[**Java编程常见问题汇总**](http://www.codeceo.com/article/java-programming-tips.html)

2016-06-27    分类：[JAVA开发](http://www.codeceo.com/article/category/develop/java)、[推荐阅读](http://www.codeceo.com/article/category/recommanded)、[编程开发](http://www.codeceo.com/article/category/develop)、[首页精华](http://www.codeceo.com/article/category/pick)[暂无人评论](http://www.codeceo.com/article/java-programming-tips.html#comments)     来源：[macrochen](http://www.codeceo.com/article/java-programming-tips.html)

分享到：更多10

每天在写Java程序，其实里面有一些细节大家可能没怎么注意，这不，有人总结了一个我们编程中常见的问题。虽然一般没有什么大问题，但是最好别这样做。另外这里提到的很多问题其实可以通过Findbugs( http://findbugs.sourceforge.net/ )来帮我们进行检查出来。

**字符串连接误用**

错误的写法：

String s = "";

for (Person p : persons) {

s += ", " + p.getName();

}

s = s.substring(2); //remove first comma

正确的写法：

StringBuilder sb = new StringBuilder(persons.size() \* 16); // well estimated buffer

for (Person p : persons) {

if (sb.length() > 0) sb.append(", ");

sb.append(p.getName);

}

**错误的使用StringBuffer**

错误的写法：

StringBuffer sb = new StringBuffer();

sb.append("Name: ");

sb.append(name + '\n');

sb.append("!");

...

String s = sb.toString();

问题在第三行，append char比String性能要好，另外就是初始化StringBuffer没有指定size，导致中间append时可能重新调整内部数组大小。如果是JDK1.5最好用StringBuilder取代StringBuffer，除非有线程安全的要求。还有一种方式就是可以直接连接字符串。缺点就是无法初始化时指定长度。

正确的写法：

StringBuilder sb = new StringBuilder(100);

sb.append("Name: ");

sb.append(name);

sb.append("\n!");

String s = sb.toString();

或者这样写：

String s = "Name: " + name + "\n!";

**测试字符串相等性**

错误的写法：

if (name.compareTo("John") == 0) ...

if (name == "John") ...

if (name.equals("John")) ...

if ("".equals(name)) ...

上面的代码没有错，但是不够好。compareTo不够简洁，==原义是比较两个对象是否一样。另外比较字符是否为空，最好判断它的长度。

正确的写法：

if ("John".equals(name)) ...

if (name.length() == 0) ...

if (name.isEmpty()) ...

**数字转换成字符串**

错误的写法：

"" + set.size()

new Integer(set.size()).toString()

正确的写法：

String.valueOf(set.size())

**利用不可变对象(Immutable)**

错误的写法：

zero = new Integer(0);

return Boolean.valueOf("true");

正确的写法：

zero = Integer.valueOf(0);

return Boolean.TRUE;

**请使用XML解析器**

错误的写法：

int start = xml.indexOf("<name>") + "<name>".length();

int end = xml.indexOf("</name>");

String name = xml.substring(start, end);

正确的写法：

SAXBuilder builder = new SAXBuilder(false);

Document doc = doc = builder.build(new StringReader(xml));

String name = doc.getRootElement().getChild("name").getText();

**请使用JDom组装XML**

错误的写法：

String name = ...

String attribute = ...

String xml = "<root>"

+"<name att=\""+ attribute +"\">"+ name +"</name>"

+"</root>";

正确的写法：

Element root = new Element("root");

root.setAttribute("att", attribute);

root.setText(name);

Document doc = new Documet();

doc.setRootElement(root);

XmlOutputter out = new XmlOutputter(Format.getPrettyFormat());

String xml = out.outputString(root);

**XML编码陷阱**

错误的写法：

String xml = FileUtils.readTextFile("my.xml");

因为xml的编码在文件中指定的，而在读文件的时候必须指定编码。另外一个问题不能一次就将一个xml文件用String保存，这样对内存会造成不必要的浪费，正确的做法用InputStream来边读取边处理。为了解决编码的问题, 最好使用XML解析器来处理。

