目 录

[1 目的 5](#_Toc334002646)

[2 背景 5](#_Toc334002647)

[3 安全编码规范 5](#_Toc334002648)

[3.1 输入验证和数据合法性校验 5](#_Toc334002649)

[3.1.1 避免SQL注入 5](#_Toc334002650)

[3.1.2 避免XML注入 5](#_Toc334002651)

[3.1.3 避免跨站点脚本（XSS） 6](#_Toc334002652)

[3.2 声明和初始化 6](#_Toc334002653)

[3.2.1 避免类初始化的相互依赖 6](#_Toc334002654)

[3.3 表达式 7](#_Toc334002655)

[3.3.1 不可忽略方法的返回值 7](#_Toc334002656)

[3.3.2 不要引用空指针 8](#_Toc334002657)

[3.3.3 使用Arrays.equals（）来比较数组的内容 8](#_Toc334002658)

[3.4 数字类型和操作 8](#_Toc334002659)

[3.4.1 防止整数溢出 8](#_Toc334002660)

[3.4.2 避免除法和取模运算分母为零 9](#_Toc334002661)

[3.5 类和方法操作 10](#_Toc334002662)

[3.5.1 数据成员声明为私有，提供可访问的包装方法 10](#_Toc334002663)

[3.5.2 敏感类不允许复制 10](#_Toc334002664)

[3.5.3 比较类的正确做法 10](#_Toc334002665)

[3.5.4 不要硬编码敏感信息 11](#_Toc334002666)

[3.5.5 验证方法参数 11](#_Toc334002667)

[3.5.6 不要使用过时、陈旧或低效的方法 11](#_Toc334002668)

[3.5.7 数组引用问题 11](#_Toc334002669)

[3.5.8 不要产生内存泄露 12](#_Toc334002670)

[3.6 异常处理 12](#_Toc334002671)

[3.6.1 不要忽略捕获的异常 12](#_Toc334002672)

[3.6.2 不允许暴露异常的敏感信息 13](#_Toc334002673)

[3.6.3 不允许抛出RuntimeException, Exception,Throwable 14](#_Toc334002674)

[3.6.4 不要捕获NullPointerException或其他父类异常 14](#_Toc334002675)

[3.7 多线程编程 15](#_Toc334002676)

[3.7.1 确保共享变量的可见性 15](#_Toc334002677)

[3.7.2 确保共享变量的操作是原子的 16](#_Toc334002678)

[3.7.3 不要调用Thread.run（），不要使用Thread.stop（）以终止线程 18](#_Toc334002679)

[3.7.4 确保执行阻塞操作的线程可以终止 18](#_Toc334002680)

[3.7.5 相互依存的任务不要在一个有限的线程池执行 19](#_Toc334002681)

[3.8 输入输出 19](#_Toc334002682)

[3.8.1 程序终止前删除临时文件 19](#_Toc334002683)

[3.8.2 检测和处理文件相关的错误 19](#_Toc334002684)

[3.8.3 及时释放资源 19](#_Toc334002685)

[3.9 序列化 20](#_Toc334002686)

[3.9.1 不要序列化未加密的敏感数据 20](#_Toc334002687)

[3.9.2 在序列化过程中避免内存和资源泄漏 21](#_Toc334002688)

[3.9.3 反序列化要在程序最小权限的安全环境中 22](#_Toc334002689)

# 安全编码规范

## 输入验证和数据合法性校验

程序接受数据可能来源于未经验证的用户，网络连接和其他不受信任的来源，如果未对程序接受数据进行校验，则可能会引发安全问题。

### 避免SQL注入

使用PreparedStatement预编译SQL,解决SQL注入问题，传递给PreparedStatement对象的参数可以被强制进行类型转换，确保在插入或查询数据时与底层的数据库格式匹配。

String sqlString = "select \* from db\_user where username=? and password=?";

PreparedStatement stmt = connection.prepareStatement(sqlString);

stmt.setString(1, username);

stmt.setString(2, pwd);

ResultSet rs = stmt.executeQuery();

