# Java语言高级特性前言

java知识是作为Android开发的语言基础，虽然现在我们已经推出了kotlin，但是基于以下原因我们还是需要好好牢牢掌握java：

1）SDK还是改成java，kotlin也需要编译成为java运行；

2）目前大量的第三方库和继承与前任的代码都是java所写的；

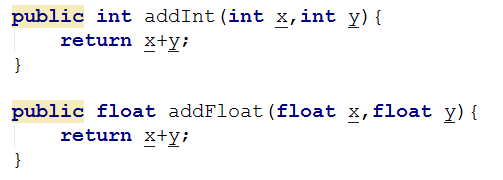
3）Java语言应用不仅仅在Android，就是在后台开发中也是一个最流行的语言；

4）大公司面试都要求我们有扎实的Java语言基础。所以，请大家不要轻视提高自己Java基础的机会，请大家认真学习，做好笔记，争取取得更大的进步。

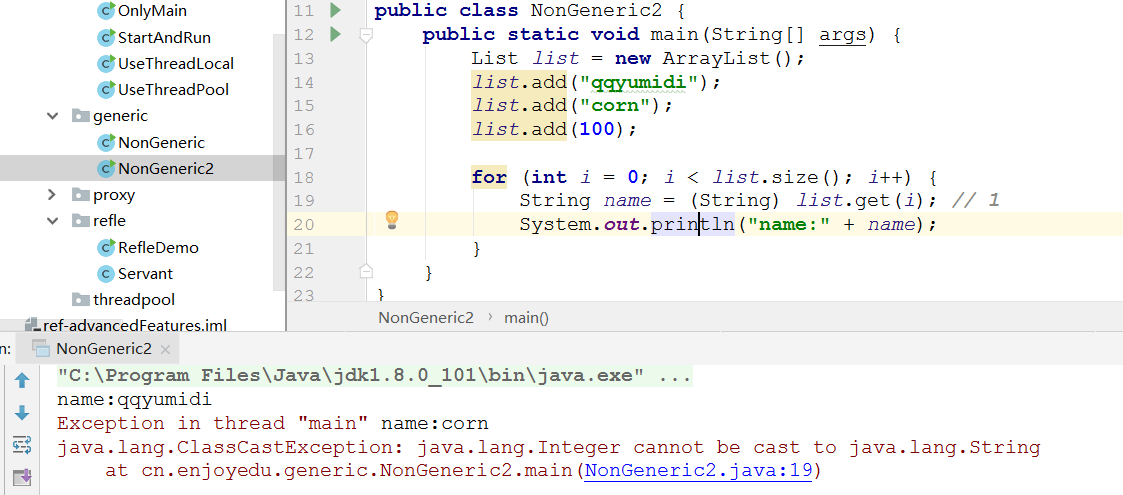
# Java中的泛型

## 为什么我们需要泛型？

通过两段代码我们就可以知道为何我们需要泛型



实际开发中，经常有数值类型求和的需求，例如实现int类型的加法, 有时候还需要实现long类型的求和, 如果还需要double类型的求和，需要重新在重载一个输入是double类型的add方法。



定义了一个List类型的集合，先向其中加入了两个字符串类型的值，随后加入一个Integer类型的值。这是完全允许的，因为此时list默认的类型为Object类型。在之后的循环中，由于忘记了之前在list中也加入了Integer类型的值或其他编码原因，很容易出现类似于//1中的错误。因为编译阶段正常，而运行时会出现“java.lang.ClassCastException”异常。因此，导致此类错误编码过程中不易发现。

 在如上的编码过程中，我们发现主要存在两个问题：

1.当我们将一个对象放入集合中，集合不会记住此对象的类型，当再次从集合中取出此对象时，改对象的编译类型变成了Object类型，但其运行时类型任然为其本身类型。

2.因此，//1处取出集合元素时需要人为的强制类型转化到具体的目标类型，且很容易出现“java.lang.ClassCastException”异常。

所以泛型的好处就是：

* 适用于多种数据类型执行相同的代码
* 泛型中的类型在使用时指定，不需要强制类型转换

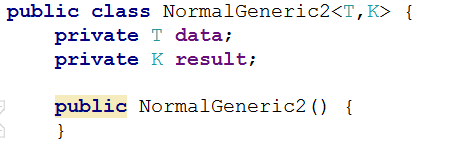
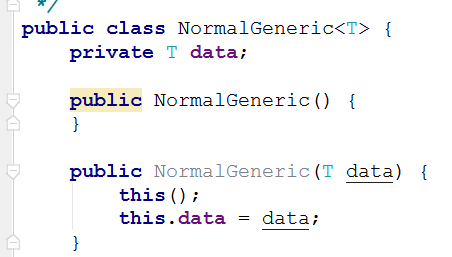
## 泛型类和泛型接口

泛型，即“参数化类型”。一提到参数，最熟悉的就是定义方法时有形参，然后调用此方法时传递实参。那么参数化类型怎么理解呢？

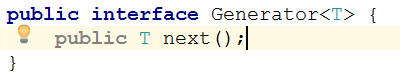
顾名思义，就是将类型由原来的具体的类型参数化，类似于方法中的变量参数，此时类型也定义成参数形式（可以称之为类型形参），然后在使用/调用时传入具体的类型（类型实参）。

泛型的本质是为了参数化类型（在不创建新的类型的情况下，通过泛型指定的不同类型来控制形参具体限制的类型）。也就是说在泛型使用过程中，操作的数据类型被指定为一个参数，这种参数类型可以用在类、接口和方法中，分别被称为泛型类、泛型接口、泛型方法。

引入一个类型变量T（其他大写字母都可以，不过常用的就是T，E，K，V等等），并且用<>括起来，并放在类名的后面。泛型类是允许有多个类型变量的。

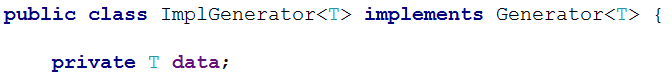


泛型接口与泛型类的定义基本相同。



而实现泛型接口的类，有两种实现方法：

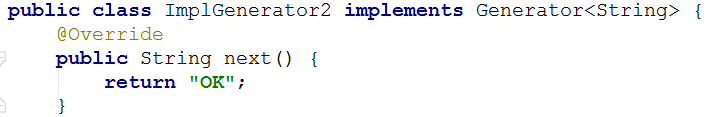
1、未传入泛型实参时：



在new出类的实例时，需要指定具体类型：

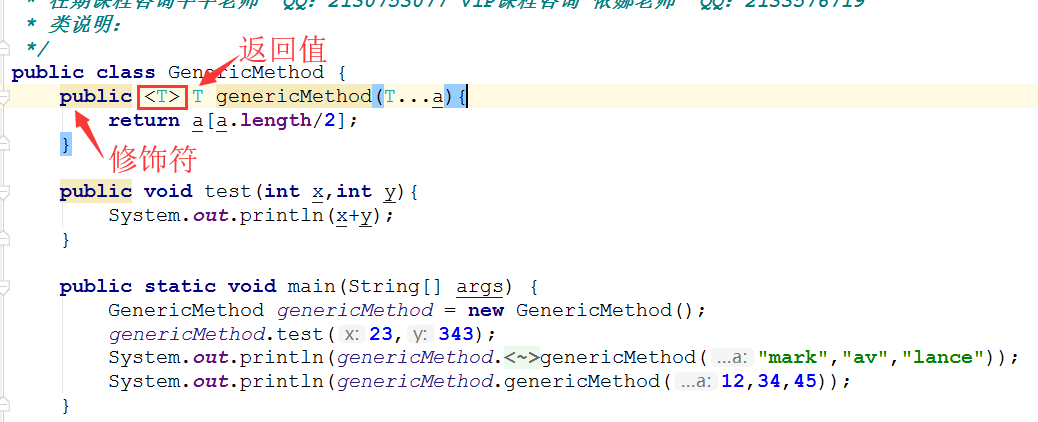


2、传入泛型实参



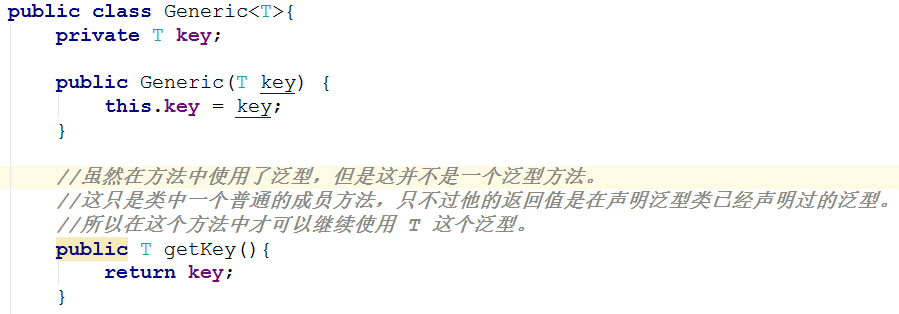
在new出类的实例时，和普通的类没区别。

## 泛型方法

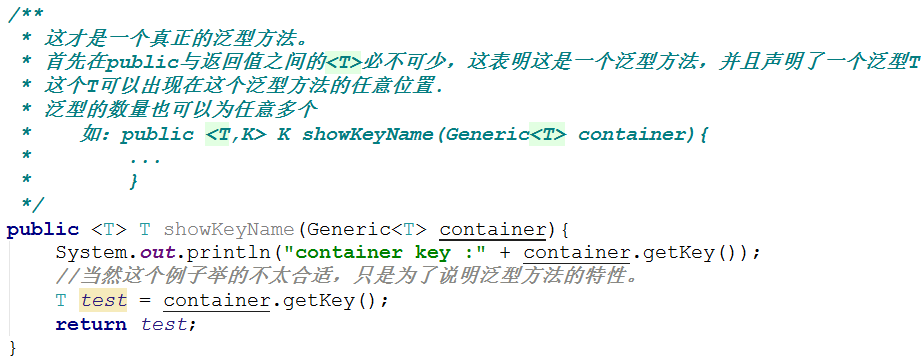
****

泛型方法，是在调用方法的时候指明泛型的具体类型 ，泛型方法可以在任何地方和任何场景中使用，包括普通类和泛型类。注意泛型类中定义的普通方法和泛型方法的区别。

普通方法：

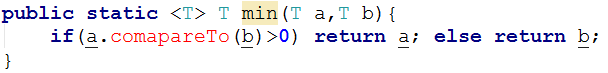


泛型方法

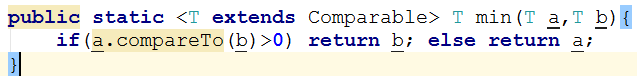


## 限定类型变量

有时候，我们需要对类型变量加以约束，比如计算两个变量的最小，最大值。



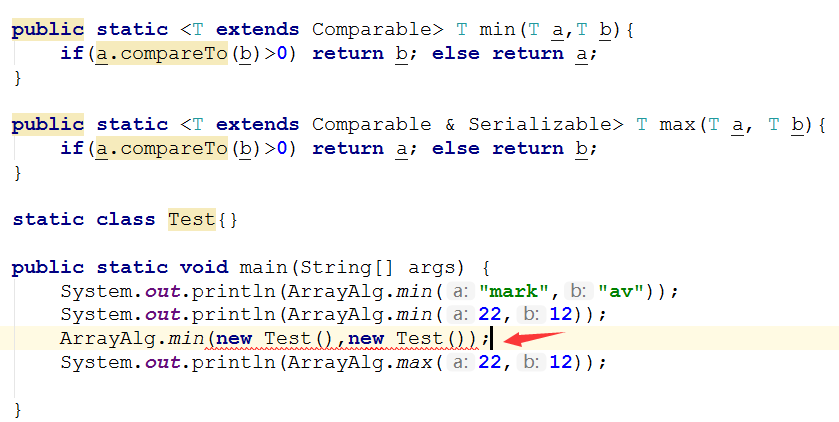
请问，如果确保传入的两个变量一定有compareTo方法？那么解决这个问题的方案就是将T限制为实现了接口Comparable的类



T **extends** Comparable中

T表示应该绑定类型的子类型，Comparable表示绑定类型，子类型和绑定类型可以是类也可以是接口。

如果这个时候，我们试图传入一个没有实现接口Comparable的类的实例，将会发生编译错误。



同时extends左右都允许有多个，如 T,V **extends** Comparable & Serializable

注意限定类型中，只允许有一个类，而且如果有类，这个类必须是限定列表的第一个。

这种类的限定既可以用在泛型方法上也可以用在泛型类上。

## 泛型中的约束和局限性

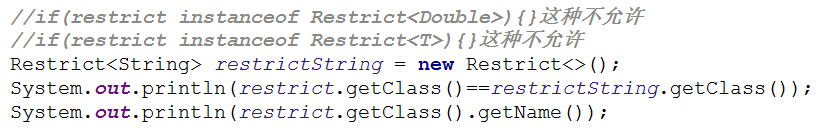
现在我们有泛型类



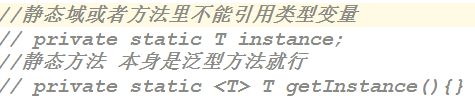
### 不能用基本类型实例化类型参数



### 运行时类型查询只适用于原始类型

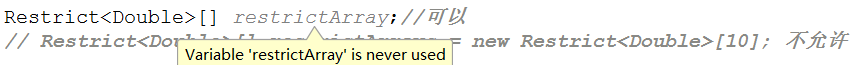


### 泛型类的静态上下文中类型变量失效

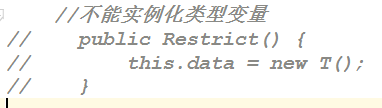


不能在静态域或方法中引用类型变量。因为泛型是要在对象创建的时候才知道是什么类型的，而对象创建的代码执行先后顺序是static的部分，然后才是构造函数等等。所以在对象初始化之前static的部分已经执行了，如果你在静态部分引用的泛型，那么毫无疑问虚拟机根本不知道是什么东西，因为这个时候类还没有初始化。

