# 一．反射

### 01\_反射(类的加载概述和加载时机)

* A:类的加载概述
  + 当程序要使用某个类时，如果该类还未被加载到内存中，则系统会通过加载，连接，初始化三步来实现对这个类进行初始化。
  + 加载
    - 就是指将class文件读入内存，并为之创建一个Class对象。任何类被使用时系统都会建立一个Class对象。
  + 连接
    - 验证 是否有正确的内部结构，并和其他类协调一致
    - 准备 负责为类的静态成员分配内存，并设置默认初始化值
    - 解析 将类的二进制数据中的符号引用替换为直接引用
  + 初始化 就是我们以前讲过的初始化步骤
* B:加载时机
  + 创建类的实例
  + 访问类的静态变量，或者为静态变量赋值
  + 调用类的静态方法
  + 使用反射方式来强制创建某个类或接口对应的java.lang.Class对象
  + 初始化某个类的子类
  + 直接使用java.exe命令来运行某个主类

### 02\_反射(类加载器的概述和分类)

* A:类加载器的概述
  + 负责将.class文件加载到内存中，并为之生成对应的Class对象。虽然我们不需要关心类加载机制，但是了解这个机制我们就能更好的理解程序的运行。
* B:类加载器的分类
  + Bootstrap ClassLoader 根类加载器
  + Extension ClassLoader 扩展类加载器
  + Sysetm ClassLoader 系统类加载器
* C:类加载器的作用
  + Bootstrap ClassLoader 根类加载器
    - 也被称为引导类加载器，负责Java核心类的加载
    - 比如System,String等。在JDK中JRE的lib目录下rt.jar文件中
  + Extension ClassLoader 扩展类加载器
    - 负责JRE的扩展目录中jar包的加载。
    - 在JDK中JRE的lib目录下ext目录
  + Sysetm ClassLoader 系统类加载器
    - 负责在JVM启动时加载来自java命令的class文件，以及classpath环境变量所指定的jar包和类路径

### 03\_反射(反射概述)

* A:反射概述
  + JAVA反射机制是在运行状态中，对于任意一个类，都能够知道这个类的所有属性和方法；
  + 对于任意一个对象，都能够调用它的任意一个方法和属性；
  + 这种动态获取的信息以及动态调用对象的方法的功能称为java语言的反射机制。
  + 要想解剖一个类,必须先要获取到该类的字节码文件对象。
  + 而解剖使用的就是Class类中的方法，所以先要获取到每一个字节码文件对应的Class类型的对象。
* B:三种方式
  + a:Object类的getClass()方法,判断两个对象是否是同一个字节码文件
  + b:静态属性class,锁对象
  + c:Class类中静态方法forName(),读取配置文件
* C:案例演示

1.Account account1 = **new** Account();

Class<? **extends** Account> class1 = account1.getClass();

System.*out*.println(class1); //class org.scbit.lsbi.unilab.pojo.Account

Account account2 = **new** Account();

Class<? **extends** Account> class2 = account2.getClass();

System.*out*.println(class1 == class2); //true

2. Class class1 = Account.**class**;

Class class2 = Account.**class**;

System.*out*.println(class1==class2); //true

3. Class class1 = Class.*forName*("org.scbit.lsbi.unilab.pojo.Account");

System.*out*.println(class1);

### 05\_反射(通过反射获取带参构造方法并使用)

* Constructor
  + Class类的newInstance()方法是使用该类无参的构造函数创建对象, 如果一个类没有无参的构造函数, 就不能这样创建了,可以调用Class类的getConstructor(String.class,int.class)方法获取一个指定的构造函数然 后再调用Constructor类的newInstance("张三",20)方法创建对象

首先，我们在开始前提出一个问题：

1.在运行时，对于一个java类，能否知道属性和方法；能否去调用它的任意方法？ 答案是肯定的。

本节所有目录如下：

1. 什么是JAVA的反射机制
2. JDK中提供的Reflection API
3. JAVA反射机制提供了什么功能
   * 获取类的Class对象
   * 获取类的Fields
   * 获取类的Method
   * 获取类的Constructor
   * 新建类的实例  
            Class<T>的函数newInstance  
            通过Constructor对象的方法newInstance
4. 调用类的函数  
            调用private函数
5. 设置/获取类的属性值  
            private属性
6. 动态创建代理类  
            动态代理源码分析
7. JAVA反射Class<T>类型源代码分析
8. JAVA反射原理分析  
           Class文件结构  
           JVM加载类对象，对反射的支持
9. JAVA反射的应用

一、什么是JAVA的反射机制

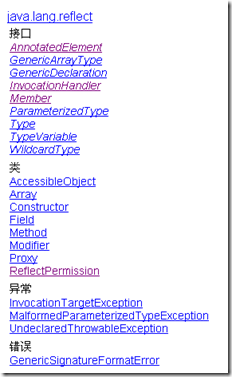
Java反射是Java被视为动态（或准动态）语言的一个关键性质。这个机制允许程序在运行时透过Reflection APIs取得任何一个已知名称的class的内部信息，包括其modifiers（诸如public, static 等）、superclass（例如Object）、实现之interfaces（例如Cloneable），也包括fields和methods的所有信息，并可于运行时改变fields内容或唤起methods。

