# 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();

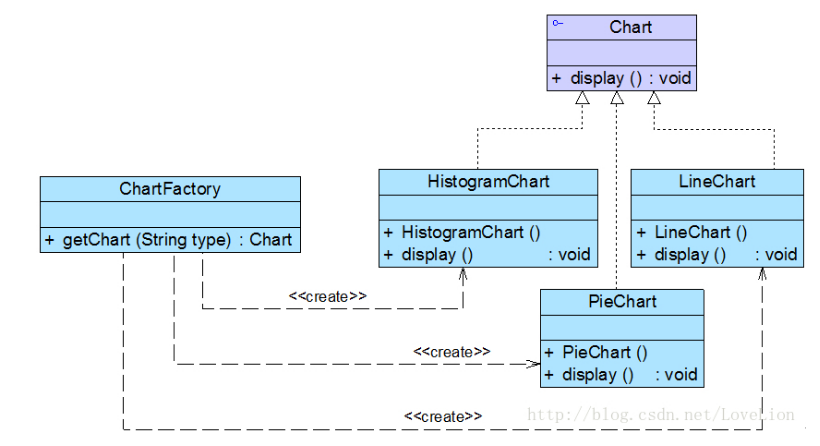
System.out.println("初始化设置折线图！");

}

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

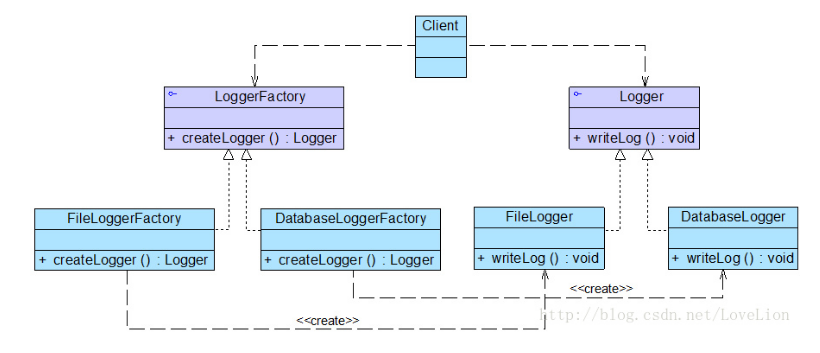
Logger logger = new FileLogger();

//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();

