备多散列函数，如果一个冲突，启用下一个算法，但要考虑到所有散列函数都失败的情况。

方法3：链接地址法

方法4：公共溢出区法

专门设置一个溢出区表，如果计算值冲突，就顺序存放到溢出区

package com.learn.hash;

public class HashTable {

private static final int NULLKEY = -65536;

private static int DEFAULT\_SIZE = 16;

int[] elems;

int size;

HashTable(){

elems = new int[DEFAULT\_SIZE];

size = DEFAULT\_SIZE;

for(int i=0;i<DEFAULT\_SIZE;i++){

elems[i] = NULLKEY;

}

}

int Hash(int key){

return key % DEFAULT\_SIZE;

}

public boolean insertHash(int key){

int addr = Hash(key);

while(elems[addr] != NULLKEY){

//存在冲突 开放定值法顺序探测

if(addr == DEFAULT\_SIZE){

return false;

}

addr = Hash(addr+1);

}

elems[addr] = key;

return true;

}

int searchHash(int key){

int addr = Hash(key);

while(elems[addr] != key){

addr = Hash(addr+1);

if(addr == DEFAULT\_SIZE){

return -1;

}

}

return addr;

}

public static void main(String[] args){

HashTable t = new HashTable();

t.insertHash(1);

t.insertHash(3);

t.insertHash(16);

t.insertHash(17);

t.insertHash(2);

System.out.println(t.searchHash(1));

System.out.println(t.searchHash(3));

System.out.println(t.searchHash(17));

System.out.println(t.searchHash(2));

}

}

* 1. 排序

排序分内排序和外排序

内排序数据量不是非常大，可以在内存中直接进行

外排序的数据量大，必须在内存和外存共同完成

排序一般包含比较和移动操作

其性能：

时间性能：尽可能减少比较、移动操作

辅助空间：临时存放数据少

算法：算法复杂度低

* + 1. 冒泡排序

冒泡排序严格来说应该是两两相邻排序，即niubisort（）;

package com.learn.sort;

import java.util.Arrays;

public class BubbleSort {

public static void main(String[] args){

int[] a1 = {1,2,3,0,8,7,6,5,9,4};

int[] a2 = a1.clone();

int[] arr = diaosisort(a1);

Arrays.stream(arr).forEach(o->{

System.out.printf("%d,",o);

});

System.out.println();

arr = niubisort(a2);

Arrays.stream(arr).forEach(o->{

System.out.printf("%d,",o);

});

}

public static int[] diaosisort(int [] arr){

int temp,count1 = 0,count2 = 0;

for (int i = 0; i < arr.length; i++) {

for (int j = i+1; j < arr.length; j++) {

count1++;

if(arr[i]>arr[j]){

count2++;

temp = arr[i];

arr[i] = arr[j];

arr[j] = temp;

}

}

}

System.out.format("共执行比较%d次，共执行移动%d次",count1,count2);

return arr;

}

public static int[] niubisort(int[] arr){

boolean flag = true;int count1 = 0,count2 = 0;

for (int i = 0; i < arr.length && flag; i++) {

flag = false;

for(int j=arr.length-1;j>i;j--){

count1++;

if(arr[j-1] > arr[j]){

count2++;

arr[j] = arr[j] ^ arr[j-1];

arr[j-1] = arr[j] ^ arr[j-1];

arr[j] = arr[j] ^ arr[j-1];

flag = true;

}

}

}

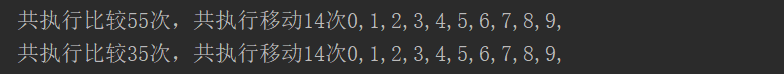
System.out.format("共执行比较%d次，共执行移动%d次",count1,count2);

return arr;

}

}

执行结果：



* + 1. 选择排序

冒泡排序通过频繁的比较与交换操作来实现排序。

选择排序算法是通过n-i次关键字间的比较，从n-i+1个记录中选出关键字最小的记录，并和第i(1<=i<=n)个记录交换。

package com.learn.sort;

import java.util.Arrays;

public class SelectSort {

public static void main(String[] args){

int[] a1 = {1,2,3,0,8,7,6,5,9,4};

int[] arr = selectSort(a1);

Arrays.stream(arr).forEach(o->{

System.out.printf("%d,",o);

});

System.out.println();

}

public static int[] selectSort(int[] arr){

int count1 = 0,count2 = 0;

for (int i = 0; i < arr.length; i++) {

int main = i;

for (int j = i+1; j < arr.length; j++) {

count1++;

if(arr[main]>arr[j]){

main = j;

}

}

if(main != i){

count2++;

arr[i] = arr[i] ^ arr[main];

arr[main] = arr[i] ^ arr[main];

arr[i] = arr[i] = arr[i] ^ arr[main];

}

}

System.out.format("共执行比较%d次，共执行移动%d次",count1,count2);

return arr;

}

}

执行结果：



* + 1. 简单插入排序

直接插入排序算法的基本操作是将一个记录插入到已经排序好的有序表中，从而得到一个新的、记录数增加1的有序表。

package com.learn.sort;

import java.util.Arrays;

public class InsertSort {

public static void main(String[] args){

int[] a1 = {1,2,3,0,8,7,6,5,9,4};

int[] arr = insertSort(a1);

Arrays.stream(arr).forEach(o->{

System.out.printf("%d,",o);

});

System.out.println();

}

public static int[] insertSort(int[] arr){

int temp,j,count1 = 0,count2 = 0;

for (int i = 1; i < arr.length; i++) {

if(arr[i]<arr[i-1]){

count1++;

temp = arr[i];

for (j= i-1; j>=0 && arr[j] > temp; j--) {

count2++;

arr[j+1] = arr[j];

}

arr[j+1] = temp;

}

}

System.out.format("执行插入操作%d次，执行移动操作%d次",count1,count2);

return arr;

}

}



* + 1. 希尔排序

希尔排序的实质就是分组插入排序，该方法又称缩小增量排序，因DL．Shell于1959年提出而得名。

该方法的基本思想是：先将整个待排元素序列分割成若干个子序列（由相隔某个“增量”的元素组成的）分别进行直接插入排序，然后依次缩减增量再进行排序，待整个序列中的元素基本有序（增量足够小）时，再对全体元素进行一次直接插入排序。因为直接插入排序在元素基本有序的情况下（接近最好情况），效率是很高的，因此希尔排序在时间效率上比前两种方法有较大提高。

package com.learn.sort;

import java.util.Arrays;

public class ShellSort {

public static int[] shellSort(int[] arr){

int count1 = 0,count2 = 0;

for(int gap = arr.length/2;gap>0;gap/=2){

for (int i = gap; i < arr.length; i++) {

int j= i;

count1++;

while(j-gap >=0 && arr[j] < arr[j-gap]){

count2++;

arr[j] = arr[j] ^ arr[j-gap];

arr[j-gap] = arr[j] ^ arr[j-gap];

arr[j] = arr[j] ^ arr[j-gap];

j-=gap;

}

}

}

System.out.format("执行比较操作%d次，执行交换操作%d次\n",count1,count2);

return arr;

}

public static void main(String[] args){

int[] a1 = {1,2,3,0,8,7,6,5,9,4};

int[] arr = shellSort(a1);

Arrays.stream(arr).forEach(o->{

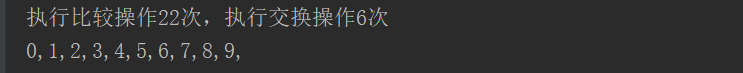
System.out.printf("%d,",o);

});

System.out.println();

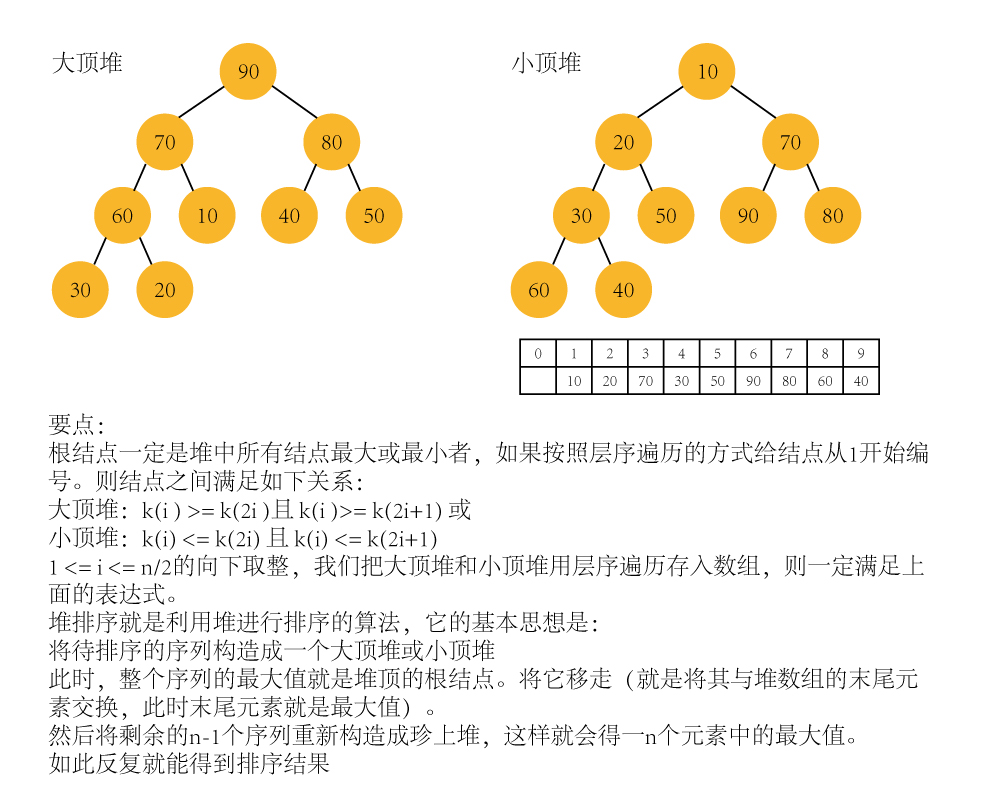
}

}



* + 1. 堆排序

对选择排序改进，利用上一次比较的结果



package com.learn.sort;

import java.util.Arrays;

public class HeapSort {

static int count = 0;

public static int[] heapSort(int[] arr){

int length = arr.length-1;

for (int i = length /2; i>0; i--){

HeapAdjust(arr,i, length);

}

for(int i = length; i>1; i--){

//将顶元素移出大顶堆

arr[1] = arr[1] ^ arr[i];

arr[i] = arr[1] ^ arr[i];

arr[1] = arr[1] ^ arr[i];

//重新构建大顶堆

HeapAdjust(arr,1,i-1);

}

System.out.format("共计执行比较%d次",count);

return arr;

}

private static void HeapAdjust(int[] arr, int s, int e) {

//2\*s为左孩子

int temp = arr[s];

for(int i=2\*s;i<=e;i\*=2){

count++;

if(i<e && arr[i] < arr[i+1]){

i++;

}

if(temp >= arr[i]){

break;

}

arr[s] =arr[i];

s = i;

}

arr[s] = temp;

}

public static void main(String[] args){

int[] a1 = {-1,1,2,3,0,8,7,6,5,9,4};

int[] arr = heapSort(a1);

Arrays.stream(arr).forEach(o->{

System.out.printf("%d,",o);

});

System.out.println();

}

}



* + 1. 归并排序

归并排序就是利用归并的思想实现的排序算法，它的原理是假设初始序列有n个记录，则可以看成是n个有序的子序列，每个子序列的升序为1，然后两两归并，得到n/2向上取整个长度为2或1的有序子序列，再两两归并……，如此重复，直到得到一个长度为n的有序序列为止，这种排序方法称为2路归并排序。

package com.learn.sort;

import java.util.Arrays;

public class MergeSort {

public static int[] mergeSort(int[] arr){

int length = arr.length;

int[] list1 = null,list2;

if(length>1){

int list1\_size = length / 2;

list1 = Arrays.copyOfRange(arr,0,list1\_size);

list2 = Arrays.copyOfRange(arr,list1\_size,length);

list1 = mergeSort(list1);

list2 = mergeSort(list2);

arr = merge(list1,list2);

}

return arr;

}

private static int[] merge(int[] list1, int[] list2) {

int i = 0,j=0,k=0;

int[] temp = new int[list1.length+list2.length];

while( i< list1.length && j<list2.length){

temp[k++] = list1[i] < list2[j]?list1[i++]:list2[j++];

}

while(i<list1.length){

temp[k++] = list1[i++];

}

while(j<list2.length){

temp[k++] = list2[j++];

}

Arrays.stream(temp).forEach(o->System.out.print(o+","));

System.out.println();

return temp;

}

public static void main(String[] args){

int[] a1 = {1,2,3,0,8,7,6,5,9,4};

int[] arr = mergeSort(a1);

System.out.print("排序完成结果：");

Arrays.stream(arr).forEach(o->{

System.out.printf("%d,",o);

});

}

}

* + 1. 快速排序

快速排序（Quicksort）是对冒泡排序的一种改进。

快速排序由C. A. R. Hoare在1962年提出。它的基本思想是：通过一趟排序将要排序的数据分割成独立的两部分，其中一部分的所有数据都比另外一部分的所有数据都要小，然后再按此方法对这两部分数据分别进行快速排序，整个排序过程可以递归进行，以此达到整个数据变成有序序列。

package com.learn.sort;

import java.util.Arrays;

public class QuickSort {

public static int[] quickSort(int[] arr){

int l=0,r=arr.length-1;

if (r <= l) return arr;

int main = arr[0];

while(l<r){

while(l<r && arr[r] > main){

r--;

}

arr[l] = arr[r];

while(l<r && arr[l] < main){

l++;

}

arr[r] = arr[l];

}

int[] list1 = quickSort(Arrays.copyOfRange(arr,0,l));

System.arraycopy(list1,0,arr,0,list1.length);

arr[l] = main;

int[] list2 = quickSort(Arrays.copyOfRange(arr,l+1,arr.length));

System.arraycopy(list2,0,arr,l+1,list2.length);

return arr;

}

public static void main(String[] args){

int[] a1 = {1,2,3,0,8,7,6,5,9,4};

int[] arr = quickSort(a1);

Arrays.stream(arr).forEach(o->{

System.out.printf("%d,",o);

});

System.out.println();

}

}

1. 加密解密
   1. BASE64编码

Base64是网络上最常见的用于传输8Bit字节代码的编码方式之一

public void base64(){

String str = "abcd";

Base64.Encoder encoder = Base64.getEncoder();

byte[] bs = encoder.encode(str.getBytes());

System.out.println(new String(bs));

Base64.Decoder decoder = Base64.getDecoder();

System.out.println(new String(decoder.decode(bs)));

}

* 1. 消息摘要
     1. 消息摘要

消息摘要（Message Digest）又称为数字摘要(Digital Digest)。，它由一个单向Hash加密函数对消息进行作用而产生。这一串密文亦称为数字指纹(Finger Print)，它有固定的长度。因此，消息摘要算法有真破解和假破解之分；真破解在于找到两个不同原数据加密成同一结果称真破解，利用数据库保存对应值反向查找称假破解。

md（Message Digest） (md2,md4,md5) 128位

字符串MD5：

public static String jdkMD5(String password) throws NoSuchAlgorithmException{

MessageDigest md = MessageDigest.getInstance("MD5");

byte[] md5Bytes = md.digest(password.getBytes());

return printHexString(md5Bytes);

}

public static String printHexString( byte[] b) {

StringBuilder sb = new StringBuilder();

for (int i = 0; i < b.length; i++) {

String hex = Integer.toHexString(b[i] & 0xFF);

if (hex.length() == 1) {

hex = '0' + hex;

}

sb.append(hex.toUpperCase());

}

return sb.toString();

}

文件MD5：

public static String getMd5ByFile(File file){

String value = null;

try (FileInputStream in = new FileInputStream(file);){

MappedByteBuffer byteBuffer = in.getChannel().map(FileChannel.MapMode.READ\_ONLY, 0, file.length());

MessageDigest md5 = MessageDigest.getInstance("MD5");

md5.update(byteBuffer);

value = new BigInteger(1, md5.digest()).toString(16);

} catch (Exception e) {

e.printStackTrace();

}

return value;

}

* + 1. 安全散列

sha（Secure Hash Algorithm）(sha1,sha2)

public static String jdkSha(String src) throws NoSuchAlgorithmException{

MessageDigest sha = MessageDigest.getInstance("SHA");

sha.update(src.getBytes());

return MD5Encryption.printHexString(sha.digest());

}

* + 1. 消息认证码

mac(Message Authentication Code) 融合md,mac,消息认证码包含私钥，用于验证数据的完整性 是数字签名核心算法

for (Provider provider1 :Security.getProviders()){

System.out.println(provider1);

for(Provider.Service service : provider1.getServices()){

System.out.println("类型："+service.getType()+" 算法:"+service.getAlgorithm());

}

}

以下是SUN公司JAVA9版本所提供的mac算法服务产品：MAC有更多算法提供

类型：KeyGenerator 算法:HmacMD5

类型：KeyGenerator 算法:HmacSHA1

类型：KeyGenerator 算法:HmacSHA224

类型：KeyGenerator 算法:HmacSHA256

类型：KeyGenerator 算法:HmacSHA384

类型：KeyGenerator 算法:HmacSHA512

public static void mac(String str) throws Exception {

//生成密钥

KeyGenerator keyGenerator = KeyGenerator.getInstance("HmacSHA512");

SecretKey secretKey = keyGenerator.generateKey();

byte[] bs = secretKey.getEncoded();

for(byte b:bs) System.out.print(b+",");

System.out.println();

//保存密钥（十六进制）

String keystr = new BigInteger(1, bs).toString(16);

System.out.println("key :"+keystr);

System.out.println();

//利用Byte数组生成密钥用于验证

bs = new byte[]{88,-110,-78,113,-17,84,-82,63,-32,113,56,-60,-38,-107,-81,-13,-110,-123,-56,124,-67,65,-120,-49,47,-96,-47,113,94,10,111,69,26,35,-56,84,-44,72,-39,-103,70,-25,-104,-26,97,32,-93,28,48,-14,-76,114,9,-15,7,94,-68,-88,-53,120,-14,-97,10,63};

secretKey=new SecretKeySpec(bs,"HmacSHA512");

//Mac对象用于生成摘要

Mac mac = Mac.getInstance("HmacSHA512");

mac.init(secretKey);

byte[] bss = mac.doFinal(str.getBytes());

for(byte b:bss){

System.out.print(b+",");

}

System.out.println();

}

* 1. 对称加密
     1. DES

1998年后实用化DES破译机的出现彻底宣告DES算法已不具备安全性，1999年NIST颁布新标准，规定DES算法只能用于遗留加密系统，但不限制使用DESede算法。当今DES算法正是推出历史舞台，AES算法称为他的替代者。

* + 1. DESede

DESede即三重DES加密算法，也被称为3DES或者Triple DES。使用三(或两)个不同的密钥对数据块进行三次(或两次)DES加密(加密一次要比进行普通加密的三次要快)。三重DES的强度大约和112-bit的密钥强度相当。通过迭代次数的提高了安全性，但同时也造成了加密效率低的问题。正因DESede算法效率问题，AES算法诞生了。

* + 1. AES

密码学中的高级加密标准（Advanced Encryption Standard，AES），又称Rijndael加密法，是美国联邦政府采用的一种区块加密标准。这个标准用来替代原先的DES，已经被多方分析且广为全世界所使用。

AES支持：

密钥长度: 128bit, default: 128 bi

工作模式: ECB/CBC/PCBC/CTR/CTS/CFB/CFB8 to CFB128/OFB/OBF8 to OFB128

填充方式: Nopadding/PKCS5Padding/ISO10126Padding/

@Test

public void aes() throws Exception {

String str = "abcdefg";

KeyGenerator keyGenerator = KeyGenerator.getInstance("AES");

SecretKey secretKey = keyGenerator.generateKey();

Cipher cipher = Cipher.getInstance("AES/ECB/PKCS5Padding");

cipher.init(Cipher.ENCRYPT\_MODE, secretKey);

byte[] bs = cipher.doFinal(str.getBytes());

cipher.init(Cipher.DECRYPT\_MODE,secretKey);

byte[] strbs = cipher.doFinal(bs);

System.out.println(new String(strbs));

}

* 1. 非对称加密
     1. DH

DH 算法加密的过程比较复杂，首先我们假设发送方是 A，接受方是 B。A 首先生成公钥和密钥，将公钥发送出去，B 接收到 A发送的公钥，然后利用该公钥生成自己的公钥和密钥，再将自己的公钥发送给 A，这个时候 A 拥有了自己的公钥，密钥和 B 的公钥，B 拥有了自己的公钥密钥和 A 的公钥。

之后， A 就可以使用 A自己的密钥 + B的公钥 获取到本地的密钥，B也是如此，这个时候 A 和 B 生成的本地密钥其实是相同的，这样的话也就变成了用相同的密钥加密，用相同的密钥解密。而且这样的话，我们数据传递过程中传递的是 A 和 B 的公钥，就算被黑客截取了也无济于事，他们不可能凭借着公钥将数据解密，从而保证了数据的安全性。

public void dh() throws Exception {

KeyPairGenerator keyPairGenerator = KeyPairGenerator.getInstance("DH");//密钥对生成器

keyPairGenerator.initialize(512);//指定密钥长度

KeyPair keyPair = keyPairGenerator.generateKeyPair();

DHPrivateKey j\_dhprivateKey = (DHPrivateKey)keyPair.getPrivate();

DHPublicKey j\_dhpublicKey = (DHPublicKey)keyPair.getPublic();

String jb64\_privateKey = new String(Base64.getEncoder().encode(j\_dhprivateKey.getEncoded()));

String jb64\_publicKey = new String(Base64.getEncoder().encode(j\_dhpublicKey.getEncoded()));

System.out.println("甲的私钥："+jb64\_privateKey);

System.out.println("由的公钥："+jb64\_publicKey);

DHParameterSpec dhParameterSpec = j\_dhpublicKey.getParams();

keyPairGenerator = KeyPairGenerator.getInstance("DH");

keyPairGenerator.initialize(dhParameterSpec);

keyPair = keyPairGenerator.generateKeyPair();

DHPublicKey y\_dhPublicKey = (DHPublicKey) keyPair.getPublic();

DHPrivateKey y\_dhPrivateKey = (DHPrivateKey) keyPair.getPrivate();

String yb64\_privateKey = new String(Base64.getEncoder().encode(y\_dhPrivateKey.getEncoded()));

String yb64\_publicKey = new String(Base64.getEncoder().encode(y\_dhPublicKey.getEncoded()));

System.out.println("乙的私钥："+yb64\_privateKey);

System.out.println("乙的公钥："+yb64\_publicKey);

//计算乙的本地密钥

KeyFactory keyFactory = KeyFactory.getInstance("DH");

X509EncodedKeySpec encodedKeySpec = new X509EncodedKeySpec(Base64.getDecoder().decode(jb64\_publicKey.getBytes()));

DHPublicKey dhPublicKey = (DHPublicKey) keyFactory.generatePublic(encodedKeySpec);

PKCS8EncodedKeySpec encodedKeySpec2 = new PKCS8EncodedKeySpec(y\_dhPrivateKey.getEncoded());

DHPrivateKey dhPrivateKey = (DHPrivateKey) keyFactory.generatePrivate(encodedKeySpec2);

KeyAgreement keyAgreement = KeyAgreement.getInstance("DH");

keyAgreement.init(dhPrivateKey);

keyAgreement.doPhase(dhPublicKey, true);

SecretKey y\_secretKey = keyAgreement.generateSecret("AES");

String yb64\_secreKey = new String(Base64.getEncoder().encode(y\_secretKey.getEncoded()));

System.out.println("乙的本地密钥："+yb64\_secreKey);

//计算甲的本地密钥

keyFactory = KeyFactory.getInstance("DH");

encodedKeySpec = new X509EncodedKeySpec(Base64.getDecoder().decode(yb64\_publicKey.getBytes()));

dhPublicKey = (DHPublicKey) keyFactory.generatePublic(encodedKeySpec);

encodedKeySpec2 = new PKCS8EncodedKeySpec(j\_dhprivateKey.getEncoded());

dhPrivateKey = (DHPrivateKey) keyFactory.generatePrivate(encodedKeySpec2);

keyAgreement = KeyAgreement.getInstance("DH");

keyAgreement.init(dhPrivateKey);

keyAgreement.doPhase(dhPublicKey, true);

