**Effective Java通俗理解（下）**

**第31条：用实例域代替序数**

　　枚举类型有一个ordinal方法，它范围该常量的序数从0开始，不建议使用这个方法，因为这不能很好地对枚举进行维护，正确应该是利用实例域，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),

7 SUCCESS(1);

8

9 private final int code; //上一条讲到枚举天生不可变，所有域都应该是final的。

10

11 ErrorCode(int code) {

12 this.code = code;

13 }

14

15 public int getCode() {

16 return code;

17 }

18 }

[复制代码](javascript:void(0);)

**第32条：用EnumSet代替位域**

　　前面说到枚举类型并“不常用”，那么这个EnumSet可能就更不常用了，首先来介绍写EnumSet是什么类型，它存在的意义是什么。

　　我们都知道HashSet不包含重复元素，同样EnumSet也和HashSet一样实现自AbstractSet，它也不包含重复元素，可以说它就是为Enum枚举类型而生，在《Thinking in Java》中这样描述“**Java SE5引入EnumSet，是为了通过enum创建一种替代品，以替代传统的基于int的“位标识”**”。本书中也是提到用EnumSet来代替位域。

　　关于EnumSet中的元素必须来自同一个Enum，并且构造一个EnumSet实例是通过静态工厂方法——noneOf，用法如下：

[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, DEVIDE;

7 }

[复制代码](javascript:void(0);)

[复制代码](javascript:void(0);)

1 import java.util.EnumSet;

2

3 /\*\*

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

5 \*/

6 public class Main {

7

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

9 EnumSet<Operation> enumSet = EnumSet.noneOf(Operation.class);

10 enumSet.add(Operation.DEVIDE);

11 System.out.println(enumSet);

12 enumSet.remove(Operation.DEVIDE);

13 System.out.println(enumSet);

14 }

15 }

[复制代码](javascript:void(0);)

　　书中提到的位域，实际上就是OR位运算，换句话说就是“并集”也就是Set所代表的就是并集，在使用int型枚举模式的时候可能会用到类似“1 || 2”，这个时候不如用Enum枚举加以EnumSet来实现。

**第33条：用EnumMap代替序数索引**

　　有了上一条EnumSet的经验，实际上EnumMap和HashMap也类同，不同的是它是为Enum为生的，同样它的键也只允许来自同一个Enum枚举，我们举例《Thinking in Java》中的例子（命令设计模式）：

[复制代码](javascript:void(0);)

1 /\*\*

2 \* 报警枚举

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

4 \*/

5 public enum AlamPoints {

6 KITCHEN, BATHROOM;

7 }

[复制代码](javascript:void(0);)

[复制代码](javascript:void(0);)

1 /\*\*

2 \* 命令接口

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

4 \*/

5 public interface Command {

6 void action();

7 }

[复制代码](javascript:void(0);)

[复制代码](javascript:void(0);)

1 import java.util.EnumMap;

2 import java.util.Map;

3

4 /\*\*

5 \* EnumMap

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

7 \*/

8 public class Main {

9

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

11 EnumMap<AlamPoints, Command> em = new EnumMap<AlamPoints, Command>(AlamPoints.class);

12 em.put(AlamPoints.KITCHEN, new Command() {

13 @Override

14 public void action() {

15 System.out.println("Kitchen fire");

16 }

17 });

18 em.put(AlamPoints.BATHROOM, new Command(){

19 @Override

20 public void action() {

21 System.out.println("Bathroom alert!");

22 }

23 });

24 for (Map.Entry<AlamPoints, Command> e : em.entrySet()) {

25 System.out.print(e.getKey() + ":");

26 e.getValue().action();

27 }

28 }

29 }

[复制代码](javascript:void(0);)

　　这个例子说明了EnumMap的基本用法，和HashMap除了在构造方法上的不同外，基本无异。至于本条目所说的不要使用序数，实际上看完这条目过后得出的结论就是利用Map而不是是使用数组这个意思。

**第34条：用接口模拟可伸缩的枚举**

　　在第30条里我们举了这么一个例子：

[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);)

　　同时提到当需要新增一个操作符时直接在源代码中新增即可，且不会忘记重写apply方法，但从软件开发的可扩展性来说这并不是一个好的方法，软件可扩展性并不是在原有代码上做修改，如果这段代码是在jar中的呢？这个时候就需要接口出场了，我们修改上述例子：

[复制代码](javascript:void(0);)

1 /\*\*

2 \* 操作符接口

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

4 \*/

5 public interface Operation {

6 double apply(double x, double y);

7 }

[复制代码](javascript:void(0);)

[复制代码](javascript:void(0);)

1 /\*\*

2 \* 基本操作符，实现自Operation接口

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

4 \*/

5 public enum BasicOperation implements Operation{

6 PLUS("+") {

7 public double apply(double x, double y) {

8 return x + y;

9 }

10 },

11 MIUS("-") {

12 public double apply(double x, double y) {

13 return x - y;

14 }

15 },

16 TIMES ("\*") {

17 public double apply(double x, double y) {

18 return x \* y;

19 }

20 },

21 DEVIDE ("/") {

22 public double apply(double x, double y) {

23 return x / y;

24 }

25 };

26 private final String symbol;

27

28 BasicOperation(String symbol) {

29 this.symbol = symbol;

30 }

31 }

[复制代码](javascript:void(0);)

　　当我们需要扩展操作符枚举的时候只需要重新实现Operation接口即可：

[复制代码](javascript:void(0);)

1 /\*\*

2 \* 扩展操作符

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

4 \*/

5 public enum ExtendedOperation implements Operation {

6 EXP ("^") {

7 public double apply(double x, double y) {

8 return Math.pow(x, y);

9 }

10 },

11 REMAINDER ("%") {

12 public double apply(double x, double y) {

13 return x % y;

14 }

15 };

16 private final String symbol;

17

18 ExtendedOperation(String symbol) {

19 this.symbol = symbol;

20 }

21 }

[复制代码](javascript:void(0);)

　　这样就达到代码的可扩展性，这样做的有一个小小的不足就是无法从一个枚举类型继承到另外一个枚举类型。

**第35条：注解优先于命名模式**

　　关于注解，我想很多人并没有亲自自定义过一个注解包括我，仅仅是在使用Spring框架时有使用过。确实对于注解来讲，个人认为一般很难接触自定义注解，除非是在公司的“自由框架组”或者“平台组”，这条我们就来学习如何编写自定义的注解。

　　首先来回顾Spring MVC中很常用的一条注解：

[复制代码](javascript:void(0);)

1 import org.springframework.stereotype.Controller;

2 import org.springframework.web.bind.annotation.RequestMapping;

3 import static org.springframework.web.bind.annotation.RequestMethod.GET;

4

5 /\*\*

6 \* Created by 余林丰 on 2017/4/12/0012.