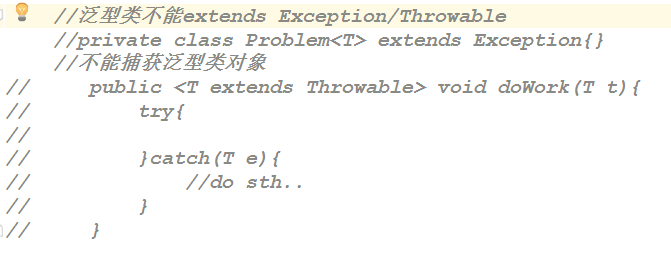
### 不能创建参数化类型的数组



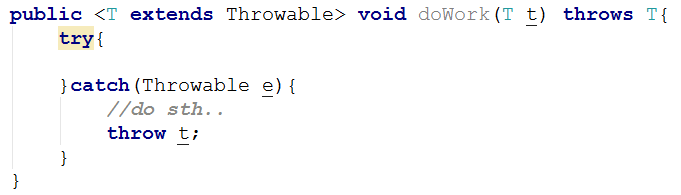
### 不能实例化类型变量



### 不能捕获泛型类的实例



但是这样可以：



## 泛型类型的继承规则

现在我们有一个类和子类



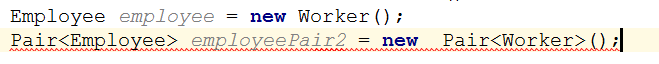


有一个泛型类

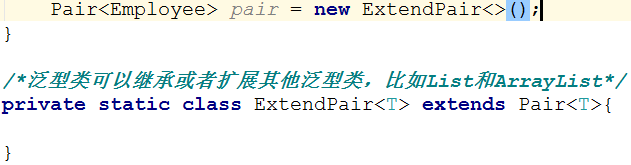


请问Pair<Employee>和Pair<Worker>是继承关系吗？

答案：不是，他们之间没有什么关系

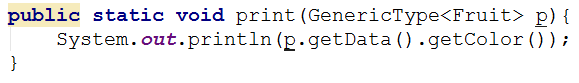


但是泛型类可以继承或者扩展其他泛型类，比如List和ArrayList



## 通配符类型

正是因为前面所述的，Pair<Employee>和Pair<Worker>没有任何关系，如果我们有一个泛型类和一个方法





现在我们有继承关系的类

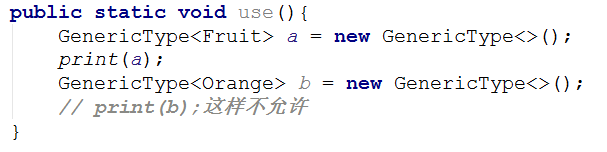








则会产生这种情况：



为解决这个问题，于是提出了一个通配符类型 ?

有两种使用方式：

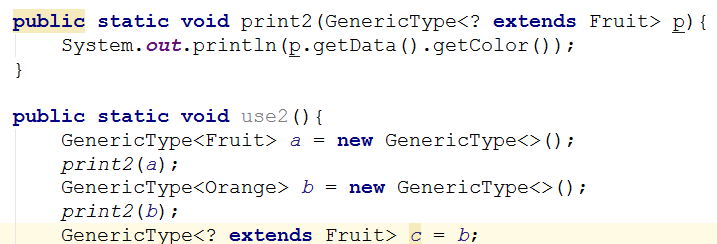
？ extends X 表示类型的上界，类型参数是X的子类

？ super X 表示类型的下界，类型参数是X的超类

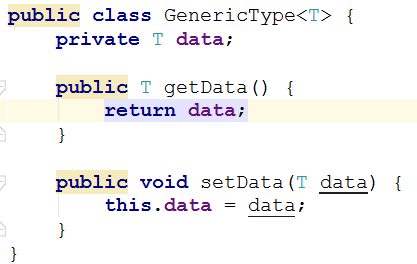
这两种 方式从名字上来看，特别是super，很有迷惑性，下面我们来仔细辨析这两种方法。

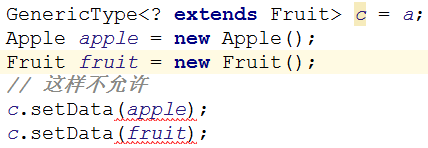
### ？ extends X

表示传递给方法的参数，必须是X的子类（包括X本身）



但是对泛型类GenericType来说，如果其中提供了get和set类型参数变量的方法的话，set方法是不允许被调用的，会出现编译错误





get方法则没问题，会返回一个Fruit类型的值。



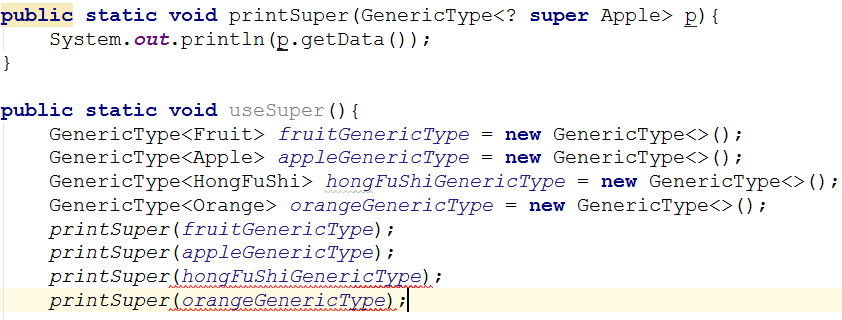
为何？

道理很简单，？ extends X 表示类型的上界，类型参数是X的子类，那么可以肯定的说，get方法返回的一定是个X（不管是X或者X的子类）编译器是可以确定知道的。但是set方法只知道传入的是个X，至于具体是X的那个子类，不知道。

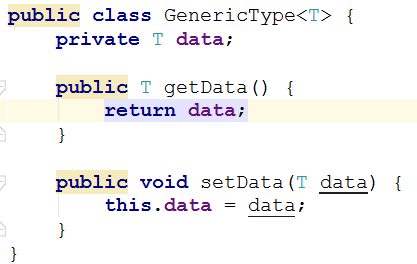
总结：主要用于安全地访问数据，可以访问X及其子类型，并且不能写入非null的数据。

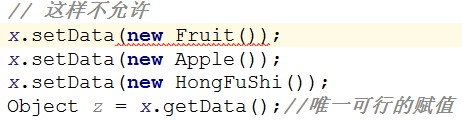
### ？ super X

表示传递给方法的参数，必须是X的超类（包括X本身）



但是对泛型类GenericType来说，如果其中提供了get和set类型参数变量的方法的话，set方法可以被调用的，且能传入的参数只能是X或者X的子类





get方法只会返回一个Object类型的值。



为何？

？ super X 表示类型的下界，类型参数是X的超类（包括X本身），那么可以肯定的说，get方法返回的一定是个X的超类，那么到底是哪个超类？不知道，但是可以肯定的说，Object一定是它的超类，所以get方法返回Object。编译器是可以确定知道的。对于set方法来说，编译器不知道它需要的确切类型，但是X和X的子类可以安全的转型为X。

总结：主要用于安全地写入数据，可以写入X及其子类型。

### 无限定的通配符 ?

表示对类型没有什么限制，可以把？看成所有类型的父类，如Pair< ?>；

比如：

ArrayList<T> al=new ArrayList<T>(); 指定集合元素只能是T类型

ArrayList<?> al=new ArrayList<?>();集合元素可以是任意类型，这种没有意义，一般是方法中，只是为了说明用法。

在使用上：

？ getFirst() ： 返回值只能赋给 Object，；

void setFirst(?) ： setFirst 方法不能被调用， 甚至不能用 Object 调用；

## 虚拟机是如何实现泛型的？

泛型思想早在C++语言的模板（Template）中就开始生根发芽，在Java语言处于还没有出现泛型的版本时，只能通过Object是所有类型的父类和类型强制转换两个特点的配合来实现类型泛化。，由于Java语言里面所有的类型都继承于java.lang.Object，所以Object转型成任何对象都是有可能的。但是也因为有无限的可能性，就只有程序员和运行期的虚拟机才知道这个Object到底是个什么类型的对象。在编译期间，编译器无法检查这个Object的强制转型是否成功，如果仅仅依赖程序员去保障这项操作的正确性，许多ClassCastException的风险就会转嫁到程序运行期之中。

泛型技术在C#和Java之中的使用方式看似相同，但实现上却有着根本性的分歧，C#里面泛型无论在程序源码中、编译后的IL中（Intermediate Language，中间语言，这时候泛型是一个占位符），或是运行期的CLR中，都是切实存在的，List＜int＞与List＜String＞就是两个不同的类型，它们在系统运行期生成，有自己的虚方法表和类型数据，这种实现称为类型膨胀，基于这种方法实现的泛型称为真实泛型。

Java语言中的泛型则不一样，它只在程序源码中存在，在编译后的字节码文件中，就已经替换为原来的原生类型（Raw Type，也称为裸类型）了，并且在相应的地方插入了强制转型代码，因此，对于运行期的Java语言来说，ArrayList＜int＞与ArrayList＜String＞就是同一个类，所以泛型技术实际上是Java语言的一颗语法糖，Java语言中的泛型实现方法称为类型擦除，基于这种方法实现的泛型称为伪泛型。

# 反射原理及动态代理模式

## 反射（Reflect）

反射之中包含了一个「反」字，所以了解反射我们先从「正」开始。

一般情况下，我们使用某个类时必定知道它是什么类，是用来做什么的。于是我们直接对这个类进行实例化，之后使用这个类对象进行操作。

反射则是一开始并不知道我要初始化的类对象是什么，自然也无法使用 new 关键字来创建对象了。这时候，我们使用 JDK 提供的反射 API 进行反射调用。**反射就是在运行时才知道要操作的类是什么，并且可以在运行时获取类的完整构造，并调用对应的方法。**

Reflection（反射）是Java被视为动态语言的关键，反射机制允许程序在执行期借助于Reflection API取得任何类的內部信息，并能直接操作任意对象的内部属性及方法。

Java反射机制主要提供了以下功能：

* 在运行时构造任意一个类的对象
* 在运行时获取任意一个类所具有的成员变量和方法
* 在运行时调用任意一个对象的方法（属性）

Java 是一门面向对象的语言。在面向对象的世界里，万事万物皆对象，既然万事万物皆对象，那么我们的类是不是对象呢？我们写的每一个类都可以看成一个对象，是 java.lang.Class 类的对象。每一个类对应的Class放在哪里呢？当我们写完一个类的Java文件，编译成class文件的时候，编译器都会将这个类的对应的class对象放在class文件的末尾。里面都保存了些什么？大家可以理解保存了类的元数据信息，一个类的元数据信息包括什么？有哪些属性，方法，构造器，实现了哪些接口等等，那么这些信息在Java里都有对应的类来表示。

### Class类

**Class是一个类，封装了当前对象所对应的类的信息**

一个类中有属性，方法，构造器等，比如说有一个Person类，一个Order类，一个Book类，这些都是不同的类，现在需要一个类，用来描述类，这就是Class，它应该有类名，属性，方法，构造器等。Class是用来描述类的类。

Class类是一个对象照镜子的结果，对象可以看到自己有哪些属性，方法，构造器，实现了哪些接口等等

对于每个类而言，JRE 都为其保留一个不变的 Class 类型的对象。一个 Class 对象包含了特定某个类的有关信息。

对象只能由系统建立对象，一个类（而不是一个对象）在 JVM 中只会有一个Class实例

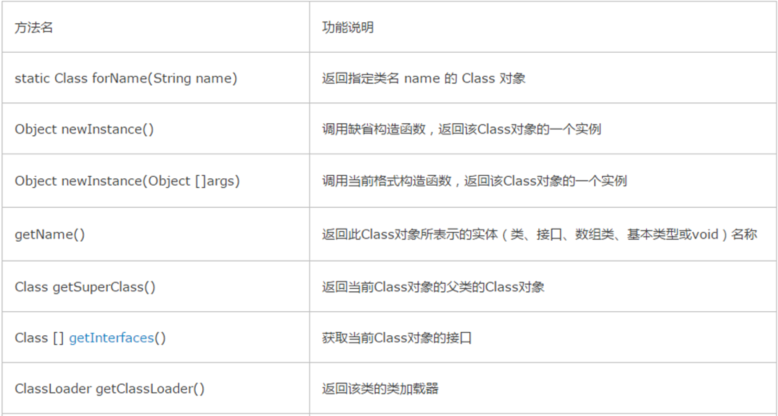
获取Class对象的三种方式

　　1.通过类名获取 类名.class

　　2.通过对象获取 对象名.getClass()

