喜马拉雅

c语言指针几个字节?

指针字节和os有关，16位2个字节、32位4个字节、64位8个字节。指针存放的是变量的地址，所以看内存的最大寻址空间（32位地址空间是2^32, 即32个bit）。

Java中long、float字节数?

```
short s; 2字节
int i; 4字节 float f; 4字节
long l; 8字节  double d; 8字节
char c; 2字节（C语言中是1字节）
byte b; 1字节
boolean bool; false/true 1字节
```

JNI 如何调用Java对象和方法?

JNIEnv 与 JavaVM： JavaVM 是 Java虚拟机在 JNI 层的代表, JNI 全局只有一个;JavaVM 在线程中的代表, 每个线程都有一个, JNI 中可能有很多个 JNIEnv。**JNIEnv 不能跨线程**。

JNIEnv在c++中是一个结构体指针，里面包含很多native的函数，用于c++调用Java层的方法，比如findclass，通过静态或者动态注册的方式，可以让c++获取到Java的类路径，从而实现对Java方法和对象的调用。

bitmap占用内存的大小，怎么优化?

图片色彩：ARGB来表示，A表示Alpha，即透明度；R表示red，即红色；G表示green，即绿色；B表示blue，即蓝色。

Bitmap.Config：

Bitmap.Config.ARGB_4444表示：每个像素占16位，A-4，R-4，G-4，B-4, 共4+4+4+4=16位。
Bitmap.Config.RGB_565表示：每个像素占16位，没有透明度，R-5，G-6，B-5，共5+6+5=16位。
Bitmap.Config.ARGB_8888表示：每个像素占32位，A-8，R-8，G-8，B-8,共8+8+8+8=32位。

内存大小计算：height* width* 图像质量每像素占有的字节数

优化点：大图片加载?

压缩：设置合适的缩放比，通过inSampleSize设置图片的缩放比。

```
//创建bitmap工厂的配置参数
    BitmapFactory.Options options = new Options();
```

```
    //返回一个null 没有bitmap    不去真正解析位图 但是能返回图片的一些信息(宽和高)
    options.inJustDecodeBounds = true;
    BitmapFactory.decodeFile("/mnt/sdcard/xxxx.jpg",options);
    //[3]获取图片的宽和高
    int imgWidth = options.outWidth;
    int imgHeight = options.outHeight;
    System.out.println("图片的宽:"+imgWidth+"-----"+imgHeight);

    //[4]计算缩放比
    int scale = 1;  //我们定义的缩放比
    int scalex =  imgWidth/screenWidth;
    int scaley = imgHeight /screenHeight;
    if (scalex >=scaley&&scalex > scale) {
        scale = scalex;
    }
    if (scaley > scalex && scaley>scale) {
        scale = scaley;
    }
    System.out.println("缩放比为:"+scale);

    //[5]按照缩放比显示图片
    options.inSampleSize = scale;

    //[6]开始真正的解析位图
    options.inJustDecodeBounds = false;
    Bitmap bitmap = BitmapFactory.decodeFile("/mnt/sdcard/dog.jpg",options);

    //[7]把bitmap显示到控件上
    iv.setImageBitmap(bitmap);
```

分块加载：在Android中BitmapRegionDecoder类的功能就是加载一张图片的指定区域。

```
// 创建实例
mDecoder = BitmapRegionDecoder.newInstance(mFile.getAbsolutePath(), false);
// 获取原图片宽高
mDecoder.getWidth();
mDecoder.getHeight();
// 加载(10, 10) - (80, 80) 区域内原始精度的Bitmap对象
Rect rect = new Rect(10, 10, 80, 80);
BitmapFactory.Options options = new BitmapFactory.Options();
options.inSampleSize = 1;
Bitmap bitmap = mDecoder.decodeRegion(rect, options);
// 回收释放Native层内存
mDecoder.recycle();
```

利用lru缓存

主流框架原理

okhttp原理?

- 同步异步请求

```
1.生成okhttpclient的对象
2.创建网络请求request和response对象
3.通过接口call来操作请求(只能执行1次)
4.通过realcall的实现具体的网络传输
5.执行完毕会将同步请求从dispatcher中移除
```

- 异步请求

```
1.生成okhttpclient的对象
2.创建网络请求request和response对象
3.通过接口call来操作请求enqueue方法
4.dispatch中定义了3个队列，缓存等待/正在执行的异步队列、正在执行的同步队列。把请求加载队列中，通过线程池进行job的运行。
5.执行拦截器china
```

- 调度器

```
1.维护请求状态
2.维护线程池

//为了实现请求的并发，Dispatcher 配置了一个线程池
public synchronized ExecutorService executorService() {
    if (executorService == null) {
      executorService = new ThreadPoolExecutor(0, Integer.MAX_VALUE, 60,
TimeUnit.SECONDS,
          new SynchronousQueue<Runnable>(), Util.threadFactory("OkHttp
Dispatcher", false));
    }
    return executorService;
  }
```

