# 设计模式

## 1.1单例模式

### 1.1.1饿汉单例

class EagerSingleton {

private static final EagerSingleton instance = new EagerSingleton();

private EagerSingleton() { }

public static EagerSingleton getInstance() {

return instance;

}

}

特点：类被加载的时候，单例类的唯一实例将被创建

### 1.1.2懒汉单例

class LazySingleton {

private volatile static LazySingleton instance = null;

private LazySingleton() { }

public static LazySingleton getInstance() {

//第一重判断

if (instance == null) {

//锁定代码块

synchronized (LazySingleton.class) {

//第二重判断

if (instance == null) {

instance = new LazySingleton(); //创建单例实例

}

}

}

return instance;

}

}

特点：

由于volatile关键字会屏蔽Java虚拟机所做的一些代码优化，可能会导致系统运行效率降低

比较：

饿汉式单例类在类被加载时就将自己实例化，它的优点在于无须考虑多线程访问问题，可以确保实例的唯一性；从调用速度和反应时间角度来讲，由于单例对象一开始就得以创建，因此要优于懒汉式单例。但是无论系统在运行时是否需要使用该单例对象，由于在类加载时该对象就需要创建，因此从资源利用效率角度来讲，饿汉式单例不及懒汉式单例，而且在系统加载时由于需要创建饿汉式单例对象，加载时间可能会比较长。+

懒汉式单例类在第一次使用时创建，无须一直占用系统资源，实现了延迟加载，但是必须处理好多个线程同时访问的问题，特别是当单例类作为资源控制器，在实例化时必然涉及资源初始化，而资源初始化很有可能耗费大量时间，这意味着出现多线程同时首次引用此类的机率变得较大，需要通过双重检查锁定等机制进行控制，这将导致系统性能受到一定影响。

### 1.1.3 IoDH

class Singleton {

private Singleton() {

}

private static class HolderClass {

private final static Singleton instance = new Singleton();

}

public static Singleton getInstance() {

return HolderClass.instance;

}

}

## 1.2工厂模式

### 1.2.1简单工厂模式

//抽象图表接口：抽象产品类

interface Chart {

public void display();

}

//柱状图类：具体产品类

class HistogramChart implements Chart {

public HistogramChart() {

System.out.println("创建柱状图！");

}

public void display() {

System.out.println("显示柱状图！");

}

}

//饼状图类：具体产品类

class PieChart implements Chart {

public PieChart() {

System.out.println("创建饼状图！");

}

public void display() {

System.out.println("显示饼状图！");

}

}

//折线图类：具体产品类

class LineChart implements Chart {

public LineChart() {

System.out.println("创建折线图！");

}

public void display() {

System.out.println("显示折线图！");

}

}

//图表工厂类：工厂类

class ChartFactory {

//静态工厂方法

public static Chart getChart(String type) {

Chart chart = null;

if (type.equalsIgnoreCase("histogram")) {

chart = new HistogramChart();

System.out.println("初始化设置柱状图！");

}

else if (type.equalsIgnoreCase("pie")) {

chart = new PieChart();

System.out.println("初始化设置饼状图！");

}

else if (type.equalsIgnoreCase("line")) {

chart = new LineChart();

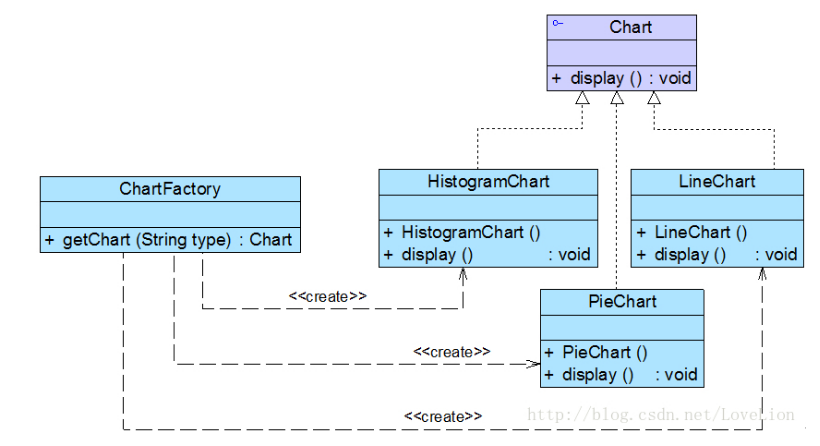
System.out.println("初始化设置折线图！");

}

return chart;

}

}



**特点：**

简单工厂模式最大的缺点是当有新产品要加入到系统中时，必须修改工厂类，需要在其中加入必要的业务逻辑，这违背了“开闭原则”。此外，在简单工厂模式中，所有的产品都由同一个工厂创建，工厂类职责较重，业务逻辑较为复杂，具体产品与工厂类之间的耦合度高，严重影响了系统的灵活性和扩展性，而工厂方法模式则可以很好地解决这一问题

### 1.2.2工厂方法模式

//日志记录器接口：抽象产品

interface Logger {

public void writeLog();

}

//数据库日志记录器：具体产品

class DatabaseLogger implements Logger {

public void writeLog() {

System.out.println("数据库日志记录。");

}

}

//文件日志记录器：具体产品

class FileLogger implements Logger {

public void writeLog() {

System.out.println("文件日志记录。");

}

}

//日志记录器工厂接口：抽象工厂

interface LoggerFactory {

public Logger createLogger();

}

//数据库日志记录器工厂类：具体工厂

class DatabaseLoggerFactory implements LoggerFactory {

public Logger createLogger() {

//连接数据库，代码省略

//创建数据库日志记录器对象

Logger logger = new DatabaseLogger();

//初始化数据库日志记录器，代码省略

return logger;

}

}

//文件日志记录器工厂类：具体工厂

class FileLoggerFactory implements LoggerFactory {

public Logger createLogger() {

//创建文件日志记录器对象

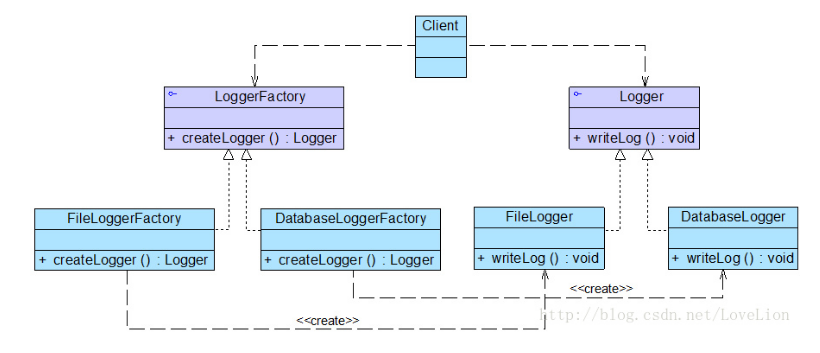
Logger logger = new FileLogger();

//创建文件，代码省略

return logger;

}

}



**工厂方法模式的主要优点如下**：

(1) 在工厂方法模式中，工厂方法用来创建客户所需要的产品，同时还向客户隐藏了哪种具体产品类将被实例化这一细节，用户只需要关心所需产品对应的工厂，无须关心创建细节，甚至无须知道具体产品类的类名。

(2) 基于工厂角色和产品角色的多态性设计是工厂方法模式的关键。它能够让工厂可以自主确定创建何种产品对象，而如何创建这个对象的细节则完全封装在具体工厂内部。工厂方法模式之所以又被称为多态工厂模式，就正是因为所有的具体工厂类都具有同一抽象父类。

(3) 使用工厂方法模式的另一个优点是在系统中加入新产品时，无须修改抽象工厂和抽象产品提供的接口，无须修改客户端，也无须修改其他的具体工厂和具体产品，而只要添加一个具体工厂和具体产品就可以了，这样，系统的可扩展性也就变得非常好，完全符合“开闭原则”。

**工厂方法模式的主要缺点如下：**

(1) 在添加新产品时，需要编写新的具体产品类，而且还要提供与之对应的具体工厂类，系统中类的个数将成对增加，在一定程度上增加了系统的复杂度，有更多的类需要编译和运行，会给系统带来一些额外的开销。

(2) 由于考虑到系统的可扩展性，需要引入抽象层，在客户端代码中均使用抽象层进行定义，增加了系统的抽象性和理解难度，且在实现时可能需要用到DOM、反射等技术，增加了系统的实现难度。

### 1.2.3抽象工厂模式

//按钮接口：抽象产品

interface Button {

public void display();

}

//Spring按钮类：具体产品

class SpringButton implements Button {

public void display() {

System.out.println("显示浅绿色按钮。");

}

}

//Summer按钮类：具体产品

class SummerButton implements Button {

public void display() {

System.out.println("显示浅蓝色按钮。");

}

}

//文本框接口：抽象产品

interface TextField {

public void display();

}

//Spring文本框类：具体产品

class SpringTextField implements TextField {

public void display() {

System.out.println("显示绿色边框文本框。");

}

}

//Summer文本框类：具体产品

class SummerTextField implements TextField {

public void display() {

System.out.println("显示蓝色边框文本框。");

}

}

//组合框接口：抽象产品

interface ComboBox {

public void display();

}

//Spring组合框类：具体产品

class SpringComboBox implements ComboBox {

public void display() {

System.out.println("显示绿色边框组合框。");

}

}

//Summer组合框类：具体产品

class SummerComboBox implements ComboBox {

public void display() {

System.out.println("显示蓝色边框组合框。");

}

}

//界面皮肤工厂接口：抽象工厂

interface SkinFactory {

public Button createButton();

public TextField createTextField();

public ComboBox createComboBox();

}

//Spring皮肤工厂：具体工厂

class SpringSkinFactory implements SkinFactory {

public Button createButton() {

return new SpringButton();

}

public TextField createTextField() {

return new SpringTextField();

}

public ComboBox createComboBox() {

return new SpringComboBox();

}

}

//Summer皮肤工厂：具体工厂

class SummerSkinFactory implements SkinFactory {

public Button createButton() {

return new SummerButton();

}

public TextField createTextField() {

return new SummerTextField();

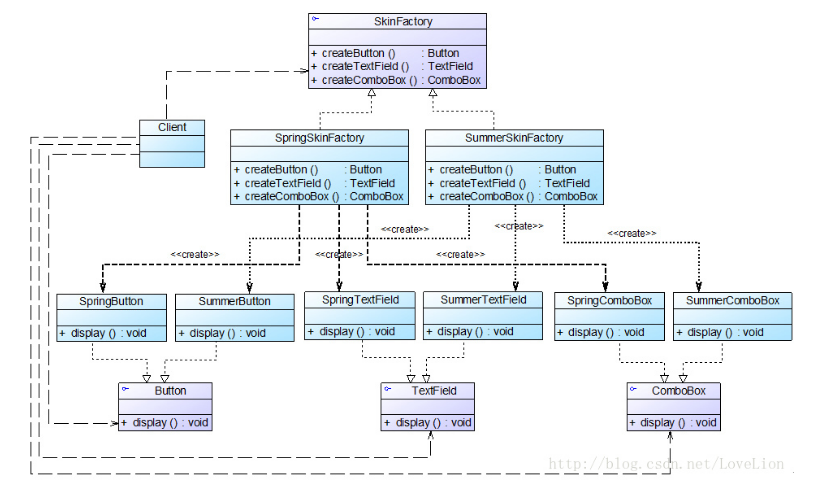
}

public ComboBox createComboBox() {

return new SummerComboBox();

}

}



**抽象工厂模式的主要优点如下：**

(1) 抽象工厂模式隔离了具体类的生成，使得客户并不需要知道什么被创建。由于这种隔离，更换一个具体工厂就变得相对容易，所有的具体工厂都实现了抽象工厂中定义的那些公共接口，因此只需改变具体工厂的实例，就可以在某种程度上改变整个软件系统的行为。

(2) 当一个产品族中的多个对象被设计成一起工作时，它能够保证客户端始终只使用同一个产品族中的对象。

(3) 增加新的产品族很方便，无须修改已有系统，符合“开闭原则”。

**抽象工厂模式的主要缺点如下：**

增加新的产品等级结构麻烦，需要对原有系统进行较大的修改，甚至需要修改抽象层代码，这显然会带来较大的不便，违背了“开闭原则”。

**在以下情况下可以考虑使用抽象工厂模式：**

(1) 一个系统不应当依赖于产品类实例如何被创建、组合和表达的细节，这对于所有类型的工厂模式都是很重要的，用户无须关心对象的创建过程，将对象的创建和使用解耦。

(2) 系统中有多于一个的产品族，而每次只使用其中某一产品族。可以通过配置文件等方式来使得用户可以动态改变产品族，也可以很方便地增加新的产品族。

(3) 属于同一个产品族的产品将在一起使用，这一约束必须在系统的设计中体现出来。同一个产品族中的产品可以是没有任何关系的对象，但是它们都具有一些共同的约束，如同一操作系统下的按钮和文本框，按钮与文本框之间没有直接关系，但它们都是属于某一操作系统的，此时具有一个共同的约束条件：操作系统的类型。