SecretKey j\_secretKey = keyAgreement.generateSecret("AES");

String jb64\_secreKey = new String(Base64.getEncoder().encode(j\_secretKey.getEncoded()));

System.out.println("甲的本地密钥："+jb64\_secreKey);

//甲给乙传数据;

Key key = new SecretKeySpec(j\_secretKey.getEncoded(),"AES");//生成本地密钥

Cipher cipher = Cipher.getInstance("AES/ECB/PKCS5Padding");

cipher.init(Cipher.ENCRYPT\_MODE, key);//设置加密模式并且初始化key

byte[] jtoydata = cipher.doFinal("朵儿".getBytes());

System.out.println("甲发出消息："+new String(jtoydata));

//已收到后解密数据

key = new SecretKeySpec(y\_secretKey.getEncoded(), "AES");//生成本地密钥

cipher = Cipher.getInstance("AES/ECB/PKCS5Padding");

cipher.init(Cipher.DECRYPT\_MODE, key);

System.out.println("已收到甲传过来的消息："+new String(cipher.doFinal(jtoydata)));

//双方的本地密钥是一致的 ，所以已跟甲传数据效果一样

}

* + 1. RSA

RSA是一种支持私钥加密，公钥解密的算法

public static void jdkRSA(String data){

try {

KeyPairGenerator keyPairGenerator = KeyPairGenerator.getInstance("RSA");

keyPairGenerator.initialize(512);

KeyPair keyPair = keyPairGenerator.generateKeyPair();

RSAPublicKey rsaPublicKey = (RSAPublicKey)keyPair.getPublic();

RSAPrivateKey rsaPrivateKey = (RSAPrivateKey)keyPair.getPrivate();

System.out.println("public key:"+ new BigInteger(1, rsaPublicKey.getEncoded()).toString(16));

System.out.println("private key:"+ new BigInteger(1, rsaPrivateKey.getEncoded()).toString(16));

//私钥加密、公钥解密--加密

PKCS8EncodedKeySpec pkcs8EncodedKeySpec = new PKCS8EncodedKeySpec(rsaPrivateKey.getEncoded());

KeyFactory keyFactory = KeyFactory.getInstance("RSA");

PrivateKey privateKey = keyFactory.generatePrivate(pkcs8EncodedKeySpec);

Cipher cipher = Cipher.getInstance("RSA");

cipher.init(Cipher.ENCRYPT\_MODE,privateKey);

byte[] result = cipher.doFinal(data.getBytes());

System.out.println("私钥加密、公钥解密--加密："+new BigInteger(1, result).toString(16));

X509EncodedKeySpec x509EncodedKeySpec = new X509EncodedKeySpec(rsaPublicKey.getEncoded());

keyFactory = KeyFactory.getInstance("RSA");

PublicKey publicKey = keyFactory.generatePublic(x509EncodedKeySpec);

cipher = Cipher.getInstance("RSA");

cipher.init(Cipher.DECRYPT\_MODE,publicKey);

result = cipher.doFinal(result);

System.out.println("私钥加密、公钥解密-解密："+new String(result));

x509EncodedKeySpec = new X509EncodedKeySpec(rsaPublicKey.getEncoded());

keyFactory = KeyFactory.getInstance("RSA");

publicKey = keyFactory.generatePublic(x509EncodedKeySpec);

cipher = Cipher.getInstance("RSA");

cipher.init(Cipher.ENCRYPT\_MODE,publicKey);

result = cipher.doFinal(data.getBytes());

System.out.println("公钥加密、私钥解密--加密："+new BigInteger(1, result).toString(16));

pkcs8EncodedKeySpec = new PKCS8EncodedKeySpec(rsaPrivateKey.getEncoded());

keyFactory = KeyFactory.getInstance("RSA");

privateKey = keyFactory.generatePrivate(pkcs8EncodedKeySpec);

cipher = Cipher.getInstance("RSA");

cipher.init(Cipher.DECRYPT\_MODE,privateKey);

result = cipher.doFinal(result);

System.out.println("公钥加密、私钥解密--加密：" +new String(result));

} catch (Exception e) {

e.printStackTrace();

}

}

* 1. 数字签名
     1. IDSA

数字签名算法（DSA仅包含数字签名）

加密方法：

|  |  |  |  |
| --- | --- | --- | --- |
| 算法 | 蜜钥长度 | 默认 | 实现方 |
| SHA1withDSA | 512-1024 | 1024 | JDK |
| SHA224withDSA | 64的整数倍 |  | BC |
| SHA256withDSA |  |  | BC |
| SHA384withDSA |  |  | BC |
| SHA512withDSA |  |  | BC |

public static void jdkDSA(String data){

KeyPairGenerator keyPairGenerator;

try {

//初始化密钥

keyPairGenerator = KeyPairGenerator.getInstance("DSA");

keyPairGenerator.initialize(512);

KeyPair keyPair = keyPairGenerator.generateKeyPair();

DSAPublicKey dsaPublicKey = (DSAPublicKey) keyPair.getPublic();

DSAPrivateKey dsaPrivateKey = (DSAPrivateKey) keyPair.getPrivate();

//执行签名

PKCS8EncodedKeySpec pkcs8EncodedKeySpec = new PKCS8EncodedKeySpec(dsaPrivateKey.getEncoded());

KeyFactory keyFactory = KeyFactory.getInstance("DSA");

PrivateKey privateKey = keyFactory.generatePrivate(pkcs8EncodedKeySpec);

Signature signature = Signature.getInstance("SHA1withDSA");

signature.initSign(privateKey);

signature.update(data.getBytes());

byte[] result = signature.sign();

System.out.println("jdk dsa sign:"+new String(Base64.getEncoder().encode(result)));

System.out.println("jdk dsa sign:"+new BigInteger(1, result).toString(16));

//验证签名

X509EncodedKeySpec x509EncodedKeySpec = new X509EncodedKeySpec(dsaPublicKey.getEncoded());

keyFactory = keyFactory.getInstance("DSA");

PublicKey publicKey = keyFactory.generatePublic(x509EncodedKeySpec);

signature = Signature.getInstance("SHA1withDSA");

signature.initVerify(publicKey);

signature.update(data.getBytes());

boolean bool = signature.verify(result);

System.out.println("签名结果："+bool);

} catch (Exception e) {

e.printStackTrace();

}

}

* + 1. DECDSA

微软：椭圆曲线数字签名算法

速度快，强度高，签名短

加密算法：

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 算法 | 蜜钥长度 | 默认 | 签名长度 | 实现方 |
| NONEwithECDSA | 112-571 | 256 | 128 | JDK/BC |
| RIPEMD160withECDSA |  |  | 160 | BC |
| SHA1withECDSA |  |  | 160 | JDK/BC |
| SHA224withECDSA |  |  | 224 | BC |
| SHA256withECDSA |  |  | 256 | JDK/BC |
| SHA384withECDSA |  |  | 384 | JDK/BC |
| SHA512withECDSA |  |  | 512 | JDK/BC |

public static void jdkEcdsa(String data){

try {

//初始化密钥

KeyPairGenerator keyPairGenerator = KeyPairGenerator.getInstance("EC");

keyPairGenerator.initialize(256);

KeyPair keyPair = keyPairGenerator.generateKeyPair();

ECPublicKey ecPublicKey = (ECPublicKey) keyPair.getPublic();

ECPrivateKey ecPrivateKey = (ECPrivateKey) keyPair.getPrivate();

//执行签名

PKCS8EncodedKeySpec pkcs8EncodedKeySpec = new PKCS8EncodedKeySpec(ecPrivateKey.getEncoded());

KeyFactory keyFactory = KeyFactory.getInstance("EC");

PrivateKey privateKey = keyFactory.generatePrivate(pkcs8EncodedKeySpec);

Signature signature = Signature.getInstance("SHA1withECDSA");

signature.initSign(privateKey);

signature.update(data.getBytes());

byte[] result = signature.sign();

System.out.println("jdk ecdsa sign:"+new String(Base64.getEncoder().encode(result)));

//验证签名

X509EncodedKeySpec x509EncodedKeySpec = new X509EncodedKeySpec(ecPublicKey.getEncoded());

keyFactory = keyFactory.getInstance("EC");

PublicKey publicKey = keyFactory.generatePublic(x509EncodedKeySpec);

signature = Signature.getInstance("SHA1withECDSA");

signature.initVerify(publicKey);

signature.update(data.getBytes());

boolean bool = signature.verify(result);

System.out.println("验证结果："+bool);

} catch (Exception e) {

e.printStackTrace();

}

}

* + 1. IRSA

加密方法

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 算法 | 密钥长度 | 默认 | 签名长度 | 实现方 |
| MD2withRSA | 512-65536 | 1024 | 与密钥长度相同 | JDK |
| MD5withRSA | 64的整数倍 | 1024 | 与密钥长度相同 | JDK |
| SHA1withRSA |  | 1024 | 与密钥长度相同 | JDK |
| SHA224withRSA |  | 2048 | 与密钥长度相同 | BC |
| SHA256withRSA |  | 2048 | 与密钥长度相同 | BC |
| SHA384withRSA |  | 2048 | 与密钥长度相同 | BC |
| SHA512withRSA |  | 2048 | 与密钥长度相同 | BC |
| RIPEMD128withRSA |  | 2048 | 与密钥长度相同 | BC |
| RIPEMD160withRSA |  | 2048 | 与密钥长度相同 | BC |

public static void Jdkrsa(String data){

try{

//初始化密钥

KeyPairGenerator keyPairGenerator = KeyPairGenerator.getInstance("RSA");

keyPairGenerator.initialize(512);

KeyPair keyPair = keyPairGenerator.generateKeyPair();

RSAPublicKey pubk = (RSAPublicKey) keyPair.getPublic();

RSAPrivateKey prik = (RSAPrivateCrtKey) keyPair.getPrivate();

//执行签名

PKCS8EncodedKeySpec pkcs8EncodedKeySpec = new PKCS8EncodedKeySpec(prik.getEncoded());

KeyFactory keyFactory = KeyFactory.getInstance("RSA");

PrivateKey privateKey = keyFactory.generatePrivate(pkcs8EncodedKeySpec);

Signature signature = Signature.getInstance("MD5withRSA");

signature.initSign(privateKey);

signature.update(data.getBytes());

byte[] result = signature.sign();

System.out.println("jdk rsa sign:"+new String(Base64.getEncoder().encode(result)));

//验证签名

X509EncodedKeySpec x509EncodedKeySpec = new X509EncodedKeySpec(pubk.getEncoded());

keyFactory = keyFactory.getInstance("RSA");

PublicKey publicKey = keyFactory.generatePublic(x509EncodedKeySpec);

signature = Signature.getInstance("MD5withRSA");

signature.initVerify(publicKey);

signature.update(data.getBytes());

boolean bool = signature.verify(result);

System.out.println("jdk rsa verify:"+bool);

} catch (Exception e) {

e.printStackTrace();

}

}

1. 设计模式
   1. 设计模式分类：
      1. 创建型模式

这些设计模式提供了一种在创建对象的同时隐藏创建逻辑的方式，而不是使用 new运算符直接实例化对象。这使得程序在判断针对某个给定实例需要创建哪些对象时更加灵活。

工厂模式（Factory Pattern）

抽象工厂模式（Abstract Factory Pattern）

单例模式（Singleton Pattern）

建造者模式（Builder Pattern）

原型模式（Prototype Pattern）

* + 1. 结构型模式

这些设计模式关注类和对象的组合。继承的概念被用来组合接口和定义组合对象获得新功能的方式。

适配器模式（Adapter Pattern）

桥接模式（Bridge Pattern）

过滤器模式（Filter、Criteria Pattern）

组合模式（Composite Pattern）

装饰器模式（Decorator Pattern）

外观模式（Facade Pattern）

享元模式（Flyweight Pattern）

代理模式（Proxy Pattern）

* + 1. 行为型模式

这些设计模式特别关注对象之间的通信。

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）

空对象模式（Null Object Pattern）

策略模式（Strategy Pattern）

模板模式（Template Pattern）

访问者模式（Visitor Pattern）

* + 1. J2EE 模式

这些设计模式特别关注表示层。这些模式是由 Sun Java Center 鉴定的。

MVC 模式（MVC Pattern）

业务代表模式（Business Delegate Pattern）

组合实体模式（Composite Entity Pattern）

数据访问对象模式（Data Access Object Pattern）

前端控制器模式（Front Controller Pattern）

拦截过滤器模式（Intercepting Filter Pattern）

服务定位器模式（Service Locator Pattern）

* + 1. 设计模式的六大原则

1、开闭原则（Open Close Principle）

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

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

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

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

这个原则是开闭原则的基础，具体内容：针对接口编程，依赖于抽象而不依赖于具体。

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

这个原则的意思是：使用多个隔离的接口，比使用单个接口要好。它还有另外一个意思是：降低类之间的耦合度。由此可见，其实设计模式就是从大型软件架构出发、便于升级和维护的软件设计思想，它强调降低依赖，降低耦合。

5、迪米特法则，又称最少知道原则（Demeter Principle）

最少知道原则是指：一个实体应当尽量少地与其他实体之间发生相互作用，使得系统功能模块相对独立。

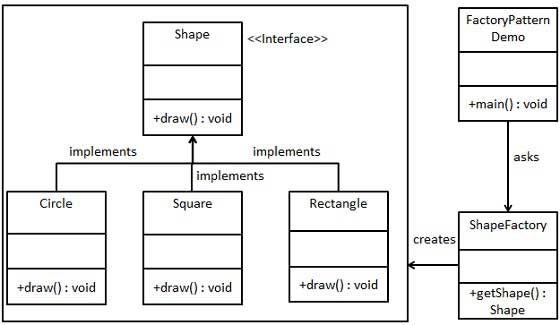
6、合成复用原则（Composite Reuse Principle）

合成复用原则是指：尽量使用合成/聚合的方式，而不是使用继承。

* 1. 工厂模式

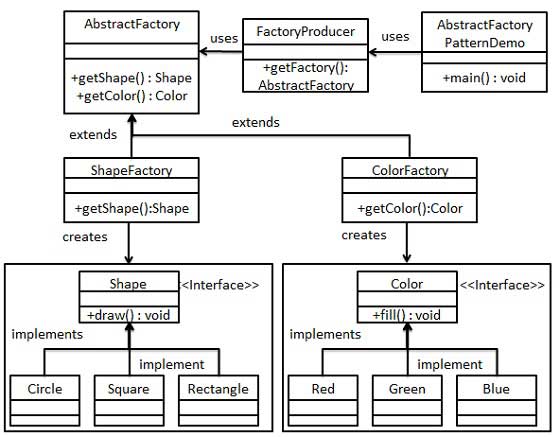
定义一个创建对象的接口，让其子类自己决定实例化哪一个工厂类，工厂模式使其创建过程延迟到子类进行。

**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。所以，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。



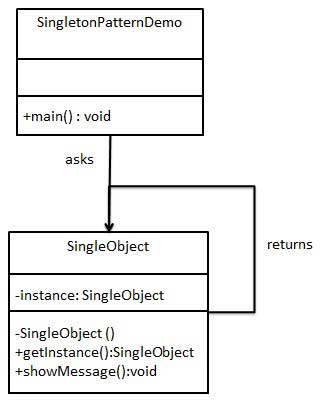
* 1. 抽象工厂模式

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。主要是解决在调用时接口选择性的问题。我们只需要使用AbstractFactory就可以



* 1. 单例模式

单例模式保证一个类只创建一个实例，并提供全局访问。从而避免一个类被频繁地创建与销毁。一般地，getInstance() 方法中需要使用同步锁 synchronized (Singleton.class) 防止多线程同时进入造成 instance 被多次实例化。



* + 1. 懒汉式，线程安全

public class Singleton {

private static Singleton instance;

private Singleton (){}

public static synchronized Singleton getInstance() {

if (instance == null) {

instance = new Singleton();

}

return instance;

}

}

* + 1. 饿汉式

public class Singleton {

private static Singleton instance = new Singleton();

private Singleton (){}

public static Singleton getInstance() {

return instance;

}

}

* + 1. 双检锁/双重校验

public class Singleton {

private volatile static Singleton singleton;

private Singleton (){}

public static Singleton getSingleton() {

if (singleton == null) {

synchronized (Singleton.class) {

if (singleton == null) {

singleton = new Singleton();

}

}

}

return singleton;

}

}

* + 1. 登记式/静态内部类

public class Singleton {

private static class SingletonHolder {

private static final Singleton INSTANCE = new Singleton();

}

private Singleton (){}

public static final Singleton getInstance() {

return SingletonHolder.INSTANCE;

}

}

* + 1. 枚举

class Resource{}

public enum SomeThing {

INSTANCE;

private Resource instance;

SomeThing() {

instance = new Resource();

}

public Resource getInstance() {

return instance;

}

}

* 1. 建造者模式

建造者模式（Builder Pattern）使用多个简单的对象一步一步构建成一个复杂的对象。主要解决在软件系统中，有时候面临着"一个复杂对象"的创建工作，其通常由各个部分的子对象用一定的算法构成；由于需求的变化，这个复杂对象的各个部分经常面临着剧烈的变化，但是将它们组合在一起的算法却相对稳定。

package com.builder;

public class User {

public String name;

public String password;

public char sex;

public int age;

public User(String name, String password, char sex, int age) {

super();

this.name = name;

this.password = password;

this.sex = sex;

this.age = age;

}

public static class UserBuilder{

private String name;

private String password;

private char sex;

private int age;

private UserBuilder(){}

public static UserBuilder create(){

return new UserBuilder();

}

public UserBuilder name(String name){

this.name = name;

return this;

}

public UserBuilder password(String password){

this.password = password;

return this;

}

public UserBuilder age(int age){

this.age = age;

return this;

}

public UserBuilder sex(char sex){

this.sex = sex;

return this;

}

public User build(){

return new User(name,password,sex,age);

}

}

}

public class Main {

public static void main(String[] args) {

User user = User.UserBuilder.create().age(10).name("zs").sex('w').build();

}

}

* 1. 原型模式

原型模式（Prototype Pattern）是用于创建重复的对象，同时又能保证性能。这种模式是实现了一个原型接口，该接口用于创建当前对象的克隆。当直接创建对象的代价比较大时，则采用这种模式。实现克隆操作，在 JAVA 继承 Cloneable，重写 clone();与通过对一个类进行实例化来构造新对象不同的是，原型模式是通过拷贝一个现有对象生成新对象的。浅拷贝实现 Cloneable，重写，深拷贝是通过实现 Serializable 读取二进制流。

public class Clo implements Cloneable {

@Override

protected Object clone() throws CloneNotSupportedException {

return super.clone();

}

}

public class Main {

public static void main(String[] args) {

Clo clo = new Clo();

try {

Clo clo1 = (Clo) clo.clone();

if(clo!=clo1){

System.out.println(clo.hashCode()+";"+clo1.hashCode());

}

} catch (CloneNotSupportedException e) {

e.printStackTrace();

}

}

}

* 1. 适配器模式

适配器模式（Adapter Pattern）是作为两个不兼容的接口之间的桥梁。这种类型的设计模式属于结构型模式，它结合了两个独立接口的功能。主要解决在软件系统中，常常要将一些"现存的对象"放到新的环境中，而新环境要求的接口是现对象不能满足的。

public class MP4Player {

public void palyMp4(String filename){

System.out.println(filename);

}

}·

public class MP3Player {

public void palyMp3(String filename){

System.out.println(filename);

}

}

public class AdapterPlayer {

public void play(String filename){

if(filename.toLowerCase().endsWith(".mp4")){

new MP4Player().palyMp4(filename);

}else if (filename.toLowerCase().endsWith(".mp3")) {

new MP3Player().palyMp3(filename);

}

}

}

public class Main {

public static void main(String args[]) {

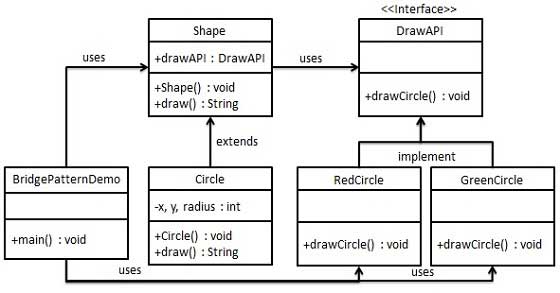
new AdapterPlayer().play("asdfasdf.mp3");

}

}

* 1. 桥接模式

桥接（Bridge）是用于把抽象化与实现化解耦，使得二者可以独立变化。这种类型的设计模式属于结构型模式，它通过提供抽象化和实现化之间的桥接结构，来实现二者的解耦。这种模式涉及到一个作为桥接的接口，使得实体类的功能独立于接口实现类。这两种类型的类可被结构化改变而互不影响。



public interface DrawAPI {

public void drawCircle(int radius, int x, int y);

}

public class RedCircle implements DrawAPI {

@Override

public void drawCircle(int radius, int x, int y) {

System.out.println("Drawing Circle[ color: red, radius: "

+ radius +", x: " +x+", "+ y +"]");

}

}

public class GreenCircle implements DrawAPI {

@Override

public void drawCircle(int radius, int x, int y) {

System.out.println("Drawing Circle[ color: green, radius: "

+ radius +", x: " +x+", "+ y +"]");

}

}

public abstract class Shape {

protected DrawAPI drawAPI;

protected Shape(DrawAPI drawAPI){

this.drawAPI = drawAPI;

}

public abstract void draw();

}

public class Circle extends Shape {

private int x, y, radius;

public Circle(int x, int y, int radius, DrawAPI drawAPI) {

super(drawAPI);

this.x = x;

this.y = y;

this.radius = radius;

}

public void draw() {

drawAPI.drawCircle(radius,x,y);

}

}

public class BridgePatternDemo {

public static void main(String[] args) {

Shape redCircle = new Circle(100,100, 10, new RedCircle());

Shape greenCircle = new Circle(100,100, 10, new GreenCircle());

redCircle.draw();

greenCircle.draw();

}

}

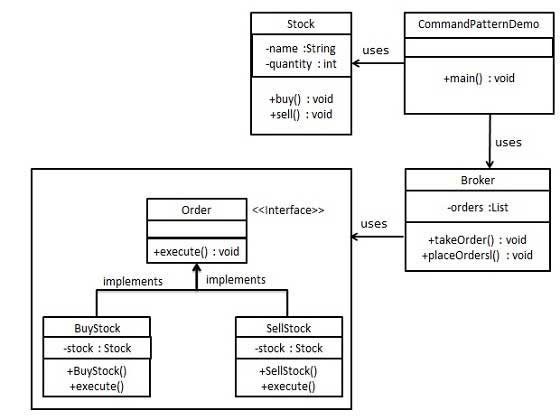
* 1. 命令模式

命令模式（Command Pattern）是一种数据驱动的设计模式，它属于行为型模式。请求以命令的形式包裹在对象中，并传给调用对象。调用对象寻找可以处理该命令的合适的对象，并把该命令传给相应的对象，该对象执行命令。

在软件系统中，行为请求者与行为实现者通常是一种紧耦合的关系，但某些场合，比如需要对行为进行记录、撤销或重做、事务等处理时或按钮事件等，都需要知道由谁在调用时，这种无法抵御变化的紧耦合的设计就不太合适。

优点： 1、降低了系统耦合度。 2、新的命令可以很容易添加到系统中去。

缺点：使用命令模式可能会导致某些系统有过多的具体命令类。



public interface Order {

void execute();

}

public class Stock {

private String name = "ABC";

private int quantity = 10;

public void buy(){

System.out.println("Stock [ Name: "+name+",

Quantity: " + quantity +" ] bought");

}

public void sell(){

System.out.println("Stock [ Name: "+name+",

Quantity: " + quantity +" ] sold");

}

}

public class BuyStock implements Order {

private Stock abcStock;

public BuyStock(Stock abcStock){

this.abcStock = abcStock;

}

public void execute() {

abcStock.buy();

}

}

public class SellStock implements Order {

private Stock abcStock;

public SellStock(Stock abcStock){

this.abcStock = abcStock;

}

public void execute() {

abcStock.sell();

}

}

import java.util.ArrayList;

import java.util.List;

public class Broker {

private List<Order> orderList = new ArrayList<Order>();

public void takeOrder(Order order){

orderList.add(order);

