Android 知识架构总结 by Jake

1. 数据结构

1.1 数组/队列

1.2 单链表

1.3 双链表

1.4 Hash 表

1.5 树

1.6 图

2. 算法

2.1 搜索

2.2 排序

> 各种排序：冒泡、选择、插入、希尔、归并、快排、堆排、桶排、基数的原理、平均时间复杂度、最坏时间复杂度、空间复杂度、是否稳定。

3. Java基础知识与JVM的理解

1. 线程
2. 内部类
3. 泛型

4. 设计模式

4.1 关于复杂度

时间复杂度

> 如何计算时间复杂度

> 常见复杂度的计算时间级别

空间复杂度

> 如何计算控件复杂度

4.2 单例模式

懒汉式

饿汉式

完整式

内部类的形式

4.3 Builder 模式

4.4 Adapter 模式

4.5 工厂模式

4.6 观察者模式

4.7 装饰者模式

5. Android 知识体系与基础知识

5.1 掌握Android自带的组件与类

5.1.1 四大组件之Activity

- **理解Activity**

  activity是独立平等的，用来处理用户操作。几乎所有的activity都是用来和用户交互的，所以activity类会创建了一个窗口，开发者可以通过setContentView(View)的接口把UI放到给窗口上。

- **重要方法**

  1. onConfigurationChanged

     前面说过，当 Activity 横竖屏切换的时候会导致 Activity 销毁并重建，哪有什么方法能避免呢？其实可以在 AndroidManifest 里面指定 android:configChanges="orientation/screenSize" 来避免重建，这时就会调用 onConfigurationChanged 方法。

     如果按上面的配置，当字体发生变化时，也会销毁重建，但是不会回调 onConfigurationChanged 方法，所以说想要监听的变化必须要包含之内。

```java
@Override
    public void onConfigurationChanged(Configuration newConfig) {
        super.onConfigurationChanged(newConfig);
        if (newConfig.orientation == Configuration.ORIENTATION_PORTRAIT) {
            //竖屏
        } else {
            //横屏
        }
    }
```

2. onTrimMemory

   当内存紧张时会回调，它在 onStop 回调之前。指导应用程序在不同的情况下进行自身的内存释放，以避免被系统直接杀掉，提高应用程序的用户体验。它和 onLowMemory 相比，它有一个 level 评级，onLowMemory 能兼容更低的版本。

```java
@Override
public void onTrimMemory(int level) {
    super.onTrimMemory(level);
    if (level == TRIM_MEMORY_UI_HIDDEN) {

    }
}
```

- **Activity 的生命周期**

  正常状态:

  onCreate()—>onStart()—>onResume()-> onPause()—>onStop()

  后台返回前台:

  onRestart()—>**onStart()**—>onResume()

  锁屏:

  onPause()->onStop()

  解锁:

  onStart()->onResume()

- **Activity 的启动模式 任务栈**

  使用`android:launchMode="standard|singleInstance|singleTask|singleTop"`来控制Acivity任务栈。

  任务栈是一种后进先出的结构。位于栈顶的Activity处于焦点状态,当按下back按钮的时候,栈内的Activity会一个一个的出栈,并且调用其`onDestory()`方法。如果栈内没有Activity,那么系统就会回收这个栈,每个APP默认只有一个栈,以APP的包名来命名.

  1. standard : 标准模式,每次启动Activity都会创建一个新的Activity实例,并且将其压入任务栈栈顶,而不管这个Activity是否已经存在。Activity的启动三回调(*onCreate()->onStart()->onResume()*)都会执行。

2. singleTop : 栈顶复用模式.这种模式下,如果新Activity已经位于任务栈的栈顶,那么此Activity不会被重新创建,所以它的启动三回调就不会执行,同时Activity的`onNewIntent()`方法会被回调.如果Activity已经存在但是不在栈顶,那么作用与*standard模式*一样.
3. singleTask: 栈内复用模式.创建这样的Activity的时候,系统会先确认它所需任务栈已经创建,否则先创建任务栈.然后放入Activity,如果栈中已经有一个Activity实例,那么这个Activity就会被调到栈顶,`onNewIntent()`,并且singleTask会清理在当前Activity上面的所有Activity.(clear top)
4. singleInstance : 加强版的singleTask模式,这种模式的Activity只能单独位于一个任务栈内,由于栈内复用的特性,后续请求均不会创建新的Activity,除非这个独特的任务栈被系统销毁了

Activity的堆栈管理以ActivityRecord为单位,所有的ActivityRecord都放在一个List里面.可以认为一个ActivityRecord就是一个Activity栈

- **Activity 的数据保存 和 恢复**
  - onSaveInstanceState

    在activity　可能被回收之前调用,用来保存自己的状态和信息，以便回收后重建时恢复数据（在onCreate()或onRestoreInstanceState()中恢复）。旋转屏幕重建activity会调用该方法，但其他情况在onpause()和onStop()状态的activity不一定会调用, 官方说明:

    > One example of when onPause and onStop is called and not this method is when a user navigates back from activity B to activity A: there is no need to call onSaveInstanceState on B because that particular instance will never be restored, so the system avoids calling it. An example when onPause is called and not onSaveInstanceState is when activity B is launched in front of activity A: the system may avoid calling onSaveInstanceState on activity A if it isn't killed during the lifetime of B since the state of the user interface of A will stay intact.

    也就是说，系统灵活的来决定调不调用该方法，**但是如果要调用就一定发生在onStop方法之前，但并不保证发生在onPause的前面还是后面**。
  - onRestoreInstanceState

    这个方法在onStart 和 onPostCreate之间调用，在onCreate中也可以状态恢复，但有时候需要所有布局初始化完成后再恢复状态。

    onPostCreate：一般不实现这个方法，当程序的代码开始运行时，它调用系统做最后的初始化工作。
  - 使用

```
public class MainActivity extends Activity {

@Override
protected void onCreate(Bundle savedInstanceState) {
    if(savedInstanceState!=null){ //判断是否有以前的保存状态信息
         savedInstanceState.get("Key");
         }
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_main);
}
 @Override
protected void onSaveInstanceState(Bundle outState) {
// TODO Auto-generated method stub
 //可能被回收内存前保存状态和信息,
   Bundle data = new Bundle();
```

```
   data.putString("key", "last words before be kill");
   outState.putAll(data);
super.onSaveInstanceState(outState);
}
 @Override
protected void onRestoreInstanceState(Bundle savedInstanceState) {
// TODO Auto-generated method stub
   if(savedInstanceState!=null){ //判断是否有以前的保存状态信息
         savedInstanceState.get("Key");
         }
super.onRestoreInstanceState(savedInstanceState);
}
}
```

- **Activity 在manifest 标签中的一些注意点**
  - taskAffinity

    在 singleTask 启动模式中，多次提到某个 Activity 所需的任务栈，什么是 Activity 所需要的任务栈呢？这就要从一个参数说起：taskAffinity，任务相关性。这个参数标识了一个 Activity 所需要的任务栈的名字，默认情况下，所有 Activity 所需的任务栈的名字为应用的包名。当然，我们可以为每个 Activity 都单独指定 taskAffinity 属性，这个属性值必须不能和包名相同，否则相当于没有设置。taskAffinity 属性主要和 singleTask 启动模式和 allowTaskReparentiong 属性配对使用，在其他情况下没有意义。

    taskAffinity 与 singleTask 配对使用：

    如果启动了设置了这两个属性的 Activity，这个 Activity 就会在 taskAffinity 设置的任务栈中。

    taskAffinity 与 allowTaskReparenting 配对使用：

    当一个应用 A 启动了应用 B 的某个 Activity 后，如果这个 Activity 的 allowTaskReparenting 属性为 true 的话，那么当应用 B 被启动后，此 Activity 会直接从应用 A 的任务栈转移到应用 B 的任务栈中。这个属性主要作用就是将这个 Activity 转移到它所属的任务栈中，例如一个短信应用收到一个带有网络链接的短信，点击链接会跳到浏览器，这时候如果 allowTaskReparenting 设置为 true 的话，打开浏览器应用就会直接显示刚才打开的网页页面，而打开短信应用后这个浏览器界面就会消失。

    启动模式了解之后，那是如何指定启动模式的方式呢?

    有两种：一种是在 AndroidMenifet 文件设置 launchMode 属性，一种是给 Intent 设置 Flag。

    如果两者都存在，后者优先级更高。

- **Scheme跳转协议**

  概念

  Android中的scheme是一种页面内跳转协议，是一种非常好的实现机制，通过定义自己的scheme协议，可以非常方便跳转app中的各个页面；通过scheme协议，服务器可以定制化告诉app跳转哪个页面，可以通过通知栏消息定制化跳转页面，可以通过H5页面跳转页面等。

  应用场景

  - 通过服务器下发跳转路径跳转相应页面

- 通过在H5页面的锚点跳转相应的页面
- 根据服务器下发通知栏消息，App跳转相应的页面（包括另外一个APP的页面，作为推广使用）

5.1.2 四大组件之Service

- 生命周期
- **startService 和 bindService的区别:**

* startService特点：

1. 使用这种start方式启动的Service的生命周期如下：onCreate()--->onStartCommand()（onStart()方法已过时） ---> onDestroy()

2. 如果服务已经开启，不会重复的执行onCreate()， 而是会调用onStart()和onStartCommand()

3. 一旦服务开启跟调用者(开启者)就没有任何关系了。

4. 开启者退出了，开启者挂了，服务还在后台长期的运行。

5.开启者不能调用服务里面的方法。

* bindService特点：

1. 绑定服务不会调用onStart()或者onStartCommand()方法

2. bind的方式开启服务，绑定服务。调用者调用unbindService解除绑定，服务也会跟着销毁。

3. 绑定者可以调用服务里面的方法。

- **onStartCommand 不同返回值 的作用是什么?**
  - START_STICKY 在运行onStartCommand后service进程被kill后，那将保留在开始状态，但是不保留那些传入的intent。不久后service就会再次尝试重新创建，因为保留在开始状态，在创建 service后将保证调用 onstartCommand。如果没有传递任何开始命令给service，那将获取到null的intent。
  - START_NOT_STICKY 在运行onStartCommand后service进程被kill后，并且没有新的intent传递给它。Service将移出开始状态，并且直到新的明显的方法（startService）调用才重新创建。因为如果没有传递任何未决定的intent那么service是不会启动，也就是期间onstartCommand不会接收到任何null的intent。
  - START_REDELIVER_INTENT 在运行onStartCommand后service进程被kill后，系统将会再次启动service，并传入最后一个intent给onstartCommand。直到调用stopSelf(int)才停止传递intent。如果在被kill后还有未处理好的intent，那被kill后服务还是会自动启动。因此onstartCommand不会接收到任何null的intent。

- **使用startForgroundService 的方法**
  1. 申请 FOREGROUND_SERVICE 权限，它是普通权限
  2. 在 onStartCommand 中必须要调用 startForeground 构造一个通知栏，不然 ANR
  3. 前台服务只能是启动服务，不能是绑定服务

- **bug**

```
android.app.RemoteServiceException
```

使用前台服务，必须提供一个通知栏，不然五秒就会 ANR。

```
  public int onStartCommand(Intent intent, int flags, int startId) {
        Log.i(TAG, "onStartCommand: ");
        NotificationManager mNotificationManager = (NotificationManager)
getSystemService(Context.NOTIFICATION_SERVICE);
        // 通知渠道的id
        String id = "my_channel_01";
        // 用户可以看到的通知渠道的名字.
        CharSequence name = "Demo";
        // 用户可以看到的通知渠道的描述
        String description = "Desc";
        int importance = NotificationManager.IMPORTANCE_HIGH;
        NotificationChannel mChannel = new NotificationChannel(id, name,
importance);
        // 配置通知渠道的属性
        mChannel.setDescription(description);
        // 设置通知出现时的闪灯（如果 android 设备支持的话）
        mChannel.enableLights(true);
        mChannel.setLightColor(Color.RED);
        // 设置通知出现时的震动（如果 android 设备支持的话）
        mChannel.enableVibration(true);
        mChannel.setVibrationPattern(new long[]{100, 200, 300, 400, 500, 400,
300, 200, 400});
        mNotificationManager.createNotificationChannel(mChannel);

        // 通知渠道的id
        String CHANNEL_ID = "my_channel_01";
        // Create a notification and set the notification channel.
        Notification notification = new Notification.Builder(this, CHANNEL_ID)
                .setContentTitle("New Message").setContentText("You've received
new messages.")
                .setSmallIcon(R.drawable.ic_launcher_foreground)
                .build();
        startForeground(1, notification);
        return super.onStartCommand(intent, flags, startId);
    }
```

> Service 的种类

5.1.3 四大组件之Broadcast

- 作用与原理

-

  原理描述

  1. 广播接收者 通过 Binder机制在 AMS 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，寻找合适的广播接收者
     （寻找依据：IntentFilter / Permission）
  4. AMS将广播发送到合适的广播接收者相应的消息循环队列中
  5. 广播接收者通过 消息循环 拿到此广播，并回调 onReceive()

  特别注意：广播发送者 和 广播接收者的执行 是 *异步的，即 广播发送者 不会关心有无接收者接收 & 也不确定接收者何时才能接收到

- 种类
  - 普通广播（Normal Broadcast）
    - 通过sendBroadcast进行发送，如果注册了Action匹配的接受者则会收到
    - 若发送广播有相应权限，那么广播接收者也需要相应权限
  - 系统广播（System Broadcast）
    - Android中内置了多个系统广播：只要涉及到手机的基本操作（如开机、网络状态变化、拍照等等），都会发出相应的广播
    - 每个广播都有特定的Intent - Filter（包括具体的action）
    - 系统广播由系统发送，不需要手动发送，只需要注册监听
  - 有序广播（Ordered Broadcast）
    - 通过sendOrderedBroadcast发送
    - 发送出去的广播被广播接收者按照先后顺序接收（有序是针对广播接收者而言的）
    - 广播接受者接收广播的顺序规则：Priority大的优先；动态注册的接收者优先
    - 先接收的可以对广播进行截断和修改
  - App应用内广播（本地广播、Local Broadcast）
    - 通过LocalBroadcastManager.getInstance(this).sendBroadcastSync();
    - App应用内广播可理解为一种局部广播，广播的发送者和接收者都同属于一个App
    - 相比于全局广播（普通广播），App应用内广播优势体现在：安全性高 & 效率高（本地广播只会在APP内传播，安全性高；不允许其他APP对自己的APP发送广播，效率高）
  - 粘性广播（Sticky Broadcast）
    - 在Android5.0 & API 21中已经失效，所以不建议使用
    - 通过sendStickyBroadcast发送
    - 粘性广播在发送后就一直存在于系统的消息容器里面，等待对应的处理器去处理，如果暂时没有处理器处理这个广播则一直在消息容器里面处于等待状态
    - 粘性广播的Receiver如果被销毁，那么下次重新创建的时候会自动接收到消息数据
- 注意点与面试点

