技术选型理由：

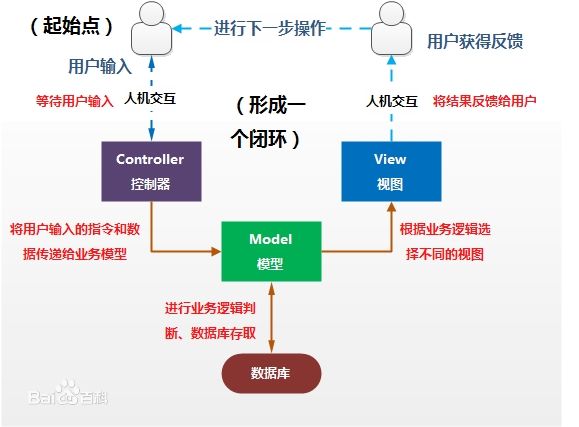
原生的Android应用使用Java编程语言开发，并且很容易的移植到其他移动平台上，可以在Windows、Mac还有Linux上都能完成Android的开发工作。谷歌为Android开发者营造了一个极为良好的开发环境。Android Studio开发工具可以极大的方便和有利于Android平台应用开发，其开发环境和模式更加的丰富和便捷，能够支持多种语音，还可以为开发者提供测试工具和各种数据分析。

此外Android Studio还支持基于Gradle的灵活的构建系统；支持构建变种版本和生成多个APK文件；支持通过代码模板来快速建立通用的app功能；支持丰富的布局编辑器，可拖放主题编辑；支持lint工具来捕获性能、可用性、版本兼容等问题；支持代码混淆和应用签名内置的Google云平台支持。因此很容易整合Google云消息和app引擎内置了ADB工具，可以方便的查看log内置了性能检测工具,可以实时的检测内存、CPU、流量、帧率。

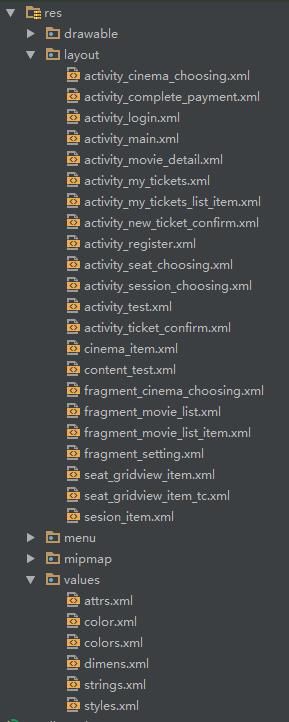
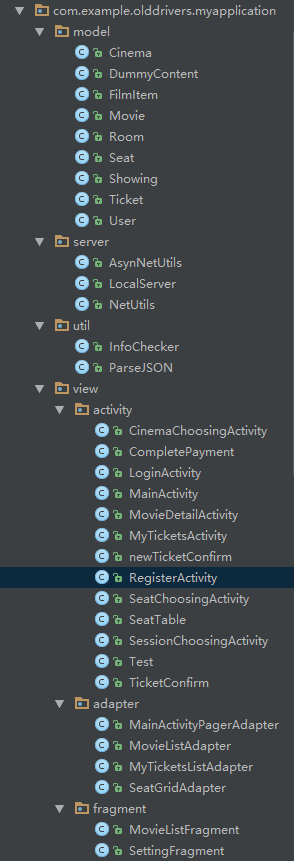
架构设计

本项目使用了MVC框架，因为代码量不算太多，很多数据处理的操作已经封装起来，所以activity并不会很臃肿，所以没有再使用MVP框架。

MVC业务流程

、

项目文件结构



M层：业务逻辑处理，网络请求，json文件处理，输入信息检查，本地信息储存，在项目util和server文件夹中

V层：应用层中处理数据显示的部分，XML布局，显示Model层的数据结果。

C层：在Android中，Activity处理用户交互问题，因此可以认为Activity是控制器，Activity从V视图层读取数据（eg.读取服务器数据），显示到界面（eg.EditText.setText），并向Model发送数据请求（eg.发起网络请求等）。