Java反射机制容许程序在运行时加载、探知、使用编译期间完全未知的classes。

换言之，Java可以加载一个运行时才得知名称的class，获得其完整结构。

二、JDK中提供的Reflection API

Java反射相关的API在包[java.lang.reflect](http://blog.csdn.net/yongjian1092/article/details/7364451" \t "_blank)中，JDK 1.6.0的reflect包如下图：

[](http://images.cnblogs.com/cnblogs_com/Quincy/201106/201106191021251045.png)

|  |  |
| --- | --- |
| Member接口 | 该接口可以获取有关类成员（域或者方法）后者构造函数的信息。 |
| AccessibleObject类 | 该类是域(field)对象、方法(method)对象、构造函数(constructor)对象的基础类。它提供了将反射的对象标记为在使用时取消默认 Java 语言访问控制检查的能力。 |
| Array类 | 该类提供动态地生成和访问JAVA数组的方法。 |
| Constructor类 | 提供一个类的构造函数的信息以及访问类的构造函数的接口。 |
| Field类 | 提供一个类的域的信息以及访问类的域的接口。 |
| Method类 | 提供一个类的方法的信息以及访问类的方法的接口。 |
| Modifier类 | 提供了 static 方法和常量，对类和成员访问修饰符进行解码。 |
| Proxy类 | 提供动态地生成代理类和类实例的静态方法。 |

三、JAVA反射机制提供了什么功能

Java反射机制提供如下功能：

在运行时判断任意一个对象所属的类

在运行时构造任意一个类的对象

在运行时判段任意一个类所具有的成员变量和方法

在运行时调用任一个对象的方法

在运行时创建新类对象

在使用Java的反射功能时，基本首先都要获取类的Class对象，再通过Class对象获取其他的对象。

这里首先定义用于测试的类:

**[java]** [view plain](http://blog.csdn.net/yongjian1092/article/details/7364451) [copy](http://blog.csdn.net/yongjian1092/article/details/7364451)

1. **class** Type{
2. **public** **int** pubIntField;
3. **public** String pubStringField;
4. **private** **int** prvIntField;
6. **public** Type(){
7. Log("Default Constructor");
8. }
10. Type(**int** arg1, String arg2){
11. pubIntField = arg1;
12. pubStringField = arg2;
14. Log("Constructor with parameters");
15. }
17. **public** **void** setIntField(**int** val) {
18. **this**.prvIntField = val;
19. }
20. **public** **int** getIntField() {
21. **return** prvIntField;
22. }
24. **private** **void** Log(String msg){
25. System.out.println("Type:" + msg);
26. }
27. }
29. **class** ExtendType **extends** Type{
30. **public** **int** pubIntExtendField;
31. **public** String pubStringExtendField;
32. **private** **int** prvIntExtendField;
34. **public** ExtendType(){
35. Log("Default Constructor");
36. }
38. ExtendType(**int** arg1, String arg2){
39. pubIntExtendField = arg1;
40. pubStringExtendField = arg2;
42. Log("Constructor with parameters");
43. }
45. **public** **void** setIntExtendField(**int** field7) {
46. **this**.prvIntExtendField = field7;
47. }
48. **public** **int** getIntExtendField() {
49. **return** prvIntExtendField;
50. }
52. **private** **void** Log(String msg){
53. System.out.println("ExtendType:" + msg);
54. }
55. }

### 网上经典帖子

1、获取类的Class对象

Class 类的实例表示正在运行的 Java 应用程序中的类和接口。获取类的Class对象有多种方式：

|  |  |
| --- | --- |
| 调用getClass | Boolean var1 = true;  Class<?> classType2 = var1.getClass();  System.out.println(classType2);  输出：class java.lang.Boolean |
| 运用.class 语法 | Class<?> classType4 = Boolean.class;  System.out.println(classType4);  输出：class java.lang.Boolean |
| 运用static method Class.forName() | Class<?> classType5 = Class.forName("java.lang.Boolean");  System.out.println(classType5);  输出：class java.lang.Boolean |
| 运用primitive wrapper classes的TYPE 语法  这里返回的是原生类型，和Boolean.class返回的不同 | Class<?> classType3 = Boolean.TYPE;  System.out.println(classType3);  输出：boolean |

|  |  |
| --- | --- |
| public [Field](http://blog.csdn.net/yongjian1092/article/details/7364451" \t "_blank)getField([String](http://blog.csdn.net/yongjian1092/article/details/7364451) name) | 返回一个 Field 对象，它反映此 Class 对象所表示的类或接口的指定公共成员字段 |
| public [Field](http://blog.csdn.net/yongjian1092/article/details/7364451)[] getFields() | 返回一个包含某些 Field 对象的数组，这些对象反映此 Class 对象所表示的类或接口的所有可访问公共字段 |
| public [Field](http://blog.csdn.net/yongjian1092/article/details/7364451" \t "_blank)getDeclaredField([String](http://blog.csdn.net/yongjian1092/article/details/7364451" \t "_blank)name) | 返回一个 Field 对象，该对象反映此 Class 对象所表示的类或接口的指定已声明字段 |
| public [Field](http://blog.csdn.net/yongjian1092/article/details/7364451)[] getDeclaredFields() | 返回 Field 对象的一个数组，这些对象反映此 Class 对象所表示的类或接口所声明的所有字段 |