> **本地广播的使用以及实现机制**

  - 基本使用：可以通过intent.setPackage(packageName)指定包名，也可以使用localBroadcastManager（常用），示例代码如下：

```
//注册应用内广播接收器
//步骤1：实例化BroadcastReceiver子类 & IntentFilter mBroadcastReceiver
mBroadcastReceiver = new mBroadcastReceiver();
IntentFilter intentFilter = new IntentFilter();

//步骤2：实例化LocalBroadcastManager的实例
localBroadcastManager = LocalBroadcastManager.getInstance(this);

//步骤3：设置接收广播的类型
intentFilter.addAction(android.net.conn.CONNECTIVITY_CHANGE);

//步骤4：调用LocalBroadcastManager单一实例的registerReceiver（）方法进行动态注册
localBroadcastManager.registerReceiver(mBroadcastReceiver, intentFilter);

//取消注册应用内广播接收器
localBroadcastManager.unregisterReceiver(mBroadcastReceiver);

//发送应用内广播
Intent intent = new Intent();
intent.setAction(BROADCAST_ACTION);
localBroadcastManager.sendBroadcast(intent);
```

- localBroadcastManager的实现机制
  1. LocalBroadcastManager高效的原因主要是因为它内部是通过Handler实现的，它的sendBroadcast()方法含义和我们平时所用的全局广播不一样，它的sendBroadcast()方法其实是通过handler发送一个Message实现的。
  2. 既然是它内部是通过Handler来实现广播的发送的，那么相比与系统广播通过Binder实现那肯定是更高效了，同时使用Handler来实现，别的应用无法向我们的应用发送该广播，而我们应用内发送的广播也不会离开我们的应用
  3. LocalBroadcastManager内部协作主要是靠这两个Map集合：mReceivers和mActions，当然还有一个List集合mPendingBroadcasts，这个主要就是存储待接收的广播对象

5.1.4 四大组件之ContentProvider

- **概述**

  ContextProvider 为存储和获取数据提供了统一的接口，可以在不同的应用程序之间安全的共享数据。它允许把自己的应用数据根据需求开放给其他应用进行增删改查。数据的存储方式还是之前的方式，它只是提供了一个统一的接口去访问数据。

- **统一资源标识符**

  统一资源标识符即 URI，用来唯一标识 ContentProvider 其中的数据，外界进程通过 URI 找到对应的 ContentProvider 其中的数据，在进行数据操作。

  URI 分为系统预置和自定义，分别对应系统内置的数据（如通讯录等）和自定义数据库。

- **系统内置 URI**

  比如获取通讯录信息所需要的 URI：ContactsContract.CommonDataKinds.Phone.CONTENT_URI。

- **自定义 URI**

```
格式:content://authority/path/id
authority:授权信息，用以区分不同的 ContentProvider
path:表名，用以区分 ContentProvider 中不同的数据表
id: ID号，用以区别表中的不同数据
示例:content://com.example.omooo.demoproject/User/1
上述 URI 指向的资源是：名为 com.example.omooo.demoproject 的 ContentProvider 中表名为 User 中 id 为 1 的数据。
```

注意，URI 也存在匹配通配符：* & #

- **MIME 数据类型**

  它是用来指定某个扩展名的文件用某种应用程序来打开。

  可以通过 ContentProvider.getType(uri) 来获得。

  每种 MIME 类型由两部分组成：类型 + 子类型。

  示例：text/html、application/pdf

- **ContentProvider的使用**
  - 组织数据方式

    ContentProvider 主要以表格的形式组织数据，同时也支持文件数据，只是表格形式用的比较多，每个表格中包含多张表，每张表包含行和列，分别对应数据。
  - 主要方法

```
public class MyProvider extends ContentProvider {

    @Override
    public boolean onCreate() {
        return false;
    }

    @Override
    public Cursor query(Uri uri, String[] projection, String selection, String[] selectionArgs, String sortOrder) {
        return null;
    }

    @Override
    public String getType(Uri uri) {
        return null;
    }

    @Override
    public Uri insert(Uri uri, ContentValues values) {
        return null;
    }

    @Override
```

```
    public int delete(Uri uri, String selection, String[] selectionArgs) {
        return 0;
    }

    @Override
    public int update(Uri uri, ContentValues values, String selection, String[]
selectionArgs) {
        return 0;
    }
}
```

- **辅助工具类**
  - ContentResolver

    统一管理不同的 ContentProvider 间的操作。 1. 即通过 URI 即可操作不同的 ContentProvider 中的数据 2. 外部进程通过 ContentResolver 类从而与 ContentProvider 类进行交互

    一般来说，一款应用要使用多个 ContentProvider，若需要了解每个 ContentProvider 的不同实现从而在完成数据交互，操作成本高且难度大，所以在 ContentProvider 类上多加一个 ContentResolver 类对所有的 ContentProvider 进行统一管理。

    ContentResolver 类提供了与 ContentProvider 类相同名字和作用的四个方法：

```
// 外部进程向 ContentProvider 中添加数据
public Uri insert(Uri uri, ContentValues values)

// 外部进程 删除 ContentProvider 中的数据
public int delete(Uri uri, String selection, String[] selectionArgs)
// 外部进程更新 ContentProvider 中的数据
public int update(Uri uri, ContentValues values, String selection, String[]
selectionArgs)

// 外部应用 获取 ContentProvider 中的数据
public Cursor query(Uri uri, String[] projection, String selection, String[]
selectionArgs, String sortOrder)
// 使用ContentResolver前，需要先获取ContentResolver
// 可通过在所有继承Context的类中 通过调用getContentResolver()来获得ContentResolver
ContentResolver resolver =  getContentResolver();

// 设置ContentProvider的URI
Uri uri = Uri.parse("content://cn.scu.myprovider/user");

// 根据URI 操作 ContentProvider中的数据
// 此处是获取ContentProvider中 user表的所有记录
Cursor cursor = resolver.query(uri, null, null, null, "userid desc");
```

  - ContentUris 用来操作 URI 的，常用有两个方法：

```
// withAppendedId（）作用：向URI追加一个id
Uri uri = Uri.parse("content://cn.scu.myprovider/user")
Uri resultUri = ContentUris.withAppendedId(uri, 7);
// 最终生成后的Uri为：content://cn.scu.myprovider/user/7
// parseId（）作用：从URL中获取ID
Uri uri = Uri.parse("content://cn.scu.myprovider/user/7")
long personid = ContentUris.parseId(uri);
//获取的结果为:7
```

- UriMatcher 在 ContentProvider 中注册 URI，根据 URI 匹配 ContentProvider 中对应的数据表。

```
// 步骤1：初始化UriMatcher对象
UriMatcher matcher = new UriMatcher(UriMatcher.NO_MATCH);
//常量UriMatcher.NO_MATCH  = 不匹配任何路径的返回码
// 即初始化时不匹配任何东西

// 步骤2：在ContentProvider 中注册URI（addURI（））
int URI_CODE_a = 1;
int URI_CODE_b = 2;
matcher.addURI("cn.scu.myprovider", "user1", URI_CODE_a);
matcher.addURI("cn.scu.myprovider", "user2", URI_CODE_b);
// 若URI资源路径 = content://cn.scu.myprovider/user1 ，则返回注册码URI_CODE_a
// 若URI资源路径 = content://cn.scu.myprovider/user2 ，则返回注册码URI_CODE_b

// 步骤3：根据URI 匹配 URI_CODE，从而匹配ContentProvider中相应的资源（match（））
@Override
  public String getType(Uri uri) {
    Uri uri = Uri.parse(" content://cn.scu.myprovider/user1");

    switch(matcher.match(uri)){
   // 根据URI匹配的返回码是URI_CODE_a
   // 即matcher.match(uri) == URI_CODE_a
    case URI_CODE_a:
      return tableNameUser1;
      // 如果根据URI匹配的返回码是URI_CODE_a，则返回ContentProvider中的名为
tableNameUser1的表
    case URI_CODE_b:
      return tableNameUser2;
      // 如果根据URI匹配的返回码是URI_CODE_b，则返回ContentProvider中的名为
tableNameUser2的表
  }
}
```

- ContentObserver 内容观察者，当 ContentProvider 中的数据发生变化时，就会触发 ContentObserver 类。

```
      // 步骤1：注册内容观察者ContentObserver
      getContentResolver().registerContentObserver（uri）;
      // 通过ContentResolver类进行注册，并指定需要观察的URI

      // 步骤2：当该URI的ContentProvider数据发生变化时，通知外界（即访问该
ContentProvider数据的访问者）
```

```
      public class UserContentProvider extends ContentProvider {
        public Uri insert(Uri uri, ContentValues values) {
        db.insert("user", "userid", values);
        getContext().getContentResolver().notifyChange(uri, null);
        // 通知访问者
      }
   }
    // 步骤3: 解除观察者
    getContentResolver().unregisterContentObserver（uri）;
    // 同样需要通过ContentResolver类进行解除
```

- **实例**

  1. **获取通讯录信息**

  这里就不需要自己写 ContentProvider 的实现了，用系统已经给的 URI。

```
    /**
     * 获取通讯录信息
     */
    private void getContactsInfo() {
        Cursor cursor = getContentResolver().query(
                ContactsContract.CommonDataKinds.Phone.CONTENT_URI, null, null,
null, null
        );
        if (cursor != null) {
            while (cursor.moveToNext()) {
                //联系人姓名
                String name =
cursor.getString(cursor.getColumnIndex(ContactsContract.CommonDataKinds.Phone.DIS
PLAY_NAME));
                //联系人手机号
                String phoneNumber =
cursor.getString(cursor.getColumnIndex(ContactsContract.CommonDataKinds.Phone.NUM
BER));
                Log.i(TAG, "getContactsInfo: name: " + name + "  phone: " +
phoneNumber);
            }
            cursor.close();
        }
    }
```

  2. **结合 SQLite**

     1. 创建数据库
     2. 自定义 ContentProvider 并注册
     3. 进程内访问数据

        a. 创建数据库：

```
```

```
  public class MySQLiteOpenHelper extends SQLiteOpenHelper {

      public MySQLiteOpenHelper(Context context) {
          super(context, "user.info", null, 1);
      }

      @Override
      public void onCreate(SQLiteDatabase db) {
          db.setPageSize(1024 * 4);
  //        db.enableWriteAheadLogging();
          db.execSQL("CREATE TABLE if not exists user (name text, age string)");
      }

      @Override
      public void onUpgrade(SQLiteDatabase db, int oldVersion, int newVersion) {

      }
  }
```

b. 自定义 ContentProvider 并注册：

```
      public class MyProvider extends ContentProvider {
          private static UriMatcher mUriMatcher;

          static {
              mUriMatcher = new UriMatcher(UriMatcher.NO_MATCH);
              mUriMatcher.addURI("com.example.omooo.demoproject.provider",
"user", 1);
          }

          private MySQLiteOpenHelper mMySQLiteOpenHelper;
          private SQLiteDatabase mSQLiteDatabase;
          private Context mContext;

          @Override
          public boolean onCreate() {
              mContext = getContext();
              mMySQLiteOpenHelper = new MySQLiteOpenHelper(mContext);
              mSQLiteDatabase = mMySQLiteOpenHelper.getWritableDatabase();
              mSQLiteDatabase.execSQL("insert into user values('Omooo','18');");
              return true;
          }

          @Nullable
          @Override
          public Cursor query(@NonNull Uri uri, @Nullable String[] projection,
@Nullable String selection, @Nullable String[] selectionArgs, @Nullable String
sortOrder) {
              return mSQLiteDatabase.query("user", projection, selection,
selectionArgs, null, null, sortOrder, null);
          }

          @Nullable
```

```
        @Override
        public String getType(@NonNull Uri uri) {
            return null;
        }

        @Nullable
        @Override
        public Uri insert(@NonNull Uri uri, @Nullable ContentValues values) {
            mSQLiteDatabase.insert("user", null, values);
            mContext.getContentResolver().notifyChange(uri, null);
            return uri;
        }

        @Override
        public int delete(@NonNull Uri uri, @Nullable String selection,
@Nullable String[] selectionArgs) {
            return 0;
        }

        @Override
        public int update(@NonNull Uri uri, @Nullable ContentValues values,
@Nullable String selection, @Nullable String[] selectionArgs) {
            return 0;
        }

    }
        <provider
                android:exported="false"
                android:authorities="com.example.omooo.demoproject.provider"
                android:name=".provider.MyProvider"/>
```

c. 进程内访问数据：

```
    private void insertTable() {
        Uri uri =
Uri.parse("content://com.example.omooo.demoproject.provider/user");
        ContentValues values = new ContentValues();
        values.put("name", "Test");
        values.put("age", "21");
        ContentResolver resolver = getContentResolver();
        resolver.insert(uri, values);
        Cursor cursor = resolver.query(uri, new String[]{"name", "age"}, null,
null, null);
        if (cursor != null) {
            while (cursor.moveToNext()) {
                String name = cursor.getString(cursor.getColumnIndex("name"));
                String age = cursor.getString(cursor.getColumnIndex("age"));
                Log.i(TAG, "insertTable: name: " + name + " age: " + age);
            }
            cursor.close();
        }
    }
```