(4) 产品等级结构稳定，设计完成之后，不会向系统中增加新的产品等级结构或者删除已有的产品等级结构。

## 1.3观察者模式

//抽象观察类

interface Observer {

public String getName();

public void setName(String name);

public void help(); //声明支援盟友方法

public void beAttacked(AllyControlCenter acc); //声明遭受攻击方法

}

//战队成员类：具体观察者类

class Player implements Observer {

private String name;

public Player(String name) {

this.name = name;

}

public void setName(String name) {

this.name = name;

}

public String getName() {

return this.name;

}

//支援盟友方法的实现

public void help() {

System.out.println("坚持住，" + this.name + "来救你！");

}

//遭受攻击方法的实现，当遭受攻击时将调用战队控制中心类的通知方法notifyObserver()来通知盟友

public void beAttacked(AllyControlCenter acc) {

System.out.println(this.name + "被攻击！");

acc.notifyObserver(name);

}

}

//战队控制中心类：目标类

abstract class AllyControlCenter {

protected String allyName; //战队名称

protected ArrayList<Observer> players = new ArrayList<Observer>(); //定义一个集合用于存储战队成员

public void setAllyName(String allyName) {

this.allyName = allyName;

}

public String getAllyName() {

return this.allyName;

}

//注册方法

public void join(Observer obs) {

System.out.println(obs.getName() + "加入" + this.allyName + "战队！");

players.add(obs);

}

//注销方法

public void quit(Observer obs) {

System.out.println(obs.getName() + "退出" + this.allyName + "战队！");

players.remove(obs);

}

//声明抽象通知方法

public abstract void notifyObserver(String name);

}

//具体战队控制中心类：具体目标类

class ConcreteAllyControlCenter extends AllyControlCenter {

public ConcreteAllyControlCenter(String allyName) {

System.out.println(allyName + "战队组建成功！");

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

this.allyName = allyName;

}

//实现通知方法

public void notifyObserver(String name) {

System.out.println(this.allyName + "战队紧急通知，盟友" + name + "遭受敌人攻击！");

//遍历观察者集合，调用每一个盟友（自己除外）的支援方法

for(Object obs : players) {

if (!((Observer)obs).getName().equalsIgnoreCase(name)) {

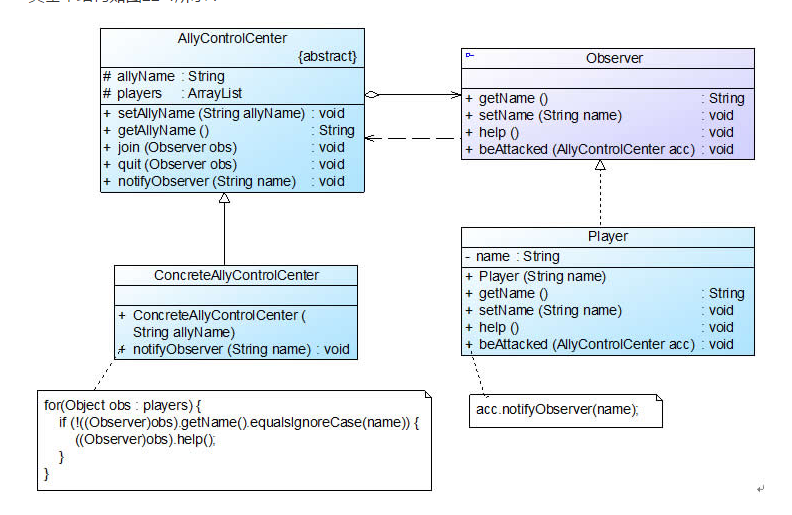
((Observer)obs).help();

}

}

}

}



Player.beAttacked() --> AllyControlCenter.notifyObserver() -->Player.help()。

**观察者模式的主要优点如下：**

(1) 观察者模式可以实现表示层和数据逻辑层的分离，定义了稳定的消息更新传递机制，并抽象了更新接口，使得可以有各种各样不同的表示层充当具体观察者角色。

(2) 观察者模式在观察目标和观察者之间建立一个抽象的耦合。观察目标只需要维持一个抽象观察者的集合，无须了解其具体观察者。由于观察目标和观察者没有紧密地耦合在一起，因此它们可以属于不同的抽象化层次。

(3) 观察者模式支持广播通信，观察目标会向所有已注册的观察者对象发送通知，简化了一对多系统设计的难度。

(4) 观察者模式满足“开闭原则”的要求，增加新的具体观察者无须修改原有系统代码，在具体观察者与观察目标之间不存在关联关系的情况下，增加新的观察目标也很方便。

**观察者模式的主要缺点如下：**

(1) 如果一个观察目标对象有很多直接和间接观察者，将所有的观察者都通知到会花费很多时间。

(2) 如果在观察者和观察目标之间存在循环依赖，观察目标会触发它们之间进行循环调用，可能导致系统崩溃。

(3) 观察者模式没有相应的机制让观察者知道所观察的目标对象是怎么发生变化的，而仅仅只是知道观察目标发生了变化。

在以下情况下可以考虑使用观察者模式：

(1) 一个抽象模型有两个方面，其中一个方面依赖于另一个方面，将这两个方面封装在独立的对象中使它们可以各自独立地改变和复用。

(2) 一个对象的改变将导致一个或多个其他对象也发生改变，而并不知道具体有多少对象将发生改变，也不知道这些对象是谁。

(3) 需要在系统中创建一个触发链，A对象的行为将影响B对象，B对象的行为将影响C对象……，可以使用观察者模式创建一种链式触发机制。

## 1.4 建造者模式

class Actor

{

private String type; //角色类型

private String sex; //性别

private String face; //脸型

private String costume; //服装

private String hairstyle; //发型

public void setType(String type) {

this.type = type;

}

public void setSex(String sex) {

this.sex = sex;

}

public void setFace(String face) {

this.face = face;

}

public void setCostume(String costume) {

this.costume = costume;

}

public void setHairstyle(String hairstyle) {

this.hairstyle = hairstyle;

}

public String getType() {

return (this.type);

}

public String getSex() {

return (this.sex);

}

public String getFace() {

return (this.face);

}

public String getCostume() {

return (this.costume);

}

public String getHairstyle() {

return (this.hairstyle);

}

}

//角色建造器：抽象建造者

abstract class ActorBuilder

{

protected Actor actor = new Actor();

public abstract void buildType();

public abstract void buildSex();

public abstract void buildFace();

public abstract void buildCostume();

public abstract void buildHairstyle();

//工厂方法，返回一个完整的游戏角色对象

public Actor createActor()

{

return actor;

}

}

//英雄角色建造器：具体建造者

class HeroBuilder extends ActorBuilder

{

public void buildType()

{

actor.setType("英雄");

}

public void buildSex()

{

actor.setSex("男");

}

public void buildFace()

{

actor.setFace("英俊");

}

public void buildCostume()

{

actor.setCostume("盔甲");

}

public void buildHairstyle()

{

actor.setHairstyle("飘逸");

}

}

//天使角色建造器：具体建造者

class AngelBuilder extends ActorBuilder

{

public void buildType()

{

actor.setType("天使");

}

public void buildSex()

{

actor.setSex("女");

}

public void buildFace()

{

actor.setFace("漂亮");

}

public void buildCostume()

{

actor.setCostume("白裙");

}

public void buildHairstyle()

{

actor.setHairstyle("披肩长发");

}

}

//恶魔角色建造器：具体建造者

class DevilBuilder extends ActorBuilder

{

public void buildType()

{

actor.setType("恶魔");

}

public void buildSex()

{

actor.setSex("妖");

}

public void buildFace()

{

actor.setFace("丑陋");

}

public void buildCostume()

{

actor.setCostume("黑衣");

}

public void buildHairstyle()

{

actor.setHairstyle("光头");

}

}

//游戏角色创建控制器：指挥者

class ActorController

{

//逐步构建复杂产品对象

public Actor construct(ActorBuilder ab)

{

Actor actor;

ab.buildType();

ab.buildSex();

ab.buildFace();

ab.buildCostume();

ab.buildHairstyle();

actor=ab.createActor();

return actor;

}

}

ActorBuilder ab; //针对抽象建造者编程

ab = (ActorBuilder)XMLUtil.getBean(); //反射生成具体建造者对象

ActorController ac = new ActorController();

Actor actor;

actor = ac.construct(ab); //通过指挥者创建完整的建造者对象

# Java

## 2.1 [TCP/UDP区别以及UDP如何实现可靠传输](http://blog.csdn.net/ls5718/article/details/52141571)

### 2.1.1区别

1、TCP面向连接（如打电话要先拨号建立连接）;UDP是无连接的，即发送数据之前不需要建立连接

2、TCP提供可靠的服务。也就是说，通过TCP连接传送的数据，无差错，不丢失，不重复，且按序到达;UDP尽最大努力交付，即不保证可靠交付  
3、TCP面向字节流，实际上是TCP把数据看成一连串无结构的字节流;UDP是面向报文的  
UDP没有拥塞控制，因此网络出现拥塞不会使源主机的发送速率降低（对实时应用很有用，如IP电话，实时视频会议等）  
4、每一条TCP连接只能是点到点的;UDP支持一对一，一对多，多对一和多对多的交互通信  
5、TCP首部开销20字节;UDP的首部开销小，只有8个字节

6、TCP的逻辑通信信道是全双工的可靠信道，UDP则是不可靠信道

### 2.1.2保证可靠传输

要使用UDP来构建可靠的面向连接的数据传输，就要实现类似于TCP协议的

超时重传（定时器）

有序接受 （添加包序号）

应答确认 （Seq/Ack应答机制）

滑动窗口流量控制等机制 （滑动窗口协议）

等于说要在传输层的上一层（或者直接在应用层）实现TCP协议的可靠数据传输机制，比如使用UDP数据包+序列号，UDP数据包+时间戳等方法。

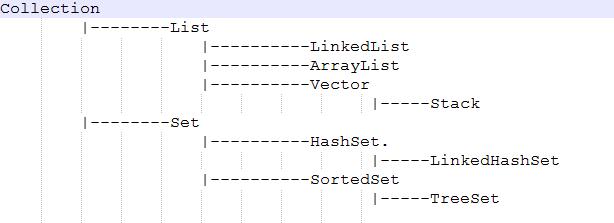
## 2.2ArrayList、LinkedList、Vector的区别

ArrayList 本质上是一个可改变大小的****数组****.当元素加入时,其大小将会动态地增长.内部的元素可以直接通过get与set方法进行访问.元素顺序存储 ,****随机访问很快，删除非头尾元素慢，新增元素慢而且费资源**** ,较适用于无频繁增删的情况 ,比数组效率低，如果不是需要可变数组，可考虑使用数组 ,****非线程安全****.

LinkedList 是一个****双链表****,在添加和删除元素时具有比ArrayList更好的性能.但在get与set方面弱于ArrayList. 适用于 ：没有大规模的随机读取，大量的增加/删除操作.****随机访问很慢，增删操作很快****，不耗费多余资源 ,允许null元素,****非线程安全.****

Vector （类似于ArrayList）但其是****同步****的，开销就比ArrayList要大。如果你的程序本身是线程安全的，那么使用ArrayList是更好的选择。 Vector和ArrayList在更多元素添加进来时会请求更大的空间。Vector每次请求其大小的双倍空间，而ArrayList每次对size增长50%.