MVC的优点：

(1)耦合性低。所谓耦合性就是模块代码之间的关联程度。利用MVC框架使得View（视图）层和Model（模型）层可以很好的分离，这样就达到了解耦的目的，所以耦合性低，减少模块代码之间的相互影响。但是在Activity中还是有关于视图UI的显示代码，因此View视图和Activity控制器并不是完全分离的，也就是说一部分View视图和Contronller控制器Activity是绑定在一个类中的。

(2)可扩展性好。由于耦合性低，添加需求，扩展代码就可以减少修改之前的代码，当需要改变UI显示的时候，无需修改Contronller（控制器）Activity的代码和Model（模型）模型中的业务逻辑代码，很好的将业务逻辑和界面显示分离，降低bug的出现率。

(3)模块职责划分明确。主要划分层M,V,C三个模块，利于代码的维护。

Android软件技术：

（1）网络请求的封装:  
1)实现并发的网络请求，在主线程进行网络请求,因为网络请求耗时较长，会报错。  
2)不同环境对应不同的网络请求方式(post/get)。

3)网络请求是异步的，在另外一个线程里面执行

4)网络请求中，把变化的、动态的抽取出来，比如请求参数、请求地址，回调函数；把静态的，不可变的封装起来，不对外暴露，比如相同的请求头、公共的请求参数，公共请求参数的处理、统一错误处理、公共的返回参数。

5)因为HttpClient早就已经不在更新，只适合低版本Android，所以网络请求框架的底层都是HttpURLConnection  
6)没有使用OkHttp等网络库

7)没有实现发送网络请求之前，判断网络连接状况

（2）java单例模式以及Android中单例模式的问题

项目中有一些变量需要储存在全局，比如说登录的用户信息，在登录后到应用结束之前这个信息应该一直被保存下来。

所以我第一个想到的就是使用单例模式，但之前都是c++中使用单例模式，于是上网学习java的单例模式

单例模式是指在一个类使用单例模式，在这个类被创建后，只可能产生一个实例供外部访问，并且提供一个全局的访问点

要求：

(1) 将采用单例设计模式的类的构造方法私有化（采用private修饰），不允许使用者调用构造方法。

(2) 在类其内部产生该类的实例化对象，并将其封装成private static类型，因为只有一个实例。

(3) 定义一个静态方法返回该类的实例。

方法一

单例模式的实现：饿汉式,线程安全 但效率比较低

public class SingletonTest {

// 定义一个私有的构造方法

private SingletonTest() {

}

// 将自身的实例对象设置为一个属性,并加上Static和final修饰符

private static final SingletonTest instance = new SingletonTest();

// 静态方法返回该类的实例

public static SingletonTest getInstancei() {

return instance;

}

}

优点是：写起来比较简单，当类SingletonTest被加载的时候，会初始化static的instanc，不存在多线程同步问题，避免了synchronized所造成的性能问题；  
缺点是：当类SingletonTest被加载的时候，会初始化static的instance，静态变量被创建并分配内存空间，从这以后，这个static的instance对象便一直占着这段内存（即便你还没有用到这个实例），当类被卸载时，静态变量被摧毁，并释放所占有的内存，因此在某些特定条件下会耗费内存。

方法二

饱汉模式  
优点是：写起来比较简单，当类SingletonTest被加载的时候，静态变量static的instance未被创建并分配内存空间，当getInstance方法第一次被调用时，初始化instance变量，并分配内存，因此在某些特定条件下会节约了内存；  
缺点是：并发环境下很可能出现多个SingletonTest实例。

代码因篇幅问题省略了

最佳实现为《Android 源码设计模式》中的方式

public class SingletonTest {  
  
private SingletonTest() {   
}   
  
public static synchronized SingletonTest getInstance() {   
return SingletonHolder.instance;   
}   
  
private static class SingletonHolder{  
private static SingletonTest instance = new SingletonTest();  
}  
}

