目录

[**介绍自己的项目，为什么使用这些框架**](#介绍自己的项目) **1**

[**项目遇到的困难**](#项目遇到的困难) **4**

[**View的绘制**](#View的绘制)

[1/自定义了一个ExpandableTextView](#自定义了一个ExpandableTextView)

[**对其的TextView**](#对其的TextView)

[**Dagger单例注入**](#Dagger单例注入)

1. **[介绍自己的项目，为什么使用这些框架](#介绍自己的项目)**

**使用dagger**：a.增加开发效率，将新建对象的活都交给dagger来做,也不用写单例模式

b.更好的管理实例类，全局性对象交给application管理。各个activity对应一个component，里面的对象与activity生命周期相同。

c.解耦，若某对象的构造函数发生变化，要修改很多代码，利用dagger解耦可以只需在module中进行修改

**使用RxJava**：正常写一个异步线程更新ui的操作很繁琐，利用RXJava使得代码可读性更强，同时切换线程的代码也非常简单，利用map还可以轻松的转换获得数据的类型，通过响应式编程可解决背压问题

**使用Retrofit:**网络请求的方式非常简单利用接口即可实现，同时适配多种callAdapter，**适配RXjava，让网络请求更加的简洁。**

**MVP：**解耦activity，让activity单纯的为显示界面的功能。交数据交互工作交给presenter来使用。

1. [**项目遇到的困难**](#项目遇到的困难)

**有关自定义View**

**[View的绘制](#View的绘制)**

Perform**Measure**——measure——onMeasure——measureChildren——children的measure

MeasureSpec

1. unspecified 要多少给多少
2. at\_most 顶天给多少
3. exactly 确切的值

对于Wrap——content需要重写onMeasure

onMeasure中，最后设置setMeasureDimension(measurewidth,measureHeight)

getChildCount

getChildAt

MeasureSpec.getMode

MeasureSpec.getSize

widthMode

getMeasureWidth

dectorView通过屏幕的大小和自身的layoutParam决定

其他的View根据父类的MeasureSpec以及自身的LayoutParam决定

Perform**Layout**——layout（确定自身的布局）——setFrame——onLayout(ViewGroup必须重写，children调用layout来完成自身的布局。)

ViewGroup一定要实现onLayout

Perform**Draw**——drawBackground——onDraw（空方法 绘制自己）——distachDraw（将绘制事件分发给子事件）——onDrawScrollBars（绘制装饰）

**Visible属性** visible可见 invisible不可见，保留控件位置，gone不可见，不保留控件位置，控件会被完全移除

**困难**。

[1/自定义了一个ExpandableTextView](#自定义了一个ExpandableTextView)

**package** com.ali.zhihu.widget;  
  
**import** android.content.Context;  
**import** android.content.res.TypedArray;  
**import** android.graphics.drawable.Drawable;  
**import** android.util.AttributeSet;  
**import** android.view.View;  
**import** android.view.ViewGroup;  
**import** android.widget.ImageButton;  
**import** android.widget.LinearLayout;  
**import** android.widget.TextView;  
  
**import** com.ali.zhihu.R;  
  
**import** org.w3c.dom.Text;  
  
*/\*\*  
 \* Created by Administrator on 2018/2/9.  
 \*/***public class** ExpandableTextView **extends** LinearLayout{  
 **private int maxCollapseLines**;  
 **private** TextView **expandableText**;  
 **private** ImageButton expandCollapseButton;  
 **private boolean collapsed**;  
 **private** Drawable **expendDrawable**;  
 **private** Drawable **collapsedDrawable**;  
 **private boolean isChange**;  
 **private** OnExpandStateChangeListener **listener**;  
  
 **public** ExpandableTextView(Context context) {  
 **super**(context,**null**);  
 }  
  
 **public** ExpandableTextView(Context context, AttributeSet attrs){  
 **super**(context,attrs);  
 init(context,attrs);  
 }  
  
 **private void** init(Context context, AttributeSet attrs){  
 setOrientation(***VERTICAL***);  
 **isChange** = **false**;  
 **collapsed** = **true**;*//表示折叠* TypedArray array = context.obtainStyledAttributes(attrs, R.styleable.***ExpandableTextView***);  
 **maxCollapseLines** = array.getInteger(R.styleable.***ExpandableTextView\_maxCollapseLines***,2);  
 **expendDrawable** = array.getDrawable(R.styleable.***ExpandableTextView\_expandDrawable***);  
 **collapsedDrawable** = array.getDrawable(R.styleable.***ExpandableTextView\_collapseDrawable***);  
 array.recycle();  
 }  
  