}

public void placeOrders(){

for (Order order : orderList) {

order.execute();

}

orderList.clear();

}

}

public class CommandPatternDemo {

public static void main(String[] args) {

Stock abcStock = new Stock();

BuyStock buyStockOrder = new BuyStock(abcStock);

SellStock sellStockOrder = new SellStock(abcStock);

Broker broker = new Broker();

broker.takeOrder(buyStockOrder);

broker.takeOrder(sellStockOrder);

broker.placeOrders();

}

}

* 1. 静态代理

静态代理：代理和被代理对象在代理之前是确定的。他们都实现相同的接口或者继承相同的抽象类。

* 1. 常用的代理模式

package com.learn.proxy;

public interface Moveable {

void move();

}

package com.learn.proxy;

import java.util.Random;

public class Car implements Moveable {

public static void main(String[] args) {

new Car().move();

}

@Override

public void move() {

System.out.println("洗车行驶中......");

try {

Thread.sleep(new Random().nextInt(1000));

} catch (InterruptedException e) {

// TODO Auto-generated catch block

e.printStackTrace();

}

System.out.println("洗车结束行驶结束......");

}

}

* + 1. 继承方式实现代理

package com.learn.proxy;

/\*\*

\* 使用继承的方式实现代理操作

\* @author zeimao77

\*

\*/

public class CarProxy01 extends Car {

public static void main(String[] args) {

new CarProxy01().move();

}

@Override

public void move() {

long stime = System.currentTimeMillis();

super.move();

System.out.println("行驶时间:"+(System.currentTimeMillis()-stime));

}

}

* + 1. 聚合的方式实现代理

package com.learn.proxy;

/\*\*

\* 使用聚合的方式实现代理

\* @author zeimao77

\*

\*/

public class CarProxy02 implements Moveable {

Car car;

CarProxy02(Car car){

this.car = car;

}

public static void main(String[] args) {

new CarProxy02(new Car()).move();

}

@Override

public void move() {

long stime = System.currentTimeMillis();

car.move();

System.out.println("行驶时间:"+(System.currentTimeMillis()-stime));

}

}

* + 1. 两者比较

使用聚合代理的方式相比继承代理的方式会更加灵活一些，如果使用继承方式去实现不同的加强需要重写更多的类

package com.learn.proxy;

public class CarProxy03 implements Moveable {

Moveable mo;

CarProxy03(Moveable mo){

this.mo=mo;

}

public static void main(String[] args) {

Moveable mov = new CarProxy03(new Car());

new CarProxy02(mov).move();

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

mov = new CarProxy02(new Car());

new CarProxy03(mov).move();

}

@Override

public void move() {

System.out.println("日志始......");

mo.move();

System.out.println("日志完......");

}

}

* 1. 动态代理

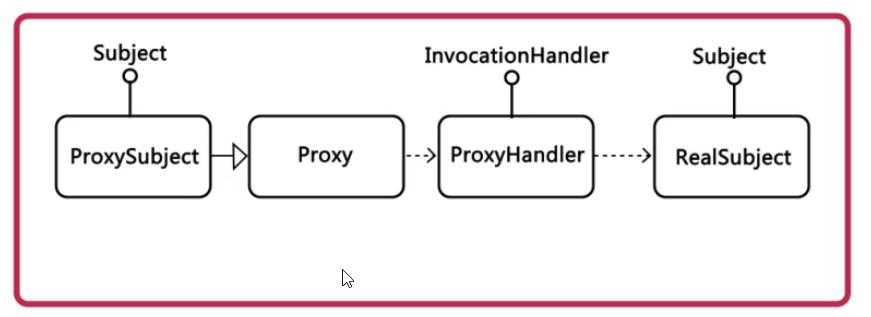
Java动态代理类位于java.lang.reflect包下，一般主要涉及到以下两个类：

1.Interface InvocationHandler：该接口中仅定义了一个方法

Public object invoke(Object obj,Mehtod method,Object[] args)在实际使用时，第一个参数obj一般是指代理类，method是被代理的方法，args为该方法的参数数组。这个抽象方法在代理类中动态实现。

2.Proxy：该类即为动态代理类

Static Object newProxyInstance(ClassLoader loader,Class[] interfaces,InvocationHandler h)：返回代理类的一个实例，返回后的代理类可以当作被代理类使用（可使用被代理类的在接口中声明过的方法）。



package com.learn.proxy;

import java.lang.reflect.InvocationHandler;

import java.lang.reflect.Method;

import java.lang.reflect.Proxy;

public class TimeHandler implements InvocationHandler {

Object target;

TimeHandler(Object target){

this.target = target;

}

public static void main(String[] args) {

Car car = new Car();

InvocationHandler h = new TimeHandler(car);

/\*\*

\* loader 类加载器

\* interfaces 实现接口

\* h InvocationHandler

\*/

Moveable m = (Moveable) Proxy.newProxyInstance(car.getClass().getClassLoader(),

car.getClass().getInterfaces(), h);

m.move();

}

/\*\*

\* 参数：

\* proxy 被代理对象

\* method 被代理对象的方法

\* args 方法的参数

\*/

@Override

public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {

long stime = System.currentTimeMillis();

method.invoke(target);

System.out.println("汽车行驶"+(System.currentTimeMillis()-stime)+"毫秒");

return null;

}

}

使用cglib实现动态代理

引入jar包

<!-- https://mvnrepository.com/artifact/cglib/cglib-nodep -->

<dependency>

<groupId>cglib</groupId>

<artifactId>cglib-nodep</artifactId>

<version>3.2.5</version>

</dependency>

package com.learn.proxy;

import java.lang.reflect.Method;

import net.sf.cglib.proxy.Enhancer;

import net.sf.cglib.proxy.MethodInterceptor;

import net.sf.cglib.proxy.MethodProxy;

public class TrainProxyCglib implements MethodInterceptor {

private Enhancer enhancer = new Enhancer();

public Object getProxy(Class<Train> clazz){

//为哪一个类来创建代理类

enhancer.setSuperclass(clazz);

enhancer.setCallback(this);

return enhancer.create();

}

public static void main(String[] args) {

TrainProxyCglib proxy = new TrainProxyCglib();

Train t = (Train) proxy.getProxy(Train.class);

t.move();

}

/\*\*

\* 拦截所有目标类方法的调用

\* arg0 目标类的实例

\* arg1 目标方法的反射对象

\* arg2 方法的参数

\* arg3代理类的实例

\*/

@Override

public Object intercept(Object arg0, Method arg1, Object[] arg2, MethodProxy arg3) throws Throwable {

System.out.println("日志处理开始。。。");

arg3.invokeSuper(arg0, arg2);

System.out.println("日志处理结束。。。");

return null;

}

}

动态代理实现：

通过Proxy的newProxyInstance返回代理对象

1. 声明一段源码，用于动态产生代理
2. 编译源码，产生新的代理类
3. 将这个类Load到内存当中，产生一个新的对象（代理对象）
4. Return代理对象
   1. 责任链模式

将接收者对象连成一条链，并在该链上传递请求，直到有一个接收者对象处理它，通过让更多对象有机会处理请求，避免了请求发送者和接收者之间的耦合。这种类型的设计模式属于行为型模式。

package com.learn.chain;

import java.util.Optional;

public abstract class Discountable {

public Optional<Discountable> nextConductor = Optional.empty();

protected double right;

void setNextConductor(Discountable nextConductor){

this.nextConductor = Optional.of(nextConductor);

}

protected boolean dodiscount(double dis){

if(dis>getRight()){

System.out.println(this.getClass().getName()+"处理了业务");

return true;

}else if(nextConductor.isPresent()) {

return nextConductor.get().dodiscount(dis);

}

return false;

}

public double getRight() {

return right;

}

}

package com.learn.chain;

public class Ceo extends Discountable {

double right = 0.5D;

public double getRight() {

return right;

}

}

package com.learn.chain;

public class Manager extends Discountable {

double right = 0.7D;

public double getRight() {

return right;

}

}

package com.learn.chain;

public class Seller extends Discountable {

double right = 0.9D;

public double getRight() {

return right;

}

}

package com.learn.chain;

public class Main {

public static void main(String[] args) {

System.out.println(new Main().getD().dodiscount(1.98));

}

private Discountable getD(){

Ceo ceo = new Ceo();

Manager manager = new Manager();

Seller seller = new Seller();

seller.setNextConductor(manager);

manager.setNextConductor(ceo);

return seller;

}

}

* 1. 观察者模式

观察者模式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。在模式中，订阅者被称观察者Observer 订阅者被称观察者Subject.

优点：

观察者模式实现了观察者和目标之间的抽象耦合

观察者模式实现了动态联动

观察者模式支持广播通信



实现：

1. 定义目标对象
2. 具体目标对象的定义
3. 观察者接口定义

package com.learn.observer;

import java.util.ArrayList;

import java.util.List;

/\*\*

\* 目标对象，它知道观察它的观察者，并提供注册（添加）和删除观察者的接口

\*/

public abstract class Subject {

//用于保存注册的观察者对象

private List<Observer> observerList = new ArrayList<Observer>();

//添加观察者

public void attach(Observer observer){

observerList.add(observer);

}

//移除观察者

public void detach(Observer observer){

observerList.remove(observer);

}

//通知观察者

protected void notifyObservers(){

observerList.forEach(o->{

o.update(this);

});

}

}

package com.learn.observer;

/\*\*

\* 这是一个观察者接口，定义一种更新的接口给那些在目标发生改变的时候被通知的对象

\*/

public interface Observer {

//更新接口

public void update(Subject subject);

}

package com.learn.observer;

public class ConcreteSubject extends Subject {

private String subjectCont;

public String getSubjectCont() {

return subjectCont;

}

public void setSubjectCont(String subjectCont) {

this.subjectCont = subjectCont;

this.notifyObservers();

}

}

package com.learn.observer;

public class ConcreteObserver implements Observer {

@Override

public void update(Subject subject) {

System.out.println(((ConcreteSubject)subject).getSubjectCont());

}

}

package com.learn.observer;

public class Main {

public static void main(String[] args){

ConcreteObserver observer1 = new ConcreteObserver();

ConcreteSubject concreteSubject = new ConcreteSubject();

concreteSubject.attach(observer1);

concreteSubject.setSubjectCont("abcd");

concreteSubject.setSubjectCont("defg");

}

}

* 1. 空对象模式

空对象模式，用一个空对象取代NULL对象实例的检查，NULL对象不是检查空值，而是反应一个不做任何动作的关系，这样NULL对象也可以在数据不可用的时候提供默认的行为。

public abstract class AbstractCustomer {

protected String name;

public abstract boolean isNil();

public abstract String getName();

}

public class RealCustomer extends AbstractCustomer {

public RealCustomer(String name) {

this.name = name;

}

@Override

public String getName() {

return name;

}

@Override

public boolean isNil() {

return false;

}

}

public class NullCustomer extends AbstractCustomer {

@Override

public String getName() {

return "Not Available in Customer Database";

}

@Override

public boolean isNil() {

return true;

}

}

public class NullPatternDemo {

public static void main(String[] args) {

AbstractCustomer customer1 = CustomerFactory.getCustomer("Rob");

AbstractCustomer customer2 = CustomerFactory.getCustomer("Bob");

AbstractCustomer customer3 = CustomerFactory.getCustomer("Julie");

AbstractCustomer customer4 = CustomerFactory.getCustomer("Laura");

System.out.println("Customers");

System.out.println(customer1.getName());

System.out.println(customer2.getName());

System.out.println(customer3.getName());

System.out.println(customer4.getName());

}

}

1. JVM原理之Call\_stub

Return CAST\_TO\_EN\_PTR(CallStub,\_call\_stub\_entry);

进行宏替换后得到展开：

Return (CallStub)(castable\_address(\_call\_stub\_entry);

其中(\*CallStub)定义如下

Typedef void (\*CallStub)(

Address link,

Intptr\_t\* result,

BasicType result\_type,

methodOopDesc\* method,

address entry\_point,

intptr\_t\* parameters,

int size\_of\_parameters,

TRAPS

);

|  |  |  |  |
| --- | --- | --- | --- |
| Typedef void (\*CallStub)( --参数说明N=4) | | | |
| (address)&link | 连接器 | 2N(%EBP) | Address link(rbp,2N) |
| Result\_val\_address | 返回地址 | 3N(%EBP) | Address returnAddress(rbp,3N) |
| Result\_Type | 返回类型 | 4N(%EBP) | Address resultType(rbp,4N) |
| Method() | JAVA方法的内部对象 | 5N(%EBP) | Address method(rbp,5N) |
| Entry\_point | 方法调用入口例程 | 6N(%EBP) | Address entryPoint(rbp,6N) |
| Args->parameters() | 方法入参 | 7N(%EBP) | Address parameters(rbp,7N) |
| Args->size\_of\_parameters() | 入参数量 | 8N(%EBP) | Address oaranetersSuze(rbp,8N) |
| CHECK | 当前线程 | 9N(%EBP) | Address thread(rbp,9N) |

CallStub call\_stub(){

Return (CallStub)(castable\_address(\_call\_stub\_entry);

}

JVM隐式的调用了函数指针，返回了一个函数指针的实例变量

* 1. Castable\_address()定义：

Inline address\_word castable\_address(address x){

Return address\_word(x);

}

而address\_word又是啥呢？

Typedef uintptr\_t address\_word

我们知道了address\_word就是unitptr\_t，而address\_word是jvm定义的，该类型与平台是有关的

|  |  |
| --- | --- |
| GlobalDefinitions\_gcc.hpp | Linux系统 |
| GlobalDefinitions\_sparcWorks.hpp | Macintosh |
| GlobalDefinitions\_visCPP.hpp | windows |

Depedef unsigned int uintptr\_t

原来address\_word就是一个unsigned int

* + 1. 连接器Link

连接器起到连接、桥梁的作用，他属于javaCallWrapper 其定义如下：

Class JavaCallWrapper: StruckObj{

Friend class VMStruct;

Private:

JavaThread\* \_thread; --当前JAVA函数所在线程

JNIHandleBlock\* \_handles; --本地调用句柄

methodOop \_callee \_method; --调用者方法对象

oop \_receiver; --被调用者（非静态方法）

JavaFrameAnchor \_anchor; --JAVA线程堆栈对象

JavaValue\* \_result; --方法所返回的值

}

从参数可知，link其实在JAVA函数的调用者与被调用者之间做连接，通过这个桥梁，我们可以实现堆栈追踪。在JAVA方法调用时，link指针将被保存到当前方法的堆栈中。

* + 1. Method()

Method()是java方法在JVM内部的表示对象

每一个JAVA方法在被JVM加载时，JVM都会在内部为这个JAVA方法建立函数模型，说白了就是保存一份JAVA方法的全部原始描述信息，JVM为JAVA方法所建立的模型中至少包含以下信息：

JAVA函数的名称，所属的JAVA类

JAVA函数的入参信息，包括入参类型，入参参数名，入参数量，参数顺序等

JAVA函数编译后的字节码信息，包括对应的字节码指令，所占用的部字节数等

JAVA函数的注解信息

JAVA函数的继承信息

JAVA函数的返回信息

JVM在调用CallStub()函数指针时，将method()对象传递进去，最终就是为了能从method()对象中获取java函数的字节码信息，JVM拿到字节码信息后，就能对字节码进行解释执行了。这也是JAVA可以实现反映的基础。

* + 1. Entry\_point

在JVM通过\_call\_stub\_entry所指向的函数调用JAVA函数之前，必须要经过Entry\_point例程，entry\_point例程会从method()中拿到函数编译后的字节码，然后才可以开始执行。

* + 1. Parameters()

在JVM真正调用JAVA函数的第一个字节码指令之前，JVM会在CallStub()函数中解析JAVA函数的入参，解析后JVM会为JAVA函数分配堆栈，并将参数逐个入栈

* + 1. Size\_of\_parameters()

Parameters()保存了基本参数，JAVA是面向对象的语言，所以参数真实保存了一序列对象的指针数组，通过内部指针，我们根本没有办法获取其结束位置，所以添加size\_of\_parameters()来确定参数数量，以便为函数分配堆栈。

* 1. \_call\_stub\_entry例程

StubRoutines::\_call\_stub\_entry = generate\_call\_stub(StubRoutines::\_call\_stub\_return\_address);

\_call\_stub\_entry顾名思义就是函数的首地址，而在generate\_call\_stub里又做了什么呢？

* + 1. Pc()

函数的第一行就是address\_start = \_pc();

而pc函数定义如下：

Address pc() const{

Return \_code\_pos;

}

JVM启动过程中，会启用非常多的例程（即一段固定的机器指令，也就是一个功能函数），例如：函数调用，字节例程，异常处理，函数返回等等。

第一个例程，一开始都有这么一行address start = \_pc(),当线程开始启动的时候，函数的返回值是0，随着函数的调用深度，这个值将会增加，这也是函数划分内存堆栈的机制，他们连成一块将是一块连续的空间。

* + 1. 定义入参

调用需要8个参数，而这8个参数使用Address link(rbp,2N)定义

N = sizeof(char\*);

如前表，定义了参数位置

* + 1. CallStub保存调用者堆栈

Generate\_call\_sub()函数的逻辑部分将从\_enter()开始，这行代码，在不同的硬件平台上，对应不同的机器指令，即保存ebp esp

* + 1. CallStub动态分配堆栈
    2. Callstub调用者保存

在每次在执行函数调用时，CS:IP寄存器会从当前调用者函数的机器指令处跳转到被调用函数， 这样CPU才能执行被调用的函数，但是当被调用函数执行完了之后，CPU需要跳转到调用者函数中继续执行下一行指令，也就是说需要恢复CS:IP值。物理机器通过CS:IP来区分一个内存中的数据到底是指令还是数据，而对于数据，物理机器一般会用edi,esi分别保存目的地址和源地址，而在JVM中，edi,esi却被赋予了更多职责。在JAVA函数调用过程中，esi会用于java了字节码寻址。每当JVM开始执行JAVA函数的某个字节码指令时，JVM会首先将ESI寄存器指令止标字节码指令的偏移地址，然后JVM跳转到该字节码所对应的每一个机器指令开始执行。所以EDI和ESI中是与调用者紧密关联的寄存器，是调用者函数的私有数据。

EBX是一个通用的寄存器，但是也经常被用来作为一段数据的基地址，例如使用汇编对一个一维数组的成员元素进行寻址，可以被EBX定位到这个一维数组的起始地址，然后使用变址定位到某一个元素，在JVM中EBX被赋予了这种非常实际的的作用。在JAVA函数调用时，EBX会用来存放JAVA函数中即将被执行的字节码指令的基地址，然后通过jmp指令跳转到该字节码位置进行字节码解释执行。因此EBX也会与EDI,ESI一样，与调用者函数息息相关，也是调用者函数的私有数据。

既然ESI,EDI,EBX都是属性调用者函数的私有数据，那么在发生函数调用前必须保存这些数据，因为在被调用者函数中，这些数据也会被调用函数所使用，其中的数据也会被调用函数修改，这样调用函数执行完毕，程序流重新回到调用者函数中，如果调用函数之前没有保存这些数据，这些数据又无法恢复，从而程序将发生异常。

ESI、EDI、DBX的保存并不是必须的，因为在很多编译器只会使用有限的寄存器保存函数特有的数据，而不会使用ESI,EDI,EBX这三类寄存器。保存的方式也很多，由于EDI、ESI、EBX可以看作是函数的私有数据，因此JVM直接保存到了被调用者函数的堆栈中。对于这一点，只是约定俗成的做法。

* + 1. CallStub 参数压栈

动态分配堆栈时，堆栈大小=java函数入参数量\*4+4\*4

最后4个保存的就是RDI,RSI,RBX,MXCSR这四个寄存器所占用的堆栈空间大小。

* + 1. 调用entry\_point例程

1. JVM原理之.class
   1. 字节码结构

Class文件是一组以8位字节为基础单位的二进制流。JAVA虚拟机规范规则，Class文件格式采用一种类似于C语言结构体的伪结构来存储，这种伪结构中只有两种数据类型，无符号数和表。无符号数属于基础数据类型，以u1,u2,u4,u8来分别代表不同字节长度的无符号数。无符号数可以用来描述数字、索引引用、数量值或按UTF8编码构成的字符串。表是由多个无符号数或其它表作为数据项构成的复合数据类型，所有表都习惯性地以“\_info”结尾。表用于描述有层次关系的复合结构的数据，整个Class文件也就是一张表。

Class文件可分为十部分

* + 1. 模数

.class文件有4个字节的模数（Magic Number）0xcafebabe

* + 1. 版本

紧接着订数的4个字节存储的是class文件的主版本号与次版本号

Minor Version: class文件第5-6字节

Major Version:class文件第7-8字节

* + 1. 常量池

紧接着版本之后的就是常量池，他可以理解为class文件的资源库。由于常量池中常量的数量不是固定的，所以在常量池的入口有一个u2类型的数据，代码常量池的个数

常量池之中主要存放两大类常量：字面量和符号引用。字面量比较接近于JAVA语言层面的常量概念，而符号引用是属于编译原理方面的概念，包括了三类常量：

类和接口的全限定名

字段的名称和描述符

方法的名称和描述符

常量池中的每一项常量都有一个表，共有十四种结构各不相同的表结构数据，它们的开始的第一位都是一个u1类型的标志位。代表当前这个常量属于哪种常量类型。

常量池的项目类型

常量池项目类型

|  |  |  |
| --- | --- | --- |
| 类型 | 标志 | 描述 |
| CONSTANT\_UTF8\_info | 1 | UTF8编码的字符串 |
| CONSTANT\_INTEGER\_info | 3 | 整型字面值 |
| CONSTANT\_FLOAT\_info | 4 | 浮点型字面值 |
| CONSTANT\_LONG\_info | 5 | 长整型字面值 |
| CONSTANT\_DOUBLE\_info | 6 | 双精度浮点型字面值 |
| DONSTANT\_CLASS\_info | 7 | 类或接口的符号引用 |
| DONSTANT\_STRING\_info | 8 | 字符器类型字面值 |
| CONSTANT\_FIELDREF\_info | 9 | 字段的符号引用 |
| CONSTANT\_METHODREF\_info | 10 | 类中方法的符号引用 |
| CONSTANT\_INTERFACEMETHODREF\_info | 11 | 接口中方法的符号引用 |
| CONSTANT\_NAMEANDTYPE\_info | 12 | 字段或方法的部分符号引用 |
| CONSTANT\_METHODHANDLE\_info | 15 | 方法句柄 |
| CONSTANT\_METHODTYPE\_info | 16 | 方法类型 |
| CONSTANT\_INVOKEDYNAMIC\_info | 18 | 一个动态方法调用点 |

常量项的结构总表

|  |  |  |  |
| --- | --- | --- | --- |
| 常量 | 项目 | 类型 | 描述 |
| CONSTANT\_UTF8\_info | Tag | U1 | 值为1 |
| Length | U2 | 占用字节数 |
| Bytes | U1 | 长度为Length的UTF8编码字符串 |
| CONSTANT\_  INTEGER\_info | Tag | U1 | 值为3 |
| Bytes | U4 | 按照高位在前存储Int |
| CONSTANT\_  FLOAT\_info | Tag | U1 | 值为4 |
| Bytes | U4 | 按照高位在前存储fllat |
| CONSTANT\_  LONG\_info | Tag | U1 | 值为5 |
| Bytes | U8 | 按照高位在前存储long |
| CONSTANT\_  DOUBLE\_info | Tag | U1 | 值为6 |
| Bytes | U8 | 按照高位在前存储double |
| CONSTANT\_  CLASS\_info | Tag | U1 | 值为7 |
| Index | U2 | 值向字符串字在值的索引 |
| CONSTANT\_  STRING\_info | Tag | U1 | 值为8 |
| Index | U2 | 指向字符串字面量的索引 |
| CONSTANT  \_FIELDREF\_info | Tag | U1 | 值为9 |
| Index | U2 | 指向声明字符的类或者接口描述符的索引 |
| Index | U2 | 指向字段描述符的索引 |
| CONSTANT\_  METHODREF\_info | Tag | U1 | 值为10 |
| Index | U2 | 指向声明方法的类描述符的索引 |
| Index | U2 | 指向名称及类型描述符的索引 |
| CONSTANT\_  INTEFFACEMETHOD\_  Info | Tag | U1 | 值为11 |
| Index | U2 | 指向声明方法的接口描述符的索引 |
| Index | U2 | 指向名称及类型描述符的索引 |
| CONSTANT\_  NAMEANDTYPE\_info | Tag | U1 | 值为12 |
| Index | U2 | 指向该字段或方法名称常量的索引 |
| Index | U2 | 指向该字段或方法描述符常量项的索引 |
| CONSTANT\_  METHODHANDLE\_info | Tag | U1 | 值为15 |
| Reference\_kind | U1 | 取值:1≤值≤9 ,它决定了句柄的类型，表示字节码行为 |
| Reference\_index | U2 | 值必须是对常量池的有效索引 |
| CONSTANT\_  METHODTYPE\_info | Tag | U1 | 值为16 |
| Descriptor\_index | U2 | 值必须是对常量池的有效索引，常量池在该 索引处的项必须是CONSTANT\_UTF8\_info结构，表示方法的描述符 |
| CONSTANT\_  INVOKEDYNAMIC\_  info | Tag | U1 | 18 |
| Bootstrap\_method  \_attrindex | U2 | 值必须是对当前class文件中引导方法表的bootstrap\_method[]数组的有效索引 |
| Name\_and  \_type\_index | U2 | 值必须是对当前常量池的有效索引，常量池在该索引处的项必须是CONSTANT\_NAMEANDTYPE\_info结构，表示方法名和方法描述符。 |