**未指定字符编码**

错误的写法：

Reader r = new FileReader(file);

Writer w = new FileWriter(file);

Reader r = new InputStreamReader(inputStream);

Writer w = new OutputStreamWriter(outputStream);

String s = new String(byteArray); // byteArray is a byte[]

byte[] a = string.getBytes();

这样的代码主要不具有跨平台可移植性。因为不同的平台可能使用的是不同的默认字符编码。

正确的写法：

Reader r = new InputStreamReader(new FileInputStream(file), "ISO-8859-1");

Writer w = new OutputStreamWriter(new FileOutputStream(file), "ISO-8859-1");

Reader r = new InputStreamReader(inputStream, "UTF-8");

Writer w = new OutputStreamWriter(outputStream, "UTF-8");

String s = new String(byteArray, "ASCII");

byte[] a = string.getBytes("ASCII");

**未对数据流进行缓存**

错误的写法：

InputStream in = new FileInputStream(file);

int b;

while ((b = in.read()) != -1) {

...

}

上面的代码是一个byte一个byte的读取，导致频繁的本地JNI文件系统访问，非常低效，因为调用本地方法是非常耗时的。最好用BufferedInputStream包装一下。曾经做过一个测试，从/dev/zero下读取1MB，大概花了1s，而用BufferedInputStream包装之后只需要60ms，性能提高了94%! 这个也适用于output stream操作以及socket操作。

正确的写法：

InputStream in = new BufferedInputStream(new FileInputStream(file));

**无限使用heap内存**

错误的写法：

byte[] pdf = toPdf(file);

这里有一个前提，就是文件大小不能讲JVM的heap撑爆。否则就等着OOM吧，尤其是在高并发的服务器端代码。最好的做法是采用Stream的方式边读取边存储(本地文件或database)。

正确的写法：

File pdf = toPdf(file);

另外，对于服务器端代码来说，为了系统的安全，至少需要对文件的大小进行限制。

**不指定超时时间**

错误的代码：

Socket socket = ...

socket.connect(remote);

InputStream in = socket.getInputStream();

int i = in.read();

这种情况在工作中已经碰到不止一次了。个人经验一般超时不要超过20s。这里有一个问题，connect可以指定超时时间，但是read无法指定超时时间。但是可以设置阻塞(block)时间。

正确的写法：

Socket socket = ...

socket.connect(remote, 20000); // fail after 20s

InputStream in = socket.getInputStream();

socket.setSoTimeout(15000);

int i = in.read();

另外，文件的读取(FileInputStream, FileChannel, FileDescriptor, File)没法指定超时时间, 而且IO操作均涉及到本地方法调用, 这个更操作了JVM的控制范围，在分布式文件系统中，对IO的操作内部实际上是网络调用。一般情况下操作60s的操作都可以认为已经超时了。为了解决这些问题，一般采用缓存和异步/消息队列处理。

**频繁使用计时器**

错误代码：

for (...) {

long t = System.currentTimeMillis();

long t = System.nanoTime();

Date d = new Date();

Calendar c = new GregorianCalendar();

}

每次new一个Date或Calendar都会涉及一次本地调用来获取当前时间(尽管这个本地调用相对其他本地方法调用要快)。  
如果对时间不是特别敏感，这里使用了clone方法来新建一个Date实例。这样相对直接new要高效一些。

正确的写法：

Date d = new Date();

for (E entity : entities) {

entity.doSomething();

entity.setUpdated((Date) d.clone());

}

如果循环操作耗时较长(超过几ms)，那么可以采用下面的方法，立即创建一个Timer，然后定期根据当前时间更新时间戳，在我的系统上比直接new一个时间对象快200倍：

private volatile long time;

Timer timer = new Timer(true);

try {

time = System.currentTimeMillis();

timer.scheduleAtFixedRate(new TimerTask() {

public void run() {

time = System.currentTimeMillis();

}

}, 0L, 10L); // granularity 10ms

for (E entity : entities) {

entity.doSomething();

entity.setUpdated(new Date(time));

}

} finally {

timer.cancel();

}

**捕获所有的异常**

错误的写法：

Query q = ...