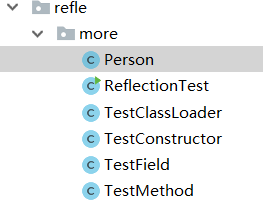
　　3.通过全类名获取 Class.forName(全类名)

Class类的常用方法



### 类加载器、构造器、Method、Field

参见包cn.enjoyedu.refle.more下对应的类



## 动态代理

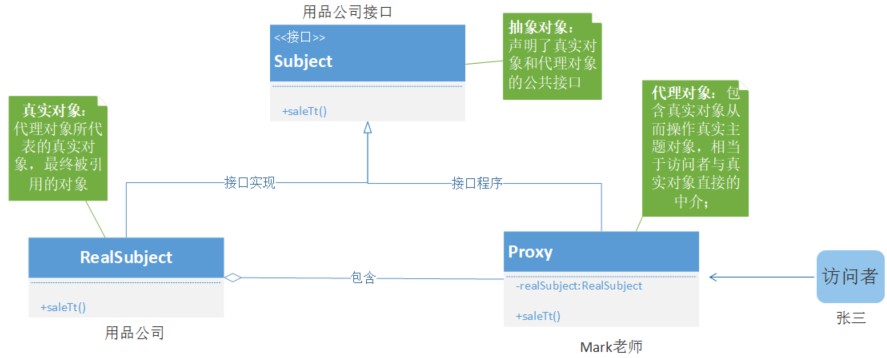
### 代理模式和静态代理

代理模式给某一个对象提供一个代理对象，并由代理对象控制对原对象的引用。通俗的来讲代理模式就是我们生活中常见的中介。

举个例子来说明：张三想买某种用品，虽然他可以自己去找，但是这确实太浪费时间和精力了，或者不好意思去买。于是张三就通过中介Mark来买，Mark来帮张三，张三只是负责选择自己喜欢的的size，然后付钱就可以了。

目的：（1）通过引入代理对象的方式来间接访问目标对象，防止直接访问目标对象给系统带来的不必要复杂性； （2）通过代理对象对原有的业务增强；

代理模式一般会有三个角色：



**抽象角色**：指代理角色和真实角色对外提供的公共方法，一般为一个接口

**真实角色**：需要实现抽象角色接口，定义了真实角色所要实现的业务逻辑，以便供代理角色调用。也就是真正的业务逻辑在此。

**代理角色**：需要实现抽象角色接口，是真实角色的代理，通过真实角色的业务逻辑方法来实现抽象方法，并可以附加自己的操作。将统一的流程控制都放到代理角色中处理！

而访问者不再访问真实角色，而是去访问代理角色。

静态代理在使用时,需要定义接口或者父类,被代理对象与代理对象一起实现相同的接口或者是继承相同父类。一般来说，被代理对象和代理对象是一对一的关系，当然一个代理对象对应多个被代理对象也是可以的。

静态代理，一对一则会出现时静态代理对象量多、代码量大，从而导致代码复杂，可维护性差的问题，一对多则代理对象会出现扩展能力差的问题。

### 动态代理

是指在使用时再创建代理类和实例

**优点**

只需要1个动态代理类就可以解决创建多个静态代理的问题，避免重复、多余代码

更强的灵活性

**缺点**

效率低，相比静态代理中 直接调用目标对象方法，动态代理则需要先通过Java反射机制 从而 间接调用目标对象方法

应用场景局限，因为 Java 的单继承特性（每个代理类都继承了 Proxy 类），即只能针对接口 创建 代理类，不能针对类创建代理类。

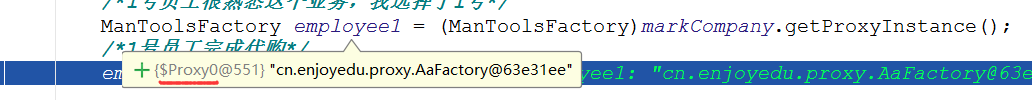
在java的动态代理机制中，有两个重要的类或接口，一个是InvocationHandler接口、另一个则是 Proxy类，这个类和接口是实现我们动态代理所必须用到的。

InvocationHandler接口是给动态代理类实现的，负责处理被代理对象的操作的，而Proxy是用来创建动态代理类实例对象的，因为只有得到了这个对象我们才能调用那些需要代理的方法。

具体使用，参见包cn.enjoyedu.proxy.dynamic下MarkCompany类

#### 动态代理实现原理

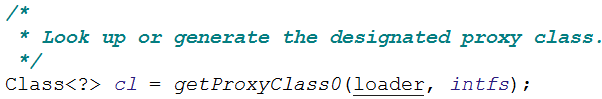
通过调试模式我们发现，动态代理里，代理类的类名是这样的：



这个代理类为何是这个名字？它是如何执行被代理对象的相关方法呢？我们在java文件编译后的目录里其实找不到这个名为$Proxy0的class文件的。

观察Proxy.*newProxyInstance*方法，与创建对象有关的代码主要有：

获得代理类的class对象：



获得代理类的构造器：



创建代理类的实例

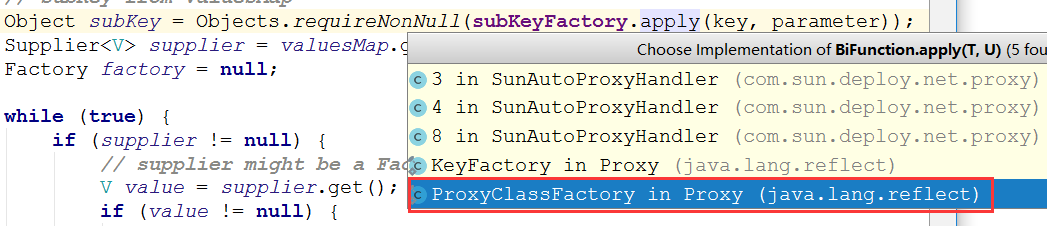


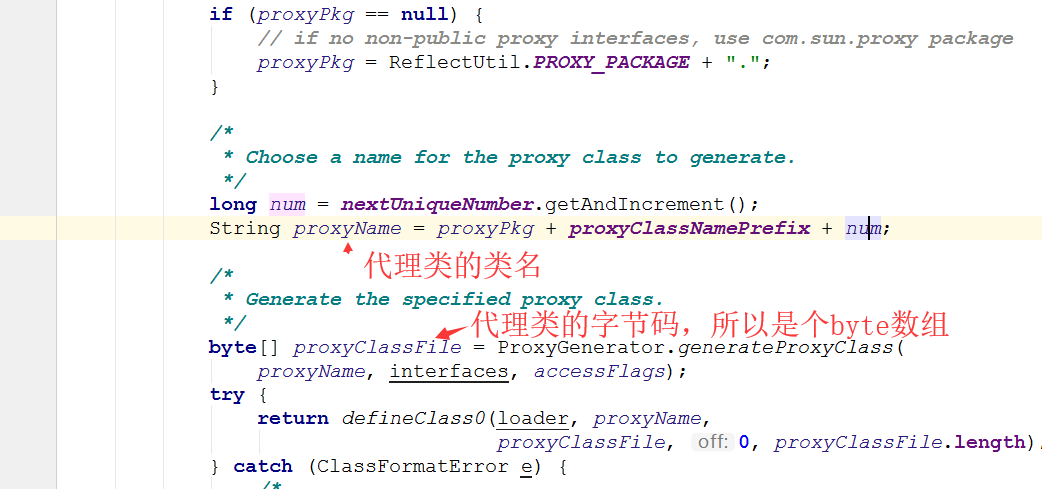
看来其中的关键点就是如何获得代理类的class对象，我们进入*getProxyClass0*方法，进而进入***proxyClassCache***.get方法，通过这个这个方法所在的类名，我们可以推测，JDK内部使用了某种机制缓存了我们的代理类的class对象，同时get方法接受的参数是被代理类的类加载器和类实现的的接口。

在这个get方法中，除去和缓存相关的操作，同时用到了被代理类的类加载器和类实现的的接口这两个参数的是



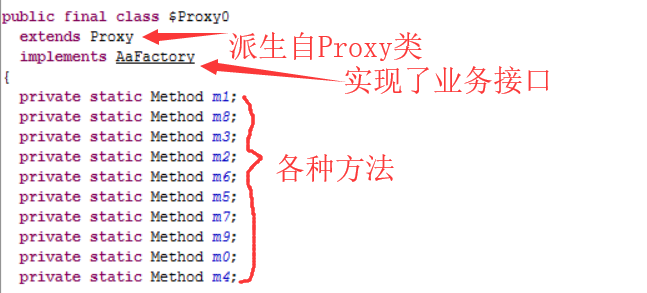
我们再进入这个方法的实现，



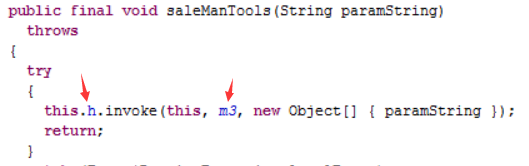


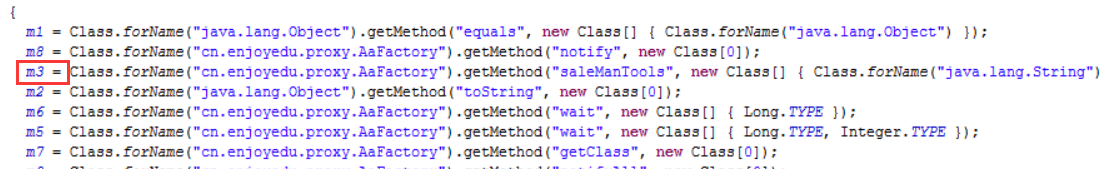
而最终生成代理类的class对象是*defineClass0*方法，但是这个方法是个native方法，所以我们不去也无法深究它，但是通过这个方法的参数我们可以明显看到它接收了上面所生成的byte数组。

而我们通过ProxyUtils，这个自己写的工具类，将这个byte数组写入文件，我们并反编译，我们将会看到

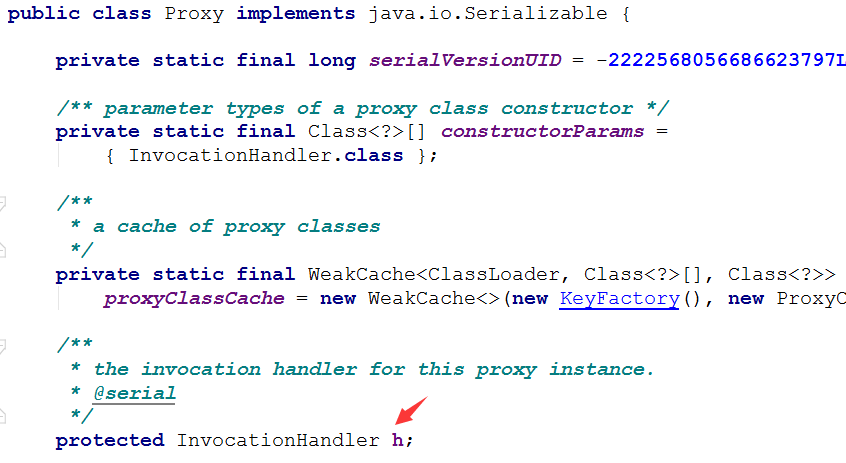


同时我们还会看到其中实现了业务接口的方法





而h则来自派生类Proxy中



这个h的实例来自哪里？不就是我们在创建代理类的实例时传入的吗？

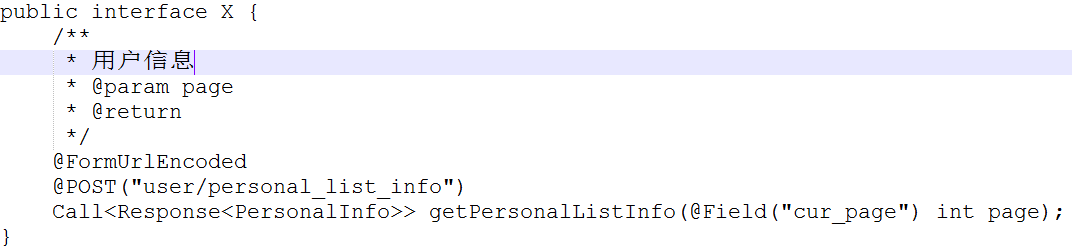


#### 使用了动态代理的`Retrofit

Retrofit简单的说就是一个网络请求的适配器，它将一个基本的Java接口通过动态代理的方式翻译成一个HTTP请求，并通过OkHttp去发送请求。此外它还具有强大的可扩展性，支持各种格式转换以及RxJava。我们基于Retrofit2解析。