7 \*/

8 @Controller

9 @RequestMapping("/")

10 public class HomeController {

11

12 @RequestMapping(value="/home", method = GET)

13 public String home() {

14 return "home";

15 }

16 }

[复制代码](javascript:void(0);)

　　选择@RequestMapping查看其源码，可以发现@RequestMapping上还有注解：

[复制代码](javascript:void(0);)

1 @Target({ElementType.METHOD, ElementType.TYPE}) //注解可使用在方法声明和类、接口或enum声明

2 @Retention(RetentionPolicy.RUNTIME) //注解应该在运行时保留

3 @Documented //注解应该被 javadoc工具记录

4 @Mapping //表明是一个web映射注解

5 public @interface RequestMapping {

6 String name() default "";

7 @AliasFor("path") //别名

8 String[] value() default {};

9 @AliasFor("value")

10 String[] path() default {};

11 RequestMethod[] method() default {};

12 String[] params() default {};

13 String[] headers() default {};

14 String[] consumes() default {};

15 String[] produces() default {};

16 }

[复制代码](javascript:void(0);)

　　定义注解时发现注解的语法有点奇怪，所有注解都只包含方法声明，但是不能为这些方法提供方法体，这些方法的行为更像是域变量，对于没有定义方法的注解称之为标记注解。方法名后通过default来声明参数的默认值，该注解的适用方法就是开头的HomeController代码示例。这是注解的基本使用和定义，一定注意：**注解永远不会改变被注解代码的语义，它们只负责提供信息供相关的程序使用。**

　　对于注解运行原理，以及如何正确使用自定义的注解在这里不做过多讲解，此条的目的在于对待“特定程序员”，注解是他们编写“工具类”、“框架类”的利器。

**第36条：坚持使用Override注解**

　　一句话，如果你要**重写**父类的方法，一定要使用Override注解，防止对方法进行了重载。

**第37条：用标记接口定义类型**

**标记接口是没有包含方法声明的接口，而只是指明一个类实现了具有某种属性的接口**，例如实现了Serializable接口表示它可被实例化。在有的情况下使用标记注解比标记接口可能更好，但书中提到了两点标记接口胜过标记注解：

　　1） 标记接口定义的类型是由被标记类的实例实现的；标记注解则没有定义这样的类型。 //这条有点不好理解，希望有大神有更加通俗地解释

　　2） 尽管标记注解可以锁定类、接口、方法，但它是针对所有，标记接口则可以被更加精确地锁定。

　　另外书中也提到标记注解优于标记接口的地方：那就是能标记程序元素而非类和接口，且在未来能给标记添加更多的信息。

**第38条：检查参数的有效性**

　　对于这一条，最常见的莫过于检查参数是否为null。

　　有时出现调用方未检查传入的参数是否为空，同时被调用方也没有检查参数是否为空，结果这就导致两边都没检查以至于出现null的值程序出错，通常情况下会规定调用方或者被调用方来检查参数的合法性，或者干脆规定都必须检查。null值的检查相当有必要，很多情况下没有检查值是否为空，结果导致抛出NullPointerException异常。

**第39条：必要时进行保护性拷贝**

　　书中所举的例子也相当有借鉴参考意义，不妨来看看：

[复制代码](javascript:void(0);)

1 import java.util.Date;

2

3 /\*\*

4 \* 开始时间不能大于结束时间

5 \* Created by yulinfeng on 2017/8/21/0021.

6 \*/

7 public class Period {

8 private final Date start; //定义为不可变的引用

9 private final Date end;

10

11 public Period(Date start, Date end) {

12 if (start.compareTo(end) > 0) {

13 throw new IllegalArgumentException(start + "after" + end);

14

15 }

16 this.start = start;

17 this.end = end;

18 }

19

20 public Date start() {

21 return start;

22 }

23

24 public Date end() {

25 return end;

26 }

27 }

[复制代码](javascript:void(0);)

　　这段代码看起来没有问题，代码中也明确了start和end是不变的。但是！Date类本身确实可变的，一定注意Date类本身是可变的。所以当出现以下代码时就会违背代码本身的意愿：

[复制代码](javascript:void(0);)

1 import java.util.Date;

2

3 /\*\*

4 \*

5 \* Created by yulinfeng on 2017/8/17

6 \*/

7 public class Main {

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

9 Date start = new Date();

10 Date end = new Date();

11 Period p = new Period(start, end); //传递的是引用的拷贝

12 end.setYear(78); //对end变量的修改是会影响到Period中的end对象，因为它们指向的是同一个实例对象

13 }

14 }

[复制代码](javascript:void(0);)

　　注释中提到传递的是引用的拷贝，那么我们在Period中重新创建一个Date实例就不再指向同一个对象实例了：

[复制代码](javascript:void(0);)

1 import java.util.Date;

2

3 /\*\*

4 \* 开始时间不能大于结束时间

5 \* Created by yulinfeng on 2017/8/21/0021.

6 \*/

7 public class Period {

8 private final Date start; //定义为不可变的引用

9 private final Date end;

10

11 public Period(Date start, Date end) {

12 /\*实例的创建应在有效性检查之前进行，避免在“从检查参数开始直到拷贝参数之间的时间段期间”从另一个线程改变类的参数\*/

13

14 this.start = new Date(start.getTime());

15 this.end = new Date(start.getTime());

16

17 if (this.start.compareTo(this.end) > 0) {

18 throw new IllegalArgumentException(start + "after" + end);

19

20 }

21 }

22

23 public Date start() {

24 return start;

25 }

26

27 public Date end() {

28 return end;

29 }

30 }

[复制代码](javascript:void(0);)

　　不过如果我们将客户端测试代码改为以下，则还是会带来新的问题，它同样是修改了Period实例，原因就是Date类是可改变的：

[复制代码](javascript:void(0);)

1 import java.util.Date;

2

3 /\*\*

4 \*

5 \* Created by yulinfeng on 2017/8/17

6 \*/

7 public class Main {

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

9 Date start = new Date();

10 Date end = new Date();

11 Period p = new Period(start, end); //传递的是引用的拷贝

12 p.end().setYear(78); //对end变量的修改是会影响到Period中的end对象，因为它们指向的是同一个实例对象

13 }

14 }

[复制代码](javascript:void(0);)

　　再次利用上述思路，在调用start()和end()方法时不直接返回成员变量，而是范围一个新的实例：

[复制代码](javascript:void(0);)

1 import java.util.Date;

2

3 /\*\*

4 \* 开始时间不能大于结束时间

5 \* Created by yulinfeng on 2017/8/21/0021.