## 2.3Collection包结构，与Collections的区别



Collection是集合继承结构中的顶层接口

Collections 是提供了对集合进行操作的强大方法的工具类 ，它包含有各种有关集合操作的静态多态方法。此类不能实例化

## 2.4HashMap和HashTable的区别

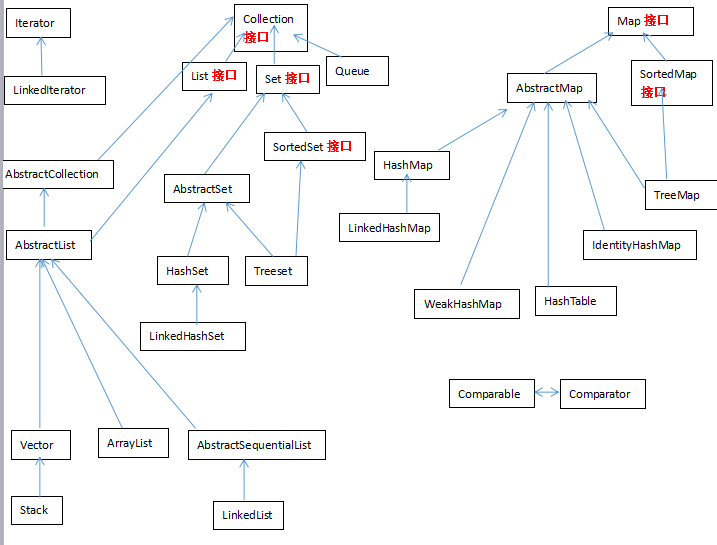
### 2.4.1HashMap

1. 键和值都可以为空
2. 非线程安全的
3. 不能保证随着时间的推移Map中的元素次序是不变的

### 2.4.2HashTable

1. 键和值都不能为空
2. 线程安全的
3. 多次访问，映射元素的位置相同

## 2.5Map、Set、List、Queue、Stack的特点与用法



### 2.5.1Map

键映射到值的对象。一个映射不能包含重复的键；每个键最多只能映射到一个值。

某些映射实现可明确保证其顺序，如 TreeMap 类；另一些映射实现则不保证顺序，如 HashMap 类。

Map中元素，可以将key序列、value序列单独抽取出来。 使用keySet()抽取key序列，将map中的所有keys生成一个Set。 使用values()抽取value序列，将map中的所有values生成一个Collection。 为什么一个生成Set，一个生成Collection？那是因为，key总是独一无二的，value允许重复。

### 2.5.2Set

一个不包含重复元素的 collection。更确切地讲，set 不包含满足 e1.equals(e2) 的元素对 e1 和 e2，并且最多包含一个 null 元素。

不可随机访问包含的元素

只能用Iterator实现单向遍历

Set 没有同步方法

### 2.5.3List

可随机访问包含的元素 元素是有序的 可在任意位置增、删元素 不管访问多少次，元素位置不变 允许重复元素 用Iterator实现单向遍历，也可用

ListIterator实现双向遍历

### 2.5.4Queue

先进先出

Queue使用时要尽量避免Collection的add()和remove()方法，而是要使用offer()来加入元素，使用poll()来获取并移出元素。它们的优点是通过返回值可以判断成功与否，add()和remove()方法在失败的时候会抛出异常。 如果要使用前端而不移出该元素，使用element()或者peek()方法。 值得注意的是LinkedList类实现了Queue接口，因此我们可以把LinkedList当成Queue来用。

Queue 实现通常不允许插入 null 元素，尽管某些实现（如 LinkedList）并不禁止插入 null。即使在允许 null 的实现中，也不应该将 null 插入到 Queue 中，因为 null 也用作 poll 方法的一个特殊返回值，表明队列不包含元素。

### 2.5.5Stack

后进先出

Stack继承自Vector（可增长的对象数组），也是同步的 它通过五个操作对类 Vector 进行了扩展 ，允许将向量视为堆栈。它提供了通常的 push 和 pop 操作，以及取堆栈顶点的 peek 方法、测试堆栈是否为空的 empty 方法、在堆栈中查找项并确定到堆栈顶距离的 search 方法。

### 2.5.6用法

如果涉及到堆栈、队列等操作，应该考虑用List；

对于需要快速插入，删除元素，应该使用LinkedList；

如果需要快速随机访问元素，应该使用ArrayList。

如果程序在单线程环境中，或者访问仅仅在一个线程中进行，考虑非同步的类，其效率较高

## 2.6Java线程

### 2.6.1线程的几种可用状态：

1. 新建( new )：新创建了一个线程对象。

2. 可运行( runnable )：线程对象创建后，其他线程(比如 main 线程）调用了该对象 的 start ()方法。该状态的线程位于可运行线程池中，等待被线程调度选中，获 取 cpu 的使用权 。

3. 运行( running )：可运行状态( runnable )的线程获得了 cpu 时间片（ timeslice ） ，执行程序代码。

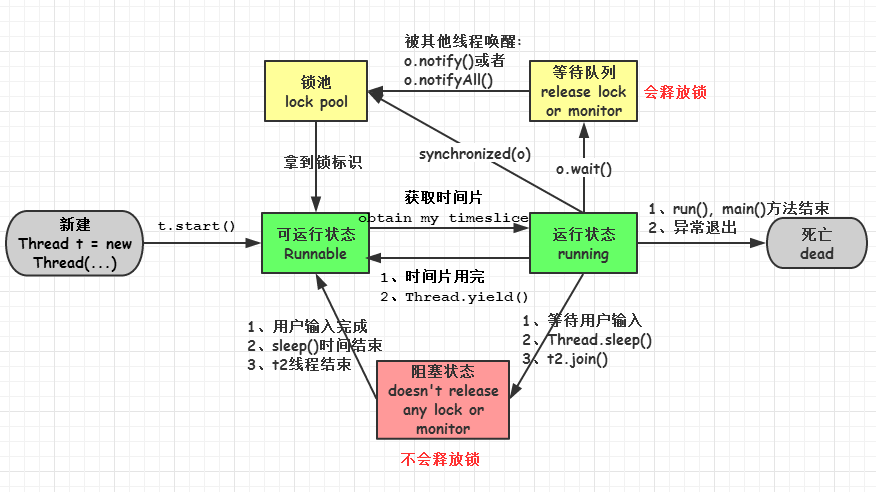
4. 阻塞( block )：阻塞状态是指线程因为某种原因放弃了 cpu 使用权，也即让出了 cpu timeslice ，暂时停止运行。直到线程进入可运行( runnable )状态，才有 机会再次获得 cpu timeslice 转到运行( running )状态。阻塞的情况分三种：

(一). 等待阻塞：运行( running )的线程执行 o . wait ()方法， JVM 会把该线程放 入等待队列( waitting queue )中。

(二). 同步阻塞：运行( running )的线程在获取对象的同步锁时，若该同步锁 被别的线程占用，则 JVM 会把该线程放入锁池( lock pool )中。

(三). 其他阻塞: 运行( running )的线程执行 Thread . sleep ( long ms )或 t . join ()方法，或者发出了 I / O 请求时， JVM 会把该线程置为阻塞状态。            当 sleep ()状态超时、 join ()等待线程终止或者超时、或者 I / O 处理完毕时，线程重新转入可运行( runnable )状态。

5. 死亡( dead )：线程 run ()、 main () 方法执行结束，或者因异常退出了 run ()方法，则该线程结束生命周期。死亡的线程不可再次复生。



### 2.6.2sleep和wait的区别：

sleep和wait的区别有：

1，这两个方法来自不同的类分别是Thread和Object

2，最主要是sleep方法没有释放锁，而wait方法释放了锁，使得敏感词线程可以使用同步控制块或者方法。

3，wait，notify和notifyAll只能在同步控制方法或者同步控制块里面使用，而sleep可以在

任何地方使用

synchronized(x){

x.notify()

//或者wait()

}

4,sleep必须捕获异常，而wait，notify和notifyAll不需要捕获异常

## 2.7类的加载顺序

(1) 父类静态代码块(包括静态初始化块，静态属性，但不包括静态方法)

(2) 子类静态代码块(包括静态初始化块，静态属性，但不包括静态方法 )

(3) 父类非静态代码块( 包括非静态初始化块，非静态属性 )

(4) 父类构造函数

(5) 子类非静态代码块 ( 包括非静态初始化块，非静态属性 )

(6) 子类构造函数

## 2.8类型转换

byte<char<short<int<long<float<double

## 2.9抽象类和接口区别

抽象类：

1.抽象类可以有构造函数，不能被实例化

2.抽象类可以有普通的成员变量，静态变量

3.抽象类可以有普通成员方法、静态方法

4.抽象类可以有抽象方法，抽象方法可以被protected、public修饰

5.抽象类中不一定有抽象方法

6.抽象类中的抽象方法需要有子类来实现，如果子类不实现，那么子类也要定义为抽象类

7.一个类只能继承一个抽象类

接口：

1.接口中不能有构造函数，不能被实例化

2.接口中的不能有普通成员变量，接口中只能有常量（被public static final修饰）

3.接口中的方法都是public的

4.一个类可以实现多个接口

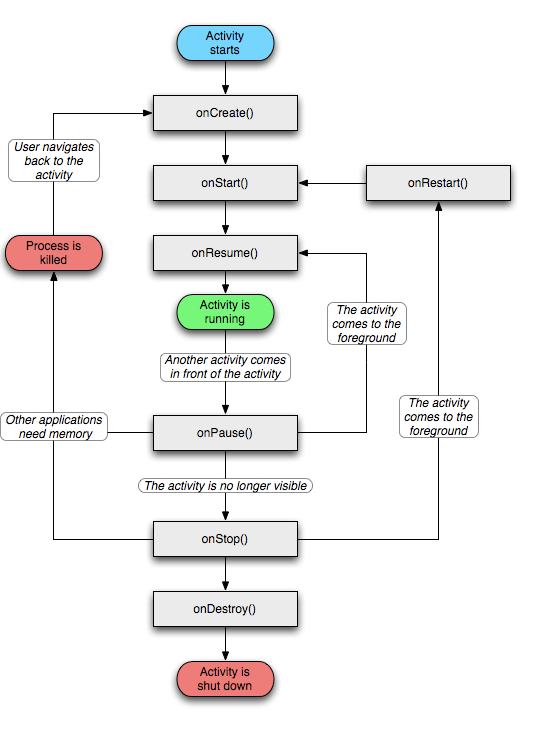
5.接口中可以有默认的default方法

6接口中定义的方法都需要有实现类来实现，如果实现类不能实现接口中的所有方法， 则实现类定义为抽象类。

# Android

## 3.1Activity

### 3.1.1生命周期



### 3.1.2Activity状态

**Active/Running:**   
Activity处于活动状态，此时Activity处于栈顶，是可见状态，可与用户进行交互。   
**Paused：**   
当Activity失去焦点时，或被一个新的非全屏的Activity，或被一个透明的Activity放置在栈顶时，Activity就转化为Paused状态。但我们需要明白，此时Activity只是失去了与用户交互的能力，其所有的状态信息及其成员变量都还存在，只有在系统内存紧张的情况下，才有可能被系统回收掉。   
**Stopped：**   
当一个Activity被另一个Activity完全覆盖时，被覆盖的Activity就会进入Stopped状态，此时它不再可见，但是跟Paused状态一样保持着其所有状态信息及其成员变量。   
**Killed：**   
当Activity被系统回收掉时，Activity就处于Killed状态。

### 3.1.3调用情况

（1）Activity启动过程中所调用的生命周期方法

Activity启动–>onCreate()–>onStart()–>onResume()依次被调用

（2）当前Activity创建完成后，按Home键回到主屏

(Activity不可见)–>onPause()–>onStop()

（3）当我们点击Home键回到主界面后，再次点击App回到Activity时

当我们再次回到原Activity时–>onRestart()–>onStart()–>onResume()

1. 当我们在原有的Activity的基础上打新的Activity时

在原Activity的基础上开启新的Activity，原Activity生命周期执行方法顺序为–>onPause()–>onStop()，事实上跟点击home键是一样的。但是这里有点要注意的是如果新的Activity使用了透明主题，那么当前Activity不会回调onStop方法。同时我们发现新Activity（SecondActivity）生命周期方法是在原Activity的onPause方法执行完成后才可以被回调，这也就是前面我们为什么说在onPause方法不能操作耗时任务的原因了

1. 点击Back键回退

退出当前Activity时–>onPause()–>onStop()–>onDestroy()

### 3.1.4进程优先级

在android系统中最重要的进程被称为前台进程，然后依次是任何可见进程、服务进程、后台进程，最后是空进程。

### 3.1.**5Android启动模式**

1. **Standard模式**

又称为标准模式，也是系统的默认模式（可以不指定），在这样模式下，每启动一个Activity都会重新创建一个Activity的新实例，并且将其加入任务栈中，而且完全不会去考虑这个实例是否已存在。

例如：

若我有一个Activity名为A1, 上面有一个按钮可跳转到A1。那么如果我点击按钮，便会新启一个Activity A1叠在刚才的A1之上，再点击，又会再新启一个在它之上……