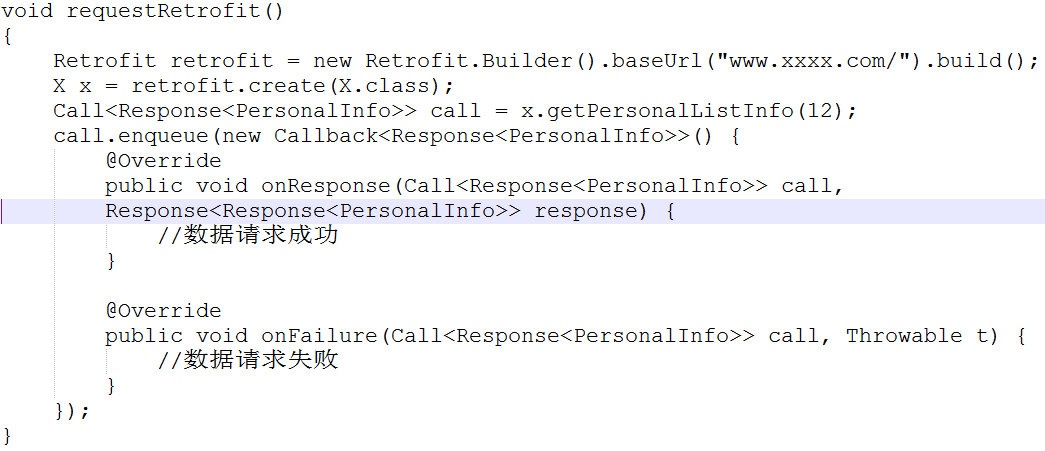
先定义一个名为X的java接口，当然里面有各种注解。

@FormUrlEncoded注解表示from表单，另外还有@Multipart等注解。@POST表示post请求，此外还可以使用@GET请求



然后如何使用的呢？

首先将域名传入构造一个Retrofit，然后通过retrofit中的create方法传入一个Java接口并得到一个x(当然x这个对象是经过处理了的)调用getPersonalListInfo(12)然后返回一个Call，最后这个Call调用了enqueue方法去异步请求http，这就是一个基本的Retrofit的网络请求。Retrofit2中Call接口的默认实现是OkHttpCall，它默认使用OkHttp3作为底层http请求client。

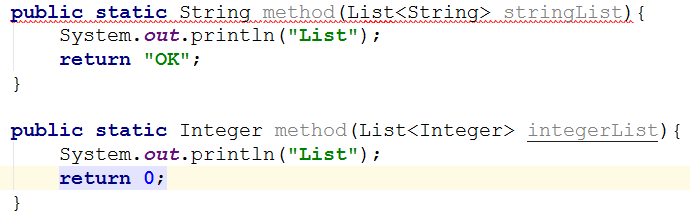


我们只定义了一个接口X，并没有实现这个接口，那么它是如何工作的呢？我们看看create方法的实现。

create()方法是个泛型方法，调用它时会返回一个范型T的对象，我们这里类型是X接口，在内部实现上，很明显了使用了动态代理返回了一个X的代理类。当调用X内部方法的时候，会调用invoke方法。invoke方法内则通过内部一系列的封装最后返回一个Call对象。



将一段Java代码编译成Class文件，然后再用字节码反编译工具进行反编译后，将会发现泛型都不见了，程序又变回了Java泛型出现之前的写法，泛型类型都变回了原生类型



上面这段代码是不能被编译的，因为参数List＜Integer＞和List＜String＞编译之后都被擦除了，变成了一样的原生类型List＜E＞，擦除动作导致这两种方法的特征签名变得一模一样。

由于Java泛型的引入，各种场景（虚拟机解析、反射等）下的方法调用都可能对原有的基础产生影响和新的需求，如在泛型类中如何获取传入的参数化类型等。因此，JCP组织对虚拟机规范做出了相应的修改，引入了诸如Signature、LocalVariableTypeTable等新的属性用于解决伴随泛型而来的参数类型的识别问题，Signature是其中最重要的一项属性，它的作用就是存储一个方法在字节码层面的特征签名[3]，这个属性中保存的参数类型并不是原生类型，而是包括了参数化类型的信息。修改后的虚拟机规范要求所有能识别49.0以上版本的Class文件的虚拟机都要能正确地识别Signature参数。

另外，从Signature属性的出现我们还可以得出结论，擦除法所谓的擦除，仅仅是对方法的Code属性中的字节码进行擦除，实际上元数据中还是保留了泛型信息，这也是我们能通过反射手段取得参数化类型的根本依据。

# 多线程与Android线程性能优化

## 基础概念

### CPU核心数和线程数的关系

多核心:也指单芯片多处理器( Chip Multiprocessors,简称CMP),CMP是由美国斯坦福大学提出的,其思想是将大规模并行处理器中的SMP(对称多处理器)集成到同一芯片内,各个处理器并行执行不同的进程。这种依靠多个CPU同时并行地运行程序是实现超高速计算的一个重要方向,称为并行处理

多线程: Simultaneous Multithreading.简称SMT.SMT可通过复制处理器上的结构状态,让同一个处理器上的多个线程同步执行并共享处理器的执行资源可最大限度地实现宽发射、乱序的超标量处理,提高处理器运算部件的利用率,缓和由于数据相关或 Cache未命中带来的访问内存延时。

核心数、线程数:目前主流CPU有双核、三核和四核,六核也在2010年发布。增加核心数目就是为了增加线程数,因为操作系统是通过线程来执行任务的,一般情况下它们是1:1对应关系,也就是说四核CPU一般拥有四个线程。但 Intel引入超线程技术后,使核心数与线程数形成1:2的关系

### CPU时间片轮转机制

我们平时在开发的时候，感觉并没有受cpu核心数的限制，想启动线程就启动线程，哪怕是在单核CPU上，为什么？这是因为操作系统提供了一种CPU时间片轮转机制。

时间片轮转调度是一种最古老、最简单、最公平且使用最广的算法,又称RR调度。每个进程被分配一个时间段,称作它的时间片,即该进程允许运行的时间。

百度百科对CPU时间片轮转机制原理解释如下:

如果在时间片结束时进程还在运行,则CPU将被剥夺并分配给另一个进程。如果进程在时间片结束前阻塞或结来,则CPU当即进行切换。调度程序所要做的就是维护一张就绪进程列表,当进程用完它的时间片后,它被移到队列的末尾

时间片轮转调度中唯一有趣的一点是时间片的长度。从一个进程切换到另一个进程是需要定时间的,包括保存和装入寄存器值及内存映像,更新各种表格和队列等。假如进程切( processwitch),有时称为上下文切换( context switch),需要5ms,再假设时间片设为20ms,则在做完20ms有用的工作之后,CPU将花费5ms来进行进程切换。CPU时间的20%被浪费在了管理开销上了。

为了提高CPU效率,我们可以将时间片设为5000ms。这时浪费的时间只有0.1%。但考虑到在一个分时系统中,如果有10个交互用户几乎同时按下回车键,将发生什么情况?假设所有其他进程都用足它们的时间片的话,最后一个不幸的进程不得不等待5s才获得运行机会。多数用户无法忍受一条简短命令要5才能做出响应,同样的问题在一台支持多道程序的个人计算机上也会发

结论可以归结如下:时间片设得太短会导致过多的进程切换,降低了CPU效率:而设得太长又可能引起对短的交互请求的响应变差。将时间片设为100ms通常是一个比较合理的折衷。

在CPU死机的情况下,其实大家不难发现当运行一个程序的时候把CPU给弄到了100%再不重启电脑的情况下,其实我们还是有机会把它KⅢ掉的,我想也正是因为这种机制的缘故。

### 什么是进程和线程

**进程是程序运行资源分配的最小单位**

进程是操作系统进行资源分配的最小单位,其中资源包括:CPU、内存空间、磁盘10等,同一进程中的多条线程共享该进程中的全部系统资源,而进程和进程之间是相互独立的。进程是具有一定独立功能的程序关于某个数据集合上的一次运行活动,进程是系统进行资源分配和调度的一个独立单位。

进程是程序在计算机上的一次执行活动。当你运行一个程序,你就启动了一个进程。显然,程序是死的、静态的,进程是活的、动态的。进程可以分为系统进程和用户进程。凡是用于完成操作系统的各种功能的进程就是系统进程,它们就是处于运行状态下的操作系统本身,用户进程就是所有由你启动的进程。

**线程是cPU调度的最小单位,必须依赖于进程而存在**

线程是进程的一个实体,是CPU调度和分派的基本单位,它是比进程更小的、能独立运行的基本单位。线程自己基本上不拥有系统资源,只拥有一点在运行中必不可少的资源(如程序计数器,一组寄存器和栈),但是它可与同属一个进程的其他的线程共享进程所拥有的全部资源。

**线程无处不在**

任何一个程序都必须要创建线程,特别是Java不管任何程序都必须启动一个main函数的主线程; Java Web开发里面的定时任务、定时器、JSP和 Servlet、异步消息处理机制,远程访问接口RM等,任何一个监听事件, onclick的触发事件等都离不开线程和并发的知识。

### 澄清并行和并发

我们举个例子,如果有条高速公路A上面并排有8条车道,那么最大的**并行**车辆就是8辆此条高速公路A同时并排行走的车辆小于等于8辆的时候,车辆就可以并行运行。CPU也是这个原理,一个CPU相当于一个高速公路A,核心数或者线程数就相当于并排可以通行的车道;而多个CPU就相当于并排有多条高速公路,而每个高速公路并排有多个车道。

当谈论**并发**的时候一定要加个单位时间,也就是说单位时间内并发量是多少?离开了单位时间其实是没有意义的。

俗话说,一心不能二用,这对计算机也一样,原则上一个CPU只能分配给一个进程,以便运行这个进程。我们通常使用的计算机中只有一个CPU,也就是说只有一颗心,要让它一心多用同时运行多个进程,就必须使用并发技术。实现并发技术相当复杂,最容易理解的是“时间片轮转进程调度算法”。

综合来说：

并发:指应用能够交替执行不同的任务,比如单CPU核心下执行多线程并非是同时执行多个任务,如果你开两个线程执行,就是在你几乎不可能察觉到的速度不断去切换这两个任务,已达到"同时执行效果",其实并不是的,只是计算机的速度太快,我们无法察觉到而已.

并行:指应用能够同时执行不同的任务,例:吃饭的时候可以边吃饭边打电话,这两件事情可以同时执行

两者区别:一个是交替执行,一个是同时执行.

### 高并发编程的意义、好处和注意事项

由于多核多线程的CPU的诞生,多线程、高并发的编程越来越受重视和关注。多线程可以给程序带来如下好处。

(1)充分利用CPU的资源

从上面的CPU的介绍,可以看的出来,现在市面上没有CPU的内核不使用多线程并发机制的,特别是服务器还不止一个CPU,如果还是使用单线程的技术做思路,明显就out了。因为程序的基本调度单元是线程,并且一个线程也只能在一个CPU的一个核的一个线程跑,如果你是个i3的CPU的话,最差也是双核心4线程的运算能力:如果是一个线程的程序的话,那是要浪费3/4的CPU性能:如果设计一个多线程的程序的话,那它就可以同时在多个CPU的多个核的多个线程上跑,可以充分地利用CPU,减少CPU的空闲时间,发挥它的运算能力,提高并发量。

就像我们平时坐地铁一样,很多人坐长线地铁的时候都在认真看书,而不是为了坐地铁而坐地铁,到家了再去看书,这样你的时间就相当于有了两倍。这就是为什么有些人时间很充裕,而有些人老是说没时间的一个原因,工作也是这样,有的时候可以并发地去做几件事情,充分利用我们的时间,CPU也是一样,也要充分利用。

(2)加快响应用户的时间

比如我们经常用的迅雷下载,都喜欢多开几个线程去下载,谁都不愿意用一个线程去下载,为什么呢?答案很简单,就是多个线程下载快啊。

我们在做程序开发的时候更应该如此,特别是我们做互联网项目,网页的响应时间若提升1s,如果流量大的话,就能增加不少转换量。做过高性能web前端调优的都知道,要将静态资源地址用两三个子域名去加载,为什么?因为每多一个子域名,浏览器在加载你的页面的时候就会多开几个线程去加载你的页面资源,提升网站的响应速度。多线程,高并发真的是无处不在。

(3)可以使你的代码模块化,异步化,简单化

例如我们在做 Android程序开发的时候,主线程的UI展示部分是一块主代码程序部分,但是UI上的按钮用相应事件的处理程序就可以做个单独的模块程序拿出来。这样既增加了异步的操,又使程序模块化,清晰化和简单化。

时下最流行的异步程序处理机制,正是多线程、并发程序最好的应用例子。

多线程应用开发的好处还有很多,大家在日后的代码编写过程中可以慢慢体会它的魅力。

多线程程序需要注意事项

(1)线程之间的安全性

从前面的章节中我们都知道,在同一个进程里面的多线程是资源共享的,也就是都可以访问同一个内存地址当中的一个变量。例如:若每个线程中对全局变量、静态变量只有读操作,而无写操作,一般来说,这个全局变量是线程安全的:若有多个线程同时执行写操作,一般都需要考虑线程同步,否则就可能影响线程安全。

(2)线程之间的死循环过程

为了解决线程之间的安全性引入了Java的锁机制,而一不小心就会产生Java线程死锁的多线程问题,因为不同的线程都在等待那些根本不可能被释放的锁,从而导致所有的工作都无法完成。假设有两个线程,分别代表两个饥饿的人,他们必须共享刀叉并轮流吃饭。他们都需要获得两个锁:共享刀和共享叉的锁。

假如线程A获得了刀,而线程B获得了叉。线程A就会进入阻塞状态来等待获得叉,而线程B则阻塞来等待线程A所拥有的刀。这只是人为设计的例子,但尽管在运行时很难探测到,这类情况却时常发生

(3)线程太多了会将服务器资源耗尽形成死机当机

线程数太多有可能造成系统创建大量线程而导致消耗完系统内存以及CPU的“过渡切换”,造成系统的死机,那么我们该如何解决这类问题呢?