2、获取类的Fields

可以通过反射机制得到某个类的某个属性，然后改变对应于这个类的某个实例的该属性值。JAVA 的Class<T>类提供了几个方法获取类的属性。

**[java]** [view plain](http://blog.csdn.net/yongjian1092/article/details/7364451) [copy](http://blog.csdn.net/yongjian1092/article/details/7364451)

1. Class<?> classType = ExtendType.**class**;
3. // 使用getFields获取属性
4. Field[] fields = classType.getFields();
5. **for** (Field f : fields)
6. {
7. System.out.println(f);
8. }
10. System.out.println();
12. // 使用getDeclaredFields获取属性
13. fields = classType.getDeclaredFields();
14. **for** (Field f : fields)
15. {
16. System.out.println(f);
17. }

输出：

public int com.quincy.ExtendType.pubIntExtendField

public java.lang.String com.quincy.ExtendType.pubStringExtendField

public int com.quincy.Type.pubIntField

public java.lang.String com.quincy.Type.pubStringField

public int com.quincy.ExtendType.pubIntExtendField

public java.lang.String com.quincy.ExtendType.pubStringExtendField

private int com.quincy.ExtendType.prvIntExtendField

可见getFields和getDeclaredFields区别：

getFields返回的是申明为public的属性，包括父类中定义，

getDeclaredFields返回的是指定类定义的所有定义的属性，不包括父类的。

3、获取类的Method

通过反射机制得到某个类的某个方法，然后调用对应于这个类的某个实例的该方法

Class<T>类提供了几个方法获取类的方法。

|  |  |
| --- | --- |
| public [Method](http://blog.csdn.net/yongjian1092/article/details/7364451" \t "_blank)getMethod([String](http://blog.csdn.net/yongjian1092/article/details/7364451) name,[Class](http://blog.csdn.net/yongjian1092/article/details/7364451)<?>... parameterTypes) | 返回一个 Method 对象，它反映此 Class 对象所表示的类或接口的指定公共成员方法 |
| public [Method](http://blog.csdn.net/yongjian1092/article/details/7364451)[] getMethods() | 返回一个包含某些 Method 对象的数组，这些对象反映此 Class 对象所表示的类或接口（包括那些由该类或接口声明的以及从超类和超接口继承的那些的类或接口）的公共 member 方法 |
| public [Method](http://blog.csdn.net/yongjian1092/article/details/7364451" \t "_blank)getDeclaredMethod([String](http://blog.csdn.net/yongjian1092/article/details/7364451" \t "_blank)name,[Class](http://blog.csdn.net/yongjian1092/article/details/7364451)<?>... parameterTypes) | 返回一个 Method 对象，该对象反映此 Class 对象所表示的类或接口的指定已声明方法 |
| public [Method](http://blog.csdn.net/yongjian1092/article/details/7364451)[] getDeclaredMethods() | 返回 Method 对象的一个数组，这些对象反映此 Class 对象表示的类或接口声明的所有方法，包括公共、保护、默认（包）访问和私有方法，但不包括继承的方法 |

**[java]** [view plain](http://blog.csdn.net/yongjian1092/article/details/7364451) [copy](http://blog.csdn.net/yongjian1092/article/details/7364451)

1. // 使用getMethods获取函数
2. Class<?> classType = ExtendType.**class**;
3. Method[] methods = classType.getMethods();
4. **for** (Method m : methods)
5. {
6. System.out.println(m);
7. }
9. System.out.println();
11. // 使用getDeclaredMethods获取函数
12. methods = classType.getDeclaredMethods();
13. **for** (Method m : methods)
14. {
15. System.out.println(m);
16. }

输出：

public void com.quincy.ExtendType.setIntExtendField(int)

public int com.quincy.ExtendType.getIntExtendField()

public void com.quincy.Type.setIntField(int)

public int com.quincy.Type.getIntField()

public final native void java.lang.Object.wait(long) throws java.lang.InterruptedException

public final void java.lang.Object.wait() throws java.lang.InterruptedException

public final void java.lang.Object.wait(long,int) throws java.lang.InterruptedException

public boolean java.lang.Object.equals(java.lang.Object)

public java.lang.String java.lang.Object.toString()

public native int java.lang.Object.hashCode()

public final native java.lang.Class java.lang.Object.getClass()

public final native void java.lang.Object.notify()

public final native void java.lang.Object.notifyAll()

private void com.quincy.ExtendType.Log(java.lang.String)

public void com.quincy.ExtendType.setIntExtendField(int)

public int com.quincy.ExtendType.getIntExtendField()