 @Override  
 **protected void** onFinishInflate() {  
 **super**.onFinishInflate();  
  
 **expandableText** = (TextView)findViewById(R.id.***expandable\_text***);  
 expandCollapseButton = (ImageButton)findViewById(R.id.***expand\_collapse***);  
 expandCollapseButton.setImageDrawable(**collapsed**?**collapsedDrawable**:**expendDrawable**);  
 expandCollapseButton.setOnClickListener(**new** OnClickListener() {  
 @Override  
 **public void** onClick(View view) {  
 **isChange** = **true**;  
 **collapsed** = !**collapsed**;  
 **if**(**listener** != **null**){  
 **listener**.onExpandStateChanged(**collapsed**);  
 }  
 expandCollapseButton.setImageDrawable(**collapsed**?**collapsedDrawable**:**expendDrawable**);  
 requestLayout();  
 }  
 });  
 }  
  
 @Override  
 **protected void** onMeasure(**int** widthMeasureSpec, **int** heightMeasureSpec) {  
 **if**(!**isChange** || expandCollapseButton.getVisibility() == ***GONE***){  
 **super**.onMeasure(widthMeasureSpec, heightMeasureSpec);  
 **return**;  
 }  
 **isChange** = **false**;  
  
 expandCollapseButton.setVisibility(***GONE***);  
 **expandableText**.setMaxLines(Integer.***MAX\_VALUE***);  
 **super**.onMeasure(widthMeasureSpec,heightMeasureSpec);  
  
 **if**(**expandableText**.getLineCount() <= **maxCollapseLines**){  
 **return**;  
 }  
 **if**(**collapsed**){  
 **expandableText**.setMaxLines(**maxCollapseLines**);  
 }  
 expandCollapseButton.setVisibility(***VISIBLE***);  
 **super**.onMeasure(widthMeasureSpec,heightMeasureSpec);  
 }  
  
 **public void** setText(String text){  
 **isChange** = **true**;  
 **expandableText**.setText(text);  
 }  
  
 **public void** setText(String text,**boolean** collapse){  
 **this**.**collapsed** = collapse;  
 expandCollapseButton.setImageDrawable(**collapsed**?**collapsedDrawable**:**expendDrawable**);  
 setText(text);  
 getLayoutParams().**height** = ViewGroup.LayoutParams.***WRAP\_CONTENT***;  
 requestLayout();  
 }  
  
 **public interface** OnExpandStateChangeListener{  
 **void** onExpandStateChanged(**boolean** collapse);  
 }  
  
 **public void** setExpandStateChangeListener(OnExpandStateChangeListener listener){  
 **this**.**listener** = listener;  
 }  
}

**[对其的TextView](#对其的TextView)**

**package** com.ali.zhihu.widget;  
  
**import** android.content.Context;  
**import** android.graphics.Canvas;  
**import** android.graphics.Paint;  
**import** android.text.Layout;  
**import** android.text.StaticLayout;  
**import** android.text.TextPaint;  
**import** android.util.AttributeSet;  
**import** android.widget.TextView;  
  
*/\*\*  
 \* Created by Administrator on 2018/2/9.  
 \*/***public class** JustifyTextView **extends** TextView {  
 **private int mLineY**;  
 **private int mViewWidth**;  
  
 **public** JustifyTextView(Context context, AttributeSet attrs){  
 **super**(context,attrs);  
 }  
  
 @Override  
 **protected void** onLayout(**boolean** changed,**int** left,**int** top,**int** right,**int** boottom){  
 **super**.onLayout(changed,left,top,right,boottom);  
 }  
  
 @Override  
 **protected void** onDraw(Canvas canvas){  
 TextPaint paint = getPaint();  
 paint.setColor(getCurrentTextColor());  
 paint.**drawableState** = getDrawableState();  
 **mViewWidth** = getMeasuredWidth() - getPaddingRight();  
 String text = getText().toString();  
 **mLineY** = getPaddingTop();  
 **mLineY** += getTextSize();  
 Layout layout = getLayout();  
  