某些系统资源是有限的,如文件描述符。多线程程序可能耗尽资源,因为每个线程都可能希望有一个这样的资源。如果线程数相当大,或者某个资源的侯选线程数远远超过了可用的资源数则最好使用资源池。一个最好的示例是数据库连接池。只要线程需要使用一个数据库连接,它就从池中取出一个,使用以后再将它返回池中。资源池也称为资源库。这里先有一个概念,后面会

多线程应用开发的注意事项很多,希望大家在日后的工作中可以慢慢体会它的危险所在。

## 认识Java里的线程

### Java里的程序天生就是多线程的

一个Java程序从main()方法开始执行，然后按照既定的代码逻辑执行，看似没有其他线程参与，但实际上Java程序天生就是多线程程序，因为执行main()方法的是一个名称为main的线程。

[6] Monitor Ctrl-Break //监控Ctrl-Break中断信号的

[5] Attach Listener //内存dump，线程dump，类信息统计，获取系统属性等

[4] Signal Dispatcher // 分发处理发送给JVM信号的线程

[3] Finalizer // 调用对象finalize方法的线程

[2] Reference Handler//清除Reference的线程

[1] main //main线程，用户程序入口

### 线程的启动与中止

#### 启动

启动线程的方式有：

1、X extends Thread;，然后X.run

2、X implements Runnable；然后交给Thread运行

3、X implements Callable；然后交给Thread运行

第1、2方式都有一个缺陷就是：在执行完任务之后无法获取执行结果。从Java 1.5开始，就提供了Callable和Future，通过它们可以在任务执行完毕之后得到任务执行结果。

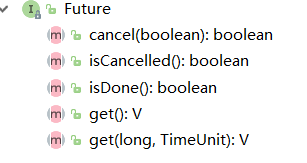
参见代码：cn.enjoyedu.concurrent.NewThread

##### Callable、Future和FutureTask

Runnable是一个接口，在它里面只声明了一个run()方法，由于run()方法返回值为void类型，所以在执行完任务之后无法返回任何结果。

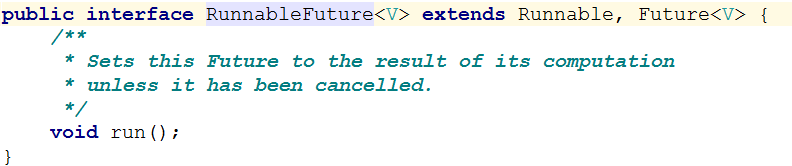
Callable位于java.util.concurrent包下，它也是一个接口，在它里面也只声明了一个方法，只不过这个方法叫做call()，这是一个泛型接口，call()函数返回的类型就是传递进来的V类型。

Future就是对于具体的Runnable或者Callable任务的执行结果进行取消、查询是否完成、获取结果。必要时可以通过get方法获取执行结果，该方法会阻塞直到任务返回结果。



因为Future只是一个接口，所以是无法直接用来创建对象使用的，因此就有了下面的FutureTask。

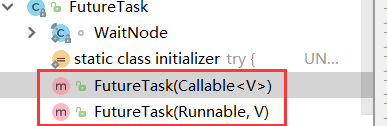




FutureTask类实现了RunnableFuture接口，RunnableFuture继承了Runnable接口和Future接口，而FutureTask实现了RunnableFuture接口。所以它既可以作为Runnable被线程执行，又可以作为Future得到Callable的返回值。

事实上，FutureTask是Future接口的一个唯一实现类。

要new一个FutureTask的实例，有两种方法



#### 中止

线程自然终止：要么是run执行完成了，要么是抛出了一个未处理的异常导致线程提前结束。

手动中止

暂停、恢复和停止操作对应在线程Thread的API就是**suspend()、resume()和stop()**。但是这些API是过期的，也就是不建议使用的。不建议使用的原因主要有：以suspend()方法为例，在调用后，线程不会释放已经占有的资源（比如锁），而是占有着资源进入睡眠状态，这样容易引发死锁问题。同样，stop()方法在终结一个线程时不会保证线程的资源正常释放，通常是没有给予线程完成资源释放工作的机会，因此会导致程序可能工作在不确定状态下。正因为suspend()、resume()和stop()方法带来的副作用，这些方法才被标注为不建议使用的过期方法。

安全的中止则是其他线程通过调用某个线程A的**interrupt()**方法对其进行中断操作, 中断好比其他线程对该线程打了个招呼，“A，你要中断了”，不代表线程A会立即停止自己的工作，同样的A线程完全可以不理会这种中断请求。因为java里的线程是协作式的，不是抢占式的。线程通过检查自身的中断标志位是否被置为true来进行响应，线程通过方法**isInterrupted()**来进行判断是否被中断，也可以调用静态方法**Thread.interrupted()**来进行判断当前线程是否被中断，不过Thread.interrupted()会同时将中断标识位改写为false。

如果一个线程处于了阻塞状态（如线程调用了thread.sleep、thread.join、thread.wait、），则在线程在检查中断标示时如果发现中断标示为true，则会在这些阻塞方法调用处抛出InterruptedException异常，并且在抛出异常后会立即将线程的中断标示位清除，即重新设置为false。

不建议自定义一个取消标志位来中止线程的运行。因为run方法里有阻塞调用时会无法很快检测到取消标志，线程必须从阻塞调用返回后，才会检查这个取消标志。这种情况下，使用中断会更好，因为，一、一般的阻塞方法，如sleep等本身就支持中断的检查，二、检查中断位的状态和检查取消标志位没什么区别，用中断位的状态还可以避免声明取消标志位，减少资源的消耗。

**注意：处于死锁状态的线程无法被中断**

## 对Java里的线程再多一点点认识

### 深入理解run()和start()

Thread类是Java里对线程概念的抽象，可以这样理解：我们通过new Thread()其实只是new出一个Thread的实例，还没有操作系统中真正的线程挂起钩来。只有执行了start()方法后，才实现了真正意义上的启动线程。

start()方法让一个线程进入就绪队列等待分配cpu，分到cpu后才调用实现的run()方法，start()方法不能重复调用。

而run方法是业务逻辑实现的地方，本质上和任意一个类的任意一个成员方法并没有任何区别，可以重复执行，可以被单独调用。

### 其他的线程方法

yield()方法：使当前线程让出CPU占有权，但让出的时间是不可设定的。也不会释放锁资源，所有执行yield()的线程有可能在进入到可执行状态后马上又被执行。

join方法：把指定的线程加入到当前线程，可以将两个交替执行的线程合并为顺序执行的线程。比如在线程B中调用了线程A的Join()方法，直到线程A执行完毕后，才会继续执行线程B。

wait()/notify()/notifyAll()：后面会单独讲述

## 线程间的共享和协作

### 线程间的共享

线程开始运行，拥有自己的栈空间，就如同一个脚本一样，按照既定的代码一步一步地执行，直到终止。但是，每个运行中的线程，如果仅仅是孤立地运行，那么没有一点儿价值，或者说价值很少，如果多个线程能够相互配合完成工作，包括数据之间的共享，协同处理事情。这将会带来巨大的价值。

Java支持多个线程同时访问一个对象或者对象的成员变量，关键字synchronized可以修饰方法或者以同步块的形式来进行使用，它主要确保多个线程在同一个时刻，只能有一个线程处于方法或者同步块中，它保证了线程对变量访问的可见性和排他性，又称为内置锁机制。

对象锁和类锁：

对象锁是用于对象实例方法，或者一个对象实例上的，类锁是用于类的静态方法或者一个类的class对象上的。我们知道，类的对象实例可以有很多个，但是每个类只有一个class对象，所以不同对象实例的对象锁是互不干扰的，但是每个类只有一个类锁。

但是有一点必须注意的是，其实类锁只是一个概念上的东西，并不是真实存在的，类锁其实锁的是每个类的对应的class对象。类锁和对象锁之间也是互不干扰的。

### 线程间的协作

线程之间相互配合，完成某项工作，比如：一个线程修改了一个对象的值，而另一个线程感知到了变化，然后进行相应的操作，整个过程开始于一个线程，而最终执行又是另一个线程。前者是生产者，后者就是消费者，这种模式隔离了“做什么”（what）和“怎么做”（How），简单的办法是让消费者线程不断地循环检查变量是否符合预期在while循环中设置不满足的条件，如果条件满足则退出while循环，从而完成消费者的工作。却存在如下问题：

1） 难以确保及时性。

2）难以降低开销。如果降低睡眠的时间，比如休眠1毫秒，这样消费者能更加迅速地发现条件变化，但是却可能消耗更多的处理器资源，造成了无端的浪费。

#### 等待/通知机制

是指一个线程A调用了对象O的wait()方法进入等待状态，而另一个线程B调用了对象O的notify()或者notifyAll()方法，线程A收到通知后从对象O的wait()方法返回，进而执行后续操作。上述两个线程通过对象O来完成交互，而对象上的wait()和notify/notifyAll()的关系就如同开关信号一样，用来完成等待方和通知方之间的交互工作。

notify()：

通知一个在对象上等待的线程,使其从wait方法返回,而返回的前提是该线程获取到了对象的锁，没有获得锁的线程重新进入WAITING状态。

notifyAll()：

通知所有等待在该对象上的线程

wait()

调用该方法的线程进入 WAITING状态,只有等待另外线程的通知或被中断才会返回.需要注意,调用wait()方法后,会释放对象的锁

wait(long)

超时等待一段时间,这里的参数时间是毫秒,也就是等待长达n毫秒,如果没有通知就超时返回

wait (long,int)

对于超时时间更细粒度的控制,可以达到纳秒

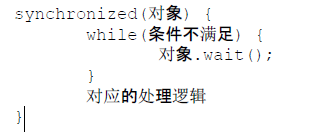
##### 等待和通知的标准范式

等待方遵循如下原则。

1）获取对象的锁。

2）如果条件不满足，那么调用对象的wait()方法，被通知后仍要检查条件。

3）条件满足则执行对应的逻辑。

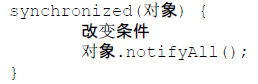


通知方遵循如下原则。

1）获得对象的锁。

2）改变条件。

3）通知所有等待在对象上的线程。



在调用wait（）之前，线程必须要获得该对象的对象级别锁，即只能在同步方法或同步块中调用wait（）方法，进入wait（）方法后，当前线程释放锁，在从wait（）返回前，线程与其他线程竞争重新获得锁，notifyAll方法一旦该对象锁被释放（notifyAll线程退出调用了notifyAll的synchronized代码块的时候），他们就会去竞争。如果其中一个线程获得了该对象锁，它就会继续往下执行，在它退出synchronized代码块，释放锁后，其他的已经被唤醒的线程将会继续竞争获取该锁，一直进行下去，直到所有被唤醒的线程都执行完毕。

##### notify和notifyAll应该用谁

尽可能用notifyall()，谨慎使用notify()，看代码。

### ThreadLocal

ThreadLocal，即线程变量，是一个以ThreadLocal对象为键、任意对象为值的存储结构。这个结构被附带在线程上，也就是说一个线程可以根据一个ThreadLocal对象查询到绑定在这个线程上的一个值, ThreadLocal往往用来实现变量在线程之间的隔离。

ThreadLocal类接口很简单，只有4个方法，我们先来了解一下：

• void set(Object value)

设置当前线程的线程局部变量的值。

• public Object get()

该方法返回当前线程所对应的线程局部变量。

• public void remove()

将当前线程局部变量的值删除，目的是为了减少内存的占用，该方法是JDK 5.0新增的方法。需要指出的是，当线程结束后，对应该线程的局部变量将自动被垃圾回收，所以显式调用该方法清除线程的局部变量并不是必须的操作，但它可以加快内存回收的速度。

• protected Object initialValue()

返回该线程局部变量的初始值，该方法是一个protected的方法，显然是为了让子类覆盖而设计的。这个方法是一个延迟调用方法，在线程第1次调用get()或set(Object)时才执行，并且仅执行1次。ThreadLocal中的缺省实现直接返回一个null。

public final static ThreadLocal<String> RESOURCE = new ThreadLocal<String>();RESOURCE代表一个能够存放String类型的ThreadLocal对象。此时不论什么一个线程能够并发访问这个变量，对它进行写入、读取操作，都是线程安全的。

