# 一、动态代理-你不必知道我的存在

**理解springAOP**

## 举例

要计算某个类的某个方法运行了多长时间？比如Tank类的move方法，要计算坦克移动了多长时间。

**接口Moveable**，里面一个move() 方法。

|  |
| --- |
| **public** **interface** Moveable {  **void** move();  } |

**实现类Tank，实现Moveable接口**

|  |
| --- |
| **public** **class** Tank **implements** Moveable{  @Override  **public** **void** move() {  //计算方法运行了多长时间  **long** start = System.*currentTimeMillis*();  System.***out***.println("Tank Moving...");  **try** {  Thread.*sleep*(**new** Random().nextInt(10000));  } **catch** (InterruptedException e) {  e.printStackTrace();  }  **long** end = System.*currentTimeMillis*();  System.***out***.println("time:"+(end-start));  }  } |

如果你能修改源代码，可以在move方法内部的前后，计算开始、结束时间，相减就是move方法执行的时间。

**如果你不能修改源代码，怎么办？**

1. **用继承**

新建Tank2 ，继承Tank，重写move()方法，在move方法的前后，加上计算时间的逻辑

|  |
| --- |
| **public** **class** Tank2 **extends** Tank{  @Override  **public** **void** move() {  //计算方法运行了多长时间  **long** start = System.*currentTimeMillis*();  **super**.move();  **long** end = System.*currentTimeMillis*();  System.***out***.println("time:"+(end-start));  }  } |

1. **用聚合**

新建Tank3 ，实现Moveable接口，重写move()方法。Tank3有一个成员变量Tank，move()方法里调用Tank的move方法，Tank3其实就是Tank的一个代理。

|  |
| --- |
| **public** **class** Tank3 **implements** Moveable{  Tank t;  **public** Tank3(Tank t) {  **super**();  **this**.t = t;  }  @Override  **public** **void** move() {  //计算方法运行了多长时间  **long** start = System.*currentTimeMillis*();  t.move();  **long** end = System.*currentTimeMillis*();  System.***out***.println("time:"+(end-start));  }  } |

**继承和聚合，都能实现计算move方法运行时长的问题，但是聚合更灵活。**

**Tank3和Tank2都是Tank的一个代理。这里就是静态的代理**

假设现在想要实现一个功能，先记录运行时间，再记录日志，那么如果用继承，就得这样写：

新建一个类，继承Tank2（记录运行时间的代理）

|  |
| --- |
| **public** **class** Tank2\_1 **extends** Tank2{  @Override  **public** **void** move() {  //记录日志  System.***out***.println("Tank start....");  **super**.move();  System.***out***.println("Tank end....");  }  } |

这样Test测试打印：

Tank start....

Tank Moving...

time:9528

Tank end....

如果想先记录时间，再记录日志呢？就要再新建一个类，顺序是，用时间的代理类，去继承日志的代理类，如果还有其他的代理，如权限检查的代理，等等，调换记录顺序，会更麻烦。。。代理类会无限制的多下去。

如果用聚合实现代理之间的组合呢？

用聚合实现代理，代理对象和**被代理对**象要实现同一个接口：

TankLogProxy：

|  |
| --- |
| **public** **class** TankLogProxy **implements** Moveable{  Moveable m;  **public** TankLogProxy(Moveable m) {  **super**();  **this**.m = m;  }  @Override  **public** **void** move() {  System.***out***.println("Tank start....");  m.move();  System.***out***.println("Tank end....");  }  } |

TankTimeProxy：

|  |
| --- |
| **public** **class** TankTimeProxy **implements** Moveable{  Moveable m;    **public** TankTimeProxy(Moveable m) {  **super**();  **this**.m = m;  }  @Override  **public** **void** move() {  //计算方法运行了多长时间  **long** start = System.*currentTimeMillis*();  System.***out***.println("start:"+start);  m.move();  **long** end = System.*currentTimeMillis*();  System.***out***.println("time:"+(end-start));  }  } |

测试：

先时间，再日志:

|  |
| --- |
| Tank tank = **new** Tank();  TankLogProxy tlp = **new** TankLogProxy(tank);  TankTimeProxy ttp = **new** TankTimeProxy(tlp);  ttp.move(); |

打印：

start:1581495475807

Tank start....

Tank Moving...

Tank end....

time:6543

先日志，再时间，只要调换测试类的代理顺序即可：

|  |
| --- |
| Tank tank = **new** Tank();  TankTimeProxy ttp = **new** TankTimeProxy(tank);  TankLogProxy tlp = **new** TankLogProxy(ttp);  tlp.move(); |

打印结果：

Tank start....

start:1581495785543

Tank Moving...

time:2139

Tank end....

可以看到，用聚合实现代理，要比用继承灵活的多！

**第二个问题，先只考虑TimeProxy**

**Moveable接口：新添加stop()方法**

|  |
| --- |
| **public** **interface** Moveable {  **void** move();  **void** stop();  } |

**Tank也实现stop方法**

|  |
| --- |
| @Override  **public** **void** stop() {  System.***out***.println("Tank Stoping...");  } |

TankTimeProxy也记录stop方法的运行时间：

|  |
| --- |
| **public** **class** TankTimeProxy **implements** Moveable{  Moveable m;    **public** TankTimeProxy(Moveable m) {  **super**();  **this**.m = m;  }  @Override  **public** **void** move() {  //计算方法运行了多长时间  **long** start = System.*currentTimeMillis*();  System.***out***.println("start:"+start);  m.move();  **long** end = System.*currentTimeMillis*();  System.***out***.println("time:"+(end-start));  }  @Override  **public** **void** stop() {  //计算方法运行了多长时间  **long** start = System.*currentTimeMillis*();  System.***out***.println("start:"+start);  m.stop();  **long** end = System.*currentTimeMillis*();  System.***out***.println("time:"+(end-start));    }  } |

