**Effective Java通俗理解（上）**

　　这篇博客是Java经典书籍《Effective Java（第二版）》的读书笔记，此书共有78条关于编写高质量Java代码的建议，我会试着逐一对其进行更为通俗易懂地讲解，故此篇博客的更新大约会持续1个月左右。

**第1条：考虑用静态工厂方法代替构造器**

　　通常情况下我们会利用类的构造器对其进行实例化，这似乎毫无疑问。但“静态工厂方法”也需要引起我们的高度注意。  
　　什么是“静态工厂方法”？这不同于设计模式中的工厂方法，我们可以理解它为“在一个类中用一个静态方法来返回这个类的实例”，例如：

public static People getInstance() {

return new People();

}

　　它是一个“方法”，那么它不同于构造器，它可以随意修改方法名，这就带来**第一个优点——有名称**。有时一个类的构造器往往不止一个，而它们的名称都是相同的，不同的仅仅是参数，如果不同参数带来不同的含义这样对于调用方来说除了注释很难理解它们有什么不同的含义。例如BigInteger(int, int, Random)返回一个素数，但调用者很难理解API设计者所要想表达的意思，如果此时有BigInteger.probablePrime静态工厂方法，则能一目了然的清楚API设计者所要想表达的含义。举一个JDK的例子：Executors类，在这个类中有newFixedThread、newSingleThreadExecutor、newCachedThreadPool等静态方法，因为它们有“名字”，所有就较为清晰的明白API的含义。

　　《Effective Java》中所提到的静态工厂方法**第二个优点在于不用重复创建一个对象**，实际上也就是勤加载或者称为饿汉式的单例模式。例如：

[复制代码](javascript:void(0);)

public class Instance() {

private static Instance instance = new Instance();

private Instance(){}

public static Instance getInstance() {

return instance;

}

}

[复制代码](javascript:void(0);)

**静态工厂方法的第三个优点，可以返回原返回类型的任何子类型的**。这句话初看不好理解，举个JDK中的例子：Collections类。

List list = Collections.synchronizedList(new ArrayList())

　　这个例子就说明了可以返回原返回类型的任何子类型的对象。  
　　关于静态工厂方法的第四个优点，在创建参数化类型实例的时候，它们使代码变得更加简洁，书中举了一个例子：

Map<String, List<String>> m = new HashMap<String, List<String>>(); //这会显得很繁琐

　　给集合类提供静态工厂方法后：

public static <K, V> HashMap<K, V> newInstance() {

return new HashMap<K, V>();

}  
https://images2017.cnblogs.com/blog/630246/201708/630246-20170803230227178-1874461714.png

 　　但是实际上从JDK7（包括JDK7）之后的集合类可以用以下简洁的代码代替：

Map<String, List<String>> m = new HashMap<>();

　　静态工厂方法也有两个缺点：一是公有的静态方法所返回的非公有类不能被实例化，也就是说Collections.synchronizedList返回的SynchronizedList不能被实例化；二是查找API比较麻烦，它们不像普通的类有构造器在API中标识出来，而是和其他普通静态方法一样，鉴于此，书中提到了几种惯用名称：

　　valueOf  
　　of　　  
　　**getInstance**  
**newInstance**  
　　getType  
　　newType

**第2条：遇到多个构造器参数时要考虑用构建器**

　　你是否写过下面类似的代码：

[复制代码](javascript:void(0);)

public void Student() {

/\*必填\*/

private String name;

private int age;

/\*选填\*/

private String sex;

private String grade;

public Student(String name, String sex) {

this(name, sex, 0);

}

public Student(String name, String sex, int age) {

this(name, sex, age, “”);

}

public Student(String name, String sex, int age, String grade) {

this.name = name;

this.sex = sex;

this.age = age;

this.grade = grade;

}

}

[复制代码](javascript:void(0);)

　　当我想实例化一个名字叫“kevin”，性别男，但是不写年龄，只有年级“1年级”，这个时候代码就：不得不要为年龄这个参数传递值。如果新增一个只含年级的构造方法，那又将多出许多代码，更严重的是，如果没有一份详尽的文档或者注释，看到如此多的构造方法将无从下手，这就是非常常见的重叠构造器。

Student student = new Student(“Kevin”, “男”, “0”, “1年级”);

　　当然还有另外一种方法，只有一个必填项的构造方法，而其他选填项利用setter方法传递。例如：

Student student = new Student(“kevin”, “男”);

student.setGrade(“1年级”);

　　这实际上导致了在构造过程中JavaBean可能处于不一致的状态，也就是说实例化对象本该是一气呵成，但现在却分割成了两大步，这会导致它线程不安全，进一步引发不可预知的后果。  
　　书中提到较为“完美”的解决方案就是利用“Builder模式（建造者模式）”，有关此设计模式可以查看《建造者模式》。这种解决方案属建造者模式的一种形式，其核心就是**不直接生成想要的对象，而是让客户端利用所有必要的参数调用构造器（或者静态工厂），得到一个builder对象，再调用类似setter的方法设置相关可选参数**。构建器模式如下所示：

[复制代码](javascript:void(0);)

/\*\*

\* 构建器模式

\* Created by yulinfeng on 2017/8/3.

\*/

public class Student {

/\*必填\*/

private String name;

private int age;

/\*选填\*/

private String sex;

private String grade;

public static class Builder {

private String name;

private int age;

private String sex = "";

private String grade = "";

public Builder(String name, int age) {

this.name = name;

this.age = age;

}

public Builder sex(String sex) {

this.sex = sex;

return this;

}

public Builder grade(String grade) {

this.grade = grade;

return this;

}

public Student build() {

return new Student(this);

}

}

private Student(Builder builder) {

this.name = builder.name;

this.age = builder.age;

this.sex = builder.sex;

this.grade = builder.grade;

}

}

[复制代码](javascript:void(0);)

　　客户端代码：

Student student = new Student.Builder("kevin", 23).grade("1年级").build();

　　这样的客户端代码很容易边写，并且便于阅读。对于不了解的可能来说利用构建器模式编写Student类不算易事，甚至代码比重叠构造器的代码更多。所以当可选参数在很多时，谨慎使用重叠构造器，而是使用构建器模式。

*2017-08-03*

**第3条：用私有构造器或者枚举类型强化Singleton属性**

　　书中的此条目下，我总结出来认为最有价值的是“**能被序列化的线程安全的类，被序列化后是否还会是单例**”。单例模式几乎人人会写，例如上面提到的勤加载（饿汉式）的单例模式：

[复制代码](javascript:void(0);)

public class Instance {

private static final Instance obj = new Instance();

private Instance() {

}

public static Instance getInstance() {

return obj;

}

}

[复制代码](javascript:void(0);)

　　毫无疑问这个单例是线程安全的，如果我们希望一个类能被序列化，那么这个类直接实现Serializable就可以了，但是这样能否保证被反序列化过后还是单例呢？

[复制代码](javascript:void(0);)

1 import java.io.Serializable;

2

3 /\*\*

4 \* 序列化单例对象

5 \* Created by 余林丰 on 2017/8/4/0004.