点back键会依照栈顺序依次退出。

1. **SingleTop**

可以有多个实例，但是不允许多个相同Activity叠加。即，如果Activity在栈顶的时候，启动相同的Activity，不会创建新的实例，而会调用其onNewIntent方法。

例如：

若我有两个Activity名为B1,B2,两个Activity内容功能完全相同，都有两个按钮可以跳到B1或者B2，唯一不同的是B1为standard，B2为singleTop。

若我意图打开的顺序为B1->B2->B2，则实际打开的顺序为B1->B2（后一次意图打开B2，实际只调用了前一个的onNewIntent方法）

若我意图打开的顺序为B1->B2->B1->B2，则实际打开的顺序与意图的一致，为B1->B2->B1->B2。

1. **SingleTask**

只有一个实例。在同一个应用程序中启动他的时候，若Activity不存在，则会在当前task创建一个新的实例，若存在，则会把task中在其之上的其它Activity destory掉并调用它的onNewIntent方法。

如果是在别的应用程序中启动它，则会新建一个task，并在该task中启动这个Activity，singleTask允许别的Activity与其在一个task中共存，也就是说，如果我在这个singleTask的实例中再打开新的Activity，这个新的Activity还是会在singleTask的实例的task中。

例如：

若我的应用程序中有三个Activity,C1,C2,C3，三个Activity可互相启动，其中C2为singleTask模式，那么，无论我在这个程序中如何点击启动，如：C1->C2->C3->C2->C3->C1-C2，C1,C3可能存在多个实例，但是C2只会存在一个，并且这三个Activity都在同一个task里面。

但是C1->C2->C3->C2->C3->C1-C2，这样的操作过程实际应该是如下这样的，因为singleTask会把task中在其之上的其它Activity destory掉。

操作：C1->C2 C1->C2->C3 C1->C2->C3->C2 C1->C2->C3->C2->C3->C1 C1->C2->C3->C2->C3->C1-C2

实际：C1->C2 C1->C2->C3 C1->C2 C1->C2->C3->C1 C1->C2

若是别的应用程序打开C2，则会新启一个task。

如别的应用Other中有一个activity，taskId为200，从它打开C2，则C2的taskIdI不会为200，例如C2的taskId为201，那么再从C2打开C1、C3，则C2、C3的taskId仍为201。

注意：如果此时你点击home，然后再打开Other，发现这时显示的肯定会是Other应用中的内容，而不会是我们应用中的C1 C2 C3中的其中一个。

1. **SingleInstance**

只有一个实例，并且这个实例独立运行在一个task中，这个task只有这个实例，不允许有别的Activity存在。

例如：

程序有三个ActivityD1,D2,D3，三个Activity可互相启动，其中D2为singleInstance模式。那么程序从D1开始运行，假设D1的taskId为200，那么从D1启动D2时，D2会新启动一个task，即D2与D1不在一个task中运行。假设D2的taskId为201，再从D2启动D3时，D3的taskId为200，也就是说它被压到了D1启动的任务栈中。

若是在别的应用程序打开D2，假设Other的taskId为200，打开D2，D2会新建一个task运行，假设它的taskId为201，那么如果这时再从D2启动D1或者D3，则又会再创建一个task，因此，若操作步骤为other->D2->D1，这过程就涉及到了3个task了。

## 3.2Service

### 3.2.1Service介绍

 Service(服务)是一个一种可以在后台执行长时间运行操作而没有用户界面的应用组件。服务可由其他应用组件启动（如Activity），服务一旦被启动将在后台一直运行，即使启动服务的组件（Activity）已销毁也不受影响。 此外，组件可以绑定到服务，以与之进行交互，甚至是执行进程间通信 (IPC)。 例如，服务可以处理网络事务、播放音乐，执行文件 I/O 或与内容提供程序交互，而所有这一切均可在后台进行，Service基本上分为两种形式：

* **启动状态**

  当应用组件（如 Activity）通过调用 startService() 启动服务时，服务即处于“启动”状态。一旦启动，服务即可在后台无限期运行，即使启动服务的组件已被销毁也不受影响，除非手动调用才能停止服务， 已启动的服务通常是执行单一操作，而且不会将结果返回给调用方。

* **绑定状态**

  当应用组件通过调用 bindService() 绑定到服务时，服务即处于“绑定”状态。绑定服务提供了一个客户端-服务器接口，允许组件与服务进行交互、发送请求、获取结果，甚至是利用进程间通信 (IPC) 跨进程执行这些操作。 仅当与另一个应用组件绑定时，绑定服务才会运行。 多个组件可以同时绑定到该服务，但全部取消绑定后，该服务即会被销毁

### 3.2.2Service启动



SimpleService继承了Service类，并重写了onBind方法，该方法是必须重写的，但是由于此时是启动状态的服务，则该方法无须实现，返回null即可，只有在绑定状态的情况下才需要实现该方法并返回一个IBinder的实现类（这个后面会详细说），接着重写了onCreate、onStartCommand、onDestroy三个主要的生命周期方法，关于这几个方法说明如下:

* **onBind()**

  当另一个组件想通过调用 bindService() 与服务绑定（例如执行 RPC）时，系统将调用此方法。在此方法的实现中，必须返回 一个IBinder 接口的实现类，供客户端用来与服务进行通信。无论是启动状态还是绑定状态，此方法必须重写，但在启动状态的情况下直接返回 null。

* **onCreate()**

  首次创建服务时，系统将调用此方法来执行一次性设置程序（在调用 onStartCommand() 或onBind() 之前）。如果服务已在运行，则不会调用此方法，该方法只调用一次

* **onStartCommand()**

  当另一个组件（如 Activity）通过调用 startService() 请求启动服务时，系统将调用此方法。一旦执行此方法，服务即会启动并可在后台无限期运行。 如果自己实现此方法，则需要在服务工作完成后，通过调用 stopSelf() 或 stopService() 来停止服务。（在绑定状态下，无需实现此方法。）

* **onDestroy()**

  当服务不再使用且将被销毁时，系统将调用此方法。服务应该实现此方法来清理所有资源，如线程、注册的侦听器、接收器等，这是服务接收的最后一个调用。

第一次调用startService方法时，onCreate方法、onStartCommand方法将依次被调用，而多次调用startService时，只有onStartCommand方法被调用，最后我们调用stopService方法停止服务时onDestory方法被回调，这就是启动状态下Service的执行周期。在服务的外部，必须使用stopService()方法停止，在服务的内部可以调用stopSelf()方法停止当前服务。至于onBind方法，只有在绑定服务时才会起作用。

context.startService() ->onCreate()- >onStartCommand()->Service running--

调用context.stopService() ->onDestroy()

context.bindService()->onCreate()->onBind()->Service running--调用>onUnbind() -> onDestroy()

### 3.2.3启动Service和绑定Service转换

先绑定服务后启动服务

  如果当前Service实例先以绑定状态运行，然后再以启动状态运行，那么绑定服务将会转为启动服务运行，这时如果之前绑定的宿主（Activity）被销毁了，也不会影响服务的运行，服务还是会一直运行下去，指定收到调用停止服务或者内存不足时才会销毁该服务。

先启动服务后绑定服务

  如果当前Service实例先以启动状态运行，然后再以绑定状态运行，当前启动服务并不会转为绑定服务，但是还是会与宿主绑定，只是即使宿主解除绑定后，服务依然按启动服务的生命周期在后台运行，直到有Context调用了stopService()或是服务本身调用了stopSelf()方法抑或内存不足时才会销毁服务。

  以上两种情况显示出启动服务的优先级确实比绑定服务高一些

### 3.2.4Service与activity通信

1. 简单通信

通过startService(intent)与stopService(intent)进行通讯，在Service中通过onStartCommand(final Intent intent,int flags,int startId)接受activity传递的参数

1. 通过Binder机制

MainActivity实现ServiceConnection类，实现这个类之后，我们还需要重写ServiceConnection类中的两个方法onServiceConnected和onServiceDisconnected，这两个方法分别是在绑定成功和服务所在进程崩溃的时候被调用，如果绑定成功了，那么onServiceConnected(ComponentName componentName, IBinder iBinder) 就会被执行，然后第二个参数IBinder正是MyService中onBind（）方法的返回值，因此我们可以通过这个返回值来想MyService传递数据。

public class MsgService extends Service {

/\*\*

\* 进度条的最大值

\*/

public static final int MAX\_PROGRESS = 100;

/\*\*

\* 进度条的进度值

\*/

private int progress = 0;

/\*\*

\* 增加get()方法，供Activity调用

\* @return 下载进度

\*/

public int getProgress() {

return progress;

}

/\*\*

\* 模拟下载任务，每秒钟更新一次

\*/

public void startDownLoad(){

new Thread(new Runnable() {

@Override

public void run() {

while(progress < MAX\_PROGRESS){

progress += 5;

try {

Thread.sleep(1000);

} catch (InterruptedException e) {

e.printStackTrace();

}

}

}

}).start();

}

/\*\*

\* 返回一个Binder对象

\*/

@Override

public IBinder onBind(Intent intent) {

return new MsgBinder();

}

public class MsgBinder extends Binder{

/\*\*

\* 获取当前Service的实例

\* @return

\*/

public MsgService getService(){

return MsgService.this;

}

}

}

//Activity类

public class MainActivity extends Activity {

private MsgService msgService;

private int progress = 0;

private ProgressBar mProgressBar;

@Override

protected void onCreate(Bundle savedInstanceState) {

super.onCreate(savedInstanceState);

setContentView(R.layout.activity\_main);

//绑定Service

Intent intent = new Intent("com.example.communication.MSG\_ACTION");

bindService(intent, conn, Context.BIND\_AUTO\_CREATE);

mProgressBar = (ProgressBar) findViewById(R.id.progressBar1);

Button mButton = (Button) findViewById(R.id.button1);

mButton.setOnClickListener(new OnClickListener() {

@Override

public void onClick(View v) {

//开始下载

msgService.startDownLoad();

//监听进度

listenProgress();

}

});

}

/\*\*

\* 监听进度，每秒钟获取调用MsgService的getProgress()方法来获取进度，更新UI

\*/

public void listenProgress(){

new Thread(new Runnable() {

@Override

public void run() {

while(progress < MsgService.MAX\_PROGRESS){

progress = msgService.getProgress();

mProgressBar.setProgress(progress);

try {

Thread.sleep(1000);

} catch (InterruptedException e) {

e.printStackTrace();

}

}

}

}).start();

}

ServiceConnection conn = new ServiceConnection() {

@Override

public void onServiceDisconnected(ComponentName name) {

}

@Override

public void onServiceConnected(ComponentName name, IBinder service) {

//返回一个MsgService对象

msgService = ((MsgService.MsgBinder)service).getService();

}

};

@Override

protected void onDestroy() {

unbindService(conn);

super.onDestroy();

}

}

缺点：每次数据都是由activity去获取，并不是Service通知过来，可以使用回调接口去完成，由Service 去触发

1. 使用广播

public class MsgService extends Service {

/\*\*

\* 进度条的最大值

\*/

public static final int MAX\_PROGRESS = 100;

/\*\*

\* 进度条的进度值

\*/

private int progress = 0;

private Intent intent = new Intent("com.example.communication.RECEIVER");

/\*\*

\* 模拟下载任务，每秒钟更新一次

\*/

public void startDownLoad(){

new Thread(new Runnable() {

@Override

public void run() {

while(progress < MAX\_PROGRESS){

progress += 5;

//发送Action为com.example.communication.RECEIVER的广播

intent.putExtra("progress", progress);

sendBroadcast(intent);

try {

Thread.sleep(1000);

} catch (InterruptedException e) {

e.printStackTrace();

}

}

}

}).start();

}

@Override

public int onStartCommand(Intent intent, int flags, int startId) {

startDownLoad();

return super.onStartCommand(intent, flags, startId);

}

@Override

public IBinder onBind(Intent intent) {

return null;

}

}

//Activity

public class MainActivity extends Activity {

private ProgressBar mProgressBar;

private Intent mIntent;

private MsgReceiver msgReceiver;

@Override

protected void onCreate(Bundle savedInstanceState) {

super.onCreate(savedInstanceState);

setContentView(R.layout.activity\_main);

//动态注册广播接收器

msgReceiver = new MsgReceiver();

IntentFilter intentFilter = new IntentFilter();

intentFilter.addAction("com.example.communication.RECEIVER");

registerReceiver(msgReceiver, intentFilter);

mProgressBar = (ProgressBar) findViewById(R.id.progressBar1);