如果一段代码重复出现了多次，就要考虑封装了，move和stop方法，都有计算时间的逻辑，可以考虑将他们封装成为方法。

现在如果要有个Car类的move方法，要记录汽车移动的时间，就需要再写个CarProxy，

如果再有个Animal类的eat方法，要记录动物吃的时间，就要有个AnimalProxy

...... 如果一个系统有100个类，就要有100个代理类出现，又出现了类爆炸。

所以现在有个需求就是：

能不能产生一个代理类，可以给所有的类做代理呢？？？

从上边的例子可以看出，用聚合产生代理，需要代理类和被代理类实现同一个接口。

现在假设，假设被代理的类都实现某一个接口，（Spring里面也是这么要求的，Spring也能用继承实现代理但是不推荐），就能给这个类生成代理。

## 下面模拟JDK的实现

站在使用者的角度，有一个专门产生代理的类，假设现在只是产生时间的代理

|  |
| --- |
| //站在使用者的角度，动态代理，Proxy产生一个代理类的对象，你根本看不到这个代理类的名字  Moveable m = (Moveable)Proxy.*newProxyInstance*();  m.move(); |

|  |
| --- |
| /\*\*  \* 产生代理的类  \* **@author** dev  \*  \*/  **public** **class** Proxy {  **public** **static** Object newProxyInstance(){  //只要能动态的 编译这段代码，就能动态的产生代理类！类的名字无所谓  //动态编译的技术：JDK6 Compiler API，CGLib（用到了ASM） ，ASM  //（CGLib、ASM不用源码来编译，能直接生成二进制文件，因为java的二进制文件格式是公开的）  //Spring内部，如果是实现接口就是用的JDK本身的API产生代理，否则就用CGLib  //换行字符串  String rt = "\r\n";  String src =  "package com.lhy.proxy;"+ rt +  "public class TankTimeProxy implements Moveable{"+rt +  " Moveable m;"+rt +    " public TankTimeProxy(Moveable m) {"+rt +  " super();"+rt +  " this.m = m;"+rt +  " }"+rt +  " @Override" +rt +  " public void move() {" +rt +  //计算方法运行了多长时间  " long start = System.currentTimeMillis();" +rt +  " System.out.println(\"start:\"+start);" +rt +  " m.move();"+rt +  " long end = System.currentTimeMillis();"+rt +  " System.out.println(\"time:\"+(end-start));"+rt +  " }"+rt +  "}";  **return** **null**;  }  } |

新建测试类，测试用java代码产生代理类，然后进行编译，然后load到内存进行加载，用反射新建一个代理类的对象。

|  |
| --- |
| **package** com.lhy.proxy;  **import** java.io.File;  **import** java.io.FileWriter;  **import** java.lang.reflect.Constructor;  **import** java.net.URL;  **import** java.net.URLClassLoader;  **import** javax.tools.JavaCompiler;  **import** javax.tools.JavaCompiler.CompilationTask;  **import** javax.tools.StandardJavaFileManager;  **import** javax.tools.ToolProvider;  **public** **class** TestCompiler {  **public** **static** **void** main(String[] args) **throws** Exception{  String rt = "\r\n";  String src =  "package com.lhy.proxy;"+ rt +  "public class TankTimeProxy implements Moveable{"+rt +  " Moveable m;"+rt +    " public TankTimeProxy(Moveable m) {"+rt +  " super();"+rt +  " this.m = m;"+rt +  " }"+rt +  " @Override" +rt +  " public void move() {" +rt +  //计算方法运行了多长时间  " long start = System.currentTimeMillis();" +rt +  " System.out.println(\"start:\"+start);" +rt +  " m.move();"+rt +  " long end = System.currentTimeMillis();"+rt +  " System.out.println(\"time:\"+(end-start));"+rt +  " }"+rt +  "}";    //1，生成代理类  String fileName = System.*getProperty*("user.dir")  +"/src/com/lhy/proxy/TankTimeProxy.java";//获取项目根路径  File file = **new** File(fileName);  FileWriter fw = **new** FileWriter(file);  fw.write(src);  fw.flush();  fw.close();    //2，将生成的类进行编译成class文件  JavaCompiler compiler = ToolProvider.*getSystemJavaCompiler*();//拿到系统默认的编译器（其实就是javac）  StandardJavaFileManager fileMgr = compiler.getStandardFileManager(**null**, **null**, **null**);//诊断监听器；语言；编码  Iterable units = fileMgr.getJavaFileObjects(fileName);  CompilationTask task = compiler.getTask(**null**, fileMgr, **null**, **null**, **null**, units);  task.call();  fileMgr.close();    //3，将class load到内存  URL[] urls = **new** URL[]{**new** URL("file:/"+ System.*getProperty*("user.dir")+"/src")};  URLClassLoader urlClassLoader = **new** URLClassLoader(urls);  Class clazz = urlClassLoader.loadClass("com.lhy.proxy.TankTimeProxy");  //System.out.println(clazz);  //4，，创建一个对象  //不能用 clazz.newInstance();创建对象因为它会调用空构造方法  Constructor<Moveable> constructor = clazz.getConstructor(Moveable.**class**);//获取某个类型参数的构造器  Moveable m = constructor.newInstance(**new** Tank());//  m.move();    }  } |

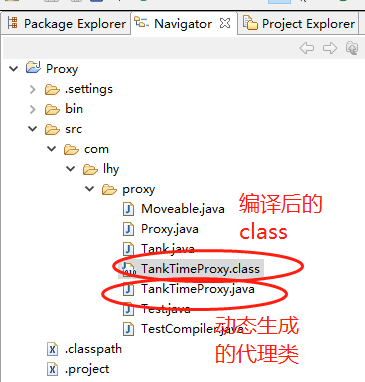
打印结果

start:1581515256858

Tank Moving...

time:6611

生成的代理类和编译后的class



测试结果可以看出，可以动态产生代理类，你看不到代理类的名字，你只要调用Proxy.newProxyInstance()方法就能返回一个代理类，这就是动态代理，用完你就可以吧代理类的代码删了。

但是现在产生的代理 是实现了Moveable接口的代理，要想产生实现任意接口的代理怎么办呢？ 只要把接口传给产生代理的方法就可以了。而且 ，接口的方法，也要动态生成，这就需要用到反射了：

反射拿到接口的方法代码：

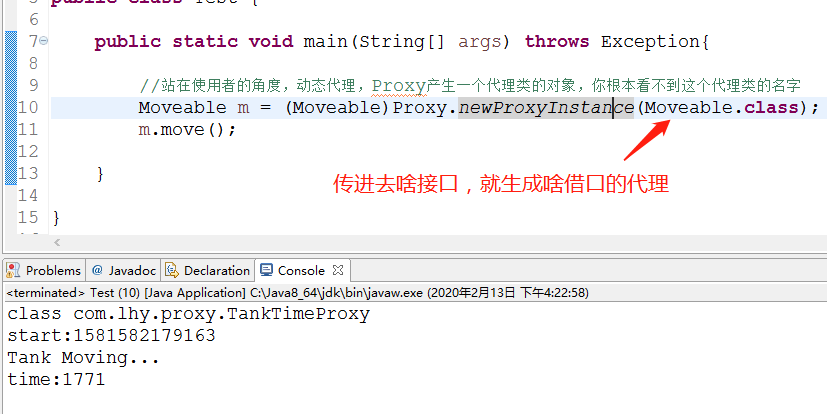
|  |
| --- |
| Method[] methods = Moveable.**class**.getMethods();  **for**(Method m : methods){  System.***err***.println(m.getName());//move  } |