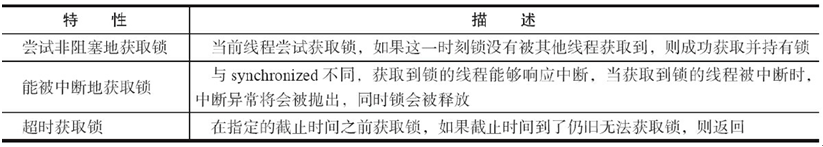
## 显式锁

### Lock接口和synchronized的比较

我们一般的Java程序是靠synchronized关键字实现锁功能的，使用synchronized关键字将会隐式地获取锁，但是它将锁的获取和释放固化了，也就是先获取再释放。synchronized属于Java语言层面的锁，也被称之为内置锁。

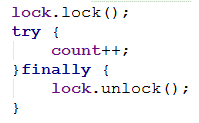
synchronized这种机制，一旦开始获取锁，是不能中断的，也不提供尝试获取锁的机制。

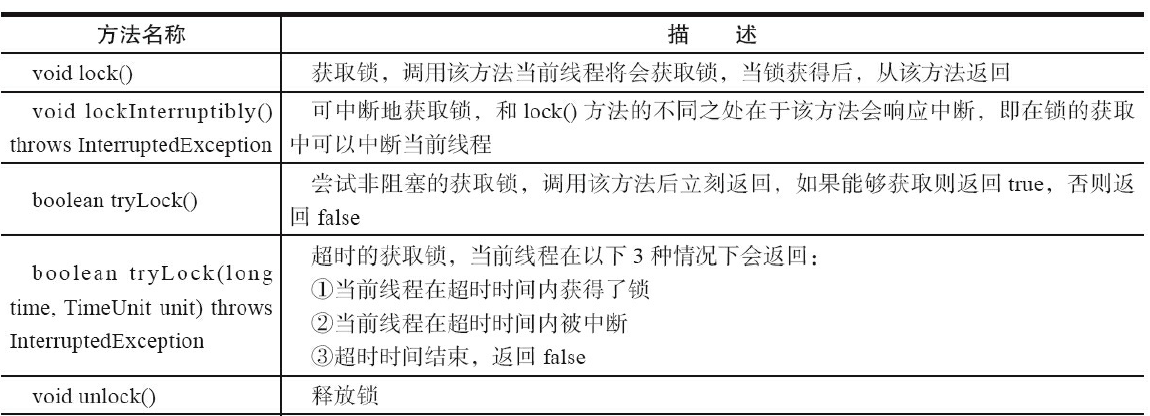
而Lock是由Java在语法层面提供的，锁的获取和释放需要我们明显的去获取，因此被称为显式锁。并且提供了synchronized不提供的机制。



### Lock接口和核心方法

在finally块中释放锁，目的是保证在获取到锁之后，最终能够被释放。





### 可重入锁ReentrantLock、所谓锁的公平和非公平

而synchronized关键字隐式的支持重进入，比如一个synchronized修饰的递归方法，在方法执行时，执行线程在获取了锁之后仍能连续多次地获得该锁。ReentrantLock在调用lock()方法时，已经获取到锁的线程，能够再次调用lock()方法获取锁而不被阻塞。

公平和非公平锁

如果在时间上，先对锁进行获取的请求一定先被满足，那么这个锁是公平的，反之，是不公平的。公平的获取锁，也就是等待时间最长的线程最优先获取锁，也可以说锁获取是顺序的。

ReentrantLock提供了一个构造函数，能够控制锁是否是公平的。事实上，公平的锁机制往往没有非公平的效率高。原因是，在恢复一个被挂起的线程与该线程真正开始运行之间存在着严重的延迟。假设线程A持有一个锁,并且线程B请求这个锁。由于这个锁已被线程A持有,因此B将被挂起。当A释放锁时,B将被唤醒,因此会再次尝试获取锁。与此同时,如果C也请求这个锁,那么C很可能会在B被完全唤醒之前获得、使用以及释放这个锁。这样的情况是一种“双赢”的局面:B获得锁的时刻并没有推迟,C更早地获得了锁,并且吞吐量也获得了提高。

### 读写锁ReentrantReadWriteLock

之前提到锁（synchronized和ReentrantLock）基本都是排他锁，这些锁在同一时刻只允许一个线程进行访问，而读写锁在同一时刻可以允许多个读线程访问，但是在写线程访问时，所有的读线程和其他写线程均被阻塞。读写锁维护了一对锁，一个读锁和一个写锁，通过分离读锁和写锁，使得并发性相比一般的排他锁有了很大提升。

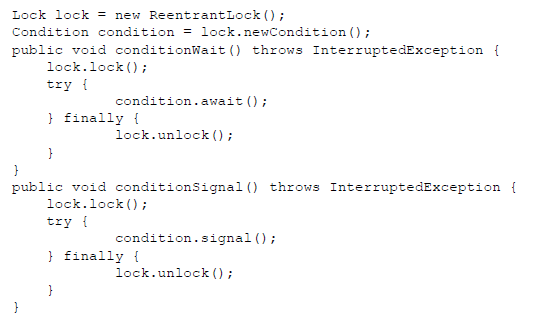
除了保证写操作对读操作的可见性以及并发性的提升之外，读写锁能够简化读写交互场景的编程方式。假设在程序中定义一个共享的用作缓存数据结构，它大部分时间提供读服务（例如查询和搜索），而写操作占有的时间很少，但是写操作完成之后的更新需要对后续的读服务可见。

一般情况下，读写锁的性能都会比排它锁好，因为大多数场景读是多于写的。在读多于写的情况下，读写锁能够提供比排它锁更好的并发性和吞吐量

### Condition接口

任意一个Java对象，都拥有一组监视器方法（定义在java.lang.Object上），主要包括wait()、wait(long timeout)、notify()以及notifyAll()方法，这些方法与synchronized同步关键字配合，可以实现等待/通知模式。Condition接口也提供了类似Object的监视器方法，与Lock配合可以实现等待/通知模式。

用Lock和Condition实现等待通知



# 线程池原理与AsyncTask

## 什么是线程池？为什么要用线程池？

Java中的线程池是运用场景最多的并发框架，几乎所有需要异步或并发执行任务的程序都可以使用线程池。线程池就是将线程进行池化，需要运行任务时从池中拿一个线程来执行，执行完毕，线程放回池中。

在开发过程中，合理地使用线程池能够带来3个好处。

第一：降低资源消耗。通过重复利用已创建的线程降低线程创建和销毁造成的消耗。

第二：提高响应速度。当任务到达时，任务可以不需要等到线程创建就能立即执行。假设一个服务器完成一项任务所需时间为：T1 创建线程时间，T2 在线程中执行任务的时间，T3 销毁线程时间。 如果：T1 + T3 远大于 T2，则可以采用线程池，以提高服务器性能。线程池技术正是关注如何缩短或调整T1,T3时间的技术，从而提高服务器程序性能的。它把T1，T3分别安排在服务器程序的启动和结束的时间段或者一些空闲的时间段，这样在服务器程序处理客户请求时，不会有T1，T3的开销了。

第三：提高线程的可管理性。线程是稀缺资源，如果无限制地创建，不仅会消耗系统资源，还会降低系统的稳定性，使用线程池可以进行统一分配、调优和监控。

## JDK中的线程池和工作机制

### 线程池的创建各个参数含义

public ThreadPoolExecutor(int corePoolSize,int maximumPoolSize,long keepAliveTime,TimeUnit unit,BlockingQueue<Runnable> workQueue,ThreadFactory threadFactory,RejectedExecutionHandler handler)

#### corePoolSize

线程池中的核心线程数，当提交一个任务时，线程池创建一个新线程执行任务，直到当前线程数等于corePoolSize；

如果当前线程数为corePoolSize，继续提交的任务被保存到阻塞队列中，等待被执行；

如果执行了线程池的prestartAllCoreThreads()方法，线程池会提前创建并启动所有核心线程。

#### maximumPoolSize

线程池中允许的最大线程数。如果当前阻塞队列满了，且继续提交任务，则创建新的线程执行任务，前提是当前线程数小于maximumPoolSize

#### keepAliveTime

线程空闲时的存活时间，即当线程没有任务执行时，继续存活的时间。默认情况下，该参数只在线程数大于corePoolSize时才有用

#### TimeUnit

keepAliveTime的时间单位

#### workQueue

workQueue必须是BlockingQueue阻塞队列。当线程池中的线程数超过它的corePoolSize的时候，线程会进入阻塞队列进行阻塞等待。通过workQueue，线程池实现了阻塞功能

##### 什么是阻塞队列

**队列：**

队列是一种特殊的线性表，特殊之处在于它只允许在表的前端（front）进行删除操作，而在表的后端（rear）进行插入操作，和栈一样，队列是一种操作受限制的线性表。进行插入操作的端称为队尾，进行删除操作的端称为队头。队列中没有元素时，称为空队列。

队列的数据元素又称为队列元素。在队列中插入一个队列元素称为入队，从队列中删除一个队列元素称为出队。因为队列只允许在一端插入，在另一端删除，所以只有最早进入队列的元素才能最先从队列中删除，故队列又称为先进先出（FIFO—first in first out）线性表。

**阻塞队列：**

1）支持阻塞的插入方法：意思是当队列满时，队列会阻塞插入元素的线程，直到队列不满。

2）支持阻塞的移除方法：意思是在队列为空时，获取元素的线程会等待队列变为非空。

阻塞队列常用于生产者和消费者的场景，生产者是向队列里添加元素的线程，消费者是从队列里取元素的线程。阻塞队列就是生产者用来存放元素、消费者用来获取元素的容器。 

•抛出异常：当队列满时，如果再往队列里插入元素，会抛出IllegalStateException（"Queuefull"）异常。当队列空时，从队列里获取元素会抛出NoSuchElementException异常。

•返回特殊值：当往队列插入元素时，会返回元素是否插入成功，成功返回true。如果是移除方法，则是从队列里取出一个元素，如果没有则返回null。

•一直阻塞：当阻塞队列满时，如果生产者线程往队列里put元素，队列会一直阻塞生产者线程，直到队列可用或者响应中断退出。当队列空时，如果消费者线程从队列里take元素，队列会阻塞住消费者线程，直到队列不为空。

•超时退出：当阻塞队列满时，如果生产者线程往队列里插入元素，队列会阻塞生产者线程一段时间，如果超过了指定的时间，生产者线程就会退出。

##### 常用阻塞队列

•ArrayBlockingQueue：一个由数组结构组成的有界阻塞队列。

•LinkedBlockingQueue：一个由链表结构组成的有界阻塞队列。

•PriorityBlockingQueue：一个支持优先级排序的无界阻塞队列。

•DelayQueue：一个使用优先级队列实现的无界阻塞队列。

•SynchronousQueue：一个不存储元素的阻塞队列。

•LinkedTransferQueue：一个由链表结构组成的无界阻塞队列。

•LinkedBlockingDeque：一个由链表结构组成的双向阻塞队列。

#### threadFactory

创建线程的工厂，通过自定义的线程工厂可以给每个新建的线程设置一个具有识别度的线程名Executors静态工厂里默认的threadFactory，线程的命名规则是“pool-数字-thread-数字”

#### RejectedExecutionHandler（饱和策略）

线程池的饱和策略，当阻塞队列满了，且没有空闲的工作线程，如果继续提交任务，必须采取一种策略处理该任务，线程池提供了4种策略：

（1）AbortPolicy：直接抛出异常，默认策略；

（2）CallerRunsPolicy：用调用者所在的线程来执行任务；

（3）DiscardOldestPolicy：丢弃阻塞队列中靠最前的任务，并执行当前任务；

（4）DiscardPolicy：直接丢弃任务；

当然也可以根据应用场景实现RejectedExecutionHandler接口，自定义饱和策略，如记录日志或持久化存储不能处理的任务。

### 线程池的工作机制

1）如果当前运行的线程少于corePoolSize，则创建新线程来执行任务（注意，执行这一步骤需要获取全局锁）。

2）如果运行的线程等于或多于corePoolSize，则将任务加入BlockingQueue。

3）如果无法将任务加入BlockingQueue（队列已满），则创建新的线程来处理任务（注意，执行这一步骤需要获取全局锁）。

4）如果创建新线程将使当前运行的线程超出maximumPoolSize，任务将被拒绝，并调用RejectedExecutionHandler.rejectedExecution()方法。

## 合理配置线程池

要想合理地配置线程池，就必须首先分析任务特性

要想合理地配置线程池，就必须首先分析任务特性，可以从以下几个角度来分析。

•任务的性质：CPU密集型任务、IO密集型任务和混合型任务。

•任务的优先级：高、中和低。

•任务的执行时间：长、中和短。

•任务的依赖性：是否依赖其他系统资源，如数据库连接。

性质不同的任务可以用不同规模的线程池分开处理。CPU密集型任务应配置尽可能小的线程，如配置Ncpu+1个线程的线程池。由于IO密集型任务线程并不是一直在执行任务，则应配置尽可能多的线程，如2\*Ncpu。混合型的任务，如果可以拆分，将其拆分成一个CPU密集型任务和一个IO密集型任务，只要这两个任务执行的时间相差不是太大，那么分解后执行的吞吐量将高于串行执行的吞吐量。如果这两个任务执行时间相差太大，则没必要进行分解。可以通过Runtime.getRuntime().availableProcessors()方法获得当前设备的CPU个数。