- 注意点与面试点

5.1.5 常用组件之Fragment

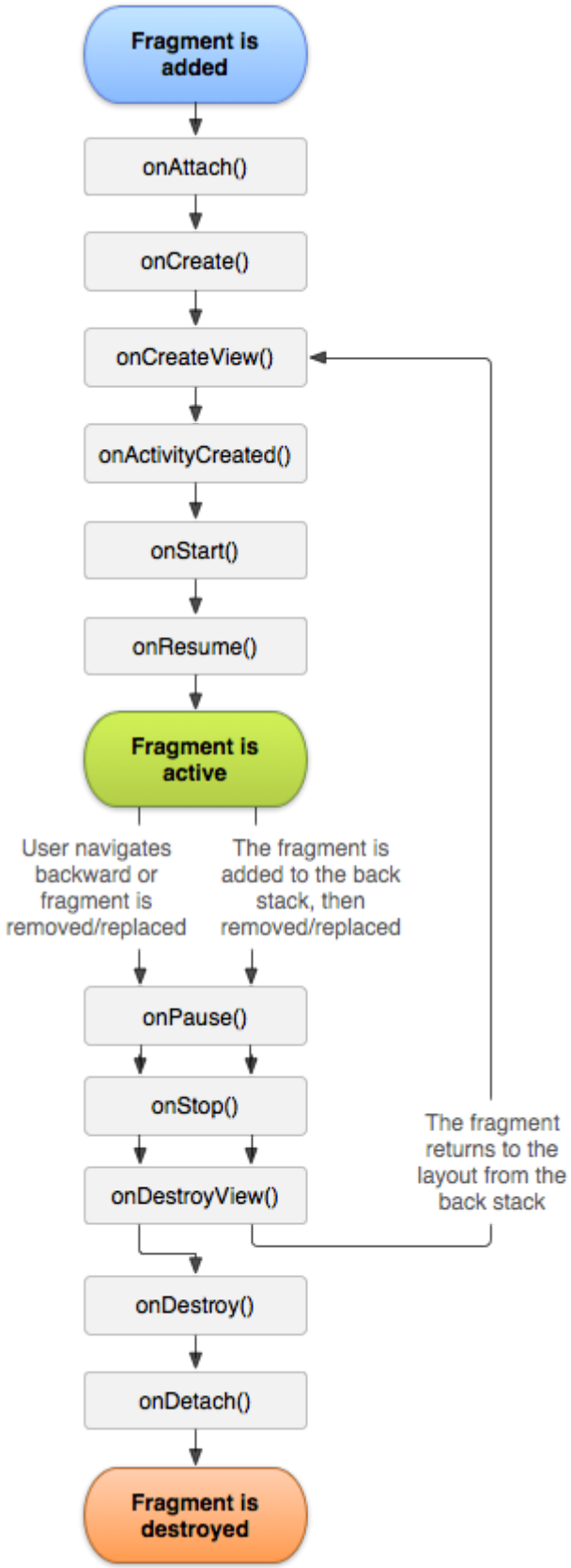
1. 什么是Fragment

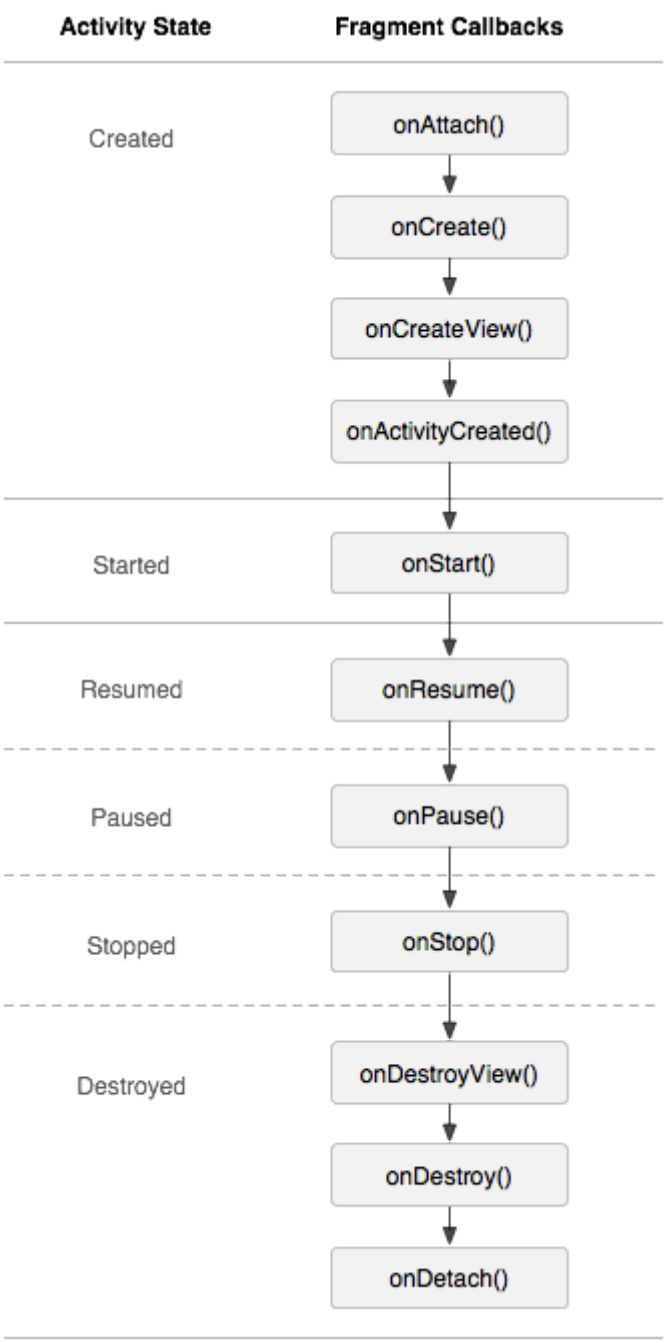
Fragment，俗称碎片，自Android 3.0开始被引进并大量使用。作为Activity界面的一部分，Fragment的存在必须依附于Activity，并且与Activity一样，拥有自己的生命周期，同时处理用户的交互动作。同一个Activity可以有一个或多个Fragment作为界面内容，并且可以动态添加、删除Fragment，灵活控制UI内容，也可以用来解决部分屏幕适配问题。

2. Fragment为什么被称为第五大组件

首先Fragment的使用次数是不输于其他四大组件的，而且Fragment有自己的生命周期，比Activity更加节省内存，切换模式也更加舒适，使用频率不低于四大组件。

3. Fragment的生命周期


Fragment is added
onAttach()
onCreate()
onCreateView()
onActivityCreated()
onStart()
onResume()
Fragment is active
User navigates backward or fragment is removed/replaced
The fragment is added to the back stack, then removed/replaced
onPause()
onStop()
onDestroyView()
The fragment returns to the layout from the back stack
onDestroy()
onDetach()
Fragment is destroyed



4. Fragment创建/加载到Activity的两种方式

- 静态加载
    1. 创建Fragment的xml布局文件
    2. 在Fragment的onCreateView中inflate布局，返回
    3. 在Activity的布局文件中的适当位置添加fragment标签，指定name为Fragment的完整类名（这时候Activity中可以直接通过findViewById找到Fragment中的控件）
- 动态加载（需要用到事务操作，常用）
    1. 创建Fragment的xml布局文件
    2. 在Fragment的onCreateView中inflate布局，返回

```
        @Nullable
        @Override
        public View onCreateView(@NonNull LayoutInflater inflater, @Nullable
ViewGroup container, @Nullable Bundle savedInstanceState) {
            return inflater.inflate(R.layout.activity_main, container, false);
        }
```

3. 初始化Fragment
4. 进行add()/remove()/replace()/attach()/detach()/hide()/addToBackStack()事务操作（都是对Fragment的栈进行操作，其中add()指定的tag参数可以方便以后通过findFragmentByTag()找到这个Fragment）
5. 提交事务：commit() 示例代码：

```
  @Override
  protected void onCreate(Bundle savedInstanceState) {
      super.onCreate(savedInstanceState);
      setContentView(R.layout.activity_main);
      getSupportFragmentManager().beginTransaction()
              .add(R.id.fragment_container, new TestFragment(), "test")
              .commit();
      TestFragment f = (TestFragment)
getSupportFragmentManager().findFragmentByTag("test");
  }
```

5. Fragment通信问题

1. 通过findFragmentByTag或者getActivity获得对方的引用（强转）之后，再相互调用对方的public方法。

   优点：简单粗暴 缺点：引入了“强转”的丑陋代码，另外两个类之间各自持有对方的强引用，耦合较大，容易造成内存泄漏

2. 通过Bundle的方法进行传值，在添加Fragment的时候进行通信

```
@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_main);
    Fragment fragment = new TestFragment();
    Bundle bundle = new Bundle();
    bundle.putString("key", "value");
    //Activity中对fragment设置一些参数
    fragment.setArguments(bundle);
    getSupportFragmentManager().beginTransaction()
            .add(R.id.fragment_container, fragment, "test")
            .commit();
}
```

   优点：简单粗暴 缺点：只能在Fragment添加到Activity的时候才能使用，属于单向通信

3. 利用eventbus进行通信

   优点：实时性高，双向通信，Activity与Fragment之间可以完全解耦 缺点：反射影响性能，无法获取返回数据，EventBUS难以维护

4. 利用接口回调进行通信（Google官方推荐）

```java
   //MainActivity实现MainFragment开放的接口
   public class MainActivity extends FragmentActivity implements FragmentListener
{
        @override
        public void toH5Page() {
            //...其他处理代码省略
        }
   }
   //Fragment的实现
   public class MainFragment extends Fragment {
        //接口的实例，在onAttach Activity的时候进行设置
        public FragmentListener mListener;
        //MainFragment开放的接口
        public static interface FragmentListener {
            //跳到h5页面
            void toH5Page();
        }
        @Override
        public void onAttach(Activity activity) {
            super.onAttach(activity);
            //对传递进来的Activity进行接口转换
            if (activity instance FragmentListener){
                mListener = ((FragmentListener) activity);
            }
        }
        // ...其他处理代码省略
   }
```

优点：既能达到复用，又能达到很好的可维护性，并且性能得到保证

缺点：假如项目很大了，Activity与Fragment的数量也会增加，这时候为每对Activity与Fragment交互定义交互接口就是一个很麻烦的问题（包括为接口的命名，新定义的接口相应的Activity还得实现，相应的Fragment还得进行强制转换）

5. 通过Handler进行通信（其实就是把接口的方式改为Handler）

   优点：既能达到复用，又能达到很好的可维护性，并且性能得到保证 缺点：Fragment对具体的Activity存在耦合，不利于Fragment复用和维护，没法获取Activity的返回数据

6. 通过广播/本地广播进行通信

   优点：简单粗暴 缺点：大材小用，存在性能损耗，传播数据必须实现序列化接口

7. 父子Fragment之间通信，可以使用getParentFragment()/getChildFragmentManager()的方式进行

6. FragmentPageAdapter和FragmentPageStateAdapter的区别

- FragmentPageAdapter在每次切换页面的时候，是将Fragment进行分离，适合页面较少的Fragment使用以保存一些内存，对系统内存不会多大影响

```
@Override
public void destroyItem(ViewGroup container, int position, Object object) {
    if (mCurTransaction == null) {
        mCurTransaction = mFragmentManager.beginTransaction();
    }
    if (DEBUG) Log.v(TAG, "Detaching item #" + getItemId(position) + ": f=" +
object
            + " v=" + ((Fragment)object).getView());
    //FragmentPageAdapter在destroyItem的时候调用detach
    mCurTransaction.detach((Fragment)object);
}
```

- FragmentPageStateAdapter在每次切换页面的时候，是将Fragment进行回收，适合页面较多的Fragment使用，这样就不会消耗更多的内存

```
@Override
public void destroyItem(ViewGroup container, int position, Object object) {
    Fragment fragment = (Fragment) object;
    if (mCurTransaction == null) {
        mCurTransaction = mFragmentManager.beginTransaction();
    }
    if (DEBUG) Log.v(TAG, "Removing item #" + position + ": f=" + object
            + " v=" + ((Fragment)object).getView());
    while (mSavedState.size() <= position) {
        mSavedState.add(null);
    }
    mSavedState.set(position, fragment.isAdded()
            ? mFragmentManager.saveFragmentInstanceState(fragment) : null);
    mFragments.set(position, null);
    //FragmentPageStateAdapter在destroyItem的时候调用remove
    mCurTransaction.remove(fragment);
}
```

7. 参考文章

Android：Activity与Fragment通信(99%)完美解决方案

5.1.6 常用组件之SharedPreference

5.1.7 常用组件之Intent

Intent知识点

IntentFilter知识点

IntentFilter 的三个属性:

Action URL Category

IntentFilter 的匹配规则

- 加载所有的Intent Filter列表
- 去掉action匹配失败的Intent Filter
- 去掉url匹配失败的Intent Filter
- 去掉Category匹配失败的Intent Filter　　判断剩下的Intent Filter数目是否为0。如果为0查找失败返回异常；如果大于0，就按优先级排序，返回最高优先级的Intent Filter

> 如何在短信中启动一个Activity

5.1.8 常用组件之Drawable

5.1.9 常用组件之Handler

- 作用与原理
- 使用方法
- 注意点与面试点

5.1.10 常用组件之AndroidManifests.mxl

- 注意点与面试点

5.1.11 常用组件之Animation

- 种类
- 使用方法
- 注意点与面试点

5.1.12 常用组件之布局文件

> 常用ViewGroup 布局

- LinearLayout
- RelativeLayout
- FrameLayout
- ConstaintLayout

  https://www.jianshu.com/p/a74557359882

5.1.13 常用组件之Bitmap

5.1.14 重要组件

- **WebView**

  WebView详解
- **RecyclerView**

  RecyclerView详解
  - 相关问题:

> RecyclerView 与 ListView 的区别

RecyclerView 相比 ListView，有一些明显的优点：

1. 默认已经实现了 View 的复用，而且复用机制更加完善
2. 支持局部刷新
3. Item 添加、删除动画，Item 实现拖拽、侧滑删除等
4. 更灵活的 LayoutManager

当然，ListView 相比 RecyclerView 也有优点：

1. addHeaderView、addFooterView 添加头视图、尾视图
2. 通过 android:divider 设置自定义分割线
3. setOnItemClickListener、setOnItemLongClickListener 设置点击事件以及长按事件

- Viewpager

  ViewPager详解

5.2 理解Android深层的类与原理

5.2.1 Binder的理解

在Android 的底层研究中, 我们会经常用到binder 的方式进行进程间通信, 那binder到底是什么? binder 的实现方式是什么样的呢?