Person p;

try {

p = (Person) q.getSingleResult();

} catch(Exception e) {

p = null;

}

这是EJB3的一个查询操作，可能出现异常的原因是：结果不唯一；没有结果；数据库无法访问，而捕获所有的异常，设置为null将掩盖各种异常情况。

正确的写法：

Query q = ...

Person p;

try {

p = (Person) q.getSingleResult();

} catch(NoResultException e) {

p = null;

}

**忽略所有异常**

错误的写法：

try {

doStuff();

} catch(Exception e) {

log.fatal("Could not do stuff");

}

doMoreStuff();

这个代码有两个问题, 一个是没有告诉调用者, 系统调用出错了. 第二个是日志没有出错原因, 很难跟踪定位问题。

正确的写法：

try {

doStuff();

} catch(Exception e) {

throw new MyRuntimeException("Could not do stuff because: "+ e.getMessage, e);

}

**重复包装RuntimeException**

错误的写法：

try {

doStuff();

} catch(Exception e) {

throw new RuntimeException(e);

}

正确的写法：

try {

doStuff();

} catch(RuntimeException e) {

throw e;

} catch(Exception e) {

throw new RuntimeException(e.getMessage(), e);

}

try {

doStuff();

} catch(IOException e) {

throw new RuntimeException(e.getMessage(), e);

} catch(NamingException e) {

throw new RuntimeException(e.getMessage(), e);

}

**不正确的传播异常**

错误的写法：

try {

} catch(ParseException e) {

throw new RuntimeException();

throw new RuntimeException(e.toString());

throw new RuntimeException(e.getMessage());

throw new RuntimeException(e);

}

主要是没有正确的将内部的错误信息传递给调用者. 第一个完全丢掉了内部错误信息, 第二个错误信息依赖toString方法, 如果没有包含最终的嵌套错误信息, 也会出现丢失, 而且可读性差. 第三个稍微好一些, 第四个跟第二个一样。

正确的写法：

try {

} catch(ParseException e) {

throw new RuntimeException(e.getMessage(), e);

}

**用日志记录异常**

错误的写法：

try {

...

} catch(ExceptionA e) {

log.error(e.getMessage(), e);

throw e;

} catch(ExceptionB e) {

log.error(e.getMessage(), e);

throw e;

}

一般情况下在日志中记录异常是不必要的, 除非调用方没有记录日志。

**异常处理不彻底**

错误的写法：

try {

is = new FileInputStream(inFile);

os = new FileOutputStream(outFile);

} finally {

try {

is.close();

os.close();

} catch(IOException e) {

/\* we can't do anything \*/

}

}

is可能close失败, 导致os没有close

正确的写法：

try {

is = new FileInputStream(inFile);

os = new FileOutputStream(outFile);

} finally {

try { if (is != null) is.close(); } catch(IOException e) {/\* we can't do anything \*/}

try { if (os != null) os.close(); } catch(IOException e) {/\* we can't do anything \*/}

}

**捕获不可能出现的异常**

错误的写法：

try {

... do risky stuff ...

} catch(SomeException e) {

// never happens

}

... do some more ...

正确的写法：

try {

... do risky stuff ...

} catch(SomeException e) {

// never happens hopefully

throw new IllegalStateException(e.getMessage(), e); // crash early, passing all information

}

... do some more ...

**transient的误用**

错误的写法：

public class A implements Serializable {

private String someState;

private transient Log log = LogFactory.getLog(getClass());

public void f() {

log.debug("enter f");

...

}

}

这里的本意是不希望Log对象被序列化. 不过这里在反序列化时, 会因为log未初始化, 导致f()方法抛空指针, 正确的做法是将log定义为静态变量或者定位为具备变量。

正确的写法：

public class A implements Serializable {

private String someState;

private static final Log log = LogFactory.getLog(A.class);

public void f() {

log.debug("enter f");

...

}

}

public class A implements Serializable {

private String someState;

public void f() {

Log log = LogFactory.getLog(getClass());

log.debug("enter f");

...