- SynchronousQueue，为有界队列，且队列大小为 0，不存 Runnable，只是用来进行生产者和消费者之间的传递任务。
- 在 OkHttp 这里，线程池只是个带缓存功能的执行器，而真正的调度是外部包了一个调度策略的。
- 最大并发数 maxRequests 默认为 64，但每个域名最大请求数 maxRequestsPerHost 默认为 5 个。

Retrofit

本质是对http的请求映射到Java的接口上，通过注解的方式来动态代理实现http的请求。

- Retrofit的对象的构建

Retrofit主要是对http请求的封装、response回应信息的封装。Retrofit对象的成员变量有baseurl、并发hashmap（主要封装了注解转换后的对象）、数据适配器等等。

- 动态代理生成http请求对象OkHttpCall，实现真正网络请求

```
public <T> T create(final Class<T> service) {
    Utils.validateServiceInterface(service);
    if (validateEagerly) {
      eagerlyValidateMethods(service);
    }
    return (T) Proxy.newProxyInstance(service.getClassLoader(), new Class<?>
[] { service },
        new InvocationHandler() {
          private final Platform platform = Platform.get();

          @Override public Object invoke(Object proxy, Method method,
@Nullable Object[] args)
              throws Throwable {
            // If the method is a method from Object then defer to normal
invocation.
            if (method.getDeclaringClass() == Object.class) {
              return method.invoke(this, args);
            }
            if (platform.isDefaultMethod(method)) {
              return platform.invokeDefaultMethod(method, service, proxy,
args);
            }
              //构建ServiceMethod，用于包裹http的请求，地址 url header body
content-type的等等
            ServiceMethod<Object, Object> serviceMethod =
                (ServiceMethod<Object, Object>) loadServiceMethod(method);
            OkHttpCall<Object> okHttpCall = new OkHttpCall<>(serviceMethod,
args);
            return serviceMethod.adapt(okHttpCall);
          }
        });
  }
```

- Call对象调用enqueue实际使用的是OkHttp的RealCall的enqueue方法（静态代理的方法），在请求结束还是会通过ConvertFactory进行数据的适配。
- 问题1:Retrofit如何实现多url的配置?

思路：通过okhttp的拦截器特性，可以自定义一个拦截器，获取request对象，在每个网络请求的接口上通过@Headers进行url path和key的绑定，在获取request对象后，通过遍历headers进行base_url的重写，最后获取HttpUrl对象进行重新构建。

Butterknife

注解框架：运行时注解，声明注解的生命周期为runtime，在运行的时候通过反射完成注入。

APT(Annotation Processing Tool)编译时解析技术：

- 注解的生命周期为CLASS,然后继承AbstractProcessor类，主要是在代码编译的时候生成对应activity/fragment对应的viewbinder类。
- Butterknife在代码编译过程中，会扫描注解找到对应的view的id，进行相关绑定事件代码的生成。

- 在调用ButterKnife.bind(this)时执行，该方法会通过反射去实例化对应的$$_ViewBinder类，并且调用该类的bind()方法。

Glide

- 图片加载默认的图片像素格式是ARGB565，从bitmap的内存大小占用比picaso就节约内存。
- 在网络图片进行获取加载的时候会根据viewtarget的大小进行图片的压缩，这样有效降低了oom发生的情况。也是因为picaso进行图片下载直接加载到内存，第一次加载图片可能会比glide要快。
- Glide可以传递activity、fragment进行图片下载和组件的生命周期绑定。
- Glide进行图片的缓存采用了内存缓存+硬盘缓存的方式，进行lru cache的时候会通过url、width、height等10多个参数进行计算key。
- 七牛云的url会动态更新的导致glide缓存失效，

```
GlideApp.with(this).load("http://goo.gl/gEgYUd").into(imageView);
```

工程实践CI/CD

Jenkins、git 、Robot、Phabricator

- Robot FrameWork部署到CI

安装Jenkins，启动Jenkins，安装Jenkins的[Robot Framework plugin](http://wiki.jenkins-ci.org/display/JENKINS/Robot+Framework+Plugin)插件，新建一个job，进行配置即可。

- Appium（用于手机自动化测试框架）

安装jdk，下载Appium，配置环境变量，运行appium-doctor检测命令。

- Phabricator（用于代码检测、提交审计的、代码diff的开源软件苦）

https://pan.baidu.com/s/1dFiAaM9

数据结构的理解

hashmap

Jdk1.7版本：数组+链表

HashMap里面实现一个静态内部类Node，其重要的属性有 key , value, next，从属性key,value我们就能很明显的看出来Entry就是HashMap键值对实现的一个基础bean。

- Hash 算法（散列表）决定集合元素的存储位置，这样可以保证能快速存、取集合元素；
- 底层数据结构是数组+链表，通过key的hashcode计算哈希值，然后确定对象在数组的中的位置；
- 当发生hash冲突的时候（不同的key得到一样的哈希值），则将对象存储在链表中，新插入的对象放在链表的头部；

