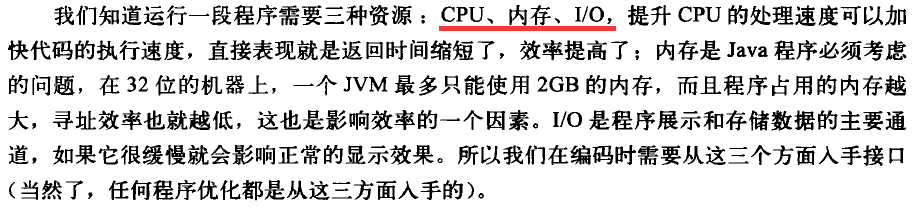
第0部分：Java性能和效率



# 提升Java性能的基本方法

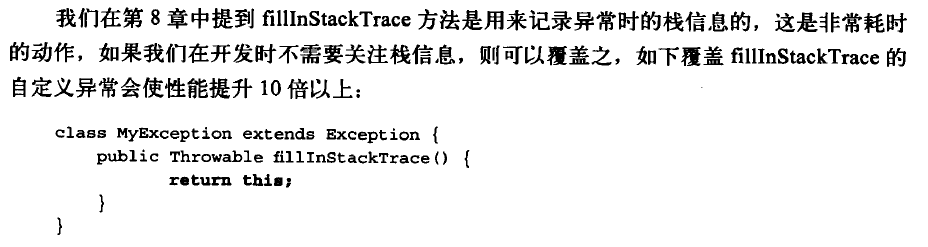
（1）不要在循环条件中计算

如果在循环（如for循环）条件中计算，则每循环一遍就要计算一次，这会降低系统效率。

（2）缩小变量的作用范围

关于变量，尽能定义在方法内，循环体内，try…catch块内，其目的是加快GC的回收。

（3）重写Exception的fillInStackTrace方法



第1部分：Java基础

# 标识符的命名和赋值

（1）包名全小写，如com.shutools.pro

（2）类名首字母大写，如Student

（3）常量全部大写并用下划线分隔，如STU\_STATES

（4）变量采用驼峰命名法，如stuName

（5）如果字母和数字必须混合使用，字母“l”务必大写，字母“O”要注释

# 少用静态导入

可能导致如下问题：

（1）难以阅读

（2）本类中的变量和方法覆盖了静态导入的方法和变量，导致执行的是本类的代码。（原因——编译器有一个“最短路径”原则：如果能够在本类中查找到的变量、方法，就不会到其他包或父类、接口中查找）

# 序列化中final变量的赋值

反序列化时final变量在以下情况不会被重新赋值

（1）通过构造函数为final变量赋值

（2）通过set方法为final变量赋值

（3）final修饰的属性不是基本类型

# Instanceof操作符

（1）左右操作数必须有继承或实现关系，否则编译失败。

（2）左操作数是null，则结果直接返回false。

如null instanceof XX、 (String)null instanceof XX，结果均是false

# 常量类部署的问题

发布应用系统时禁止使用类文件替换方式，整体war包发布才是万全之策。

<http://hussion.iteye.com/blog/1322308>

# equals

（1）避免null值

|  |  |
| --- | --- |
| 方式一（推荐使用）：  public void getXX(String name) {  if ("value".equals(name)) {  }  } | 方式二：  public void getXX(String name) {  if (name.equals("value")) {  }  } |

推荐方式一，避免name变量为null时，出现空指针异常

（2）覆盖equals方法必须覆盖hashCode方法

# 判断奇偶

使用num%2==0

如果使用num%2==1，当num为负奇数数时，Java会返回-1，则被当做偶数

# 包装类的知识

（1）基本类型优先考虑，避免带来developer理解错误

（2）自动拆装箱只有在赋值的时候才发生

第2部分：面向对象

# 静态

（1）静态变量一定要先声明后赋值

（2）不要重写（覆盖）静态方法

# 构造函数

（1）构造函数尽量只初始化成员变量，避免在构造函数中初始化其他类

（2）使用构造代码块精炼程序

构造代码块：在类中没有任何的前缀或后缀，并使用 {} 括起来的代码片段

# 工具类

（1）不允许实例化，不仅构造私有，还有抛异常。

因为反射机制还是可以实现私有构造的实例化

|  |  |
| --- | --- |
| 方式一：  private XxUtil{  } | 方式二（推荐使用）：  private XxUtil{  throw new Error(“不要实例化我！”);  } |