 **if**(layout == **null**){  
 **return**;  
 }  
  
 Paint.FontMetrics fm = paint.getFontMetrics();  
 **int** textHeight = (**int**)(Math.*ceil*((fm.**descent** - fm.**ascent**)));  
 textHeight = (**int**)(textHeight \* layout.getSpacingMultiplier() + layout.getSpacingAdd());  
 **int** textLine = layout.getLineCount();  
  
 **for**(**int** i = 0; i < textLine;i++){  
 **int** lineStart = layout.getLineStart(i);  
 **int** lineEnd = layout.getLineEnd(i);  
 String lineText = text.substring(lineStart,lineEnd);  
 **float** lineWidth = StaticLayout.*getDesiredWidth*(text, lineStart, lineEnd, paint);  
  
 **if**(i == textLine - 1){  
 *//最后一行则直接画* canvas.drawText(lineText,0,**mLineY**,paint);  
 }**else**{  
 **if**(needScale(lineText)){  
 drawScaleText(canvas,lineText,paint,lineWidth);  
 }**else**{  
 canvas.drawText(lineText,0,**mLineY**,paint);  
 }  
 }  
 **mLineY** += textHeight;  
 }  
  
 }  
 **private void** drawScaleText(Canvas canvas, String lineText, TextPaint paint,**float** lineWidth){  
 **int** x = 0;  
 **int** gapCount = lineText.length() - 1;  
 */\*if(lineText.length() > 2 && lineText.charAt(0) == ' ' && lineText.charAt(1) == ' '){  
 float blankWidth = StaticLayout.getDesiredWidth(" ",paint);  
 x += blankWidth;  
 canvas.drawText(" ",0,mLineY,paint);  
 i = 2;  
 }\*/* **float** gap = (**mViewWidth** - lineWidth)/gapCount;  
 **for**(**int** i = 0;i < lineText.length();i++){  
 canvas.drawText(String.*valueOf*(lineText.charAt(i)),x,**mLineY**,paint);  
 x += gap + StaticLayout.*getDesiredWidth*(String.*valueOf*(lineText.charAt(i)),paint);  
 }  
 }  
  
 **private boolean** needScale(String lineText){  
 **if**(lineText == **null** || lineText.length() == 0){  
 **return false**;  
 }  
 *//如果没有遇到原先设置的换行，则自动调整字间的间距* **return** lineText.charAt(lineText.length() - 1) != **'\n'**;  
 }  
}

1. **[Dagger单例注入](#Dagger单例注入)**

@Scope

@Retention(RetentionPolicy.Running)

Public @interface perActivity{

}

Public class person{

Public person(){

}

}

@Module

Public class mainModule{

@Provide

@PerActivity

Person providePerson(){

}

}

@perActivity

@component(dependencies = AppComponent,module = mainModule.class)

Interface mainComponent{

Void inject(Activity actibity);

FragmentComponent fragmentComponent();

}

Public class activity eatends Activity{

@inject

Person person;

}

则在activity生命周期中，只会创建一个person。

由于被Scope修饰的对象工程会被存入DoubleCheck中

DoubleCheck用类似双重if的方式创建单例模式，保证只会创建一个对象

若没用Scope修饰，则在DaggerMainComponent中通过工厂创建

若在此时用Singleton修饰，不会创造全局单例，Singleton其实为Scope的一种，若想用它创建，只能创建在Application中，由于Application是全局性的，所以被Singleton创建出来的单例就是全局性的。

/\*\*

\* 自定义一个限定符

\*/

@Qualifier//限定符

@Documented

@Retention(RetentionPolicy.RUNTIME)

public @interface Type {

String value() default "";//默认值为""

}

@Module

public class SaladModule {

.........

@Type("normal")//使用限定符来区别使用哪个构造函数返回对象

@Provides

public Apple provideNormalApple() {

return new Apple();

}

@Type("color")//使用限定符来区别使用哪个构造函数返回对象

@Provides

public Apple provideColorApple(String color) {

return new Apple(color);

}

// 由于我们的Apple构造函数里使用了String,所以这里要管理这个String(★否则报错)

// int等基本数据类型是不需要这样做的

@Provides

public String provideString(){

return new String();

}

@Component(modules = {SaladModule.class})//指明要在那些Module里寻找依赖

public interface SaladComponent {

.........

