**final** String url = **"https://www.baidu.com/img/bd\_logo1.png"**;  
Glide.*with*(**this**) //创建一个加载图片的实例  
 .load(url) //支持加载各种各样的图片资源，包括网络图片、本地图片、应用资源、二进制流、Uri对象等

.placeholder(R.drawable.***test\_image***) //添加占位图

.error(R.drawable.***error***) *//出现异常的时候显示的占位图*

.skipMemoryCache(**true**) *//禁用内存缓存*

.diskCacheStrategy(DiskCacheStrategy.***NONE***) *//禁用硬盘缓存*

* *Option*
* DiskCacheStrategy.NONE： 表示不缓存任何内容。
* DiskCacheStrategy.SOURCE： 表示只缓存原始图片。
* DiskCacheStrategy.RESULT： 表示只缓存转换过后的图片（默认选项）。
* DiskCacheStrategy.ALL ： 表示既缓存原始图片，也缓存转换过后的图片。

.into(**iv**); //不仅仅是只能接收ImageView类型的参数，还支持很多更丰富的用法

Glide.with()创建一个加载图片的实例。 🚶三步：with()->load()->into();最简单的使用方法；

内存缓存：

LRUC ache算法

Glide的图片加载过程中会调用两个方法来获取内存缓存，loadFromCache()采用LRU算法，loadFromActiveResources()采用弱引用

正在使用中的图片使用弱引用来进行缓存，不在使用中的图片使用LruCache来进行缓存的功能。

硬盘缓存：

.preload(); *//预加载图片 加载而不显示图片*

@Override  
**public void** onResourceReady(GlideDrawable resource, GlideAnimation glideAnimation) {  
 **iv**.setImageDrawable(resource);  
} //加载图片完成 显示图片的回调方法

**retrofit:**

动态的URL路径设置 @Path(**"name"**)

ps: <http://url.com/users/jackyang> 获取jackyang的信息

**public interface** IUser {  
 @GET(**"**users**"**)  
 Call<List<User>> getUser(@Path(**"name"**)String name); *//动态的URL路径设置*}

查询参数设置 @Query(**"name"**)

ps: [http://url.com/users?name=jackyang](http://url.com/users/jackyang) 获取jackyang的信息

**public interface** IUser {  
 @GET(**"**users**"**)  
 Call<List<User>> getUser(@Query(**"name"**)String name); *//动态的URL路径设置*}

源码：

动态代理：

在invoke方法里面，拿到所有的参数，注解信息然后就可以去构造RequestBody，再去构建Request，得到Call对象封装后返回。

ServiceMethod主要用于将我们接口中的方法转化为一个Request对象

**private** okhttp3.Call createRawCall() **throws** IOException {  
 Request request = **serviceMethod**.toRequest(**args**);  
 okhttp3.Call call = **serviceMethod**.**callFactory**.newCall(request);  
 **if** (call == **null**) {  
 **throw new** NullPointerException(**"Call.Factory returned null."**);  
 }  
 **return** call;  
}

性能优化：

1布局优化：

1. 减少布局的嵌套（减少了绘制的时间）
2. 采用较简单的viewgroup ps:LinearLayout FrameLayout
3. 采用<liclude>.<merge>标签
4. 用ViewStub按需加载布局;

2 内存优化

··总要防止内存泄漏

1. 静态变量导致内存泄漏
2. Bitmap导致内存泄漏
3. Cursor游标导致内存泄漏
4. 动画导致内存泄漏 （ps：动画没有在 onDestroy方法里取消 animator.cancel(), 导致view会被动画一直持有，而view持有Activity，导致Activity 无法释放）

**安卓里的线程和线程池：**

**AsyncTask(里面有采用线程池(总要是通过Executor来派生特定类型的线程池)):封装了Thread和Handler**

**package** com.yw.threaddemo;  
  
**import** android.os.AsyncTask;  
  
*/\*\*  
 \* Created by yw on 17/10/25.  
 \* author jackyang  
 \* com.yw.threaddemo  
 \*/***public class** AsyncTaskDemo **extends** AsyncTask<String, Integer, String> {  
  