6 \*/

7 public class Period {

8 private final Date start; //定义为不可变的引用

9 private final Date end;

10

11 public Period(Date start, Date end) {

12 /\*实例的创建应在有效性检查之前进行，避免在“从检查参数开始直到拷贝参数之间的时间段期间”从另一个线程改变类的参数\*/

13

14 this.start = new Date(start.getTime());

15 this.end = new Date(start.getTime());

16

17 if (this.start.compareTo(this.end) > 0) {

18 throw new IllegalArgumentException(start + "after" + end);

19

20 }

21 }

22

23 public Date start() {

24 return new Date(start.getTime());

25 }

26

27 public Date end() {

28 return new Date(end.getTime());

29 }

30 }

[复制代码](javascript:void(0);)

　　至此，Period就是真正的不可变了，当然有经验的程序员通常不会Date作为参宿类型，而是直接使用long。这个例子对类的不可变性很有参考意义。

**第40条：谨慎设计方法签名**

　　方法签名不仅仅是指方法命名，还包括方法所包含的参数。

　　方法命名要遵循一定的规则和规律，可参考JDK的命名；方法所包含的参数最好不应超过4个，如果超过4个则应考虑拆分成多个方法或者创建辅助类用来保存参数的分组。

2017-08-21

**第41条：慎用重载**

　　面向对象有三个特性：继承、封装、多态。重载所体现的就是多态。

　　重载和重写有区别，重写是子类的方法重新实现父类的方法，包括方法名和参数都要相同；重载则不用要求是要继承，只要求拥有相同的方法名，参数类型不同个数不同都可以称之为重载。

　　此条目下书中建议慎用重载，在举几个例子过后就能很清除地明白为什么需要慎用。

[复制代码](javascript:void(0);)

1 import java.util.\*;

2

3 /\*\*

4 \* 令人疑惑的重载示例

5 \* Created by yulinfeng on 8/22/17.

6 \*/

7 public class CollectionClassifier {

8 public static String classify(Set<?> set) {

9 return "Set";

10 }

11

12 public static String classify(List<?> list) {

13 return "List";

14 }

15

16 public static String classify(Collection<?> collection) {

17 return "Unknown Collection";

18 }

19

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

21 Collection<?>[] collections = {new HashSet<String>(), new ArrayList<String>(), new HashMap<String, String>().values()};

22 for (Collection<?> c : collections) {

23 System.out.println(classify(c));

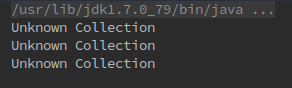
24 }

25 }

26 }

[复制代码](javascript:void(0);)

　　运行结果可能有点让人疑惑：



　　这引出了一个问题，先说结论：重载方法的选择是静态的，而对于重写方法的选择则是动态的。在《深入理解Java虚拟机》第8.3.2章节中有讲解，这涉及到重载与重写在Java虚拟机中是如何实现的。

　　再来看一个关于重载的例子：

[复制代码](javascript:void(0);)

1 /\*\*

2 \* 静态分派——重载

3 \* Created by yulinfeng on 8/22/17.

4 \*/

5 public class StaticDispatch {

6 static abstract class Human{

7 }

8

9 static class Man extends Human {

10 }

11

12 static class Woman extends Human {

13 }

14

15 public void sayHello(Human guy) {

16 System.out.println("hello, guy!");

17 }

18

19 public void sayHello(Man guy) {

20 System.out.println("hello, gentleman!");

21 }

22

23 public void sayHello(Woman guy) {

24 System.out.println("hello, lady!");

25 }

26

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

28 Human man = new Man();

29 Human woman = new Woman();

30 StaticDispatch staticDispatch = new StaticDispatch();

31 staticDispatch.sayHello(man);

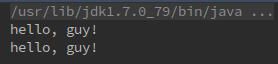
32 staticDispatch.sayHello(woman);

33 }

34 }

[复制代码](javascript:void(0);)

　　思考上面例子的执行结果，实际上有了第一个例子，这个例子可能也能“猜”得到。



　　这两个例子实际上有异曲同工之妙，相似点在：

　　第一个例子：for (Collection<?> c : collections)。第二个例子：Human man = new Man();Human woman = new Woman();。

　　实际上“Human”称为变量的静态类型（外观类型），而“Man”和“Woman”则称为变量的实际类型。本例中实际上就是定义了两个静态类型但实际类型不同的变量，但是**虚拟机在重载时是通过参数的静态类型而不是实际类型作为判定一句的，静态类型又是在编译器可知的，所以在编译时就能确定使用哪个重载版本**。这样看来上面两个例子的执行结果就能解释了，因为它们的静态类型确定为了Collection和Human，故在编译时就选用了这两个参数类型的重载方法。**所有依赖静态类型来定位方法执行版本的分派动作称为静态分派。**

　　下面我们顺便说下“重写”。

[复制代码](javascript:void(0);)

1 /\*\*

2 \* 动态分派——重写

3 \* Created by yulinfeng on 8/22/17.

4 \*/

5 public class DynamicDispatch {

6 static abstract class Human {

7 protected abstract void sayHello();

8 }

9

10 static class Man extends Human {

11 @Override

12 protected void sayHello() {

13 System.out.println("man say hello");

14 }

15 }

16

17 static class Woman extends Human {

18 @Override

19 protected void sayHello() {

20 System.out.println("woman say hello");

21 }

22 }

23

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

25 Human man = new Man();

26 Human woman = new Woman();

27 man.sayHello();

28 woman.sayHello();

29 man = new Woman();

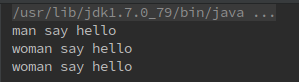
30 man.sayHello();

31 }

32 }

[复制代码](javascript:void(0);)

　　这个例子的执行结果应该不会感到疑惑了：



　　显然这并不是根据静态类型来选择的重写方法，简单推理一下可得只是按照实际类型来选择确定的重写方法，而实际类型又是在编译时不可知只有在运行时才确定的，所以这有个较为专业的叫法——动态分配。**也就是说在运行时根据实际类型确定方法执行版本的分派过程称为动态分配。**

　　清楚了重载和重写的虚拟机实现过后，回到此条目的主题——慎用重载。

　　同样给出书中的例子就能说明，重载一定要慎重使用：

[复制代码](javascript:void(0);)

1 import java.util.ArrayList;

2 import java.util.List;

3 import java.util.Set;

4 import java.util.TreeSet;

5

6 /\*\*

7 \* 慎用重载

8 \* Created by yulinfeng on 8/22/17.