# 推荐覆盖toString方法

Debug阶段方法输出对象时可以看见类的具体值。

# 正确使用String、StringBuffer、StringBuilder

（1）String类的使用场景

在字符串不经常变化的场景中可以使用String类，例如常量的声明、少量的变量运算等。

（2）StringBuffer类的使用场景

在频繁进行字符串的运算（如拼接、替换、删除等），并且运行在多线程的环境中，例如Xml解析、Http参数解析和封装等。

（3）StringBuilder类的使用场景

在频繁进行字符串的运算（如拼接、替换、删除等），并且运行在单线程的环境中，例如SQL语句拼装、Json封装等。

# 自由选择字符串拼接方法

（1）“+”方法拼接

String str += “c”;

编译器对其做了优化，String str = new StringBuilder(str).append(“c”).toString()

（2）concat()方法拼接

String str = str. concat( “c”);

原理是数组拷贝

（3）append()方法拼接

StringBuilder sb = new StringBuilder();

sb.append(“c”);

String str = sb.toString();

优先采用方法1（”+”方法）,如果使用到循环时采用方法3（append()方法）

# 最值算法

查看建议64

第3部分：集合和泛型

# 集合分类

|  |  |
| --- | --- |
| List | ArrayList |
| LinkedList |
| Vector线程安全的动态数组 |
| Stack |
| Set | HashSet |
| TreeSet |
| 非排序Map | HashMap |
| HashTable |
| Properties是HashTable的子类，其主要用途是从Property文件中加载数据，并提供方便的读写操作 |
| 排序Map | TreeMap |

# Java泛型是类型擦除

|  |
| --- |
| public class ListDemo {  public void method1(List<String> strList) {    }    public void method1(List<Integer> intList) {    }  } |

编译不通过。泛型在编译后都会被清除掉

（1）List<String>、 List<Integer>、List<T>擦除后的类型为List

（2）List<String>[] 擦除后的类型为List []

（3）List<? extends E>、List<? super E> 擦除后的类型为List<E>

# 建议使用顺序List<T>、List<?>、List<Object>

List<T>可进行读写操作（add、remove），因为的类型是固定T类型，在编码期不需要进行任何的转型操作

List<?>是只读类型，但可以删除元素。

因为编译器不知道List中容纳的是什么类型的元素，也就无法校验类型是否安全，而且List<?>读取的元素是Object类型，需要主动转型。

第4部分：注解和反射

# forName动态加载类文件

动态加载指在程序运行时加载需要的类库文件。

forname只是把一个类加载到内存中，并不保证由此产生一个实例对象，也不会执行任何方法，但由于Java类的加载机制，会执行static代码块

forname不适用加载数组，因为只是加载数组类，没有给长度

反射操作数组使用Array类，不要采用通过的反射处理。

# 静态

对于反射效率问题，不要做任何的提前优化和预期，很少有项目是因为反射问题引起系统效率故障的

第5部分：多线程

# 静态

# 静态

# 静态

# 静态

# 静态

第6部分：开源世界

很难想象一个项目不使用开源产品的情形，

所有的框架都自己写，所有的工具类都自己堆砌，所有的运行容器都自己建立，这不是一个健康的项目，

# 采用开源工具

开源工具的好处：提升代码质量，减少bug产生，降低工作量，但一旦项目中的工具杂乱无章时就会产生依赖的无序性，这会导致代码的中隐藏着炸弹。

选择原则：

1. 普适性原则
2. 唯一性原则

相同的工具只选择一种，不要让多种相同或相似职能的工具共存。

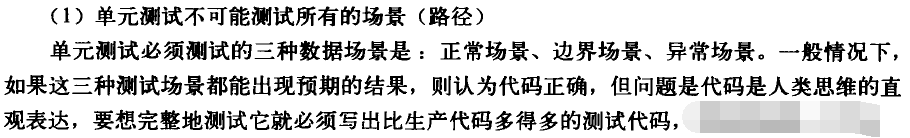
1. “大树纳凉”原则

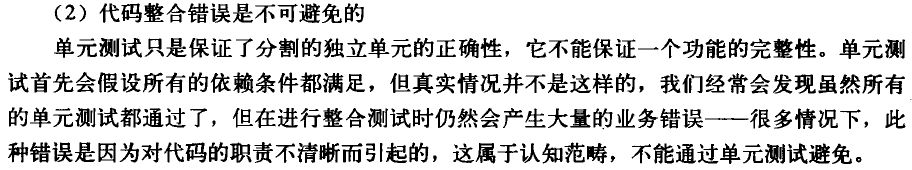
选择工具包时应寻找比较有名的开源组织，如Apache、Spring、Google