Button mButton = (Button) findViewById(R.id.button1);

mButton.setOnClickListener(new OnClickListener() {

@Override

public void onClick(View v) {

//启动服务

mIntent = new Intent("com.example.communication.MSG\_ACTION");

startService(mIntent);

}

});

}

@Override

protected void onDestroy() {

//停止服务

stopService(mIntent);

//注销广播

unregisterReceiver(msgReceiver);

super.onDestroy();

}

/\*\*

\* 广播接收器

\* @author len

\*

\*/

public class MsgReceiver extends BroadcastReceiver{

@Override

public void onReceive(Context context, Intent intent) {

//拿到进度，更新UI

int progress = intent.getIntExtra("progress", 0);

mProgressBar.setProgress(progress);

}

}

}

1. 总结

Activity调用bindService (Intent service, ServiceConnection conn, int flags)方法，得到Service对象的一个引用，这样Activity可以直接调用到Service中的方法，如果要主动通知Activity，我们可以利用回调方法

 Service向Activity发送消息，可以使用广播，当然Activity要注册相应的接收器。比如Service要向多个Activity发送同样的消息的话，用这种方法就更好

### 3.2.5Service保活

## 3.3BroadcastReceiver

### 3.3.1实现原理

1. 广播接受者通过Binder机制在AMS(Activity Manager Service)中注册
2. 广播发送者通过Binder机制向AMS发送广播
3. AMS根据广播发送者要求，在已注册列表中，寻找合适的广播接受者，依据IntentFilter/Permission
4. AMS将广播发送到合适的广播接受者对应色消息循环队列中
5. 广播接受者通过消息循环拿到广播，并回调onReceive();

### 3.3.2注册方式

1. 静态注册

在AndroidManifest.xml中注册，当应用程序关闭后，有广播传来也会调用

1. 动态注册

通过context()的registerReceiver()方法进行动态注册，伴随着activity的生命周期，当activity销毁的时候不接受信息，最好在onResume()中注册，在onPause()中注销

### 3.3.3广播分类

1. 普通广播

开发者自身自身定义intent的广播

Intent intent = new Intent();

intent.setAction(BROADCAST\_ACTION);

sendBroadcast(intent);

1. 系统广播

系统自身定义的广播，监听手机基本操作，如开机、网络变化等，会发出相应广播

1. 有序广播

发送的广播被接受者按照先后顺序接受，有序是针对广播接受者，通过设置Priority属性设置

1. 从大到小
2. Priority属性相同时，动态注册的广播优先

特点：

1. 接受广播按照顺序接受
2. 先接收的广播接收者对广播进行拦截，后接收的广播接收者不再接收此广播
3. 先接收的广播接收者可以对广播进行修改，后面接收的将是修改后的

发送方式：

sendOrderedBroadcast(intent);

1. App内部广播（Local Broadcast）

出现原因：

1. 其他App可能发出和本App需要类型相同的广播，导致App不断接收广播并处理
2. 其他App接收到本App发送的广播

解决方案：

1. 将全局广播设置成局部广播
2. 注册广播时将exported设置为false
3. 增加权限设置
4. 使用intent.setPackage去指定接收者的包名

2.使用LocalBroadcastManager类

## 3.4ContentProvider

## 3.5Fragment

### 3.5.1生命周期



### 3.5.2与activity生命周期结合



onCreate过程

01-22 15:30:28.091: E/HJJ(10315): Activity &&&& onCreate...

01-22 15:30:28.091: E/HJJ(10315): ArrayListFragment \*\*\*\* onAttach...

01-22 15:30:28.091: E/HJJ(10315): ArrayListFragment \*\*\*\* onCreate...

01-22 15:30:28.115: E/HJJ(10315): ArrayListFragment \*\*\*\* onCreateView...

01-22 15:30:28.123: E/HJJ(10315): ArrayListFragment \*\*\*\* onActivityCreated...

onStart过程

01-22 15:30:28.123: E/HJJ(10315): Activity &&&& onStart...

01-22 15:30:28.123: E/HJJ(10315): ArrayListFragment \*\*\*\* onStart...

onResume过程

01-22 15:30:28.123: E/HJJ(10315): Activity &&&& onResume...

01-22 15:30:28.123: E/HJJ(10315): ArrayListFragment \*\*\*\* onResume...

onPause过程

01-22 15:31:26.748: E/HJJ(10315): ArrayListFragment \*\*\*\* onPause...

01-22 15:31:26.748: E/HJJ(10315): Activity &&&& onPause...

onStop过程

01-22 15:31:27.638: E/HJJ(10315): ArrayListFragment \*\*\*\* onStop...

01-22 15:31:27.638: E/HJJ(10315): Activity &&&& onStop...

onStart过程

01-22 15:31:57.537: E/HJJ(10315): Activity &&&& onStart...

01-22 15:31:57.537: E/HJJ(10315): ArrayListFragment \*\*\*\* onStart...

onResume过程

01-22 15:31:57.537: E/HJJ(10315): Activity &&&& onResume...

01-22 15:31:57.537: E/HJJ(10315): ArrayListFragment \*\*\*\* onResume...

onPause过程

01-22 15:32:47.412: E/HJJ(10315): ArrayListFragment \*\*\*\* onPause...

01-22 15:32:47.412: E/HJJ(10315): Activity &&&& onPause...

onStop过程

01-22 15:32:47.865: E/HJJ(10315): ArrayListFragment \*\*\*\* onStop...

01-22 15:32:47.865: E/HJJ(10315): Activity &&&& onStop...

onDestroy过程

01-22 15:32:47.865: E/HJJ(10315): ArrayListFragment \*\*\*\* onDestroyView...

01-22 15:32:47.865: E/HJJ(10315): ArrayListFragment \*\*\*\* onDestroy...

01-22 15:32:47.865: E/HJJ(10315): ArrayListFragment \*\*\*\* onDetach...

01-22 15:32:47.865: E/HJJ(10315): Activity &&&& onDestroy...

## 3.6Handler

### 3.6.1相关概念

**Message**

定义：消息，理解为线程间通讯的数据单元（Handler接受和处理的消息对象。）

Message Queue

消息队列，用于存放Handler发送的消息，先进先出

Handler

是Message的主要处理者，负责将Message添加到队列或者处理Looper分派过来的Message

Looper

扮演Message Queue和Handler之间桥梁的角色，循环取出Message Queue的Message,将取出的Message交给对应的Handler

### 3.6.2方法

**Handler**

* 提供sendMessage方法，将消息放置到队列中
* 提供handleMessage方法，定义个各种消息的处理方式；

**Looper**

* Looper.prepare()：实例化Looper对象；为当前线程生成一个消息队列；
* Looper.loop() ：循环从消息队列中获取消息，交给Handler处理；此时线程处于无限循环中，不停的从MessageQueue中获取Message 消息 ；如果没有消息就阻塞

**MessageQueue**

* 提供enqueueMessage 方法，将消息根据时间放置到队列中；
* 提供next方法，从队列中获取消息，没有消息的时候阻塞；

**Handler工作流程解释**  
异步通信传递机制步骤主要包括异步通信的准备、消息发送、消息循环和消息处理

1. 异步通信的准备  
   包括Looper对象的创建&实例化、MessageQueue队列的创建和Handler的实例化
2. 消息发送  
   Handler将消息发送到消息队列中
3. 消息循环  
   Looper执行Looper.loop()进入消息循环，在这个循环过程中，不断从该Message Queue取出消息，并将取出的消息派发给创建该消息的Handler
4. 消息处理  
   调用该Handler的dispatchMessage(msg)方法，即回调handleMessage(msg)处理消息

### 3.6.3Looper

Looper主要是prepare()和loop()两个方法.

在prepare()判断已经创建MessageQueue，未创建则调用Looper构造方法创建一个MessageQueue。

在loop()中，进行无限循环去取消息，若无消息则阻塞，若取到消息，则msg.target.dispatchMessage(msg);把消息交给msg的target的dispatchMessage方法去处理。

### 3.6.4Handler

Handler主要完成：

1. 在主线程中发消息给MessageQueue

使用Handler之前，会初始化一个Handler实例，在创建实例时，会通过Loop.myLooper()获取当前线程的Looper对象，不存在则抛出异常。

Handler想MessageQueue发送消息可以发送post和send两种，相比send方法，post最大不同在于更新操作可以直接在重写run方法定义，在发送消息最终都会调用queue.enqueueMessage()将消息放入对流

1. 处理Looper派发来的消息：dispatchMessage();

调用send是的handlerMessage回调方法或者post时的msg回调

### 3.6.5MessageQueue

MessageQueue包含enqueueMessage入队操作和next()出队操作

### 3.6.6其他

* 一个Thread（线程）只能有一个Looper，可以有多个Handler
* 一个Looper可以绑定多个Handler；
* 一个Handler只能绑定一个Looper；

### 3.6.7ThreadLocal

ThreadLocal是线程内部的数据存储类，通过他可以在指定的线程中存储数据，数据存储后，只有指定线程才可以获取到存储的数据，其他线程无法获取到数据，在Looper中使用到ThreadLocal来存储Looper保证获取对应的Looper，Looper在prepare方法以及中set进去，在执行loop方法时取出对应的参数

在ThreadLocal中提供set和get方法来存储这一线程对应的数据ThreadLocal的Values对象

## 3.7Android性能优化

### 3.7.1UI

#### 3.7.1.1UI卡顿

App帧率应该保持在60帧/秒,记我们需要在16ms中完成CPU和GPU的计算、绘制和渲染工作，否则会出现丢帧卡顿问题

虚拟机在执行GC垃圾回收操作时所有线程（包括UI线程）都需要暂停，当GC垃圾回收完成之后所有线程才能够继续执行（这个细节下面小节会有详细介绍）。也就是说当在16ms内进行渲染等操作时如果刚好遇上大量GC操作则会导致渲染时间明显不足，也就从而导致了丢帧卡顿问题。

出现的原因：

1. Layout过于复杂，在16ms中无法完成渲染
2. 同一时间执行动画次数过多，导致CPU或者GPU负载过重
3. View过度绘制，导致某一像素被绘制多次
4. 频繁触发measure、layout,导致耗时过多
5. 内存频繁触发GC,导致阻塞渲染操作

#### 3.7.1.2解决

1. 使用HierarchyViewer分析UI性能
2. 使用GPU过度绘制分析UI性能
3. 使用GPU呈现模式图及ＦＰＳ考核ＵＩ性能
4. 使用Lint进行资源和冗余UI布局优化

### 3.7.2 app启动优化

### 3.7.3资源优化

### 3.7.4网络请求优化

### 3.7.5电池优化

## 3.8View绘制流程

### 3.8.1简介

View的绘制是从ViewRoot的performTraversals开始，经过measure、layout、draw绘制UI,measure用来测量View的宽高，layout用来确定View在父容器中的位置，draw用来进行UI绘制

### 3.8.2measure()

 主要作用：为整个View树计算实际的大小，即设置实际的高(对应属性:mMeasuredHeight)和宽(对应属性:mMeasureWidth)，每个View的控件的实际宽高都是由父视图和本身视图决定的。

  具体的调用链如下：

          ViewRoot根对象地属性mView(其类型一般为ViewGroup类型)调用measure()方法去计算View树的大小，回调

View/ViewGroup对象的onMeasure()方法，该方法实现的功能如下：

         1、设置本View视图的最终大小，该功能的实现通过调用setMeasuredDimension()方法去设置实际的高(对应属性： mMeasuredHeight)和宽(对应属性：mMeasureWidth) ；

         2 、如果该View对象是个ViewGroup类型，需要重写该onMeasure()方法，对其子视图进行遍历的measure()过程。

               2.1  对每个子视图的measure()过程，是通过调用父类ViewGroup.java类里的measureChildWithMargins()方法去实现，该方法内部只是简单地调用了View对象的measure()方法。(由于measureChildWithMargins()方法只是一个过渡层更简单的做法是直接调用View对象的measure()方法)。

### 3.8.3layout()

 主要作用 ：为将整个根据子视图的大小以及布局参数将View树放到合适的位置上。

     具体的调用链如下：

       host.layout()开始View树的布局，继而回调给View/ViewGroup类中的layout()方法。具体流程如下

        1 、layout方法会设置该View视图位于父视图的坐标轴，即mLeft，mTop，mLeft，mBottom(调用setFrame()函数去实现)