6 \*/

7 public class Instance implements Serializable {

8 private static final Instance obj = new Instance();

9 private Instance() {

10

11 }

12 public static Instance getInstance() {

13 return obj;

14 }

15 }

[复制代码](javascript:void(0);)

　　测试代码：

[复制代码](javascript:void(0);)

1 import java.io.FileInputStream;

2 import java.io.FileOutputStream;

3 import java.io.ObjectInputStream;

4 import java.io.ObjectOutputStream;

5

6 /\*\*

7 \* 序列化与反序列化单例对象

8 \* Created by 余林丰 on 2017/8/4/0004.

9 \*/

10 public class Main {

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

12 ObjectOutputStream out = new ObjectOutputStream(new FileOutputStream("D:\\objFile.obj"));

13 Instance instance = Instance.getInstance();

14 out.writeObject(instance);

15 out.close();

16

17 ObjectInputStream in = new ObjectInputStream(new FileInputStream("D:\\objFile.obj"));

18 Instance instance1 = (Instance) in.readObject();

19 in = new ObjectInputStream(new FileInputStream("D:\\objFile.obj"));

20 Instance instance2 = (Instance) in.readObject();

21 System.out.println("obj1 hashcode:" + instance1.hashCode());

22 System.out.println("obj2 hashcode:" + instance2.hashCode());

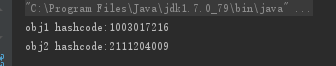
23 in.close();

24 }

25 }

[复制代码](javascript:void(0);)

　　比较两个实例对象的hash值，可以看到执行结果为：



　　这就说明被反序列化过后便不再是单例。要保证单例还必须在单例类中实现readResolve的方法：

[复制代码](javascript:void(0);)

1 import java.io.Serializable;

2

3 /\*\*

4 \* 序列化单例对象

5 \* Created by 余林丰 on 2017/8/4/0004.

6 \*/

7 public class Instance implements Serializable {

8 private static final Instance obj = new Instance();

9 private Instance() {

10

11 }

12 public static Instance getInstance() {

13 return obj;

14 }

15 private Object readResolve(){

16 return obj;

17 }

18 }

[复制代码](javascript:void(0);)

　　如此一来的执行结果为：

https://images2017.cnblogs.com/blog/630246/201708/630246-20170804223018553-2020776419.png

　　显然此时被序列化过后还是单例，至于为什么要实现readResolve，这个方法并不是Serializable接口的方法，在此我并没有深究，究其原因估计是在反序列化的时候会调用这个奇怪的方法。

　　另外书中还提到了另外高档的一种单例模式，此方法既能保证线程安全，也能保证被反序列化后还是单例。

[复制代码](javascript:void(0);)

1 /\*\*

2 \* 枚举类型单例

3 \* Created by 余林丰 on 2017/8/4/0004.

4 \*/

5 public enum Instance {

6 INSTANCE //同样可以像普通类一样定义普通的方法变量等

7 }

[复制代码](javascript:void(0);)

　　将测试代码第13行稍作修改：

[复制代码](javascript:void(0);)

1 import java.io.FileInputStream;

2 import java.io.FileOutputStream;

3 import java.io.ObjectInputStream;

4 import java.io.ObjectOutputStream;

5

6 /\*\*

7 \* 序列化与反序列化单例对象

8 \* Created by 余林丰 on 2017/8/4/0004.

9 \*/

10 public class Main {

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

12 ObjectOutputStream out = new ObjectOutputStream(new FileOutputStream("D:\\objFile.obj"));

13 Instance instance = Instance.INSTANCE;

14 out.writeObject(instance);

15 out.close();

16

17 ObjectInputStream in = new ObjectInputStream(new FileInputStream("D:\\objFile.obj"));

18 Instance instance1 = (Instance) in.readObject();

19 in = new ObjectInputStream(new FileInputStream("D:\\objFile.obj"));

20 Instance instance2 = (Instance) in.readObject();

21 System.out.println("obj1 hashcode:" + instance1.hashCode());

22 System.out.println("obj2 hashcode:" + instance2.hashCode());

23 in.close();

24 }

25 }

[复制代码](javascript:void(0);)

　　运行结果为：

https://images2017.cnblogs.com/blog/630246/201708/630246-20170804223224303-867610534.png

　　可以看到它自带序列化机制，且防止了多次实例化，非常高档的一个单例实现。

**第4条：通过私有构造器强化不可实例的能力**

　　在此条书中建议有的工具类例如Arrays等，对它们进行实例化并没有意义，所以应该在它们的构造方法上应该使用private修饰。

*2017-08-04*

**第5条：避免创建不必要的对象**

　　此条目下书中提到“**当你应该重用现有对象的时候，请不要创建新的对象**”。

　　最为突出的例子莫过于字符串常量的创建，众所周知String字符串有两种创建方式

String str = “hello”;

String str = new String(“hello”);

　　第一种String字符串的创建是在方法区（JDK7后改到了堆内存）中的常量池中创建一个”hello”常量，将来若再有一个字符串变量为“hello”时将直接指向常量池中的“hello”变量而不用重新创建；第二种则是会在堆变量中创建一个新的String实例，将来若再以此方式创建一个字符串变量为“hello”时还是会重新创建一个String实例。显然第二种方式创建了一个“不必要”的实例，相比较而言第一种方式更加高效。

　　另外一个例子则是将”true”变量转换为Boolean类型也有以下两种转换方式：

Boolean b = new Boolean(“true”);

Boolean b = Boolean.valueOf(“true”);

　　第一种转换方式也是每次都会在堆内存中创建一个新的Boolean实例；第二种查看其源代码便知不会每次返回一个新的实例，返回的是一个在编译器就已经确定了的static final Boolean型变量：

[复制代码](javascript:void(0);)

public static final Boolean TRUE = new Boolean(true);

public static final Boolean FALSE = new Boolean(false);

public static Boolean valueOf(String s) {

return toBoolean(s) ? TRUE : FALSE;

}

private static boolean toBoolean(String name) {

return ((name != null) && name.equalsIgnoreCase("true"));

}

[复制代码](javascript:void(0);)

　　书中举了一个例子是否是在1946年至1964年出生来说明，这个例子在现实当中也很常见：

[复制代码](javascript:void(0);)

1 import java.util.Calendar;

2 import java.util.Date;

3 import java.util.TimeZone;

4

5 /\*\*

6 \* “是否在1946-1965年出生”

7 \* 这在现实中应该是比较常见的一种写法

8 \* Created by yulinfeng on 8/5/17.