4、获取类的Constructor

通过反射机制得到某个类的构造器，然后调用该构造器创建该类的一个实例

Class<T>类提供了几个方法获取类的构造器。

|  |  |
| --- | --- |
| public [Constructor](http://blog.csdn.net/yongjian1092/article/details/7364451)<[T](http://blog.csdn.net/yongjian1092/article/details/7364451)> getConstructor([Class](http://blog.csdn.net/yongjian1092/article/details/7364451)<?>... parameterTypes) | 返回一个 Constructor 对象，它反映此 Class 对象所表示的类的指定公共构造方法 |
| public [Constructor](http://blog.csdn.net/yongjian1092/article/details/7364451)<?>[] getConstructors() | 返回一个包含某些 Constructor 对象的数组，这些对象反映此 Class 对象所表示的类的所有公共构造方法 |
| public [Constructor](http://blog.csdn.net/yongjian1092/article/details/7364451)<[T](http://blog.csdn.net/yongjian1092/article/details/7364451)> getDeclaredConstructor([Class](http://blog.csdn.net/yongjian1092/article/details/7364451)<?>... parameterTypes) | 返回一个 Constructor 对象，该对象反映此 Class 对象所表示的类或接口的指定构造方法 |
| public [Constructor](http://blog.csdn.net/yongjian1092/article/details/7364451)<?>[] getDeclaredConstructors() | 返回 Constructor 对象的一个数组，这些对象反映此 Class 对象表示的类声明的所有构造方法。它们是公共、保护、默认（包）访问和私有构造方法 |

**[java]** [view plain](http://blog.csdn.net/yongjian1092/article/details/7364451) [copy](http://blog.csdn.net/yongjian1092/article/details/7364451)

1. // 使用getConstructors获取构造器
2. Constructor<?>[] constructors = classType.getConstructors();
3. **for** (Constructor<?> m : constructors)
4. {
5. System.out.println(m);
6. }
8. System.out.println();
10. // 使用getDeclaredConstructors获取构造器
11. constructors = classType.getDeclaredConstructors();
12. **for** (Constructor<?> m : constructors)
13. {
14. System.out.println(m);
15. }
17. 输出：
18. **public** com.quincy.ExtendType()
20. **public** com.quincy.ExtendType()
21. com.quincy.ExtendType(**int**,java.lang.String)

5、新建类的实例

通过反射机制创建新类的实例，有几种方法可以创建

|  |  |
| --- | --- |
| 调用无自变量ctor | 1、调用类的Class对象的newInstance方法，该方法会调用对象的默认构造器，如果没有默认构造器，会调用失败.  Class<?> classType = ExtendType.class;  Object inst = classType.newInstance();  System.out.println(inst);  输出：  Type:Default Constructor  ExtendType:Default Constructor  [com.quincy.ExtendType@d80be3](mailto:com.quincy.ExtendType@d80be3)    2、调用默认Constructor对象的newInstance方法  Class<?> classType = ExtendType.class;  Constructor<?> constructor1 = classType.getConstructor();  Object inst = constructor1.newInstance();  System.out.println(inst);  输出：  Type:Default Constructor  ExtendType:Default Constructor  com.quincy.ExtendType@1006d75 |
| 调用带参数ctor | 3、调用带参数Constructor对象的newInstance方法  Constructor<?> constructor2 =  classType.getDeclaredConstructor(int.class, String.class);  Object inst = constructor2.newInstance(1, "123");  System.out.println(inst);  输出：  Type:Default Constructor  ExtendType:Constructor with parameters  com.quincy.ExtendType@15e83f9 |

6、调用类的函数

通过反射获取类Method对象，调用Field的Invoke方法调用函数。

**[java]** [view plain](http://blog.csdn.net/yongjian1092/article/details/7364451) [copy](http://blog.csdn.net/yongjian1092/article/details/7364451)

1. Class<?> classType = ExtendType.**class**;
2. Object inst = classType.newInstance();
3. Method logMethod = classType.<strong>getDeclaredMethod</strong>("Log", String.**class**);
4. logMethod.invoke(inst, "test");
6. 输出：
7. Type:Default Constructor
8. ExtendType:Default Constructor
9. <font color="#ff0000">Class com.quincy.ClassT can not access a member of **class** com.quincy.ExtendType with modifiers "private"</font>
11. <font color="#ff0000">上面失败是由于没有权限调用**private**函数，这里需要设置Accessible为**true**;</font>
12. Class<?> classType = ExtendType.**class**;
13. Object inst = classType.newInstance();
14. Method logMethod = classType.getDeclaredMethod("Log", String.**class**);
15. <font color="#ff0000">logMethod.setAccessible(**true**);</font>
16. logMethod.invoke(inst, "test");

7、设置/获取类的属性值

通过反射获取类的Field对象，调用Field方法设置或获取值

**[java]** [view plain](http://blog.csdn.net/yongjian1092/article/details/7364451) [copy](http://blog.csdn.net/yongjian1092/article/details/7364451)

1. Class<?> classType = ExtendType.**class**;
2. Object inst = classType.newInstance();
3. Field intField = classType.getField("pubIntExtendField");
4. intField.<strong>setInt</strong>(inst, 100);
5. **int** value = intField.<strong>getInt</strong>(inst);

四、动态创建代理类

代理模式：代理模式的作用=为其他对象提供一种代理以控制对这个对象的访问。

代理模式的角色：

抽象角色：声明真实对象和代理对象的共同接口

代理角色：代理角色内部包含有真实对象的引用，从而可以操作真实对象。

真实角色：代理角色所代表的真实对象，是我们最终要引用的对象。

动态代理：

|  |  |
| --- | --- |
| java.lang.reflect.Proxy | Proxy 提供用于创建动态代理类和实例的静态方法，它还是由这些方法创建的所有动态代理类的超类 |
| InvocationHandler | 是代理实例的调用处理程序 实现的接口，每个代理实例都具有一个关联的调用处理程序。对代理实例调用方法时，将对方法调用进行编码并将其指派到它的调用处理程序的 invoke 方法。 |