9 \*/

10 public class Main {

11

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

13 Set<Integer> set = new TreeSet<Integer>();

14 List<Integer> list = new ArrayList<Integer>();

15 for (int i = -3; i < 3; i++) {

16 set.add(i);

17 list.add(i);

18 }

19 for (int i = 0; i < 3; i++) {

20 set.remove(i);

21 list.remove(i);

22 }

23 System.out.println(set + " " + list);

24 }

25 }

[复制代码](javascript:void(0);)

　　这个例子本身想要的打印结果是[-3, -2, -1] [-3, -2, -1]，结果运行确是：

https://images2017.cnblogs.com/blog/630246/201708/630246-20170822222643621-2019968731.png

　　原因就在于对于List的remove方法有两个，这两个在这里产生了歧义，其中一个是删除下标索引元素，另一个是删除集合中的元素。如果将第21行

list.remove(i);

修改为

list.remove(（Integer）i);

则避免得到了我们想要的结果。

　　这就是重载带来的“危害”，稍不留神就出现了致命问题。比较好的解决办法学习ObjectOutputStream类中做法：writeBoolean(boolean)，writeInt(int)，writeLong(long)。如果两个重载方法的参数类型很相似，那一定得考虑这样做是否容易造成“程序的误解”。

**第42条：慎用可变参数**

　　具有可变参数的方法可以传入0个或者多个参数，这种方法我相信自己写的可能在少数，用得最多的可能要属反射中的getDeclaredMethod方法：

public Method getDeclaredMethod(String name, Class<?>... parameterTypes)

　　为什么要慎用，其中有一个原因就是很有可能在没有传入参数的时候程序没有做任何保护而导致程序错误。另外有一个原因就是它会带来一定的性能问题，EnumSet类在传入少量参数的时候是直接调用具体的方法，只有在传入大量参数时才会调用可变参数的方法，这也是它在性能方面有优势的原因，因为**可变参数的每次调用都会导致进行一次数组分配和初始化。**

　　总之，“在定义参数数目不定的方法时，可变参数是一种很方便的方式，但是它们不应该被过度滥用。如果使用不当，会产生混乱的结果”。

**第43条：返回零长度的数组或者集合，而不是null**

　　我在阅读《Google Guava官方教程》时所读到最开头就是——**使用和避免null：null是模棱两可的，会引起令人困惑的错误，有些时候它让人很不舒服**。

　　例如对于一个Map，调用其get(key)方法，此时若返回null，可能表示这个key所对应的值本身就是null，或者表示这个Map中没有这个key值。这是两种截然不同的语义。

　　书中仅是说明对于零长度的数组或者集合不应该返回null，实际上对于所有的情况，都不要轻易返回null，特别是在语义不清的情况，更别说返回null时有的客户端程序并没有处理null的这种情况。

　　如果一定要用到null，更好的办法是单独维护它。

**第44条：为所有导出的API元素编写文档注释**

　　说实话，别说文档注释，连普通的方法注释也不是人人都会去编写，连参数和返回值，以及方法用途都不会注明。如今越来越强大的IDE，只要轻轻敲几个快捷键就能方便的生成文档注释模板，不要觉得麻烦，起码的参数、返回值、用途的注释一定要写。

　　关于文档注释这里不再做介绍。

2017-08-22

**第45条：将局部变量的作用域最小化**

　　之前在第13条谈到过成员变量的可访问性最小化，此处指的在一个方法中定义的局部变量。

　　简单一句话，用到的时候在定义，不要提前定义，当然在某种特殊情况下例外。想到之前看的《重构：改善既有代码的设计》书中有关成员变量的看法是不建议在程序中使用成员变量而是使用查询来代替。

**第46条：for-each循环优先于传统的for循环**

　　对于for-each的语法格式：

for (Element e : elements) {

doSomething(e);

}

　　当对集合、数组中的元素只做遍历时应首选for-each，而不是通过for循环手动移动数组下标。

**第47条：了解和使用类库**

　　JDK中内置了大量的工具类库，但很多“不为人知”。这实际上考研的是编程人员对Java基础的掌握程度，例如输出数组的方法：Arrays.toString等等，再比如判断是否字符串为空是实际上有isEmpty方法的。书中建议每个程序员都应该熟悉**java.lang、java.util**。

在进行工程项目类的开发时，不应重复造轮子，利用现有的已成熟的技术能避免很多bug和其他问题。除非自己业余爱好研究，重复造轮子我认为就很能提高编程水平了。

**第48条：如果需要精确的答案，请避免使用float和double**

　　float和double表示浮点类型数据，对于要精确到小数点的数值运算，通常下意识的会选择float或者double类型，但实际上这两种类型对于精确的计算是存在一定隐患的。

　　对于精确的数值计算，首推**BigDecimal**，或者可以使用int、long型将单位缩小不再有小数点。书中举了详细例子来说明。

**第49条：基本类型优先于装箱基本类型**

　　简单回顾下Java中的数据类型分为：基本类型和引用类型。基本类型包括：byte、short、int、long、float、double、char、boolean。引用类型则是String、List等。

　　对于int型的数据我们对它的数值大小判断直接上就是“==”，但对于其装箱类型Integer呢？

[复制代码](javascript:void(0);)

1 /\*\*

2 \* 装箱基本类型Integer的数值比较

3 \* Created by 余林丰 on 2017/8/23

4 \*/

5 public class Main {

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

7 Integer a = 10;

8 Integer b = 10;

9 Integer c = 1000;

10 Integer d = 1000;

11 System.out.println(a == b);

12 System.out.println(c == d);

13 }

14 }

[复制代码](javascript:void(0);)