9 \*/

10 public class Person {

11 private final Date birthDate;

12

13 public Person(Date birthDate) {

14 this.birthDate = birthDate;

15 }

16

17 public boolean isBabyBoomer() {

18 Calendar gmtCal = Calendar.getInstance(TimeZone.getTimeZone("GMT")); //新创建Calendar实例

19 gmtCal.set(1949, Calendar.JANUARY, 1, 0, 0, 0);

20 Date boomStart = gmtCal.getTime(); //新创建Date实例

21 gmtCal.set(1965, Calendar.JANUARY, 1, 0, 0, 0);

22 Date boomEnd = gmtCal.getTime(); //新创建Date实例

23 return birthDate.compareTo(boomStart) >= 0 && birthDate.compareTo(boomEnd) < 0;

24 }

25 }

[复制代码](javascript:void(0);)

　　这样的代码我相信人人都写过类似的，书中提到这种代码的写法每次都创建新的实例对象，而实际上是不必要的，而给出了一种比较高效的代码示例：

[复制代码](javascript:void(0);)

1 import java.util.Calendar;

2 import java.util.Date;

3 import java.util.TimeZone;

4

5 /\*\*

6 \* “是否在1946-1965年出生”

7 \* 这在现实中应该是比较常见的一种写法

8 \* Created by yulinfeng on 8/5/17.

9 \*/

10 public class Person {

11 private final Date birthDate;

12 private static final Date BOOM\_START;

13 private static final Date BOOM\_END;

14 static {

15 Calendar gmtCal = Calendar.getInstance(TimeZone.getTimeZone("GMT")); //新创建Calendar实例

16 gmtCal.set(1949, Calendar.JANUARY, 1, 0, 0, 0);

17 BOOM\_START = gmtCal.getTime(); //新创建Date实例

18 gmtCal.set(1965, Calendar.JANUARY, 1, 0, 0, 0);

19 BOOM\_END = gmtCal.getTime(); //新创建Date实例

20 }

21 public Person(Date birthDate) {

22 this.birthDate = birthDate;

23 }

24

25 public boolean isBabyBoomer() {

26 return birthDate.compareTo(BOOM\_START) >= 0 && birthDate.compareTo(BOOM\_END) < 0;

27 }

28 }

[复制代码](javascript:void(0);)

　　利用静态代码块在类Person初始化的时候创建对象，而在之后调用方法判断时不再每次重新创建新的实例对象，这种写法有点“烧脑”，确实有点“不符合”编码习惯，大概是因为这是需要思考才能写出来的原因吧。

**第6条：消除过期的对象引用**

　　此条目较为容易理解，之所以要消除过期的对象引用其目的就在于尽量避免内存泄露的问题，书中举了一个“栈”的例子，其中在弹出一个元素时代码如下：

[复制代码](javascript:void(0);)

public Object pop() {

if (size == 0) {

throw new EmptyStackException();

}

return elements[--size];

}

[复制代码](javascript:void(0);)

　　可以看到弹出元素时仅仅是将元素弹出后在将数组的索引-1，实际上数组维护的那个元素引用还在，也就是说那个被弹出的元素并不会被GC，如此一来最后就很有可能导致内存泄露的问题。解决此类的办法就是，当不再使用这个元素时，将其置为null。

[复制代码](javascript:void(0);)

public Object pop() {

if (size == 0) {

throw new EmptyStackException();

}

Object result = elements[--size];

elements[size] = null;

return result;

}

[复制代码](javascript:void(0);)

　　实际上当你写出上面的代码时一定注意这并不是在任意条件下都成立，你应该仔细思考此时的引用是否为过期引用，“**清空对象引用应该是一种例外，而不是一种规范行为**”。

**第7条：避免使用终结方法**

　　此处所谓的终结方法指的就是finalize()方法，这个方法可能对于从C++转向Java的新手感到混淆，因为在C++中有一个“析构函数”，析构函数所代表的意义就是在这个对象垃圾回收前所做的一些动作例如资源的关闭等。对于Java来说垃圾回收是自动的，或者称之为不可预知或不可控，尽管finalize方法所代表的也是在垃圾回收前所做的一些动作，但对于GC的时间你不能掌握，也就是说不能保证finalize方法会被及时执行，这是很危险的，一般情况下这个方法不会被用到。

　　终结方法既然存在那它就并不是毫无用处，第一种用途就是充当一个“安全网”，终结方法“本该”是在GC前做一些清理动作，但GC的时间未知，也就是终结方法执行时间未知，对于FileInputStream类我们都知道应该在try-finally中对其调用close方法，但也许我们会忘记编写此方法，在FileInputStream源代码中就实现了终结方法目的就在于如果忘记了close方法，至少还有终结方法，虽然可能不能得到及时执行，但晚执行总比不执行好吧。第二个用途可能使用的场景就可能比较少，JVM只回收普通对象，对于本地对象（也就是不是Java实现的对象），JVM并不会对它进行回收，此时我们就可以在终结方法中对本地对象进行一些清理操作，但一定记住一定要是“不拥有关键资源的前提”，且在子类中重写了终结方法一定要现实调用super.finalize()，否则父类的终结方法不会被调用。

　　综上，对于终结方法，一般代码中并不会使用，如果要使用一定要考虑上面两种用途是否值得去做，万万**不应该依赖终结方法来更新重要的持久状态**。

*2017-08-06*

**第8条：覆盖equals时请遵守通用约定**

　　对于equals方法，在编码中最常用的可能就是比较两个字符串是否是值相等的。需要自己重写equals方法的场景可能并不是人人都能有幸碰到，而如果碰到了该怎么办，本条目下书中说明了重写equals方法时需要遵守的一些通用约定，如果不遵守这些约定可能导致无法和其他类配合使用。

　　equals方法来自于Object类：

public boolean equals(Object obj) {

return (this == obj);

}

　　可以看到在Object类中equals比较的两个实例是否是引用相等的，换句话说，在不考虑“值相等”的情况下，每个实例都是独一无二的，每个实例都只与它自身相等。何时需要重写equals方法呢？顶级类只提供了引用是否相等，如果你需要自己实现一个逻辑是否相等，此时则需要重写equals方法，例如String类，但当在重写equals方法时，应该遵守以下约定：

**自反性：对于任何非null的引用值x，x.equals(x)必须返回true。**也就是说一个类的实例一定是与它本身相等的，不管你怎么实现它的逻辑判断，但它的“本”不能忘。

**对称性：对于任何非null的引用值x和y，当且仅当y.equals(x)返回true时，x.equals(y)必须返回true。**这条比较好理解，x=1,y=1,你不能y=x而x!=y吧。

**传递性：对于任何非null的引用值x、y和z，如果x.equals(y)返回true，并且y.equals(z)也返回true，那么x.equals(z)也必须返回true。**显而易见。

**一致性：对于任何非null的引用值x和y，只要equals的比较操作在对象中的所用的信息没有被修改，多次调用x.equals(y)就会一致地返回true，或者一致地返回false。**这条显然，你不能多调用判断几次它的结果就产生变化了吧。

**对于任何非null的引用值x，x.equals(null)必须返回null。**

　　一定要反复检查测试自己重写的equals方法是否遵守以上约定，否则程序可能会变得不正常，因为许多类，包括所有的集合类都依赖于是否遵守了equals约定。书中举了详细的例子来说明上述约定，这里不再叙述。