在第一次使用时创建，效率高，并且线程安全。

但是！在Android中只是这样使用单例模式是会出问题的

如果你在接收数据的Activity中，按下Home键返回桌面，长时间的没有返回你的App。系统有可能会在系统内存不足的时候杀掉进程。当你再从最近程序运行列表进入你的App，系统会默认恢复刚刚离开的状态，直接进入接收数据的Activity中，然后调用各个生命周期方法回调，其中只要运行到从单例模式类取数据行，程序就会弹出空指针NullPointerException异常导致崩溃，或者取到的数据为空。

因为上一次的单例模式的数据已经在系统内存不足，并且程序在后台的时候被清除掉了

查找解决方法如下

还是直接将数据通过intent传递给 Activity 。

使用官方推荐的几种方式将数据持久化到磁盘上，再取数据。

在使用数据的时候总是要对变量的值进行非空检查，这样还是取不到数据

使用EventBus传递数据时采用onSaveInstanceState(Bundle outState)方法保存数据，使用onCreate(Bundle savedInstanceState)等待恢复取值。

这里因为项目比较简单，直接将信息存在sharedpreferences中，这样存在磁盘就不会被清除丢失了，不过这样的缺点是存在安全和隐私泄露的问题。

2）Android中的消息处理

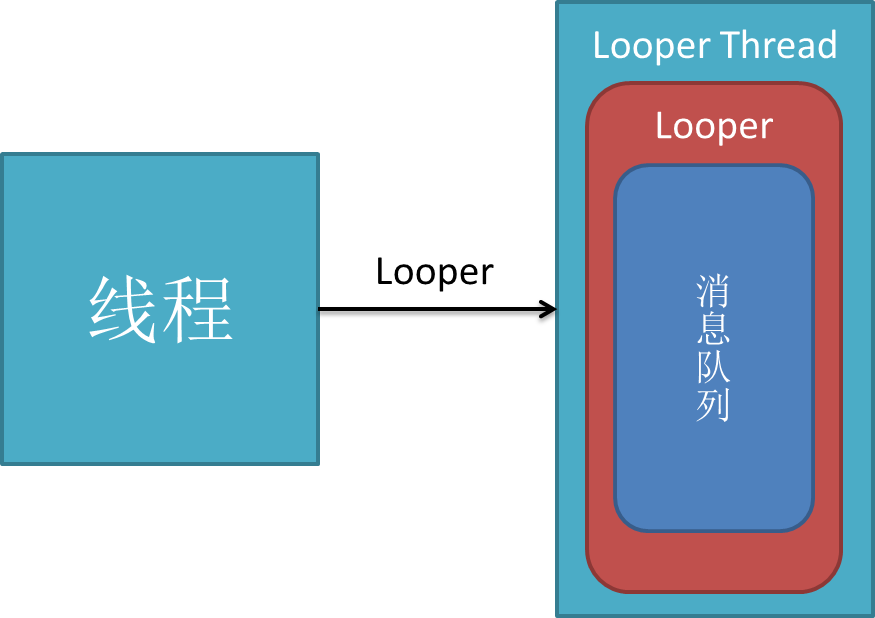
首先是Looper类

Looper可以使一个普通线程变成Looper线程，所谓Looper线程就是循环工作的线程。

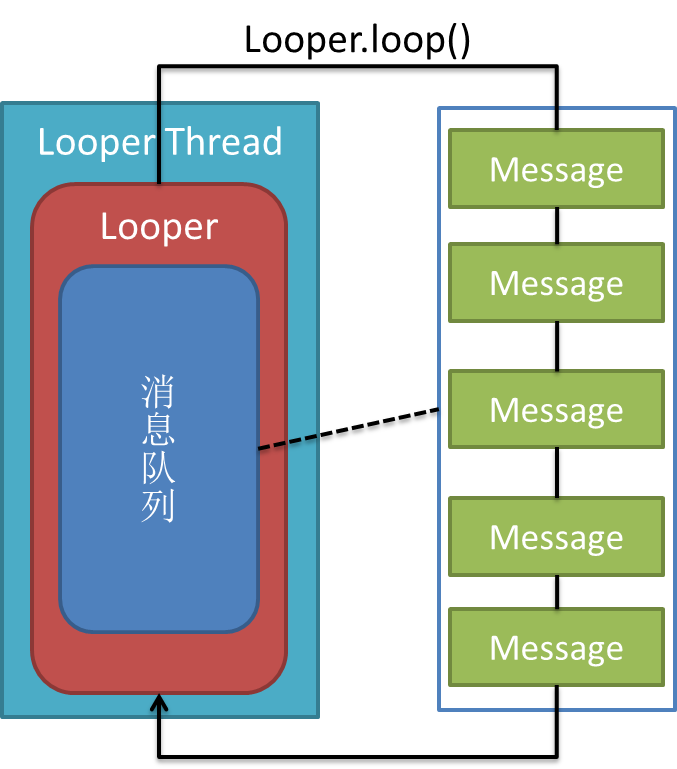
public class LooperThread extends Thread {  
 @Override  
 public void run() {  
 // 将当前线程初始化为Looper线程  
 Looper.prepare();  
   
 // ...其他处理，如实例化handler  
   
 // 开始循环处理消息队列  
 Looper.loop();  
 }  
}

通过上面两行核心代码，你的线程就升级为Looper线程了

Looper.prepare()



Looper.loop()