}

}

**不必要的初始化**

错误的写法：

public class B {

private int count = 0;

private String name = null;

private boolean important = false;

}

这里的变量会在初始化时使用默认值:0, null, false, 因此上面的写法有些多此一举。

正确的写法：

public class B {

private int count;

private String name;

private boolean important;

}

**最好用静态final定义Log变量**

private static final Log log = LogFactory.getLog(MyClass.class);

这样做的好处有三：

* 可以保证线程安全
* 静态或非静态代码都可用
* 不会影响对象序列化

**选择错误的类加载器**

错误的代码：

Class clazz = Class.forName(name);

Class clazz = getClass().getClassLoader().loadClass(name);

这里本意是希望用当前类来加载希望的对象, 但是这里的getClass()可能抛出异常, 特别在一些受管理的环境中, 比如应用服务器, web容器, Java WebStart环境中, 最好的做法是使用当前应用上下文的类加载器来加载。

正确的写法：

ClassLoader cl = Thread.currentThread().getContextClassLoader();

if (cl == null) cl = MyClass.class.getClassLoader(); // fallback

Class clazz = cl.loadClass(name);

**反射使用不当**

错误的写法：

Class beanClass = ...

if (beanClass.newInstance() instanceof TestBean) ...

这里的本意是检查beanClass是否是TestBean或是其子类, 但是创建一个类实例可能没那么简单, 首先实例化一个对象会带来一定的消耗, 另外有可能类没有定义默认构造函数. 正确的做法是用Class.isAssignableFrom(Class) 方法。

正确的写法：

Class beanClass = ...

if (TestBean.class.isAssignableFrom(beanClass)) ...

**不必要的同步**

错误的写法：

Collection l = new Vector();

for (...) {

l.add(object);

}

Vector是ArrayList同步版本。

正确的写法：

Collection l = new ArrayList();

for (...) {

l.add(object);

}

**错误的选择List类型**

根据下面的表格数据来进行选择

|  |  |  |
| --- | --- | --- |
|  | ArrayList | LinkedList |
| add (append) | O(1) or ~O(log(n)) if growing | O(1) |
| insert (middle) | O(n) or ~O(n\*log(n)) if growing | O(n) |
| remove (middle) | O(n) (always performs complete copy) | O(n) |
| iterate | O(n) | O(n) |
| get by index | O(1) | O(n) |

**HashMap size陷阱**

错误的写法：

Map map = new HashMap(collection.size());

for (Object o : collection) {

  map.put(o.key, o.value);

}

这里可以参考guava的Maps.newHashMapWithExpectedSize的实现. 用户的本意是希望给HashMap设置初始值, 避免扩容(resize)的开销. 但是没有考虑当添加的元素数量达到HashMap容量的75%时将出现resize。

正确的写法：

Map map = new HashMap(1 + (int) (collection.size() / 0.75));

**对Hashtable, HashMap 和 HashSet了解不够**

这里主要需要了解HashMap和Hashtable的内部实现上, 它们都使用Entry包装来封装key/value, Entry内部除了要保存Key/Value的引用, 还需要保存hash桶中next Entry的应用, 因此对内存会有不小的开销, 而HashSet内部实现其实就是一个HashMap. 有时候IdentityHashMap可以作为一个不错的替代方案. 它在内存使用上更有效(没有用Entry封装, 内部采用Object[]). 不过需要小心使用. 它的实现违背了Map接口的定义. 有时候也可以用ArrayList来替换HashSet.

这一切的根源都是由于JDK内部没有提供一套高效的Map和Set实现。

**对List的误用**

建议下列场景用Array来替代List:

* list长度固定，比如一周中的每一天
* 对list频繁的遍历，比如超过1w次
* 需要对数字进行包装(主要JDK没有提供基本类型的List)

比如下面的代码。

错误的写法：

List<Integer> codes = new ArrayList<Integer>();

codes.add(Integer.valueOf(10));

codes.add(Integer.valueOf(20));

codes.add(Integer.valueOf(30));

codes.add(Integer.valueOf(40));

正确的写法：

int[] codes = { 10, 20, 30, 40 };

错误的写法：

// horribly slow and a memory waster if l has a few thousand elements (try it yourself!)