- 在元素查找的时候，最坏情况下，是需要遍历完所有的元素才能找到对应的对象，在jdk1.8里面，如果链表长度过于大，通过红黑树进行优化查找速度。

日志收集系统设计思路

需求？日志级别（error、warn、info）、日志回捞（客户端主动上报（时机）、服务器的主动回捞数据）、安全的考虑（日志存储位置、上传的时候的时候进行加密处理）、日志的读写性能（file、mmap）

- 基于 mmap 的特性，即使用户强杀了进程，日志文件也不会丢失，并且会在下次初始化 Log4a 的时候回写到日志文件中。
- mmap函数完成内存的映射和访问，相对于普通的IO操作，更提升性能。
- 常规文件操作为了提高读写效率和保护磁盘，使用了页缓存机制，这是由OS控制的。这样造成读文件时需要先将文件页从磁盘拷贝到页缓存中，由于页缓存处在内核空间，不能被用户进程直接寻址，所以还需要将页缓存中数据页再次拷贝到内存对应的用户空间中。这样，通过了两次数据拷贝过程，才能完成进程对文件内容的获取任务。写操作也是一样，待写入的buffer在内核空间不能直接访问，必须要先拷贝至内核空间对应的主存，再写回磁盘中（延迟写回），也是需要两次数据拷贝。

  而使用mmap操作文件中，由于不需要经过内核空间的数据缓存，只使用一次数据拷贝，就从磁盘中将数据传入内存的用户空间中，供进程使用。 mmap的关键点是实现了用户空间和内核空间的数据直接交互而省去了空间不同数据不通的繁琐过程。因此mmap效率更高。
- 当进程被强杀、内存不足的时候会进行数据的回写。

项目组织架构搭建会怎么做

- 业务逻辑和ui进行隔离
- 架构模式 mvvm 引入google的jetpack架构组件，viewmodel+livedata

鱼泡泡题目：

1.音频录制的步骤 **MediaRecorder**、**MediaCodec**用过吗?

2.TextureView, SurfaceView和GLSurfaceView 的区别？项目中如何取舍

3.视频数据帧获取bitmap从哪里来？如何被消费

4.Frame Buffer项目中用到吗？FBO?

5.进程之间通信的方式

6.activity的启动流程源码分析

7.jetpack的使用怎么用

8.Android优化（电量、内存、卡顿等）

9.mvvm 的理解

10.mmap 机制作用？一个进程通过获取另外一个进程的指针，可以通过指针来获取里面的内容吗?

音视频相关

SurfaceView、TextureView、GLSurfaceview

SurfaceView解决视频播放的问题、采用单独的线程来渲染ui，防止界面过于复杂大致16ms加载卡顿问题。SurfaceView和view的主要区别：

- View适用于主动更新的情况，而SurfaceView则适用于被动更新的情况，比如频繁刷新界面。
- View在主线程中对页面进行刷新，而SurfaceView则开启一个子线程来对页面进行刷新。
- View在绘图时没有实现双缓冲机制，SurfaceView在底层机制中就实现了双缓冲机制。
- **Surface不在View hierachy中，它的显示也不受View的属性控制，所以不能进行平移，缩放等变换，也不能放在其它ViewGroup中，一些View中的特性也无法使用。**

> 双缓冲技术：双缓冲技术是把要处理的图片在内存中处理好之后，再将其显示在屏幕上。双缓冲主要是为了解决 反复局部刷屏带来的闪烁。把要画的东西先画到一个内存区域里，然后整体的一次性画出来。

SurfaceView怎么使用?

```
public class SurfaceViewTemplate extends SurfaceView implements
SurfaceHolder.Callback, Runnable {
    private SurfaceHolder mSurfaceHolder;
    //绘图的Canvas
    private Canvas mCanvas;
    //子线程标志位
    private boolean mIsDrawing;
    public SurfaceViewTemplate(Context context) {
        this(context, null);
    }

    public SurfaceViewTemplate(Context context, AttributeSet attrs) {
        this(context, attrs, 0);
    }

    public SurfaceViewTemplate(Context context, AttributeSet attrs, int
defStyleAttr) {
        super(context, attrs, defStyleAttr);
        initView();
    }

    @Override
    public void surfaceCreated(SurfaceHolder holder) {
        mIsDrawing = true;
        //开启子线程
        new Thread(this).start();
    }

    @Override
    public void surfaceChanged(SurfaceHolder holder, int format, int width,
int height) {
```

```
    }

    @Override
    public void surfaceDestroyed(SurfaceHolder holder) {
        mIsDrawing = false;
    }

    @Override
    public void run() {
        while (mIsDrawing){
            drawSomething();
        }
    }
    //绘图逻辑
    private void drawSomething() {
        try {
            //获得canvas对象
            mCanvas = mSurfaceHolder.lockCanvas();
            //绘制背景
            mCanvas.drawColor(Color.WHITE);
            //绘图
        }catch (Exception e){

        }finally {
            if (mCanvas != null){
                //释放canvas对象并提交画布
                mSurfaceHolder.unlockCanvasAndPost(mCanvas);
            }
        }
    }

    /**
     * 初始化View
     */
    private void initView(){
        mSurfaceHolder = getHolder();
        mSurfaceHolder.addCallback(this);
        setFocusable(true);
        setKeepScreenOn(true);
        setFocusableInTouchMode(true);
    }
}
```