* + 1. 访问标志

常量池结束后，紧接着的是两个字节代表访问标志（access\_flags）,这个标志用于识别一些类或接口层次的访问信息，包括：这个class是类还是接口，修饰内容

访问标志字段值

|  |  |  |
| --- | --- | --- |
| 标志 | 值 | 含义 |
| ACC\_PUBLIC | 0X0001 | 是否为Public |
| ACC\_FINAL | 0X0010 | 是否被声明为FINAL |
| ACC\_SUPER | 0X0020 | 是否允许使用invokespecial字节码指令的新语义，JDK1.0.2之后编译出来的都是真 |
| ACC\_INTERFACE | 0X0200 | 标志这是一个接口 |
| ACC\_ABSTRACT | 0X0400 | 是否为abstract类型，对于接口或抽象类都是真 |
| ACC\_SYNTHETIC | 0X1000 | 标识这个类并非由用户代码产生 |
| ACC\_ANNOTATIION | 0X2000 | 标识这是一个注解 |
| ACC\_ENUM | 0X4000 | 标识这是一个枚举 |

* + 1. 本类索引

两个字节表示本类的全局限定名

0x0003表示类索引为3 #3

* + 1. 父类索引

两个字节表示父类的全局限定名

0X0009表示父类索引为9 #9

* + 1. 接口索引

头两个字节表示接口数量，以后字段为接口描述

#0000表示索引集合大小为0

* + 1. 字段表

字段表用于描述接口或类中声明的变量。字段包括了类级变量或实例变量，它可以包括的信息有：字段的作用域，类级变量还是实例级变量，可变性，并发可见性，可否序列化，字段数据类型，字段名称。这些信息各个修饰符都是布尔值，而字段名、字段类型是无法固定的，只能引用常量池中的常量来描述。

在字段的开始位置，第一个U2类型的数据为容量计数器fields\_count，0x0003表示有3个字段。

字段表结构

|  |  |  |
| --- | --- | --- |
| 类型 | 名称 | 数量 |
| U2 | Access\_flags | 1 |
| U2 | Name\_index | 1 |
| U2 | Descriptor\_index | 1 |
| U2 | Attributes | 1 |
| Attribute\_info | Attributes | Attributes\_count |

字段访问标志

|  |  |  |
| --- | --- | --- |
| 标志 | 标志值 | 含义 |
| ACC\_PUBLIC | 0X0001 | PUBLIC修饰 |
| ACC\_PRIVATE | 0X0002 | PRIVATE修饰 |
| ACC\_PROTECTED | 0X0004 | PROTECTED修饰 |
| ACC\_STATIC | 0X0008 | STATIC修饰 |
| ACC\_FINAL | 0X0010 | FINAL修饰 |
| ACC\_VOLATILE | 0X0040 | VOLATILE修饰 |
| ACC\_TRANSIENT | 0X0080 | STRANSIENT修饰 |
| ACC\_SYNTHETIC | 0X1000 | 是否由编译器自动产生 |
| ACC\_ENUM | 0X4000 | 是否ENUM |

描述符标识

|  |  |  |  |
| --- | --- | --- | --- |
| 标识符 | 含义 | 标识符 | 含义 |
| B | 基本byte | J | 基本long |
| C | 基本char | S | 基本short |
| D | 基本double | Z | 基本boolean |
| F | 基本float | V | 特殊类型void |
| I | 基本int | L | 对象类型 |

对于数组类型，每一个维度都会使用一个前置“[”字符来描述。

* + 1. 方法表

Class文件存储格式中对方法的描述与对字段的描述几乎采用了完全一致的方式，方法表的结构如同字段表一样，依次包括了访问标志，名称索引，描述符索引，属性表集合几项。

|  |  |  |
| --- | --- | --- |
| 类型 | 名称 | 数量 |
| U2 | Access\_flags | 1 |
| U2 | Name\_index | 1 |
| U2 | Descriptor\_index | 1 |
| U2 | Attributes\_count | 1 |
| Attribute\_info | Attributes | Attributes\_count |

方法表访问标志

|  |  |  |
| --- | --- | --- |
| 标志名称 | 标志值 | 含义 |
| ACC\_PUBLIC | 0X0001 | PUBLIC修饰 |
| ACC\_PRIVATE | 0X0002 | PRIVATE修饰 |
| ACC\_PROTECTED | 0X0004 | PROTECTED修饰 |
| ACC\_STATIC | 0X0008 | STATIC修饰 |
| ACC\_FINAL | 0X0010 | FINAL修饰 |
| ACC\_SYNCHRONIZED | 0X0020 | SYNCHRONIZED修饰 |
| ACC\_BRIDGE | 0X0040 | 是否由编译器产生的桥接方法 |
| ACC\_VARARGS | 0X0080 | 是否接收不定参 |
| ACC\_NATIVE | 0X0100 | 是否NATIVE方法 |
| ACC\_ABSTRACT | 0X0400 | ABSTRACT修饰 |
| ACC\_STRICTFP | 0X0800 | 是否为strictfp |
| ACC\_SYNTHETIC | 0X1000 | 是否由编译器自动产生 |

* + 1. 属性表

属性表结构

|  |  |  |
| --- | --- | --- |
| 类型 | 名称 | 数量 |
| U2 | Attribute\_name\_index | 1 |
| U2 | Attribute\_length | 1 |
| U1 | Info | Attribute\_length |

Code属性表结构

|  |  |  |
| --- | --- | --- |
| 类型 | 名称 | 数量 |
| U2 | Attribute\_name\_index | 1 |
| U4 | Attribute\_length | 1 |
| U2 | Max\_stack | 1 |
| U2 | Max\_locals | 1 |
| U4 | Code\_length | 1 |
| U1 | Code | Code\_length |
| U2 | Exception\_table\_length | 1 |
| Exception\_info | Exception\_table | Exception\_table\_length |
| U2 | Attributes\_count | 1 |
| Attribute\_info | Attributes | Attributes\_count |

Attribute\_name\_index是一项指向UTF8\_info型常量的索引，常量固定为Code，它代表了该属性的属性名称，attribute\_length指示了属性值的长度，由于属性名称索引与属性长度一共是6个字节，所以属性值的长度固定为整个属性表的长度减去6个字节。

Max\_stack代表了操作数栈深度的最大值，在方法执行的任意时刻，操作数栈都不会操作这个深度，虚拟机运行的时候需要根据这个值来分配栈帧的操作栈深度。

Max\_locals代表了局部变量表所需要的存储空间。这里max\_local的单位是slot，Slot是虚拟机为局部变量分配内存所使用的最小单位。对于不超过32位的数据类型，每个局部变量占用1个Slot，而double和long这两种64位数据类型需要2个Slot来存放。方法参数、包括隐藏参数this，显示异常处理时参数、方法体中定义的局部变量都需要使用局部变量表来存放，另外并不是在方法中用到了多少个局部局部变量，就把这些局部变量所占的Slot之和作为max\_locals的值，原因是局部变量表中的Slot可以重用，当代表执行超过一个局部变量的作用域时，这个局部变量所占的Slot就可以被其它局部变量所使用，编译器会根据变量的作用域来分类Slot并分配各个变量使用，然后计算出Max\_locals的大小。

Code\_length和code用来存储JAVA源程序编译后生成的字节码指令。Code\_length代表字节码长度，code是用于存储字节码指令的一系列字节流。既然名为字节码指令，那么每个指令就可是一个U1类型的单字节，当虚拟机读取到Code中的一个字节码时，就可以相应地找到这个字节码代表的是什么指令，并且可以知道这条指令后面是否需要参数，以及参数应该如何去理解。我们知道一个U1数据类型的取值范围为0x00-0xFF,对应十进制0-255，也就是一共可以表达256条指令。目前JAVA虚拟机已经定义了200条编码值对应的指令含义。关于code\_length虽然是一个u4类型的长度值，理论上最大可以达到2^32-1，但是虚拟机规范中限制了一个方法不允许超过65535条字节码指令，如果超过这个限制，javac将会拒绝编译。

虚拟机规范预定义属性

|  |  |  |
| --- | --- | --- |
| 属性名称 | 使用位置 | 含义 |
| Code | 方法表 | 代码编译成的字节码指令 |
| ConstantValue | 字段表 | Final关键字定义的常量值 |
| Deprecated | 类、方法表、字段表 | 被志明为deprecated的方法和字段 |
| Exceptions | 方法表 | 方法抛出的异常 |
| enclosingMehtod | 类文件 | 仅当一个类为局部类或者匿名类时才拥有这个属性，这个属性用于标识这个类所在的外围方法 |
| InnerClasses | 类文件 | 内部类列表 |
| LineNumberTable | Code属性 | JAVA源码的行号与字节码的指令对应关系 |
| LocalVariableTable | Code属性 | 方法的局部变量描述 |
| StackMapTable | Code属性 | JDK1.6新增，供新的类型检查验证器检查和处理目标方法的局部变量和操作数栈所需要的类型是否匹配 |
| Sinature | 类、方法表、字段表 | JDK1.5新增，这个属性用于支持泛型情况下的方法签名，在JAVA中任何类、接口初始化方法或成员的泛型签名如果包含子类型变量或参数化类型，则Sinature属性会记录泛型签名信息，由于JAVA的泛型彩擦除法实现，在为了避免类型信息被擦除后导致签名混乱，需要这个属性记录泛型中的相关信息。 |
| SourceFile | 类文件 | 记录源文件名称 |
| SourceDebugExtension | 类文件 | JDK1.6新增，用于存储额外的调试信息 |
| Synthetic | 类、方法表、字段表 | 标识方法或字段为编译器自动生成的 |
| LocalVariableTypeTable | 类 | JDK1.5中新增的属性，它使用特征签名代替描述符，是为了引入泛型语法后能描述泛型参数化类型而添加 |
| RuntimeVisibleAnnottations | 类、方法表、字段表 | JDK1.5中新增，为动态注解提供支持，用于注明哪些注解是支行时可见的 |
| RuntimeInvisibleAnnotations | 类、方法表、字段表 | JDK1.5中新增，作用与RuntimeInvisibleAnnotations属性作用刚好相反，用于指明哪些注解是支行时不可见的 |
| RuntimeVasibleParameter  Annotations | 方法表 | JDK1.5新增，与RuntimeVisibleAnnotaions属性类似，中不过作用方法参数 |
| RuntimeInvisibleParameter  Annotations | 方法表 | JDK1.5新增，作用与RuntimeInvixibleAnnotations属性类似，只不过作用对象为方法参数。 |
| AnnotationDefault | 方法表 | JDK1.5新增，用于记录注解类元素的默认值 |
| BootstrapMethods | 类文件 | JDK1.7新增，用于保存invokedynamic指令引用 的引导方法限定符。 |

Exception属性表

Exception属性是在方法表中与Code平级的一项属性。

|  |  |  |
| --- | --- | --- |
| 类型 | 名称 | 数量 |
| U2 | Attribute\_name\_index | 1 |
| U4 | Attribute\_length | 1 |
| U2 | Number\_of\_exceptions | 1 |
| U2 | Exception\_index\_table | Number\_of\_exceptions |

LineNumberTable属性表

LineNumberTable属性用于描述JAVA源码行号与字节码行号（字节码的偏移量）之间的对应关系。它并不是运行时必需的属性。但默认会生成到Class文件中，可以在JAVAC中使用-g:none或-g:lines选项来取消或要求生成这项信息。如果不生成此属性，对程序运行产生的最主要影响就是在抛出异常时，堆栈中将不会显示出错的行号，并且在调试程序的时候无法按照源码来设置断点。

|  |  |  |
| --- | --- | --- |
| 类型 | 名称 | 数量 |
| U2 | Attribute\_name\_index | 1 |
| U4 | Attribute\_length | 1 |
| U2 | Line\_number\_table\_length | 1 |
| Line\_number\_info | Line\_numebr\_table | Line\_number\_table\_length |

Line\_number\_table是一个数量为Line\_number\_table\_length、类型为Line\_number\_info的集合，line\_number\_info包括了start\_pc和line\_number两个u2类型的数据项，前者是字节码行号，后者是JAVA源码行号。

LocalVariableTable属性

LocalVariableTable属性用于描述栈帧中局部变量表中的变量与JAVA源码中定义的变量之间的关系，它不是运行时必需的属性，默认也不会生成到Class文件中，可以在javac中使用-g:none或-g:vars选项来取消或者要求生成这项信息。如果没有生成这项信息，最大影响就是当其它人引用这个方法时，所有的参数名称都将丢失，IDE可能会使用如arg0之类的占位符来代替原有的参数名，这对程序运行没有影响，但会给代码编写带来较大的不便，而且在调试期间调试器无法根据参数名称从运行上下文中获取参数值。其表结构：

|  |  |  |
| --- | --- | --- |
| 类型 | 名称 | 数量 |
| U2 | Attribute\_name\_index | 1 |
| U4 | Attribute\_length | 1 |
| U2 | Local\_variable\_table\_length | 1 |
| Local\_variable\_info | Local\_variable\_table | Local\_variable\_table\_length |

其中local\_variable\_info选项代表了一个本帧与源码中的局部变量的关联，结构如下所示

|  |  |  |
| --- | --- | --- |
| 类型 | 名称 | 数量 |
| U2 | Start\_pc | 1 |
| U2 | Length | 1 |
| U2 | Name\_index | 1 |
| U2 | Descriptor\_index | 1 |
| U2 | Index | 1 |

Start\_pc和length属性分别代表了这个局部变量的生命周期开始的字节码偏移量及其作用范围覆盖的长度，两者结合起来就是这个局部变量在字节码之中的作用域范围。

Name\_index,descriptor\_index都指向常量池中utf8型常量的索引，分别代表了局部变量的名称及该局部变量描述符。Index是这个局部变量在栈局部变量表中Slot的位置。当这个变量的数据类型是64位类型时，它占用的Slot为index和index+1两个位置。

SourceFile属性

SourceFile属性用于记录生成这个Class文件的源码文件名称，这个属性也是可选的，可以使用javac的-g:none或-g:source选项来关闭或要求生成这项信息。在JAVA中，对于大多数类来说，类名和文件名是一致的，但是有一些特殊情况，比如类部类。如果不生成这项属性，当抛出异常时，堆栈中将不会显示出错代表所属的文件名。这个属性是一个定长的属性。

|  |  |  |
| --- | --- | --- |
| 类型 | 名称 | 数量 |
| U2 | Attribute\_name\_index | 1 |
| U4 | Attribute\_length | 1 |
| U2 | Sourcefile\_index | 1 |

Sourcefile\_index指定常量池中UTF8型常量索引。

ConstantValue属性

ConstantValue属性的作用是通知虚拟机自动为静态变量赋值。只有被static关键字修饰的变量才可以使用这项属性。在JAVA程序里，“int x =123”的“static int x=123”这样的变量定义是非常常见的。但虚拟机对这两种变量赋值的方式和时刻都有所不同。对于非static类型的变量的赋值是在实例构造器<init>方法中进行的。而对于类变量，则有两种方式可以选择：赋值在类构造器<init>方法中进行，或者作用ConstantValue属性来赋值。目前sun javac编译器选择的是：如果同时使用final static修饰一个常量并且这个变量的类型是基本数据类型或java.lang.String的话，就生成ConstantValue属性来进行初始化。否则在init方法中初始化。

|  |  |  |
| --- | --- | --- |
| 类型 | 名称 | 数量 |
| U2 | Attribute\_name\_index | 1 |
| U4 | Attribute\_length | 1 |
| U2 | Constantvalue\_index | 1 |

ConstantValue属性是一个定长属性，constant\_index数据选项代表了常量池中一个字面量常量的引用，字面量常量可以是CONSTANT\_Long\_info、CONSTANT\_Float\_info、CONSTANT\_Double\_info、CONSTANT\_Integer\_info、CONSTANT\_String\_info常量中的一种。

InnerClasses属性

InnerClasses属性用于记录内部类与宿主类之间的关联。如果一个类中定义了内部类，那编译器将会为它及所包含的内部类生成InnerClasses属性。

|  |  |  |
| --- | --- | --- |
| 类型 | 名称 | 数量 |
| U2 | Attribute\_name\_index | 1 |
| U4 | Attribute\_length | 1 |
| U2 | Number\_of\_classes | 1 |
| Inner\_classes\_info | Inner\_classes | Numer\_of\_classes |

数据项number\_of\_classes代表需要记录多少个内部类信息，每一个内部类信息都由一个inner\_classes\_info表进行描述。

|  |  |  |
| --- | --- | --- |
| 类型 | 名称 | 数量 |
| U2 | Inner\_class\_info\_index | 1 |
| U2 | Outer\_class\_info\_index | 1 |
| U2 | Inner\_name\_index | 1 |
| U2 | Inner\_class\_access\_flags | 1 |

Inner\_class\_info\_index指向常量池中CONSTANT\_Class\_info型常量索引，代表内部类的符号引用

Outer\_class\_info\_index指向常量池中CONSTANT\_Class\_info型常量索引，代表宿主类的符号引用

Inner\_name\_index指向CONSTATN\_Utf8\_info型常量索引，代表内部类的名称，如果是匿名内部类，则值为0

Inner\_class\_access\_flags内部类的访问标志。

Inner\_class\_access\_flags标志

|  |  |  |
| --- | --- | --- |
| 标志名称 | 标志值 | 含义 |
| ACC\_PUBLIC | 0X0001 | 内部类是否为Public |
| ACC\_PRIVATE | 0X0002 | 内部类是否为private |
| ACC\_PROTECTED | 0x0004 | 内部类是否为protected |
| ACC\_STATIC | 0X0008 | 内部类是否为static |
| ACC\_FINAL | 0X0010 | 内部类是否为final |
| ACC\_INTERFACE | 0X0020 | 内部类是否为synchronized |
| ACC\_ABSTRACT | 0X0400 | 内部类是否为abstract |
| ACC\_SYNTHETIC | 0X1000 | 内部类是否并非由用户代表产生 |
| ACC\_ANNOTATION | 0X1000 | 内部类是否是一个注解 |
| ACC\_ENUM | 0X4000 | 内部类是否是一个枚举 |

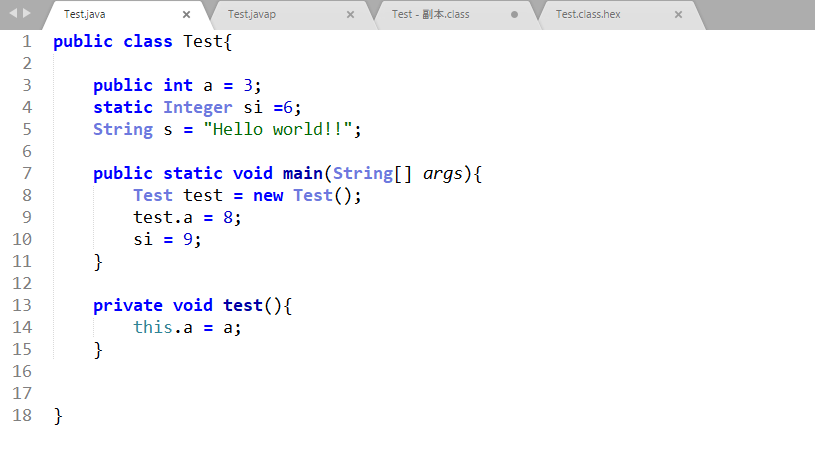
Deprecated及Synthetic属性

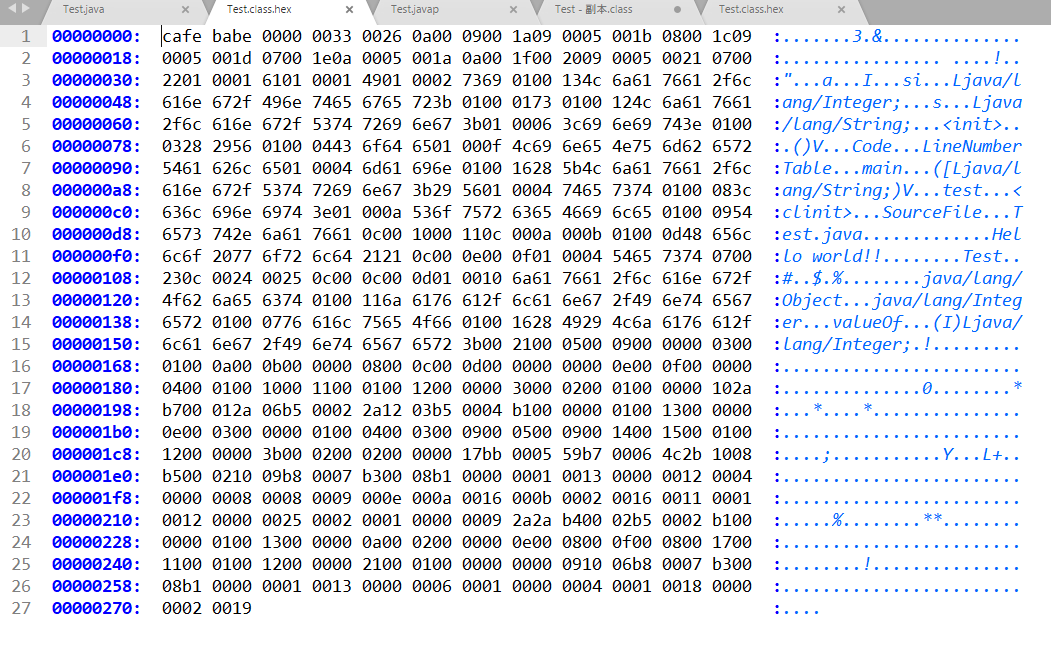
Deprecated和Synthetic属性都属于标志类型的布尔属性，只存在有和没有的区别

|  |  |  |
| --- | --- | --- |
| 类型 | 名称 | 数量 |
| U2 | Attribute\_name\_index | 1 |
| U4 | Attribute\_length | 1 |

Attribute\_length数据项的值必需为0x00000000,因为没有任何属性需要设置

* 1. 解析示例





Javap

Classfile /C:/Users/zeimao77/Desktop/Test.class

Last modified 2017-12-2; size 628 bytes

MD5 checksum 98b54ffcffe2c3aa5fcf0106e2c8ded1

Compiled from "Test.java"

public class Test

SourceFile: "Test.java"

minor version: 0

major version: 51

flags: ACC\_PUBLIC, ACC\_SUPER

Constant pool:

#1 = Methodref #9.#26 // java/lang/Object."<init>":()V

#2 = Fieldref #5.#27 // Test.a:I

#3 = String #28 // Hello world!!

#4 = Fieldref #5.#29 // Test.s:Ljava/lang/String;

#5 = Class #30 // Test

#6 = Methodref #5.#26 // Test."<init>":()V

#7 = Methodref #31.#32 // java/lang/Integer.valueOf:(I)Ljava/lang/Integer;

#8 = Fieldref #5.#33 // Test.si:Ljava/lang/Integer;

#9 = Class #34 // java/lang/Object

#10 = Utf8 a

#11 = Utf8 I

#12 = Utf8 si

#13 = Utf8 Ljava/lang/Integer;

#14 = Utf8 s

#15 = Utf8 Ljava/lang/String;

#16 = Utf8 <init>

#17 = Utf8 ()V

#18 = Utf8 Code

#19 = Utf8 LineNumberTable

#20 = Utf8 main

#21 = Utf8 ([Ljava/lang/String;)V

#22 = Utf8 test

#23 = Utf8 <clinit>

#24 = Utf8 SourceFile

#25 = Utf8 Test.java

#26 = NameAndType #16:#17 // "<init>":()V

#27 = NameAndType #10:#11 // a:I

#28 = Utf8 Hello world!!