　　我们来分析下String类中重写的equals方法：

[复制代码](javascript:void(0);)

//String.equals

public boolean equals(Object anObject) {

if (this == anObject) { //是否等于自身

return true;

}

if (anObject instanceof String) { //类型是否相等

String anotherString = (String) anObject; //转换类型

int n = value.length;

if (n == anotherString.value.length) { //先判断长度是否相等

char v1[] = value;

char v2[] = anotherString.value;

int i = 0;

while (n-- != 0) { //一个一个字符判断值是否相等

if (v1[i] != v2[i])

return false;

i++;

}

return true;

}

}

return false;

}

[复制代码](javascript:void(0);)

　　String中equals方法的实现实际上就是书中给我们的重写equals的一些诀窍：

　　1、使用==操作检查“对象是否为这个对象的引用”，这不是必须的，只是作为一种性能优化，例如Integer类中并无此项判断。

　　2、使用instanceof操作符检查“参数是否为正确的类型”。

　　3、把参数转换成正确的类型。

　　4、对于该类中的每个“关键”域，检查参数中的域是否与该对象中对应的域想匹配。

**第9条：覆盖equals时总要覆盖hashCode**

　　如果这个类仅仅是重写了equals方法而没有重写hashCode，那么这个类和基于散列的集合类一起工作时就会出现问题。

　　首先明确一个概念，**两个对象使用equals返回true，则它们的hashCode也一定相等；如果两个对象的hashCode相等，则它们的equals则不一定相等。**这个概念和散列函数相关，在《哈希》这篇博客里我曾谈到过有关散列（哈希）相关的知识。

　　如何实现hashCode，当然你可以使hashCode返回一个固定的数值，任何对象的hashCode都是一个固定的数值，这没有问题。但当它与基于散列的集合类一起工作时，这些元素将具有相同的散列码，进而使得所有对象都被映射到统一散列桶中，使得散列表退化为链表。散列函数应该如何编写在《哈希》 一文中有提到常用的散列算法，这里不再叙述。

**第10条：始终要覆盖toString**

　　这条建议我在实际当中遇到过，因为当时几乎并没有人去重写toString方法，使得我不得不在后来去将几乎所有的POJO类的toString方法都重写了。原因在于在有的场景下会打印一条日志，日志的内容就是POJO类的属性字段值，这个时候toString的意义很明显的就体现出来了，好在eclipse能按照一定的格式自动生成toString方法。有的类是自己已经重新实现了toString方法例如集合类。

**第11条：谨慎地覆盖clone**

　　按照书中的话来讲，**能不重写clone就不要去重写**，因为它带来的问题太多了。我们暂且不讨论这里面的陷阱有多少，只从对Java基础知识的掌握程度来说明什么是clone，以及什么是“深拷贝”和“浅拷贝”。

　　首先观察以下代码，并思考对象在内存中的分配以及引用的变化：

[复制代码](javascript:void(0);)

1 /\*\*

2 \* Created by 余林丰 on 2017/8/7.

3 \*/

4 public class Student {

5 private String name;

6 private int age;

7

8 public Student(String name, int age) {

9 this.name = name;

10 this.age = age;

11 }

12

13 public String getName() {

14 return name;

15 }

16

17 public void setName(String name) {

18 this.name = name;

19 }

20

21 public int getAge() {

22 return age;

23 }

24

25 public void setAge(int age) {

26 this.age = age;

27 }

28 }

[复制代码](javascript:void(0);)

[复制代码](javascript:void(0);)

1 /\*\*

2 \* Created by 余林丰 on 2017/8/7/0004.

3 \*/

4 public class Main {

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

6 Student stu = new Student("kevin", 23);

7 Student stu2 = stu;

8 stu2.setAge(0);

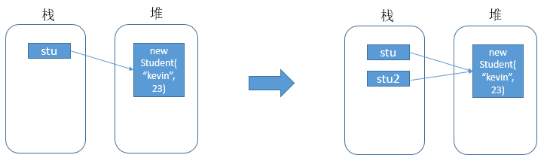
9 System.out.println(stu.getAge());

10 }

11 }

[复制代码](javascript:void(0);)

　　这是一段很简单的代码，Student对象实例stu、stu2在内存中的分配及引用分别如下图所示：



　　所以代码中出现修改stu2实例的age字段时，stu中的age字段也被修改了，原因很简单因为它们的引用指向的都是同一个对象实例。

　　那如果我们想在实例化一个name=”kevin”，age=23的Student实例怎么办呢？当然可以再写一段Student stu2 = new Student(“kevin”, 23);如果再重新构造一个对象实例很复杂，能不能直接复制呢？显然，使Student实现Cloneable接口并重写clone方法即可，注意此时的重写clone方法在里面仅有一句代码即是即调用父类的clone方法，而不是自定义实现：

[复制代码](javascript:void(0);)

1 /\*\*

2 \* Created by 余林丰 on 2017/8/7/0007.

3 \*/

4 public class Student implements Cloneable{

5 private String name;

6 private int age;

7

8 public Student(String name, int age) {

9 this.name = name;

10 this.age = age;

11 }

12

13 public String getName() {

14 return name;

15 }

16

17 public void setName(String name) {

18 this.name = name;

19 }

20

21 public int getAge() {

22 return age;

23 }

24

25 public void setAge(int age) {

26 this.age = age;

27 }

28

29 @Override

30 protected Student clone() throws CloneNotSupportedException {

31 return (Student)super.clone();

32 }

33 }

[复制代码](javascript:void(0);)

[复制代码](javascript:void(0);)

1 /\*\*

2 \*

3 \* Created by 余林丰 on 2017/8/7/0004.

4 \*/

5 public class Main {

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

7 Student stu = new Student("kevin", 23);

8 Student stu2 = stu.clone();

9 stu2.setAge(0);

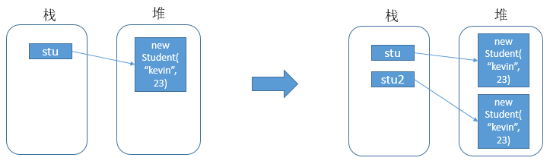
10 System.out.println(stu.getAge());

11 }

12 }

[复制代码](javascript:void(0);)

　　调用clone方法产生的对象实例并不是之前的实例，而是在堆上重新实例化了一个各个参数类型值都相同的实例，所以此时修改stu2的age字段并不会影响到stu，看起来clone就是一个构造器的作用——创建实例。



　　上面我们仅仅是说明了什么是clone，接下来我们接着来讲解什么是“深拷贝”和“浅拷贝”。

　　在上面的例子Student类中，我们新增一个引用型变量Test类：

[复制代码](javascript:void(0);)

1 public class Student implements Cloneable{

2 private String name;

3 private int age;

4 private Test test;

5 public Student(String name, int age) {

6 this.name = name;

7 this.age = age;

8 }

9

10 public String getName() {

11 return name;

12 }

13

14 public void setName(String name) {

15 this.name = name;

16 }

17

18 public int getAge() {

19 return age;

20 }

21

22 public void setAge(int age) {

23 this.age = age;

24 }

25 public String getTest() {

26 return test;

27 }

28

29 public void setTest(Test test) {

30 this.test= test;

31 }

32 @Override

33 protected Student clone() throws CloneNotSupportedException {

34 return (Student)super.clone();

35 }

36 }

[复制代码](javascript:void(0);)

[复制代码](javascript:void(0);)

1 \*\*

2 \*

3 \* Created by 余林丰 on 2017/8/7/0004.