　　它的执行结果有点令人疑惑：

https://images2017.cnblogs.com/blog/630246/201708/630246-20170823222124886-2142726846.png

　　这是由于在Integer中会缓存-128~127的小数值，在自动装箱的时候对这些小数值能直接比较。再来看下面例子：

[复制代码](javascript:void(0);)

1 /\*\*

2 \* 装箱基本类型Integer的数值比较

3 \* Created by 余林丰 on 2017/8/23

4 \*/

5 public class Main {

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

7 Integer a = new Integer(10);

8 Integer b = new Integer(10);

9 Integer c = new Integer(1000);

10 Integer d = new Integer(1000);

11 System.out.println(a == b);

12 System.out.println(c == d);

13 }

14 }

[复制代码](javascript:void(0);)

　　它的执行结果为：

https://images2017.cnblogs.com/blog/630246/201708/630246-20170823222155214-291583361.png

　　这里不再涉及自动装箱，有点类似String的比较，你把它当做引用类型，引用类型的“==”比较的都是其引用是否相等就能理解了。看到这里估计就差不多明白了，基本类型在有的情况优先于装箱基本类型，况且装箱基本类型还会带来性能问题。

　　而对于有的特定条件，例如集合的类型参数，这个时候就必须使用装箱类型，没得选。

　　综上，在有的选的情况下，有限考虑基本类型。

**第50条：如果其他类型更合适，则尽量避免使用字符串**

　　关于字符串是个万年不变的“考题”。Java中字符串不是一种基本类型，它实际上还是有char数组实现，在C中也并没有字符串而只有char类型。String字符串从诞生开始就注定不一般。

　　书中提到几种不应该使用字符串的情况，我认为在编码过程往往为了图省事将什么类型都定义为字符串，例如手机号，甚至是boolean类型定义为”true”或者”false”字符串。不应为了省事轻易使用字符串，而要选择更为恰当的类型。

**第51条：当心字符串连接的性能**

　　我们都知道String字符串是不可变的，每次对一个字符串变量的赋值实际上都在内存中开辟了新的空间。如果要经常对字符串做修改应该使用StringBuilder（线程不安全）或者StringgBuffer（线程安全），其中StringBuilder由于不考虑线程安全，它的速度更快。

**第52条：通过接口引用对象**

　　考虑程序代码的灵活性应该优先使用接口而不是类来引用对象，例如：

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

　　这样带来的好处就是可以更换list的具体实现只需一行代码，之前有谈到将接口作为参数的类型，这两者配合使用就能最大限度实现程序的灵活性。

　　但如果是类实现了接口，但是它提供了接口中不存在的额外方法，且程序依赖这些额外方法，这个时候用接口来代替类引用对象就不合适了。

2017-08-23

**第53条：接口优先于反射机制**

　　其实在有关接口的建议中所推崇的都是面向接口编程，此条也不例外。

　　首先是反射的使用一定要慎重，它能在运行时访问对象，但它也有以下负面影响：

**丧失了编译时类型检查**

**执行反射访问所需要的代码非常笨拙和冗长（这需要一定的编码能力）**

**性能损失**

　　在使用反射时利用接口指的是，在编译时无法获取相关的类，但在编译时有合适的接口就可以引用这个类，当在运行时以反射方式创建实例后，就可以通过接口以正常的方式访问这些实例。

**第54条：谨慎地使用本地方法**

　　所谓的本地方法就是在JDK源码中你所看到在有的方法中会有“native”关键字的方法，这种方法表示用C或者C++等本地程序设计语言编写的特殊方法。之所以会存在本地方法的原因主要有：访问特定平台的接口、提高性能。

　　实际上估计很少很少在代码中使用本地方法，就算是在设计比较底层的库时也不会使用到，除非要访问很底层的资源。当使用到本地方法时唯一的要求就是全面再全面地测试，以确保万无一失。

**第55条：谨慎地进行优化**

　　我在实际编码过程中，常常听到别人说，这么实现性能可能会好一点，少了个什么什么性能会好一点，甚至是少了个局部变量也会提到这么性能要好一点，能提高一点是一点。

　　书中引用了三句话提出了截然不同的观点：

**很多计算上的过失都被归咎于效率（没有必要达到的效率），而不是任何其他的原因——甚至包括盲目地做傻事。**

**——William A.Wulf**

**不要去计较效率上的一些小小的得失，在97%的情况下，不成熟的优化才是一切问题的根源。**

**——Donald E.Knuth**

**在优化方面，我们应该遵守两条规则：**

**规则1：不要进行优化**

**规则2:（仅针对专家）：还是不要进行优化——也就是说，在你还没有绝对清晰的未优化方案之前，请不要进行优化。**

**——M.A.Jackson**

　　上面几句话看似让你不要做优化，这当然不可能。实际上是在编码中如果你没有考虑清楚就冒然想当然的去做优化，常常可能是得不偿失，就像我开头提到的那样，甚至为了优化性能而去减少一个局部变量。

　　正确的做法应该是，写出结构优美、设计良好的代码，不是写出快的程序。

　　性能的问题应该有数据做支撑，也就是有性能测试软件对程序测试来评判出性能问题出现在哪个地方，从而做针对性的修改。

**第56条：遵守普遍接受的命名惯例**

　　阿里巴巴针对Java已经出了一份《阿里巴巴Java开发手册》，这本手册就是很好的参考。就说书里没有的一点，不要使用拼音！

**第57条：只针对异常的情况才使用异常**

　　接着这几条建议是针对异常，关于异常我个人感觉就是人人都会用，但并不是人人都能用得好。有公司自研框架就规定了如何处理异常的方法，以供程序员统一异常处理。

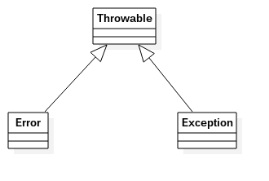
　　此条建议在书中所给出的例子我相信实际上也没有几个会写出来，异常就是超出意料之外的错误，也就是说正常的控制流逻辑是不会走到异常的，所以不要再正常的控制流中出现异常来对程序做正常逻辑处理。