#29 = NameAndType #14:#15 // s:Ljava/lang/String;

#30 = Utf8 Test

#31 = Class #35 // java/lang/Integer

#32 = NameAndType #36:#37 // valueOf:(I)Ljava/lang/Integer;

#33 = NameAndType #12:#13 // si:Ljava/lang/Integer;

#34 = Utf8 java/lang/Object

#35 = Utf8 java/lang/Integer

#36 = Utf8 valueOf

#37 = Utf8 (I)Ljava/lang/Integer;

{

public int a;

flags: ACC\_PUBLIC

static java.lang.Integer si;

flags: ACC\_STATIC

java.lang.String s;

flags:

public Test();

flags: ACC\_PUBLIC

Code:

stack=2, locals=1, args\_size=1

0: aload\_0

1: invokespecial #1 // Method java/lang/Object."<init>":()V

4: aload\_0

5: iconst\_3

6: putfield #2 // Field a:I

9: aload\_0

10: ldc #3 // String Hello world!!

12: putfield #4 // Field s:Ljava/lang/String;

15: return

LineNumberTable:

line 1: 0

line 3: 4

line 5: 9

public static void main(java.lang.String[]);

flags: ACC\_PUBLIC, ACC\_STATIC

Code:

stack=2, locals=2, args\_size=1

0: new #5 // class Test

3: dup

4: invokespecial #6 // Method "<init>":()V

7: astore\_1

8: aload\_1

9: bipush 8

11: putfield #2 // Field a:I

14: bipush 9

16: invokestatic #7 // Method java/lang/Integer.valueOf:(I)Ljava/lang/Integer;

19: putstatic #8 // Field si:Ljava/lang/Integer;

22: return

LineNumberTable:

line 8: 0

line 9: 8

line 10: 14

line 11: 22

static {};

flags: ACC\_STATIC

Code:

stack=1, locals=0, args\_size=0

0: bipush 6

2: invokestatic #7 // Method java/lang/Integer.valueOf:(I)Ljava/lang/Integer;

5: putstatic #8 // Field si:Ljava/lang/Integer;

8: return

LineNumberTable:

line 4: 0

}

解析

【魔数】cafe babe

【主版本】0000

【次版本】0033

【常量池】

【常量池-计数】0026

【常量池-item1-tag】0a 00 09 001a

【常量池-item2-tag】09 0005 001b

【常量池-item3-tag】08 001c

【常量池-item4-tag】09 0005 001d

【常量池-item5-tag】07 001e

【常量池-item6-tag】0a 0005 001a

【常量池-item7-tag】0a 001f 0020

【常量池-item8-tag】09 0005 0021

【常量池-item9-tag】07 0022

【常量池-itema-tag】01 0001 61

【常量池-itemb-tag】01 0001 49

【常量池-itemc-tag】01 0002 7369

【常量池-itemd-tag】01 0013 4c 6a61 7661 2f6c 616e 672f 496e 7465 6765 723b

【常量池-iteme-tag】01 00 01 73

【常量池-itemf-tag】 01 00 12 4c6a 6176 612f 6c61 6e67 2f53 7472 696e 673b

【常量池-item10-tag】01 0006 3c69 6e69 743e

【常量池-item11-tag】01 0003 2829 56

【常量池-item12-tag】01 0004 436f 6465

【常量池-item13-tag】01 000f 4c69 6e65 4e75 6d62 6572 5461 626c 65

【常量池-item14-tag】01 0004 6d61 696e

【常量池-item15】0100 1628 5b4c 6a61 7661 2f6c 616e 672f 5374 7269 6e67 3b29 56

【常量池-item16】01 0004 7465 7374

【常量池-item17】0100 083c 636c 696e 6974 3e

【常量池-item18】01 000a 536f 7572 6365 4669 6c65

【常量池-item19】0100 0954 6573 742e 6a61 7661

【常量池-item1a】0c00 1000 11

【常量池-item1b】0c 000a 000b

【常量池-item1c】0100 0d48 656c 6c6f 2077 6f72 6c64 2121

【常量池-item1d】0c00 0e00 0f

【常量池-item1e】01 0004 5465 7374

【常量池-item1f】0700 23

【常量池-item20】0c 0024 0025

【常量池-item21】0c 000c 000d

【常量池-item22】01 0010 6a61 7661 2f6c 616e 672f 4f62 6a65 6374

【常量池-item23】0100 116a 6176 612f 6c61 6e67 2f49 6e74 6567 6572

【常量池-item24】0100 0776 616c 7565 4f66

【常量池-item25】0100 1628 4929 4c6a 6176 612f 6c61 6e67 2f49 6e74 6567 6572 3b

【访问标识】 00 21

【全局限定名】00 05

【父类的全局限定名】00 09

【接口数量】00 00

【字段】

【字段数量】00 03

【字段-item1】0001 000a 000b 0000

【字段-item2】0008 000c 000d 0000

【字段-item3】0000 000e 000f 0000

【方法】

【方法数量】0004

【方法-item1-access-flags】00 01

【方法-item1-access-nameindex】00 10

【方法-item1-access-descriptorindex】00 11

【方法-item1-attributecount】00 01

【方法-item1-code-attributeNameIndex】00 12

【方法-item1-code-attributeLength】00 0000 30

【方法-item1-code-maxStack】0002

【方法-item1-code-maxlocals】0001

【方法-item1-code-codeLength】00 0000 10

【方法-item1-code-code】2ab7 0001 2a06 b500 022a 1203 b500 04b1

【方法-item1-code-exceptionTableLength】0000

【方法-item1-code-exceptionTable】

【方法-item1-code-attributeCount】00 01

【方法-item1-code-attribute】00 13

0013表明指向常量池的第19号常量即 #19 = Utf8 LineNumberTable

由前面分析的lineNumberTable包含4个元素

【方法-item1-code-attribute-nameindex】

【方法-item1-code-attribute-length】0000 000e

表明接下来的十五个属性属于attribute属性，后面的数据不属性此属性

【方法-item1-code-attribute-lineNumberTableLength】00 03

【方法-item1-code-attribute-lineNumberTable】0000 0001 0004 0003 0009 0005

【方法-item2-access-flags】0009

【方法-item2-access-nameindex】00 14

【方法-item2-access-descriptorindex】00 15

【方法-item2-attributecount】00 01

【方法-item2-code-attributeNameIndex】00 12

【方法-item2-code-attributeLength】0000 003b

【方法-item2-code-maxStack】0002

【方法-item2-code-maxlocals】 0002

【方法-item2-code-codeLength】0000 0017

【方法-item2-code-code】bb00 0559 b700 064c 2b10 08b5 0002 1009 b800 07b3 0008 b1

【方法-item2-code-exceptionTableLength】0000

【方法-item2-code-exceptionTable】

【方法-item2-code-attributeCount】0001

【方法-item2-code-attribute】0013

【方法-item2-code-attribute-nameindex】

【方法-item2-code-attribute-length】0000 0012

【方法-item2-code-attribute-lineNumberTableLength】0004

【方法-item2-code-attribute-lineNumberTable】0000 0008 0008 0009 000e 000a 0016 000b

【方法-item3-access-flags】0002

【方法-item3-access-nameindex】0016

【方法-item3-access-descriptorindex】0011

【方法-item3-attributecount】0001

【方法-item3-code-attributeNameIndex】0012

【方法-item3-code-attributeLength】0000 0025

【方法-item3-code-maxStack】0002

【方法-item3-code-maxlocals】 0001

【方法-item3-code-codeLength】0000 0009

【方法-item3-code-code】2a2a b400 02b5 0002 b1

【方法-item3-code-exceptionTableLength】0000

【方法-item3-code-exceptionTable】

【方法-item3-code-attributeCount】0001

【方法-item3-code-attribute】0013

【方法-item3-code-attribute-nameindex】

【方法-item3-code-attribute-length】0000000a

【方法-item3-code-attribute-lineNumberTableLength】00 02

【方法-item3-code-attribute-lineNumberTable】00 0000 0e00 0800 0f

这个方法存储每一个属性的索引：

【方法-item4-access-flags】00 08

【方法-item4-access-nameindex】00 17

【方法-item4-access-descriptorindex】0011

【方法-item4-attributecount】0001

【方法-item4-code-attributeNameIndex】0012

【方法-item4-code-attributeLength】0000 0021

【方法-item4-code-maxStack】0001

【方法-item4-code-maxlocals】 0000

【方法-item4-code-codeLength】0000 0009

【方法-item4-code-code】1006 b800 07b3 0008 b1

【方法-item4-code-exceptionTableLength】0000

【方法-item4-code-exceptionTable】

【方法-item4-code-attributeCount】0001

【方法-item4-code-attribute】0013

【方法-item4-code-attribute-nameindex】

【方法-item4-code-attribute-length】0000 0006

【方法-item4-code-attribute-lineNumberTableLength】0001

【方法-item4-code-attribute-lineNumberTable】0000 0004

【属性-attributes-count】0001

【属性-attributes-nameIndex】0018 ==SourceFile

【属性-attributes-length】0000 0002

【属性-attributes-value】0019 ==sourceFileIndex

1. JVM原理之内存管理

JVM在执行JAVA程序的过程中，会把它所管理的内存划分成若干个不同的数据区，每个区域都有各自的用途，以及创建和销毁的时间。模拟JAVA虚拟机规范的规定，JAVA虚拟机所管理的内存会包括以下几个运行时数据区域。

* 1. 运行时数据区
     1. 方法区

方法区（Method Area）与JAVA堆一样，是各个线程共享的内存区域，它用于存储已被虚拟机加载的类信息，常量，静态变量、即时编译器编译后的代码等数据。JVM规范把方法区作为堆的一个逻辑部分，但它却是一个非堆。

有时也将方法区称为永久代，可以不实现GC,但方法区与永久区不等价，HotSpot虚拟机已将GC扩展到了方法区。这个区域的回收目标主要是针对常量池的回收和类型的卸载，一般来说这个区域的回收“成绩”比较难另人满意，特别是类型的卸载，条件相当苛刻，但这部分区域的回收确实是必要的。在SUM的BUG列表中，曾多个版本由于没有对此区域回收而导致内存泄漏。根据JVM规范规定，当方法区无法满足内存的分配需要时，将抛出OutOfMemoryError异常。

运行时常量池是方法区的一部分，Class文件中除了有类的版本、字段、方法、接口等描述信息外，还有一些信息是常量池用于编译期生成的各种字面量和符号引用，这部分内容将在类加载后进入方法区的支行时常量池中存放。

import java.util.ArrayList;

/\*\*

\* -XX:PermSize=20m -XX:MaxPermSize=20m

\* Java HotSpot(TM) 64-Bit Server VM warning: Ignoring option PermSize; support was removed in 8.0

\* Java HotSpot(TM) 64-Bit Server VM warning: Ignoring option MaxPermSize; support was removed in 8.0

\*/

public class ConstantOom {

public static void main(String args[]){

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

int i = 0;

while (true){

list.add(String.valueOf(i++).intern());

}

}

}

* + 1. 运行时常量池

运行时常量池是方法区的一部分，Class文件中除了有类的版本、字段、方法、接口等描述信息外，还有一项信息就是常量池，用于存放编译期生成的各种字面量和符号引用，这部分内容将在类加载后存放到方法区的运行时常量池中。

Java虚拟机对Class文件的每一部分的格式都有严格的规定，每一个字节用于存放哪种数据都必须符合规范上的要求。但对于运行时常量池，虚拟机并没有细节要求，不同的提供商可以自由实现。

运行时常量池要对于Class文件常量池的另外一个重要特性就是它是动态的，JAVA语言并不要求只能在编译期产生，可以在运行期间将新的常量放入池中，这种特性被开发利用较多的是String类的intern()方法

既然是方法区的一部分，自然会受到方法法内存的限制，当申请不到足够内存时会抛出OutOfMemoryError异常。

* + 1. 虚拟机栈

虚拟机栈是线程私有的，它的生命同期与线程相同，虚拟机栈描述的是JAVA方法执行的内存模型；每个方法在执行的同时都会创建一个栈帧用于存储局部变量表、操作数栈、动态链接、方法出口等信息。每个方法从调用直到执行完成的过程，就对应着一个栈帧在虚拟机栈中入栈到出栈的过程。

局部变量表存放了编译期可知的各种基本数据类型、对象引用、和return Address（指向code的地址）。其中63位长度的Long和double类型的数据会占用两人上局部变量空间（Slot），其余的数据类型只占1个。局部变量表所需的内存空间在编译期间完成分配，当进入一个方法时，所需要分配的空间是完成确定的，在方法运行期间不会改变局部变量表的大小。

这个区域规定了两种异常：如果线程请求的栈深度大于虚拟机所允许的深度抛出StackOverflowError异常（当前大部分JVM都可动态扩展），如果扩展时无法申请到足够的内存，会抛出OutOfMemoryError异常。

* + 1. 本地方法栈

为NATIVE方法服务，虚拟机规范中并没有强制规范，可以自由实现有些虚拟机（HotSpot）直接将本地方法栈与虚拟机栈是合并的。也正因此，-Xoos设置本地方法栈参数无效。都会抛出StackOverflowError和OutOfMemoryError异常。

package com.oom.test;

/\*\*

\* -Xss256k

\* 如果线程请求的栈尝试大于虚拟机所允许的最大深度，将抛出StackOverfolwError异常

\* Exception in thread "main" java.lang.StackOverflowError

\*/

public class StackOOM {

int a;

public static void main(String[] args){

new StackOOM().test();

}

void test(){ a++;test(); }

}

* + 1. 堆

堆内存是JVM所管理的内存中最大的一块，堆内存是所有线程共享的，在虚拟机启动时创建。此内存区域的唯一目的就是存放对象实例，几乎所有的对象都存入在堆中，但是不是所有。JAVA虚拟机规范规定，所有对象实例以及数组都要在堆上分配，但随着技术的发展，也不“绝对”了。

堆是垃圾收集器管理的主要区域。从内存回收的角度来看，现在收集器都采用分代回收算法，所以堆内存还可以分为新生代和老年代；再细致一点有Eden空间、Form Survivor空间、To Survivor空间等。从内存分配角度来看，线程共享的JAVA堆中可能划分出多个线程私有的分配缓冲区。

Java堆内存可以处于物理机上不连续的内存空间中，只要逻辑上连续的就可以了，在实现时即可以按照可扩展来实现，也可以固定大小，并且堆中没有内存完成实例分配，也无法再扩展时会抛出OutOfMemoryError异常。

import java.util.ArrayList;

/\*\*

\* -Xms20m -Xmx20m -XX:+HeapDumpOnOutOfMemoryError

\* Exception in thread "main" java.lang.OutOfMemoryError: Java heap space

\*/

public class HeadOom {

public static void main(String args[]){

ArrayList<OOMObject> list = new ArrayList<OOMObject>();

while(true){ list.add(new OOMObject());}

}

}

class OOMObject{

byte bytes[] = null;

public OOMObject(){ bytes= new byte[1024\*1024]; }

}

* + 1. 程序计数器

程序计数器（Program Counter Retister）是一块较小的内存空间，是当前线程所执行的字节码的行号指示器。由于JAVA虚拟机的多线程是通过线程轮流切换并分配处理器执行时间的方式来实现的，在任何一个确定的时刻，一个处理器都只会执行一条程序中的指令。因此，为了线程切换后能恢复到正确的执行位置，每条线程都需要有一个独立的程序计数器，各条线程之间计数器都互不影响，独立存储，我们这类内存区域为线程私有的内存。如果线程执行的是NATIVE方法，则计数器值为空。此内存区域的唯一一个在JAVA虚拟机规范中没有规定任何OutOfMemoryError情况的区域。

* + 1. 直接内存

直接内存并不是JVM的一部分，也不是JAVA虚拟机规范中定义的内存区域。但是这部分也被频繁的使用，且也会导致OutOfMemoryError异常出现。是JDK1.4引入了NIO类，引入了基于通道与缓冲区的I/O方式，可以使用NATIVE函数库直接分配堆外内存，然后通过一个存储在JAVA堆中的DirectByteBuffer对象作为这块内存的引用进行操作。这样可以避免在JAVA堆和NATIVE堆中来回复制数据。提高性能。本机直接内存分配不会受到JAVA堆的限制，但会受到本机物理硬件或系统的限制。

import sun.misc.Unsafe;

import java.lang.reflect.Field;

public class DirectMemoryOOM {

private static final int \_OME = 1024 \* 1024;

/\*\*、

\* VM Args: -Xmx20m -Xmx20m -XX:MaxDirectMemorySize=10M

\* @param args

\* @throws IllegalAccessException

\*/

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

Field unsafeField = Unsafe.class.getDeclaredFields()[0];

unsafeField.setAccessible(true);

Unsafe unsafe = (Unsafe) unsafeField.get(null);

while (true){

unsafe.allocateMemory(\_OME); //真正分配内存

}

}

}

* 1. 对象
     1. 创建对象

JAVA是一门面向对象的语言，在JAVA语言运行过程中无时无刻都有对象被创建出来。当遇到new指令时，首先将去检查这个指令的参数是否能在常量池中定位到一个类的符号引用，并且检查这个符号引用代表的类是否已经被加载、解析和初始化过，如果没有，那必须先执行相应的类加载过程。

在类加载检查通过后，接下来虚拟机将为新生对象分配内存，对象所需内存的大小在类加载完成后便可以完全确定，为对象分配空间的任务赞同于把一块确定大小的空间从JAVA堆中划分出来。假设JAVA堆内存是绝对规整的，所有用过的内存都放在一边，没有使用的内存放在另一边，中间放着一个指针作为分界点的指示器，那所分配内存仅仅是把指针向空闲空间挪动与对象大小相等的距离即可，这种分配方式称“指针碰撞”如果不是归整的，已使用的内存为未使用的内存相互交错，那就没有办法简单地进制指针碰撞了，虚拟机就必须维护一个列表，记录上哪些内存是可用的，在分配的时候从列表中指到一块足够大的空间划分给对象并更新表记录，这种分配方式称“空闲列表”。选择哪种分配方式由JAVA堆是否归整决定，而JAVA堆是否规整又由所采用的垃圾收集器是否带有压缩整理功能决定。

除如何划分可用空间外，还有另外一个需要考虑的问题是对象创建在虚拟机中是非常频繁的行为，即使是仅仅作一个指针所指向的位置，在并发情况下也并不是线程安全的。可能出现正在给对象A分配内存，指针还没来及修改，对象B又同时使用了原来的指针来分配内存的情况。解决这个问题有两种解决方案，一种是对分配内存空间的动作进行同步处理—实际上虚拟机彩CAS配上失败重试的方式保证更新操作的原子性，另一种是把内存分配 的动作按照线程划分在不同的空间之中进行，即每个线程在JAVA堆中预先分配一小块内存，称本地线程分配缓冲（Thread Local Allocation Buffer,TLAB）。哪个线程要分配内存，就在哪个线程的TLAB上分配，只有TLAB用完并分配新的TLAB时，才需要同步锁定。虚拟机是否使用TLAB可通过-XX:+/-UseTLAB参数来设定。

内存分配完成后，虚拟机需要将分配到的内存空间都初始化为零值（不包括对象头），如果使用TLAB，这一工作过程也可以提前至TLAB分配时进行。这一步操作保证了对象的实例字段在JAVA代码中可以不赋值初始值就直接使用，程序能访问到这些字段的数据类型所对应的零值。

接下来，虚拟机要对对象进行必要的设置，例如对象是哪个类的实例、如何找到元数据、对象的哈希码、GC的分代年龄等信息。这些信息存放在对象的对象头中，根据虚拟机当前的支行状态的不同，是否启用偏向锁等，对对象设置对象头。

以上工作仅仅完成JVM的初始化，一个新的对象已经产生，但从JAVA程序的视角来说，对象创建刚刚开始 <init>方法还没有执行。只有执行<init>方法，对象会按程序员的意愿进行初始化，这样一个对象才算完全产生出来。

* + 1. 内存分配

对象会优先分配在Eden中

大对象直接进入老年代

长期存活的对象进入老年代

GC分为新生代GC(MinorGC),老年代GC(Major GC/Full GC),JAVA对象大多数都不需要长期生存，朝生夕死，所以MinorGC会非常频繁，一般回收速度也非常快，发生在老年代的GC经常伴随至少一次MinorGC,但也不绝对，所以Full GC会比MinorGC慢许多。

虚拟机也并非只有达到MaxTenuringThreshold年龄要求的对象才能晋升到老年代，如果Survivor空间中相同年龄所有对象大小的总和大于Survivor空间的一半，年龄大于或等于该年龄的对象就可以直接进入老年代。

* + 1. 担保

在发生Minor GC时，虚拟机会检测之前每次晋升到老年代的平均大小（经验值）是否大小老年代的剩余空间大小，如果大于，则改为直接进行一次Full GC.如果小于，则查看HandlePromotionFailure设置是否允许担保失败，如果允许进行Minor GC;如果不允许，则改为Full GC

新生代使用的是复制算法，但为了内存利用率，只会使用其中一个Survivor空间来作为轮换备份，因此当出现大量对象在Minor GC后仍然存活时，就需要老年代进行分配担保，让Survivor无法容纳的对象直接进入老年代。因为在GC前根本不知道有多少对象会在此次GC前存活下来，所以只能取平均值来判断。如果空间不够，则只能进行一次FullGC来获取更多空间

* + 1. 对象布局

在HotSpot虚拟机中，对象在内存中存储的布局可以分为三个区域：对象头、实例数据、对齐填充。

HotSpot虚拟机的对象头包括两部分信息，每一部分存储对象自身的支行时数据，如哈希码、GC分代年龄、锁状态、线程持有的锁、移向线程ID、偏向时间戳等。这部分被称为“Mark Word”，Mark Word设计成为一个非固定的数据结构与便在极小的空间中存储尽量多的信息，对象头的另一部分是类型指针，即对象指向它的类元数据指针。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，也就是说查找对象的元数据并不一定要经过对象本身。如果对象是一个数组，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，但从数组的元数据却无法确定数组的大小。

接下来的实例数据部分是对象真正存储的有效信息，也是程序代码中所定义的各类型的字段内容。无论是从父类继承下来的，还是在子类中定义的，都需要记录起来。这部分的存储顺序会受虚拟机分配策略参数和字段在JAVA源中定义顺序的影响。从分配 策略上，相同宽度的字段总是被分配到一起。

第三部分对齐填充并不是必然存在的，也没有特别的含义，它仅仅起着占位符的作用。HotSpot VM的自动内存管理系统要求对象起始地址必须是8个字段的整数倍。

* + 1. 对象访问

目前主流的访问方式有使用句柄和直接指针两种方式。

如果使用句柄访问，JAVA堆中将会划分出一块内存来作为句柄池，reference中存储的就是对象的句柄地址，而句柄中包含了对象实例数据与类型数据各自的具体地址信息。

如果使用直接指针访问，那么JAVA堆对象的布局中就必须考虑如何放置访问类型数据的相关信息，而reference中存储的直接就是对象地址。

这两种访问方式各有优势，使用句柄访问方式最大好处就是reference中存储的是稳定句柄地址，在对象移动（GC时移动是非常普遍的行为）时只会改变句柄中的实例数据指针。使用直接指针方式最大的好处就是速度更快，它节省了需要再一次定位的开销。由于对象的访问频繁，这类开销积少成多也是非常可观的执行成本。

* 1. GC

在垃圾收集器对堆进行回收前，第一件事情就是要确定对象已死去

* + 1. 引用计数法
    2. 可达性分析法

C#,JAVA主流程序语言都是通过可达性分析法来判定对象是否存活的。 这个算法的基本思想就是通过一系列称为“GC ROOTS”的对象作为起始点，从这些节点开始向下搜索，搜索所走过的路径称为引用链，当一个对象到GC ROOTS没有任何引用链相连则证明此对象是不可用的。在JAVA语言中可作为GC ROOTS的对象包括：