GLSurfaceview: 是 Surfaceview 的子类，它主要是在 SurfaceView 的基础上实现了一个 GLThread（EGLContext创建GL环境所在线程即为GL线程），绘制的工作直接通过OpenGL来进行，绘制的内容默认情况下依旧是绘制到SurfaceView所提供的Surface上。

TextureView：

- 与SurfaceView一样继承View，它可以将内容流直接投影到View中。和SurfaceView不一样它不会在**WMS中单独创建窗口，而是作为View hierachy中的一个普通View，因此可以和其它普通View一样进行移动，旋转，缩放，动画等变化。**

- TextureView必须在硬件加速的窗口中。TextureView重载了draw()方法，其中主要SurfaceTexture中收到的图像数据作为纹理更新到对应的HardwareLayer中。

实时滤镜

采用opengl es2.0 + TextureView方案来做。TextureView充当布局view，使用SurfaceTexture来获取预览图像数据，TextureView内置的SurfaceTexture用来配合EGL来将图像显示到屏幕上，自定义一个SurfaceTexture来接收Camera的预览图像来做二次处理（黑白滤镜）。

Android架构组件

livedata

livedata是一个可以保存数据和观察数据的组件，可以感知Activity/Fragment的生命周期，确保数据的更新是在组件的active状态周期执行。livedata可以和任何的集合数据一起使用，通常在viewmodel中使用。

- 创建livedata

```
public class NameViewModel extends ViewModel {

// 创建一个包含String的LiveData
private MutableLiveData<String> mCurrentName;

public MutableLiveData<String> getCurrentName() {
      if (mCurrentName == null) {
          mCurrentName = new MutableLiveData<String>();
      }
        return mCurrentName;
    }
}
```

- 使用livedata

在viewmodel中创建livedata，创建Observer对象并实现回掉接口onchange（），将Observer对象使用observer()函数attach到LiveData对象。

```
public class NameActivity extends AppCompatActivity {

    private NameViewModel mModel;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);

        // Other code to setup the activity...

        // Get the ViewModel.
```

```
        mModel = ViewModelProviders.of(this).get(NameViewModel.class);

        // Create the observer which updates the UI.
        final Observer<String> nameObserver = new Observer<String>() {
            @Override
            public void onChanged(@Nullable final String newName) {
                // Update the UI, in this case, a TextView.
                mNameTextView.setText(newName);
            }
        };

        // Observe the LiveData, passing in this activity as the
LifecycleOwner and the observer.
        mModel.getCurrentName().observe(this, nameObserver);
    }
}
```

viewmodel

ViewModel类旨在以一种有生命周期的方式存储和管理与UI相关的数据。 ViewModel类允许数据在屏幕旋转等配置变化后存活。

- 创建viewmodel

```
public class MyViewModel extends ViewModel {
private MutableLiveData<List<User>> users;
public LiveData<List<User>> getUsers() {
    if (users == null) {
        users = new MutableLiveData<List<Users>>();
        loadUsers();
    }
    return users;
}
private void loadUsers() {
    // Do an asyncronous operation to fetch users.
}}
```

- activity中去使用

```
public class MyActivity extends AppCompatActivity {
   public void onCreate(Bundle savedInstanceState) {
   // Create a ViewModel the first time the system calls an activity's
onCreate() method.
   // Re-created activities receive the same MyViewModel instance created by
the first activity.

   MyViewModel model = ViewModelProviders.of(this).get(MyViewModel.class);
   model.getUsers().observe(this, users -> {
       // update UI
   });
   }}
```

Java基础

集合相关

ArrayList分析：

- 内部是数组组成的数据结构，在初始化对象的时候回默认一个长度为10的数组集合。
- ArrayList内部实现是数组，且当数组长度不够时，数组的会进行原数组长度的**1.5倍扩容**。
- ArrayList内部元素是**可以重复的**。且**有序**的，因为是按照数组一个一个进行添加的。
- ArrayList是**线程不安全的**，因为其内部添加、删除、等操作，没有进行同步操作。
- ArrayList**增删元素速度较慢**，因为内部实现是数组，每次操作都会对数组进行复制操作，复制操作是比较耗时的（System.arraycopy）

LinkedList分析：

- LinkedList方法内部实现是双向链表，且内部有fist与last指针控制数据的增加与删除等操作
- LinkedList内部元素是可以重复，且有序的。因为是按照链表进行存储元素的。
- LinkedList线程是非线程安全的，因为其内部添加、删除、等操作，没有进行同步操作。
- LinkedList增删元素速度较快。