修改后的产生代理的类：

用反射拿到接口的所有方法，动态的构建代理类的方法

|  |
| --- |
| **package** com.lhy.proxy;  **import** java.io.File;  **import** java.io.FileWriter;  **import** java.lang.reflect.Constructor;  **import** java.lang.reflect.Method;  **import** java.net.URL;  **import** java.net.URLClassLoader;  **import** javax.tools.JavaCompiler;  **import** javax.tools.JavaCompiler.CompilationTask;  **import** javax.tools.StandardJavaFileManager;  **import** javax.tools.ToolProvider;  /\*\*  \* 产生代理的类  \* **@author** dev  \*  \*/  **public** **class** Proxy {  **public** **static** Object newProxyInstance(Class interfaces) **throws** Exception{//动态传入接口，其实jdk可以传多个接口  //换行字符串  String rt = "\r\n";  String methodStr = "";  //反射拿到接口的所有的方法  Method[] methods = interfaces.getMethods();  **for**(Method m : methods){  methodStr += "@Override"+rt +  "public void "+ m.getName()+ "() {"+  //计算方法运行了多长时间  " long start = System.currentTimeMillis();" +rt +  " System.out.println(\"start:\"+start);" +rt +  " m."+m.getName() +"();" +rt +  " long end = System.currentTimeMillis();"+rt +  " System.out.println(\"time:\"+(end-start));"+rt +  "}";  }    //只要能动态的 编译这段代码，就能动态的产生代理类！类的名字无所谓  //动态编译的技术：JDK6 Compiler API，CGLib（用到了ASM） ，ASM  //（CGLib、ASM不用源码来编译，能直接生成二进制文件，因为java的二进制文件格式是公开的）  //Spring内部，如果是实现接口就是用的JDK本身的API产生代理，否则就用CGLib    String src =  "package com.lhy.proxy;"+ rt +  "public class TankTimeProxy implements "+ interfaces.getName() +"{"+rt +  " Moveable m;"+rt +    " public TankTimeProxy(Moveable m) {"+rt +  " super();"+rt +  " this.m = m;"+rt +  " }"+rt +  methodStr +  "}";    //1，生成代理类  String fileName = System.*getProperty*("user.dir")  +"/src/com/lhy/proxy/TankTimeProxy.java";//获取项目根路径  File file = **new** File(fileName);  FileWriter fw = **new** FileWriter(file);  fw.write(src);  fw.flush();  fw.close();    //2，将生成的类进行编译成class文件  JavaCompiler compiler = ToolProvider.*getSystemJavaCompiler*();//拿到系统默认的编译器（其实就是javac）  StandardJavaFileManager fileMgr = compiler.getStandardFileManager(**null**, **null**, **null**);//诊断监听器；语言；编码  Iterable units = fileMgr.getJavaFileObjects(fileName);  CompilationTask task = compiler.getTask(**null**, fileMgr, **null**, **null**, **null**, units);  task.call();  fileMgr.close();    //3，将class load到内存  URL[] urls = **new** URL[]{**new** URL("file:/"+ System.*getProperty*("user.dir")+"/src")};  URLClassLoader urlClassLoader = **new** URLClassLoader(urls);  Class clazz = urlClassLoader.loadClass("com.lhy.proxy.TankTimeProxy");  System.***out***.println(clazz);  //4，，创建一个对象  //不能用 clazz.newInstance();创建对象因为它会调用空构造方法  Constructor<Moveable> constructor = clazz.getConstructor(Moveable.**class**);//获取某个类型参数的构造器  Object obj = constructor.newInstance(**new** Tank());//    **return** obj;  }  } |

测试



产生的代理类TankTimeProxy：

|  |
| --- |
| **package** com.lhy.proxy;  **public** **class** TankTimeProxy **implements** com.lhy.proxy.Moveable {  Moveable m;  **public** TankTimeProxy(Moveable m) {  **super**();  **this**.m = m;  }  @Override  **public** **void** move() {  **long** start = System.*currentTimeMillis*();  System.***out***.println("start:" + start);  m.move();  **long** end = System.*currentTimeMillis*();  System.***out***.println("time:" + (end - start));  }  } |

结论：

到目前为止，已经可以动态创建某个接口的代理类，并调用代理类的方法，但是目前的代理只是实现了时间的代理，代理的逻辑是写死的，肯定不能写死，那怎么写活呢？

思路：代理的逻辑，可以自己指定

写一个处理代理逻辑的接口

|  |
| --- |
| **import** java.lang.reflect.Method;  **public** **interface** InvocationHandler {  /\*\*  \* 代理执行的逻辑  \* **@param** o 方法所属的对象  \* **@param** m 要执行的方法  \*/  **public** **void** invoke(Object o,Method m);  } |

时间的代理类的处理逻辑，实现InvocationHandler 接口

|  |
| --- |
| **import** java.lang.reflect.Method;  **public** **class** TimeHandler **implements** InvocationHandler{  //被代理类  **private** Object target;    **public** TimeHandler(Object target) {  **super**();  **this**.target = target;  }  @Override  **public** **void** invoke(Object o,Method m) {  **long** start = System.*currentTimeMillis*();  System.***out***.println("start:" + start);  **try** {  m.invoke(target, **new** Object[]{});  } **catch** (Exception e) {  e.printStackTrace();  }  **long** end = System.*currentTimeMillis*();  System.***out***.println("time:" + (end - start));    }  } |

