**Effective Java 阅读笔记**

**第二章 创建和销毁对象**

**第1条： 考虑用静态工厂方法代替构造器**

静态工厂方法与构造器相比有四大优势：

(1)**静态工厂方法有名称，具有适当名称的静态工厂方法易于使用、易于阅读**；

(2)**不必每次在调用它们的时候都创建一个新的对象**；

(3)**可以返回原返回类型的任何子类型的对象**；

(4)**在创建参数化类型实例的时候，它们使代码变得更加简洁**。

同时静态工厂方法也有两大缺点：

(1)**类如果不含公有的或者受保护的构造器，就不能被子类化；**

(2)**它们与其它静态方法实际上没有任何区别**。

静态工厂方法的一些惯用名称如下：

（1）valueOf: 该方法返回的实例与它的参数具有相同的值，实际上是类型转换方法；

（2）of：valueOf的简洁替代，在EnumSet中使用并流行起来；

（3）getInstance： 返回的实例是通过方法的参数来描述的，对于单例模式(Singleton)来说，该方法无参数，并返回唯一的实例；

（4）newInstance：功能同getInstance，但与getInstance不同的是，它能够确保返回的每个实例都与其它实例不同；

（5）getType：像getInstance一样，但是在工厂方法处于不同的类中的时候使用。Type表示工厂方法返回的工厂类型；

（6）newType： 像newInstance一样，但是在工厂方法处于不同的类中的时候使用。Type表示工厂方法返回的对象类型。

因此，在写程序的时候我们可以优先考虑静态工厂方法，然后再考虑构造器。

**第2条： 遇到多个构造器参数时要考虑用构建器**

多参构造器方法：

（1）重叠构造器模式：当有许多参数时，客户端代码会很难编写，并且不利于代码阅读；

（2）JavaBean模式：调用一个无参构造器，然后调用setter方法来设置参数；

（3）Builder模式：既安全，可读性又好。

如果类的构造器或者静态工厂中具有多个参数，设计这种类时，Builder模式就是种不错的选择。

**第3条： 用私有构造器或者枚举类型强化Singleton属性**

实现Singleton的三种方法：

|  |
| --- |
| 1. 公有静态成员是个final域：   //Singleton with public final field  public class Elvis {  public static final Evis INSTANCE = new Elvis();  private Elvis() {…}  public void leaveTheBuilding() {…}  }   1. 公有成员是个静态工厂方法：   // Singleton with static factory  public class Elvis {  private static final Elvis INSTANCE = new Elvis();  private Elvis() {…}  public static Elvis getInstance() { return INSTACNE; }  public void leaveTheBuilding() { … }  }   1. 包含单个元素的枚举类型：   // Enum singleton – the preferred approach  public enum Elvis {  INSTANCE;  public void leaveTheBuilding() { … } |

**第4条： 通过私有构造器强化不可实例化的能力**

当我们在写一个工具类的时候，我们为这个工具类定义了很多静态方法，通过“类名.方法名”这种方式来使用这个类的功能，所以我们不希望这个类能被实例化，但是如果我们不定义构造器，Java编译器会自动的提供一个public无参的缺省构造器给这个类，所以这时创建一个私有构造器，来强化其不可实例化的能力，示例代码如下：

|  |
| --- |
| // Noninstantiable utility class  public class UtilityClass {  //Suppress default constructor for noninstantiability  private UtilityClass() {  throw new AssertionError();  }  …// Remainder omitted  } |

由于构造器是private，所以在类的外部不能访问它，AssertionError()可以避免不小心在类的内部调用构造器，它保证该类在任何情况下都不会被实例化。

**第5条： 避免创建不必要的对象**

(1). 若一个方法频繁调用且每次生成相同的内部实例，可以作为static，如Map的keyset；

(2). 维护自己的对象池来避免重复创建对象不是很好的做法，除非对象是非常重量级的，Object pool也增加了内存占用

**第6条： 消除过期的对象引用**

可能发生内存泄漏的几种方式：

1. 类自己申请内存管理；
2. 缓存，易忘记管理；

（3） 监听器和其他回调，回调此时可以做成弱引用。

**第7条： 避免使用终结方法**

终结方法(finally)通常是不可预测的，是很危险的，一般情况下是不必要的。使用终结方法会导致行为不稳定、降低性能，以及可移植性问题。除非作为安全网，或者是为了终止非关键的本地资源，否则请不要使用终结方法。

上周作业

有三个线程A、B、C（线程名称或id），循环打印10次ABCABC…

|  |
| --- |
| **import** java.util.concurrent.atomic.AtomicInteger;    **public** **class** Print **extends** Thread{  **private** AtomicInteger synObj;  **private** **int** count;  **private** String s;  **private** **int** flag;  **private** **int** total = 0;    **public** Print(**int** count,AtomicInteger atomicInteger,String s,**int** flag) {  **this**.count = count;  **this**.synObj = atomicInteger;  **this**.s = s;  **this**.flag = flag;  }  **public** **void** run() {  **while**(**true**) {  **synchronized**(synObj) {  **if**(synObj.intValue()%3 == flag) {  total++;  synObj.set(synObj.intValue()+1);  System.***out***.print(s);  synObj.notifyAll();  **if**(total == count) {  **break**;  }  }**else** {  **try**{  synObj.wait();  }**catch**(Exception e){  e.printStackTrace();  }  }  }  }  }  **public** **static** **void** main(String[]args) **throws** Exception {  AtomicInteger synObj = **new** AtomicInteger(0);  Print a = **new** Print(10,synObj,"A",0);  Print b = **new** Print(10,synObj,"B",1);  Print c = **new** Print(10,synObj,"C",2);  a.start();  b.start();  c.start();  }  } |