动态Proxy是这样的一种类:

它是在运行生成的类，在生成时你必须提供一组Interface给它，然后该class就宣称它实现了这些interface。你可以把该class的实例当作这些interface中的任何一个来用。当然，这个Dynamic Proxy其实就是一个Proxy，它不会替你作实质性的工作，在生成它的实例时你必须提供一个handler，由它接管实际的工作。

在使用动态代理类时，我们必须实现InvocationHandler接口

步骤：

1、定义抽象角色

public interface Subject {

public void Request();

}

2、定义真实角色

public class RealSubject implements Subject {

@Override

public void Request() {

// TODO Auto-generated method stub

System.out.println("RealSubject");

}

}

3、定义代理角色

public class DynamicSubject implements InvocationHandler {

private Object sub;

public DynamicSubject(Object obj){

this.sub = obj;

}

@Override

public Object invoke(Object proxy, Method method, Object[] args)

throws Throwable {

// TODO Auto-generated method stub

System.out.println("Method:"+ method + ",Args:" + args);

method.invoke(sub, args);

return null;

}

}

4、通过Proxy.newProxyInstance构建代理对象

RealSubject realSub = new RealSubject();

InvocationHandler handler = new DynamicSubject(realSub);

Class<?> classType = handler.getClass();

Subject sub = (Subject)Proxy.newProxyInstance(classType.getClassLoader(),

realSub.getClass().getInterfaces(), handler);

System.out.println(sub.getClass());

5、通过调用代理对象的方法去调用真实角色的方法。

sub.Request();

输出：

class $Proxy0 新建的代理对象，它实现指定的接口

Method:public abstract void DynamicProxy.Subject.Request(),Args:null

RealSubject 调用的真实对象的方法

# 二．多线程

## 一、概念

### 1.1 基本概念

**进程**是程序执行的一个实例，比如说，10个用户同时执行IE，那么就有10个独立的进程(尽管他们共享同一个可执行代码)。

进程的特点，每一个进程都有自己的独立的一块内存空间、一组资源系统。其内部数据和状态都是完全独立的。怎么看待多进程？进程的优点是提高CPU运行效率，在同一时间内执行多个程序，即并发执行。但是从严格上讲,也不是绝对的同一时刻执行多个程序，只不过CPU在执行时通过时间片等调度算法不同进程高速切换。总结来说：

* 进程由操作系统调度，简单而且稳定
* 进程之间的隔离性好，一个进程崩溃不会影响其它进程
* 单进程编程简单
* 在多核情况下可以把进程和CPU进行绑定，充分利用CPU

当然，多进程也有一些缺点：

* 一般来说进程消耗的内存比较大
* 进程切换代价很高，进程切换也像线程一样需要保持上一个进程的上下文环境
* 在web编程中，如果一个进程来处理一个请求的话，如果要提高并发量就要提高进程数，而进程数量受内存和切换代价限制

**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.

同类的多个线程共享一块内存空间和一组系统资源，线程本身的数据通常只有CPU的寄存器数据，以及一个供程序执行时的堆栈。线程在切换时负荷小，因此，线程也被称为轻负荷进程。一个进程中可以包含多个线程。

在JVM中，本地方法栈、虚拟机栈和程序计数器是线程隔离的，而堆区和方法区是线程共享的。关于JVM中的资源分配，可参考我的另一篇文章【[JVM内存管理及GC](http://blog.csdn.net/suifeng3051/article/details/48292193)】：<http://blog.csdn.net/suifeng3051/article/details/48292193>

### 1.2 进程线程的区别

* 地址空间：进程内的一个执行单元；进程至少有一个线程；它们共享进程的地址空间;而进程有自己独立的地址空间
* 资源拥有：进程是资源分配和拥有的单位，同一个进程内的线程共享进程的资源
* 线程是处理器调度的基本单位，但进程不是
* 二者均可并发执行

注： 关于并发与并行

并发：多个事件在同一时间段内一起执行

并行：多个事件在同一时刻同时执行

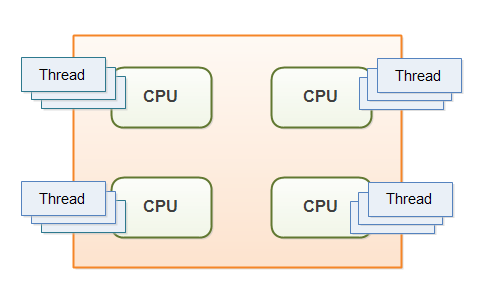
* 1
* 2
* 3

### 1.3 多任务

在一开始，一个计算机只有一个CPU，这个CPU一次也只能运行一个任务。然而随着计算机技术的发展，一个CPU也可以“同时”运行多个任务，这就诞生了多任务。但这里的同时并不是真正的同时，操作系统通过切换各个应用来实现CPU的共享，在CPU内部各个程序其实是交替执行的。