产生代理类的Proxy：

|  |
| --- |
| **import** java.io.File;  **import** java.io.FileWriter;  **import** java.lang.reflect.Constructor;  **import** java.lang.reflect.Method;  **import** java.net.URL;  **import** java.net.URLClassLoader;  **import** javax.tools.JavaCompiler;  **import** javax.tools.JavaCompiler.CompilationTask;  **import** javax.tools.StandardJavaFileManager;  **import** javax.tools.ToolProvider;  /\*\*  \* 产生代理的类  \* **@author** dev  \*  \*/  **public** **class** Proxy {  /\*\*  \*  \* **@param** interfaces 代理实现的接口  \* **@param** h 代理处理逻辑  \* **@return**  \* **@throws** Exception  \*/  **public** **static** Object newProxyInstance(Class interfaces,InvocationHandler h) **throws** Exception{//动态传入接口，其实jdk可以传多个接口  //换行字符串  String rt = "\r\n";  String methodStr = "";  //反射拿到接口的所有的方法  Method[] methods = interfaces.getMethods();  **for**(Method m : methods){  methodStr += "@Override"+rt +  "public void "+ m.getName()+ "() {"+  " try{"+rt+  " Method md = "+ interfaces.getName()+".class.getMethod(\""+m.getName()+"\");"+rt+  " h.invoke(this,md);"+rt+ //this->代理对象  " }catch(Exception e){e.printStackTrace();}"+  "}";  }    //只要能动态的 编译这段代码，就能动态的产生代理类！类的名字无所谓  //动态编译的技术：JDK6 Compiler API，CGLib（用到了ASM） ，ASM  //（CGLib、ASM不用源码来编译，能直接生成二进制文件，因为java的二进制文件格式是公开的）  //Spring内部，如果是实现接口就是用的JDK本身的API产生代理，否则就用CGLib    String src =  "package com.lhy.proxy;"+ rt +  "import java.lang.reflect.Method;"+rt+  "public class $Proxy1 implements "+ interfaces.getName() +"{"+rt +  " com.lhy.proxy.InvocationHandler h;"+rt+  " public $Proxy1(InvocationHandler h) {"+rt +  " this.h = h;"+rt +  " }"+rt +  methodStr +  "}";    //1，生成代理类  String fileName = System.*getProperty*("user.dir")  +"/src/com/lhy/proxy/$Proxy1.java";//获取项目根路径  File file = **new** File(fileName);  FileWriter fw = **new** FileWriter(file);  fw.write(src);  fw.flush();  fw.close();    //2，将生成的类进行编译成class文件  JavaCompiler compiler = ToolProvider.*getSystemJavaCompiler*();//拿到系统默认的编译器（其实就是javac）  StandardJavaFileManager fileMgr = compiler.getStandardFileManager(**null**, **null**, **null**);//诊断监听器；语言；编码  Iterable units = fileMgr.getJavaFileObjects(fileName);  CompilationTask task = compiler.getTask(**null**, fileMgr, **null**, **null**, **null**, units);  task.call();  fileMgr.close();    //3，将class load到内存  URL[] urls = **new** URL[]{**new** URL("file:/"+ System.*getProperty*("user.dir")+"/src")};  URLClassLoader urlClassLoader = **new** URLClassLoader(urls);  Class clazz = urlClassLoader.loadClass("com.lhy.proxy.$Proxy1");    //4，，创建一个对象  //不能用 clazz.newInstance();创建对象因为它会调用空构造方法  Constructor constructor = clazz.getConstructor(InvocationHandler.**class**);//获取某个类型参数的构造器  Object obj = constructor.newInstance(h);//    **return** obj;  }  } |

测试代码：

|  |
| --- |
| **public** **static** **void** main(String[] args) **throws** Exception{  InvocationHandler h = **new** TimeHandler(**new** Tank());  //站在使用者的角度，动态代理，Proxy产生一个代理类的对象，你根本看不到这个代理类的名字  Moveable m = (Moveable)Proxy.*newProxyInstance*(Moveable.**class**,h);  m.move();  } |

打印结果：

start:1581596505206

Tank Moving...

time:5193

产生的代理类$Proxy1:

|  |
| --- |
| **import** java.lang.reflect.Method;  **public** **class** $Proxy1 **implements** com.lhy.proxy.Moveable {  com.lhy.proxy.InvocationHandler h;  **public** $Proxy1(InvocationHandler h) {  **this**.h = h;  }  @Override  **public** **void** move() {  **try** {  Method md = com.lhy.proxy.Moveable.**class**.getMethod("move");  h.invoke(**this**, md);  } **catch** (Exception e) {  e.printStackTrace();  }  }  } |

现在，想实现什么代理，只要实现InvocationHandler接口，自定义代理的处理逻辑，即可实现代理，这就是动态代理。

## 实际举例说明

UserMgr接口：

|  |
| --- |
| **public** **interface** UserMgr {  **void** addUser();  } |

UserMgr实现类

|  |
| --- |
| **public** **class** UserMgrImpl **implements** UserMgr {  @Override  **public** **void** addUser() {  System.***err***.println("插入到数据库user表");  System.***err***.println("记录到日志表");  }  } |

事务代理处理逻辑TransitionHandler：

|  |
| --- |
| **import** java.lang.reflect.Method;  **import** com.lhy.proxy.InvocationHandler;  **public** **class** TransitionHandler **implements** InvocationHandler{  **private** Object target;  **public** TransitionHandler(Object target) {  **this**.target = target;  }  @Override  **public** **void** invoke(Object o, Method m) {  System.***err***.println("事务开始....");  **try** {  m.invoke(target, **new** Object[]{});  } **catch** (Exception e) {  e.printStackTrace();  System.***err***.println("事务回滚....");  }  System.***err***.println("事务提交....");  }  } |

测试类：

|  |
| --- |
| **import** com.lhy.proxy.InvocationHandler;  **import** com.lhy.proxy.Proxy;  **public** **class** Client {  **public** **static** **void** main(String[] args) **throws** Exception{  UserMgr userMgr = **new** UserMgrImpl();  InvocationHandler h = **new** TransitionHandler(userMgr);  UserMgr proxy = (UserMgr)Proxy.*newProxyInstance*(UserMgr.**class**, h);  proxy.addUser();  }  } |

运行：

事务开始....

插入到数据库user表

记录到日志表

事务提交....

产生的事务代理类：

|  |
| --- |
| **import** java.lang.reflect.Method;  **public** **class** $Proxy1 **implements** com.lhy.proxy.test.UserMgr {  com.lhy.proxy.InvocationHandler h;  **public** $Proxy1(InvocationHandler h) {  **this**.h = h;  }  @Override  **public** **void** addUser() {  **try** {  Method md = com.lhy.proxy.test.UserMgr.**class**.getMethod("addUser");  h.invoke(**this**, md);  } **catch** (Exception e) {  e.printStackTrace();  }  }  } |

从运行结果可以看出，已经控制了事务!