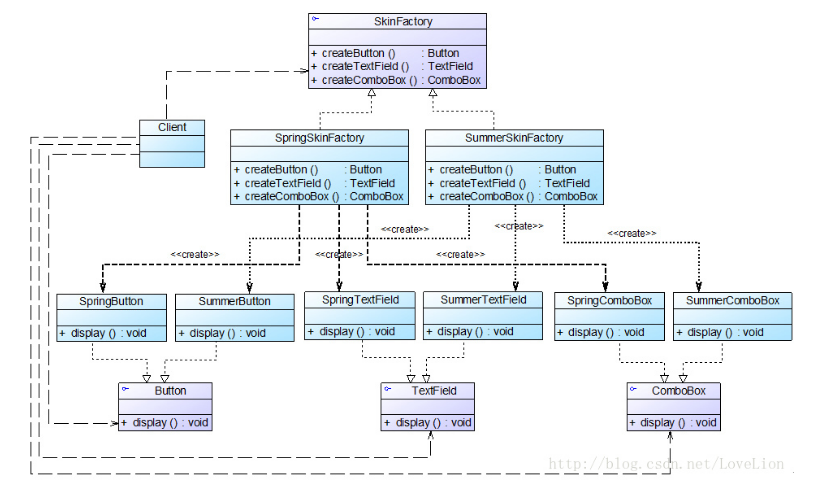
}

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)) {

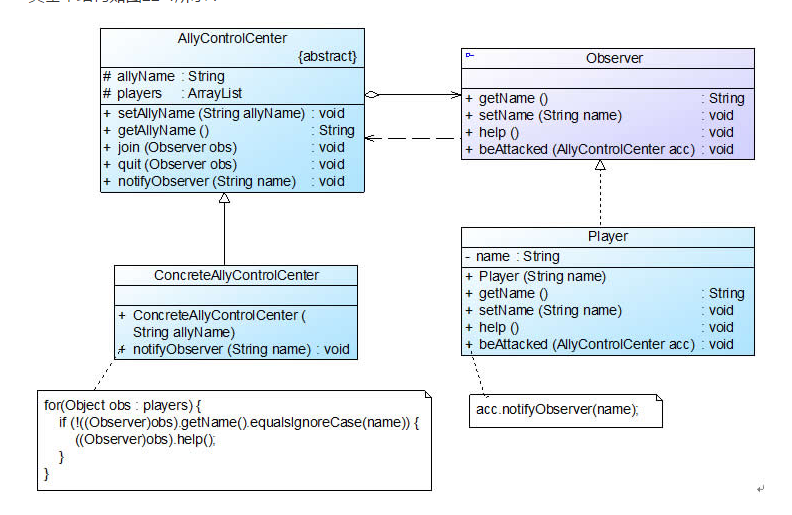
((Observer)obs).help();

}

}

}

}



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;

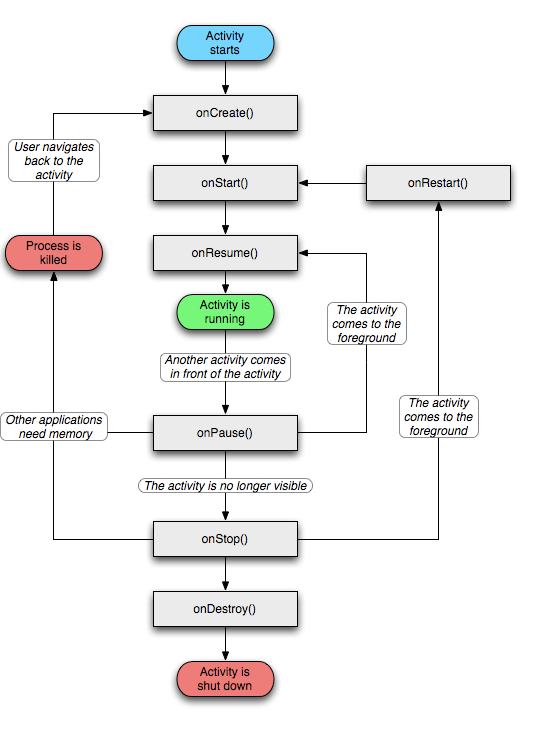
actor = ac.construct(ab); //通过指挥者创建完整的建造者对象

# Java

# 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

## 3.3BroadcastReceiver

## 3.4ContentProvider

## 3.5Fragment

## 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；

## 3.7Android性能优化

## 3.8View绘制流程

## 3.9Android事件分发机制

## 3.10自定义UI

## 3.11多线程

### 3.11.1继承Thread类

### 3.11.2实现Runnable接口

### 3.11.3AsyncTask

### 3.11.4Handler

### 3.11.5HandlerThread

### 3.11.6IntentService

## 3.12动画

## 3.13IPC通信方式

## 3.14AIDL

## 3.15Android新特性

## 3.16Binder机制

## 3.17GC

## 3.18Android安全

1. 混淆
2. Jni
3. 加固、去壳

## 3.19内存分析

## 3.20apk瘦身

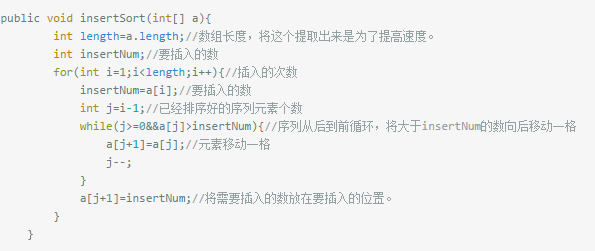
1. 删除无用资源，使用lint删除无用资源
2. ProGuard混淆
3. 移除无用库
4. 图片压缩
5. Gradle开启shrinkResources

## 3.21内存泄漏

## 3.22开源框架

# 排序算法

## 4.1直接插入排序



## 4.2希尔排序

# 5.查找算法