 @Override  
 **protected void** onPreExecute() {  
 *//在ui线程中执行，ps：先弹一个加载框 表示你正在处理* **super**.onPreExecute();  
 }  
  
 @Override  
 **protected void** onPostExecute(String s) {  
 *//在ui线程中执行* **super**.onPostExecute(s);  
 }  
  
 @Override  
 **protected void** onProgressUpdate(Integer... values) {  
 *//当进度发生改变时，在ui线程中执行* **super**.onProgressUpdate(values);  
 }  
  
 @Override  
 **protected void** onCancelled(String s) {  
 *//在ui中执行，当任务被取消时执行* **super**.onCancelled(s);  
 }  
  
 @Override  
 **protected** String doInBackground(String... params) {  
 *//在子线程中执行 返回result onPostExecute得到回调的结果* **return null**;  
 }  
}

**new** AsyncTaskDemo().execute(**"yw"**); *//创建和执行task*

**注意事项：**

**AsyncTask必须在主线程里创建**

execute()必须在主线程里调用;

**IntentService:**

**采用了HandlerThread**

@Override  
**public void** onCreate() {  
 *//* ***TODO: It would be nice to have an option to hold a partial wakelock*** *// during processing, and to have a static startService(Context, Intent)  
 // method that would launch the service & hand off a wakelock.* **super**.onCreate();  
 HandlerThread thread = **new** HandlerThread(**"IntentService["** + **mName** + **"]"**);  
 thread.start();  
  
 **mServiceLooper** = thread.getLooper();  
 **mServiceHandler** = **new** ServiceHandler(**mServiceLooper**);  
}

**HandlerThread:**

**继承Thread，可以使用handler的Thread；run()方法里会**Looper.prepare()**创建一个消息队列，并且开启消息循环**Looper.*loop*();

**public class** HandlerThread **extends** Thread {

…

@Override  
**public void** run() {  
 **mTid** = Process.*myTid*();  
 Looper.*prepare*();  
 **synchronized** (**this**) {  
 **mLooper** = Looper.*myLooper*();  
 notifyAll();  
 }  
 Process.*setThreadPriority*(**mPriority**);  
 onLooperPrepared();  
 Looper.*loop*();  
 **mTid** = -1;  
}

…}

**安卓消息机制 Handler消息机制**

handler :

**public boolean** sendMessageAtTime(Message msg, **long** uptimeMillis) {  
 MessageQueue queue = **mQueue**;  
 **if** (queue == **null**) {  
 RuntimeException e = **new** RuntimeException(  
 **this** + **" sendMessageAtTime() called with no mQueue"**);  
 Log.*w*(**"Looper"**, e.getMessage(), e);  
 **return false**;  
 }  
 **return** enqueueMessage(queue, msg, uptimeMillis);  
}

enqueueMessage:将消息插入消息队列里；

IPC机制：

进程：一般指一个执行单元，例如一个应用。

线程: cpu的调度的最小单元。

·····进程可以包含多个线程。

安卓里进程间通信：一般采用binder或者socket

一个应用里创建多个进程：

在四大组件里指定 配置

创建私有进程

**android:process=":progress" 进程为包名+:progress**

**创建全局进程**

**android:process="cn.com.progeress" 进程为cn.com.progeress**

**二者区别：**

**其他应用不可以跟私有进程在同一个进程**

**其他应用可以通过ShareUID跟全局进程在同一个进程**

**序列化接口：**

**Serializable**

**Parcelable**

**二者的区别：**

**Serializable是java中的序列化接口**

**Serializable使用较简单，占用内存大**

**Parcelable是安卓中的序列化接口**

**Parcelable使用较复杂，占用内存小**

**一．Bundle：**

**Activity service** broadcast 通过Intent中传递Bundle数据，数据必须能够被序列化

二．文件共享：

注意线程同步问题，防止多个线程改同一个文件

三.Messenger:

底部的实现用到了binder

**public** Messenger(IBinder target) {  
 **mTarget** = IMessenger.Stub.asInterface(target);  
}

Messenger跨进程通信底部的实现采用了Binder

**看Messenger的构造函数**

\*\*public final class Messenger implements Parcelable\*\*

public Messenger(Handler target) {

mTarget = target.getIMessenger();

}

public Messenger(IBinder target) {

mTarget = IMessenger.Stub.asInterface(target);

}

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10

\*\*

结合例子来帮助理解

\*\*

* 先创建个服务进程 MService.class

package com.yw.messengerdemo.service;

import android.app.Service;

import android.content.Intent;

import android.os.Bundle;

import android.os.Handler;

import android.os.IBinder;

import android.os.Message;

import android.os.Messenger;

import android.os.RemoteException;

import android.util.Log;

/\*\*

\* Created by yw on 17/10/26.

\* author jackyang

\* com.yw.messengerdemo.service

\*/

public class MService extends Service {

private final static class MHandler extends Handler {

@Override

public void handleMessage(Message msg) {

switch (msg.what) {

case 0:

String temp = msg.getData().getString("test");

Log.d("yw", temp);

//给客户端回复消息

Messenger m = msg.replyTo;

//创建一个消息Message

Message message = Message.obtain(null, 1);

Bundle bundle = new Bundle();

bundle.putString("msgByService", "客户端小弟，你少给老子发消息啦,我知道你发的消息是：" + temp);

message.setData(bundle);

try {

//发送消息

m.send(message);

} catch (RemoteException e) {

e.printStackTrace();

}

break;

}

super.handleMessage(msg);

}

}

//实例一个 Messenger

private final static Messenger mMessenger = new Messenger(new MHandler());

@Override

public IBinder onBind(Intent intent) {

//提供给客户端Binder

return mMessenger.getBinder();

}

* 配置文件里注册下服务 android:process=”:MService” 创建一个进程

<?xml version="1.0" encoding="utf-8"?>

<manifest xmlns:android="http://schemas.android.com/apk/res/android"

package="com.yw.messengerdemo" >

<application

android:allowBackup="true"

android:icon="@mipmap/ic\_launcher"

android:label="@string/app\_name"

android:supportsRtl="true"

android:theme="@style/AppTheme" >

<activity

android:name=".MainActivity"

android:label="@string/app\_name"

android:theme="@style/AppTheme.NoActionBar" >

<intent-filter>

<action android:name="android.intent.action.MAIN" />

<category android:name="android.intent.category.LAUNCHER" />

</intent-filter>

</activity>

<service android:name=".service.MService"

android:process=":MService" />

</application>

</manifest>

* MainActivity.class 模拟客户端给服务器发消息

package com.yw.messengerdemo;

import android.content.ComponentName;

import android.content.Context;

import android.content.Intent;

import android.content.ServiceConnection;

import android.os.Bundle;

import android.os.Handler;

import android.os.IBinder;

import android.os.Message;

import android.os.Messenger;

import android.os.RemoteException;

import android.support.v7.app.AppCompatActivity;

import android.view.View;

import android.widget.EditText;

import android.widget.TextView;

import com.yw.messengerdemo.service.MService;

public class MainActivity extends AppCompatActivity {

private EditText mEt;

private Messenger mMessenger;

private TextView mTv;

@Override

protected void onCreate(Bundle savedInstanceState) {

super.onCreate(savedInstanceState);

setContentView(R.layout.activity\_main);

mEt = (EditText) findViewById(R.id.et);

mTv = (TextView) findViewById(R.id.tv);

Intent intent = new Intent(this, MService.class);

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

}

public void send\_msg(View view) {

Message message = Message.obtain();

message.what = 0;

message.replyTo = new Messenger(mHandler); //给服务器传一个Messager 为了让服务器用这个Messenger回消息

Bundle b = new Bundle();

String msg = mEt.getText().toString();

b.putString("test", msg);

message.setData(b);

try {

mMessenger.send(message);