List<Mergeable> l = ...;

for (int i=0; i < l.size()-1; i++) {

    Mergeable one = l.get(i);

    Iterator<Mergeable> j = l.iterator(i+1); // memory allocation!

    while (j.hasNext()) {

        Mergeable other = l.next();

        if (one.canMergeWith(other)) {

            one.merge(other);

            other.remove();

        }

    }

}

正确的写法：

// quite fast and no memory allocation

Mergeable[] l = ...;

for (int i=0; i < l.length-1; i++) {

    Mergeable one = l[i];

    for (int j=i+1; j < l.length; j++) {

        Mergeable other = l[j];

        if (one.canMergeWith(other)) {

            one.merge(other);

            l[j] = null;

        }

    }

}

实际上Sun也意识到这一点, 因此在JDK中, Collections.sort()就是将一个List拷贝到一个数组中然后调用Arrays.sort方法来执行排序。

**用数组来描述一个结构**

错误用法：

/\*\*

\* @returns [1]: Location, [2]: Customer, [3]: Incident

\*/

Object[] getDetails(int id) {...

这里用数组+文档的方式来描述一个方法的返回值. 虽然很简单, 但是很容易误用, 正确的做法应该是定义个类。

正确的写法：

Details getDetails(int id) {...}

private class Details {

public Location location;

public Customer customer;

public Incident incident;

}

**对方法过度限制**

错误用法：

public void notify(Person p) {

...

sendMail(p.getName(), p.getFirstName(), p.getEmail());

...

}

class PhoneBook {

String lookup(String employeeId) {

Employee emp = ...

return emp.getPhone();

}

}

第一个例子是对方法参数做了过多的限制, 第二个例子对方法的返回值做了太多的限制。

正确的写法：

public void notify(Person p) {

...

sendMail(p);

...

}

class EmployeeDirectory {

Employee lookup(String employeeId) {

Employee emp = ...

return emp;

}

}

**对POJO的setter方法画蛇添足**

错误的写法：

private String name;

public void setName(String name) {

this.name = name.trim();

}

public void String getName() {

return this.name;

}

有时候我们很讨厌字符串首尾出现空格, 所以在setter方法中进行了trim处理, 但是这样做的结果带来的副作用会使getter方法的返回值和setter方法不一致, 如果只是将JavaBean当做一个数据容器, 那么最好不要包含任何业务逻辑. 而将业务逻辑放到专门的业务层或者控制层中处理。

正确的做法：

person.setName(textInput.getText().trim());

**日历对象(Calendar)误用**

错误的写法：

Calendar cal = new GregorianCalender(TimeZone.getTimeZone("Europe/Zurich"));

cal.setTime(date);

cal.add(Calendar.HOUR\_OF\_DAY, 8);

date = cal.getTime();

这里主要是对date, time, calendar和time zone不了解导致. 而在一个时间上增加8小时, 跟time zone没有任何关系, 所以没有必要使用Calendar, 直接用Date对象即可, 而如果是增加天数的话, 则需要使用Calendar, 因为采用不同的时令制可能一天的小时数是不同的(比如有些DST是23或者25个小时)

正确的写法：

date = new Date(date.getTime() + 8L \* 3600L \* 1000L); // add 8 hrs

**TimeZone的误用**

错误的写法：

Calendar cal = new GregorianCalendar();

cal.setTime(date);

cal.set(Calendar.HOUR\_OF\_DAY, 0);

cal.set(Calendar.MINUTE, 0);

cal.set(Calendar.SECOND, 0);

Date startOfDay = cal.getTime();

这里有两个错误, 一个是没有没有将毫秒归零, 不过最大的错误是没有指定TimeZone, 不过一般的桌面应用没有问题, 但是如果是服务器端应用则会有一些问题, 比如同一时刻在上海和伦敦就不一样, 因此需要指定的TimeZone.