> **Binder 是什么?**

首先Binder是Android系统进程间通信(IPC)方式之一。

Binder使用Client－Server通信方式。Binder框架定义了四个角色：Server,Client,ServiceManager以及Binder驱动。其中Server,Client,ServiceManager运行于用户空间，驱动运行于内核空间。Binder驱动程序提供设备文件/dev/binder与用户空间交互，Client、Server和Service Manager通过open和ioctl文件操作函数与Binder驱动程序进行通信。

Server创建了Binder实体，为其取一个字符形式，可读易记的名字，将这个Binder连同名字以数据包的形式通过Binder驱动发送给ServiceManager，通知ServiceManager注册一个名字为XX的Binder，它位于Server中。驱动为这个穿过进程边界的Binder创建位于内核中的实体结点以及ServiceManager对实体的引用，将名字以及新建的引用打包给ServiceManager。ServiceManager收数据包后，从中取出名字和引用填入一张查找表中。但是一个Server若向ServiceManager注册自己Binder就必须通过0这个引用和ServiceManager的Binder通信。Server向ServiceManager注册了Binder实体及其名字后，Client就可以通过名字获得该Binder的引用了。Clent也利用保留的0号引用向ServiceManager请求访问某个Binder：我申请名字叫XX的Binder的引用。ServiceManager收到这个连接请求，从请求数据包里获得Binder的名字，在查找表里找到该名字对应的条目，从条目中取出Binder引用，将该引用作为回复发送给发起请求的Client。

当然，不是所有的Binder都需要注册给ServiceManager广而告之的。Server端可以通过已经建立的Binder连接将创建的Binder实体传给Client，当然这条已经建立的Binder连接必须是通过实名Binder实现。由于这个Binder没有向ServiceManager注册名字，所以是匿名Binder。Client将会收到这个匿名Binder的引用，通过这个引用向位于Server中的实体发送请求。匿名Binder为通信双方建立一条私密通道，只要Server没有把匿名Binder发给别的进程，别的进程就无法通过穷举或猜测等任何方式获得该Binder的引用，向该Binder发送请求。

> **Binder 的运行机制是什么样的呢?**

Linux内核实际上没有从一个用户空间到另一个用户空间直接拷贝的函数，需要先用copy_from_user()拷贝到内核空间，再用copy_to_user()拷贝到另一个用户空间。为了实现用户空间到用户空间的拷贝，mmap()分配的内存除了映射进了接收方进程里，还映射进了内核空间。所以调用copy_from_user()将数据拷贝进内核空间也相当于拷贝进了接收方的用户空间，这就是Binder只需一次拷贝的‘秘密’。

最底层的是Android的ashmen(Anonymous shared memory)机制，它负责辅助实现内存的分配，以及跨进程所需要的内存共享。AIDL(android interface definition language)对Binder的使用进行了封装，可以让开发者方便的进行方法的远程调用，后面会详细介绍。Intent是最高一层的抽象，方便开发者进行常用的跨进程调用。

从英文字面上意思看，Binder具有粘结剂的意思那么它是把什么东西粘接在一起呢？在Android系统的Binder机制中，由一系统组件组成，分别是Client、Server、Service Manager和Binder驱动，其中Client、Server、Service Manager运行在用户空间，Binder驱动程序运行内核空间。Binder就是一种把这四个组件粘合在一起的粘连剂了，其中，核心组件便是Binder驱动程序了，ServiceManager提供了辅助管理的功能，Client和Server正是Binder驱动和ServiceManager提供的基础设施上，进行Client-Server之间的通信。

1. Client、Server和ServiceManager实现在用户空间中，Binder驱动实现在内核空间中
2. Binder驱动程序和ServiceManager在Android平台中已经实现，开发者只需要在用户空间实现自己的Client和Server
3. Binder驱动程序提供设备文件/dev/binder与用户空间交互，Client、Server和ServiceManager通过open和ioctl文件操作函数与Binder驱动程序进行通信
4. Client和Server之间的进程间通信通过Binder驱动程序间接实现
5. ServiceManager是一个守护进程，用来管理Server，并向Client提供查询Server接口的能力

服务器端：一个Binder服务器就是一个Binder类的对象。当创建一个Binder对象后，内部就会开启一个线程，这个线程用户接收binder驱动发送的消息，收到消息后，会执行相关的服务代码。

Binder驱动：当服务端成功创建一个Binder对象后，Binder驱动也会相应创建一个mRemote对象，该对象的类型也是Binder类，客户就可以借助这个mRemote对象来访问远程服务。

客户端：客户端要想访问Binder的远程服务，就必须获取远程服务的Binder对象在binder驱动层对应的binder驱动层对应的mRemote引用。当获取到mRemote对象的引用后，就可以调用相应Binde对象的服务了。

在这里我们可以看到，客户是通过Binder驱动来调用服务端的相关服务。首先，在服务端创建一个Binder对象，接着客户端通过获取Binder驱动中Binder对象的引用来调用服务端的服务。在Binder机制中正是借着Binder驱动将不同进程间的组件bind(粘连)在一起，实现通信。

mmap将一个文件或者其他对象映射进内存。文件被映射进内存。文件被映射到多个页上，如果文件的大小不是所有页的大小之和，最后一个页不被使用的空间将会凋零。munmap执行相反的操作，删除特定地址区域的对象映射。

当使用mmap映射文件到进程后，就可以直接操作这段虚拟地址进行文件的读写等操作，不必再调用read,write等系统调用。但需注意，直接对该段内存写时不会写入超过当前文件大小的内容。

使用共享内存通信的一个显而易见的好处是效率高，因为进程可以直接读写内存，而不需要任何数据的拷贝。对于像管道和消息队列等通信方式，则需要在内核和用户空间进行四次的数据拷贝，而共享内存则只拷贝两次内存数据：一次从输入文件到共享内存区，另一次从共享内存到输出文件。实际上，进程之间在共享内存时，并不总是读写少量数据后就解除映射，有新的通信时，再重新建立共享内存区域，而是保持共享区域，直到通信完成为止，这样，数据内容一直保存在共享内存中，并没有写回文件。共享内存中的内容往往是在解除内存映射时才写回文件的。因此，采用共享内存的通信方式效率是非常高的。

aidl主要就帮助了我们完成了包装数据和解包的过程，并调用了transact过程，而用来传递的数据包我们就称为parcel

AIDL:xxx.aidl -> xxx.java ,注册service

1. 用aidl定义需要被调用方法接口
2. 实现这些方法
3. 调用这些方法

5.2.2 Context

5.2.3 IPC

5.3 Android 的View

5.3.1 Android 的自带控件

5.3.2 Android 的自定义View 相关

5.3.3 关于View 的可见性

https://www.jianshu.com/p/54a2af8f8e2b

5.3.4 关于SurfaceView

使用方式

```
public class SurfaceViewTest extends Activity
{
    // SurfaceHolder负责维护SurfaceView上绘制的内容
    private SurfaceHolder holder;
    private Paint paint;

    @Override
    public void onCreate(Bundle savedInstanceState)
    {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.main);
        paint = new Paint();
        SurfaceView surface = (SurfaceView) findViewById(R.id.show);
        // 初始化SurfaceHolder对象
        holder = surface.getHolder();
        holder.addCallback(new Callback()
        {
            @Override
            public void surfaceChanged(SurfaceHolder arg0, int arg1, int arg2,
                    int arg3)
            {
            }

```

```
            @Override
            public void surfaceCreated(SurfaceHolder holder)
            {
                // 锁定整个SurfaceView
                Canvas canvas = holder.lockCanvas();
                // 绘制背景
                Bitmap back = BitmapFactory.decodeResource(
                    SurfaceViewTest.this.getResources()
                    , R.drawable.sun);
                // 绘制背景
                canvas.drawBitmap(back, 0, 0, null);
                // 绘制完成，释放画布，提交修改
                holder.unlockCanvasAndPost(canvas);
                // 重新锁一次，"持久化"上次所绘制的内容
                holder.lockCanvas(new Rect(0, 0, 0, 0));
                holder.unlockCanvasAndPost(canvas);
            }

            @Override
            public void surfaceDestroyed(SurfaceHolder holder)
            {
            }
        });
        // 为surface的触摸事件绑定监听器
        surface.setOnTouchListener(new OnTouchListener()
        {
            @Override
            public boolean onTouch(View source, MotionEvent event)
            {
                // 只处理按下事件
                if (event.getAction() == MotionEvent.ACTION_DOWN)
                {
                    int cx = (int) event.getX();
                    int cy = (int) event.getY();
                    // 锁定SurfaceView的局部区域，只更新局部内容
                    Canvas canvas = holder.lockCanvas(new Rect(cx - 50,
                            cy - 50, cx + 50, cy + 50));
                    // 保存canvas的当前状态
                    canvas.save();
                    // 旋转画布
                    canvas.rotate(30, cx, cy);
                    paint.setColor(Color.RED);
                    // 绘制红色方块
                    canvas.drawRect(cx - 40, cy - 40, cx, cy, paint);
                    // 恢复Canvas之前的保存状态
                    canvas.restore();
                    paint.setColor(Color.GREEN);
                    // 绘制绿色方块
                    canvas.drawRect(cx, cy, cx + 40, cy + 40, paint);
                    // 绘制完成，释放画布，提交修改
                    holder.unlockCanvasAndPost(canvas);
                }
                return false;
```

```
            }
        });
    }
    }

/**
上面的程序为SurfaceHolder添加了一个CallBack实例，该Callback中定义了如下三个方法：

void surfaceChanged(SurfaceHolder holder, int format, int width, int height):当一个surface的格式或大小发生改变时回调该方法。
void surfaceCreated(SurfaceHolder holder):当surface被创建时回调该方法
void surfaceDestroyed(SurfaceHolder holder):当surface将要被销毁时回调该方法
**/


```

> **SurfaceHolder提供了如下方法来获取Canvas对象**

1. Canvas lockCanvas():锁定整个SurfaceView对象，获取该Surface上的Canvas
2. Canvas lockCanvas(Rect dirty):锁定SurfaceView上Rect划分的区域，获取该Surface上的Canvas
3. unlockCanvasAndPost(canvas):释放绘图、提交所绘制的图形，需要注意，当调用SurfaceHolder上的unlockCanvasAndPost方法之后，该方法之前所绘制的图形还处于缓冲之中，下一次lockCanvas()方法锁定的区域可能会“遮挡”它

> **为什么要使用SurfaceView 来做过渡动画?**

因为View的绘图存在以下缺陷：

1. View缺乏双缓冲机制
2. 当程序需要更新View上的图像时，程序必须重绘View上显示的整张图片
3. 新线程无法直接更新View组件

> invalidate()和postInvalidate() 的区别及使用

http://blog.csdn.net/mars2639/article/details/6650876

5.3.5 自定义View 的其他知识点

ViewDragHelper

https://blog.csdn.net/yanbober/article/details/50419059

坐标系统

https://blog.csdn.net/yanbober/article/details/50419117

Scroller

https://blog.csdn.net/yanbober/article/details/49904715

5.4 Android 中的多线程

Android 中的多线程

5.5 Android 中的存储方式

Android 中原生的存储方式主要有以下几种常用方式

- SQLite：SQLite是一个轻量级的数据库，支持基本的SQL语法，是常被采用的一种数据存储方式。 Android为此数据库提供了一个名为SQLiteDatabase的类，封装了一些操作数据库的api
- SharedPreference： 除SQLite数据库外，另一种常用的数据存储方式，其本质就是一个xml文件，常用于存储较简单的参数设置。
- File： 即常说的文件（I/O）存储方法，常用语存储大数量的数据，但是缺点是更新数据将是一件困难的事情。
- ContentProvider: Android系统中能实现所有应用程序共享的一种数据存储方式，由于数据通常在各应用间的是互相私密的，所以此存储方式较少使用，但是其又是必不可少的一种存储方式。例如音频，视频，图片和通讯录，一般都可以采用此种方式进行存储。每个Content Provider都会对外提供一个公共的URI（包装成Uri对象），如果应用程序有数据需要共享时，就需要使用Content Provider为这些数据定义一个URI，然后其他的应用程序就通过Content Provider传入这个URI来对数据进行操作。

5.6 Android 中的相机与相册

Android 中原生的存储方式主要有以下几种常用方式

5.10 Android 的细节注意点

1. 全局异常处理
2. 序列化

6.Android进阶知识点

Android 开发的优化

- **应用的内存优化**

- **应用的启动时间优化**
- **应用安装包Size 的裁剪**
- **应用的多机型适配**
- **电量的优化**
- **Android 中的内存泄露**
- **Android 进程的保活**

  理解Android 的保活,首先要理解Android 中的进程:

> **Android 中有几种进程?**

1. 前台进程：即与用户正在交互的Activity或者Activity用到的Service等，如果系统内存不足时前台进程是最后被杀死的
2. 可见进程：可以是处于暂停状态(onPause)的Activity或者绑定在其上的Service，即被用户可见，但由于失去了焦点而不能与用户交互
3. 服务进程：其中运行着使用startService方法启动的Service，虽然不被用户可见，但是却是用户关心的，例如用户正在非音乐界面听的音乐或者正在非下载页面自己下载的文件等；当系统要空间运行前两者进程时才会被终止
4. 后台进程：其中运行着执行onStop方法而停止的程序，但是却不是用户当前关心的，例如后台挂着的QQ，这样的进程系统一旦没了有内存就首先被杀死
5. 空进程：不包含任何应用程序的程序组件的进程，这样的进程系统是一般不会让他存在的

> **如何避免应用在后台被杀死 Service的保活 和 进程保活一致**

1. Service设置成START_STICKY
   - kill 后会被重启（等待5秒左右），重传Intent，保持与重启前一样
2. 提升service优先级
   - 在AndroidManifest.xml文件中对于intent-filter可以通过`android:priority = "1000"`这个属性设置最高优先级，1000是最高值，如果数字越小则优先级越低，**同时适用于广播**。
   - 【结论】目前看来，priority这个属性貌似只适用于broadcast，对于Service来说可能无效
3. 提升service进程优先级
   - Android中的进程是托管的，当系统进程空间紧张的时候，会依照优先级自动进行进程的回收
   - 当service运行在低内存的环境时，将会kill掉一些存在的进程。因此进程的优先级将会很重要，可以在startForeground()使用startForeground()将service放到前台状态。这样在低内存时被kill的几率会低一些。
   - 【结论】如果在极度极度低内存的压力下，该service还是会被kill掉，并且不一定会restart()
4. onDestroy方法里重启service
   - service +broadcast 方式，就是当service走onDestory()的时候，发送一个自定义的广播，当收到广播的时候，重新启动service
   - 也可以直接在onDestroy()里startService
   - 【结论】当使用类似口口管家等第三方应用或是在setting里-应用-强制停止时，APP进程可能就直接被干掉了，onDestroy方法都进不来，所以还是无法保证
5. 监听系统广播判断Service状态
   - 通过系统的一些广播，比如：手机重启、界面唤醒、应用状态改变等等监听并捕获到，然后判断我们的Service是否还存活，别忘记加权限

    - 【结论】这也能算是一种措施，不过感觉监听多了会导致Service很混乱，带来诸多不便
6. 在JNI层,用C代码fork一个进程出来
    - 这样产生的进程,会被系统认为是两个不同的进程.但是Android5.0之后可能不行
7. root之后放到system/app变成系统级应用

大招: 放一个像素在前台(手机QQ)

- **Android 中的图片加载优化**

  图片加载优化,我们会常听到三级缓存这个方式, 那什么是三级缓存呢?