推荐：Apache Commons的BeanUtils、Lang

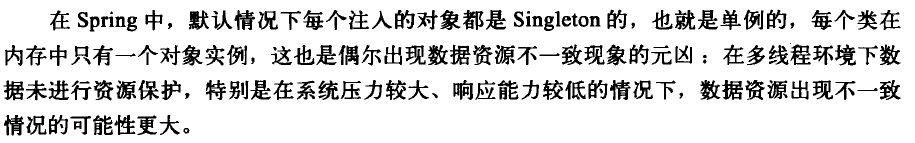
# 不要完全依靠单位测试来发现问题

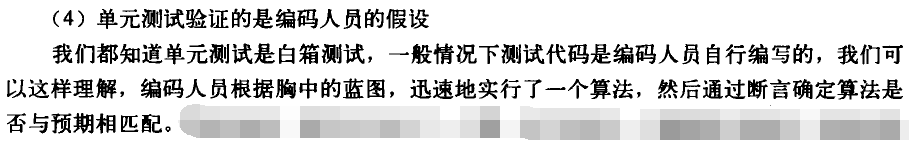
单位测试的目的是保证各个独立分割的程序单元的正确性，但不能保证代码中的所有错误都能被单元测试挖掘出来，原因有以下4点。



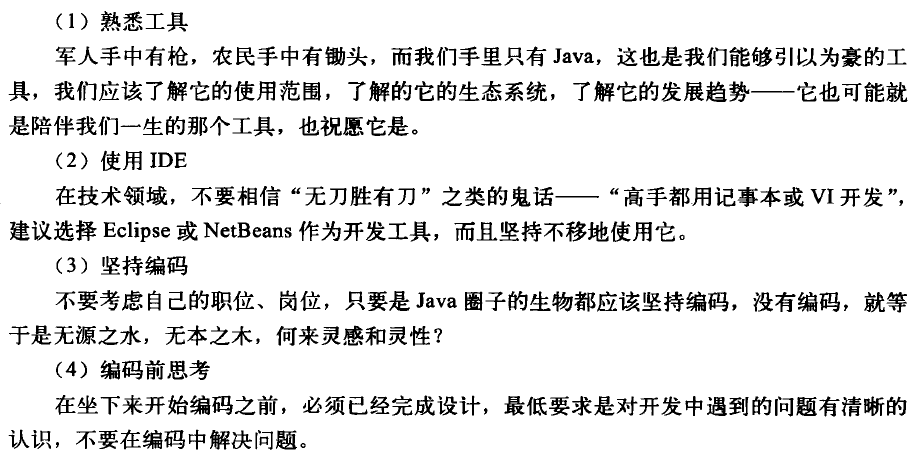


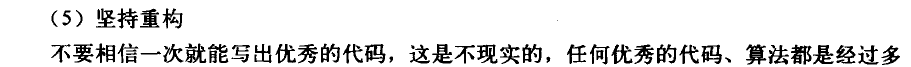