动态代理：不用修改原来的实现的代码，就能在原来基础上前后插入一些内容

AOP：可插拔的，可以将代理配置在配置文件，想实现什么样的代理就实现什么样的代理。代理之间是可以叠加的

AOP的运用：日志、事务、权限。。。。

# 工厂模式-理解Spring的Bean工厂

封装“老张开车去东北”里面的交通工具，封装交通工具Car

## 只给司机一辆车（单例、多例）

### 顺带讲解单例

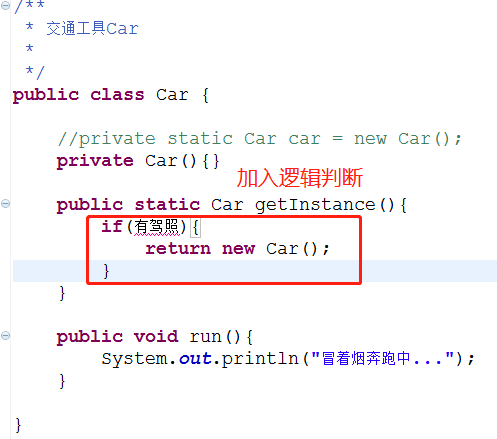
要求只能有一辆车，别人不能new Car，只有Car自己能控制newCar的逻辑。私有化构造方法，别人就不能new了。

|  |
| --- |
| /\*\*  \* 交通工具Car  \*  \*/  **public** **class** Car {    //private static Car car = new Car();  **private** Car(){}    **public** **static** Car getInstance(){  **return** **new** Car();  }    **public** **void** run(){  System.***out***.println("冒着烟奔跑中...");  }  } |

工厂就是自主生产自己的产品，不再依赖于new。比如你想new我家的一个抽屉，你想拿钱就拿钱， 肯定不行。

但是我要给你提供一个方法：getChouTi(); 我就能在get方法里面做各种各样的限制了。

比如返回Car的getInstance方法，可以做逻辑判断

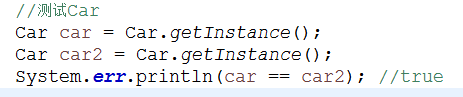


再回到上面的要求，只有一辆车，这么做：自己new一个Car，调用getInstance时候，返回这个Car

|  |
| --- |
| **public** **class** Car {    **private** **static** Car *car* = **new** Car();  **private** Car(){}    **public** **static** Car getInstance(){  **return** *car*;  }    **public** **void** run(){  System.***out***.println("冒着烟奔跑中...");  }  } |

测试

getInstance两次看是不是一辆车：



打印true，说明是一辆车

这个模式叫 单例，又有人叫这个getInstance方法叫静态工厂方法。

任何方法，里面控制了产生对象的逻辑，都可以叫工厂方法。

### 多例

如果Car类里面返回的不是一个Car，里面有一个List装了一堆的Car，getInstance的时候随机返回一个，这个又有人起了个名字 叫---多例

|  |
| --- |
| **public** **class** Car {    //private static Car car = new Car();  **private** **static** List<Car> *cars* = **new** ArrayList<>();    **static**{  //静态初始化cars  *cars*.add(**new** Car());  *cars*.add(**new** Car());  }    **private** Car(){}    **public** **static** Car getInstance(){  //return car;  //随机返回一个Car，这里就不随机了  **return** *cars*.get(1);  }    **public** **void** run(){  System.***out***.println("冒着烟奔跑中...");  }  } |

JDBC连接池，里面装的Connection，就是多例。

## 任意定制交通工具的类型和生产过程

自然就想起了多态，抽取一个借口：Moveable，然后让Car实现Moveable接口：

|  |
| --- |
| **public** **interface** Moveable {  **void** run();  } |

Car的实现：

|  |
| --- |
| **public** **class** Car **implements** Moveable{  @Override  **public** **void** run() {  System.***out***.println("冒着烟奔跑中...");  }  } |

飞机的实现：

|  |
| --- |
| **public** **class** Plane **implements** Moveable{  @Override  **public** **void** run() {  System.***out***.println("扇着翅膀飞呀飞...");  }  } |

测试类：

调用的时候，父类引用指向子类对象，多态，我new谁，就调用的是谁，很随意就换了交通工具：

|  |
| --- |
| Moveable m = **new** Car();  m.run();  m = **new** Plane();  m.run();  打印结果：  冒着烟奔跑中...  扇着翅膀飞呀飞... |

还有其他任何交通工具 比如交通工具是哈利波特的扫帚，就直接实现Moveable接口，你就可以直接new Broom(); 了 。

现在存在的问题就是，可以任意的new 交通工具，构造方法是公开的。现在想对任意交通工具的生产过程也能够定制的话。有了上面单例的思路，现在第一个想到的还是，在交通工具类里面写一个静态的方法控制new 的过程。这里比如飞机，把产生飞机的过程单独一个类拿出来，比如叫飞机工厂PlaneFactory：：

|  |
| --- |
| //飞机工厂类  **public** **class** PlaneFactory {  **public** Plane createPlane(){  //单例、多例、条件检查自己控制  **return** **new** Plane();  }  } |

测试 ：

|  |
| --- |
| //飞机工厂  PlaneFactory factory = **new** PlaneFactory();  Moveable m = factory.createPlane();  m.run();  打印结果：  扇着翅膀飞呀飞... |

如果现在想有一个Car工厂，那么很简单，就是这样：

|  |
| --- |
| //Car工厂类  **public** **class** CarFactory {  **public** Car createPlane(){  //单例、多例、条件检查自己控制  **return** **new** Car();  }  } |

测试代码：

以前是开着飞机，现在想换成开Car，需要把飞机工厂换成了Car工厂，调用他的creteCar方法。这样太别扭了。

|  |
| --- |
| //飞机工厂  //PlaneFactory factory = new PlaneFactory();  //换成Car工厂,整个工厂方法都得换  CarFactory factory = **new** CarFactory();  Moveable m = factory.createCar();  m.run(); |