### 1.4 多线程

为了进一步提高CPU利用率，多线程便诞生了。一个程序中可以运行多个线程，多个线程可以同时执行，从整个应用角度上看，这个应用好像独自拥有多个CPU一样。虽然多线程进一步提高了应用的执行效率，但是由于线程之间会共享内存资源，这也会导致一些资源同步问题，另外，线程之间的切换也会对资源有所消耗（后面会讲到）。

这里需要注意的是，如果一台电脑只有一个CPU核心，那么多线程也并没有真正的“同时”运行，它们之间需要通过相互切换来共享CPU核心，所以，只有一个CPU核心的情况下，多线程不会提高应用效率。但是，现代计算机一般都会有多个CPU，并且每个CPU可能还会有多个核心，所以在现代硬件资源条件下，多线程编程可以极大的提高应用效率。   


### 1.5 多线程的调度

在Java程序中，JVM负责线程的调度。线程调度是值按照特定的机制为多个线程分配CPU的使用权。

调度的模式有两种：分时调度和抢占式调度。分时调度是所有线程轮流获得CPU使用权，并平均分配每个线程占用CPU的时间；抢占式调度是根据线程的优先级别来获取CPU的使用权。JVM的线程调度模式采用了抢占式模式。

### 1.6 多线程编程面临的问题

* 更复杂的设计 : 多线程在访问共享数据时需要进行同步（在java中需要使用synchronized关键字），某些情况下需要考虑线程的执行顺序和相互配合
* 上下文切换： 上CPU需要从一个线程切换到另一个线程时，它需要先保存当前线程的本地数据和程序指针，然后再加载要切换线程的本地数据和程序指针
* 更多的系统资源：处理需要CPU时间以外，每个线程还需要额外的内存空间来保存它的本地数据栈，更需要操作系统资源来管理多个线程，所以应用程序的线程数量一定要根据实际情况合理安排

关于多线程编程中的资源同步，请参考另一篇文章【 [Java synchronized 介绍](http://blog.csdn.net/suifeng3051/article/details/48711405)】：<http://blog.csdn.net/suifeng3051/article/details/48711405>

## 二、线程的实现

Java中实现多线程，一种是继承Thread类，一种是实现Runable接口。

## 2.1 继承Thread类

/\*\*

\* 继承Thread类,直接调用run方法

\* \*/

class hello extends Thread {

public hello() {

}

public hello(String name) {

this.name = name;

}

public void run() {

for (int i = 0; i < 5; i++) {

System.out.println(name + "运行 " + i);

}

}

public static void main(String[] args) {

hello h1=new hello("A");

hello h2=new hello("B");

h1.start();

h2.start();

}

private String name;

}

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13
* 14
* 15
* 16
* 17
* 18
* 19
* 20
* 21
* 22
* 23
* 24

注意：在实际启动进程的时候，我们直接调用的并不是Thread子类中run方法，而是调用的Thread线程的start方法，因为线程start运行需要本地操作系统支持，start启动线程会调用操作系统native函数来支持线程运行。

### 2.2 实现runnable接口

package com.heaven.xiancheng;

public class TestRunnable implements Runnable{

private int count =100;

public void run(){

for(int i=0;i<200;i++){

if(count >0){

System. out.println(Thread.currentThread().getName()+ " "+count --);

}

}

}

public static void main(String[] args) {

TestRunnable r= new TestRunnable();

Thread t1= new Thread(r,"A" );

Thread t2= new Thread(r,"B" );

t1.start();

t2.start();

}

}

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13
* 14
* 15
* 16
* 17
* 18
* 19

### 2.3 两者区别

实现Runnable接口比继承Thread类有更多的优势，所以我推荐大家尽量使用实现runnable接口的形式，以下是其优点

- 适合多个相同的程序代码的线程去处理同一个资源

- 可以避免java中的单继承的限制

- 增加程序的健壮性，代码可以被多个线程共享，代码和数据独立。

* 1
* 2
* 3
* 4

## 三、线程的状态

### 3.1 线程的五种状态类型

1. 新建状态（New）：新创建了一个线程对象。

2. 就绪状态（Runnable）：线程对象创建后，其他线程调用了该对象的start()方法。该状态的线程位于可运行线程池中，变得可运行，等待获取CPU的使用权。

3. 运行状态（Running）：就绪状态的线程获取了CPU，执行程序代码。

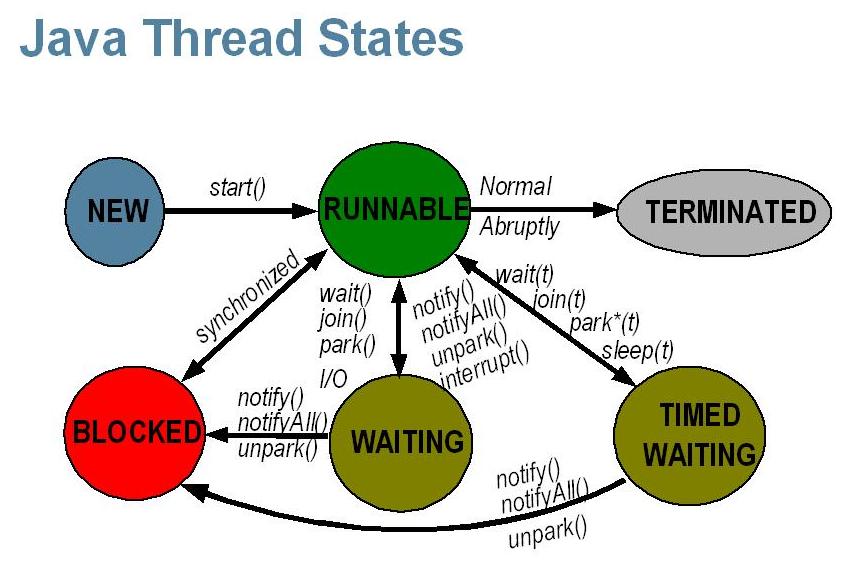
4. 阻塞状态（Blocked）：塞状态是线程因为某种原因放弃CPU使用权，暂时停止运行。直到线程进入就绪状态，才有机会转到运行状态。