  接下来回调onLayout()方法(如果该View是ViewGroup对象，需要实现该方法，对每个子视图进行布局) ；

       2、如果该View是个ViewGroup类型，需要遍历每个子视图chiildView，调用该子视图的layout()方法去设置它的坐标值。

### 3.8.4draw

由ViewRoot对象的performTraversals()方法调用draw()方法发起绘制该View树，值得注意的是每次发起绘图时，并不会重新绘制每个View树的视图，而只会重新绘制那些“需要重绘”的视图，View类内部变量包含了一个标志位DRAWN，当该视图需要重绘时，就会为该View添加该标志位。

### 3.8.5invalidate()

请求重绘View树，即draw()过程，假如视图发生大小没有变化就不会调用layout()过程，并且只绘制那些“需要重绘的”视图，即谁(View的话，只绘制该View ；ViewGroup，则绘制整个ViewGroup)请求invalidate()方法，就绘制该视图,发生在UI线程

### 3.8.6postInvalidate()

同invalidate()，发生在子线程中

### 3.8.7requestLayout()

只是对View树重新布局layout过程包括measure()和layout()过程，不会调用draw()过程，但不会重新绘制任何视图包括该调用者本身。

## 3.9Android事件分发机制

### 3.9.1MotionEvent事件分发

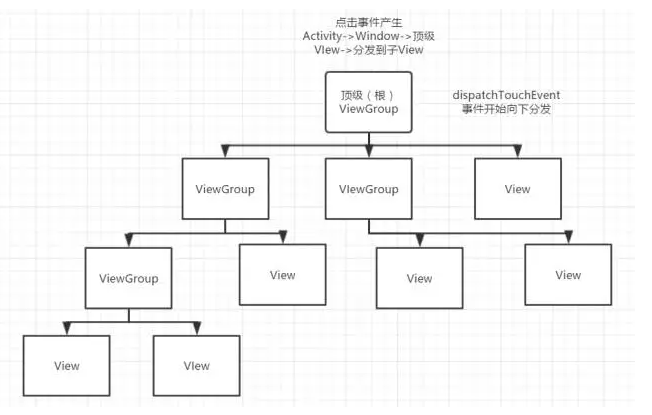
事件分发需要View的三个重要方法来共同完成：

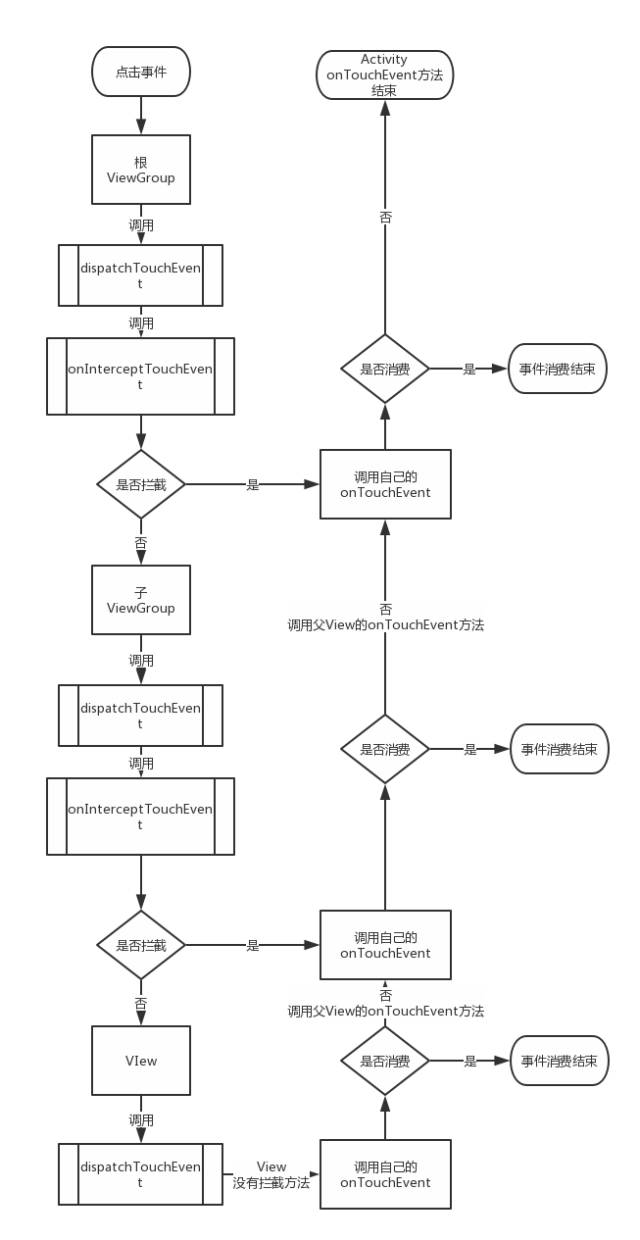
**public boolean dispatchTouchEvent(MotionEvent event)**  
 通过方法名我们不难猜测，它就是事件分发的重要方法。那么很明显，如果一个MotionEvent传递给了View，那么dispatchTouchEvent方法一定会被调用！  
 **返回值：表示是否消费了当前事件。可能是View本身的onTouchEvent方法消费，也可能是子View的dispatchTouchEvent方法中消费。返回true表示事件被消费，本次的事件终止。返回false表示View以及子View均没有消费事件，将调用父View的onTouchEvent方法**

**public boolean onInterceptTouchEvent(MotionEvent ev)**  
 事件拦截，当一个ViewGroup在接到MotionEvent事件序列时候，首先会调用此方法判断是否需要拦截。**特别注意，这是ViewGroup特有的方法，View并没有拦截方法  
 返回值：是否拦截事件传递，返回true表示拦截了事件，那么事件将不再向下分发而是调用View本身的onTouchEvent方法。返回false表示不做拦截，事件将向下分发到子View的dispatchTouchEvent方法。**

**public boolean onTouchEvent(MotionEvent ev)**  
 真正对MotionEvent进行处理或者说消费的方法。在dispatchTouchEvent进行调用。  
**返回值：返回true表示事件被消费，本次的事件终止。返回false表示事件没有被消费，将调用父View的onTouchEvent方法**







## 3.10自定义UI

### 3.10.1实现步骤

1. 继承View类或其子类

　 2. 复写view中的一些函数

　３.为自定义View类增加属性（两种方式）

　４.绘制控件（导入布局）

　５.响应用户事件

　６.定义回调函数（根据自己需求来选择）

### 3.10.2方法

* onDraw()

　　view中onDraw()是个空函数，也就是说具体的视图都要覆写该函数来实现自己的绘制。对于ViewGroup则不需要实现该函数，因为作为容器是“没有内容“的（但必须实现dispatchDraw()函数，告诉子view绘制自己）。

* onLayout()

　　主要是为viewGroup类型布局子视图用的，在View中这个函数为空函数。

* onMeasure()

　　用于计算视图大小（即长和宽）的方式，并通过setMeasuredDimension(width, height)保存计算结果。

* onTouchEvent

　　定义触屏事件来响应用户操作。

## 3.11多线程

### 3.11.1继承Thread类

### 3.11.2实现Runnable接口

### 3.11.3AsyncTask

#### 3.11.3.1介绍

AsyncTask封装了线程池和Handler,通过AsyncTask可以方便地执行后台任务以及在主线程中访问UI,但是AsyncTask不适合执行特别耗时的后台任务

#### 3.11.3.2方法

1. onPreExcute（），在主线程中执行，在异步任务执行之前，此方法会被调用，一般用来做一些准备工作。
2. doInBackground（Params...Params）,在线程池中执行，此方法用于执行异步任务。可以通过publicProgress方法来更新任务进度，publicProgress方法会调用onProgressUpdate方法，需要返回结果给onPostExecute（）。
3. onProgressUpdate，在主线程中执行，当后台任务执行进度发生变化会调用此方法
4. onPostExecute（Result result）在主线程中执行，异步任务执行之后，会调用此方法，其中result参数是后台任务的返回值，doInBackground的返回值。

#### 3.11.3.3实现原理

### 3.11.4Handler

### 3.11.5HandlerThread

#### 3.11.5.1简介

HandlerThread继承自Thread，它的run（）方法中通过Looper.prepare（）创建消息队列，并通过Looper.loop（）开启了消息循环，这样就可以在HandlerThread中创建Handler来通知HandlerThread执行一个具体操作

#### 3.11.5.2原理

### 3.11.6IntentService

## 3.12动画

### 3.12.1动画分类

动画可以分为三类：View动画、帧动画和属性动画

### 3.12.2View动画

View动画也叫Tween（补间）动画，可以在一个视图容器中执行一些列简单变换（位置、大小、旋转、透明度）

#### 3.12.2.1TranslateAnimation

android:fromXDelta 起始点X轴坐标（数值、百分数、百分数p，譬如50表示以当前View左上角坐标加50px为初始点、50%表示以当前View的左上角加上当前View宽高的50%做为初始点、50%p表示以当前View的左上角加上父控件宽高的50%做为初始点）

android:fromYDelta 起始点Y轴坐标

android:toXDelta 结束点X轴坐标

android:toYDelta 结束点Y轴坐标

#### 3.12.2.2ScaleAnimation

android:fromXScale 初始X轴缩放比例，1.0表示无变化

android:toXScale 结束X轴缩放比例

android:fromYScale 初始Y轴缩放比例

android:toYScale 结束Y轴缩放比例

android:pivotX 缩放起点X轴坐标（数值、百分数、百分数p，譬如50表示以当前View左上角坐标加50px为初始点、50%表示以当前View的左上角加上当前View宽高的50%做为初始点、50%p表示以当前View的左上角加上父控件宽高的50%做为初始点）

android:pivotY 缩放起点Y轴坐标

#### 3.12.2.3RotateAnimation

android:fromDegrees 旋转开始角度，正代表顺时针度数，负代表逆时针度数

android:toDegrees 旋转结束角度，正代表顺时针度数，负代表逆时针度数

android:pivotX 缩放起点X坐标（数值、百分数、百分数p，譬如50表示以当前View左上角坐标加50px为初始点、50%表示以当前View的左上角加上当前View宽高的50%做为初始点、50%p表示以当前View的左上角加上父控件宽高的50%做为初始点）

android:pivotY 缩放起点Y坐标

#### 3.12.2.4AlphaAnimation

android:fromAlpha 动画开始的透明度（0.0到1.0，0.0是全透明，1.0是不透明）

android:toAlpha 动画结束的透明度

#### 3.12.2.5Animation

android:detachWallpaper 是否在壁纸上运行

android:duration 动画持续时间，毫秒为单位

android:fillAfter 控件动画结束时是否保持动画最后的状态

android:fillBefore 控件动画结束时是否还原到开始动画前的状态

android:fillEnabled 与android:fillBefore效果相同

android:interpolator 设定插值器（指定的动画效果，譬如回弹等）

android:repeatCount 重复次数

android:repeatMode 重复类型有两个值，reverse表示倒序回放，restart表示从头播放

android:startOffset 调用start函数之后等待开始运行的时间，单位为毫秒

android:zAdjustment 表示被设置动画的内容运行时在Z轴上的位置（top/bottom/normal），默认为normal

### 2.12.3帧动画

帧动画，允许实现幻灯片的效果，动画的实质是Drawable,一般放在res/drawable/目录下，使用AnimationDrawable进行动画启动。

### 3.12.4属性动画

#### 3.12.4.1ObjectAnimator

#### 3.12.4.2ValueAnimator

#### 3.12.4.3AnimatorSet

## 3.13IPC通信方式

### Bundle

简单易用  但是只能传输Bundle支持的对象 常用于四大组件间进程间通信

### 文件共享

简单易用  但不适合在高并发的情况下 并且读取文件需要时间 不能即时通信   常用于并发程度不高 并且实时性要求不高的情况

### AIDL

功能强大 支持一对多并发通信 支持即时通信   但是使用起来比其他的复杂 需要处理好多线程的同步问题  常用于一对多通信 且有RPC 需求的场合(服务端和客户端通信)

### Messenger

功能一般 支持一对多串行通信 支持实时通信  但是不能很好处理高并发情况 只能传输Bundle支持的类型  常用于低并发的无RPC需求一对多的场合

### ContentProvider

在数据源访问方面功能强大 支持一对多并发操作 可扩展call方法  可以理解为约束版的AIDL  提供CRUD操作和自定义函数  常用于一对多的数据共享场合

### Socket

功能强大 可以通过网络传输字节流 支持一对多并发操作  但是实现起来比较麻烦 不支持直接的RPC   常用于网络数据交换

## 3.15Android新特性

### 3.15.1 Android5.0新特性

* MaterialDesign设计风格
* 支持多种设备
* 支持64位ART虚拟机

### 3.15.2 Android6.0新特性

1. 运行时权限

将App权限拆分为两种，安装时权限和运行时权限

1. 电池管理

### 3.15.3 Android7.0新特性

1. 应用件共享文件，严格模式引入
2. 分屏
3. Java8
4. 夜间模式

### 3.15.4 Android8.0新特性

## 3.16Binder机制

### 3.16.1介绍

Binder通信采用C/S架构，从组件视角来说，包含Client、Server、ServiceManager以及binder驱动，其中ServiceManager用于管理系统中的各种服务。架构图如下所示：