正确的写法：

Calendar cal = new GregorianCalendar(user.getTimeZone());

cal.setTime(date);

cal.set(Calendar.HOUR\_OF\_DAY, 0);

cal.set(Calendar.MINUTE, 0);

cal.set(Calendar.SECOND, 0);

cal.set(Calendar.MILLISECOND, 0);

Date startOfDay = cal.getTime();

**时区(Time Zone)调整的误用**

错误的写法：

public static Date convertTz(Date date, TimeZone tz) {

Calendar cal = Calendar.getInstance();

cal.setTimeZone(TimeZone.getTimeZone("UTC"));

cal.setTime(date);

cal.setTimeZone(tz);

return cal.getTime();

}

这个方法实际上没有改变时间, 输入和输出是一样的. 关于时间的问题可以参考这篇文章: http://www.odi.ch/prog/design/datetime.php 这里主要的问题是Date对象并不包含Time Zone信息. 它总是使用UTC(世界统一时间). 而调用Calendar的getTime/setTime方法会自动在当前时区和UTC之间做转换。

**Calendar.getInstance()的误用**

错误的写法：

Calendar c = Calendar.getInstance();

c.set(2009, Calendar.JANUARY, 15);

Calendar.getInstance()依赖local来选择一个Calendar实现, 不同实现的2009年是不同的, 比如有些Calendar实现就没有January月份。

正确的写法：

Calendar c = new GregorianCalendar(timeZone);

c.set(2009, Calendar.JANUARY, 15);

**Date.setTime()的误用**

错误的写法：

account.changePassword(oldPass, newPass);

Date lastmod = account.getLastModified();

lastmod.setTime(System.currentTimeMillis());

在更新密码之后, 修改一下最后更新时间, 这里的用法没有错,但是有更好的做法: 直接传Date对象. 因为Date是Value Object, 不可变的. 如果更新了Date的值, 实际上是生成一个新的Date实例. 这样其他地方用到的实际上不在是原来的对象, 这样可能出现不可预知的异常. 当然这里又涉及到另外一个OO设计的问题, 对外暴露Date实例本身就是不好的做法(一般的做法是在setter方法中设置Date引用参数的clone对象). 另外一种比较好的做法就是直接保存long类型的毫秒数。

正确的做法：

account.changePassword(oldPass, newPass);

account.setLastModified(new Date());

**SimpleDateFormat非线程安全误用**

错误的写法：

public class Constants {

public static final SimpleDateFormat date = new SimpleDateFormat("dd.MM.yyyy");

}

SimpleDateFormat不是线程安全的. 在多线程并行处理的情况下, 会得到非预期的值. 这个错误非常普遍! 如果真要在多线程环境下公用同一个SimpleDateFormat, 那么做好做好同步(cache flush, lock contention), 但是这样会搞得更复杂, 还不如直接new一个实在。

**使用全局参数配置常量类/接口**

public interface Constants {

String version = "1.0";

String dateFormat = "dd.MM.yyyy";

String configFile = ".apprc";

int maxNameLength = 32;

String someQuery = "SELECT \* FROM ...";

}

很多应用都会定义这样一个全局常量类或接口, 但是为什么这种做法不推荐? 因为这些常量之间基本没有任何关联, 只是因为公用才定义在一起. 但是如果其他组件需要使用这些全局变量, 则必须对该常量类产生依赖, 特别是存在server和远程client调用的场景。

比较好的做法是将这些常量定义在组件内部. 或者局限在一个类库内部。

**忽略造型溢出(cast overflow)**

错误的写法：

public int getFileSize(File f) {

long l = f.length();

return (int) l;

}

这个方法的本意是不支持传递超过2GB的文件. 最好的做法是对长度进行检查, 溢出时抛出异常。

正确的写法：

public int getFileSize(File f) {

long l = f.length();

if (l > Integer.MAX\_VALUE) throw new IllegalStateException("int overflow");

return (int) l;

}

另一个溢出bug是cast的对象不对, 比如下面第一个println. 正确的应该是下面的那个。

long a = System.currentTimeMillis();

long b = a + 100;

System.out.println((int) b-a);

System.out.println((int) (b-a));

**对float和double使用==操作**

错误的写法：

for (float f = 10f; f!=0; f-=0.1) {

System.out.println(f);

}

上面的[浮点数](http://www.codeceo.com/article/float-number.html)递减只会无限接近0而不会等于0, 这样会导致上面的for进入死循环. 通常绝不要对float和double使用==操作. 而采用大于和小于操作. 如果java编译器能针对这种情况给出警告. 或者在java语言规范中不支持浮点数类型的==操作就最好了。

正确的写法：

for (float f = 10f; f>0; f-=0.1) {

System.out.println(f);

}

**用浮点数来保存money**

错误的写法：

float total = 0.0f;

for (OrderLine line : lines) {

total += line.price \* line.count;

}

double a = 1.14 \* 75; // 85.5 将表示为 85.4999...