@Type("normal")//关键靠这个限定符来区分调用哪个构造函数

Apple provideNormalApple();//这个方法名并不重要，只要是provide开头就行，但是建议和module里一致

@Type("color")

Apple provideColorApple();//若想暴露给dependenci，则需要提供该函数，若想暴露给subComponent 则不用。

String provideString();

//注意：下面的这个方法，表示要将以上的三个依赖注入到某个类中

//这里我们把上面的三个依赖注入到Salad中

//因为我们要做沙拉

void inject(Salad salad);

}

}

public class Salad {

.........

@Inject

@Type("normal")

Apple normalApple;

@Inject

@Type("color")

Apple colorApple;

public Salad() {

// DaggerSaladComponent编译时才会产生这个类，

// 所以编译前这里报错不要着急(或者现在你先build一下)

SaladComponent saladComponent = DaggerSaladComponent.create();

saladComponent.inject(this);//将saladComponent所连接的SaladModule中管理的依赖注入本类中

makeSalad(pear, banana, saladSauce);

}

private void makeSalad(Pear pear, Banana banana, SaladSauce saladSauce) {

Log.e("TAG", "我在搅拌制作水果沙拉");

}

}

1. **Activity生命周期，给你具体场景，生命周期应该是怎样的**

OnCreate——onStart——onRestoreInstanceState——onresume——onPause——onsaveOnstanceState——onStop——onDestroy

Single

singleTop（防止点击多次通知栏）

singleTask(保证只有一个聊天界面)

singleInstance(微信共享地图界面)

1. **Service的启动方式，如何关闭**

startService start之后，activity与Service无任何关系，activity不可以控制Service 用stopService关闭

bindService bind之后，activity与Service绑定，activity可以控制Service做一些事情

要unBinde

所有的客户端共享一个IBinder，要关闭BindService需要所有的客户端解除绑定

1. **广播两种注册方式，有什么区别，是不是所有的接收者都能够接收广播，广播如何截断**

静态注册：在androidManifest.xml文件中静态注册，该广播接收器android不可以自动销毁。在activity销毁后，任然可以接受信息。在<application>下添加<receiver>,在<receiver>中添加多个<intent-filter>

动态注册：继承BroadcastReceiver，重写onReceive方法。在registerReceiver中注册广播接收器，传参为BroadCastReceiver与IntentFilter，可以用unregisterReceiver解除注册，解除注册后无法接收到广播

<Intent-filter

<action> intent行为

<category> intent类型 默认有个default 注册activity时一定要添加

<date>数据类型，intent要传递的数据

/>

Public class myBroadcastReceiver extends BroadCastReceiver{

@Override

Public void onReceive(Context context,Intent intent){

}

}

Public class myActivity extends Activity{

myBroadcastReceiver;

IntentFilter intentfilter = new IntentFilter();

intentfilter.addAction(“aaaa”);

registerReceiver(myReceiver,intentfilter);

Intent = new Intent();

Intent.setAction(“ss”);

sendBroadcast(intent);

}

广播分为普通广播，有序广播，本地广播。

有序广播按接收器的优先级接受广播，可用abortBroadcast()对广播进行拦截。BroadcastReceiver可以使用setResult系列函数来结果传给下一个BroadcastReceiver，通过getResult系列函数来取得上个BroadcastReceiver返回的结果，并可以abort系列函数来让系统丢弃该广播让，使用该广播不再传送到别的BroadcastReceiver。短信接受为有序广播，可以拦截短信，电话。

本地广播广播只在本地传播。防止所有应用程序能接收到带来安全隐患

滞留（粘性）广播，若无接受者，一直滞留到有接受者了才接受，保障动态注册广播接收器时能迅速收到广播，监听电池电量

广播事件：来电，电量过低，电量充满，屏幕关闭，屏幕唤醒，飞行模式，

1. 事件分发，如何拦截事件
2. View绘制，specMeasure
3. Java集合 如何实现生产者消费者
4. **Java如何构造线程**
5. new Thread 重写run函数
6. 用Runnable参数去创造Thread
7. 用线程池去构造