调用loop方法后，Looper线程就开始真正工作了，它不断从自己的MQ中取出队头的消息(也叫任务)执行。

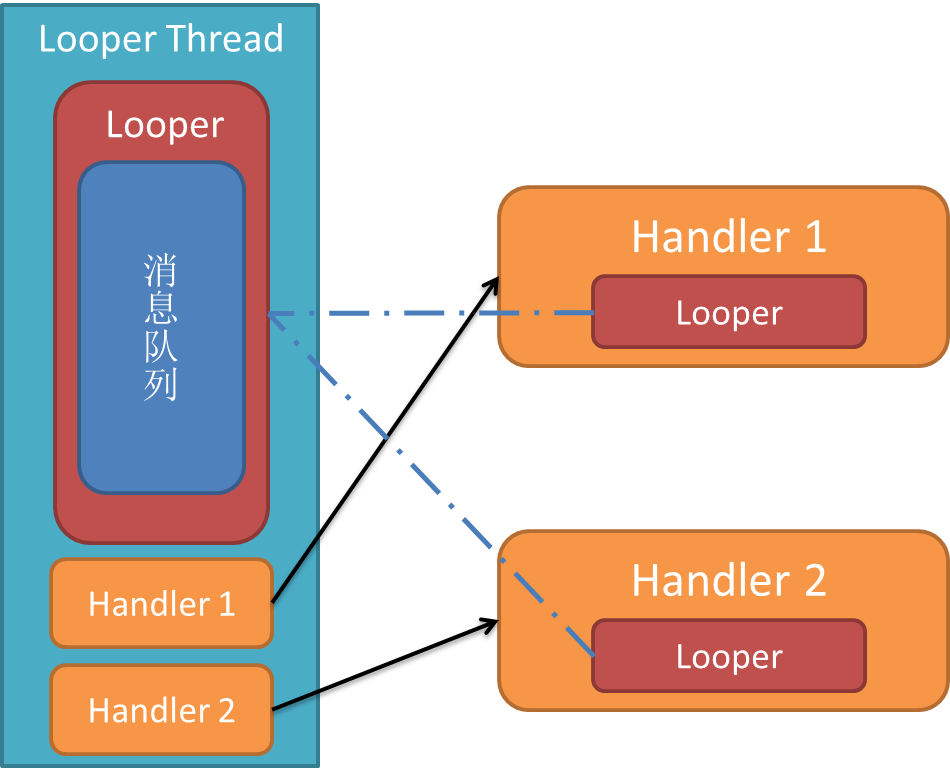
1.每个线程有且最多只能有一个Looper对象，它是一个ThreadLocal

2.Looper内部有一个消息队列，loop()方法调用后线程开始不断从队列中取出消息执行

3.Looper使一个线程变成Looper线程。

如何在消息队列中添加消息呢，就要使用Handler了

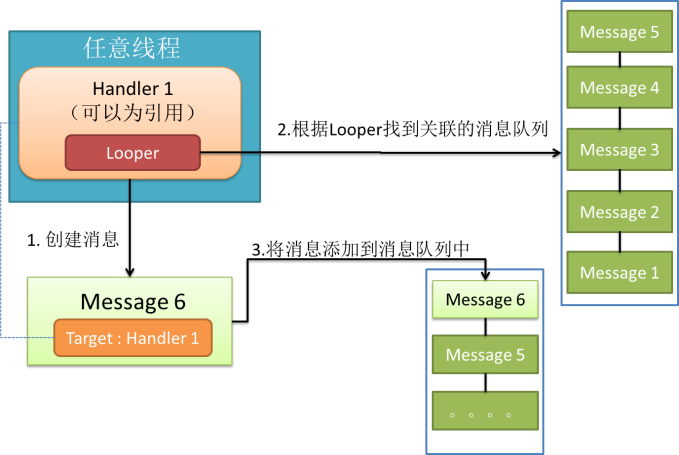
handler扮演了往MQ上添加消息和处理消息的角色（只处理由自己发出的消息），通知MQ它要执行一个任务(sendMessage)，并在loop到自己的时候执行该任务(handleMessage)，整个过程是异步的



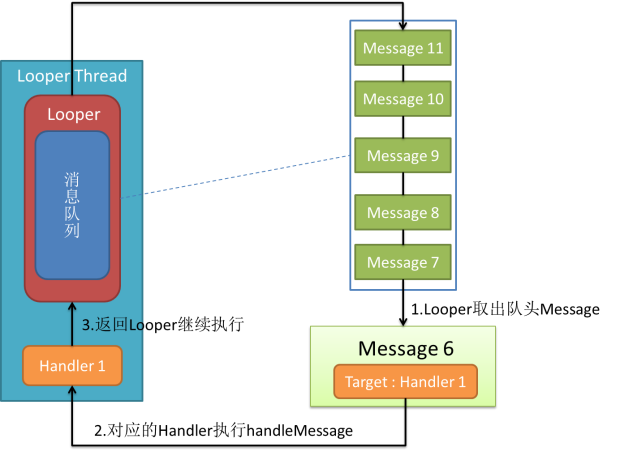
一个线程可以有多个Handler，但是只能有一个Looper！

有了handler之后，我们就可以使用 [post(Runnable)](http://developer.android.com/reference/android/os/Handler.html" \l "post(java.lang.Runnable)), [postAtTime(Runnable, long)](http://developer.android.com/reference/android/os/Handler.html" \l "postAtTime(java.lang.Runnable, long)),[postDelayed(Runnable,long)](http://developer.android.com/reference/android/os/Handler.html" \l "postDelayed(java.lang.Runnable, long)), [sendEmptyMessage(int)](http://developer.android.com/reference/android/os/Handler.html" \l "sendEmptyMessage(int)), [sendMessage(Message)](http://developer.android.com/reference/android/os/Handler.html" \l "sendMessage(android.os.Message)), [sendMessageAtTime(Message, long)](http://developer.android.com/reference/android/os/Handler.html" \l "sendMessageAtTime(android.os.Message, long))和[sendMessageDelayed(Message, long)](http://developer.android.com/reference/android/os/Handler.html" \l "sendMessageDelayed(android.os.Message, long))这些方法向MQ上发送消息了