HashMap：

- HashMap是数组+链表+红黑树（JDK1.8增加了红黑树部分）实现的底层数据结构。
- Node是HashMap的一个内部类，实现了Map.Entry接口，本质是就是一个映射(键值对)。
- HashMap是线程非安全的，建议使用ConcurrentHashMap提升并发性能。

HashTable：Hashtable是遗留类，很多映射的常用功能与HashMap类似，不同的是它承自Dictionary类，并且是线程安全的，任一时间只有一个线程能写Hashtable，并发性不如ConcurrentHashMap。

LinkedHashMap分析

HashSet分析

LinkedHashSet分析

ArrayMap、SparseMap、与HashMap的对比

ConcurrentHashMap分析

数据结构

- android如何实现环形缓冲区

Android基础

activity

生命周期:

Android 系统是通过 Activity 的栈来对 activtiy 进行管理， Activity 有四种状态，Active/Paused/Stopped/Killed。

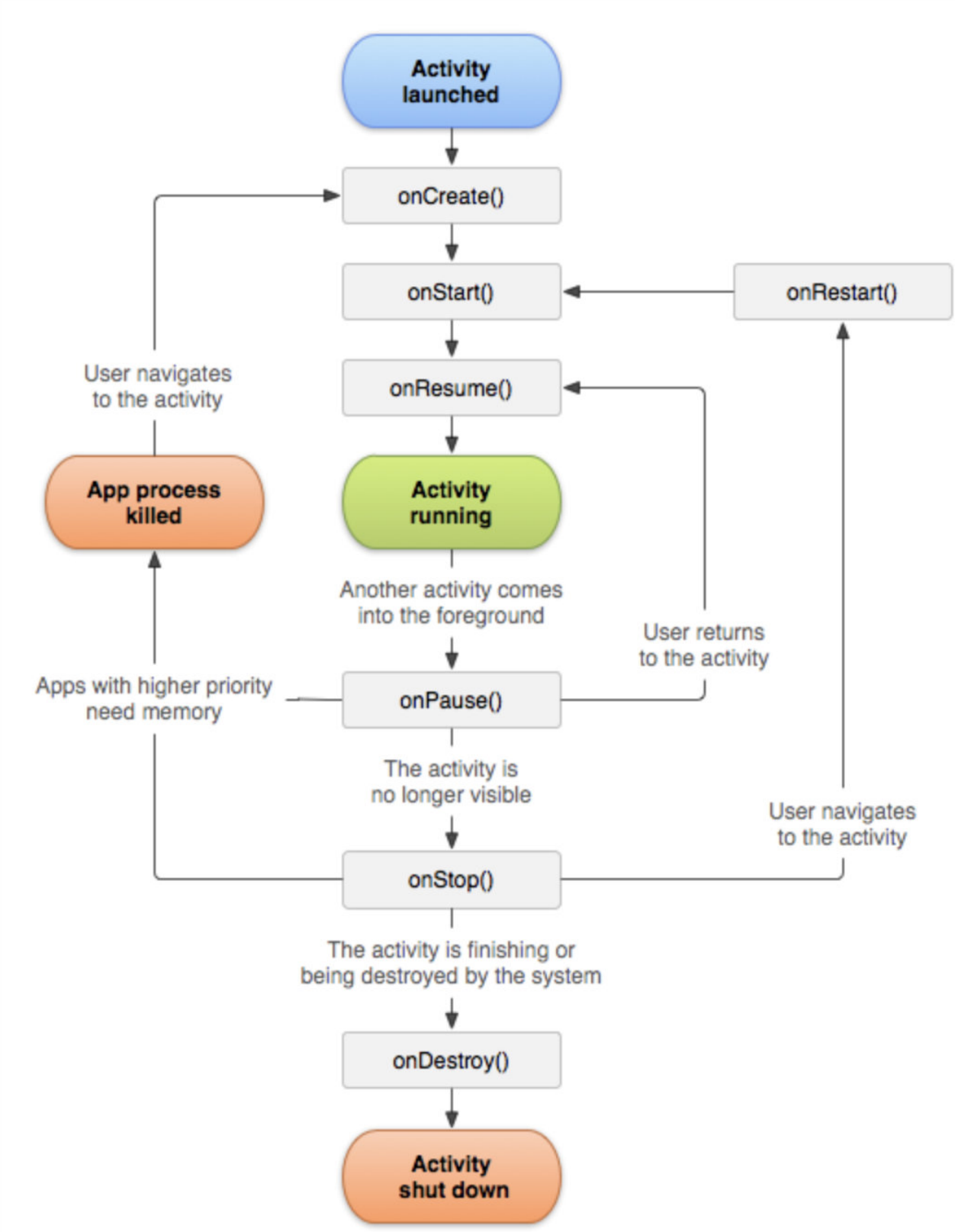


图 1. Activity 生命周期。

异常情况下的生命周期：

有些设备配置可能会在运行时发生变化（例如屏幕方向、键盘可用性及语言）。 发生这种变化时，Android 会重启正在运行的 `Activity`（先后调用 `onDestroy()` 和 `onCreate()`）。重启行为旨在通过利用与新设备配置匹配的备用资源自动重新加载您的应用，来帮助它适应新配置。在 Activity 生命周期中，Android 会在销毁 Activity 之前调用 `onSaveInstanceState()`，以便您保存有关应用状态的数据。 然后，您可以在 `onCreate()` 或 `onRestoreInstanceState()` 期间恢复 Activity 状态。