  > 三级缓存

  - 网络加载，不优先加载，速度慢，浪费流量
  - 本地缓存，次优先加载，速度快
  - 内存缓存，优先加载，速度最快

  > 三级缓存的原理

  - 首次加载 Android App 时，肯定要通过网络交互来获取图片，之后我们可以将图片保存至本地SD卡和内存中
  - 之后运行 App 时，优先访问内存中的图片缓存，若内存中没有，则加载本地SD卡中的图片
  - 总之，只在初次访问新内容时，才通过网络获取图片资源

  > 网络图片的加载 以及LRU

Android 的开发模式理解

- MVC
- MVP
- MVVM

Android 的开源框架

1. 注解ICO框架
2. 网络OKHttp 框架
3. 图片加载框架
    - Fresco
    - Glide
4. 数据库框架
    - Realm
    - GreenDao
5. 消息机制框架 EventBus
6. 布局框架

    - vlayout
7. 内存监控
    - LeakCanery

Android 的应用发布流程

Android 运行原理

*Android 的开机过程

* Android 的启动流程

* Android 的打包流程

* APK 的安装流程

Android 前沿知识点

- 热修复
- Kotlin
- Rxjava+Retrofit
- git
- gradle
- databinding

Android 推送方案

https://www.cnblogs.com/Joanna-Yan/p/6241354.html

7. 网络协议相关

计算机网络TCP/IP模型

1. 计算机网络TCP/IP五层模型

2. TCP和UDP协议的区别

TCP和UDP协议的区别是什么?

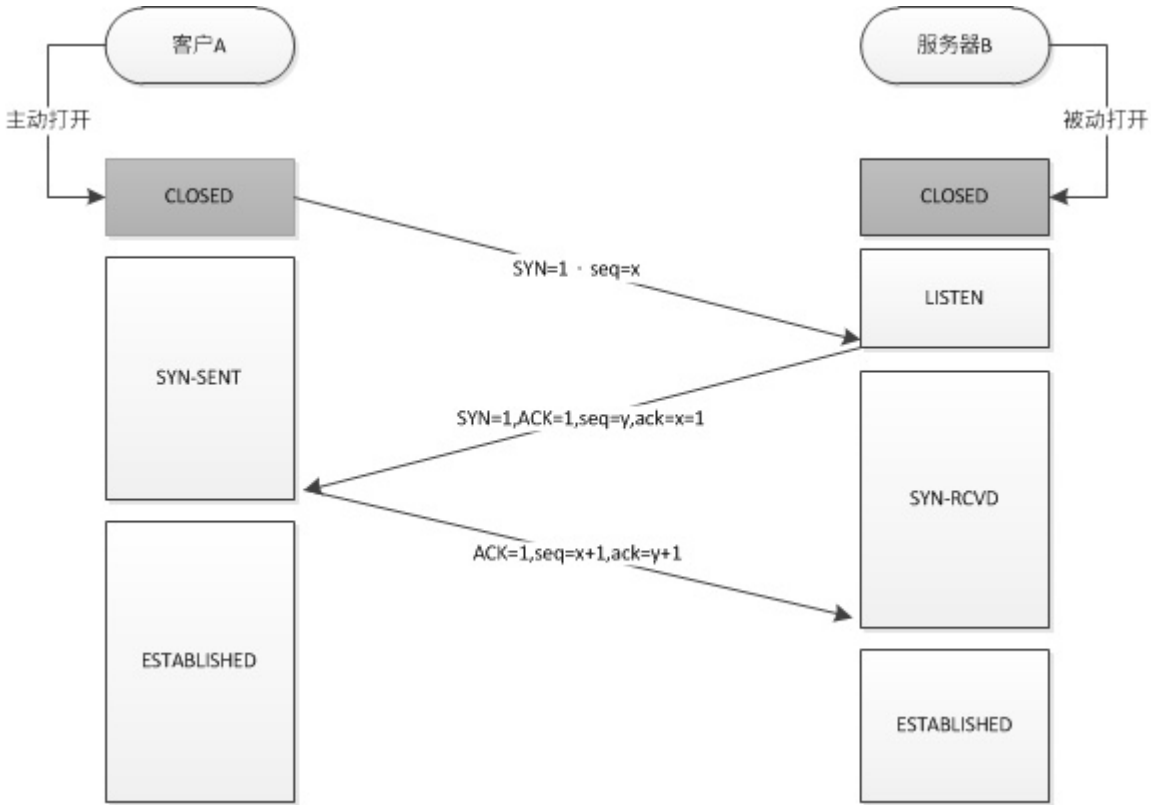
- TCP：面向链接的保证可靠传输的协议，通过TCP协议传输，得到的是一个顺序的无差错的数据流，能提供两台计算机之间的可靠数据传输。
- UDP协议：无链接的协议，每个数据包都是一个独立的信息，包括原地址和目的地址，他在网络上可以通过任何的途径把信息传递到目的地。至于能否达到目的地、达到目的地的时间、信息的准确性都不能得到保证。

既然有可靠的TCP协议，为什么还需要不可靠的UDP协议?

- TCP协议是面向连接的、可靠的、有序的、以字节流的方式发送数据，通过三**次握手**方式建立连接，形成传输数据的通道，在连接中进行大量数据的传输，效率会稍低。可靠的协议肯定需要付出代价，因此TCP的传输效率远远不如UDP高。例如对数据的时间和检验必然会消耗计算机的计算时间和网络带宽。
- 不是所有的程序都需要保证数据传递的可靠性，例如视频聊天、音频传输、游戏、直播技术等只要保证数据连贯性即可，不考虑数据是否安全等问题，所以这些场景下用UDP更合适。

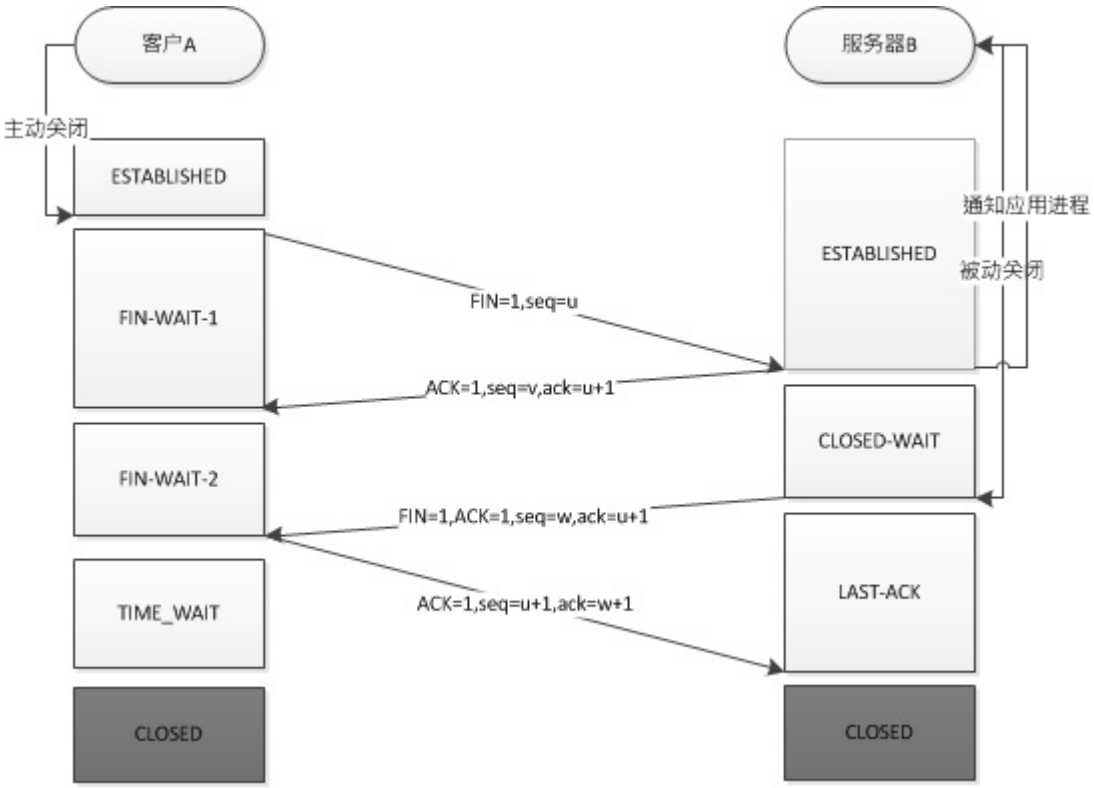
3. TCP中的三次握手和四次挥手

三次握手过程?



1. A对B说：我的序号是x，我要向你请求连接；（第一次握手，发送SYN包，然后进入SYN_SEND状态）
2. B听到之后对A说：我的序号是y，期待你下一句序号是y+1的话（意思就是收到了序号为x的话，即ack=x+1），同意建立连接。（第二次握手，发送ACK_SYN包，然后进入SYN_RCVD状态）
3. A听到B说同意建立连接之后，对A说：与确认你同意与我连接（ack=y+1,ACK=1,seq=x+1）。（第三次握手，A已进入ESTABLISHED状态）
4. B听到A的确认之后，也进入ESTABLISHED状态。

四次挥手过程?



1. A与B交谈结束之后，A要结束此次会话，对B说：我要关闭连接了（seq=u,FIN=1）。（第一次挥手，A进入FIN_WAIT_1）
2. B收到A的消息后说：确认，你要关闭连接了。（seq=v,ack=u+1,ACK=1）（第二次挥手，B进入CLOSE_WAIT）
3. A收到B的确认后，等了一段时间，因为B可能还有话要对他说。（此时A进入FIN-WAIT-2）
4. B说完了他要说的话（只是可能还有话说）之后，对A说，我要关闭连接了。（seq=w, ack=u+1,FIN=1, ACK=1）(第三次挥手)
5. A收到B要结束连接的消息后说：已收到你要关闭连接的消息。（seq=u+1,ack=w+1,ACK=1）(第四次挥手，然后A进入CLOSED)
6. B收到A的确认后，也进入CLOSED。

为什么需要三次握手?

- 三次握手的目的是建立可靠的通信信道，说到通讯，简单来说就是数据的发送与接收，而三次握手最主要的目的就是**双方确认自己与对方的发送与接收机能正常**。
- 三次握手的另外一个目的就是通过发送特定的数据包，确认双方都支持TCP，告知对方用TCP传输。

为什么要四次挥手?

- 根本原因是，一方发送FIN只表示自己发完了所有要发的数据，但还允许对方继续把没发完的数据发过来。
- 与三次握手类似，通过发送特定的数据包，确认双方都支持TCP，告知对方用TCP传输。

4. 参考文章

互联网协议入门（一）

互联网协议入门（二）

TCP三次握手与四次挥手过程

简明理解三次握手和四次挥手

图解TCP通信三次握手和四次挥手

HTTP协议

1. HTTP协议是什么?

协议：计算机通信网络中两台计算机之间进行通信所必须的共同遵守的规定或规则。

HTTP协议：是计算机网络中应用层中的协议，基于TCP/IP通信协议来传递数据，属于应用层的面向对象的超文本传输协议。

2. HTTP协议的特点有哪些?

- 简单快速：客户向服务器请求服务时，只需传送请求方法和路径。请求方法常用的有GET、HEAD、POST。每种方法规定了客户与服务器联系的类型不同。由于HTTP协议简单，使得HTTP服务器的程序规模小，因而通信速度很快。
- 灵活：HTTP允许传输任意类型的数据对象。正在传输的类型由Content-Type加以标记。
- 非持续连接：无连接的含义是限制每次连接只处理一个请求。服务器处理完客户的请求，并收到客户的应答后，即断开连接。采用这种方式可以节省传输时间。
- 无状态：HTTP协议是无状态协议。无状态是指协议对于事务处理没有记忆能力。缺少状态意味着如果后续处理需要前面的信息，则它必须重传，这样可能导致每次连接传送的数据量增大。另一方面，在服务器不需要先前信息时它的应答就较快。
- 支持B/S及C/S模式。