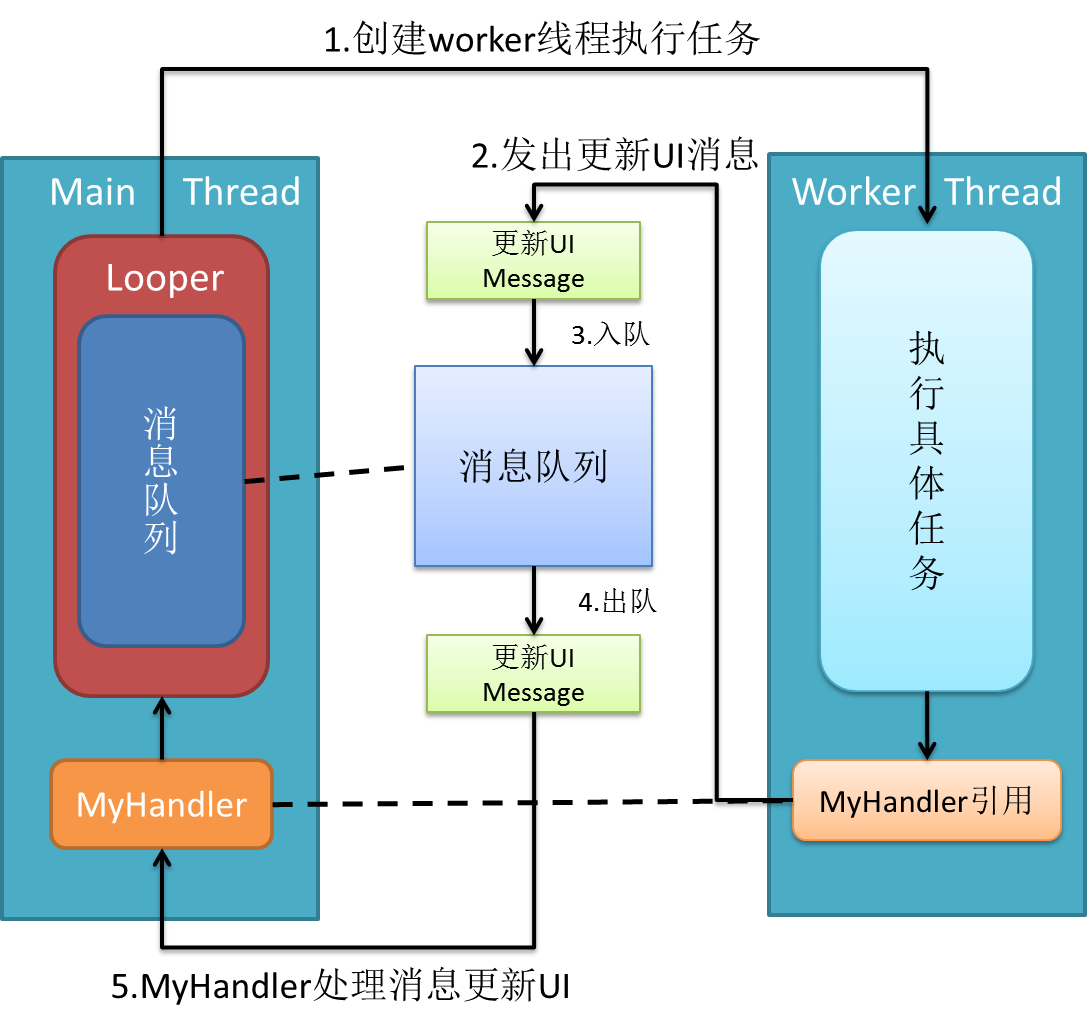
1.handler可以在任意线程发送消息，这些消息会被添加到关联的MQ上。



2.handler是在它关联的looper线程中处理消息的。



这就解决了android最经典的不能在其他非主线程中更新UI的问题。android的主线程也是一个looper线程



利用handler的一个solution就是在activity中创建handler并将其引用传递给worker thread，worker thread执行完任务后使用handler发送消息通知activity更新UI。

这次项目中的网络访问就是这样实现异步操作和更新UI的。