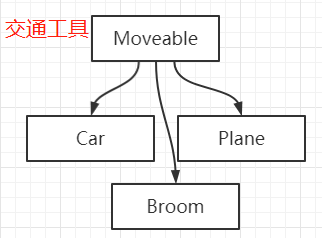
有没有什么办法，从飞机换成Car的时候，只换工厂的实现就行呢？自然就想起了多态，有多态就得有父类、子类。所以工厂类需要有一个父类。Factory本来是产生交通工具的，抽象出一个产生交通工具的工厂：

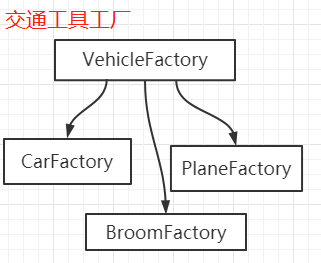
|  |
| --- |
| //交通工具工厂  **public** **abstract** **class** VehicleFactory {  //具体生成什么交通工具由子类决定，这里是抽象的。  **public** **abstract** Moveable create();  } |

这时候让CarFactory和PlaneFactory去继承VehicleFactory ：

|  |
| --- |
| //Car工厂类  **public** **class** CarFactory **extends** VehicleFactory{    @Override  **public** Moveable create() {  //单例、多例、条件检查自己控制  **return** **new** Car();  }  } |
| //飞机工厂类  **public** **class** PlaneFactory **extends** VehicleFactory {    @Override  **public** Moveable create() {  //单例、多例、条件检查自己控制  **return** **new** Plane();  }  } |

换了工厂的实现，就可以换交通工具了，比如你加了一个哈利波特的魔法扫帚，需要加一个Broom类实现Moveable接口，和一个BroomFactory工厂类继承VehicleFactory就可以了。





|  |
| --- |
| //扫帚  **public** **class** Broom **implements** Moveable{  @Override  **public** **void** run() {  System.***out***.println("扫帚摇着尾巴呼呼呼...");  }  } |

|  |
| --- |
| //扫帚工厂类  **public** **class** BroomFactory **extends** VehicleFactory {    @Override  **public** Moveable create() {  //单例、多例、条件检查自己控制  **return** **new** Broom();  }  } |

此时测试代码就成了这样子：

|  |
| --- |
| //机车工厂，new飞机工厂实例  VehicleFactory factory = **new** PlaneFactory();  Moveable m = factory.create();  m.run();  //换成Car工厂  factory = **new** CarFactory();  m = factory.create();  m.run();  //换成扫帚工厂  factory = **new** BroomFactory();  m = factory.create();  m.run(); |

打印结果；

扇着翅膀飞呀飞...

冒着烟奔跑中...

扫帚摇着尾巴呼呼呼...

在某一个维度上有了可扩展了，不仅可以控制产生交通工具的类型，还可以控制产生交通工具的生产过程。需要改的只有一个地方：要是需要换交通工具，站在客户角度，需要改的只有交通工具的工厂，其他地方都不用动。如果用了配置文件的话，代码都不用该，只改配置文件即可。后面再说。

关于抽象类和接口的选择：

比如上面的交通工具工厂，是一个抽象类，也可以设计成接口，么问题。

假如这个概念在我们脑子是确确实实存在的，就用抽象类，

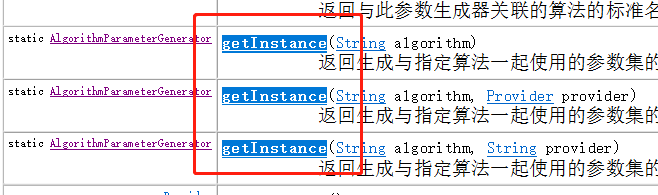
假如这个概念只是某些方面的特性：比如会飞的，会跑的，就用接口

假如两个概念模糊的时候，不知道选择哪个的时候，就用接口，原因是，从实现了这个接口后，还能从其它的抽象类继承，更灵活。

## 抽象工厂

看一下JDK里面，先看getInstance方法，有一大堆，不同类里面有同样的方法getInstance。

这些大多数都是静态的工厂方法，是不是单例不一定，得看具体的实现。



各种各样的Factory也很多。像下面的加密的key，就不适合new，用一个工厂去实现它，产生的时候可以实现各种各种算法，检测各种资质，new的话构造方法只能是写死的，用工厂的话，实现一个子类的时候还可以控制生产过程，更灵活。



总而言之，getInstance和Factory，JDK里面很常用。下面开始说抽象工厂。

回到最原始的状态，我们有一辆车Car。

## 控制一系列的产品（车、武器、食品补给）

现在让这个人，开着车，拿着AK47，吃着苹果。 意思就是，这是一些列的产品，要控制这一些列的产品的生产。

比如你要装修，海尔整体厨房，有微波炉，油烟机，洗衣机，电磁炉。一系列的产品。

|  |
| --- |
| **public** **class** Car{    **public** **void** run() {  System.***out***.println("冒着烟奔跑中...");  }  } |

|  |
| --- |
| **public** **class** AK47 {  **public** **void** shoot(){  System.***out***.print("哒哒哒....");  }  } |

|  |
| --- |
| **public** **class** Apple {  **public** **void** getName(){  System.***out***.println("Apple...");  }  } |

测试类：

|  |
| --- |
| Car car = **new** Car();  car.run();  AK47 ak = **new** AK47();  ak.shoot();  Apple apple = **new** Apple();  apple.getName(); |

打印：

冒着烟奔跑中...

哒哒哒....

Apple...

产生这一系列的产品，需要有一个默认的工厂：

|  |
| --- |
| //默认的工厂  **public** **class** DefaultFactory {  **public** Car createCar(){  **return** **new** Car();  }  **public** AK47 createAK47(){  **return** **new** AK47();  }  **public** Apple createApple(){  **return** **new** Apple();  }  } |

此时的测试程序，只要new出来一个默认工厂，就可以生产这一系列的产品了：

|  |
| --- |
| DefaultFactory factory = **new** DefaultFactory();  Car car = factory.createCar();  car.run();  AK47 ak = factory.createAK47();  ak.shoot();  Apple apple = factory.createApple();  apple.getName(); |

打印：

冒着烟奔跑中...

哒哒哒....

Apple...