虚拟机栈（栈帧中的本地变量表）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JNI（Native方法）引用的对象

* + 1. 再谈引用

在JDK1.2之前，JAVA的引用定义很传统，如果reference类型的数据中存储的数值代表的是另一块内存的起始地址，就称这块内存代表着一个引用，这种定义很纯粹，但太过狭隘，一个对象在这种定义下只有被引用和未被引用两种状态。而对于一些“食之无味，弃之可惜”的对象显得无能为力。我们希望这样描述这样一类对象，当内存空间还足够时，则保留在内存中，如果内存紧张，则抛弃这些对象。JDK1.2后对概念进行扩展，将引用分为强引用、软引用、弱引用、虚引用4种

强引用指在程序中普遍存在的，只要强引用存在垃圾回收就永远不会回收掉被引用的对象。

软引用是用来描述一些还有用但并非必须的对象，对于软引用关联着的对象，在系统将要发生内存溢出异常之前，将会把这些对象列进回收范围之中进行二次回收。如果这次回收还没有足够的内存，才会抛出内存溢出异常。

弱引用也是用来描述非必需对象的，但是它的强度比软引用更弱一些，被弱引用关联的对象只能生存到下一次垃圾收集发生之前，当垃圾收集器工作时，无论当前内存是否足够，都会回收掉只被弱引用关联的对象。WeakReference类来实现弱引用。

虚引用也称为幽灵引用或者幻影引用，它是最弱的一种引用关系，一个对象是否有虚引用的存在，完全不会对其生存时间构成影响，也无法通过虚引用来取得一个对象实例。为一个对象设置虚引用关联的唯一目的就是能在这个对象被收集器回收时收到一个系统通知。

* + 1. 生死

即使在可达性分析算法中不可达的对象，也并非是“非死不可”的。这时候它们暂处于“缓刑”阶段，要真正宣告一个对象死亡，至少要经历两次标记过程：如果对象在进行可达性分析后发现没有与GR ROOTS相连接的引用链，那它将会被被第一次标记并且进行一次筛选，筛选的条件是此对象是否有必要执行finalize方法。当对象没有覆盖finalize()方法，或者finalize()方法已经被虚拟机调用过，虚拟机将这两种情况都视为“没有必要执行”。

如果这个对象被判定为有必要执行Finalize()方法，那么这个对象将会放置在一个叫F-Queue的队列中，并在稍后由一个由虚拟机自动建立的、低优先级的Finalizer线程去执行它。这里所谓的“执行”是指虚拟机会触发这个方法，但并不承诺会等待它运行结束，在执行此方法时是对象逃脱死亡的最后机会。只需要在Finalize()方法中重新与引用链的任何一个对象建立联系，譬如把this赋值给某个类变量或成员变量，那么在第二次标记时将会被移出“即将回收”的集合。

这个方法应尽量避免使用。因为这是JAVA刚诞生时为了使C/C++程序员更容易接受而做的妥协。使用它不确定性大。所以尽量使用try-finally或者其它方法，可以做得更好。

package com.learn;

public class FinalizeEscapeGC {

public static FinalizeEscapeGC FEGC;

public void isAlive(){

System.out.println("yes i am is staill alive");

}

@Override

protected void finalize() throws Throwable {

super.finalize();

System.out.println("finalize method executed!!!");

FEGC = this;

}

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

FEGC = new FinalizeEscapeGC();

FEGC = null;

//在这次被失去引用并且回收时会调用finalize()方法，所以可以成功逃脱GC

System.gc();

Thread.sleep(500);

if(FEGC != null){

FEGC.isAlive();

}else{

System.out.println("on no i am dead:");

}

//因为finalize()只会调用一次机制，所以在这次并不能成功逃脱GC

FEGC = null;

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

System.gc();

Thread.sleep(500);

if(FEGC != null){

FEGC.isAlive();

}else{

System.out.println("on no i am dead:");

}

}

}

* + 1. 回收方法区

JAVA虚拟机规范中说过可以不要求虚拟机在方法区实现垃圾收集，而且在方法区中进行垃圾收集的“性价比”比较低；永久代的垃圾收集主要回收两部分类容：废弃常量和无用的类。回收废弃常量与回收JAVA堆中的对象非常类似。常量池中字面量只要没有任何String对象引用此对象，也没有其他地方引用这个字面量，如果这时发生内存回收，而且有必要的话，这个常量将被系统清理。判断一个常量是否是“废弃常量”比较简单，而要判定一个类是否是“无用的类”的条件则相对苛刻，需要同时满足下面三个条件才算“无用的类”：

* 该类所有的实例都已经被回收，也就是JAVA堆中不存在该类的任何实例
* 加载该类的CLASSLoader已经被回收
* 该类对应的JAVA.LANG.CLASS对象没有在任何地方被引用，无法在任何地方通过反射方法访问该类的方法。

虚拟机可以对满足这三个条件的无用类进行回收，这里说的可以并不是必然回收。HotSpot虚拟机提供了-Xnoclassgc参数进行控制，还可以使用-verbose:class以及-XX:+TranceClassLoading、-XX:+TranceClassUnLoading查看类加载和卸载信息，其中-verbose:class和-XX:+TranceClassLoading可以在Product版的虚拟机中使用，-XXs:+TranceClassUnLoading参数需要FastDebug版的虚拟机支持。

在大量使用反射、动态代理、CFLIB等ByteCode框架、动态生成JSP以及OSGI这类频繁自定义ClassLoader的场景都需要虚拟机具备类卸载的功能，惟保证永久代不会溢出。

* + 1. 回收算法

各个虚拟机操作内存的方法是各不相同的。

标记-清除算法

标记-清除算法有两个不足的问题：一个是效率问题，标记和清除两个过程的效率都不高，另一个是空间问题，标记清除之后会产生大量不连续的内存碎片，空间碎片太多可能会导致以后在程序支行过程中需要较大的对象时，无法找到足够的连续内存而不得不提前触发另一次垃圾收集动作。

复制算法

为了解决效率问题，一种称为“复制”的收集算法出现了，它将可用内存安排好容量划分为大小相等的两块，每次只使用其中一块，当这一块的内存用完了，就将还存活的对象复制到另外一块上面，然后再把已使用过的内存空间一次清理掉，这样使用每次都是对整半个半区进行内存回收。这种算法实现简单，运行高效。只是这种算法的代理是将内存缩小为原来的一半，代价较高。

商业虚拟机都使用这种算法来回收新一代。因为新生代多数对象98%都“朝生夕死”的，所以并不需要按1:1的比例来划分内存。而是将内存划分为较大的Eden和两块较小的Survivor空间。每次使用Eden和一块Survivor空间。HotSpot虚拟机默认Eden和Survivor的大小比较为8:1，也就是每次新生代中可用内存的空间大小为整个新生代容量的90%，只有10%的内存会浪费。 当Survivor空间不够用时，需要依赖其他内存（老年代）进行分配担保。

标记整理算法

由于老年代的特点，老年代一般采用标记-整理算法，标记过程仍然与标记-清除算法一样，但后续步骤不是直接对可回收对象进行清理，而是让所有存在的对象都向一端移动，然后直接清理掉端边界以外的内存。

分代收集算法

分代收集算法是当前商业虚拟机的垃圾收集都采用的算法。这种算法保是根据对象存活周期的不同将内存划分为几块，一般是把JAVA堆分为新生代和老年代，这样可以根据各个年代的特点采用最适当的收集算法。

* + 1. 枚举根节点

可达性分析中从GC ROOTS节点找引用链这个操作为例，可作为GC ROOTS的节点主要在全局性引用（如常量或类静态属性）与执行上下文（栈帧的本地变量表）中，现在很多应用仅仅方法区就有数百兆，如果逐个检查引用，那么必须会消耗很多时间。

另外，可达性分析对执行时间的敏感还体现在GC停顿上，因为这基分析工作必须在一个能确保一致性的快照中进行-这里“一致性”的意思是指在整个分析期间整个执行系统看起来就像被冻结在某个时间点上，不可以出现过程中对象引用关系还在不断变量的情况，该 点不滞的话分析结果准确性不无法得到保证。这点是导致GC进行必须停顿所有JAVA线程的其中一个重要原因（Stop the World），即使是在号称几乎不会停顿的CMS收集器中，枚举根节点的操作也必须停顿。

在HotSpot中，使用了一组称为OopMap的数据结构来存放对象的引用。在类加载完成的时候，HotSpot就把对象内什么偏移量上是什么类型的数据计算出来，在JIT编译过程中，也会在特定的位置记录下栈和寄存器中哪些位置是引用。这样GC在扫描的时候就可以直接得知这些信息了。

* + 1. 安全点

在OopMap的协助下，HotSpot可以快速且准确地完成GC ROOTS枚举，但是一个很现实的问题随之而来，可能导致引用关系变化，或者说OopMap的指令非常多，如果为每一条指令都生成对应的OopMap，那将会需要大量的额外空间，这样GC的空间成本将会变得非常高。

实际上HotSpot也的克没有为每条指令都生成OopMap,前面已经提到，只是在特定的位置记录了这些信息，这些益称为安全点，即使程序执行时并非在所有的地方都可以停顿下来开始GC,只有在到达安全点才能暂停。安全点的选定既不能太少致于让GC等待时间太长，也不能过于频繁以致于过分增大支行时的负荷。所以，安全点的选定基本上是以程序“是否具有让唾弃长时间执行的特征”为标准进行选定的—由于每条指令执行的时间都非常短暂，程序不太可能因为指令流长度太长这个原因而过长时间支行“长时间执行”的最明显特征就是指令序列复用，例如方法调用、循环跳转、异常中转等，所以具有这些功能的指令才会产生安全点。

对于安全点，另一个需要考虑的问题是如何在GC必生时所有的线程都跑到最近的安全点上停顿下来。这里有两种方案可选：抢选式中断和主动式中断，其中抢先式中断不需要线程的执行代码主动去配合，在GC发生时，首先把所有线程全部中断，如果发现线程中断的地方不在好人一生平安点上，就恢复线程，让它跑到安全点上。现在几乎没有虚拟机实现采用抢先式中断来暂停线程从而响应GC事件。主动式中断的思想是当GC需要中断线程的时候，不直接对线程操作，仅仅当简单地设置一个标志，各个线程执行时主动去轮询这个标志，发现中断标志为真时就自已中断挂起。轮询标志的地方和安全点是重合的，另外再加上创建对象需要分配的内存的地方。

* + 1. 安全区域

当线程处于Sleep状态或者Blocked状态，这时候线程无法响应JVM的中断请求，走到安全点去挂机，JVM不太可能等待线程重新被分配CPU的时间，对于这种情况，就只有靠安全区域来解决。安全区域是指在一段代码片段中，引用关系不会发生变化，在这个区域的任何地方开始GC都是安全的，我们也可以把安全区看做被扩展的安全点。在线程执行到安全区中的代码时，首先标识自己已经进入了安全区域，那样，当在这段时间里JVM要发起GC时，就不用管标识自己为安全区域的状态的线程了，在线程要离开安全区时，它要检查系统是否已经完成了根节点枚举，如果完成了，那线路就继续执行，否则就必须等待直到可以安全离开安全区的信号为止。

* 1. 垃圾收集器



上图展示了七种作用于不同分代的收集器，如果两具收集器之间存在连线，就说明 它们可以搭配使用。虚拟机所处的区域，则表示他们属性新生代收集器还是老年代收集器。

JAVA虚拟机对垃圾收集器应如何实现并没有任何规定，因此不同厂商、不同版本的虚拟机所提供的垃圾收集器有较大差别，并且一般都会提供参数供用户根据自己的应用特点和要求组合出各个年代所使用的收集器。

* + 1. Serial收集器

这个收集器是一个单线程的收集器，他的“单线程”的意义不并仅仅说明它只会使用一个CPU或一条收集线程去完成垃圾收集工作，更重要的是它在进行垃圾收集时，必须其它所有工作线程，直到它收集结束。“Stop the World”要在用户不可见的情况下把所有用户正常工作的线程全部停掉，这对很多应用来说都是难以接受的。它是虚拟机支行在Client模式下的默认新生代收集器。它也有着优于其它收集器的地方：简单而高效（相对单线程比），对于单个CPU的环境 来说，Serial收集器没有线程交互开销，专心做垃圾收集。在用户的桌面应用场景中，分配给虚拟机管理的内存一般来说不会很大，收集几十兆基本一两百兆的新生代，停顿时间一般在几十毫秒到一百来毫秒内，只要不是频繁发生，都是容易接受的。

* + 1. ParNew收集器

ParNew收集器其实就是Serial收集器的多线程版本，除了使用多条线程进行垃圾收集之外，其余行为包括Serial收集器可用的所有控制参数、收集算法、Stop the world、对象分配规则、回收策略等都与Serial收集器完全一样，在实现上，这两种收集器也共用了相当多的代码。ParNew收集器除了多线程收集之外，其他与Serial收集器相比并没有太多创新之处，但它却是许多支行在Server模式下的虚拟机中首选的新生代收集器，其中有一个与性能无关但很重要的原因是，除了Serial收集器外，目前只有它能与CMS收集器配合工作。

可以使用参数-XX:ParallelGCThreads参数来限制垃圾具集的线程数。

* + 1. Parallel Scavenge收集器

Parallel Scavenge收集器是一个新生代收集器，它也是使用复制算法的收集器，又是并行的多线程收集器…..Parallel Scavenge收集器的特点是它的关注点与其他收集器不同，CMS等收集器的关注点是尽可能地缩短垃圾收集时用户线程的停顿时间，而Parallel Scavenge收集器的目标则是达到一个可控制的吞吐量。所谓吞吐量就是CPU用户代码的时间与创建CPU总消耗时间的比值，即吞吐量=支行用户代码时间/（支行用户代码时间+垃圾收集时间）。

停顿时间越短就越适合需要与用户交互的程序，良好的响应速度能提升用户体验，而高吞吐量则可以高效率地利用CPU时间，尽快完成程序的运算任务，主要适合在后台运算而不需要太多交互的任务。

Parallel Scavenge收集器提供了两个参数用于精确控制吞吐量，分别是控制最大垃圾收集停顿时间的-XX:MaxGCPauseMillis参数以及直接设置吞吐量大小的-XX:GCTimeRatio参数。MaxGCPauseMillis参数允许的值是一个大于0的毫秒数，收集器将尽可能地保证内存回收花费的时间不超过设定值。如果将值设置过小，将导致GC频繁发生，降低了吞吐量。

GCTimeRatio参数的值应当是一个大于0且小于100的整数，也就是垃圾收集时间占总时间的比率，相当于吞吐量的倒数。如果设置为19，那么允许的最大GC时间就占总时间的5%（1/(1+19)）。

由于与吞吐量关系密切，Parallel Scavenge收集器也经常称为“吞吐量优先”收集器。除上两个参数外，Parallel Scavenge收集器还有一个参数-XX:+UseAdaptiveSizePolicy，这是一个开关参数，当这个参数打开后，就不需要手动指定 新生代和大小、Eden与Survivor区的比例，晋升老年代对象年龄（-XX:PretenureSizeThreshold）等细节参数了，虚拟机会根据当前系统的运行情况收集性能监控信息，动态调整这些参数以提供最合适的停顿时间或最大的吞吐量，这种调节方式称为GC自适应的调节策略。自适应调节策略也是Parallel Scavenge收集器与ParNew收集器的一个重要区别。

* + 1. Serial Old收集器

Serial Old是Serial收集器的老年代版本，它同样是一个单线程收集器，使用“标记-整理”算法。这个收集器的主要意义也是在于给Client模式下的虚拟机使用。如果在Server模式下，那么它主要还有两在用途：一个用途是与Parallel Scavenge收集器搭配使用，另一种是作为CMS收集器的后预案，在并发收集发生Concurrent Mode Failure时使用。

* + 1. Parallel Old收集器

Parallel Old是Parallel Scavenge收集器的老年代版本，使用多线程和“标记-整理”算法。这个收集器在JDK1.6提供。Parallel Old收集器出现后，“吞吐量优先”收集器终于有了比较名副其实的应用组合，在注重吞吐量以及CPU资源敏感的场合，都可以优先考虑Parallel Scavenge加Parallel Old收集器。

* + 1. CMS收集器

CMS收集器是一种以获取最短回收停顿时间为目标的收集器。目前很大一部分是JAVA应用集中在互联网站或B/S系统的服务端上，这类应用重视服务的响应速度，希望系统停顿时间最短，以给用户带来较好的体验，CMS收集器就非常符合这类应用的需求。

CMS收集器是基于“标记-清除”算法实现的，它的动作过程相对于前面几种收集器来说更复杂，整个过程分4步:

* 初始标记
* 并发标记
* 重新标记
* 并发清除

其中，初始标记、重新标记这两步仍然需要”Stop the Wrold“。初始标记仅仅只是标记一下GC ROOTS能直接关联到的对象，速度很快，并发标记阶段就是进行GC ROOTS Tracing的过程，而重新标记阶段则是为了修正并发标记期间因用户程序继续支行而导致标记产生变动的那 部分对象的标记记录，这个阶段的停顿时间一般会比初始标记阶段稍长一些，但远比并发标记的时间短。由于整个过程耗时最长的并发标记和并发清除过程收集器都可以与用户线程一起工作，所以，总体来说，CMS收集器的内存回收过程是与用户线程一起并发执行的。

CMS也叫并发低停顿收集器（Concurrent Low Pause Collector）是一款非常优秀的收集器，但有3个明显的缺点：

CMS收集器对CPU资源敏感，并发执行会导致应用程序变慢，总吞吐量降低。当CPU不足4个时，CMS对用户程序 的影响可能会较大，这可以是让人无法接受的。

CMS收集器无法处理浮动垃圾，可能出现”Concurrent Mode Failure”失败而导致另一次Full GC的产生。由于CMS并发清理阶段用户线程还在支行着，伴随程序支行自然就还会有新的垃圾不断产生，这一部分垃圾出现在标记过程中，CMS无法在当次收集中处理掉它们，只好留到下一次GC时再清理。这一部分垃圾就被称为“浮云垃圾”。由于垃圾收集阶段用户线程还需要支行，那也就还需要预留有足够的内存空间给用户线程使用，因此CMS收集器不能像其他收集器那样等到老年代几乎完全被填满再进行收集，需要预留一部分空间提供并发收集时的程序运作使用。在JDK1.6中，CMS收集器的启动阈值设置到了92%，要是CMS支行期间预留的内存无法满足程序需要时，就会出现一次“Concurrent Mode Failure”失败，这时虚拟机将启动后备预案：临时启动Serial Old收集器来重新进行老年代的垃圾收集，这样停顿时间就很长了。所以说参数-XX:CMSInitiatingOccupancyFraction设置得太高容易导致大量“Concurrent Mode Failure”失败，性能反而降低。

CMS是一款基于“标记-清除”算法来实现的收集器，这意味着收集结束时会有大量空间碎片产生。空间碎片过多时，将会给大对象分配带来很大麻烦，往往会出现老年代还有很大空间剩余，但是无法找到足够大的连续空间来分配当前对象，不得不提前触发一次FULL GC。为了解决这个问题，CMS收集器提供了一个-XX:UseCMSCompactAtFullCollectio开关参数（默认开启），用于在CMS收集器顶不住要进行FullGC时开启内存内存碎片的合并整理过程，内存整理的过程是无法并发的，空间碎片问题没有了，但停顿时间不得不变长。虚拟机设计者还提供了另外一个参数-XX:CMSFullGCsBeforeCompaction,这个参数是用于设置执行多少次不压缩的FULL GC后，跟着来一次带压缩的（默认值为0，表示每次进入FULLGC都进行碎整理）。

* + 1. G1收集器

G1收集器是当今收集器技术发展的最前沿成果之一。是一款面向服务端的垃圾收集器。HtoSpot团队希望在未来可以靠它替换掉CMS收集器。其特点：

并行与并发：G1能充分利用多CPU、多核环境下的硬件优势，使用多个CPU来缩短Stop The World停顿的时间，部分其他收集器原来需要停顿JAVA线程执行的GC动作，G1收集器仍然可以通过并发的方式让JAVA程序继续执行。

分代收集：与其他收集器一样，分代概念在G1中依然得以保留。虽然G1可以不需要其他收集器配合就能独立管理整个GC堆，但它能够采用不同的方式去处理新创建的对象和已经存活了一段时间、熬过多次GC的旧对象以获取更好的收集效果。

空间整理：与CMS的标记-清理算法不同，G1从整体来看是基于“标记-整理”算法实现的收集器，从局部上来看是基本“复制”算法实现的，但无论如何，这两种算法都意味着G1运作期间不会产生内存空间碎片，收集后能提供规整的可用内存。这种我有利于程序长时间运行，分配大对象时不会因为无法找到连续的空间而提前触发下一次GC

可预测的停顿：这是G1相对于CMS的另一大优势，降低停顿时间是G1和CMS共同的关注点，但G1除了追求低停顿外，还能建立可预测的停顿时间模型，能让使用者明确指定在一个长度的M毫秒的时间片段内，消耗在垃圾收集上的时间不得超过N毫秒，这几乎已经是实时JAVA的垃圾收集器的特征了。

在G1之前的其他收集器里进行收集的范围都是整个新生代或者老年代，而G1不再是这样。使用G1收集器时，JAVA堆的内存布局就与其他收集器有很大差别，它将整个JAVA堆划分为多个大小相等的独立区域，虽然还保留有新生代和老年代的概念，但新生代和老年代不再是物理隔离的了，他们都是一部分Region(不需要连续)的集合。

G1收集器之所以能建立可预测的停顿时间模型，是因为它可以有计划地避免在整个JAVA堆中进行全区域的垃圾收集。G1跟踪各个Region里面的垃圾堆积的价值大小（回收获得的空间大小以及回收所需要时间的经验值），在后台维护一个优先列表，每次根据允许的收集时间，优先回收价值最大的Region。这种使用Region划分内存空间以及有优先级的区域回收方式，保证了G1收集器在有限的时间内可以获取尽可能高的收集效率。

|  |  |
| --- | --- |
| 参数 | 描述 |
| UseSerialGC | 虚拟机运行时在Client模式的下的默认值，打开此开关后，使用Serial+Serial Old的收集器组合进行内存回收 |
| UseParNewGC | 打开此开关后，使用PerNew+Serial Old收集器 |
| UseConcMarkSweepGC | 使用ParNew+CMS+Serial Old组合进行内存回收，Serial将作为CMS的后备收集器使用 |
| UseParallelGC | 虚拟机运行在Server模式下的默认值，使用ParallelScavenge+Serial Old的收集器组合进行内存回收 |
| UseParallelOldGC | Parallel Scavenge + Parallel Old |
| SurvivorRatio | Eden:Survivor的容量比 默认是8 |
| PretenureSizeThreshold |  |
| MaxTenuringThreshold | 每次GC后年龄加1，超过年龄的直接进入老年代 |
| UseAdaptiveSizePolicy | 动态调整JAVA堆中各个区域的大小以及进入老年代的年龄 |
| HandlePromotionFailure | 是否允许分配担保失败 |
| ParallelGCThreads | GC运行的线程数量 |
| GCTimeRatio | GC时间占用总时间的比率，默认99 即允许1%的GC时间 |
| MaxGCPauseMillis | GC的最大停顿时间，仅在使用Parallel Scavenge收集器时生效 |
| CMSInitiatingOccupancyFraction | 设置CMS收集器在老年代空间被使用多少后触发垃圾收集，默认68%，仅在使用GMS收集器时生效 |
| UseCMSCompactAtFullCollection | 设置CMS收集器在完成垃圾收集后是否整理内存，仅GMS收集器时生效 |
| CMSFullGCsBeforeCompaction | 设置CMS收集器在进行若干次垃圾收集后启动一次内存碎片整理，仅在使用GMS时生效 |

* 1. 内存交互基础

Java内存模型的主要目标是定义程序中各个变量的访问规则，即在虚拟机 中将变量存储到内存和从内存中取出变量这样的底层细节。此处的变量与JAVA编程中的变量有区别的，它包括了实例字段、静态字段和构成数组对象的元素，但不包括局部变量与方法参数，因为后者是线程私有的，不会被共享，不存在竞争的问题。为了获得更好的性能，java内存模型并没有限制执行引擎使用处理器的特定寄存器或缓存来和主内存进行交互，也没有限制即时编译器调整代码执行顺序这类权利。