4 \*/

5 public class Main {

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

7 Student stu = new Student("kevin", 23);

8 Student stu2 = stu.clone();

9 stu2.setAge(0);

10 System.out.println(stu.getAge());

11 }

12 }

[复制代码](javascript:void(0);)

　　实际上测试这段代码可知，clone出来的stu2确实和stu是两个对象实例，但它们的成员变量实际上确是指向的同一个引用（通过比较hashCode可知），这也就是所谓的“浅拷贝”。对应的“深拷贝”则是所有的成员变量都会真正的做一份拷贝。怎么做到“深拷贝”，则是要求将类中的所有引用型变量都要clone。

[复制代码](javascript:void(0);)

1 /\*\*

2 \* 深拷贝

3 \* Created by yulinfeng on 8/7/17.

4 \*/

5 public class Student implements Cloneable{

6 private String name;

7 private int age;

8 private Test test;

9

10 public Student(String name, int age) {

11 this.name = name;

12 this.age = age;

13 }

14

15 public String getName() {

16 return name;

17 }

18

19 public void setName(String name) {

20 this.name = name;

21 }

22

23 public int getAge() {

24 return age;

25 }

26

27 public void setAge(int age) {

28 this.age = age;

29 }

30

31 public Test getTest() {

32 return test;

33 }

34

35 public void setTest(Test test) {

36 this.test = test;

37 }

38

39 @Override

40 protected Object clone() throws CloneNotSupportedException {

41 Student stu = (Student)super.clone();

42 stu.test = test.clone(); //Test类也要继承Cloneable

43 return stu;

44 }

45 }

[复制代码](javascript:void(0);)

　　书中是不建议自定义重写clone方法的，如果非要重写书中总结为一句话：**clone方法就是一个构造器，你必须确保它不会伤害到原始的对象，并确保正确地创建被克隆对象中的约束条件。**

　　再说一个与本条目无关的点，查看Cloneable接口实际上可以发现里面什么方法都没有，clone方法却来自Object类，继承了Cloneable接口为什么就能重写clone方法了呢？原因在于clone方法在Object类中的修饰符是protected，而Cloneable接口和Object处于同一个包下，熟悉修饰符的都知道protected的权限限定在同一个包下或者其子类。Cloneable和Object同属于一个包，Cloneable自然能继承clone方法，继承了Cloneable接口的成为了它的子类同样也就继承了clone方法。

2017-08-07

**第12条：考虑实现Comparable接口**

　　关于Comparable接口其中只有一个方法——compareTo。此方法和equals有类似之处，不过它所表达的含义相比equals要更多。equals通常是比较两个值是否相等，相等返回true，不相等返回false。compareTo则约定为第1对象若“大于”第2个对象则返回整数，“等于”则返回0，“小于”则返回负数，compareTo能约定更为复杂的“比较”，例如比较两个字符串进行字典序的比较，str = “abc”, str2 = “abd”，str.equals(str2)返回false，而str.compareTo(str2)则返回正数。compareTo与equals一样同样需要遵守自反性、对称性、传递性。同样有一个强烈的建议就是compareTo应该返回和equals方法相同的结果，但如果不一致，也不是不可以，就是最好能在注释中写明两个方法返回的结果不同。

**第13条：使类和成员的可访问性最小化**

　　如何正确地使用访问权限？首先在定义一个成员变量时类型前会有一个修饰符public 、protected、private（或者没有）。在未学习到“面向对象”时，初学时老师为了讲解方便直接将变量定义为了public，慢慢接触到了面向对象的三大特性：继承、封装、多态，我们也学会了将成员变量的访问权限定义为private。

　　此条实际上就是讲解面向对象的三大特性之一——封装。

**private——只有在生命该成员的类才能访问，其他类都不能访问。**

**default（默认访问修饰符）——又称为“包级私有”，也就是说只有在同一个包下的类才能访问，就算是它的子类但不是在同一包下也不能访问。**

**protected——有两种情况可以访问：1、和default一样同一个包下的类能够访问。2、它的子类也能访问。**

**public——任何类都能访问。**

　　为什么书中提到提到要将可访问性最小化呢？实际上原因在于可维护，一旦你将可访问性置为protected或者public，意味着很大范围的类都能对它就行访问、修改、引用等等，如果你修改了这个变量意味着你要同时修改很多其他的类。但如果可访问性很小private的话，意味着你修改了这个变量，其他类并不知道啊，只对你自己有关系改好你自身就可以了不用担心还有其他哪个地方有用到。

　　对于成员变量通常使用private类，如果完全不提供访问的渠道或者途径似乎也是“死”的，这是setter/getter方法，有人在初学时可能会发出疑问，既然提供了能访问修改这个成员变量的方法，那何不直接置为public呢？原因在于方法是可以提供检查的，它能检查传入的值是否符合规范。

　　另外还有一条规则限制了降低方法的可访问性的能力，那就是**如果方法覆盖了超类中的一个方法，子类中的访问级别就不允许低于超类中的访问级别。**就是说如果父类的方法是public，那么子类就不能是private；如果父类的方法是protected，那么子类就只能是protected或者public。

**第14条：在公有类中使用访问方法而非公有域**

　　这一条实际上我们在上一条已经说到对于类的成员变量，我们通常对它的访问控制置为private，并为它提供getter/setter方法。

2017-08-08

**第15条：使可变性最小化**

　　注意第13条指的是“可访问性”主要讲访问修饰符，而这一条指的是“可变性”讲的是final关键字。

　　我曾在这篇[《“不可变的对象”与“不可变的对象引用”》](http://www.cnblogs.com/yulinfeng/p/5983190.html)提到过final关键字，在此我们重新回顾final关键字。对于final关键字，我想很多人对于它的用法基本上是处于以下用法：

public static final String NAME = “hello world”;

　　把它和static关键字配合使用使得程序中不会出现硬编码，而是以常量的形式出现，这样也便于修改和阅读，但是final的功效远远不止于此。

　　首先来了解下什么是不可变的对象，最为常见的就是String类，这个类大家都清楚是不可变的，至于为什么在赋值过后还能对它进行赋值那是因为在常量池中又定义了一个常量使得引用指向了这个新的常量，而不是在原有值的基础上做修改。