当需要吧这一系列产品全换掉的话，把这个工厂换掉就可以了：

这里新建一个工厂，魔法工厂：

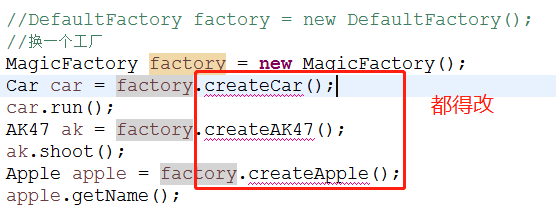
|  |
| --- |
| //哈利波特的魔法工厂  **public** **class** MagicFactory {  //交通工具：扫把  **public** Broom createBroom(){  **return** **new** Broom();  }    //武器：魔法棒  **public** MagicStick createMagicStick(){  **return** **new** MagicStick();  }  //食物：毒蘑菇  **public** MushRoom createMushRoom(){  **return** **new** MushRoom();  }  } |

|  |
| --- |
| //扫帚  **public** **class** Broom{  **public** **void** run() {  System.***out***.println("扫帚摇着尾巴呼呼呼...");  }  } |

|  |
| --- |
| //武器：魔法棒  **public** **class** MagicStick {  } |

|  |
| --- |
| //食物：毒蘑菇  **public** **class** MushRoom {  } |

但是此时，站在客户的角度，想把工厂从DefaultFactory换到MagicFactory：



有了之前的经验，这里自然就联想到，工厂创建的不能是具体的类，要是一个接口/或者是一个抽象类，比如你的Car，工厂创建的不能是Car，要是Car的父类，这样在换工厂的时候，下面的代码才可以不用改动。所以，我们要建一个抽象工厂，然后让DefaultFactory、MagicFactory都去继承/实现这个工厂。而且工厂的返回值，都是抽象类或者接口。

|  |
| --- |
| //抽象工厂  **public** **abstract** **class** AbstractFactory {  //生产 交通工具  **public** **abstract** Vehicle createVehicle();  //生产 武器  **public** **abstract** Weapon createWeapon();  //生产食物  **public** **abstract** Food createFood();  } |

|  |
| --- |
| //交通工具  **public** **abstract** **class** Vehicle {  //实现由子类决定  **public** **abstract** **void** run();  } |

|  |
| --- |
| //食物  **public** **abstract** **class** Food {  **public** **abstract** **void** printName();  } |

|  |
| --- |
| //武器  **public** **abstract** **class** Weapon {  //  **public** **abstract** **void** shoot();  } |

产品类：都继承产品的抽象类

|  |
| --- |
| **public** **class** Car **extends** Vehicle{  @Override  **public** **void** run() {  System.***out***.println("冒着烟奔跑中...");  }  } |
| //扫帚  **public** **class** Broom **extends** Vehicle{  @Override  **public** **void** run() {  System.***out***.println("扫帚摇着尾巴呼呼呼...");  }  } |
| //食物：毒蘑菇  **public** **class** MushRoom **extends** Food {  @Override  **public** **void** printName() {  System.***out***.println("mushroom");  }  } |

|  |
| --- |
| **public** **class** Apple **extends** Food {  @Override  **public** **void** printName() {  System.***out***.println("apple");  }  } |

|  |
| --- |
| **public** **class** AK47 **extends** Weapon{  **public** **void** shoot(){  System.***out***.println("哒哒哒....");  }  } |

|  |
| --- |
| //武器：魔法棒  **public** **class** MagicStick **extends** Weapon {  @Override  **public** **void** shoot() {  System.***out***.println("fire hu hu hu ...");  }  } |

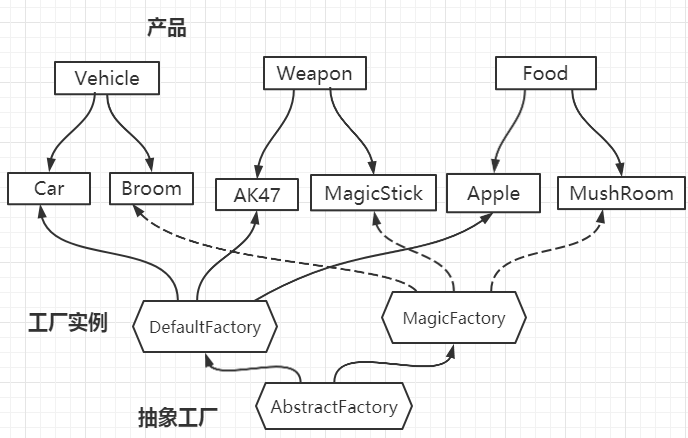
魔法工厂继承抽象工厂：

|  |
| --- |
| //哈利波特的魔法工厂  **public** **class** MagicFactory **extends** AbstractFactory {  //交通工具：扫把  **public** Vehicle createVehicle(){  **return** **new** Broom();  }    //武器：魔法棒  **public** Weapon createWeapon(){  **return** **new** MagicStick();  }  //食物：毒蘑菇  **public** Food createFood(){  **return** **new** MushRoom();  }  } |

DefaultFactory继承抽象工厂：

|  |
| --- |
| //默认的工厂  **public** **class** DefaultFactory **extends** AbstractFactory{  @Override  **public** Food createFood() {  **return** **new** Apple();  }  @Override  **public** Vehicle createVehicle() {  **return** **new** Car();  }  @Override  **public** Weapon createWeapon() {  **return** **new** AK47();  }  } |

最终形成的类结构是这样的：



测试程序：

|  |
| --- |
| //换一个工厂，只需要改动这一处，就可以了，换一个工厂，就把生产的系列产品都换了  AbstractFactory factory = **new** MagicFactory(); //new DefaultFactory();  //换一个工厂  Vehicle vehicle = factory.createVehicle();  vehicle.run();  Weapon weapon = factory.createWeapon();  weapon.shoot();  Food food = factory.createFood();  food.printName(); |

**DefaultFactory打印：**

冒着烟奔跑中...

