异步请求 内部采用了线程池来处理任务和Handler来发消息给UI线程

Android 3.0以后才有的类

核心线程： CPU核心数

最大线程： CPU核心数 * 2 +1

闲置线程存活时长：30

最大缓存任务列队：128

处理机制：

内部还有一个创建线程的工厂类：

- AsyncTask主要存在以下局限性：
    - 在Android 4.1版本之前，AsyncTask类必须在主线程中加载，这意味着对AsyncTask类的第一次访问必须发生在主线程中；在Android 4.1以及以上版本则不存在这一限制，因为ActivityThread（代表了主线程）的main方法中会自动加载AsyncTask
    - AsyncTask对象必须在主线程中创建
    - AsyncTask对象的execute方法必须在主线程中调用
    - 一个AsyncTask对象只能调用一次execute方法

AsyncTask存在的问题：

1、在Activity中创建的AsyncTask会随着Activity的销毁而销毁。然而事实并非如此。AsyncTask会一直执行, 直到doInBackground()方法执行完毕（如果我们的Activity销毁之前，没有取消AsyncTask，这有可能让我们的AsyncTask崩溃(crash)。因为它想要处理的view已经不存在了。所以，我们总是必须确保在销毁活动之前取消任务。总之，我们使用AsyncTask需要确保AsyncTask正确地取消。）

2、如果AsyncTask被声明为Activity的非静态的内部类，那么AsyncTask会保留一个对创建了AsyncTask的Activity的引用。如果Activity已经被销毁，AsyncTask的后台线程还在执行，它将继续在内存里保留这个引用，导致Activity无法被回收，引起内存泄露。

3、

使用试例：

执行线程：加载数据

执行完线程之后:更新UI

参数类型： <Void, Void, Void>

参数一：要传入的数据类型 对应在 doInBackground(String 方法中

参数二：返回任务的进度更新数据

onProgressUpdate(Integer[] values)

参数三：要返回的数据类型　　onPostExecute(Boolean result)

六个方法：

execute()：执行任务　这个方法如果给传递了参数那么这个参数是在doInBackground()这个方法中接收

doInBackground()：　执行线程：加载数据:run 非UI线程里面

onPreExecute：　执行线程之前：更新UI：run UI线程

onPostExecute：　执行完线程之后:更新UI，run UI线程

publishProgress

onProgressUpdate　更新进度的方法

execute()方法：

```
new AsyncTask<Void, Void, Void>() {
  执行线程之前：更新UI
protected void onPreExecute() {
        比如隐藏控件
};
执行线程：加载数据:run 非UI线程里面
@Override
protected Void doInBackground(Void... params) {
        请求数据
```

相当于：

```
new Thread(){
    @Override
    public void run() {
     请求数据
    }
 }.start();
}
执行完线程之后:更新UI
protected void onPostExecute(Void result) {
        返回请求结果
   };
}.execute(可以有参数);
```

注：这里传入的参数在doInBackground方法

接收

线程池：

任务采用了阻塞的集合来保存

int corePoolSize : 核心线程数

int maximumPoolSize : 最大线程数

long keepAliveTime : 零时线程存活时间

TimeUnit unit : 时间单位　　TimeUnit.SECONDS

BlockingQueue<Runnable> workQueue :　线程对列数　new LinkedBlockingDeque<>(10)

ThreadFactory threadFactory :　线程池工厂　默认：Executors.defaultThreadFactory()

RejectedExecutionHandler handler : 超出线程对列的处理机制

```
ThreadPoolExecutor executor = new ThreadPoolExecutor(5, 8, 1, TimeUnit.SECONDS, new LinkedBlockingDeque<>(10),Executors.defaultThreadFactory(), handler);
```

处理机制有4种： handler

1、当前任务抛异常

2、不做任务处理

3、请主线程来处理

4、将处理最长时间或池中停滞时间最长的弹出去，然后直接处理当前最新的任务

```
RejectedExecutionHandler handler = new ThreadPoolExecutor.AbortPolicy();    //抛异常  默认操作
RejectedExecutionHandler handler1 = new ThreadPoolExecutor.CallerRunsPolicy();   //直接运行
RejectedExecutionHandler handler2 = new ThreadPoolExecutor.DiscardOldestPolicy(); //请主线程处理
RejectedExecutionHandler handler3 = new ThreadPoolExecutor.DiscardPolicy();   //忽略
```

使用线程池的好处：
1、降低资源消耗。重复利用已创建线程，降低线程创建与销毁的资源消耗。
2、提高响应效率。任务到达时，不需等待创建线程就能立即执行。
3、提高线程可管理性。
4、防止服务器过载。内存溢出、CPU耗尽。

用法：
我们通常使用它方便地生产各种类型的线程池，不同类型线程池的创建：
如：**ExecutorService executorService = Executors.*newSingleThreadExecutor*();** 一个线程的线程池
这是一个线程任务类

```
public class ThreadChi implements Runnable{
      public void run(){
            for(int i=0;i<10;i++){

System.out.println(Thread.currentThread().getName()+":"+i);
            }
      }
}
```

1、创建大小不固定的线程池
这是一个主函数中的创建线程池的方式，具有缓冲功能的线程池，系统根据需要创建线程，线程会被缓冲到线程池中
如果线程池大小超过了处理任务所需要的线程
线程池就会回收空闲的线程池，当处理任务增加时，
线程池可以增加线程来处理任务
线程池不会对线程的大小进行限制
线程池的大小依赖于操作系统

```
ExecutorService es=Executors.newCachedThreadPool();
for(int i=0;i<10;i++){
                  ThreadChi tc=new ThreadChi();
                  es.execute(tc);    执行线程任务
```

```
            }
            es.shutdown();
```

2、创建具一个可重用的，有固定数量的线程池

* 每次提交一个任务就提交一个线程，直到线程达到线城池大小，就不会创建新线程了

* 线程池的大小达到最大后达到稳定不变，如果一个线程异常终止，则会创建新的线程

```
ExecutorService es=Executors.newFixedThreadPool(2);
            for(int i=0;i<10;i++){
               ThreadChi tc=new ThreadChi();
               es.execute(tc);
            }
            es.shutdown();
```

3、创建只有一个线程的线程池 ，按照提交顺序执行

```
ExecutorService es=Executors.newSingleThreadExecutor();
      for(int i=0;i<10;i++){
         ThreadChi tc=new ThreadChi();
            es.execute(tc);
      }
            es.shutdown();
```

4、创建定时线程

```
ScheduledExecutorService es=Executors.newScheduledThreadPool(2);
      ThreadChi tc=new ThreadChi();
```

参数1：目标对象　　参数2：隔多长时间开始执行线程　　参数3：执行周期　　参数4：时间单位

```
      es.scheduleAtFixedRate(tc, 3, 1, TimeUnit.MILLISECONDS);
```

其它方法：

Executors：是一个没有继承任何类的方法，其作为一个工厂方法类，用于创建形形色色的线程池等对象

shutdown： 启动线程池的有序关闭过程，其等待已经提交的所有任务完成之后关闭线程池。
当调用该方法后，线程池不再接受其他任务。并且该方法是非阻塞的，不会等待所有任务执行成功后返回。