　　以下是final的常见用法：

　　1、 用final关键字修饰类，使得该类不可以被继承。例如String类就利用final修改使得它不可以被继承。

　　2、 用final关键字修饰成员变量，使得该变量变为不可变的对象引用，此时应该给它赋初值，之后它不能被重新赋值，准确来讲是它的引用不可改变。注意是引用不可改变，不代表被引用的对象内部不能改变，如果引用的对象本身是一个引用类型例如：

[复制代码](javascript:void(0);)

private Student stu = new Student();

stu.setName(“kevin”);

stu.setName(“ylf”); //这是合法的

//stu = new Student(); //这不合法，因为它改变了stu的引用

[复制代码](javascript:void(0);)

　　3、 比较通用的就是开篇提到的和static配合使用定义常量。

　　个人认为final关键字的使用比较巧妙，用的好可以省去大量代码来保证线程安全，用的不好则是东施效颦带来严重后果，有关final关键字和线程安全可以参考《Java并发编程实战》一书对于fianl的讲解。

**第16条：复合优先于继承**

　　这条建议也是十分常见，最实际的例子就是[《组合模式》](http://www.cnblogs.com/yulinfeng/p/5863159.html)的运用。在JDK中一个很明显的例子就是显示锁的实现，例如ReentrantLock继承自Lock接口，但其内部实现则是通过AbstractQueueSynchronizer。

**第17条：要么为继承而设计，并提供文档说明，要么就禁止继承**

　　不要过度设计。

　　面向对象编程，从一开始被洗脑难免在上手写代码时都会首先思考有没有公共方法啊，能不能把两个类抽象成一个父类再继承啊等，慎重使用继承，当要使用继承时一定要在文档注释中写明重写这个方法会给其他方法带来什么影响。书中给出建议如果类并不是为了继承而生，那么这个类应该用final修饰禁止子类化。

**第18条：接口优于抽象类**

　　接口和抽象类的异同这是Java基础中的基础。我们不妨来回顾下Java和抽象类的区别：

**接口不能被实例化不能有构造器，抽象类也不能被实例化但可以有构造器；**

**接口中不能有实现方法（JDK8在接口中可以有实现方法，称“默认方法”），抽象类中可以有实现方法，也可以没有；**

**接口方法的默认修饰符就是public，不可以用其他修饰符，抽象类可以有public、protected、default。**

　　回到“接口优于抽象类”的问题上来，原因就是Java只支持单继承，但可以实现多个接口。有抽象类的地方基本上都可以看到其中的方法很多是模板方法（有关模板方法模式参考[《模板方法模式》](http://www.cnblogs.com/yulinfeng/p/5902164.html)）。

**第19条：接口只用于定义类型**

　　这个条目中建议接口不要只用于定义常量使之成为常量接口，如果一个类只有常量应该使用枚举类型或者不可实例化的工具类。JDK中的反例就是java.io.ObjectStreamConstant。

**第20条：类层次优于标签类**

　　标签类是指在类中定义了一个变量，使用该变量的值来控制该做什么动作。

　　书中举例：定义一个Figure类，使用Shapre变量，可以传入“长方形”或者“圆形”，根据传入的类型不同调用共同的方法。这个就是一个标签类，如果新增一个“三角形”的话，就得修改这个标签类的代码。

　　更好的方法就是利用继承，合理利用继承能更好的体现面向对象的多态性。

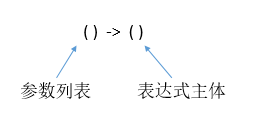
**第21条：用函数对象表示策略**

　　什么是函数对象？实际上这是在JDK8之前没有Java不支持lamda表达式，方法参数不能传递一个方法只能通过传递对象的方式“曲线救国”，例如Arrays.sort(T[] a, Comparator<? super T> c)方法，第一个参数传递数组，根据传入第二个自定义的比较类中的比较方法进行排序。如果能传入函数指针、Lambda表达式等，那就自然不用传递一个类。

　　从JDK8开始Java已经支持了lambda表达式，不妨简单了解下JDK8的lambda表达式。

　　广义上来讲JDK8中lambda表达式有两个部分组成：**一是lambda表达式本身，二是函数式接口**。函数式接口实际上就是指只包含一个抽象方法的接口，比如Runnable接口只包含run抽象方法。而lambda表达式本身实际上则是对抽象方法的实现。

　　首先lambda表达式的语法格式如下所示：



　　例如

(n) -> System.out.println(n)

表示打印n。

　　上面提到lambda是对抽象方法的实现，那么实际上这条lambda表达式对应的就是：

public void demo(String n) {

System.out.println(n);

}

　　lambda表达式不过是一个匿名方法实现而已，接下来我们看看到底是如何使用lambda表达式。

[复制代码](javascript:void(0);)

1 package com.jdk8;

2

3 /\*\*

4 \* 函数式接口，只包含一个抽象方法

5 \* Created by 余林丰 on 2017/8/15/0015.

6 \*/

7 public interface LambdaDemo {

8 void demo(String n);

9 }

[复制代码](javascript:void(0);)

[复制代码](javascript:void(0);)

1 package com.jdk8;

2

3 /\*\*

4 \* lambda例子

5 \*/

6 public class App {

7 public static void main( String[] args ) {

8 LambdaDemo lambdaDemo = (n) -> System.out.println(n); //实例化LambdaDemo类，同时也lambda表达式实现了demo方法

9 lambdaDemo.demo("hello lambda");

10 }

11 }

[复制代码](javascript:void(0);)

2017-08-15

**第22条：优先考虑静态成员类**

　　一般情况下我们可能提到最多的是“内部类”这个名词，实际上在类“内部”的类叫做“嵌套类”。嵌套类分为四种：静态成员类、非静态成员类、匿名类和局部类。除了静态成员类，其余三种才被称之为内部类。

　　匿名类提及的比较多：

[复制代码](javascript:void(0);)

public class App {

public static void main( String[] args ) {

Thread thread = new Thread(new Runnable() {

@Override

public void run() {

System.out.println("这是一个内部类");

}

}); //回顾一下lambda表达式，可表示为Thread thread = new Thread(() -> System.out.println("这是一个内部类"));

thread.start();

}

}

[复制代码](javascript:void(0);)

　　静态成员类相比较于非静态成员类就是多了一个static关键字修饰类，另外一个更重要的区别在于**非静态成员类的每个实例都包含一个额外的指向外围对象的引用，保存这份引用要耗费时间和空间。**

　　举个例子，在JDK7中，HashMap内部使用Entry类表示每个键-值对，这个类是static静态的，如果将static去掉仍然可以工作，但每个entry中将会包含一个指向该Map的引用，这样就浪费了空间和时间。

　　那么什么时候使用静态什么时候使用非静态呢？

　　书中给出了比较明确的原则：**如果声明成员类不要求访问外围实例，就要始终把static修饰符放在它的声明中**。也就是说如果成员类和外围实例类有交互，那这个类就应该是非静态的，如果没有交互而是作为外围类的一个组件存在在应使用静态的。