Excutor ecxutor = Excutor.newFixedThreadPoll(10);

newFixedThreadPoll:corePoolSize = 10;maxmunPollSize = 10;keepalive = 0;LinkedBlockingQueue

newCachedThreadPoll(10); corePollSize = 10 maxmunPollSize = 10,keepalive = 60s,SynchronizedBlokingQueue;

newSingleThreadExcutor() corePollsize = 1 maxmunPollsize = 1; keepalive = 0 linkedBlokingQueue; 若线程没完成异常跳出，会创建一个线程继续运行该任务

newScheduledThreadPoll 可周期完成某个工作

线程池无返回

Executor.excute(runnable);

有一阻塞队列，若当前线程数小于核心线程数的话，直接创建任务，若当前线程数大于核心线程数小于最大线程数，叫任务加入队列，若队列阻塞，当前线程数小于最大线程数，则直接创建线程，否则按handler机制去处理线程，最常见的方法是将线程丢弃。当运行完一线程，看队列里是否有任务，有则取任务出来放入线程工作，线程由Work实现

线程池有返回值

利用Callable Future FutureTask使用，这只能与线程池一起使用

1. myCallable implements Callable<Integer>{

public Integer call(){

}

}

1. Future future = ExcutorService.submit(myCallable);
2. FutureTask<Integer> futureTask = new FutureTask<>(myCallable);
3. //可以用get得到结果
4. //判断任务是否完成 isDone
5. //删除任务 cancel
6. executorService。Submit(futureTask);
7. 利用AsyncTask（封装了线程池以及Handle）

Public myAsyncTask extends AsyncTask<params,progress,result>{

Public void onPreExcute(){

}

Public result doInBackground(Params… params){

publishProgress()

}

Public onPostExcute(Result result){

}

Public onProgressUpdate(progress… progress){

}

}

执行 new myAsuncTask().excute();

默认串行执行，防止doInBackgroud访问共享变量线程不安全

并行执行：myAsyncTask.excuteOnExecutor(AsyncTask.Thread\_POLL\_EXECUTOR)

必须初始化在主线程（防止在非ui线程更新ui），

有多个线程容易崩溃，不用AsyncTask后，系统中也会存在剩余线程

原理：有一work（callable）封装在futureTask中，执行excute，将任务装入队列中，真正执行excuteOnExecuter（在里面执行onPreExcute，将params付给work，利用线程池submit任务）调用work的call（在里面执行doInBackground），完成后postResult将结果创给messageQueue，利用handle处理，调用onPostExecute，publishProgress也是发送message，调用onProgressUpdate

1. **Java锁有哪些，具体到如何使用**

**有关同步：**

内存分配：cpu指令运行告诉，物理内存读取较慢，为了增加运行速度，每个cpu对应一个高速缓存，将数据从物理内存读取到高速缓存中，处理数据后将数据写会物理内存。若多个线程同时处理这个数据就会出现问题，提出缓存一致性原则：对某个数据进行写操作时，令其他高速缓存中的该数据失效，将该数据的新值立刻更新到物理内存中，其他高速缓存从物理内存中重新读取该数据。

Java内存分配：每个线程对应一个工作内存，工作内存中不能访问，从主存中读取数据到工作内存中进行操作，之后写会主存。

原子性：一系列操作要不不发生，要不一起发生

可见性：对数据进行写操作后，立即更新到主存中

Volatile在底层相当于在写操作后面加了一个内存屏障，内存屏障之前的语句一定先于内存屏障发生，内存屏障之后的语句一定后于内存屏障发生。同时会强制更新一次cpu缓存，写操作会导致其他cpu中的对应缓存失效。即保证了volatile的可见性与禁止语句重排。Volatile将更新的值立即更新到主存，若其他线程要读取该值会在主存读取

但volatile不保证原子性，自增操作不具备原子性，在自增写操作之前有读取操作，加操作，可能有多个线程已经完成读取操作，不需要再向主存读取。

CAS 比较和交换：若再内存中的值与旧的预期值一样，就将该值更新为某个值。

**有关锁：**

1. synchronized

**锁的等级：**方法锁，对象锁，类锁，若方法为静态方法则为类锁，否则为对象锁

**原理：**每个对象都有一个monitor锁，线程请求访问这个对象就是看monitor是否被占用，（monitor的owner中存放该线程的唯一标识，线程对象头中的lockword指向monitor的起始位置）。若某个线程得到了monitor，执行monitorenter操作，可以多次获得锁。退出时需要monitorexist