System.out.println(Math.round(a)); // 输出值为85

BigDecimal d = new BigDecimal(1.14); //造成精度丢失

这个也是一个老生常谈的错误. 比如计算100笔订单, 每笔0.3元, 最终的计算结果是29.9999971. 如果将float类型改为double类型, 得到的结果将是30.000001192092896. 出现这种情况的原因是, 人类和计算的计数方式不同. 人类采用的是十进制, 而计算机是二进制.二进制对于计算机来说非常好使, 但是对于涉及到精确计算的场景就会带来误差. 比如银行金融中的应用。

因此绝不要用浮点类型来保存money数据. 采用浮点数得到的计算结果是不精确的. 即使与int类型做乘法运算也会产生一个不精确的结果.那是因为在用二进制存储一个浮点数时已经出现了精度丢失. 最好的做法就是用一个string或者固定点数来表示. 为了精确, 这种表示方式需要指定相应的精度值.

BigDecimal就满足了上面所说的需求. 如果在计算的过程中精度的丢失超出了给定的范围, 将抛出runtime exception.

正确的写法：

BigDecimal total = BigDecimal.ZERO;

for (OrderLine line : lines) {

BigDecimal price = new BigDecimal(line.price);

BigDecimal count = new BigDecimal(line.count);

total = total.add(price.multiply(count)); // BigDecimal is immutable!

}

total = total.setScale(2, RoundingMode.HALF\_UP);

BigDecimal a = (new BigDecimal("1.14")).multiply(new BigDecimal(75)); // 85.5 exact

a = a.setScale(0, RoundingMode.HALF\_UP); // 86

System.out.println(a); // correct output: 86

BigDecimal a = new BigDecimal("1.14");

**不使用finally块释放资源**

错误的写法：

public void save(File f) throws IOException {

OutputStream out = new BufferedOutputStream(new FileOutputStream(f));

out.write(...);

out.close();

}

public void load(File f) throws IOException {

InputStream in = new BufferedInputStream(new FileInputStream(f));

in.read(...);

in.close();

}

上面的代码打开一个文件输出流, 操作系统为其分配一个文件句柄, 但是文件句柄是一种非常稀缺的资源, 必须通过调用相应的close方法来被正确的释放回收. 而为了保证在异常情况下资源依然能被正确回收, 必须将其放在finally block中. 上面的代码中使用了BufferedInputStream将file stream包装成了一个buffer stream, 这样将导致在调用close方法时才会将buffer stream写入磁盘. 如果在close的时候失败, 将导致写入数据不完全. 而对于FileInputStream在finally block的close操作这里将直接忽略。

如果BufferedOutputStream.close()方法执行顺利则万事大吉, 如果失败这里有一个潜在的bug(http://bugs.sun.com/view\_bug.do?bug\_id=6335274): 在close方法内部调用flush操作的时候, 如果出现异常, 将直接忽略. 因此为了尽量减少数据丢失, 在执行close之前显式的调用flush操作。

下面的代码有一个小小的瑕疵: 如果分配file stream成功, 但是分配buffer stream失败(OOM这种场景), 将导致文件句柄未被正确释放. 不过这种情况一般不用担心, 因为JVM的gc将帮助我们做清理。

// code for your cookbook

public void save() throws IOException {

File f = ...