Client/Server/ServiceManage之间的相互通信都是基于Binder机制。既然基于Binder机制通信，那么同样也是C/S架构，则图中的3大步骤都有相应的Client端与Server端。

1. **[注册服务(addService)](http://gityuan.com/2015/11/14/binder-add-service/)**：Server进程要先注册Service到ServiceManager。该过程：Server是客户端，ServiceManager是服务端。
2. **[获取服务(getService)](http://gityuan.com/2015/11/15/binder-get-service/)**：Client进程使用某个Service前，须先向ServiceManager中获取相应的Service。该过程：Client是客户端，ServiceManager是服务端。
3. ****使用服务****：Client根据得到的Service信息建立与Service所在的Server进程通信的通路，然后就可以直接与Service交互。该过程：client是客户端，server是服务端。

### 3.16.2为什么使用

1. Binder数据拷贝只需要一次，而管道、消息队列、Socket都需要2次，但共享内存方式一次内存拷贝都不需要；从性能角度看，Binder性能仅次于共享内存
2. Binder是基于C/S架构的，简单解释下C/S架构，是指客户端(Client)和服务端(Server)组成的架构，Client端有什么需求，直接发送给Server端去完成，架构清晰明朗，Server端与Client端相对独立，稳定性较好；
3. 传统Linux IPC的接收方无法获得对方进程可靠的UID/PID，从而无法鉴别对方身份；每个AndroidApp都有一个唯一的UId,Binder可以判断UID来进行权限判断和管理

## 3.17GC

### 3.17.1JVM

Java 程序运行时的内存分配策略有三种,分别是**静态分配**,**栈式分配**,和**堆式分配**，对应的，三种存储策略使用的内存空间主要分别是**静态存储区**（也称方法区）、**栈区**和**堆区**。   
 **静态存储区（方法区）**：主要存放静态数据、全局 static 数据和常量。这块内存在程序编译时就已经分配好，并且在程序整个运行期间都存在。   
 **栈区** ：当方法被执行时，方法体内的局部变量都在栈上创建，并在方法执行结束时这些局部变量所持有的内存将会自动被释放。因为栈内存分配运算内置于处理器的指令集中，效率很高，但是分配的内存容量有限。   
 **堆区** ： 又称动态内存分配，通常就是指在程序运行时直接 new 出来的内存。这部分内存在不使用时将会由 Java 垃圾回收器来负责回收。   
**栈与堆的区别：**   
 在方法体内定义的（局部变量）一些基本类型的变量和对象的引用变量都是在方法的栈内存中分配的。当在一段方法块中定义一个变量时，Java 就会在栈中为该变量分配内存空间，当超过该变量的作用域后，该变量也就无效了，分配给它的内存空间也将被释放掉，该内存空间可以被重新使用。   
 堆内存用来存放所有由 new 创建的对象（包括该对象其中的所有成员变量）和数组。在堆中分配的内存，将由 Java 垃圾回收器来自动管理。在堆中产生了一个数组或者对象后，还可以在栈中定义一个特殊的变量，这个变量的取值等于数组或者对象在堆内存中的首地址，这个特殊的变量就是我们上面说的引用变量。我们可以通过这个引用变量来访问堆中的对象或者数组。

### 3.17.2垃圾回收机制

在JVM堆中有一颗root树，当不在此树上的对象就是需要被回收的

## 3.18Android安全

1. 混淆
2. Jni
3. 加固、去壳

## 3.19内存分析

## 3.20apk瘦身

1. 删除无用资源，使用lint删除无用资源
2. ProGuard混淆
3. 移除无用库
4. 图片压缩
5. Gradle开启shrinkResources

## 3.21内存泄漏

### 3.21.1Handler内存泄漏解决

#### 3.21.1.1原因

当使用内部类（包括匿名类）来创建Handler的时候，Handler对象会隐式地持有一个外部类对象（通常是一个Activity）的引用（不然你怎么可能通过Handler来操作Activity中的View？）。而Handler通常会伴随着一个耗时的后台线程（例如从网络拉取图片）一起出现，这个后台线程在任务执行完毕（例如图片下载完毕）之后，通过消息机制通知Handler，然后Handler把图片更新到界面。然而，如果用户在网络请求过程中关闭了Activity，正常情况下，Activity不再被使用，它就有可能在GC检查时被回收掉，但由于这时线程尚未执行完，而该线程持有Handler的引用（不然它怎么发消息给Handler？），这个Handler又持有Activity的引用，就导致该Activity无法被回收（即内存泄露），直到网络请求结束（例如图片下载完毕）。另外，如果你执行了Handler的postDelayed()方法，该方法会将你的Handler装入一个Message，并把这条Message推到MessageQueue中，那么在你设定的delay到达之前，会有一条MessageQueue -> Message -> Handler -> Activity的链，导致你的Activity被持有引用而无法被回收。

#### 3.21.1.2解决方案

方法1：

在activity将要被销毁的时候使用Handler的removeCallbacks()方法进行回调，把消息对象从消息队列中移除

方法2：将Handler声明为静态类

在Java 中，非静态的内部类和匿名内部类都会隐式地持有其外部类的引用，静态的内部类不会持有外部类的引用。  
 静态类不持有外部类的对象，所以你的Activity可以随意被回收。由于Handler不再持有外部类对象的引用，导致程序不允许你在Handler中操作Activity中的对象了。所以你需要在Handler中增加一个对Activity的弱引用（WeakReference）。

static class MyHandler extends Handler

{

WeakReference<Activity> mWeakReference;

public MyHandler(Activity activity)

{

mWeakReference=new WeakReference<Activity>(activity);

}

@Override

public void handleMessage(Message msg)

{

final Activity activity=mWeakReference.get();

if(activity!=null)

{

if (msg.what == 1)

{

noteBookAdapter.notifyDataSetChanged();

}

}

}

}

### 3.21.2单例造成的内存泄漏

当单例使用activity对应的Context时，activity被销毁的时候，但是因为单例还持有这个对象的引用，对象就不能正常的回收，会造成内存泄漏。此时需要把单例持有的引用换成Application的context。

### 3.21.3资源未关闭造成的内存泄漏

对于使用了BraodcastReceiver，ContentObserver，File，Cursor，Stream，Bitmap等资源的使用，应该在Activity销毁时及时关闭或者注销，否则这些资源将不会被回收，造成内存泄漏。

## 3.22开源框架

### 3.22.1OkHttp

#### 1.整体流程IMG_256

#### Dispatcher

维护了

\* 运行中的异步请求队列 runningAsyncCalls   
 \* 就绪状态的异步请求队列 readyAsyncCalls   
 \* 运行中的同步请求队列 runningSyncCalls (注意名字的细微差别)   
 \* 线程池 executorService

每当有新的**同步**请求到Dispatcher碗里来，直接加入到同步运行中队列。   
每当有新的**异步**请求到Dispatcher碗里来，那么通过maxRequests(最大请求数)和maxRequestsPerHost(相同host最大请求数)来判定，是应该进到运行中的队列并立即执行呢，还是进到就绪队列中，等待运行队列有空间了，再进到运行队列中。

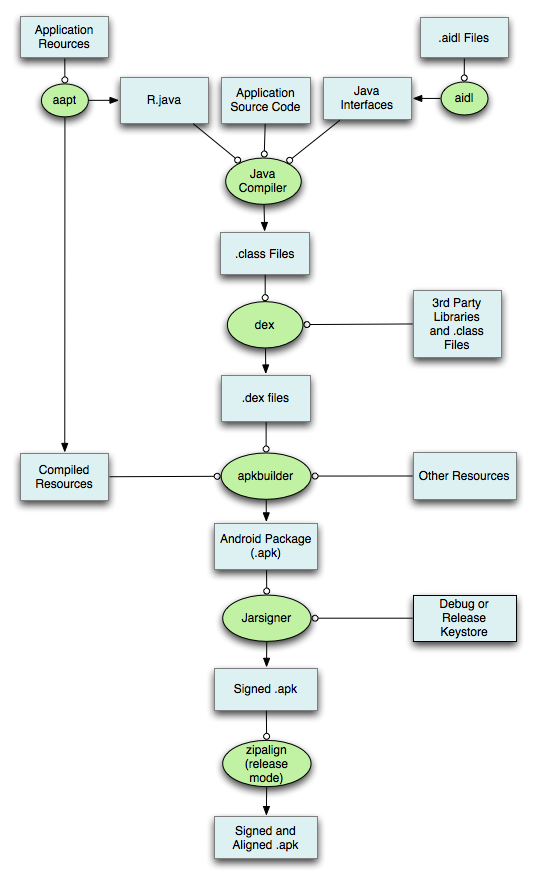
1.初始化一个executorService线程

2.首先调用enqueue()将异步请求存放在runningAsyncCalls并且在线程池中执行此线程 或者readyAsyncCalls 中，也可以通过executed将同步请求放在runningSyncCalls 中

3.当请求执行完成调用finished方法会调用promoteCalls方法来调整请求继续往后执行

#### 3.ConnectionPool

## 3.23Apk构建流程



通常的构建过程就是如上图所示，下面是具体描述：

1.AAPT(Android Asset Packaging Tool)工具会打包应用中的资源文件，如AndroidManifest.xml、layout布局中的xml等，并将xml文件编译为二进制形式，当然assets文件夹中的文件不会被编译，图片及raw文件夹中的资源也会保持原来的形态，需要注意的是raw文件夹中的资源也会生成资源id。AAPT编译完成之后会生成R.java文件。

2.AIDL工具会将所有的aidl接口转化为java接口。

3.所有的java代码，包括R.java与aidl文件都会被Java编译器编译成.class文件。

4.Dex工具会将上述产生的.class文件及第三库及其他.class文件编译成.dex文件（dex文件是Dalvik虚拟机可以执行的格式），dex文件最终会被打包进APK文件。

5.ApkBuilder工具会将编译过的资源及未编译过的资源（如图片等）以及.dex文件打包成APK文件。

6.生成APK文件后，需要对其签名才可安装到设备，平时测试时会使用debug keystore，当正式发布应用时必须使用release版的keystore对应用进行签名。

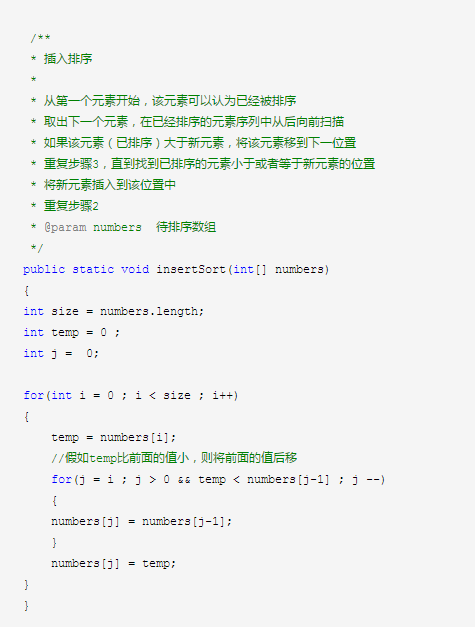
7.如果对APK正式签名,还需要使用zipalign工具对APK进行对齐操作,这样做的好处是当应用运行时减少内存的开销。

3.24Dalvik和JVM区别

1. Dalvik 基于寄存器，而 JVM 基于栈。基于寄存器的虚拟机对于更大的程序来说，在它们编译的时候，花费的时间更短。
2. Dalvik执行.dex格式的字节码，而JVM执行.class格式的字节码。
3. Dalvik 和 Java SDK的SDK不同。
4. Dalvik 和 Java 运行环境的区别
5. Dalvik 经过优化，允许在有限的内存中同时运行多个虚拟机的实例，并且每一个Dalvik 应用作为一个独立的Linux 进程执行。独立的进程可以防止在虚拟机崩溃的时候所有程序都被关闭。