锁的种类：无锁状态——偏量级锁——轻量级锁——重量级锁，方向不可逆

偏量级锁只适用于只有一个线程时，一个线程持有锁之后就不会再释放，除非用CAS替换对象头的线程id。

步骤：一个对象，一个线程用CAS方法尝试将对象头的线程id附为自己的id，若成功则获得锁。若不成功则到达安全点时，线程被挂起，偏向锁变为轻量锁，执行同步操作

轻量级锁：适用于两个线程交替获取锁的状态，当一个对象处于无锁或偏向时，一个线程尝试获取对象锁，则在栈中建立一个lockRecord表，通过CAS方法将对象中的markWord复制到lockRecord,若复制成功，则将锁状态改为轻量级锁，将Object MarkWord指向lockRecord，将lockRecord中的owner指向Object。否则判断markWord是否已经指向lockRecord，若是直接变为轻量，否则膨胀为重量级锁，尝试用自旋。

重量级锁：是互斥锁，锁造作之所以要尽量避免是因为，其原理为获得monitor，而获得monitor本质依赖操作系统的重量级锁，这设计到从用户态到内核态的转化，非常繁琐，因此需要尽量的避免

有关锁的其他优化：锁自选，循环，防止从用户态转到内核态，因为一般线程持有锁的时间很短。自适应自旋，设定循环次数，动态调整；锁粗化，连续的锁变为一个锁。锁消除，若觉得没有必要则消除锁

关于Lock：

<http://www.cnblogs.com/dolphin0520/p/3923167.html>

Lock与Synchronized的差别

Synchronized为java的一个关键字，原理为获得对象的monitor进行锁操作，可重入（已经得到对象锁后，访问对象的其他同步方法，不用再次请求锁），不可中断，自动解锁，若一个线程因没有获得锁处于阻塞状态，他只能一直处于阻塞状态直到锁被其他线程释放，无法分开读写操作。

Lock为java的一个借口，内部有多个函数，可中断

1.lock（）获得锁

2.tryLock() 尝试获得锁，若不获得，则返回false

3.lockInterrupt（）抛出interruptException

4.unLock（）解锁

4.newCondition获得某个条件，await == wait signal = notify

Lock需要自己主动解锁，发送异常时并不会自动解锁

ReentrantLock 为lock的实现类，可重入锁

Lock lock1 = new ReetrantLock(true);//true为公平锁，按线程等待时间长短获得锁

Try{

Lock1.lock();

}catch(InterruptException e){

e.printStackTrace();

}finally{

Lock1.unlock();//保证锁一定会被释放，防止死锁

}

//trylock

If(lock1.trylock){

Try{}catch{}finally{lock1.unlock}

}

//lockInterrupt

Function throws interruptedException

Lock.lockInterrupt();

Try{}finally{释放锁}

ReadWriteLock 为一借口

1. readLock（）//获得读锁 可同时进行读操作
2. writelock（）//获得写锁 不可同时进行写操作与读写操作
3. 实现为ReentrantReadWriteLock

利用condition实现生产者与消费者模型，线程间的协作：

import java.util.ArrayDeque;

import java.util.Deque;

import java.util.concurrent.locks.Condition;

import java.util.concurrent.locks.Lock;

import java.util.concurrent.locks.ReentrantLock;

public class ProducterAndConsume {

Deque<Integer> queue = new ArrayDeque<>();

int size = 3;

Lock lock = new ReentrantLock();

Condition emptyQueue = lock.newCondition();

Condition fullQueue = lock.newCondition();

public Thread produce() {

Thread producter = new Thread() {

@Override

public void run() {

try {

lock.lock();

if(queue.size() == size) {

System.out.println("队列已满");

fullQueue.await();

}

queue.offer(1);

System.out.println("生产");

emptyQueue.signal();

}catch(InterruptedException e ){

e.printStackTrace();

}finally {

lock.unlock();

}

}

};

return producter;

}

public Thread consume() {

Thread consume = new Thread() {

@Override

public void run() {

try{

lock.lock();

if(queue.isEmpty()) {

emptyQueue.await();

System.out.println("等待生产者生产");

}

queue.poll();

System.out.println("消费");

fullQueue.signal();