3. URI、URL、URN

基本概念：

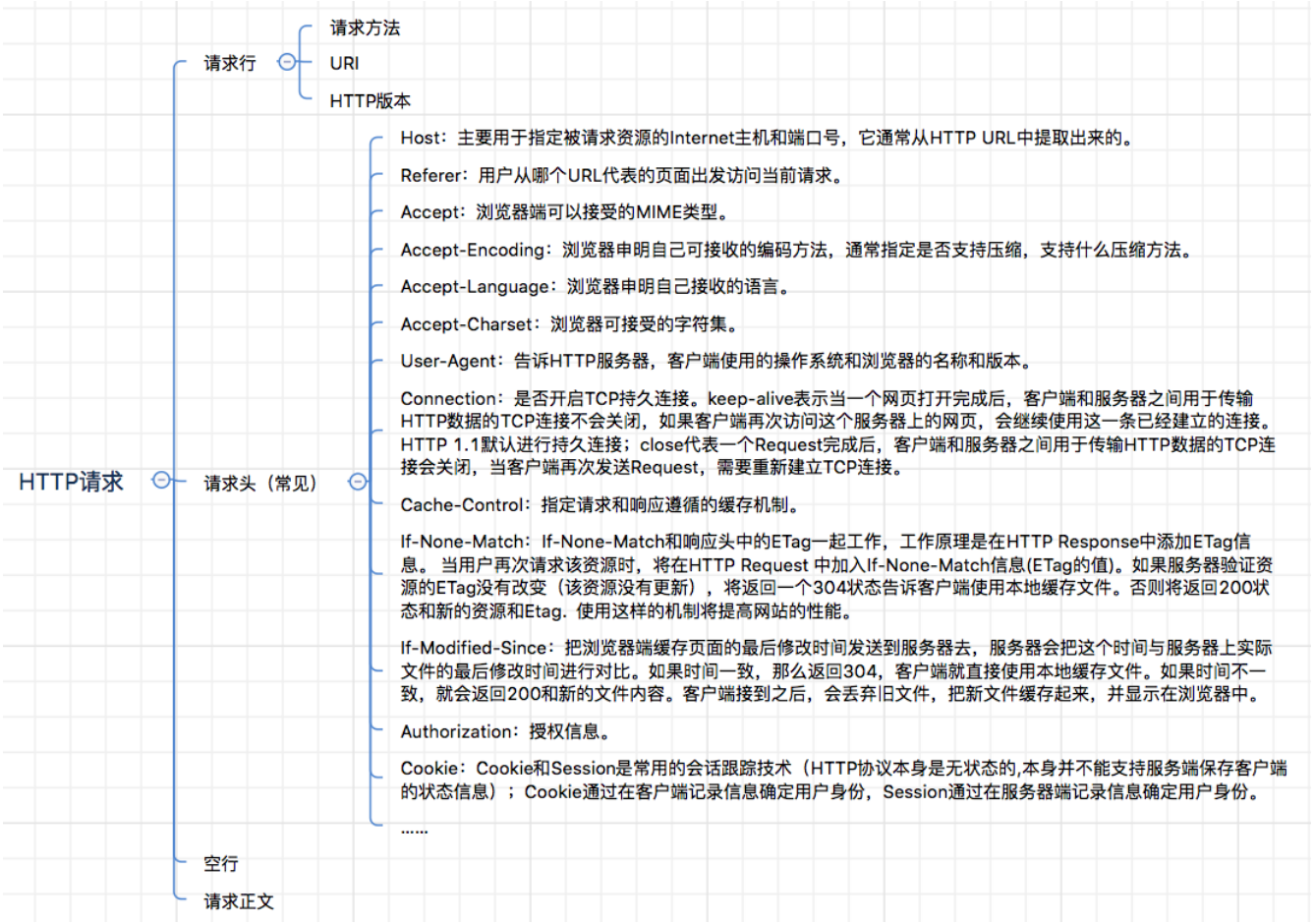
- URI(Uniform Resource Identifier)：统一资源标识符，用来唯一的标识一个资源。
    - 访问资源的命名机制
    - 存放资源的主机名
    - 资源自身的名称，由路径表示
- URL(Uniform Resource Locator)：统一资源定位符，是一种具体的URi，即URL可以用来标识一个资源，而且还指明了如何定位这个资源。
    - 协议：常见的有ftp(文件传输协议)、http(超文本传输协议)、mailto(电子邮件)、file(特定主机文件名)
    - 存放资源的主机IP地址、端口号
        - IP地址：为实现网络中不同计算机之间的通信，每台计算机都必须有一个唯一的标识-IP地址。它是识别网络通讯的实体，可以理解为主机，也可以理解为每个路由器的端口。
        - 端口号：一个通讯实体他可以拥有很多的通讯程序同时提供网络服务，这个时候就要通过端口号来区分通讯程序，一个通讯实体不能有两个通讯程序使用同一个端口号。端口号范围为0-65535，其中0-1023位为系统保留。
    - 主机资源的具体地址
    - 请求参数
    - 锚点
- URN(Uniform Resource Name)：统一资源命名，是通过名字来标识资源，与其所处的位置无关，这样即使资源的位置发生变动，其URN也不会变化。用于标识持久性Internet资源，URN可以提供一种机制，用于查找和检索定义特定命名空间的架构文件。（P2P下载中使用的磁力链接是URN的一种实现，它可以持久化的标识一个BT资源，资源分布式的存储在P2P网络中，无需中心服务器用户即可找到并下载它。）

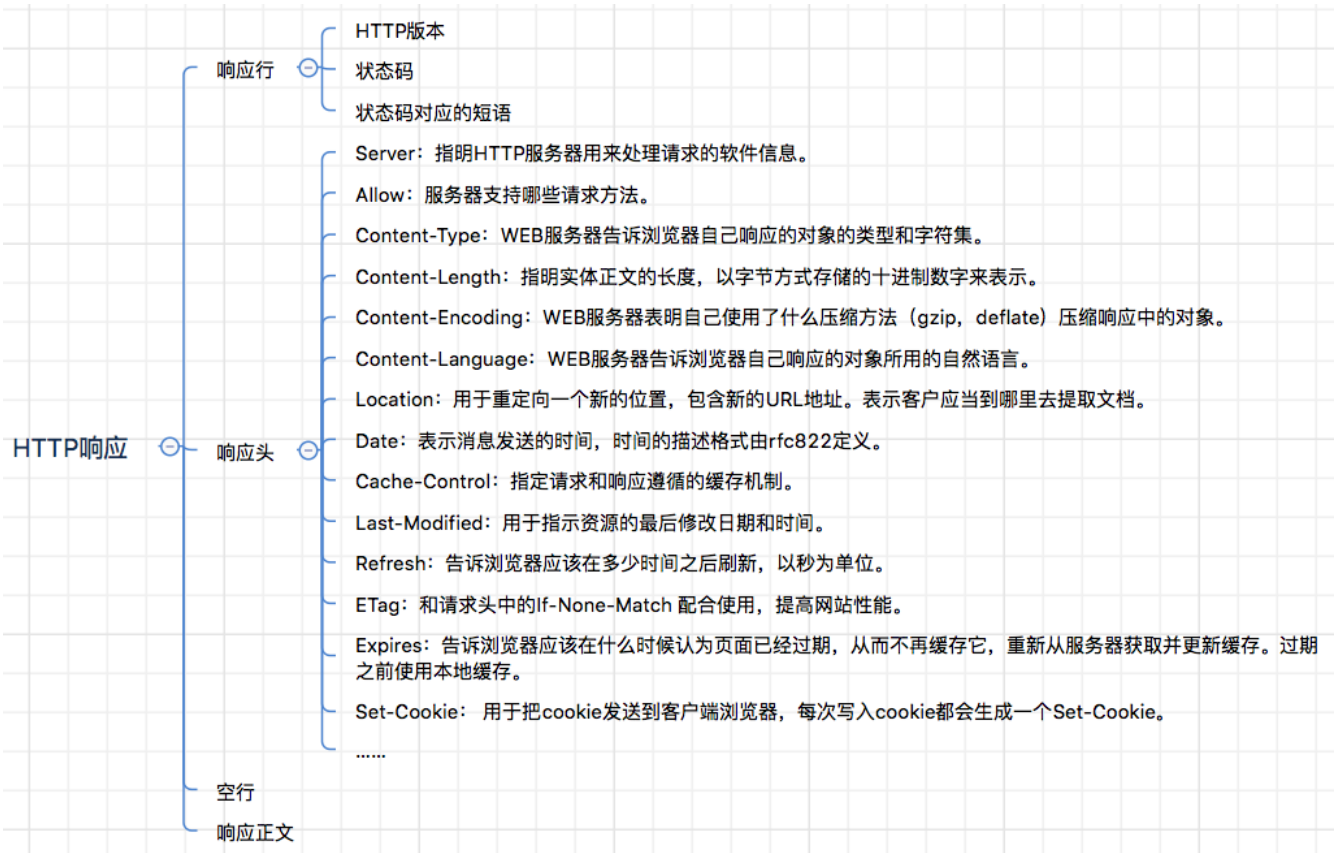
URI与URL/URN直接的区别?

- HTTP协议中使用URI来传输数据和建立连接。
- URL、URN都是一种具体的URI，URL与URN都是URI的子集；而URI属于URL和URN更高层次的抽象定义，URI使用两种方法标识资源：
  - URL：用地址标识
  - URN：用名称标识
- URL强调的是资源的路径、URN强调的是资源的名称；而URI强调的是资源本身

4. 打开浏览器输入URL然后发生了什么?

5. HTTP请求和响应





6. 影响HTTP请求的因素有哪些?

HTTP是建立在TCP协议之上，所以HTTP协议的瓶颈及其优化技巧都是基于TCP协议本身的特性。影响一个HTTP网络请求的因素主要有两个：带宽和延迟：

- 带宽：拨号上网的阶段带宽是一个比较严重影响请求的问题；但是现在网络基础建设已经使得带宽得到极大的提升，我们不再会担心由带宽而影响网速，那么就只剩下延迟了。
- 延迟：
    - 浏览器阻塞（HOL Blocking）：浏览器会因为一些原因阻塞请求。浏览器对于同一个域名，同时只能有4个连接（这个根据浏览器内核不同可能会有所差异），超过浏览器最大连接数限制，后续请求就会被阻塞。
    - DNS查询（DNS Lookup）：浏览器需要知道目标服务器的IP才能建立连接。将域名解析为 IP 的这个系统就是DNS。这个通常可以利用DNS缓存结果来达到减少这个时间的目的。
    - 建立连接（Initial Connection）：HTTP是基于TCP协议的，浏览器最快也要在第三次握手时才能捎带HTTP请求报文，达到真正的建立连接，但是连接无法复用会导致每次请求都经历三次握手和慢启动。三次握手在高延迟的场景下影响较明显，慢启动则对文件类大请求影响较大。

7. HTTP1.0、HTTP1.1、HTTP2.0

HTTP1.0、HTTP1.1的主要区别：

- 缓存处理：在HTTP1.0中主要使用header里的If-Modified-Since,Expires来做为缓存判断的标准，HTTP1.1则引入了更多的缓存控制策略例如Entity tag，If-Unmodified-Since, If-Match, If-None-Match等更多可供选择的缓存头来控制缓存策略。
- 带宽优化及网络连接的使用：HTTP1.0中，存在一些浪费带宽的现象，例如客户端只是需要某个对象的一部分，而服务器却将整个对象送过来了，并且不支持断点续传功能，HTTP1.1则在请求头引入了range头域，它允许只请求资源的某个部分，即返回码是206（Partial Content），这样就方便了开发者自由的选择以便于充分利用带宽和连接。
- 错误通知的管理：在HTTP1.1中新增了24个错误状态响应码，如409（Conflict）表示请求的资源与资源的当前状态发生冲突；410（Gone）表示服务器上的某个资源被永久性的删除。
- Host头处理：在HTTP1.0中认为每台服务器都绑定一个唯一的IP地址，因此，请求消息中的URL并没有传递主机名（hostname）。但随着虚拟主机技术的发展，在一台物理服务器上可以存在多个虚拟主机（Multi-homed Web Servers），并且它们共享一个IP地址。HTTP1.1的请求消息和响应消息都应支持Host头域，且请求消息中如果没有Host头域会报告一个错误（400 Bad Request）。
- 长连接：HTTP 1.1支持长连接（PersistentConnection）和请求的流水线（Pipelining）处理，在一个TCP连接上可以传送多个HTTP请求和响应，减少了建立和关闭连接的消耗和延迟，在HTTP1.1中默认开启Connection: keep-alive，一定程度上弥补了HTTP1.0每次请求都要创建连接的缺点。

HTTP1.0和1.1现存的一些问题：

- HTTP1.x在传输数据时，每次都需要重新建立连接，无疑增加了大量的延迟时间，特别是在移动端更为突出。
- HTTP1.x在传输数据时，所有传输的内容都是明文，客户端和服务器端都无法验证对方的身份，这在一定程度上无法保证数据的安全性。
- HTTP1.x在使用时，header里携带的内容过大，在一定程度上增加了传输的成本，并且每次请求header基本不怎么变化，尤其在移动端增加用户流量。

- 虽然HTTP1.x支持了keep-alive，来弥补多次创建连接产生的延迟，但是keep-alive使用多了同样会给服务端带来大量的性能压力，并且对于单个文件被不断请求的服务(例如图片存放网站)，keep-alive可能会极大的影响性能，因为它在文件被请求之后还保持了不必要的连接很长时间。

HTTP1.1、HTTP2.0的主要区别：

- 新的二进制格式（Binary Format）：HTTP1.x的解析是基于文本。基于文本协议的格式解析存在天然缺陷，文本的表现形式有多样性，要做到健壮性考虑的场景必然很多，二进制则不同，只认0和1的组合。基于这种考虑HTTP2.0的协议解析决定采用二进制格式，实现方便且健壮。
- 多路复用（MultiPlexing）：即连接共享，即每一个request都是是用作连接共享机制的。一个request对应一个id，这样一个连接上可以有多个request，每个连接的request可以随机的混杂在一起，接收方可以根据request的 id将request再归属到各自不同的服务端请求里面。多路复用原理图：
- Header压缩：如上文中所言，对前面提到过HTTP1.x的header带有大量信息，而且每次都要重复发送，HTTP2.0使用encoder来减少需要传输的header大小，通讯双方各自cache一份header fields表，既避免了重复header的传输，又减小了需要传输的大小。
- 服务端推送（server push）：同SPDY一样，HTTP2.0也具有server push功能。

8. GET请求和POST请求的区别

基本区别：

- 行为上的区别：GET请求是从服务器请求数据；POST请求是向服务器提交要数据
- 数据的类型、大小是否有限制：GET请求有限制；POST请求没有
- 安全性：GET请求提交的数据都在URL中，可见，不安全；相对来说POST请求更加安全
- JS中获取参数上的区别：GET请求在JS中是通过`Request.QueryString["xxx"]`获取URL中的参数，而POST请求在JS中是通过`Request.Form["xxx"]`获取Form表单中的参数

详细区别：

	GET	POST
后退按钮/刷新	无害	数据会被重新提交（浏览器应该告知用户数据会被重新提交）。
书签	可收藏为书签	不可收藏为书签
缓存	能被缓存	不能缓存
编码类型	application/x-www-form-urlencoded	application/x-www-form-urlencoded 或 multipart/form-data。为二进制数据使用多重编码。
历史	参数保留在浏览器历史中。	参数不会保存在浏览器历史中。
对数据长度的限制	是的。当发送数据时，GET 方法向 URL 添加数据；URL 的长度是受限制的（URL 的最大长度是 2048 个字符）。	无限制。
对数据类型的限制	只允许 ASCII 字符。	没有限制。也允许二进制数据。
安全性	与 POST 相比，GET 的安全性较差，因为所发送的数据是 URL 的一部分。 在发送密码或其他敏感信息时绝不要使用 GET ！	POST 比 GET 更安全，因为参数不会被保存在浏览器历史或 web 服务器日志中。
可见性	数据在 URL 中对所有人都是可见的。	数据不会显示在 URL 中。

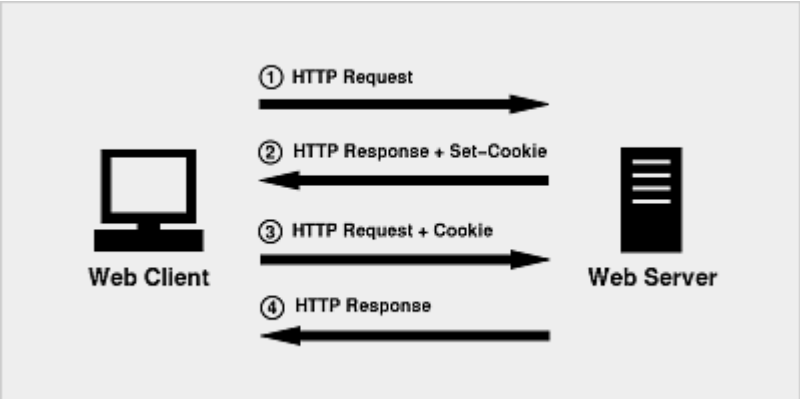
9. Cookie和Session的区别

Cookie的基本概念与工作原理

Cookie的基本概念：Cookie技术是客户端的会话跟踪解决方案，弥补了HTTP协议无状态的缺点。Cookie就是由服务器发给客户端的特殊信息，而这些信息以文本文件的方式存放在客户端，然后客户端每次向服务器发送请求的时候都会带上这些特殊的信息。