启动模式：singletop（栈顶复用，会调用onnewIntent（）） singleinstance（全局唯一，整个系统只有一个，只能有一个任务栈，比如来电界面） singletask（通过taskaffinity查找是否有存在的独立栈，没有则开启一个独立的栈，如果没有配置taskaffinity，会在当前app的activtiy的栈空间找是否存在当前的activtiy，有则启动，并且清除这个activity上面的所有的activity） standard

组件通信：activity intent bundle

activity 通过bundle传递数据给fragment 或者定一个一个public方法，在fragment的onAttach方法中进行强转进行获取。

fragment传递数据给activtiy：eventbus 或者 接口回掉（fragment定义接口，activity实现这个接口，onAttach（）方法中获取activity的对象，onDetach()对象设为空，防止内存泄漏）

activty和service之间的通信：

- 利用绑定服务向service传递数据
- 通过startService 传递intent 在service onStartCommand（）中获取intent
- 通过callback+handler进行传递

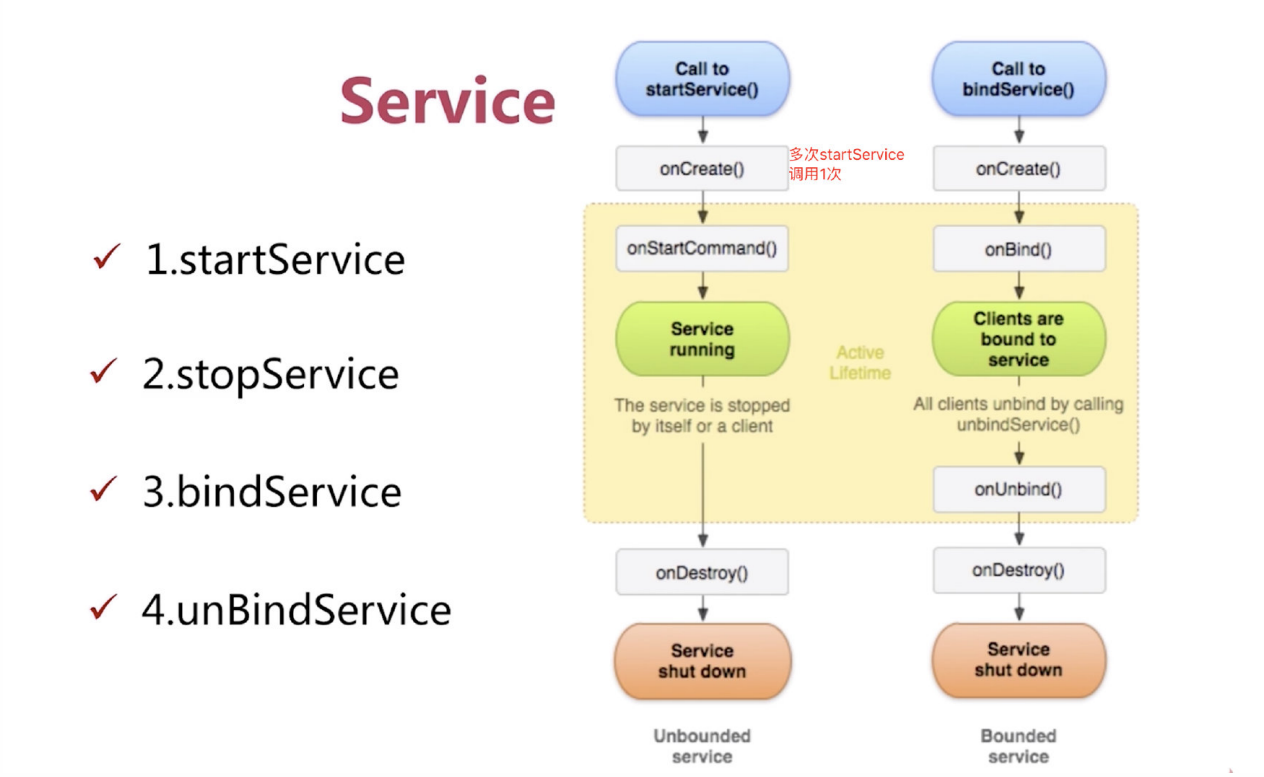
activity的启动源码：

service

thread：程序执行的最小单元。在activity的线程无法做精确的控制。

service：是android的一种机制，运行在安卓的主线程。在后台执行相关的job。

生命周期：

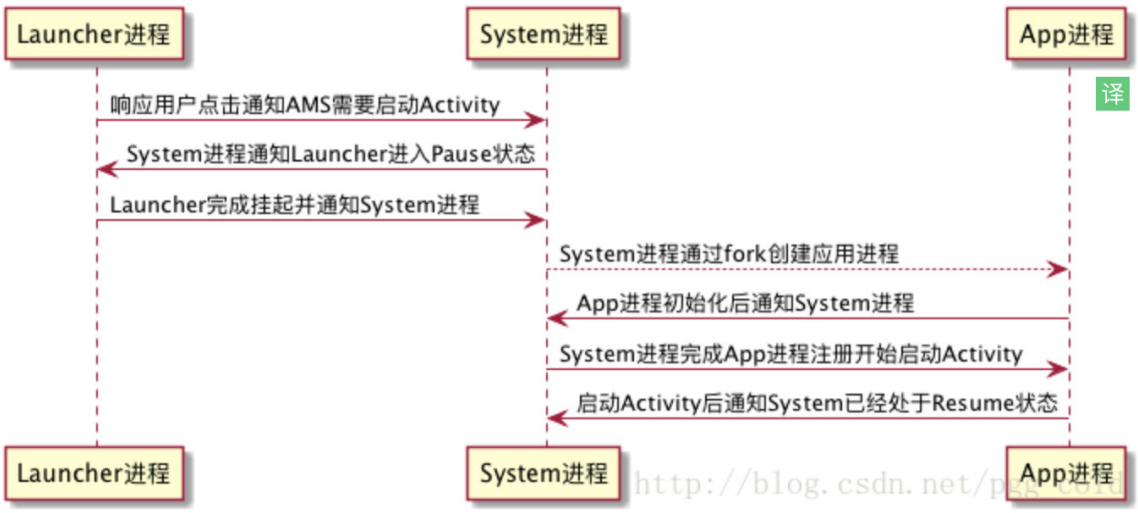


IntentService：可以执行耗时操作，耗时任务执行完成后自主关闭service，内部是开启一个handlerthread，来执行异步任务，通过handler、looper来完成消息的发送到handlerthread执行操作。

启动服务和绑定服务：

- 启动服务优先级高
- 服务在托管进程的的主线程中运行（ui）

activity的启动流程

- 

startActivityForResult在FragmentActivity和Fragment中的异同

1. Fragment和FragmentActivity都能接收到自己的发起的请求所返回的结果

   > 那当然，就是这么设计的。

2. FragmentActivity发起的请求，Fragment完全接收不到结果

   > 被FragmentActivity拦截了，没有转发到Fragment。

3. Fragment发起的请求，虽然在FragmentActivity中能获取到结果，但是requestCode完全对应不上