}catch(InterruptedException e) {

e.printStackTrace();

}finally {

lock.unlock();

}

}

};

return consume;

}

public static void main(String[] args) {

ProducterAndConsume p = new ProducterAndConsume();

//p.produce().start();

p.consume().start();

p.produce().start();

}

}

1. **Sleep与wait差别，如何唤醒**

Sleep通过Thread静态调用，wait属于Object，由于与锁有关的事物都存在于Object中，因此wait会释放锁，sleep不会释放锁，由于与锁有关，wait需要用在同步块中，sleep则没有限制

线程状态有 new runnable（持有锁，等待cpu分配时间） running blocked（等待锁） killed，

Wait running——blocked

Sleep running——runnable

可用interrupt设置线程中断标志，t.interrupt()，t为某个线程对象。

T除在非阻塞状态，调用interrupt将线程阻塞状态设置为true，isInterrupt（判断线程阻塞状态）返回true

T处在阻塞状态（sleep wait synchronized），调用interrupt，会进入异常捕捉，interruptedException，阻塞状态设为false；

Interrupted（线程静态方法 ）在第一次检测时，若线程中断标志被设为true则返回true，同时jiang中断标志清空。

1. **public** **class** Interrupt  {
2. **public** **static** **void** main(String[] args) **throws** Exception {
3. Thread t = **new** Thread(**new** Worker());
4. t.start();
6. Thread.sleep(200);
7. t.interrupt();
9. System.out.println("Main thread stopped.");
10. }
12. **public** **static** **class** Worker **implements** Runnable {
13. **public** **void** run() {
14. System.out.println("Worker started.");
16. **try** {
17. Thread.sleep(500);
18. } **catch** (InterruptedException e) {
19. Thread curr = Thread.currentThread();
20. //再次调用interrupt方法中断自己，将中断状态设置为“中断”
21. curr.interrupt();
22. System.out.println("Worker IsInterrupted: " + curr.isInterrupted());
23. System.out.println("Worker IsInterrupted: " + curr.isInterrupted());
24. System.out.println("Static Call: " + Thread.interrupted());//clear status
25. System.out.println("---------After Interrupt Status Cleared----------");
26. System.out.println("Static Call: " + Thread.interrupted());
27. System.out.println("Worker IsInterrupted: " + curr.isInterrupted());
28. System.out.println("Worker IsInterrupted: " + curr.isInterrupted());
29. }
31. System.out.println("Worker stopped.");
32. }
33. }
34. }

执行结果

1. Worker started.
2. Main thread stopped.
3. Worker IsInterrupted: true
4. Worker IsInterrupted: true
5. Static Call: true
6. ---------After Interrupt Status Cleared----------
7. Static Call: false
8. Worker IsInterrupted: false
9. Worker IsInterrupted: false
10. Worker stopped.
11. 了解哪些设计模式，单利模式，dlc原理

**private** **static** ProducterAndConsume *instance*;

**public** **static** ProducterAndConsume getInstance1() {

**if**(*instance* == **null**) {

**synchronized**(ProducterAndConsume.**class**) {

**if**(*instance* == **null**) {

*instance* = **new** ProducterAndConsume();

}

}

}

**return** *instance*;

}

**private** **static** ProducterAndConsume *sInstance* = **new** ProducterAndConsume();

**public** **static** ProducterAndConsume getInstance2() {

**return** *sInstance*;

}

**public** **static** **class** Singleton{

**private** **static** ProducterAndConsume *instance* = **new** ProducterAndConsume();

}

**public** **static** ProducterAndConsume getInstance3() {

**return** Singleton.*instance*;

}

Static静态与静态函数能保证线程安全，是因为静态变量在类创建的时候就会初始化，调用静态内部类的静态变量时，导致加载静态内部类，从而初始化静态对象。

1. Java四种引用

强引用 关于集合clear

Element[i] = o;

For(int I = 0; I < arrayList.size();i++){

//正确的清楚操作

Element[i] == null;//错误的为o = null

}

软引用，内存不够gc

可以用作图片缓存，内存不够就gc

弱引用，一gc就gc

由于防止内存泄漏sanqi

虚引用，与引用队列一起使用，由于跟踪gc情况，若一个虚引用被gc，会被放入引用队列中，之后可以做一些其他的操作