2017-08-27

**第58条：对可恢复的情况使用受检异常，对程序错误使用运行时异常**

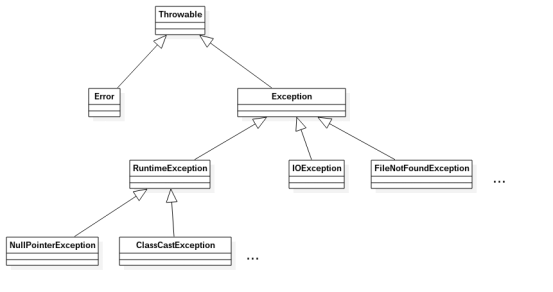
　　这里会涉及几个异常相关的概念，先对异常分类做一个简单的梳理。

　　所有的异常、错误都继承自Throwable，它直接包含了两个子类Error和Exception。



　　Error用来表示编译时和系统错误，Exception是可以被抛出的基本类型，通常情况下我们只关心Exception异常，而Error错误是程序员控制不了的系统错误。

　　对于Exception异常又分为两种：受检查的异常和不受检查的异常，受检查的异常直接继承自Exception，不受检查的异常则继承自RuntimeException（RuntimeException也继承自Exception)。



　　受检查的异常在编码中就是需要被try-catch捕获或者通过throws抛出的异常，例如在进行I/O操作时候常常都会明确要求对文件的操作需要对异常进行处理。

　　而对于不受检查的异常就是不需要在代码中体现但也可以对这种异常出现时做处理，例如经常遇到的就是空指针异常（NullPointerException）。因为不受检查的异常全部继承自RuntimeException，此条目中所说的“运行时异常”也就是不受检查的异常。

　　什么时候使用受检查的异常（throws Exception），什么时候使用不受检查的异常（throws RuntimeException），本书中给出原则是：**如果期望调用者能够适当地恢复，对于这种情况就应该使用受检的异常。对于程序错误，则使用运行时异常。**例如在数据库的事务上通常就会对异常做处理，防止出现数据不一致等情况的发生。

**第59条：避免不必要地使用受检的异常**

　　关于这条建议书中实际上是建议如何更好的“设计”异常处理。

　　对于异常本身初衷是使程序具有更高的可靠性，特别是受检查的异常。但滥用受检查的异常可能就会使得程序变得负责，给程序员带来负担。

　　两种情况同时成立的情况下就可以使用受检查的异常：1、正确地使用API并不能阻止这种异常条件的产生；2、如果一旦产生异常，使用API的程序员可以立即采取有用的动作。以上两种情况成立时，就可以使用受检查的异常，否则可能就是徒增烦恼。

　　另外在一个方法抛出受检查异常时，也需要仔细考量，因为对于调用者来讲就必须处理做相应处理，或捕获或继续向上抛出。如果是一个方法只抛出一个异常那么实际上可以将抛出异常的方法重构为boolean返回值来代替。

**第60条：优先使用标准的异常**

　　对异常的时候都知道可以自定义异常，但往往在尚不标准的开发工程中都是不管三七二十一统统捕获或者抛出Exception异常。实际上更好的是使用内置的标准的异常而不是这么笼统。例如参数值不合法就抛出llegalArgumentException等。

　　借用书中的话：**专家级程序员与缺乏经验的程序员一个最主要的区别在于，专家追求并且通常也能够实现高度的代码重用。**

**第61条：抛出与抽象相对应的异常**

　　看题目会觉得不知所云，这实际上讲的是“异常转译”，所谓异常转译简单来说指的就是将一个异常转换成另外一个异常，例如AbstractSequentialList#get方法。

　　首先查看对于列表List类 get方法的规范要求：

[复制代码](javascript:void(0);)

//List

/\*\*

\* Returns the element at the specified position in this list.

\*

\* @param index index of the element to return

\* @return the element at the specified position in this list

\* @throws IndexOutOfBoundsException if the index is out of range

\* (<tt>index &lt; 0 || index &gt;= size()</tt>)

\*/

E get(int index);

[复制代码](javascript:void(0);)

　　可以看到关于List的get方法，其子类需要抛出未受检的异常（RuntimeException）——IndexOutOfBoundsException。

[复制代码](javascript:void(0);)

//AbstractSequentialList#get

/\*\*

\* Returns the element at the specified position in this list.

\*

\* <p>This implementation first gets a list iterator pointing to the

\* indexed element (with <tt>listIterator(index)</tt>). Then, it gets

\* the element using <tt>ListIterator.next</tt> and returns it.

\*

\* @throws IndexOutOfBoundsException {@inheritDoc}

\*/

public E get(int index) {

try {

return listIterator(index).next();

} catch (NoSuchElementException exc) {

throw new IndexOutOfBoundsException("Index: "+index); //异常转译，底层抛出NoSuchElementException，转译为IndexOutOfBoundsException。

}

}

[复制代码](javascript:void(0);)

　　可以看到AbstractSequentialList#get方法的异常抛出就做了异常转译，这么做的原因有几点：符合List对get方法的规范；便于理解。

　　另外还有一点就是“异常链”：底层的异常被传到高层的异常，高层的异常提供访问方法来获得底层的异常。这样做的好处在于高层能查看低层异常的原因，其坏处就是逐层上抛会消耗大量资源。

　　总之，就是当低层抛出异常时，此时要考虑是否做异常转译，使得上层方法的调用者易于理解。

**第62条：每个方法抛出的异常都要有文档**

　　这里书中主要提到要为抛出的异常建立Java的文档注释即Javadoc的@throws标签。

　　对于一些未受检的异常同样也应该在文档注释中说明，例如上面提到的List#get。

**第63条：在细节信息中包含能捕获失败的信息**

　　关于此条目的建议可以归结于如何编写好的日志。

　　在刚工作的时候关于日志的打印对我来说基本是不知道怎么来写的，不知道在哪里写，不知道怎么写，需要包含什么内容。

　　对于异常的捕获，在平时只在IDE中的控制显示并且没有日志记录他通常都会这么写：

try {

doSomething();

} catch (Exception e) {

e.printStackTrace(); //打印出异常的堆栈信息

}

　　在生产环境中当然不能这么写而需要打印到日志中，除了堆栈信息外还需要一个可跟踪的信息，这通常是一个用户的ID，总之就是需要可定位、可分析。

**第64条：努力使失败保持原子性**

　　失败的方法调用应该使对象保持在被调用之前的状态，具有这种属性的方法被称为具有失败原子性。

　　失败过后，我们不希望这个对象不可用。在数据库事务操作中抛出异常时通常都会在异常做回滚或者恢复处理，要实现对象在抛出异常过后照样能处在一种定义良好的可用状态之中，有以下两个办法：