　　最后一个是局部类，只要是在任何“可以声明局部变量的地方”，都可以声明局部类，用得最少，如果要使用那也必须非常简短。

**第23条：请不要新代码中使用原生态类型**

　　从这条开始涉及泛型相关的点。

　　从JDK5开始Java新增了“泛型”新特性，例如：List<String>，在这之前则只有List不会限定类型。

　　如今的JDK版本中还是可以写原生类型，但这会带来隐藏的不安全问题。如果在声明一个列表变量时不使用泛型而使用原生类型，如下：

[复制代码](javascript:void(0);)

/\*\*

\*

\* Created by 余林丰 on 2017/8/17

\*/

public class Main {

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

List list = new ArrayList();

list.add("hello");

list.add(2);

}

}

[复制代码](javascript:void(0);)

　　可以看到可以插入任意类型的数据。如果使用泛型：

[复制代码](javascript:void(0);)

/\*\*

\*

\* Created by 余林丰 on 2017/8/17

\*/

public class Main {

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

List<String> list = new ArrayList<String>();

list.add("hello");

//list.add(2); //编译期就会出错

}

}

[复制代码](javascript:void(0);)

　　简而言之，使用泛型相对“安全”，从一开始就能限定数据类型，防止之后不小心插入了错误的类型，而对于原生态类型则不会检查插入的类型，有可能在以后插入了其他类型而只有在运行时才抛出异常，所以鼓励使用泛型。

　　另外书中提到了“?”通配符和“<Object>”之间的区别，可参考[《Java中的Object，T（泛型），?区别》](http://www.cnblogs.com/yulinfeng/p/6056038.html)。

**第24条：消除非受检警告**

　　在使用泛型时，常常会不可避免的出现一些警告，这些警告可能并不会引起开发人员的注意，例如：

https://images2017.cnblogs.com/blog/630246/201708/630246-20170817224912975-125185960.png

　　我们应该在代码中尽量消除警告，如果无法消除警告，同时可以证明引起警告的代码是类型安全的，可以使用@SuppreWarnings(“unchecked”)注解，并在注释中加以解释。

**第25条：列表优先于数组**

　　这条建议可引申出列表和数组的区别是什么，列表的内部实现。

　　首先数组是协变的，这里的“变”指的是数据类型，而不是说数组的长度，数组的长度当然从一开始就确定不可改变，但对于以下代码确实合法的：

[复制代码](javascript:void(0);)

/\*\*

\*

\* Created by 余林丰 on 2017/8/17

\*/

public class Main {

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

Object[] objects = new Long[1];

objects[0] = "hello world";

System.out.println(objects[0]);

}

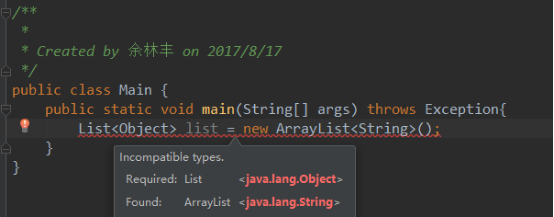
}

[复制代码](javascript:void(0);)

　　但合法仅存在于编译时期，在运行时会抛出以下错误：

https://images2017.cnblogs.com/blog/630246/201708/630246-20170817225002443-1010344475.png

　　泛型则是不可变的，这个不可变指的也是数据类型例如以下代码：



　　这段代码在编译期就会报错。综上，利用数组只有在运行时才会报错，利用列表在编译时就会报错，我们当然希望在错误能在编译时尽早发现。

　　大家可能都试图写过“泛型数组”：

List<String>[] lists = new ArrayList<String>[1];

　　但是却发现是错误的，编译时抛出Generic array creation错误，并且一时难以想象为什么不能创建泛型数组，书中举了以下例子来说明：

[复制代码](javascript:void(0);)

List<String>[] lists = new ArrayList<String>[1]; //先假设能创建泛型数组

List<Integer> intList = Arrays.asList(42); //Integer列表

Object[] objects = lists; //根据数组的“协变性”是合法的，例如上面提到的Object[] objects = new Long[1]

objects[0] = intList; //List<String>和List<Integer>在运行时类型会被擦除为List

String s = lists[0].get(0); //上一步操作过后，实际上取出的是一个initList，即取出是一个Integer

[复制代码](javascript:void(0);)

　　假设第一步不会报错，那么上面的例子在编译时就不会出错，但一到了运行时最后一句话就会抛出ClassCastException异常，也就是说与其在运行时出错，不如将它提前到编译时即不允许创建泛型数组。**这就是为什么创建泛型数组是非法的原因：因为它不是类型安全的。要是它合法，编译器在其他正确的程序中发生的转换就会在运行时失败，并出现一个ClassCastExcetion异常。这就违背了泛型系统提供的基本保证。**

　　不过有一个例外，以下的创建方式却是合法的：

List<?>[] lists = new ArrayList<?>[1];

　　泛型在运行时它的类型会被擦除，也就是说泛型是不可具体化的，它在运行时所包含的信息比它在编译时所包含的信息更少。唯一可具体化的参数化类型就是无限制的通配符类型，也就是上面提到的例子如List<?>。

此条目几乎一直在说数组和泛型不能很好的配合使用，如果遇到泛型的情况，应该首先考虑列表。

**第26条：优先考虑泛型**

　　引用书中的话“**一般来说，将集合声明参数化，以及使用JDK所提供的泛型和泛型方法，这些都不太困难。编写自己的泛型会比较困难一些，但是值得花些时间去学习如何编写**”。

　　举一个简单例子来帮助我们如何正确编写泛型：

[复制代码](javascript:void(0);)

/\*\*

\* Created by 余林丰 on 2017/8/17/0017.

\*/

public class Test<E> {

private E[] elements;

public Test() {

//elements = new E[16]; //编译时出错，不能创建不可具体化的类型的数组

elements = (E[]) new Object[16];

}

}

[复制代码](javascript:void(0);)

　　还有另外一种方法：

[复制代码](javascript:void(0);)

/\*\*

\* Created by 余林丰 on 2017/8/17/0017.

\*/

public class Test<E> {

private Object[] elements;

public Test() {

//elements = new E[16]; //编译时出错，不能创建不可具体化的类型的数组

elements = new Object[16];

}

public E test() {

return (E) elements[0];

}

}

[复制代码](javascript:void(0);)

　　当在编写不可避免要使用数组时，可参考以上两种数组和泛型的实现方式。

**第27条：优先考虑泛型方法**

　　泛型方法即在定义方法的返回值前加上<E>，例如Collections.sort方法，至于优点不再多说，一句话能用泛型尽量用泛型。

public static <T> void sort(List<T> list, Comparator<? super T> c)

　　这个方法的第二个参数实际上是下条要提到的有限制通配符。

**第28条：利用有限制通配符来提升API的灵活性**

　　之前我们提到了<?>形式的无限制通配符，这里则是有限制通配符。上一条目中已经出现了有限制通配符，它一共有这么2种：

**<? extends E>**：表示可接受E类型的子类型；

**<? super E>**：表示可接受E类型的父类型。

**第29条：优先考虑类型安全的异构容器**

　　“异构”的英文heterogeneous意为多种多样的，书中所举的例子我认为非常有参考价值，仔细品味。

[复制代码](javascript:void(0);)

import java.util.HashMap;

import java.util.Map;

/\*\*

\* 异构容器

\* Created by yulinfeng on 8/17/17.