Cookie的工作原理：

1. 客户端发送一个http请求到服务器端
2. 服务器端发送一个http响应到客户端，其中包含Set-Cookie头部
3. 客户端发送一个http请求到服务器端，其中包含Cookie头部
4. 服务器端发送一个http响应到客户端

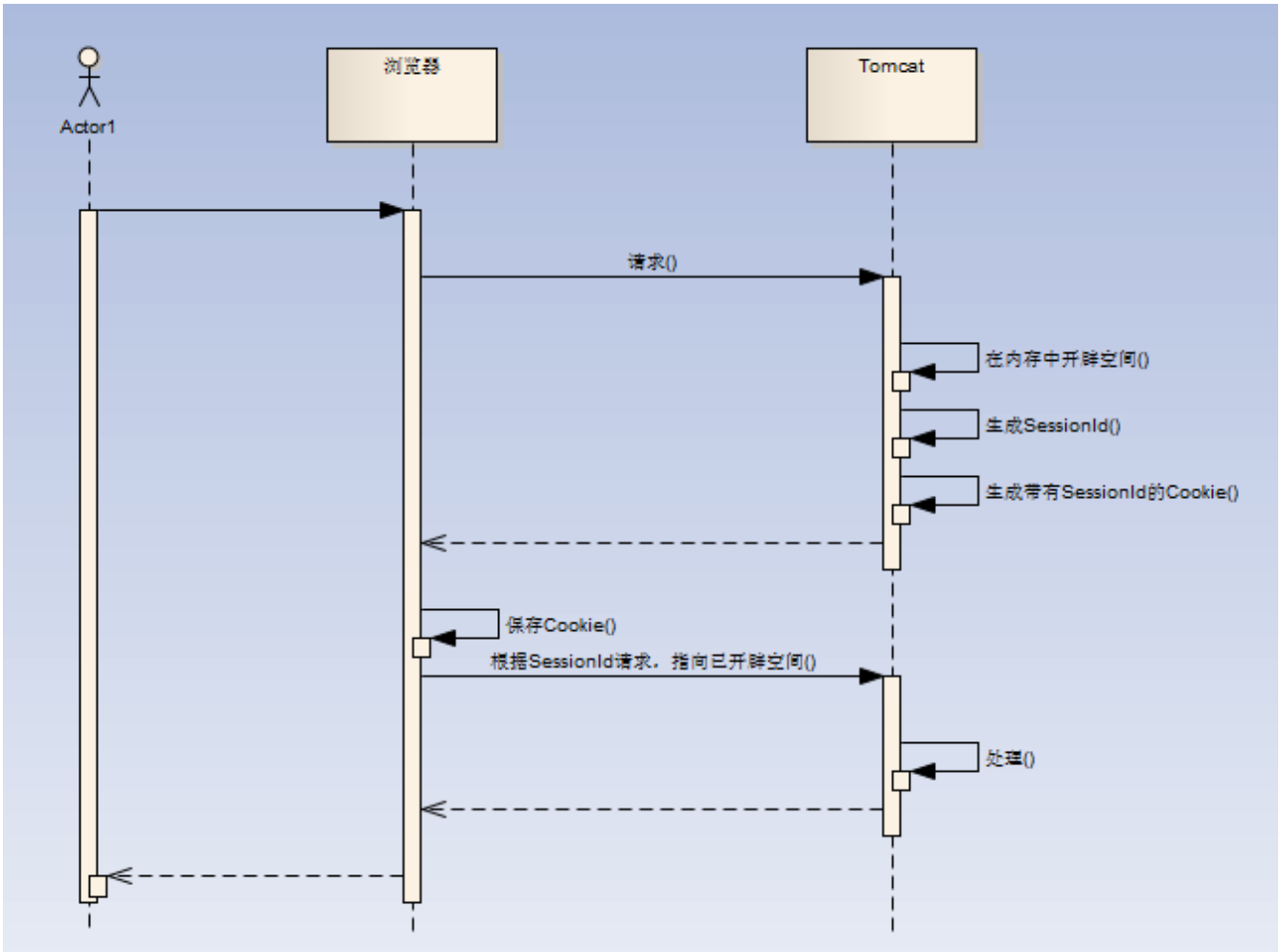


Session的基本概念与工作原理

Session的基本概念：Session是另一种记录客户状态的机制，不同的是Cookie保存在客户端浏览器中，而Session保存在服务器上。客户端浏览器访问服务器的时候，服务器把客户端信息以某种形式记录在服务器上。这就是Session。客户端浏览器再次访问时只需要从该Session中查找该客户的状态就可以了。

Session的工作原理：

1. 服务端程序运行的过程中创建Session，并且分配Session ID
2. 调用Session相关的方法往Session中增加内容，而这些内容只会保存在服务器中，发到客户端的只有Session ID（通过Cookie发送）
3. 当客户端再次发送请求的时候，会将这个Session ID带上，服务器接受到请求之后就会依据Session ID找到相应的Session，从而再次使用之。



Cookie和Session的区别

1. 存取方式的不同
    - Cookie中只能保管ASCII字符串，假如需求存取Unicode字符或者二进制数据，需求先进行编码。Cookie中也不能直接存取Java对象。若要存储略微复杂的信息，运用Cookie是比拟艰难的。
    - Session中能够存取任何类型的数据，包括而不限于String、Integer、List、Map等。Session中也能够直接保管Java Bean乃至任何Java类，对象等，运用起来十分便当。能够把Session看做是一个Java容器类。
2. 隐私策略的不同
    - Cookie存储在客户端中，对客户端是可见的，客户端的一些程序可能会窥探、复制以至修正Cookie中的内容。
    - Session存储在服务器上，对客户端是透明的，不存在敏感信息泄露的风险。
    - 使用Cookie时，最好对敏感的信息信息加密，提交到服务器后再进行解密。
    - Session是放在服务器上，Session里任何隐私都能够有效的保护。
3. 有效期上的不同
    - 要实现登录信息长期有效，运用Cookie会是比较好的选择。只需要设置Cookie的过期时间属性为一个很大很大的数字。
    - Session的有效期依赖于名为JSESSIONID的Cookie，而该Cookie的过期时间默认为–1，只需关闭了浏览器该Session就会失效，因而Session不能完成信息永世有效的效果。运用URL地址重写也不能完成。而且假如设置Session的超时时间过长，服务器累计的Session就会越多，越容易招致内存溢出。
4. 服务器压力的不同
    - Session是保管在服务器端的，每个用户都会产生一个Session。假如并发访问的用户十分多，会产生十分多的Session，耗费大量的内存。因而像Google、Baidu这样并发访问量极高的网站，是不太可能运用

Session来追踪客户会话的。
    - Cookie保管在客户端，不占用服务器资源。假如并发的用户十分多，Cookie是很好的选择。关于Google、Baidu来说，Cookie或许是唯一的选择。

10. 参考文章

关于HTTP协议，一篇就够了

HTTP协议详解

互联网协议入门（一）

互联网协议入门（二）

URI、URL和URN的区别

URL、URN、URI的区别?

详解URL的组成

百度百科-URN

浏览器中输入url后发生了什么

当你在浏览器中输入url地址之后发生了什么?

当在浏览器地址栏里输入URL后会发生什么事情

DNS解析过程，以及Wireshark抓包数据包分析

一篇文章带你详解 HTTP 协议（网络协议篇一）

http协议-思维导图

HTTP,HTTP2.0,SPDY,HTTPS你应该知道的一些事

HTTP1.0 HTTP 1.1 HTTP 2.0主要区别

HTTP 方法：GET 对比 POST

GET和POST请求的区别-深入答案

理解Cookie和Session机制

详解Cookie和Session

Cookie与Session的区别

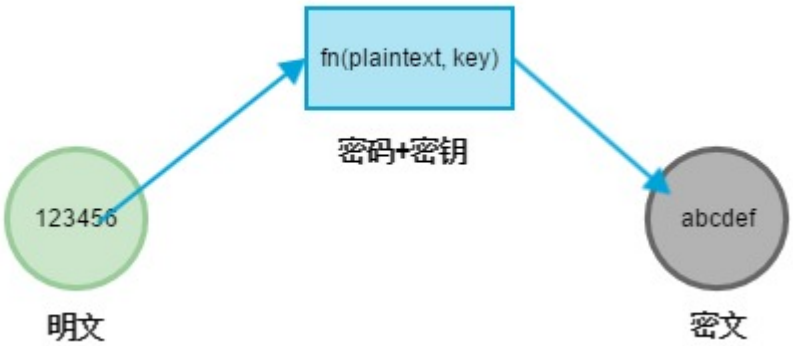
DNS原理及其解析过程【精彩剖析】

密码学基础知识

1. 密码学基础概念

- 加密/解密：计算机术语加密/解密是一种用于加密或者解密的算法。分为对称加密和非对称加密。
- 密钥：密钥是一种参数，它是在使用加密/解密算法过程中输入的参数。同一个明文在相同的密码算法和不同的密钥计算下会产生不同的密文。很多知名的密码算法都是公开的，密钥才是决定密文是否安全的重要参数，通常密钥越长，破解的难度越大。密钥分为对称密钥与非对称密钥。

- Hash算法：Hash算法是一种单向算法，通过Hash算法可以对目标数据生成一段特定长度、唯一的hash值，但是不能通过这个hash值重新计算出原始的数据，因此也称之为摘要算法。经常被用在不需要数据还原的密码加密以及数据完整性校验上，常用的算法有MD2，MD4，MD5，SHA等。
- 明文/密文：明文是加密之前的原始数据，密文是通过密码运算后得到的结果成为密文。



2. 对称加密和非对称加密

对称加密

对称加密：又称为共享密钥加密，对称加密在加密和解密的过程中使用的密钥是相同的，常见的对称加密算法有DES、3DES、AES、RC5、RC6。

- 优点：计算速度快。
- 缺点：密钥需要在通讯的两端共享，维护比较麻烦，保密性差。如果所有客户端都共享同一个密钥，那么可以凭借一个密钥破解所有人的密文了；如果每个客户端与服务端单独维护一个密钥，那么服务端需要管理大量的密钥。

非对称加密

非对称加密：又称为公开密钥加密，服务端会生成一对密钥，一个私钥保存在服务端，仅自己知道，另一个是公钥，公钥可以自由发布供任何人使用。客户端的明文通过公钥加密后的密文需要用私钥解密。RSA，ECC，DSA(数字签名)。

- 优点：非对称加密保密性好。与对称加密相比，非对称加密无需在客户端和服务端之间共享密钥，只要私钥不发给任何用户，即使公钥在网上被截获，也无法被解密，仅有被窃取的公钥是没有任何用处的。
- 缺点：加密和解密花费的时间较长，不适合对大文件加密而只适合对少量的数据加密。

非对称加解密的过程：

1. 服务端生成配对的公钥和私钥
2. 私钥保存在服务端，公钥发送给客户端
3. 客户端使用公钥加密明文传输给服务端
4. 服务端使用私钥解密密文得到明文

RSA算法的原理：

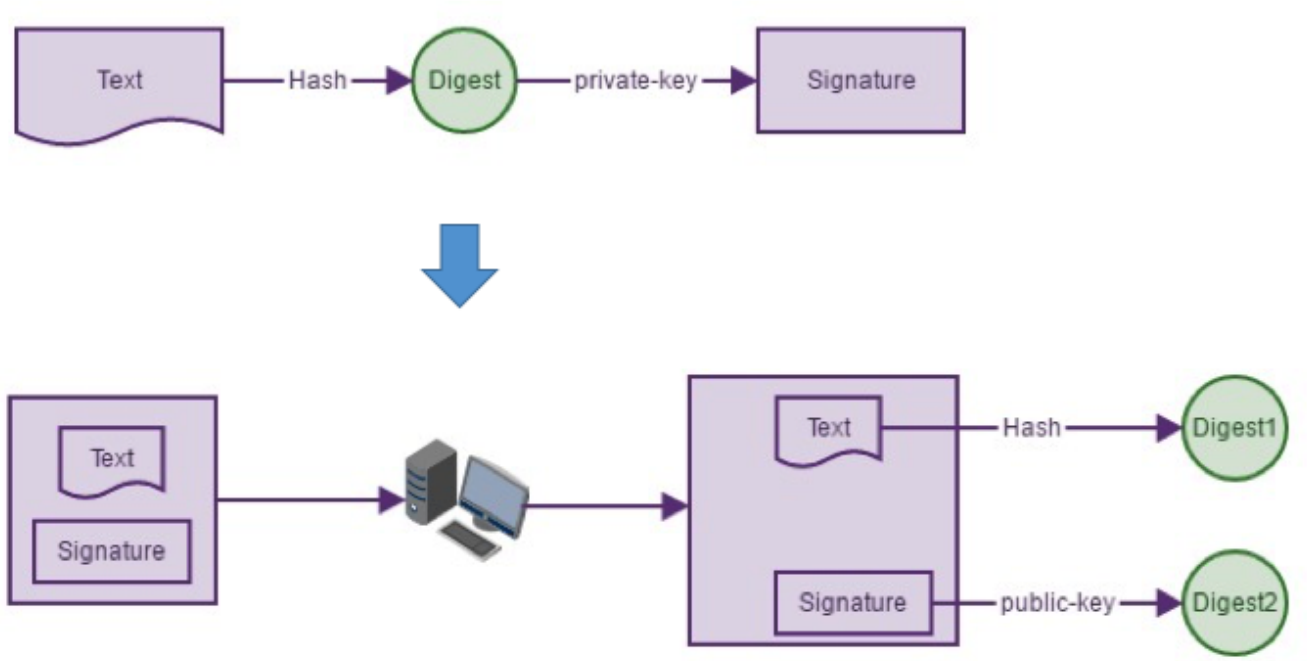
- RSA算法基于一个十分简单的数论事实：将两个大质数相乘十分容易，但是想要对其乘积进行因式分解却极其困难。可以将乘积公开作为公钥，两个质数作为加密解密的关键。