JAVA内存模型规定了所有的变量都存储在主内存（JAVA虚拟机的主机存）中，每条线程还有自己的工作内存，线程的工作内存中保存了被该线程使用到的变量的主内存副本拷贝，线程对变量的所有操作都必须在工作内存中进行，而不能直接读写主内存中的变量。不同的线程之间不可以相互访问，线程之间的变量值传递必需通过主内存来实现。

关于主内存与工作内存的交互协议，JAVA内存模型定义了以下八种操作：

1. Lock（锁定）：作用于主内存的变量，它把一个变量标识为一条线程独占状态。
2. Unlock(解锁)：作用于主内存被锁定的变量，将锁释放。
3. Read（读取）：作用于主内存的变量，把一个变量的值从主内存传输到线程工作内存中。
4. Load(载入)：作用于工作内存的变量，它把read操作得到的值放入工作内存的变量副本中。
5. Use(使用)：作用于工作内存的变量，它把工作内存的中的一个变量的值传递给执行引擎。
6. Assign(赋值)：作用于工作内存的变量，它把一个从执行引擎接收到的值赋值给工作内存的变量。
7. Store（存储）：作用于工作内存的变量，它把工作内存中的一个变量的值传送到主内存中，为write操作使用
8. Write(写入)：作用于主内存的变量，它把store操作的获取的值放入到主内存的变量中。

除此之外，JAVA内存模型还定义了八种操作时必须遵守的八条规则：

1. 不允许read和Load、store和write操作之一单出现，即不允许一变量从主内存读取之后工作内存不接收或从工作内存发起回写操作但主内存不操作的情况。同时要求操作必须按顺序执行，但不保证两条指令连续执行。
2. 不允许一个线程丢弃它的最近的assign操作，即变量在工作内存中改变了之后必须把该变量同步回主内存。
3. 不允许工作线程没有发生任何assign操作前提下把数据同步回主内存。
4. 一个新的变量只能在主内存中诞生，不允许在工作内存中直接使用一个未被初始化（load或assign操作）的变量。
5. 一个变量在同一时刻只允许一条线程对其进行Lock操作，但lock操作可以被同一线程执行多次，多次执行后只有两同次数的unlock操作才会解锁
6. 如果一个变量执行Lock操作，将会清空工作内存中此变量的值，在执行引擎使用这个变量前，需要重新load或assign操作初始化值
7. 如果一个变量事先没有被Lock操作，则不允许对它执行Unlock操作;也不允许unlock一个被其它线程锁定住的变量
8. 对一个变量执行Unlock操作之前，必需先把此变量同步到主内存中。
   1. volatile

对于volatile，java内存模型专门定义了一些特殊的访问规则：当一个变量被定义为volatile之后这个变量就具备了两种特性：第一是保证此变量对所有线程是可见的，这里的可见性是指当一条线程修改了这个变量的值，新值对于其它线程来说是可以立即得到通知的。而普通变量不能做到。

package com.learn;

import java.util.concurrent.ExecutorService;

import java.util.concurrent.Executors;

public class VolatileTest {

public static volatile int race = 0;

static void increase() {

race++;

}

static final int THREADS\_COUNT = 10;

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

ExecutorService pool = Executors.newFixedThreadPool(10);

for(int i=0;i<THREADS\_COUNT;i++){

pool.execute(()->{

for (int j=0;j<10000;j++){

increase();

}

System.out.println("["+Thread.currentThread().getName()+"]执行完毕");

});

}

pool.shutdown();

while(!pool.isTerminated()){Thread.yield();};

System.out.println(race);

}

}

执行结果：



由结果不难发现，volatile并不能保证安全，这个问题出在increase()方法中，volatile只保证了将数据Load的时候是最新的，但在执行加指令的时候很可能其它线程已经将值改变了，所以此时值会小于100000;这就是volatile的第一层含义。

package com.learn;

public class Volatile2 {

/\*volatile\*/ boolean flag = false;

public static void main(String[] args){

Volatile2 v = new Volatile2();

new Thread(()->{

try {

Thread.sleep(100); //业务处理

} catch (InterruptedException e) {

e.printStackTrace();

}

System.out.println(String.format("[%s]执行完毕！",Thread.currentThread().getName()));

v.flag = true;

},"初始化线程").start();

new Thread(()->{

while(!v.flag){;}

System.out.println("收到通知");

},"等待线程").start();

}

}

上面这也是一个好的例子，如果去掉了vaolaile关键字，取数据的时候并不从主内存去取，导致等待进程不能获取到最近结果，从而导致程序进行不下去。

Valitile的第二语义是禁止指令的重排序优化，普通变量仅仅会保证在该方法执行过程中所有依赖赋值结果的地方都能获取到正确的值，而不能保证变量赋值操作的顺序与程序代码的执行顺序一致。

如果将上面示例改变为业务代码，由于指令重排，很可能导致指令重排而使v.flag = true;提前执行，而实际业务并没有处理完成，如果等待进程依赖于初始化进程，这会对等待进程的业务处理造成影响。如果添加valitile关键字，将不会对指令重排从而避免问题发生。

* 1. Long和double

JAVA内存模型要定义了八个原子性操作（见本章 内存交互基础），但并没有要求64位数据读写操作的原子性，这就是所谓的Long和double的非原子性协定。也就是说如果有一个被多个线程共享并未声明volatile的64位数据，某些线程可能会读取到一个既非原值也非新值的奇怪结果。只在是虚拟机的实现上，几乎所有商用虚拟机都选择把64位数据的读写操作作为原子操作来对待。

* 1. 原子性、可见性、有序性

原子性：

由JAVA内存模型直接保证的原子性操作包括read、load、assign、use、store、wirte六个，如果需要理更大范围的原子性保证，JAVA内存模型提供了lock、unlock操作，尽管此操作没有向用户公开，但提供了字节码指令monitorenter和monitorexit来隐式地使用这两个操作，反映到代码中就是同步块-synchronized关键字

可见性：

可见性就是指当一个线程修改了共享变量的值，其它线程能够立即得知这个修改。除了volatile外，JAVA还有两个关键字能实现可见性，它们是synchronized和final。同步块的可见性是由“对一个变量执行unlock操作之前，必须先把此变量同步回主内存中”这条规则获得的。而final关键字的可见性是指：被final修饰的字段在构造器中一量被初始化完成，并且构造器没有把this的引用传递出去，那么其它线程中就能看见final字段的值。

有序性：

如果在本线程内观察，所有的操作都是有序的;如果在一个线程中观察另一个线程，所有的操作都是无序的。JAVA语言提供了volatile和synchronized两个关键字来保证线程之间操作的有序性，volatile关键字本身就包含了禁止指令重排序的语义，而synchronized则是由“一个变量在同一个时刻只允许一条线程对其进行Lock操作”这条规则获得了，这条规则决定了持有同一个锁的两个同步块只能串行地进入。

* 1. Java与线程

线程是比进程更轻量级的调度执行单位，线程的引入，可以把一个进程的资源分配和执行调度分开，各个线程既可以共享进程资源，又可以独立调度。Java语言提供了不同硬件和操作系统平台下对线程操作的统一处理，每个java.lang.Thread类的实例不代表了一个线程。不过Thread类的大部分的JAVAAPI都被声明为Native。这就意味着这个方法没有使用或无法使用平台无关的手段来实现。实现线程主要有三种方式：使用内核线程实现，使用用户线程实现，使用用户线程加轻量级进程实现。

* + 1. 使用内核线程实现

内核线程就是直接由操作系统内核支持的线程，这种线程由内核来完成线程切换，内核通过操作调度器对线程进行调度，并负责将线程的任务映射到各个处理器上。每个内核线程都可以看做是内核的一个分身，这样操作系统就有能力同时处理多件事情，支持多线程的内核就叫多线程内核。

程序一般不会直接使用内核线程，而是去使用内核线程的一种高级接口—轻量级进程，轻量级进程就是我们通常意义所讲的线程，由于每个轻量级进程都有一个内核线程支持，因此只有先支持内核线程，才能有轻量级进程。这种轻量级进程与内核进程之间1:1的关系称为一对一线程模型。

由于内核线程的支持，每个轻量级进程都成为一个独立的调度单元，即使有一个轻量级进程在系统调用中阻塞了，也不会影响整个进程继续工作，但是轻量级进程具有它的局限性：首先，由于基于内核线程的实现，所以各种操作如创建、析构、同步都需要进行系统调用。而系统调用的代价相对较高，需要在用户态和内核态中来回切换。其次。每个轻量级进程都需要一个内核线程的支持，因此轻量级进程要消耗一定的内核资源，因此一个系统支持轻量级进程的数量是有限的。

* + 1. 使用用户线程实现

从广义来讲，一个线程只要不是内核线程，它就是用户线程，从这个定义上来讲，轻量级进程也属于用户线程，但轻量级进程的实现始终是建立在内核之上的，许多操作都要进行系统调用，因此效率会受到限制。

而狭义上的用户线程是完全建立在用户空间的线程库上的，系统内核并不能感知到线程存在和实现。用户线程的建立、同步、销毁和调度完全在用户态中完成，不需要内核的帮助，因此操作可以非常快速且低消耗，也可以支持规模更大的线程数量。这种进程与用户线程之间1：N的关系称为一对多的线程模型。

使用用户线程的优势在于不需要内核支持，劣势也在没有内核支持，所有的线程操作都需要用户程序自己处理。

* + 1. 混合实现

线程除了依赖内核线程实现和完全由用户程序自己实现外，还有一种将内核线程与用户线程一起使用的实现方式。在这种混合实现下，既存在用户线程，也存在轻量级进程。用户线程还是完全建立在用户空间中，因此用户线程的创建、切换、析构等操作依然廉价，并且可以支持大规模的用户线程并发。而操作系统提供的轻量级进程则作为用户线程和内核线程之间的桥梁，这样可以使用内核提供的线程调度功能及处理器映射，并且用户线程的系统调用要通过轻量级线程来完成，大大降低了进程被阻塞的风险。在这种模式中，用户线程与轻量级进程的数量比是不定的，是M:N的关系。

* + 1. JAVA的实现

JAVA虚拟机规范中并没有限定JAVA线程需要使用哪种线程模型来实现。Sun JDK来说,它的windows和linux版都是一对一的线程模型来实现的。

* 1. JAVA线程调度

线程调度是指系统为线程分配处理器使用权的过程，主要调度方式有两种，分别是协同式线程调度和抢占式线程调度。

如果使用协同式调度的多线程系统，线程的执行时间由线程本身来控制，线程把自己的工作执行完成之后，要主动通知系统切换到其它线程上去，协同式多线程的最大好处就是实现简单，而且由于线程要把自己的事情干完后才会进行线程切换，切换操作是线程自己可知的，所以没有线程同步问题。它的坏处就是执行时间不可控。

如果使用抢占式调度的多线程系统，每个线程将由系统来分配执行时间，线程的切换不由线程本身来决定（Thread.yield()可以让出执行时间）。在这种实现方式下，线程的执行时间是系统可控的，也不会有一个线程导致整个进程阻塞的问题，JAVA使用的线程调度方式就是抢占式的。虽然JAVA线程调度是系统自动完成的，但我们还可以建议系统分配资源，可以使用优先级来完成，JAVA语言设置了10个级别的线程优先级。

1. JVM原理之工具
   1. JPS

JPS命令可以列出正在支行的虚拟机进程，并显示虚拟机执行主类名称以及这些进程的本地虚拟机唯一ID。

Jsp [options] [hostid]

参数选项：

|  |  |
| --- | --- |
| 选项 | 作用 |
| -q | 只输出LVMID,省略主类名称 |
| -m | 输出虚拟机进程启动时传递给主类Main的参数 |
| -l | 输出主类的全名，如果执行的是jar包，输出jar路径 |
| -v | 输出虚拟机进程启动时JVM参数 |

* 1. Jstat

Jstat是用于监视虚拟机各种支行状态信息的命令行工具。它可以显示本地或远程虚拟机进程中的类装载、内存、垃圾收集、JIT编译等支行数据。

Jstat [ option vmid [interval [s|ms] [count] ]]

对于命令格式vmid与lvmid是一致的，如果是远程虚拟机进程，那VMID的格式是：

[protocol:] [//\_]lvmid[@hostname[:port]/servername]

参数interval和count代表查询间隔和次数，如果省略这两个参数，说明只查询一次。

Jstat –gc 2764 250 20 每250毫秒查询一次进程2764垃圾收集状况，一共查询20次。

Jstat工具主要选项

|  |  |
| --- | --- |
| 选项 | 作用 |
| -class | 监视类装载、卸载数量、总空间以及类装载所耗费的时间 |
| -gc | 监视java堆状况，包括eden区、两个survivor区、老年代、永久代等的容量、已用空间、GC时间合计等信息 |
| -gccapacity | 监视内容与-gc基本相同，但输出主要关注JAVA堆各个区域使用到的最大、最小空间 |
| -gcutil | 监视内容与-gc基本相同，但输出主要关注已使用空间占空空间的百分比 |
| -gccause | 与-gcutil功能一样，但是会额外输出导致上一次GC产生的原因 |
| -gcnew | 监视新生代GC状况 |
| -gcnewcapacity | 监视内容与-gcnew基本相同，输出主要关注使用到的最大、最小空间 |
| -gcold | 监视老年代GC状况 |
| -gcoldcapacity | 监视内容与-gcold基本相同，输出主要关注使用到的最大、最小空间 |
| -gcpermcapacity | 输出永久代使用到的最大、最小空间 |
| -compiler | 输出JIT编译器编译过的方法、耗时等信息 |
| -printcompilation | 输出已经被JIT编译的方法 |

* + 1. jstat –class<pid>

显示加载class的数量，及所占空间等信息。

|  |  |
| --- | --- |
| 显示列名 | 具体描述 |
| Loaded | 装载的类的数量 |
| Bytes | 装载类的占用的字节数 |
| Unloaded | 卸载类的数量 |
| Bytes | 制裁类的字节数 |
| Time | 装载和卸载类所花费的时间 |

* + 1. jstat -compiler <pid>

显示VM实时编译的数量等信息。

|  |  |
| --- | --- |
| 显示列名 | 具体描述 |
| Compiled | 编译任务执行数量 |
| Failed | 编译任务执行失败数量 |
| Invalid | 编译任务执行失效数量 |
| Time | 编译任务消耗时间 |
| FailedType | 最后一个编译失败任务的类型 |
| FailedMethod | 最后一个编译失败任务所在的类及方法 |

* + 1. jstat -gc <pid>

可以显示gc的信息，查看gc的次数，及时间。

|  |  |
| --- | --- |
| 显示列名 | 具体描述 |
| S0C | 年轻代中第一个SURVIVOR(幸存区)的容量（字节） |
| S1C | 年轻代中第二个survivor（幸存区）的容量 (字节) |
| S0U | 年轻代中第一个survivor（幸存区）目前已使用空间 (字节) |
| S1U | 年轻代中第二个survivor（幸存区）目前已使用空间 (字节) |
| EC | 年轻代中Eden（伊甸园）的容量 (字节) |
| EU | 年轻代中Eden（伊甸园）目前已使用空间 (字节) |
| OC | Old代的容量 (字节) |
| OU | Old代目前已使用空间 (字节) |
| PC | Perm(持久代)的容量 (字节) |
| PU | Perm(持久代)目前已使用空间 (字节) |
| YGC | 从应用程序启动到采样时年轻代中gc次数 |
| YGCT | 从应用程序启动到采样时年轻代中gc所用时间(s) |
| FGC | 从应用程序启动到采样时old代(全gc)gc次数 |
| FGCT | 从应用程序启动到采样时old代(全gc)gc所用时间(s) |
| GCT | 从应用程序启动到采样时gc用的总时间(s) |

* + 1. jstat -gccapacity <pid>

可以显示，VM内存中三代（young,old,perm）对象的使用和占用大小

|  |  |
| --- | --- |
| 显示列名 | 具体描述 |
| NGCMN | 年轻代(young)中初始化(最小)的大小(字节) |
| NGCMX | 年轻代(young)的最大容量 (字节) |
| NGC | 年轻代(young)中当前的容量 (字节) |
| S0C | 年轻代中第一个survivor（幸存区）的容量 (字节) |
| S1C | 年轻代中第二个survivor（幸存区）的容量 (字节) |
| EC | 年轻代中Eden（伊甸园）的容量 (字节) |
| OGCMN | old代中初始化(最小)的大小 (字节) |
| OGCMX | old代的最大容量(字节) |
| OGC | old代当前新生成的容量 (字节) |
| OC | Old代的容量 (字节) |
| PGCMN | perm代中初始化(最小)的大小 (字节) |
| PGCMX | perm代的最大容量 (字节) |
| PGC | perm代当前新生成的容量 (字节) |
| PC | Perm(持久代)的容量 (字节) |
| YGC | 从应用程序启动到采样时年轻代中gc次数 |
| FGC | 从应用程序启动到采样时old代(全gc)gc次数 |

* + 1. jstat -gcutil <pid>

统计gc信息

|  |  |
| --- | --- |
| 显示列名 | 具体描述 |
| S0 | 年轻代中第一个survivor（幸存区）已使用的占当前容量百分比 |
| S1 | 年轻代中第二个survivor（幸存区）已使用的占当前容量百分比 |
| E | 年轻代中Eden（伊甸园）已使用的占当前容量百分比 |
| O | old代已使用的占当前容量百分比 |
| P | perm代已使用的占当前容量百分比 |
| YGC | 从应用程序启动到采样时年轻代中gc次数 |
| YGCT | 从应用程序启动到采样时年轻代中gc所用时间(s) |
| FGC | 从应用程序启动到采样时old代(全gc)gc次数 |
| FGCT | 从应用程序启动到采样时old代(全gc)gc所用时间(s) |
| GCT | 从应用程序启动到采样时gc用的总时间(s) |

* + 1. jstat -gcnew <pid>

年轻代对象的信息。

|  |  |
| --- | --- |
| 显示列名 | 具体描述 |
| S0C | 年轻代中第一个survivor（幸存区）的容量 (字节) |
| S1C | 年轻代中第二个survivor（幸存区）的容量 (字节) |
| S0U | 年轻代中第一个survivor（幸存区）目前已使用空间 (字节) |
| S1U | 年轻代中第二个survivor（幸存区）目前已使用空间 (字节) |
| TT | 持有次数限制 |
| MTT | 最大持有次数限制 |
| EC | 年轻代中Eden（伊甸园）的容量 (字节) |
| EU | 年轻代中Eden（伊甸园）目前已使用空间 (字节) |
| YGC | 从应用程序启动到采样时年轻代中gc次数 |
| YGCT | 从应用程序启动到采样时年轻代中gc所用时间(s) |

* + 1. jstat -gcnewcapacity<pid>

年轻代对象的信息及其占用量。

|  |  |
| --- | --- |
| 显示列名 | 具体描述 |
| NGCMN | 年轻代(young)中初始化(最小)的大小(字节) |
| NGCMX | 年轻代(young)的最大容量 (字节) |
| NGC | 年轻代(young)中当前的容量 (字节) |
| S0CMX | 年轻代中第一个survivor（幸存区）的最大容量 (字节) |
| SOC | 年轻代中第一个survivor（幸存区）的容量 (字节) |
| S1CMS | 年轻代中第二个survivor（幸存区）的最大容量 (字节) |
| S1C | 年轻代中第二个survivor（幸存区）的容量 (字节) |
| ECMX | 年轻代中Eden（伊甸园）的最大容量 (字节) |
| EC | 年轻代中Eden（伊甸园）的容量 (字节) |
| YGC | 从应用程序启动到采样时年轻代中gc次数 |
| FGC | 从应用程序启动到采样时old代(全gc)gc次数 |

* + 1. jstat -gcold <pid>

old代对象的信息。

|  |  |
| --- | --- |
| 显示列名 | 具体描述 |
| PC | Perm(持久代)的容量 (字节) |
| PU | Perm(持久代)目前已使用空间 (字节) |
| OC | Old代的容量 (字节) |
| OU | Old代目前已使用空间 (字节) |
| YGC | 从应用程序启动到采样时年轻代中gc次数 |
| FGC | 从应用程序启动到采样时old代(全gc)gc次数 |
| FGCT | 从应用程序启动到采样时old代(全gc)gc所用时间(s) |
| GCT | 从应用程序启动到采样时gc用的总时间(s) |

* + 1. stat -gcoldcapacity <pid>

old代对象的信息及其占用量。

|  |  |
| --- | --- |
| 显示列名 | 具体描述 |
| OGCMN | old代中初始化(最小)的大小 (字节) |
| OGCMX | old代的最大容量(字节) |
| OGC | old代当前新生成的容量 (字节) |
| OC | Old代的容量 (字节) |
| YGC | 从应用程序启动到采样时年轻代中gc次数 |
| FGC | 从应用程序启动到采样时old代(全gc)gc次数 |
| FGCT | 从应用程序启动到采样时old代(全gc)gc所用时间(s) |
| GCT | 从应用程序启动到采样时gc用的总时间(s) |

* + 1. jstat -gcpermcapacity<pid>

 perm对象的信息及其占用量。

|  |  |
| --- | --- |
| 显示列名 | 具体描述 |
| PGCMN | perm代中初始化(最小)的大小 (字节) |
| PGCMX | perm代的最大容量 (字节) |
| PGC | perm代当前新生成的容量 (字节) |
| PC | Perm(持久代)的容量 (字节) |
| YGC | 从应用程序启动到采样时年轻代中gc次数 |
| FGC | 从应用程序启动到采样时old代(全gc)gc次数 |
| FGCT | 从应用程序启动到采样时old代(全gc)gc所用时间(s) |
| GCT | 从应用程序启动到采样时gc用的总时间(s) |

* + 1. jstat -printcompilation <pid>

当前VM执行的信息。

|  |  |
| --- | --- |
| 显示列名 | 具体描述 |
| Compiled | 编译任务的数目 |
| Size | 方法生成的字节码的大小 |
| Type | 编译类型 |
| Method | 类名和方法名用来标识编译的方法。类名使用/做为一个命名空间分隔符。方法名是给定类中的方法。上述格式是由-XX:+PrintComplation选项进行设置的 |

* 1. jinfo

实时查看调整虚拟机的各项参数

格式：jinfo [option] <pid>

* 1. Jmap:java内存映像工具
  2. jconsole

1. JVM原理之类加载

在JAVA语言主里面，类型的加载、连接和初始化过程都是在程序支行期间完成的。这种策略虽然会令类加载时稍微增加一些性能开销，但会为JAVA应用程序提供高度的灵活性，JAVA天生可以动态扩展的语言特性就是依赖支行期间动态加载和动态连接这个特点实现的。

类的加载时机

* 1. 类的加载时机

类从被加载到虚拟机内存中开始，到卸载出内存为止，它的整个生命周期包括：加载、验证、准备、解析、初始化、使用和卸载七个阶段。其中验证、准备、解析三个部分统称为连接。其中加载、验证、准备、初始化和卸载五个阶段的顺序是确定的，类的加载过程必须按这种顺序按部就班地开始，而解析阶段则不一定：它在某些情况下可以在初始化阶段之后再开始，这是为了支持JAVA语言的支行时绑定（也称为动态绑定或晚期绑定）。

1. 虚拟机规范严格fwpgb有且只有五种情况下必须立即对类进行初始化：
2. 当遇到new、getstatic、putstatic、invokestatic这四条字节码指令，如果类没有进行过初始化，是需要先触发其初始化。
3. 当初始化一个类的时候，如果发现其父类还没有进行过初始化，则需要先触发其父类的初始化
4. 当虚拟机启动时，用户需要指定一个要执行的主类，虚拟机会先初始化这个类
5. 当使用JDK1.7的动态语言支持时，如果一个java.lang.invoke.MethodHandle实例最后的解析结果REF\_getStatic、REF\_putStatic、REF\_invodeStatic的方法句柄，并且这个方法句柄所对应的类没有进行过初始化，则需要先触发其初始化。
   1. 类的加载过程
      1. 加载

加载阶段，虚拟机需要完成以下三件事情：

1. 通过一个类的全限定名来获取定义此类的二进制字节流
2. 将这个字节流所代表的静态存储结构转化为方法区的运行时数据结构
3. 在内存中生成一个代表这个类的java.lang.class对象，作为这个方法区这个类的各种数据的访问入口。