（3）部分代码无法（或很难）测试

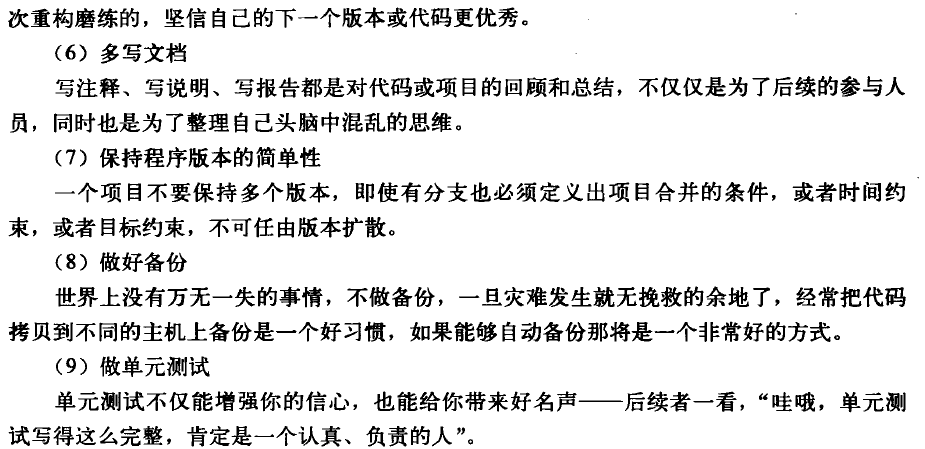


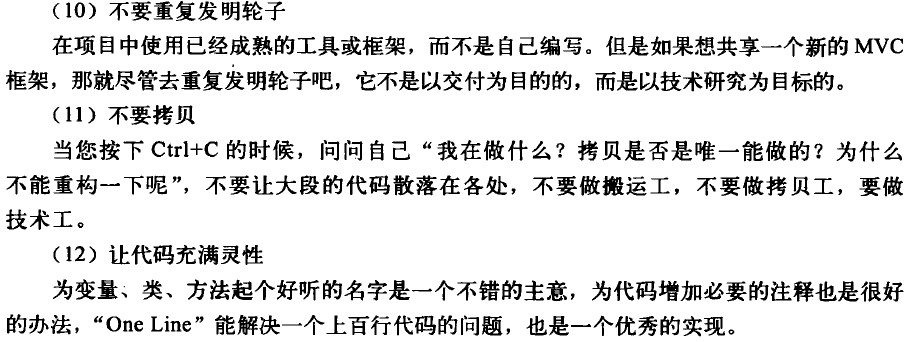


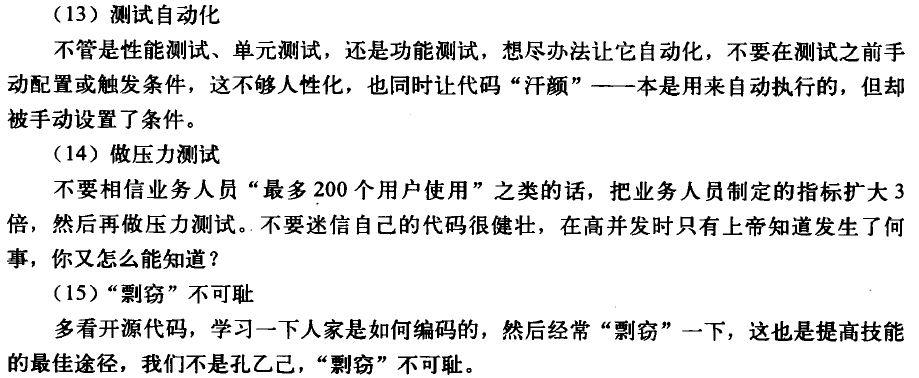
# 以技术员自律而不是工人

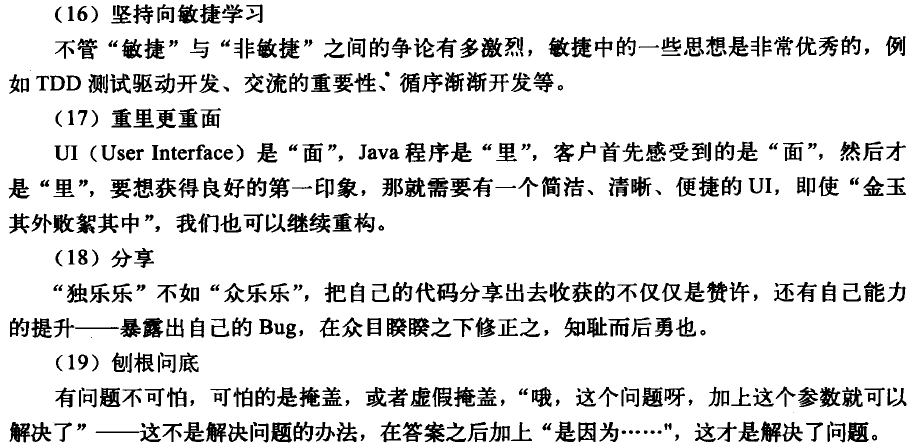


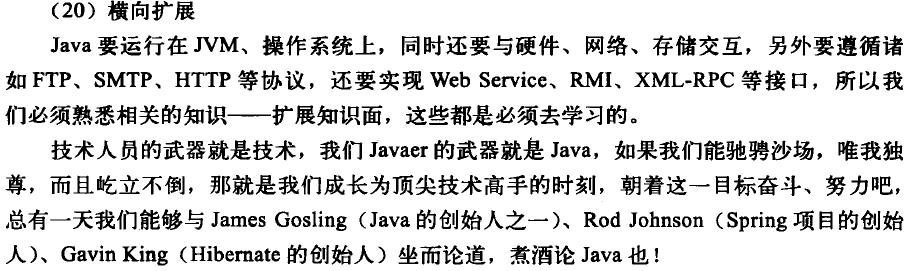












# 静态