5. 死亡状态（Dead）：线程执行完了或者因异常退出了run()方法，该线程结束生命周期。

其中阻塞又可能是由以下几种情况造成：

1. 调用 sleep(毫秒数)，使线程进入“睡眠”状态。在规定的时间内，这个线程是不会运行的。
2. 用 suspend()暂停了线程的执行。除非线程收到 resume()消息，否则不会返回“可运行”状态。
3. 用 wait()暂停了线程的执行。除非线程收到 nofify()或者 notifyAll()消息，否则不会变成“可运行“。
4. 线程正在等候一些 IO（输入输出）操作完成。
5. 线程试图调用另一个对象的“同步”方法，但那个对象处于锁定状态，暂时无法使用。

### 3.2 线程状态图



## 四、线程的阻塞

阻塞指的是暂停一个线程的执行以等待某个条件发生（如某资源就绪）。Java 提供了大量方法来支持阻塞，下面让我们逐一分析。

### 4.1 sleep() 方法

sleep()允许指定以毫秒为单位的一段时间作为参数，它使得线程在指定的时间内进入阻塞状态，不能得到CPU 时间，指定的时间一过，线程重新进入可执行状态。典型地，sleep() 被用在等待某个资源就绪的情形：测试发现条件不满足后，让线程阻塞一段时间后重新测试，直到条件满足为止。

### 4.2 suspend() 和 resume() 方法

两个方法配套使用，suspend()使得线程进入阻塞状态，并且不会自动恢复，必须其对应的resume() 被调用，才能使得线程重新进入可执行状态。典型地，suspend() 和 resume() 被用在等待另一个线程产生的结果的情形：测试发现结果还没有产生后，让线程阻塞，另一个线程产生了结果后，调用 resume() 使其恢复。

### 4.3 yield() 方法

yield() 使得线程放弃当前分得的 CPU 时间，但是不使线程阻塞，即线程仍处于可执行状态，随时可能再次分得 CPU 时间。调用 yield() 的效果等价于调度程序认为该线程已执行了足够的时间从而转到另一个线程。

### 4.4 wait() 和 notify() 方法

两个方法配套使用，wait() 使得线程进入阻塞状态，它有两种形式，一种允许指定以毫秒为单位的一段时间作为参数，另一种没有参数，前者当对应的 notify() 被调用或者超出指定时间时线程重新进入可执行状态，后者则必须对应的 notify() 被调用。初看起来它们与 suspend() 和 resume() 方法对没有什么分别，但是事实上它们是截然不同的。区别的核心在于，前面叙述的所有方法，阻塞时都不会释放占用的锁（如果占用了的话），而这一对方法则相反。

**在这里需要重点介绍下wait()和notify()**

首先，前面叙述的所有方法都隶属于 Thread 类，但是这一对却直接隶属于Object 类，也就是说，所有对象都拥有这一对方法。初看起来这十分不可思议，但是实际上却是很自然的，因为这一对方法阻塞时要释放占用的锁，而锁是任何对象都具有的，调用对象的 wait() 方法导致线程阻塞，并且该对象上的锁被释放。而调用对象的notify()方法则导致因调用该对象的 wait() 方法而阻塞的线程中随机选择的一个解除阻塞（但要等到获得锁后才真正可执行）。

其次，前面叙述的所有方法都可在任何位置调用，但是这一对方法却必须在 synchronized 方法或块中调用，理由也很简单，只有在synchronized 方法或块中当前线程才占有锁，才有锁可以释放。同样的道理，调用这一对方法的对象上的锁必须为当前线程所拥有，这样才有锁可以释放。因此，这一对方法调用必须放置在这样的 synchronized 方法或块中，该方法或块的上锁对象就是调用这一对方法的对象。若不满足这一条件，则程序虽然仍能编译，但在运行时会出现IllegalMonitorStateException 异常。

最后，关于 wait() 和 notify() 方法再说明两点：

1. 调用 notify() 方法导致解除阻塞的线程是从因调用该对象的 wait() 方法而阻塞的线程中随机选取的，我们无法预料哪一个线程将会被选择，所以编程时要特别小心，避免因这种不确定性而产生问题

2. 除了 notify()，还有一个方法 notifyAll() 也可起到类似作用，唯一的区别在于，调用 notifyAll() 方法将把因调用该对象的 wait() 方法而阻塞的所有线程一次性全部解除阻塞。当然，只有获得锁的那一个线程才能进入可执行状态。