# 排序算法

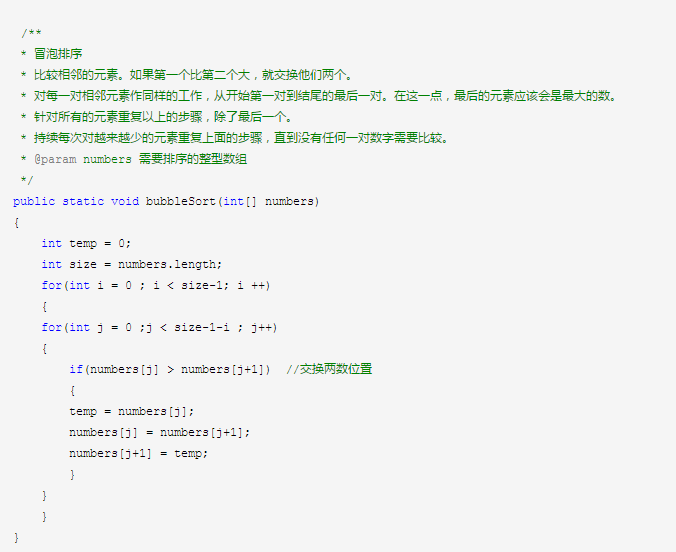
## 4.1直接插入排序



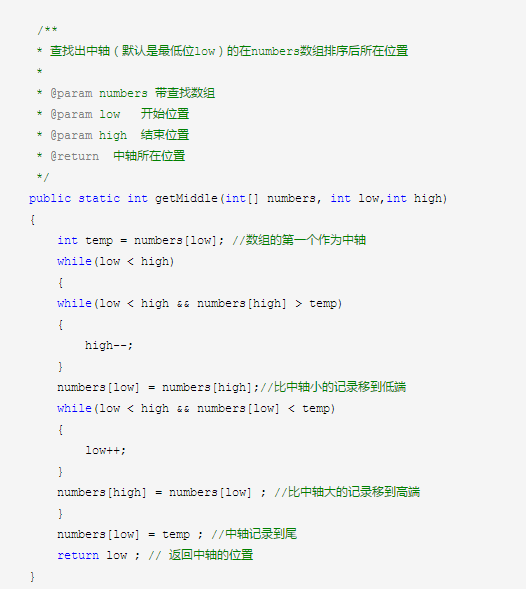
## 4.2希尔排序



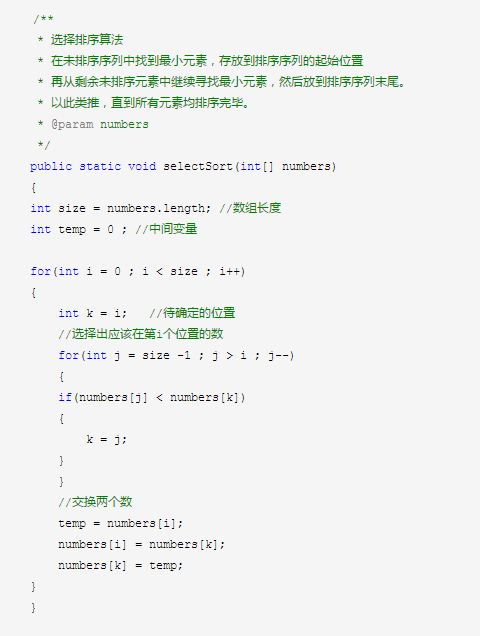
## 4.3冒泡排序



## 4.4快速排序

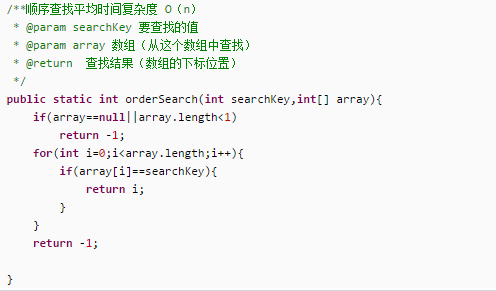


## 4.5选择排序



# 查找算法

## 5.1顺序查找



## 5.2二分查找



# 剑指offer

## 6.1输入一棵二叉搜索树，将该二叉搜索树转换成一个排序的双向链表。要求不能创建任何新的结点，只能调整树中结点指针的指向

public class TreeNode {

int val = 0;

TreeNode left = null;

TreeNode right = null;

public TreeNode(int val) {

this.val = val;

}

}

二叉查找树（英语：Binary Search Tree），也称二叉搜索树、有序二叉树（英语：ordered binary tree），排序二叉树（英语：sorted binary tree），是指一棵空树或者具有下列性质的二叉树：

任意节点的左子树不空，则左子树上所有结点的值均小于它的根结点的值；

任意节点的右子树不空，则右子树上所有结点的值均大于它的根结点的值；

任意节点的左、右子树也分别为二叉查找树；

没有键值相等的节点。

//递归调用 左 根 右 遍历

public class Solution {

//双向链表的左边头结点和右边头节点

TreeNode leftHead = null;

TreeNode rightHead = null;

public TreeNode Convert(TreeNode pRootOfTree) {

//递归调用叶子节点的左右节点返回null

if(pRootOfTree==null) return null;

//第一次运行时，它会使最左边叶子节点为链表第一个节点

Convert(pRootOfTree.left);

if(rightHead==null){

leftHead= rightHead = pRootOfTree;

}else{

//把根节点插入到双向链表右边，rightHead向后移动

rightHead.right = pRootOfTree;

pRootOfTree.left = rightHead;

rightHead = pRootOfTree;

}

//把右叶子节点也插入到双向链表（rightHead已确定，直接插入）

Convert(pRootOfTree.right);

//返回左边头结点

return leftHead;

}

}

## 6.2操作给定的二叉树，将其变换为源二叉树的镜像

public void Mirror(TreeNode root) {

if(root==null) return;

//交换左右子节点

TreeNode tempNode;

tempNode = root.left;

root.left = root.right;

root.right = tempNode;

Mirror(root.left);

Mirror(root.right);

}

## 6.3在一个二维数组中，每一行都按照从左到右递增的顺序排序，每一列都按照从上到下递增的顺序排序。请完成一个函数，输入这样的一个二维数组和一个整数，判断数组中是否含有该整数。

public boolean Find(int [][] array,int target) {

int i = array.length -1;

int j = array[0].length -1;

int n = 0;

while(j>=0 && n<=i){

if(target == array[n][j]){

return true;

}else if(target >array[n][j]){

n++;

}else{

j--;

}

}

return false;

}

从右上角的点开始

## 6.4输入一个整数，输出该数二进制表示中1的个数。其中负数用补码表示。

public int NumberOf1(int n) {

int count =0;

while(n!=0){

count++;

n = n&(n-1);

}

return count;

}

## 6.5从上往下打印出二叉树的每个节点，同层节点从左至右打印。

public ArrayList<Integer> PrintFromTopToBottom(TreeNode root) {

ArrayList<Integer> listVal =new ArrayList<Integer>();

if(root==null) return listVal;

ArrayList<TreeNode> listNode =new ArrayList<TreeNode>();

listNode.add(root);

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

TreeNode t = listNode.get(i);

listVal.add(t.val);

if(t.left!=null) listNode.add(t.left);

if(t.right!=null) listNode.add(t.right);

}

return listVal;

}

## 6.6输入一个链表，从尾到头打印链表每个节点的值

public ArrayList<Integer> printListFromTailToHead(ListNode listNode) {

Stack<Integer> st = new Stack<Integer>();

while(listNode!=null){

st.push(listNode.val);

listNode = listNode.next;

}

ArrayList<Integer> list = new ArrayList<Integer>();

while(!st.empty()){

list.add(st.pop());

}

return list;

}

## 6.7输入两个整数序列，第一个序列表示栈的压入顺序，请判断第二个序列是否为该栈的弹出顺序。假设压入栈的所有数字均不相等。例如序列1,2,3,4,5是某栈的压入顺序，序列4，5,3,2,1是该压栈序列对应的一个弹出序列，但4,3,5,1,2就不可能是该压栈序列的弹出序列。

public boolean IsPopOrder(int [] pushA,int [] popA) {

if(pushA.length==0 || popA.length==0){

return false;

}

Stack<Integer> stack = new Stack<Integer>();

int j=0;

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

stack.push(pushA[i]);

while(j<popA.length && stack.peek()==popA[j]){

j++;

stack.pop();

}

}

return stack.isEmpty();

}

## 6.8输入一个链表，反转链表后，输出链表的所有元素。

public class ListNode {

int val;

ListNode next = null;

ListNode(int val) {

this.val = val;

}

}

public ListNode ReverseList(ListNode head) {

if(head==null || head.next==null){

return head;

}

ListNode p=head.next;

ListNode newHead=ReverseList(p);

p.next = head;

head.next =null;

return newHead;

}

## 6.9一只青蛙一次可以跳上1级台阶，也可以跳上2级……它也可以跳上n级。求该青蛙跳上一个n级的台阶总共有多少种跳法

public int JumpFloorII(int target) {

　　 if(target<=1) return 1;

return 1<<(target-1);

}

## 6.10输入两个单调递增的链表，输出两个链表合成后的链表，当然我们需要合成后的链表满足单调不减规则。

public ListNode Merge(ListNode list1,ListNode list2) {

if(list1==null) return list2;

if(list2==null) return list1;

ListNode mergeHead=null;

if(list1.val<=list2.val){

mergeHead = list1;

mergeHead.next = Merge(list1.next,list2);

}else{

mergeHead = list2;

mergeHead.next = Merge(list1,list2.next);

}

return mergeHead;

}

## 6.11请实现一个函数，将一个字符串中的空格替换成“%20”。例如，当字符串为We Are Happy.则经过替换之后的字符串为We%20Are%20Happy。

public String replaceSpace(StringBuffer str) {

if(str==null){

return null;

}

StringBuffer sb = new StringBuffer();

char [] strChar = str.toString().toCharArray();

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

if(strChar[i]==' '){

sb.append("%20");

}else{

sb.append(strChar[i]);

}

}

return sb.toString();

}

## 6.12数组中有一个数字出现的次数超过数组长度的一半，请找出这个数字。例如输入一个长度为9的数组{1,2,3,2,2,2,5,4,2}。由于数字2在数组中出现了5次，超过数组长度的一半，因此输出2。如果不存在则输出0。

public int MoreThanHalfNum\_Solution(int [] array) {

if(array.length==0 || array==null) return 0;

int temp;

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

temp = array[i];

int count=0;

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

if(array[j]==temp){

count++;

}

}

if(count>(array.length/2)){

return temp;

}

}

return 0;

}

## 6.13现在要求输入一个整数n，请你输出斐波那契数列的第n项

public int Fibonacci(int n) {

int i = 0, j = 1;

for(;n>0;n--){

j += i;

i = j-i;

}

return i;

}

## 6.14把一个数组最开始的若干个元素搬到数组的末尾，我们称之为数组的旋转。输入一个非递减序列的一个旋转，输出旋转数组的最小元素。例如数组{3,4,5,1,2}为{1,2,3,4,5}的一个旋转，该数组的最小值为1。

public int minNumberInRotateArray(int [] array) {

if(array.length==0){

return 0;

}

for(int i =0;i<array.length-2;i++){

if(array[i+1] < array[i]){

return array[i+1];

}

}

return array[0];

}

## 6.15用两个栈来实现一个队列，完成队列的Push和Pop操作。 队列中的元素为int类型。

public class Solution {

Stack<Integer> stack1 = new Stack<Integer>();

Stack<Integer> stack2 = new Stack<Integer>();

public void push(int node) {

stack1.push(node);

}

public int pop() {

if(stack2.isEmpty()){

while(!stack1.isEmpty()){

stack2.push(stack1.pop());

}

}

return stack2.pop();

}

}

## 6.16输入一个整数数组，实现一个函数来调整该数组中数字的顺序，使得所有的奇数位于数组的前半部分，所有的偶数位于位于数组的后半部分，并保证奇数和奇数，偶数和偶数之间的相对位置不变。

public void reOrderArray(int [] array) {

　　if(array.length==0 || array==null){

return;

}

List<Integer> oddList = new ArrayList<Integer>();

List<Integer> evenList = new ArrayList<Integer>();

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

if(array[i]%2==0){

evenList.add(array[i]);

}else{

oddList.add(array[i]);

}

}

oddList.addAll(evenList);

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

array[i] = oddList.get(i);

}

}

## 6.17输入一个链表，输出该链表中倒数第k个结点。

public ListNode FindKthToTail(ListNode head,int k) {

if(head==null || k<=0){

return null;

}

ListNode last = head;

for(int i=1;i<k;i++){

last = last.next;

if(last==null){

return null;

}

}

while(last.next!=null){

head = head.next;

last = last.next;

}

return head;

}