   > 如果是Fragment发起的请求，那么在FragmentActivity.onActivityResult获取到的requestCode，其实是经过映射之后一个的大于0xffff的值，已经不是最初Fragment发请求时的requestCode了。

view的绘制流程

ActivityThread: 创建activty的时候，会拿到activity对应的window对象，通过window对象创建decorView，并addvie到window。所有的绘制流程从ViewRootImpl的performTraversals（）开始。

测量始于DecorView，通过不断的遍历子View的measure方法，根据ViewGroup的MeasureSpec及子View的LayoutParams来决定子View的MeasureSpec，进一步获取子View的测量宽高，然后逐层返回，不断保存ViewGroup的测量宽高。

MeasureSpec：UNSPECIFIED、EXACTYLY（match_parent、精确值）、AT_MOST（wrap_content）

- **对于一个直接继承自View的自定义View来说，它的wrap_content和match_parent属性的效果是一样的**，因此如果要实现自定义View的wrap_content，则要重写onMeasure方法，对wrap_content属性进行处理。

- ViewGroup类提供了measureChild，measureChild和measureChildWithMargins方法，简化了父子View的尺寸计算。

layout：

- measure操作完成后得到的是对每个View经测量过的measuredWidth和measuredHeight，layout操作完成之后得到的是对每个View进行位置分配后的mLeft、mTop、mRight、mBottom，这些值都是相对于父View来说的。
- 使用View的getWidth()和getHeight()方法来获取View测量的宽高，必须保证这两个方法在onLayout流程之后被调用才能返回有效值。

ondraw：6个部分

```
1.draw the background, if needed
2.对View的内容进行绘制。调用ondraw ()
3.dispatchDraw () 绘制子view
4.滚动条的绘制等等
```

invalidate和postInvalidate：在UI线程和非UI线程进行view树的绘制。

requestLayout()方法：会调用measure过程和layout过程，不会调用draw过程。

事件分发机制

1.事件的传递是从viewgrouop到view，从上到下。viewgroup onTouchEvent（消费）、onInterceptTouchEvent（拦截事件）、dispatchTouchEvent（分发）。