* 1
* 2
* 3

[这篇文章](http://blog.csdn.net/suifeng3051/article/details/51852526)中有wait()/notify()的代码实例。

## 五、线程的其它问题

### 5.1 Thread.Join

把指定的线程加入到当前线程，原本两个线程可以并发执行，join之后变成了两个线程顺序执行。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。

public class TestJoin {

public static void main(String[] args) throws InterruptedException {

Thread t1 = new Thread(new JoinA(),"A");

Thread t2 = new Thread(new JoinB(),"B");

t1.start(); //main函数所在的主线程调用了实现了run()方法的JoinA子线程

t1.join(); //主线程获得子线程的锁，阻塞直到子线程完成

t2.start();

}

}

class JoinA implements Runnable {

private int i;

@Override

public void run() {

while (i <= 10) {

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

i++;

}

}

}

class JoinB implements Runnable {

private int i;

@Override

public void run() {

while (i <= 10) {

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

i++;

}

}

}

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13
* 14
* 15
* 16
* 17
* 18
* 19
* 20
* 21
* 22
* 23
* 24
* 25
* 26
* 27
* 28
* 29
* 30
* 31
* 32

执行上面程序从运行结果可以看出两个线程是顺序执行的。其实是当主线程调用子线程的join()方法时，主线程变获得了子线程对象的锁，因此被子线程阻塞直到子线程退出。

我们可以看一下join()的源码：

public final synchronized void join(long millis)

throws InterruptedException {

long base = System.currentTimeMillis();

long now = 0;

if (millis < 0) {

throw new IllegalArgumentException("timeout value is negative");

}

if (millis == 0) {

while (isAlive()) {

wait(0);

}

} else {

while (isAlive()) {

long delay = millis - now;

if (delay <= 0) {

break;

}

wait(delay);

now = System.currentTimeMillis() - base;

}

}

}

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13
* 14
* 15
* 16
* 17
* 18
* 19
* 20
* 21
* 22
* 23
* 24
* 25

join方法实现是通过wait。当main线程调用t.join()时候，main线程会获得线程对象t的锁,调用该对象的wait()，直到该对象唤醒main线程,比如退出后。

### 5.2 线程的休眠与中断

public class TestInterrupt implements Runnable{

@Override

public void run() {

System. out.println("thread run..." );

try {

System. out.println("begin to sleep..." );

Thread. sleep(10000);

} catch (InterruptedException e) {

System. out.println("sleep was interrupted" );

e.printStackTrace();

}

}

public static void main(String[] args) {

TestInterrupt ti= new TestInterrupt();

Thread t= new Thread(ti);

t.start();

try {

Thread. sleep(1000);

} catch (InterruptedException e) {

// TODO Auto-generated catch block

e.printStackTrace();

}

t.interrupt(); //中断线程运行

}

}

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13
* 14
* 15
* 16
* 17
* 18
* 19
* 20
* 21
* 22
* 23
* 24
* 25
* 26
* 27

### 5.3 线程的优先级

public class TestPriority implements Runnable {

@Override

public void run() {

for (int i = 0; i < 5; ++i) {

System. out.println(Thread.currentThread().getName() + "运行" + i);

}

}

public static void main(String[] args) {

TestPriority tp= new TestPriority();

Thread t1= new Thread(tp,"A" );

Thread t2= new Thread(tp,"B" );

Thread t3= new Thread(tp,"C" );

t1.setPriority(1);

t2.setPriority(8);

t3.setPriority(3);

t1.start();

t2.start();

t3.start();

}

}

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13
* 14
* 15
* 16
* 17
* 18
* 19
* 20
* 21

注意：不要误以为优先级越高就先执行，谁先执行还是取决于谁先取得CPU资源。

### 5.4 线程的礼让

在线程操作中，也可以使用yield()方法，将一个线程的操作暂时交给其他线程执行。

public class TestYield implements Runnable{

@Override

public void run() {

for(int i=0;i<10;++i){

System. out.println(Thread.currentThread().getName()+ "运行"+i);

if(i==3){

System. out.println("线程的礼让" );

Thread. yield();

}

}

}

public static void main(String[] args) {

Thread h1= new Thread(new TestYield(),"A");

Thread h2= new Thread(new TestYield(),"B");

h1.start();

h2.start();

}

}

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13
* 14
* 15
* 16
* 17
* 18
* 19

### 5.5 同步与死锁

线程同步问题，当各个线程共用一个资源时，有可能导致线程同步问题。在JAVA中，是没有类似于PV操作、进程互斥等相关的方法的。JAVA的进程同步是通过synchronized()来实现的，需要说明的是，JAVA的synchronized()方法类似于操作系统概念中的互斥内存块，在JAVA中的Object类型中，都是带有一个内存锁的，在有线程获取该内存锁后，其它线程无法访问该内存，从而实现JAVA中简单的同步、互斥操作。关于这部分内容，请参考我的另一篇文章：   
【[Javasynchronized介绍](http://blog.csdn.net/suifeng3051/article/details/48711405)】：<http://blog.csdn.net/suifeng3051/article/details/48711405>

参考文章：   
<http://blog.csdn.net/bzwm/article/details/3881392>   
<http://www.cnblogs.com/techyc/p/3286678.html>