哒哒哒....

apple

**把工厂换为MagicFactory打印：**

扫帚摇着尾巴呼呼呼...

fire hu hu hu ...

mushroom

抽象工厂：生产一系列的产品，如果你想换掉一系列的产品，或者你想在这一系列的产品上进行扩展，以及想对这一些列产品的生成过程进行控制，用抽象工厂。

### 抽象工厂和普通工厂的优缺点：

**普通工厂：**

可以在产品的维度上进行扩展，可以产生新的产品，可以产生新的产品的工厂。也就是说可以在产品的维度进行扩展。

在普通工厂想要产生产品系列，就会特别麻烦。产生一个产品，就会出现一个产品的Factory。就会出现”工厂泛滥”。

**抽象工厂：**

能换工厂，生产新的产品系列，但是不能产生新的产品品种，新添加一个产品，就要在抽象工厂里面加 createXXX(); 方法，所有子类都要实现这个方法。要改动的地方太多。

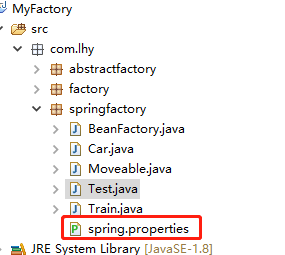
有没有一种工厂，结合普通工厂和抽象工厂的有点呢？

既可以随意添加产品品种，又很方便的添加产品系列？ 没有。

Spring提供了一种方案，Spring的Bean工厂。

Spring说，你就不要这么Moveable m = **new** Car(); 就行new Car了，你给配置到配置文件里。

测试java读取properties配置文件反射生成对象



spring.properties：

VehicleType=com.lhy.springfactory.Car

测试代码：

|  |
| --- |
| **public** **static** **void** main(String[] args) **throws** Exception{  Properties props = **new** Properties();  props.load(Test.**class**.getClassLoader().getResourceAsStream("com/lhy/springfactory/spring.properties"));  String vehicleTypeName = props.getProperty("VehicleType");  System.***out***.println(vehicleTypeName);  //反射生成对象  Object o = Class.*forName*(vehicleTypeName).newInstance();  Moveable m = (Moveable)o;  m.run();  } |

打印结果：

com.lhy.springfactory.Car

冒着烟奔跑中...

把配置文件换成或者Trian:

VehicleType=com.lhy.springfactory.Train

执行测试代码，打印：

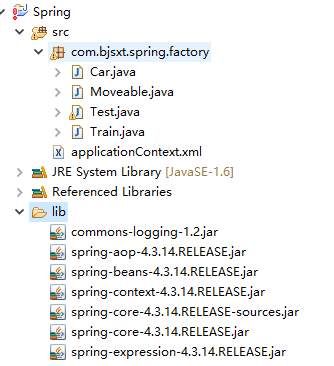
com.lhy.springfactory.Train

小火车呜呜呜...

可以看到，只是改了配置文件，就可以动态控制生成的类了。代码都不用动。Spring就是这样的思路。

最简单的使用Spring：

引入必须的jar包，



Spring要求的配置文件：

|  |
| --- |
| <?xml version=*"1.0"* encoding=*"UTF-8"*?>  <beans xmlns=*"http://www.springframework.org/schema/beans"*  xmlns:xsi=*"http://www.w3.org/2001/XMLSchema-instance"*  xsi:schemaLocation=*"http://www.springframework.org/schema/beans*  *http://www.springframework.org/schema/beans/spring-beans-3.0.xsd"*>  <bean id=*"v"* class=*"com.bjsxt.spring.factory.Train"*>  </bean>    <!-- //v=com.bjsxt.spring.factory.Car -->  </beans> |

测试程序：

|  |
| --- |
| **package** com.bjsxt.spring.factory;  **import** java.io.IOException;  **import** org.springframework.beans.factory.BeanFactory;  **import** org.springframework.context.support.ClassPathXmlApplicationContext;  **public** **class** Test {  /\*\*  \* **@param** args  \* **@throws** IOException  \*/  **public** **static** **void** main(String[] args) **throws** Exception {  BeanFactory f = **new** ClassPathXmlApplicationContext("applicationContext.xml");  Object o = f.getBean("v");  Moveable m = (Moveable)o;  m.run();  }  } |

配置文件配置的火车类，打印：

小火车呜呜呜...

换成Car，打印：

冒着烟奔跑中...

## 下面来模拟Spring的Bean工厂

Spring的BeanFactory ，就是一个容器，是用一个map实现的，就是从配置文件读取 <bean id=*"v"* class=*"com.bjsxt.spring.factory.Train"*/> 这样的配置，遍历解析这样的xml配置，以id为key，class后的类全限定名用反射生成的对象为value，放到这个map中去。当用的时候，直接map.get(id); 获取这个Bean对象。

ClassPathXmlApplicationContext是BeanFactory的一种实现。这里模拟这种实现。

这里模拟Spring的Bean工厂：

|  |
| --- |
| **public** **interface** BeanFactory {  Object getBean(String id);  } |

ClassPathXmlApplicationContext：

|  |
| --- |
| **public** **class** ClassPathXmlApplicationContext **implements** BeanFactory{  //存放一个个Bean对象的容器，  **private** Map<String, Object> container = **new** HashMap<String, Object>();    // 构造方法找到配置文件，读取xml配置文件  **public** ClassPathXmlApplicationContext(String fileName) **throws** Exception{  SAXBuilder sb = **new** SAXBuilder();  Document doc = sb.build(**this**.getClass().getClassLoader()  .getResourceAsStream(fileName));  Element root = doc.getRootElement();  List list = ~~XPath~~.*selectNodes*(root, "/beans/bean");  System.***out***.println(list.size());  **for** (**int** i = 0; i < list.size(); i++) {  Element bean = (Element) list.get(i);  String id = bean.getAttributeValue("id");  String clazz = bean.getAttributeValue("class");  Object o = Class.*forName*(clazz).newInstance();  container.put(id, o);  System.***out***.println(id + " " + clazz);  }  }  //读取配置文件，读取id为传进来的id的Bean，实例化  @Override  **public** Object getBean(String id) {  **return** container.get(id);  }    } |

测试程序：

|  |
| --- |
| **public** **static** **void** main(String[] args) **throws** Exception{    BeanFactory f = **new** ClassPathXmlApplicationContext("com/lhy/springfactory/applicationContext.xml");  Object o = f.getBean("v");  Moveable m = (Moveable)o;  m.run();    Train trian = (Train)f.getBean("trian");  trian.run();    } |

applicationContext.xml

|  |
| --- |
| <?xml version=*"1.0"* encoding=*"UTF-8"*?>  <beans>  <bean id=*"v"* class=*"com.lhy.springfactory.Car"*/>  <bean id=*"trian"* class=*"com.lhy.springfactory.Train"*/>  <!-- //v=com.bjsxt.spring.factory.Car -->  </beans> |

打印：

2

v com.lhy.springfactory.Car

trian com.lhy.springfactory.Train

冒着烟奔跑中...

小火车呜呜呜...

这样，就把类配置在了配置文件里。

# 三 、设计模式的的六大原则及其含义

# 四、常见的单例模式以及各种实现方式的优缺点，哪一种最好，手写常见的单利模式

# 五、设计模式在实际场景中的应用