3. 数字签名

- 数字签名：**用于验证传输的内容的真实性**。数字签名是非对称加密的一种应用场景。数字签名是反过来用私钥来加密，通过与之配对的公钥来解密。

数字签名的步骤：

1. 服务端把报文经过Hash处理后生成摘要信息Digest，摘要信息使用私钥加密之后就生成签名，服务器把签名连同报文一起发送给客户端。
2. 客户端接收到数据后，把签名提取出来用解密，如果能正常的解密出来Digest2，那么就能确认是对方发的。
3. 客户端把报文Text提取出来做同样的Hash处理，得到的摘要信息Digest1，再与之前解密出来的Digest2对比，如果两者相等，就表示内容没有被篡改，否则内容就是被人改过了。因为只要文本内容哪怕有任何一点点改动都会Hash出一个完全不一样的摘要信息出来。



4. 数字证书

- 数字证书：简称CA，**用于验证密钥的真实性**，它由第三方权威机构给某网站颁发的一种认可凭证，这个凭证是被大家（浏览器）所认可的。Android已经把将近150个CA根证书（数字证书认证机构认证过的证书）内置在我们手机中。这150多个证书被全世界信赖。

数字证书的颁发过程：

1. 生成密钥对：用户首先产生自己的密钥对，并将公共密钥及部分个人身份信息传送给认证中心。
2. 确认身份：认证中心在核实身份后，将执行一些必要的步骤，以确信请求确实由用户发送而来。
3. 颁发证书：认证中心将发给用户一个数字证书，该证书内包含用户的个人信息和他的公钥信息，同时还附有认证中心的签名信息。
4. 使用证书：用户就可以使用自己的数字证书进行相关的各种活动。

数字证书的认证过程：

1. 证书安装：客户端操作系统已经存在一些权威的数字证书（Android系统在不root的情况下，是安全的）
2. 证书传输：客户端请求服务端，客户端收到服务端的消息，消息带有权威机构颁发给服务端的数字证书A
3. 证书查找：客户端从本地已经存在的权威数字证书中去寻找与服务端相同的证书B，如果存在，则服务端的证书真实，否则不真实
4. 证书验证：客户端用证书B的公钥对证书A里面的摘要和摘要算法进行解密，然后使用这个摘要算法计算A证书的摘要，将这个计算的摘要与放在证书中的摘要对比。如果一致，说明服务端的证书肯定没有被修改过并且证

书是权威机构发布的，服务端的证书中的公钥肯定是真实的，以后就采用这个公钥进行安全通信。

5. 参考文章

HTTPS中的加密算法相关概念

摘要，数字签名，数字证书认证过程详解(非HTTPS)

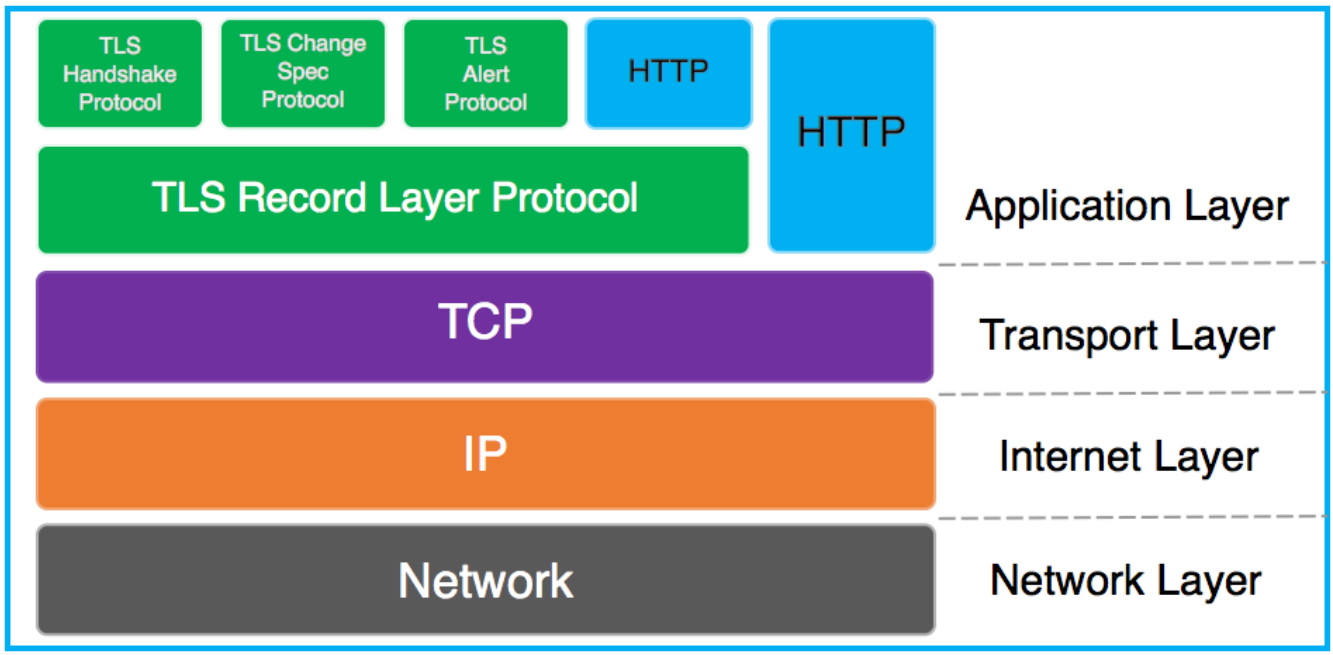
数字证书的颁发过程

HTTPS

1. HTTPS的基本概念

HTTPS：HTTPS（HTTP over TLS）并不是一个单独的协议，而是对工作在加密连接TLS上的常规HTTP协议的称呼。主要用到了对称加密、非对称加密、数字证书等技术来实现加密传输，保证通信过程的安全性。

HTTPS在TCP/IP协议栈中TLS(各子协议）和HTTP的关系如下所示：



HTTPS的特点如下：

- 优点：传输安全可靠
- 缺点：加解密相关的计算耗时，导致HTTPS的通信速度慢

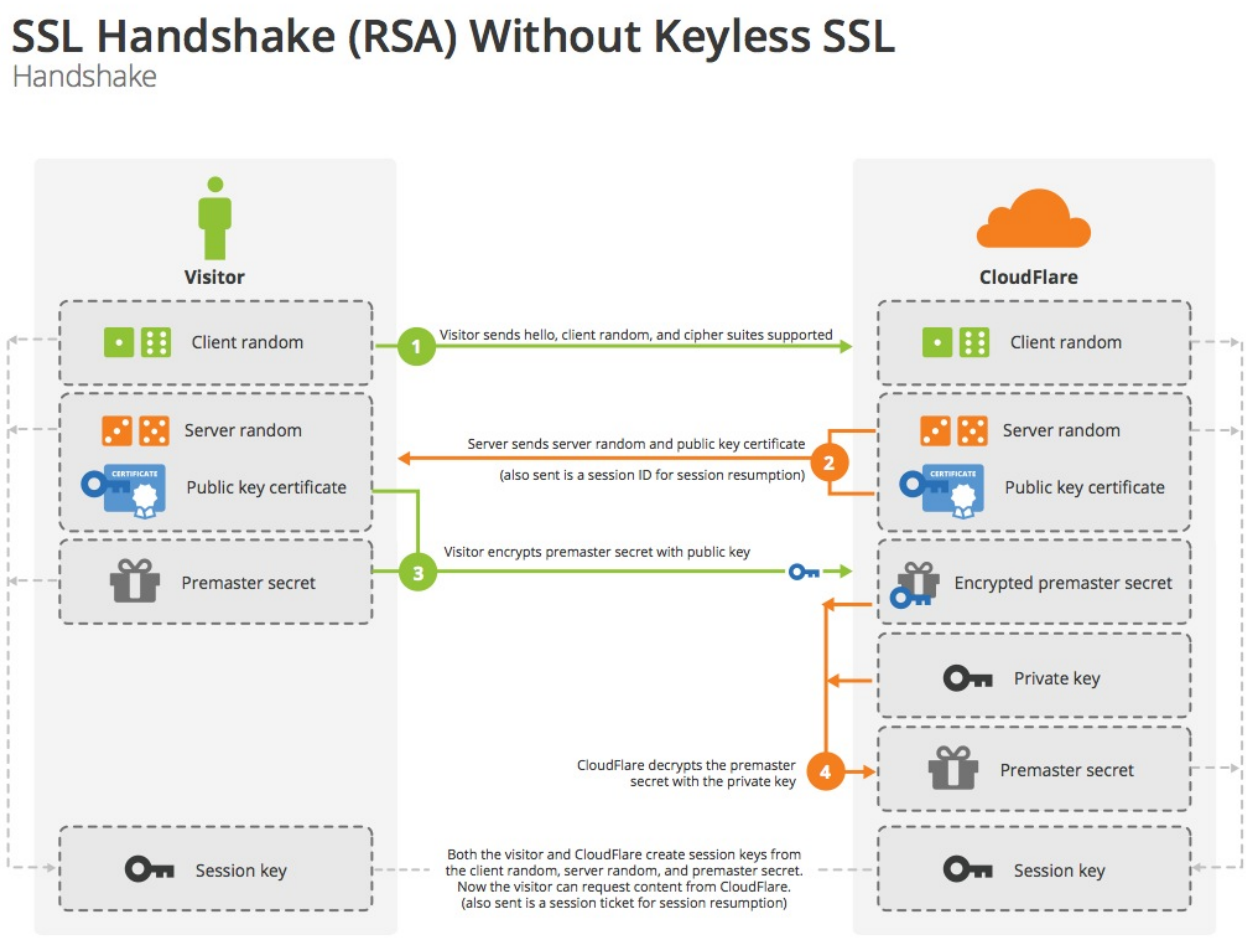
2. HTTP与HTTPS的主要区别：

- 数据完整性：HTTPS内容传输经过完整性校验
- 数据安全性/隐私性：HTTPS内容经过对称加密，每个连接生成一个唯一的加密密钥；HTTP所有传输的内容都是明文
- 身份认证：第三方无法伪造服务端（客户端）身份
- HTTPS协议需要到CA申请证书，一般免费证书很少，需要交费
- HTTP和HTTPS使用的是完全不同的连接方式，用的端口也不一样，前者是80，后者是443
- HTTPS可以有效的防止运营商劫持，解决了防劫持的一个大问题

3. TLS协议及其握手过程

- TLS：SSL是一种安全传输协议，TLS是SSL的3.0升级版，采用的是非对称加密。TLS包括TLS Record Protocol和TLS Handshake Protocol
- TLS Handshaking Protocols：提供实现了身份认证，包括Handshake protocol，Change Ciper Spec protocol和Alert protocol
- TLS Record Protocol：保证了数据完整性和隐私性

TLS握手过程（通过三个随机数保证传输的安全）：



1. 客户端发送随机数client_random和支持的加密方式列表（明文）

2. 服务器返回随机数server_random，选择的加密方式和服务器证书链（明文）
3. 客户端验证服务器证书，使用证书中的公钥加密premaster secret发送给服务端（RSA）。并且发送编码改变、握手结束通知
4. 服务端使用私钥解密premaster secret
5. 服务端端分别通过client_random，server_random和premaster secret生成master secret。服务端也返回编码改变、握手结束通知
6. 客户端也生成master secret
7. 两端通过master secret用于对称加密后续通信内容

4. HTTPS如何改造以及需要关注的问题（服务端Android）

HTTPS如何改造?

- 购买或者生成SSL证书。如果是购买证书，需要在购买或者生成证书之后，在证书提供的网站上配置自己的域名，将证书下载下来。
- 证书持有：客户端持有服务端的公钥证书，并持有自己的私钥，服务端持有客户的公钥证书，并持有自己私钥。
- 服务器端改造：配置自己的web服务器，同时进行代码改造（例如修改Tomcat配置文件等），部署。
- Android端改造：由于Retrofit是基于OkHttp实现的，因此想通过Retrofit实现HTTPS需要给Retrofit设置一个OkHttp代理对象SSLSocketFactory用于处理HTTPS的握手过程。其中需要实现SSLSocketFactory的getSSLCertifcation方法实现证书的读取、加载、初始化SSLContext等工作。

```
OkHttpClient okHttpClient = new OkHttpClient.Builder()
    .sslSocketFactory(SSLHelper.getSSLCertifcation(context))//为OkHttp对象设置SocketFactory用于双向认证
    .hostnameVerifier(new UnSafeHostnameVerifier())
    .build();
Retrofit retrofit = new Retrofit.Builder()
    .baseUrl("https://xxx:xxxx")
    .addConverterFactory(GsonConverterFactory.create())//添加 json 转换器
    .addCallAdapterFactory(RxJavaCallAdapterFactory.create())//添加 RxJava 适配器
    .client(okHttpClient)//添加OkHttp代理对象
    .build();
```

- 证书验证：建立连接的时候，客户端利用服务端的公钥证书来验证服务器是否上是目标服务器；服务端利用客户端的公钥来验证客户端是否是目标客户端。（请参考RSA非对称加密以及HASH校验算法）。服务端给客户端发送数据时，需要将服务端的证书发给客户端验证，验证通过才运行发送数据，同样，客户端请求服务器数据时，也需要将自己的证书发给服务端验证，通过才允许执行请求。

HTTPS优化的注意点?

- 访问速度：HTTPS降低用户访问速度（HTTPS未经任何优化的情况下要比HTTP慢几百毫秒以上，特别在移动端可能要慢500毫秒以上）。SSL握手，HTTPS对速度会有一定程度的降低，但是只要经过合理优化和部署（例如使用代理服务器等），HTTPS对速度的影响完全可以接受。
- 服务端压力：相对于HTTPS降低访问速度，其实更需要关心的是服务器端的CPU压力，HTTPS中大量的密钥算法计算（加密、解密、Hash等），会消耗大量的CPU资源，只有足够的优化（例如选取合适的加密、解密、完整性校验算法等），HTTPS的机器成本才不会明显增加。

参考文章

理解 HTTPS 的工作原理

Android HTTPS 自制证书实现双向认证(OkHttp + Retrofit + Rxjava)

https是如何工作的?

浅谈HTTPS协议和SSL、TLS之间的区别与关系

腾讯HTTPS性能优化实践

HTTPS 性能优化技巧

阿里聚安全组件

8. 其他相关能力

8.1 项目总结

8.2 面试技巧

8.3 编程技巧

关于位运算

关于正则表达式

https://blog.csdn.net/bobo89455100/article/category/6604866/2?orderby=UpdateTime