　　1） 设计一个不可变的对象。不可变的对象被创建之后它就处于一致的状态之中，以后也不会发生变化。

　　2） 在执行操作之前检查参数的有效性。例如对栈进行出栈操作时提前检查栈中的是否还有元素。

　　3） 在失败过后编写一段恢复代码，使对象回滚到操作开始前的状态。

　　在对象的一份临时拷贝上执行操作，操作完成过后再用临时拷贝中的结果代替对象的内容，如果操作失败也并不影响原来的对象。

**第65条：不要忽略异常**

　　所谓的忽略就是写出以下代码：

try {

doSomething();

} catch (Exception e) {

}

　　当然强烈不这样写，如果一定要这么写，那就需要加以注释。

**第66条：同步访问共享的可变数据**

　　在编程中同步是一个专业术语，所谓同步指的发出一个调用时，如果没有得到结果就不返回，直到有结果后再返回。另外相对应的是异步，指的是发出一个调用时就立即返回而不在乎此时有没有结果。

　　同步和异步关注的是“消息通信机制”，通常我们提到同步的时候实际上只理解了它一部分或者干脆理解为“互斥”，这是不全对的，例如Java中synchronized关键字，经常听到教育我们说，要互斥访问某个共享变量且需要保证它线程安全的时候就用synchronized关键字。

**互斥表示当一个对象被一个线程修改的时候，可以阻止另一个线程观察到对象内部不一致的状态。同步不仅包含这层意义还包含：它可以保证进入同步方法或者同步代码块的每个线程，都看到由同一个锁保护的之前所有修改效果。**

　　对于synchronized我相信几乎人人都知道它是线程安全的重要保证，这里不再叙述它如何保证。着重强调几个术语：**活性失败：线程A对某变量值的修改，可能没有立即在线程B体现出来。**这是由于Java内存模型造成的原因，一个线程修改某个变量后并不会立即写入主存而是写到线程自身所维护的内存中，这个时候导致另一个线程从主存中取出的值并不是最新的，使用synchronized可保证这种可见性，当然还有volatile关键字。**安全性失败：例如i++操作并不是原子的，而是先+1再复制，这就有两个动作，而这两个动作的完成很有可能导致中间穿插两个线程，这个时候就会导致程序计算结果出错。**

　　简而言之，**当多个线程共享可变数据的时候，每个读或者写数据的线程都必须执行同步，以确保线程安全，程序正确运行。**

**第67条：避免过度同步**

　　上一条谈到要使用同步，这一条告诉我们不要过度使用。对于在同步区域的代码，千万不要擅自调用其他方法，特别是会被重写的方法，因为这会导致你无法控制这个方法会做什么，严重则有可能导致死锁和异常。通常，应该在同步区域内做尽可能少的工作。**获得锁，检查共享数据，根据需要转换数据，然后放掉锁。**

**第68条：executor和task优先于线程**

　　之所以推荐executor和task原因就在于这样便于管理。

　　在java.util.concurrent包与Exececutor Framework相关知识点可查看[《12.ThreadPoolExecutor线程池原理及其execute方法》](http://www.cnblogs.com/yulinfeng/p/7021293.html)、[《13.ThreadPoolExecutor线程池之submit方法》](http://www.cnblogs.com/yulinfeng/p/7039979.html)、[《14.Java中的Future模式》](http://www.cnblogs.com/yulinfeng/p/7045648.html)。

**第69条：并发工具优先于wait和notify**

　　从JDK5新增加的java.util.concurret并发包中共提供了三个方面的并发工具：Executor Framework（这在上一条中有提到）、并发集合（Concurrent Collection）以及同步器（Synchronizer）。有关并发包的相关知识我有过一个源码解读，不过很遗憾只包含前两个方面，关于同步器暂未做深入了解，参考<http://www.cnblogs.com/yulinfeng/category/998911.html>

　　随着JDK的发展，基于原始的同步操作wait和notify已不再提倡使用，因为基础所以很多东西需要自己去保证，越来越多并发工具类的出现应该转而学习如何使用更为高效和易用的并发工具。

**第70条：线程安全性的文档化**

　　书中提到了很有意思的情景，有人会下意识的去查看API文档此方法是否包含synchronized关键字，如不包含则认为不是线程安全，如包含则认为是线程安全。实际上线程安全不能“要么全有要么全无”，它有多种级别：

**不可变的——也就是有final修饰的类，例如String、Long，它们就不用外部同步。**

**无条件的线程安全——这个类没有final修饰，但其内部已经保证了线程安全，例如并发包中的并发集合类，同样它们无需外部同步。**

**有条件的线程安全——这个有的方法需要外部同步，而有的方法则和“无条件的线程安全”一样无需外部同步。**

**非线程安全——这就是最“普通”的类了，内部的任何方法想要保证安全性就必须要外部同步。**

　　线程对立的——这种类就可以忽略不计了，这个类本身不是线程安全，并且就算外部同样同样也不是线程安全的，JDK中很少很少，几乎不计，自身也不会写出这样的类，或者也不要写出这种类。

可见队员线程是否安全不能仅仅做安全与不安全这种笼统的概念，更不能根据synchronized关键字来判断是否线程安全。你应该在文档注释中注明是以上哪种级别的线程安全，如果是有条件的线程安全不仅需要注明哪些方法需要外部同步，同时还需要注明需要获取什么对象锁。

**第71条：慎用延迟初始化**

　　延迟初始化又称懒加载或者懒汉式，这在单例模式中很常见。

　　众所周知单例模式大致分为懒汉式和饿汉式，这两种方式各有其优缺点。对于饿汉式会在初始化类或者创建实例的时候就进行初始化操作，而对于懒汉式则相反它只有在实际用到访问的时候才进行初始化。

　　至于用何种方式通常来讲并没有太大的讲究，几乎是看个人习惯。而此书却单独列了一条来说明延迟初始化使用不当所带来的危害。

　　1） 使用延迟初始化时一定要考虑它的线程安全性，通常此时会利用synchronized进行同步。

　　2） 若需要对静态域使用延迟初始化，且需要考虑性能，则使用lazy initialization holder class模式：

[复制代码](javascript:void(0);)

1 /\*\*

2 \* lazy initialization holder class

3 \* Created by yulinfeng on 2017/8/30/0030.