OutputStream out = new BufferedOutputStream(new FileOutputStream(f));

try {

out.write(...);

out.flush(); // don't lose exception by implicit flush on close

} finally {

out.close();

}

}

public void load(File f) throws IOException {

InputStream in = new BufferedInputStream(new FileInputStream(f));

try {

in.read(...);

} finally {

try { in.close(); } catch (IOException e) { }

}

}

数据库访问也涉及到类似的情况：

Car getCar(DataSource ds, String plate) throws SQLException {

Car car = null;

Connection c = null;

PreparedStatement s = null;

ResultSet rs = null;

try {

c = ds.getConnection();

s = c.prepareStatement("select make, color from cars where plate=?");

s.setString(1, plate);

rs = s.executeQuery();

if (rs.next()) {

car = new Car();

car.make = rs.getString(1);

car.color = rs.getString(2);

}

} finally {

if (rs != null) try { rs.close(); } catch (SQLException e) { }

if (s != null) try { s.close(); } catch (SQLException e) { }

if (c != null) try { c.close(); } catch (SQLException e) { }

}

return car;

}

**finalize方法误用**

错误的写法：

public class FileBackedCache {

private File backingStore;

...

protected void finalize() throws IOException {

if (backingStore != null) {

backingStore.close();

backingStore = null;

}

}

}

这个问题Effective Java这本书有详细的说明. 主要是finalize方法依赖于GC的调用, 其调用时机可能是立马也可能是几天以后, 所以是不可预知的. 而JDK的API文档中对这一点有误导：建议在该方法中来释放I/O资源。

正确的做法是定义一个close方法, 然后由外部的容器来负责调用释放资源。

public class FileBackedCache {

private File backingStore;

...

public void close() throws IOException {

if (backingStore != null) {

backingStore.close();

backingStore = null;

}

}

}

在JDK 1.7 (Java 7)中已经引入了一个AutoClosable接口. 当变量(不是对象)超出了try-catch的资源使用范围, 将自动调用close方法。

try (Writer w = new FileWriter(f)) { // implements Closable

w.write("abc");

// w goes out of scope here: w.close() is called automatically in ANY case

} catch (IOException e) {

throw new RuntimeException(e.getMessage(), e);

}

**Thread.interrupted方法误用**

错误的写法：

try {

Thread.sleep(1000);

} catch (InterruptedException e) {

// ok

}

or

while (true) {

if (Thread.interrupted()) break;

}

这里主要是interrupted静态方法除了返回当前线程的中断状态, 还会将当前线程状态复位。

正确的写法：

try {

Thread.sleep(1000);

} catch (InterruptedException e) {

Thread.currentThread().interrupt();

}

or

while (true) {

if (Thread.currentThread().isInterrupted()) break;

}

**在静态变量初始化时创建线程**

错误的写法：

class Cache {

private static final Timer evictor = new Timer();

}

Timer构造器内部会new一个thread, 而该thread会从它的父线程(即当前线程)中继承各种属性。比如context classloader, [ThreadLocal](http://www.codeceo.com/article/threadlocal-usage.html)以及其他的安全属性(访问权限)。 而加载当前类的线程可能是不确定的，比如一个线程池中随机的一个线程。如果你需要控制线程的属性，最好的做法就是将其初始化操作放在一个静态方法中，这样初始化将由它的调用者来决定。

正确的做法：

class Cache {

private static Timer evictor;

public static setupEvictor() {

evictor = new Timer();

}

}

**已取消的定时器任务依然持有状态**

错误的写法：

final MyClass callback = this;

TimerTask task = new TimerTask() {

public void run() {

callback.timeout();

}

};

timer.schedule(task, 300000L);

try {

doSomething();

} finally {

task.cancel();

}

上面的task内部包含一个对外部类实例的应用, 这将导致该引用可能不会被GC立即回收. 因为Timer将保留TimerTask在指定的时间之后才被释放. 因此task对应的外部类实例将在5分钟后被回收。

正确的写法：

TimerTask task = new Job(this);

timer.schedule(task, 300000L);

try {

doSomething();

} finally {

task.cancel();

}

static class Job extends TimerTask {

private MyClass callback;

public Job(MyClass callback) {

this.callback = callback;

}

public boolean cancel() {

callback = null;

return super.cancel();

}

public void run() {

if (callback == null) return;

callback.timeout();

}

}