2事件处理是从view 到viewgroup。从下到上。

3.当点击事件产生后，事件会传递给当前的Activity，由Activity中的PhoneWindow完成，PhoneWindow再把事件处理工作交给DecorView，之后再有DecorView将事件处理工作交给ViewGroup。

4.传递优先级：onTouchListener.onTouch > onTouchEvent > onClickListener.onClick。

异步消息

handler

handler是用来进行异步消息处理的。可以用于线程之间的消息传递，包含MessageQueue（Message）、Looper、Handler四大组件。handler可以发送和处理消息。当handle发送一个消息时候，message会被添加到消息队列中，looper组件用于从队列中取出消息，交给handler进行消息处理。

- 在子线程中创建handler对象一定要调用Looper.prepare()方法。获取当前线程的looper对象。looper中通过threadlocal确保每个线程获取的looper是不一样的。通过调用Looper.loop()来开启消息队列的循环。

Android存储

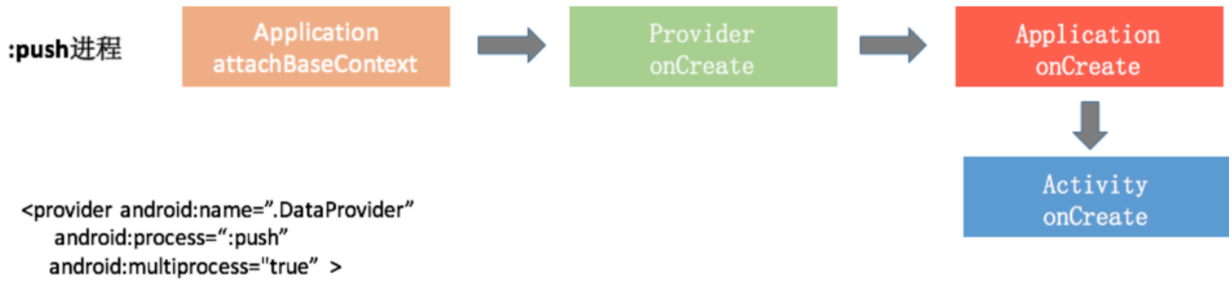
SP

- 跨进程不安全。由于没有使用跨进程的锁。
- 加载缓慢。SharedPreferences 文件的加载使用异步线程，而且加载线程并没有设置线程优先级。
- 全量写入。无论是调用 commit() 还是 apply()，即使我们只改动其中的一个条目，都会把整个内容全部写到文件。
- 容易导致anr，程序在崩溃时候或者其他的异常情况发生，Android系统会调用apply（）把数据异步保存到磁盘。

ContentProvider

提供跨进程的数据操作访问，可以利用cp访问系统的电话通讯录、音乐信息等等。也可以在程序中自定义自己的cp，给其他的程序进行访问。

- ContentProvider 的生命周期默认在 Application onCreate() 之前，而且都是在主线程创建的。我们自定义的 ContentProvider 类的构造函数、静态代码块、onCreate 函数都尽量不要做耗时的操作，会拖慢启动速度。



-

Android设计类题目

图片加载框架

图片压缩 主要是考考虑图片的压缩比列samplesize

```
public int calculateInSampleSize(BitmapFactory.Options options,
            int reqWidth, int reqHeight) {

        final int width = options.outWidth;
        final int height = options.outHeight;

        int inSampleSize = 1;
        if (height > reqHeight || width > reqWidth) {
            final int halfHeight = height / 2;
            final int halfWidth = width / 2;
            while ((halfHeight / inSampleSize) > reqHeight
                    && (halfWidth / inSampleSize) > halfWidth) {
```

```
                inSampleSize *= 2;
            }
        }
        return inSampleSize;
    }
```

图片缓存： LruCache 完成内存的缓存、DiskLruCache完成磁盘的缓存。

缓存大小计算：最大可用内存的1/8 当前对象的大小重写sizeOf（）

```
public class BitmapLruCache extends LruCache<String, Bitmap> {
    //设置缓存大小，建议当前应用可用最大内存的八分之一 即(int)
(Runtime.getRuntime().maxMemory() / 1024 / 8)
    public BitmapLruCache(int size) {
        super(size);
    }

    //计算当前节点的内存大小 这个方法需要重写 不然返回1
    @Override
    protected int sizeOf(String key, Bitmap value) {
        return value.getByteCount() / 1024;
    }

    //当节点移除时该方法会回调，可根据需求来决定是否重写该方法
    @Override
    protected void entryRemoved(boolean evicted, String key, Bitmap oldValue,
Bitmap
            newValue) {
        super.entryRemoved(evicted, key, oldValue, newValue);
    }
}
```

LruCache：内部实现了linkedhashmap，构造hashmap的时候使用访问排序（accessOrder），可以让访问到的元素自动插入到linked的尾部。

DiskLruCache:通过file文件流来创建缓存文件，通过读写文件流的方来确定缓存文件，上次bitmap通过文件描述符来构造对象。

提供同步加载方法，或者异步加载方法给外界调用，通过handle来判断是否ui线程等等。