## AsyncTask

### 为什么需要AsyncTask？

在Android当中，当一个应用程序的组件启动的时候，并且没有其他的应用程序组件在运行时，Android系统就会为该应用程序组件开辟一个新的线程来执行。默认的情况下，在一个相同Android应用程序当中，其里面的组件都是运行在同一个线程里面的，这个线程我们称之为Main线程。当我们通过某个组件来启动另一个组件的时候，这个时候默认都是在同一个线程当中完成的。

在Android当中，通常将线程分为两种，一种叫做Main Thread，除了Main Thread之外的线程都可称为Worker Thread。

当一个应用程序运行的时候，Android操作系统就会给该应用程序启动一个线程，这个线程就是我们的Main Thread，这个线程非常的重要，它主要用来加载我们的UI界面，完成系统和我们用户之间的交互，并将交互后的结果又展示给我们用户，所以Main Thread又被称为UI Thread。

Android系统默认不会给我们的应用程序组件创建一个额外的线程，所有的这些组件默认都是在同一个线程中运行。然而，某些时候当我们的应用程序需要完成一个耗时的操作的时候，例如访问网络或者是对数据库进行查询时，此时我们的UI Thread就会被阻塞。例如，当我们点击一个Button，然后希望其从网络中获取一些数据，如果此操作在UI Thread当中完成的话，当我们点击Button的时候，UI线程就会处于阻塞的状态，此时，我们的系统不会调度任何其它的事件，更糟糕的是，当我们的整个现场如果阻塞时间超过5秒钟(官方是这样说的)，这个时候就会出现 ANR (Application Not Responding)的现象，此时，应用程序会弹出一个框，让用户选择是否退出该程序。对于Android开发来说，出现ANR的现象是绝对不能被允许的。

另外，由于我们的Android UI控件是线程不安全的，所以我们不能在UI Thread之外的线程当中对我们的UI控件进行操作。因此在Android的多线程编程当中，我们有两条非常重要的原则必须要遵守：

* 绝对不能在UI Thread当中进行耗时的操作，不能阻塞我们的UI Thread
* 不能在UI Thread之外的线程当中操纵我们的UI元素

既然在Android当中有两条重要的原则要遵守，那么我们可能就有疑问了？我们既不能在主线程当中处理耗时的操作，又不能在工作线程中来访问我们的UI控件，那么我们比如从网络中要下载一张图片，又怎么能将其更新到UI控件上呢？这就关系到了我们的主线程和工作线程之间的通信问题了。在Android当中，提供了两种方式来解决线程直接的通信问题，一种是通过Handler的机制，这个时候就很可能自己会去封装一下thread+handler了，正是因为这类需求很多，google就帮我们封装了一下。其实我们也可以自己封装，但是我相信99%程序员自己封装的东西比不上google的。所以另外一种就是今天要详细讲解的 AsyncTask 机制。

### 原理分析

AsyncTask是个abstract类，所以在使用时需要实现一个AsyncTask的具体实现类，一般来说会覆盖4个方法，我们以前面所说的从网络中下载一张图片，然后更新到UI控件来说明：

（1）onPreExecute()：在执行后台下载操作之前调用，将下载等待动画显示出来，运行在主线程中；

（2）doInBackground()：核心方法，执行后台下载操作的方法，必须实现的一个方法，运行在子线程中；这个方法是执行在子线程中的。在onPreExecute()执行完后，会立即开启这个方法。

（3）onProgressUpdate()：在下载操作doInBackground()中调用publishProgress()时的回调方法，用于更新下载进度，运行在主线程中；

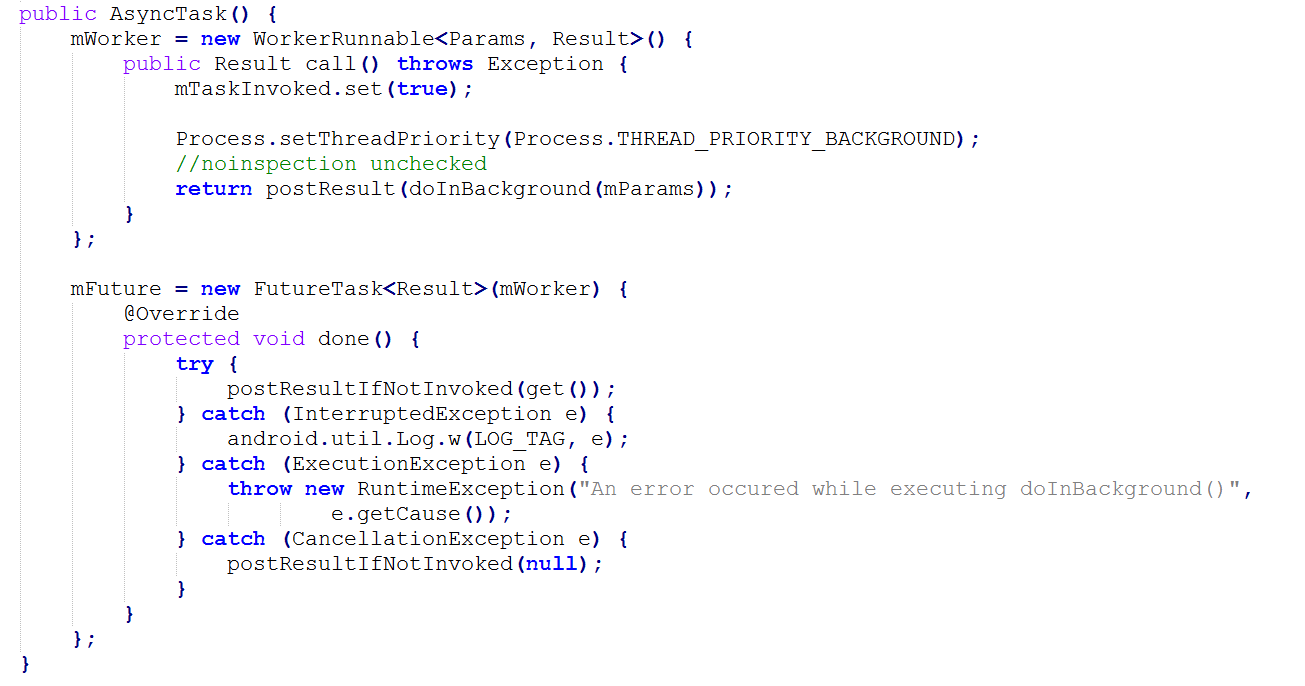
（4）onPostExecute()：后台下载操作完成后调用，将下载等待动画进行隐藏，并更新UI，运行在主线程中；

然后在主线程中

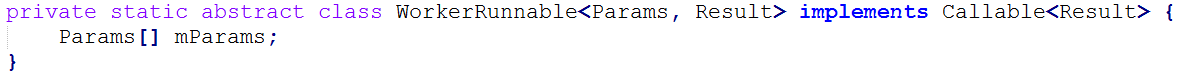


通过上面的分析，我们可以知道，AsyncTask的构造方法和execute方法是我们分析AsyncTask的重点。

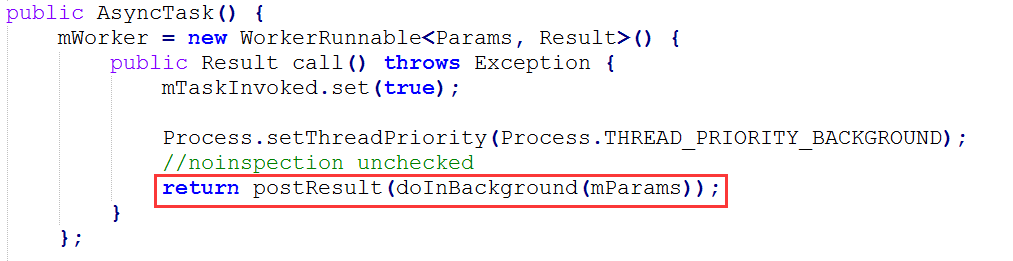
#### 1）构造方法

AsyncTask的构造方法中

mWorker代表了AsyncTask要执行的任务，是对Callable接口的封装，意味着这个任务是有返回值的



mFuture代表了AsyncTask要执行的任务的返回结果，其实就是个FutureTask，安装FutureTask标准用法，mWorker作为Callable被传给了mFuture，那么mFuture的结果就从mWorker执行的任务中取得。仔细看mWorker，return语句返回的结果就是我们前面所说的doInBackground()的执行结果：

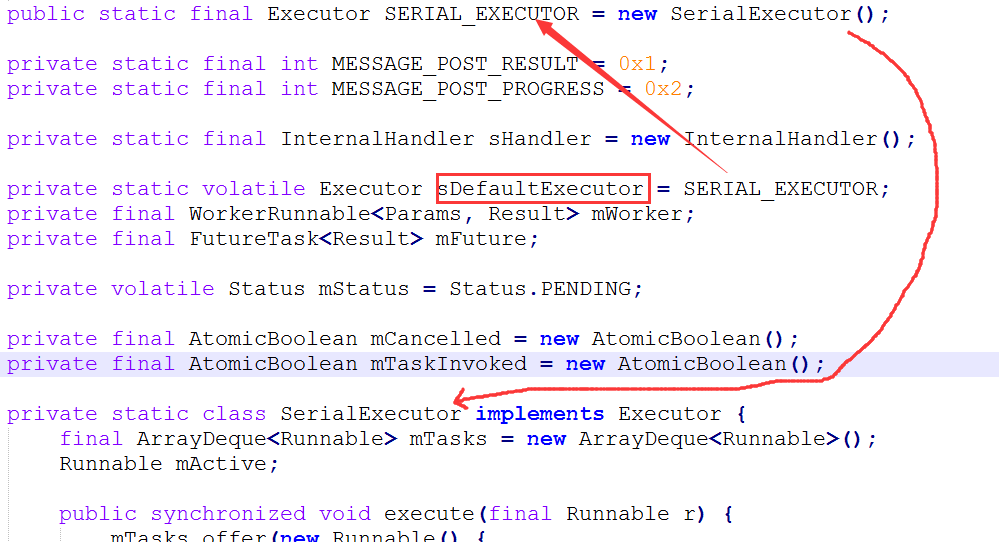


#### 2）再看执行流程

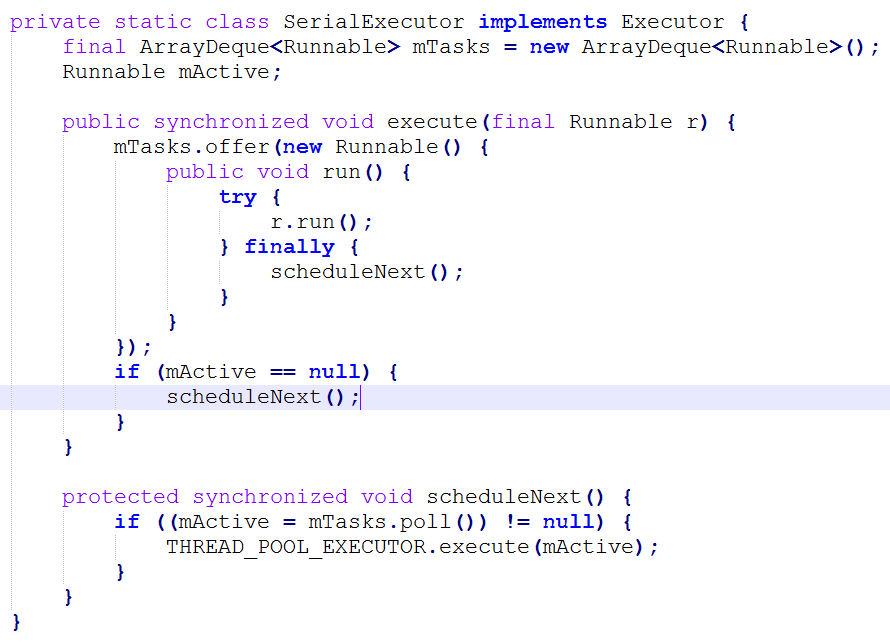
查看源码execute()🡪executeOnExecutor(sDefaultExecutor, params)🡪exec.execute(mFuture)

到了这一步，将mFuture传递给了AsyncTask的执行器进行执行。AsyncTask的执行器缺省是sDefaultExecutor。

找到成员变量sDefaultExecutor，最终定位到



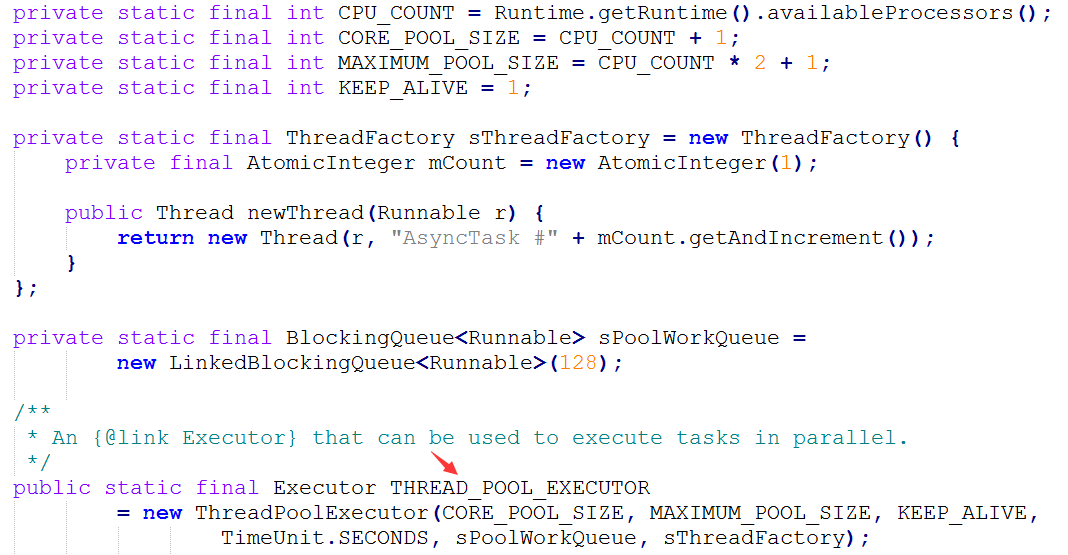
SerialExecutor是对JDK里Executor的一个实现，被声明为一个静态变量，我们仔细看SerialExecutor的实现，