4 \*/

5 public class Singleton {

6 private static class SingletonHolder {

7 private static Singleton singleton = new Singleton();

8 }

9

10 private Singleton() {

11

12 }

13

14 public static Singleton getInstance() {

15 return SingletonHolder.singleton;

16 }

17 }

[复制代码](javascript:void(0);)

　　这种模式不同于传统的延迟加载，当调用getInstance时候第一次读取SingletonHolder.singleton，导致SingletonHolder类得到初始化，这个类在装载并被初始化的时候会初始化它的静态域即Singleton实例，getInstance方法并没有使用被synchronized同步，并且只是执行一个域的访问这种延迟初始化的方式实际上并没有增加任何访问成本。

　　3） 若需要对实例域使用延迟初始化，且需要考虑性能，则使用双重检查模式，这种模式其实也是为了避免synchronized带来的锁定开销：

[复制代码](javascript:void(0);)

1 /\*\*

2 \* double-check idiom

3 \* Created by yulinfeng on 2017/8/30/0030.

4 \*/

5 public class Singleton {

6 private volatile Singleton singleton;

7

8 private Singleton() {

9

10 }

11 public Singleton getInstance() {

12 Singleton result = singleton;

13 if (result == null) {

14 synchronized (this) {

15 result = singleton;

16 if (result == null) {

17 singleton = result = new Singleton();

18 }

19 }

20 }

21 return result;

22 }

23 }

[复制代码](javascript:void(0);)

　　通常用的比较多的可能是对静态域应用双重检查模式。

　　最后书中的建议就是正常地进行初始化，而对于延迟初始化则徐亚慎重考虑它的性能和安全性。

**第72条：不要依赖于线程调度器**

　　第69条说的是executor和task优先于线程，此处又指不要依赖，实际上这里的不要依赖指的是不要将正确性依赖于线程调度器。例如：调整线程优先级，线程的优先级是依赖于操作系统的并不可取；调用Thrad.yield是的线程获得CPU执行机会，这也不可取。所以不要将程序的**正确性**依赖于线程调度器。

**第73条：避免使用线程组**

　　ThreadGroup我想大部分人别说用了，可能连听都没有听过，包括我也是，记住不要使用就行了。

**第74条：谨慎地实现Serializable接口**

　　这是本书最后一个章节——序列化。将一个对象编码成一个字节流这个过程就称作该对象序列化，相反的过程就被称作反序列化。

　　在Java中的类只需要实现Serializable接口即可被序列化，很放便也很简单，但这并不意味着可以随随便便的实现Serializable接口，它会带来以下几个代价：

　　1） 实现Serializable接口后就基本等同于将这个对象如同API一样暴露发布出去，这意味着你不可随意更改这个类，也就是大大降低了“改变这个类的实现”的灵活性。

　　2） 增加了出现Bug和安全漏洞的可能性，一个类的构造器往往是用来构建一个类约束关系。序列化机制是一种语言之外的对象创建机制，反序列化可以看作是一个“隐藏的构造器”，这也就是说如果按照默认的反序列化机制很容易不按照约定的构造器建立约束关系，以及很容易使对象的约束关系遭到破坏，以及遭受到非法访问。

　　3） 随着类的版本的改变，测试的负担增加。因为类的改变需要不断检查测试新版本与旧版本之间的“序列化-反序列化”是否兼容。

　　上面这三点是代价，在有的条件下是值得的，例如很常见的如果一个类将加入到某个框架中，并且该框架依赖序列化来实现对象的传输和持久化这个时候就需要这个类实现Serializable接口。

　　另外书中举了JDK中为了继承而设计的实现了Serializable接口的类,**Throwable类实现了Serializable接口，所以RMI的异常可以从服务器端传到客户端。Component实现了Serializable接口，因此GUI可以被发送、保护和恢复。HttpServlet实现了Serializable接口，因此会话状态可以被缓存。**

　　尽管一个类要实现序列化很简单，但实现前一定要想好以及设计好这个类是否需要序列化，是否值得付出上面三个代价。

**第75条：考虑使用自定义的序列化形式**

　　在这里书中谈到了对象什么时候可使用默认的序列化形式，而又在什么时候需要自定义序列化形式。**如果一个对象的物理表示法等同于它的逻辑内容，可能就适用于使用默认的序列化形式。**逻辑内容意思是这个类所要想表达传递的含义是什么，而物理表示法则表示逻辑内容所要传达含义的具体实现。

[复制代码](javascript:void(0);)

import java.io.Serializable;

/\*\*

\* 默认的自定义形式

\* Created by yulinfeng on 2017/8/31/0031.

\*/

public class Name implements Serializable{

/\*\*

\* last name

\* @serial //表示把这些文档信息放在有关序列化形式的特殊文档页中

\*/

private String lastName;

private String firstName;

private String middleName;

}

[复制代码](javascript:void(0);)

　　这个例子的逻辑来看表示一个人的姓名，从物理实现来看利用三个实例域直接就精确地反映了逻辑内容就能直接使用默认的序列化形式。

　　书中另外举了一个物理表示不同于逻辑内容的例子，一个表示字符串序列的类：

[复制代码](javascript:void(0);)

import java.io.Serializable;

/\*\*

\*

\* Created by yulinfeng on 2017/8/31/0031.

\*/

public class StringList implements Serializable{

private int size = 0;

private Entry head = null;

private static class Entry implements Serializable {

String data;

Entry next;

Entry previous;

}

}

[复制代码](javascript:void(0);)

　　可以看到物理表示是通过一个双向链表来表示的，其实我也不是很懂什么叫做“物理表示等同于逻辑内容”这句话，如果有人看到这里了希望能请教请教。

　　当一个对象的物理表示法与它的逻辑数据内容有实质性的区别时，使用默认序列化形式会有以下4个缺点：

　　1） 内部实现被束缚，这个很好理解，比如上面的例子内部实现是链表，意味着以后的版本会一直使用这个数据结构，而不能更改，或者更改导致很大的成本。

　　2） 消耗空间，上面个例子中序列化时就会记录所有的链接关系这是实现细节不值得记录，所以会消耗过多的空间。

　　3） 消耗时间，同样要遍历的时候也要逐个遍历，也会消耗时间

　　4） 栈溢出，如果递归遍历过多可能会造成栈的溢出。

　　书中有修订版，主要实现了writeObject和readObject方法以实现自定义的序列化形式。关于序列化的内容，我大概接下来会单独写一篇来详细介绍。