通过类的全限定名获取定义此类的二进制字节流，并没有指明二进制字节流要从一个class文件中获取，准备地说根本没有指明要从哪里获取、怎样获取。这也是日后JAR、WAR、EAR格式的基础，还可以从网络中获取，运行时动态生成，这也是动态代理的基础。

加载阶段既可以使用系统提供的引导类加载器去完成，也可以由用户自定义的类加载器去完成，开发人员可以通过定义一个自己的类加载器去控制字节流的获取方式（即重写一个类加载器的loadClass()方法）。对于数组类而言，情况会不所不同，数组类本身不通过类加载器创建，它是由于JAVA虚拟机直接创建的。但数组类与类加载器仍然有很密切的关系，因为数组类的元素类型最终是靠类加载器去完成创建的，一个数组类的创建过程遵循以下规则：

如果数组类的组件类型是引用类型，那就递归采用定义的加载过程加载这个组件类型，这个数组类将在加载该组件类型的类加载器的类名称空间上被标识。

如果不是引用类型，JAVA虚拟机将会把数组类标记为引用类加载器关联。

数组类的可见性与它的组件类型的可以性一致，如果组件类型不是引用类型，那数组类的可见性将默认Public。

加载阶段完成后，虚拟机外部的二进制字节流就按虚拟机需要的格式存储在方法区之中，方法区中的数据存储格式由虚拟机实现自行定义。然后在内存中实例化一个java.lang.Class类的对象（并没有明确定规是JAVA堆中，对于HotSpot虚拟机而言，它虽然是对象，存储在方法区里面）这个对象作为程序访问方法区中的这些类型数据的外部接口。

加载阶段与连接阶段的部分内部（如一部分字节码文件格式验证动作）是交叉进行的，加载阶段沿未完成，连接阶段可能已经开始。但这些夹在加载阶段之中进行的动作，仍然属于连接阶段的内容，这两个阶段的开始时间仍然保持着固定的先后顺序。

* + 1. 验证

验证是连接阶段的第一步，这一阶段的目的是为了确保class文件的字节流中包含的信息符合当前虚拟机的要求，并且不会危害虚拟机自身的安全。验证阶段是非常重要的，这个阶段是否严谨，直接决定了JAVA虚拟机是否能承受恶意代码的攻击，从执行性能的角度上讲，验证阶段的工作量在虚拟机的类加载子系统中又占了相当大一部分。验证阶段大到致上会完成下面四个阶段的动作：

1. 文件格式验证

验证字节流是否符合CLASS文件格式规范，

是否魔数0xcafebabe开头

主、次版本是否兼容

常量池的常量中是否有不被支持的常量

指向常量的各种索引值中是否指定不存在的常量或不符合类型的常量

CONSTANT\_UTF8\_info型的常量中是否有不符合UTF8编码的数据

CLASS文件中各个部分及文件本身是否有被删除的或附加的其他信息……

1. 元数据验证

元数据验证是对信息进行语义分析，以保证其描述的信息符合JAVA语言规范的要求

这个类是否有父类

这个类的父类是否继承了不允许被继承的类

如果这个类不是抽象类，是否实现了其父类的接口之中要求实现的所有方法

类中的字段、方法是否与父类产生矛盾

1. 字节码验证

字节码难是最复杂的一个阶段，主要目的是通过数据流和控制流分析，确定程序语义是合法的、符合逻辑的。这个阶段将对类的方法体进行校验分析，保证被校验类的方法在支行时不会做出危害虚拟机安全的事件，保证任意时刻操作数栈的数据类型与指令代码序列都能配合工作。

保证中转指令不会中转到方法体外的字节码指令上。

保证方法体中的类型转换是有效的

当然通过程序去校验程序逻辑是无法做到绝对准确的

1. 符号引用验证

最后一个阶段的检验发生在虚拟机将符号引用转化为直接引用的时候，这个转化动作将在连接的第三个阶段-解析阶段中发生。符号引用验证可以看做是对类自身以外（常量池中各种符号引用）的信息进行匹配性校验，通常需要校验下列内容：

符号引用中通过字符串描述的全限定名是否能找到对应的类

在指定类中是否存在符合方法的字段描述符以及简单名称所描述的方法和字段

符号引用中的类、字段、方法的访问性是否可被当前类访问

……

符号引用验证的目的是确保解析动作可以正常执行，如果无法通过符号引用验证，那么将会抛出IncompatibleClassChangeError异常的子类。

对于虚拟机的类加载机制来说，验证阶段是一个非常重要的，但不是必要的阶段。如果所支行的全部代码都已经被反复验证过，那么在实施阶段就可以考虑使用-Xverify:none参数来关闭大部分类验证措施，以缩短虚拟机类加载的时间。

* + 1. 准备

准备阶段是正式为类变量分配内存并设置类变量初始值的阶段，这些变量所使用的内存针在方法区中进行分配。这个阶段进行内存分配的仅仅包括类变量，不包含实例变量。通常情况下初始值为数据类型零值

* + 1. 解析

解析阶段是虚拟机将常量池内的符号引用替换为直接引有的过程。

符号引用：符号引用是以一组符号为描述所引用的目标，符号可以是任何形式的字面量，只要使用时能无歧义地定位到目标即可。符号引用与虚拟机的内存布局无关，引用的目标并不一定已经加载到内存当中，各种虚拟机实现的内存布局可以各不相同，但它们能接受的符号引用必须都是一致的，因为符号引用的字面量形式明确定义在JAVA虚拟机规范中的Class文件格式中。

直接引用：直接引用可以是直接指向目标的指针、相对偏移量或是一个能间接定位到目标的句柄。直接引用是虚拟机实现的内存布局相关的，同一个符号引用在不同虚拟机实例上翻译出来的直接引用一般不会相同。如果有了直接引用，那引用的目标必定已经在内存中存在。

解析的动作主要针对类或接口、字段、类方法、接口方法、方法类型、方法句柄和调用点限定符7类符号引用进行，分别对应于常量池的七种常量类型。

* + 1. 初始化

类初始化阶段是类加载过程的最后一步，前面的类加载过程中，除了在加载阶段用户应用程序可以通过自定义类型加载器参与之外，其余动作完全由虚拟机主导和控制。

在准备阶段，变量已经赋过一次系统要求的初始值，而在初始化阶段，则根据程序员通过程序制定的主观计划去初始化类变量和其他资源。初始化阶段是执行类构造器<init>()方法的过程。

<clinit>()方法是由编译器自动收集类中的所有变量的赋值动作和静态语句块中的语句合并产生的，编译器收集的顺序是由语句在源文件中出现的顺序所决定的，静态语句块中只能访问到定义在静态语句块之前的变量，定义在它之后的变量，在前面的静态语句块可以赋值，但不能访问。

<clinit>()方法与类的构造函数不同，它不需要显示地调用父类构造器，虚拟机会保证在子类的<clinit>()方法执行之前，父类的<clinit>()方法已经执行完毕。因此虚拟机中第一个被执行的<clinit>()方法类肯定是java..lang.Object.

<clinit>()方法地类或接口来说并不必必需的，如果一个类中没有静态语句块，也没有对变量的赋值操作，那么编译器可以不为这个类生成<clinit>()。

接口中不可以使用静态语句块，但仍然有变量的赋值操作，因此接口与类一样都会生成<clinit>()方法。但接口与类不同的是，执行接口的<clinit>()方法不需要先执行父类的<clinit>()方法，只有当父接口中定义的变量使用时，父接口才会初始化。另外，接口实现类在初始化时也一样不会执行接口的<clinit>()方法。

虚拟机会保证一个类的<clinit>()方法在多线程环境中被正确的加锁、同步。如果多个线程同时去初始化一个类，那么只会有一个线程去执行这个类的<clinit>()方法，其它线程都需要阻塞等待，直到活动线程执行<clinit>()方法完毕。如果一个类的<clinit>()方法吕有耗时很长的操作，就可能造成多个进程阻塞。

* 1. 类与类加载器
     1. 类与类加载器

类加载器虽然用于实现类的加载动作，但它在JAVA程序中起到的作用却远远不限于类加载阶段。对于任意一个类，都需要由加载它的类加载器这个类本身一同确立其在JAVA虚拟机中的唯一性，每一个类加载器都拥有一个独立的类名称空间。也就是说只有两个类是由同一个类加载器加载的前提下比较才有意义，哪怕它们都出自同一个class文件。

* + 1. 双亲委派模型

从JAVA虚拟机的角度来说，只存在两种不同的类加载器：一种是启动类加载器，这个类加载器使用c++语言实现，是虚拟机自身的一部分；另一种就是所有其他的类加载器，这些类加载器都由JAVA语言实现，独立于虚拟机外部，并且全部继承自抽象类java.lang.ClassLoader。

绝大部分JAVA程序都会使用到系统提供的以下三种类加载器

启动类加载器：这个类加载器负责将存放在<JAVA\_HOME>\lib目录中的，或者被-Xbootclasspath参数所指定的路径中的，并且是虚拟机识别的类库加载虚拟机内存中。这个类载器无法被JAVA程序直接引用，用户在编写自定义类加载器时，如果需要把加载请求委派给引导类加载器，那直接使用Null代替即可。

扩展类加载器：这个加载器由sun.misc.Launcher$ExtClassLoader实现，它负责加载<JAVA\_HOME>\lib\ext目录中的，或者被java.ext.dirs系统变量所指定的路径中的所有类库开发者可以直接使用扩展类加载器

应用类加载器：这个类加载器由sun.misc.Launcher$AppClassLoader实现，由于这个类加载器是ClassLoader中的getSystemClassLoader()方法的返回值，所以一般称为系统类加载器，它负责加载用户类路径上所指定的类库，开发者可以直接使用这个类加载器。

我们的应用程序都是由这三个类加载器互相配合进行加载的，如果有必要，还可以加入自己定义的加载器。

双亲委派模型的工作过程：如果一个类加载器收到一个类加载的请求，它首先不会自己去加载这个类，而是把这个请求委派给父类加载器去完成，每一个层次的类加载器都是如此，因此所有的加载请求最终都应该传送到顶层的启动类加载器中，只有当父加载器反馈自己无法加载请求时，子加载器才会尝试自己加载。

使用双亲委派模型来组织类加载器之间的关系，有一个显而易见的好处就是JAVA类随着它的类加载器一起具备一种带有优先级的层次关系。例如类java.lang.Object，它存放在rt.jar之中，无论哪一个类加载器要加载这个类，最终都是委派给处于模型最顶端的启动类加载器进行加载，因此Object类在程序的各种类加载器环境中都是同一个类。

实现双新委派的代码都集中在java.lang.ClassLoader的loadClass()方法之中：先检查是否已经被加载过，若没有加载则调用父加载器的loadClass()方法， 若父类加载为空则默认使用启动类加载器作为父加载器，如果父类加载失败，抛出ClassNotFoundException异常，再调用自己的findClass()方法进行加载。

双亲委派模型并不是一个强制性约束模型，而是JAVA设计者推荐给开发者们的类加载实现

1. 虚拟机字节码执行引擎

JAVA虚拟机规范中制定了虚拟机字节码执行引擎的概念模型，这个模型成为各种虚拟机执行紫苑的统一外观。在不同的虚拟实现里面，执行引擎在执行JAVA代码的时候可能会有解释执行和编译执行两种选择，也可能两者兼备。

* 1. 运行时栈帧结构

栈帧是用于支持虚拟机进行方法调用和方法执行的数据结构。它是虚拟机运行时数据区的虚拟机栈的栈元素。栈存储了方法的局部变量表、操作数栈、动态连接和方法返回地址等信息。每一个方法从调用开始到执行完成的过程，都对应着一个栈帧在虚拟机里面从入栈到出栈的过程。

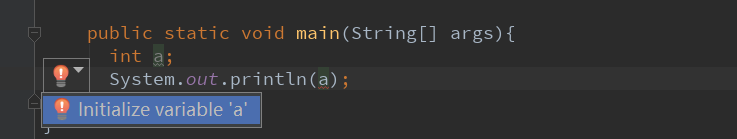
一个线程的方法调用链可能会很长，很多方法都同时处于执行状态，对于执行引擎来讲，活动线程中，只有栈顶的栈帧才是有效的，称为当前栈帧，这个栈帧所关联的方法称为当前方法，执行引擎所运行的所有字节码指令都只针对当前栈帧进行操作。

* + 1. 局部变量表

局部变量表是一组变量值存储空间，用于存放方法参数和方法内部定义的局部变量。在生成Class文件时，方法Code属性的max\_locals数据项中确定了该方法所需分的最大局部变量表的容量。

方法执行时，虚拟机是使用局部变量表完成参数值到参数列表的传递过程的，如果是实例方法，那么局部变量表中第0位索引的Slot默认是用于传递方法怕属对象实例的引用。在方法中使用this来访问。其余参数则按照参数表的顺序来排列，占用从1开始的局部变量Slot，参数表分配完成后，再根据方法体内部定义的变量顺序和作用域分配其余的Slot.

局部变量表不像类变量那样存在“准备阶段”。局部变量定义了但没有初始化的变量是不可以使用的。



* + 1. 操作数栈

操作数栈也称操作栈。它是一个后进先出栈。最大深度在编译的时候写入到code属性的max\_stacks数据项中。

当一个方法开始执行的时候，这个方法的操作数栈是空的，在方法执行过程中，会有各种字节码指令向操作栈写入或读取内容。

* + 1. 动态连接

每个栈帧都包含一个指向运行时常量池中该栈帧所属方法的引用，持有这个引用是为了支持方法调用过程中的动态连接（有一些常量池中的符号引用在方法第一次运行时才解析转化为直接引用）。

* + 1. 方法返回地址

方法退出一般有遇到返回指令的正常退出还有出现异常时的异常退出，而退出方法的本质是方法栈的栈帧出栈，此时要恢复栈帧和寄存器的值。所以需要保存这部分信息。

* 1. 方法调用

方法调用并不同于执行。它的任务只是确实被调用方法的版本。

* + 1. 解析

所有方法调用中的目标方法在Class文件里都仅仅是一个常量池的符号引用，在类加载的解析阶段，会将其中的一部分符号引用转化为直接引用，这种解析的前提是：方法在程序真正运行之前就有一个可确定的调用版本，并且这个方法的调用版本在运行期是不可以改变的。也就是说调用目标在程序代表写好、编译进行时就必需确定下来，这类方法调用称为解析。

Java虚拟机提供了四条方法调用字节码指令：

Invokestatic 调用静态方法

Invokespecial 调用实例构造器、私有方法、父类方法

Invokevirtual 调用所有的虚方法

Invokeinterface 调用接口方法，会在运行时确实一个实现此接口的对象。

能被invokestatic和invokespecial指令调用的方法，都可以在解析阶段确定唯一的调用版本，符合这个条件的有静态方法、私有方法、实例构造器、父类方法四类，它们都在类加载的时候就会把符号引用解析为该方法的直接引用，这些方法被称为非虚方法。另外被final修饰的方法，虽然使用invokevirtual指令来调用，但是是无法覆盖的，没有其它版本，所以无需对方法接收者进行多态选择。

* + 1. 分派
* 重载与静态分派

package com.learn;

public class StaticDispatch {

static abstract class Human{}

static class Man extends Human{}

static class Woman extends Human{}

public void sayHello(Human human){

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

}

public void sayHello(Man guy){

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

}

public void sayHello(Woman woman){

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

}

public static void main(String[] args){

Human man = new Man();

Human woman = new Woman();

StaticDispatch sd = new StaticDispatch();

sd.sayHello(man);

sd.sayHello(woman);

sd.sayHello((Man) man);

sd.sayHello((Woman) woman);

}

/\*\*

\* hello human

\* hello human

\* Hello guy

\* hello woman

\*/

}

public static void main(java.lang.String[]);

descriptor: ([Ljava/lang/String;)V

flags: ACC\_PUBLIC, ACC\_STATIC

Code:

stack=2, locals=4, args\_size=1

0: new #7 // class StaticDispatch$Man

3: dup

4: invokespecial #8 // Method StaticDispatch$Man."<init>":()V

7: astore\_1

8: new #9 // class StaticDispatch$Woman

11: dup

12: invokespecial #10 // Method StaticDispatch$Woman."<init>":()V

15: astore\_2

16: new #11 // class StaticDispatch

19: dup

20: invokespecial #12 // Method "<init>":()V

23: astore\_3

24: aload\_3

25: aload\_1

26: invokevirtual #13 // Method sayHello:(LStaticDispatch$Human;)V

29: aload\_3

30: aload\_2

31: invokevirtual #13 // Method sayHello:(LStaticDispatch$Human;)V

34: aload\_3

35: aload\_1

36: checkcast #7 // class StaticDispatch$Man

39: invokevirtual #14 // Method sayHello:(LStaticDispatch$Man;)V

42: aload\_3

43: aload\_2

44: checkcast #9 // class StaticDispatch$Woman

47: invokevirtual #15 // Method sayHello:(LStaticDispatch$Woman;)V

50: return

LineNumberTable:

line 16: 0

line 17: 8

line 18: 16

line 19: 24

line 20: 29

line 21: 34

line 22: 42

line 23: 50

使用哪个重载版本，完全取决于传入参数的数量和数据类型。我们从运行结果发现虚拟机在重载时是通过参数的静态类型而不是实际类型作为判定依据的。从javap的结果来看，在编译时就已经确定了调用版本

package com.learn;

import java.io.Serializable;

public class Overload {

public static void sayHello(Object arg){

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

}

public static void sayHello(int arg){

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

}

public static void sayHello(long arg){

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

}

public static void sayHello(Serializable arg){

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

}

public static void sayHello(Character arg){

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

}

public static void sayHello(char... args){

System.out.println("Hello char[]");

}

public static void sayHello(char arg){

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

}

public static void main(String[] args){

sayHello('a');

}

/\*\*依次注释执行结果顺序

\* Hello char

\* Hello int

\* Hello long

\* Hello Character

\* Hello Serializable

\* Hello object

\* Hello char[]

\*/

}

* 重写与动态分派

package com.learn;

public class DynamicDispatch {

static abstract class Human {

protected abstract void sayHello();

}

static class Man extends Human {

@Override

protected void sayHello() {

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

}

}

static class Woman extends Human {

@Override

protected void sayHello() {

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

}

}

public static void main(String[] args){

Human man = new Man();

Human woman = new Woman();

man.sayHello();

woman.sayHello();

}

}

从javap的结果来看，在编译时虚拟机根本就不知道调用哪个方法，因为字节码的指令和参数都是相同的。

public static void main(java.lang.String[]);

descriptor: ([Ljava/lang/String;)V

flags: ACC\_PUBLIC, ACC\_STATIC

Code:

stack=2, locals=3, args\_size=1

0: new #2 // class DynamicDispatch$Man

3: dup

4: invokespecial #3 // Method DynamicDispatch$Man."<init>":()V

7: astore\_1

8: new #4 // class DynamicDispatch$Woman

11: dup

12: invokespecial #5 // Method DynamicDispatch$Woman."<init>":()V

15: astore\_2

16: aload\_1

17: invokevirtual #6 // Method DynamicDispatch$Human.sayHello:()V

20: aload\_2

21: invokevirtual #6 // Method DynamicDispatch$Human.sayHello:()V

24: return

LineNumberTable:

line 19: 0

line 20: 8

line 21: 16

line 22: 20

line 23: 24

* 单分派与多分派

public class Dispatch {

static class QQ {}

static class ALI {}

public static class Father {

public void hard(QQ arg){

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

}

public void hard(ALI arg){

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

}

}

public static class Son extends Father {

public void hard(QQ arg){

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

}

public void hard(ALI arg){

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

}

}

public static void main(String[] args){

Father father = new Father();

Father son = new Son();

father.hard(new QQ());

son.hard(new QQ());

}

}

从javap可以看出，重载方法在编译阶段就已经确定好了，而重写方法在运行时指定。

public static void main(java.lang.String[]);

descriptor: ([Ljava/lang/String;)V

flags: ACC\_PUBLIC, ACC\_STATIC

Code:

stack=3, locals=3, args\_size=1

0: new #2 // class Dispatch$Father

3: dup

4: invokespecial #3 // Method Dispatch$Father."<init>":()V

7: astore\_1

8: new #4 // class Dispatch$Son

11: dup

12: invokespecial #5 // Method Dispatch$Son."<init>":()V

15: astore\_2

16: aload\_1

17: new #6 // class Dispatch$QQ

20: dup

21: invokespecial #7 // Method Dispatch$QQ."<init>":()V

24: invokevirtual #8 // Method Dispatch$Father.hard:(LDispatch$QQ;)V

27: aload\_2

28: new #6 // class Dispatch$QQ

31: dup

32: invokespecial #7 // Method Dispatch$QQ."<init>":()V

35: invokevirtual #8 // Method Dispatch$Father.hard:(LDispatch$QQ;)V

38: return

LineNumberTable:

line 23: 0

line 24: 8

line 25: 16

line 26: 27

line 27: 38

由于动态分派是非常频繁的动作，为了提高搜索效率，最常用的优化手段就是在类的方法区中建立一个虚方法表，使用虚方法表索引来代替元数据的查找以提高性能。

* 1. 基于栈的指令集与解释器

JAVA编译器输出的指令流，基本上是一种基于栈的指令集架构，指令流里面的指令大部分都是零地址指令，它们依赖操作树栈进行工作。与之相对的另外一套常用的指令集就是寄存器指令集，最典型的就是X86的二地址指令集。基于栈的指令集的优点就是可移植性，寄存器由硬件直接提供，程序直接依赖这些硬件寄存器则不可避免地要受到硬件的约束。栈指令集的主要缺点是执行速度相对来说会慢一些。

* 1. 字节码生成与动态代表

在Spring中，有非常多使用动态代理来对Bean增加的例子，动态代表的所谓动态是针对使用JAVA代码实际编写了代表类的静态代表而言的。在原始类和接口还未知的时候，就确定代理类的行为，当代理类与原始类脱离直接联系后，就可以很灵活地重用于不同的应用场景中。

package com.learn;

import java.lang.reflect.InvocationHandler;

import java.lang.reflect.Method;

import java.lang.reflect.Proxy;

public class DynamicProxyTest {

interface IHello {void sayHello();}

static class Hello implements IHello {

public void sayHello(){

System.out.println("Hello");

}

}

static class DynamicProxy implements InvocationHandler {

Object originalObj;

Object bind(Object originalObj){

this.originalObj = originalObj;

return Proxy.newProxyInstance(originalObj.getClass().getClassLoader(),originalObj.getClass().getInterfaces(),this);

}

@Override

public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {

System.out.println("welcome");

return method.invoke(originalObj,args);

}

}

public static void main(String[] args){

System.getProperties().put("sun.misc.ProxyGenerator.saveGeneratedFiles","true");

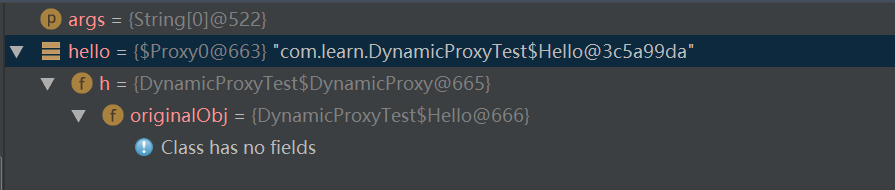
IHello hello = (IHello) new DynamicProxy().bind(new Hello());

hello.sayHello();

}

}

从调试结果来看，hello是一个代理对象，在调用sayHello对象的时候，仅仅是执行了代理对象的sayHello方法。代理会生成一个$Proxy0.class文件。通过反编译可以看到这个代理做了什么。



代理反编译结果：

package com.learn;

import java.lang.reflect.InvocationHandler;

import java.lang.reflect.Method;

import java.lang.reflect.Proxy;

public class DynamicProxyTest {

interface IHello {void sayHello();}

static class Hello implements IHello {

public void sayHello(){

System.out.println("Hello");

}

}

static class DynamicProxy implements InvocationHandler {

Object originalObj;

Object bind(Object originalObj){

this.originalObj = originalObj;

return Proxy.newProxyInstance(originalObj.getClass().getClassLoader(),originalObj.getClass().getInterfaces(),this);

}

@Override

public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {

System.out.println("welcome");

return method.invoke(originalObj,args);

}

}

public static void main(String[] args){

System.getProperties().put("sun.misc.ProxyGenerator.saveGeneratedFiles","true");

IHello hello = (IHello) new DynamicProxy().bind(new Hello());

hello.sayHello();

}

}