\*/

public class Favorites {

private Map<Class<?>, Object> favorites = new HashMap<Class<?>, Object>();

public <T> void putFavorite(Class<T> type, T instance) {

if (type == null) {

throw new NullPointerException();

}

favorites.put(type, instance);

}

public <T> T getFavorite(Class<T> type) {

return type.cast(favorites.get(type));

}

}

[复制代码](javascript:void(0);)

[复制代码](javascript:void(0);)

/\*\*

\* Created by yulinfeng on 8/17/17.

\*/

public class Main {

public static void main(String[] args) {

Favorites f = new Favorites();

f.putFavorite(String.class, "Java");

f.putFavorite(Integer.class, 0xcafebabe);

f.putFavorite(Class.class, Favorites.class);

String favoriteString = f.getFavorite(String.class);

Integer favoriteInteger = f.getFavorite(Integer.class);

Class<?> favoriteClass = f.getFavorite(Class.class);

System.out.printf("%s %x %s", favoriteString, favoriteInteger, favoriteClass.getName());

}

}

[复制代码](javascript:void(0);)

　　Favorite类使用起来有点Map的感觉，putFavorite方法就类似Map.put，或者说用Map不就能实现吗？例如：

[复制代码](javascript:void(0);)

import java.util.HashMap;

import java.util.Map;

/\*\*

\* Created by yulinfeng on 8/17/17.

\*/

public class Main {

public static void main(String[] args) {

Map<Class<?>, Object> map = new HashMap<Class<?>, Object >();

map.put(String.class, "Java");

map.put(Integer.class, 122);

System.out.println(map.get(String.class));

System.out.println(map.get(Integer.class));

}

}

[复制代码](javascript:void(0);)

　　能运行和上面结果一致，但问题就在于以下代码：

[复制代码](javascript:void(0);)

import java.util.HashMap;

import java.util.Map;

/\*\*

\* Created by yulinfeng on 8/17/17.

\*/

public class Main {

public static void main(String[] args) {

Map<Class<?>, Object> map = new HashMap<Class<?>, Object >();

map.put(String.class, "Java");

map.put(Integer.class, 122);

Object str = map.get(String.class); //Integer str = (Integer) map.get(String.class);

Object in = map.get(Integer.class);

}

}

[复制代码](javascript:void(0);)

　　根据键取出来的值是Object，也就是说这是很危险的一件事情，如果代码写成上面注释那样的话在编译时是无法判断的，只有在运行时才会抛出异常。记住，能在编译时检查就在编译时检查而不要等到真正运行起来才做检查，这也就是上面Favorite所带来的好处，它是类型安全的，同时它也是异构的，这个例子值得细细品味。

2017-08-17

**第30条：用enum代替int常量**

　　对于枚举类型可能并“不常用”，之所以打引号是因为，“不常用”并不是它不好用，而是因为某项原因例如项目外包或者某个项目不需要自己来维护等等，只是追求快速地实现功能。实际上枚举类型对于强化项目代码的结构和规整很有帮助，是一个必不可少所需要强掌握的技能。

　　对于使用过枚举的来说，可能最常见的用的比较多的地方就是错误码了，例如：

[复制代码](javascript:void(0);)

1 /\*\*

2 \* 枚举类型错误码

3 \* Created by yulinfeng on 8/20/17.

4 \*/

5 public enum ErrorCode {

6 FAILURE(0, "操作失败"),

7 SUCCESS(0, "操作成功");

8

9 private int code;

10 private String msg;

11 ErrorCode(int code, String msg) {

12 this.code = code;

13 this.msg = msg;

14 }

15

16 public int getCode() {

17 return code;

18 }

19

20 public void setCode(int code) {

21 this.code = code;

22 }

23

24 public String getMsg() {

25 return msg;

26 }

27

28 public void setMsg(String msg) {

29 this.msg = msg;

30 }

31 }

[复制代码](javascript:void(0);)

[复制代码](javascript:void(0);)

1 /\*\*

2 \* Created by yulinfeng on 8/17/17.

3 \*/

4 public class Main {

5

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

7 System.out.println(ErrorCode.SUCCESS.getCode() + ErrorCode.SUCCESS.getMsg());

8 }

9 }

[复制代码](javascript:void(0);)

　　这可能是枚举使用比较常见的一种用法。实际上枚举还有其它一些比较“高级”的用法，我们不妨从书中举例来一一说明。首先用枚举来实现加减乘除四种操作：

[复制代码](javascript:void(0);)

1 /\*\*

2 \* 加减乘除枚举

3 \* Created by yulinfeng on 8/20/17.

4 \*/

5 public enum Operation {

6 PLUS, MINUS, TIMES, DIVIDE;

7

8 double apply(double x, double y) {

9 switch (this) {

10 case PLUS: return x + y;

11 case MINUS: return x - y;

12 case TIMES: return x \* y;

13 case DIVIDE: return x / y;

14 }

15 throw new AssertionError("Unknow op:" + this);

16 }

17 }

[复制代码](javascript:void(0);)

[复制代码](javascript:void(0);)

1 /\*\*

2 \* Created by yulinfeng on 8/17/17.

3 \*/

4 public class Main {

5

6 public static void main(String[] args) {

7 double x = 1.1;

8 double y = 2.2;

9 double result = Operation.PLUS.apply(x, y);

10 System.out.println(result);

11 }

12 }

[复制代码](javascript:void(0);)

　　试想如果还需要新增另外一种操作的时候却忘了新增case怎么办？编译时编译器并不会给出任何提示，同样的功能考虑以下实现能很好的避免这种遗忘新增case的情况：

[复制代码](javascript:void(0);)

1 /\*\*

2 \* 加减乘除枚举

3 \* Created by yulinfeng on 8/20/17.

4 \*/

5 public enum Operation {

6 PLUS {

7 double apply(double x, double y) {

8 return x + y;

9 }

10 },

11 MIUS {

12 double apply(double x, double y) {

13 return x - y;

14 }

15 },

16 TIMES {

17 double apply(double x, double y) {

18 return x \* y;

19 }

20 },

21 DEVIDE {

22 double apply(double x, double y) {

23 return x / y;

24 }

25 };

26

27 abstract double apply(double x, double y);

28 }

[复制代码](javascript:void(0);)

　　如果我们想要新增XXX操作，如果不实现apply方法则不会编译通过，这种方法很巧妙的躲过了因为人为失误带来的隐患。

　　枚举确实是一种比较神奇的类型，它不需要你new一个实例，并且在**枚举天生就是不可变的，因此所有的域应该为final**。一定记住像错误码或者一组固定常量的时候就要优先考虑使用枚举，而不是定义int常量甚至直接将字符串硬编码到代码中。