内部声明了一个双端队列ArrayDeque类型的mTasks（双端队列中offer方法表示从队列尾插入，poll()表示从队列头获取元素）。

每次调用execute，就创建一个Runnable匿名内部类对象，这个对象存入mTasks，在匿名内部类的run函数里面调用传入参数r.run()。然后通过一个scheduleNext函数把mTasks里面的所有对象通过THREAD\_POOL\_EXECUTOR.execute(mActive)执行一遍。说穿了，也就是说SerialExecutor类会把所有的任务丢入一个容器，之后把容器里面的所有对象**一个一个的排队（串行化）**执行THREAD\_POOL\_EXECUTOR.execute(mActive);

至于这个THREAD\_POOL\_EXECUTOR，是这样定义的：

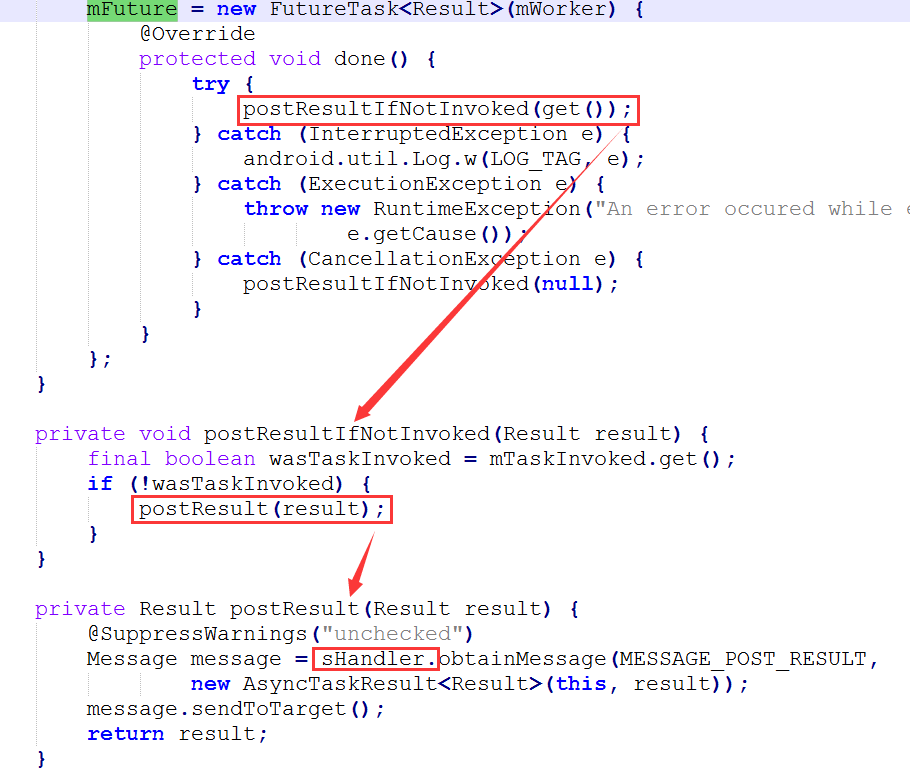


我们可以看到这个线程池，被声明为一个静态变量，同时初始化的参数是：

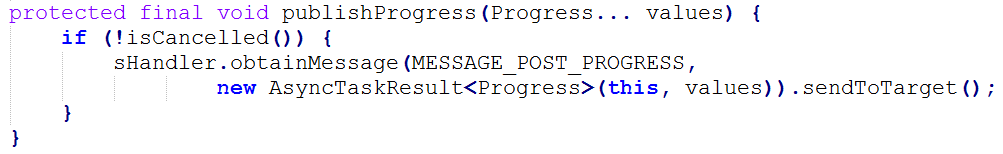
核心线程数量，这里取得是CPU个数 + 1， 第二个参数是最大线程数量，这里是CPU个数 \* 2 + 1，第五个参数是缓冲区的队列，这里是个LinkedBlockingQueue，这个队列的最大容量是128。

#### 3)结果和进度的通知

AsyncTask的执行结果和进度是怎么通知给UI线程的呢？检视mFuture

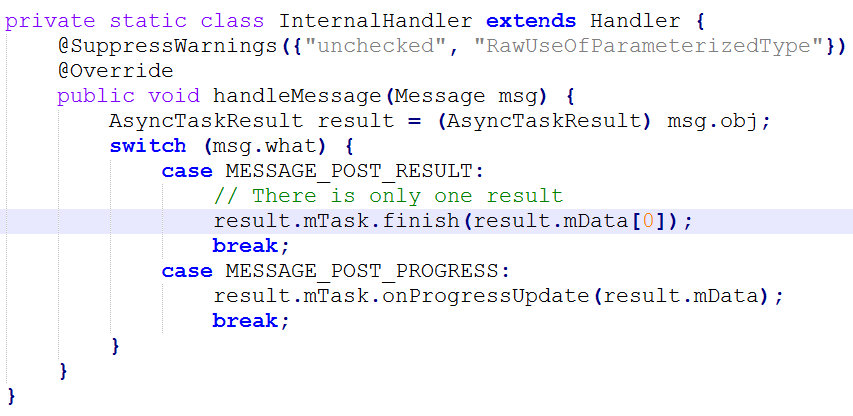


和更新进度时我们会调用的publishProgress方法



我们可以看到都调用了sHandler





说明当子线程需要和UI线程进行通信时，其实就是通过这个handler，往UI线程发送消息。

#### 总结：

1）、线程池的创建：

在创建了AsyncTask的时候，会默认创建两个线程池SerialExecutor和ThreadPoolExecutor，SerialExecutor负责将任务串行化，ThreadPoolExecutor是真正执行任务的地方，且无论有多少个AsyncTask实例，两个线程池都会只有一份。

2）、任务的执行：

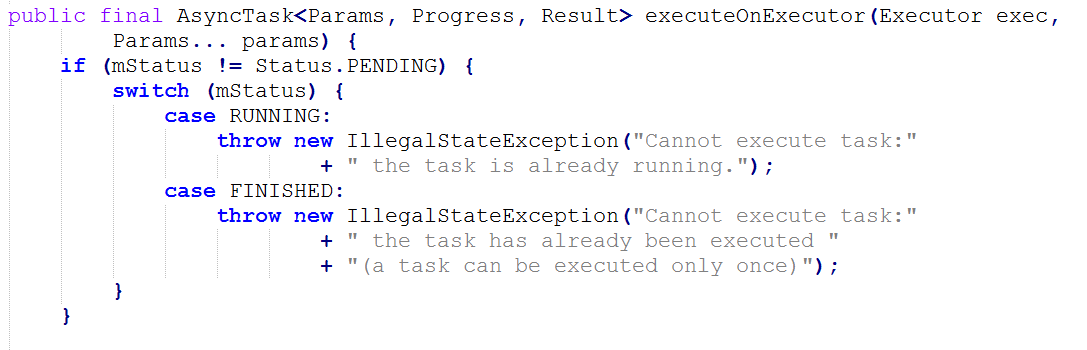
在execute中，会执行run方法，当执行完run方法后，会调用scheduleNext()不断的从双端队列中轮询，获取下一个任务并继续放到一个子线程中执行，直到异步任务执行完毕。

3）、消息的处理：

在执行完onPreExecute()方法之后，执行了doInBackground()方法，然后就不断的发送请求获取数据；在这个AsyncTask中维护了一个InternalHandler的类，这个类是继承Handler的，获取的数据是通过handler进行处理和发送的。在其handleMessage方法中，将消息传递给onProgressUpdate()进行进度的更新，也就可以将结果发送到主线程中，进行界面的更新了。

4）、使用AsyncTask的注意点

通过观察代码我们可以发现，每一个new出的AsyncTask只能执行一次execute()方法，多次运行将会报错，如需多次，需要新new一个AsyncTask。



### AsyncTask优缺点

**AsyncTask：**

优点：AsyncTask是一个轻量级的异步任务处理类，轻量级体现在，使用方便、代码简洁上，而且整个异步任务的过程可以通过cancel()进行控制；

缺点：不适用于处理长时间的异步任务，一般这个异步任务的过程最好控制在几秒以内，如果是长时间的异步任务就需要考虑多线程的控制问题；当处理多个异步任务时，UI更新变得困难。

**Handler:**

优点：代码结构清晰，容易处理多个异步任务；

缺点：当有多个异步任务时，由于要配合Thread或Runnable，代码可能会稍显冗余。

**总之，**AsyncTask不失为一个非常好用的异步任务处理类，只要不是频繁对大量UI进行更新，可以考虑使用；而Handler在处理大量UI更新时可以考虑使用。

# 补充知识：CAS

## 什么是原子操作？如何实现原子操作？

假定有两个操作A和B，如果从执行A的线程来看，当另一个线程执行B时，要么将B全部执行完，要么完全不执行B，那么A和B对彼此来说是原子的。

实现原子操作可以使用锁，锁机制，满足基本的需求是没有问题的了，但是有的时候我们的需求并非这么简单，我们需要更有效，更加灵活的机制，synchronized关键字是基于阻塞的锁机制，也就是说当一个线程拥有锁的时候，访问同一资源的其它线程需要等待，直到该线程释放锁，这里会有些问题：首先，如果被阻塞的线程优先级很高很重要怎么办？其次，如果获得锁的线程一直不释放锁怎么办？（这种情况是非常糟糕的）。还有一种情况，如果有大量的线程来竞争资源，那CPU将会花费大量的时间和资源来处理这些竞争（事实上CPU的主要工作并非这些），同时，还有可能出现一些例如死锁之类的情况，最后，其实锁机制是一种比较粗糙，粒度比较大的机制，相对于像计数器这样的需求有点儿过于笨重

实现原子操作还可以使用当前的处理器基本都支持CAS()的指令，只不过每个厂家所实现的算法并不一样罢了，**每一个CAS操作过程都包含三个运算符：一个内存地址V，一个期望的值A和一个新值B，操作的时候如果这个地址上存放的值等于这个期望的值A，则将地址上的值赋为新值B，否则不做任何操作。**CAS的基本思路就是，如果这个地址上的值和期望的值相等，则给其赋予新值，否则不做任何事儿，但是要返回原值是多少。**循环CAS就是在一个循环里不断的做CAS操作，直到成功为止。**怎么实现线程安全呢？语言层面不做处理，我们将其交给硬件—CPU和内存，利用CPU的多处理能力，实现硬件层面的阻塞，再加上volatile变量的特性即可实现基于原子操作的线程安全。

## CAS实现原子操作的三大问题

1) ABA问题。因为CAS需要在操作值的时候，检查值有没有发生变化，如果没有发生变化则更新，但是如果一个值原来是A，变成了B，又变成了A，那么使用CAS进行检查时会发现它的值没有发生变化，但是实际上却变化了。ABA问题的解决思路就是使用版本号。在变量前面追加上版本号，每次变量更新的时候把版本号加1，那么A→B→A就会变成1A→2B→3A。

举个通俗点的例子，你倒了一杯水放桌子上，干了点别的事，然后同事把你水喝了又给你重新倒了一杯水，你回来看水还在，拿起来就喝，如果你不管水中间被人喝过，只关心水还在，这就是ABA问题。如果你是一个讲卫生讲文明的小伙子，不但关心水在不在，还要在你离开的时候水被人动过没有，因为你是程序员，所以就想起了放了张纸在旁边，写上初始值0，别人喝水前麻烦先做个累加才能喝水。

2) 循环时间长开销大。自旋CAS如果长时间不成功，会给CPU带来非常大的执行开销。

3) 只能保证一个共享变量的原子操作。当对一个共享变量执行操作时，我们可以使用循环CAS的方式来保证原子操作，但是对多个共享变量操作时，循环CAS就无法保证操作的原子性，这个时候就可以用锁。还有一个取巧的办法，就是把多个共享变量合并成一个共享变量来操作。比如，有两个共享变量i＝2，j=a，合并一下ij=2a，然后用CAS来操作ij。从Java 1.5开始，JDK提供了AtomicReference类来保证引用对象之间的原子性，就可以把多个变量放在一个对象里来进行CAS操作。