} catch (RemoteException e) {

e.printStackTrace();

}

}

//接受服务器的消息

private Handler mHandler = new Handler(new Handler.Callback() {

@Override

public boolean handleMessage(Message msg) {

switch (msg.what) {

case 1:

mTv.setText(msg.getData().getString("msgByService"));

break;

}

return false;

}

});

//创建一个服务连接

private ServiceConnection mServiceConn = new ServiceConnection() {

@Override

public void onServiceConnected(ComponentName name, IBinder service) {

mMessenger = new Messenger(service);

}

@Override

public void onServiceDisconnected(ComponentName name) {

}

};

@Override

protected void onDestroy() {

super.onDestroy();

unbindService(mServiceConn);

}

}

**四．Binder:**

**Aidl文件是系统会完美提供的一种实现binder的工具；**

**服务端：**

**创建一个Service监听客户端的连接请求，在AIDL里申明个接口暴露给客户端，最后在Service中实现这个Aidl接口。**

**客户端：**

**绑定服务端的Service，将服务端返回的Binder对象转为Aidl接口所属的类型，那么就可以调用Aidl中的方法。**

**注意点：**

**aidl中每个实现了接口的类，都必须新建一个同名的aidl文件，并在其中申明为parcelable类型；**

**aidl接口只支持方法，不支持静态常量；**

**五．ContentProvider**

**是系统提供给不同应用之间进行数据共享的方式**

**底层实现是Binder**

**六．Socket：**

**流式套接字（TCP）和用户数据报套接字（UDP）2种；**

**注意事项：**

**java.net.SocketException: Permission denied**

**加入这个权限：**

<uses-permission android:name="android.permission.INTERNET" />

**回归本质 IO流：**

按流的方向分：输入流和输出流。

   按处理数据单位分：字节流和字符流。

   按实现功能分：节点流和处理流。

字节流：一次读入或读出是8位二进制。

   字符流：一次读入或读出是16位二进制。

**CoordinatorLayout熟悉**

**md熟悉**

**NavigationView:**

app:headerLayout="@layout/header "

app:menu="@menu/menu\_drawer"

**Toolbar**:

Toolbar是Android 5.0推出的一个新的导航控件用于取代之前的ActionBar，由于其高度的可定制性、灵活性、具有Material Design风格等优点

兼容5.0以下版本：

compile 'com.android.support:appcompat-v7:25.+'

**隐藏掉之前的**ActionBar、

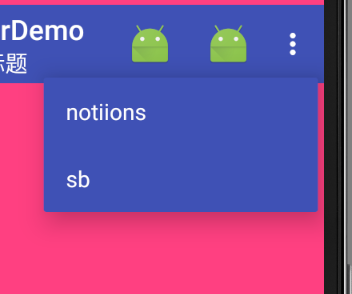
设置Theme.AppCompat.Light.NoActionBar这个主题间好啦。

<style name="Theme.AppCompat.Light.NoActionBar">  
 <item name="windowActionBar">false</item>  
 <item name="windowNoTitle">true</item>  
</style>

**添加菜单：**

**指定弹出菜单的主题**

app:popupTheme="@style/Toolbar.Popup.Theme"

****

<style name="Toolbar.Popup.Theme" parent="@style/ThemeOverlay.AppCompat.Dark">  
 <item name="android:colorBackground">@color/colorPrimary</item>

<item name="android:textColorPrimary">#000000</item>

<item name=" actionOverflowMenuStyle">@style/Toolbar.Popup.Position.Theme</item>  
</style>

<style name="Toolbar.Popup.Position.Theme" parent="Widget.AppCompat.Light.PopupMenu.Overflow">  
 <item name="overlapAnchor">false</item> <!--把该属性改为false即可使menu位置位于toolbar之下-->  
</style>

**注意：**

android:textColorPrimary指定弹出菜单的字体颜色；

android:colorBackground指定弹出菜单的背景颜色；

actionOverflowMenuStyle 设置弹出菜单的位置主题

**杯空 18:21:22**

**Fragmentation解析**