### 避免XML注入

通过StringBulider 或 StringBuffer 拼接XML文件时，需对输入数据进行合法性校验。

对数量quantity 进行合法性校验，控制只能传入0-9的数字：

if (!Pattern.matches("[0-9]+", quantity)) {

// Format violation

}

String xmlString = "<item>\n<description>Widget</description>\n" +

"<price>500</price>\n" +

"<quantity>" + quantity + "</quantity></item>";

outStream.write(xmlString.getBytes());

outStream.flush();

### 避免跨站点脚本（XSS）

对产生跨站的参数进行严格过滤，禁止传入<SCRIPT>标签

//定义需过滤的字段串 <script>

String s = "\uFE64" + "script" + "\uFE65";

// 过滤字符串标准化

s = Normalizer.normalize(s, Form.NFKC);

// 使用正则表达式匹配inputStr是否存在<script>

Pattern pattern = Pattern.compile(inputStr);

Matcher matcher = pattern.matcher(s);

if (matcher.find()) {

// Found black listed tag

throw new IllegalStateException();

} else {

// ...

}

## 声明和初始化

### 避免类初始化的相互依赖

例：

错误的写法：

public class Cycle {

private final int balance;

private static final Cycle c = new Cycle();

private static final int deposit = (int) (Math.random() \* 100); // Random deposit

public Cycle() {

balance = deposit - 10; // Subtract processing fee

}

public static void main(String[] args) {

System.out.println("The account balance is: " + c.balance);

}

}

类加载时初始化指向Cycle类的静态变量c，而类Cycle的无参构造方法又依赖静态变量deposit，导致无法预期的结果。

正确的写法：

public class Cycle {

private final int balance;

private static final int deposit = (int) (Math.random() \* 100); // Random deposit

private static final Cycle c = new Cycle(); // Inserted after initialization of required fields

public Cycle() {

balance = deposit - 10; // Subtract processing fee

}

public static void main(String[] args) {

System.out.println("The account balance is: " + c.balance);

}

}

## 表达式

### 不可忽略方法的返回值

忽略方法的放回值可能会导致无法预料的结果。

错误的写法：

public void deleteFile(){

File someFile = new File("someFileName.txt");

someFile.delete();

}

正确的写法：

public void deleteFile(){

File someFile = new File("someFileName.txt");

if (!someFile.delete()) {

// handle failure to delete the file

}

}

### 不要引用空指针

当一个变量指向一个NULL值，使用这个变量的时候又没有检查，这时会导致。NullPointerException。

在使用变量前一定要做是否为NULL值的校验。

### 使用Arrays.equals（）来比较数组的内容

数组没有覆盖的Object. equals()方法，调用Object. equals()方法实际上是比较数组的引用，而不是他们的内容。程序必须使用两个参数Arrays.equals（）方法来比较两个数组的内容

public void arrayEqualsExample() {

int[] arr1 = new int[20]; // initialized to 0

int[] arr2 = new int[20]; // initialized to 0

Arrays.equals(arr1, arr2); // true

}

## 数字类型和操作

### 防止整数溢出

使用[java.lang.Number](mk:@MSITStore:D:\\工作目录\\API文档\\java_api.CHM::/java/lang/Number.html" \o "java.lang 中的类). BigInteger类进行整数运算，防止整数溢出。

**public** **class** BigIntegerUtil {

**private** **static** **final** BigInteger *bigMaxInt* = BigInteger.*valueOf*(Integer.*MAX\_VALUE*);

**private** **static** **final** BigInteger *bigMinInt* = BigInteger.*valueOf*(Integer.*MIN\_VALUE*);

**public** **static** BigInteger intRangeCheck(BigInteger val) **throws** ArithmeticException {

**if** (val.compareTo(*bigMaxInt*) == 1 || val.compareTo(*bigMinInt*) == -1) {

**throw** **new** ArithmeticException("Integer overflow");

}

**return** val;

}

**public** **static** **int** addInt(**int** v1, **int** v2) **throws** ArithmeticException {

BigInteger b1 = BigInteger.*valueOf*(v1);

BigInteger b2 = BigInteger.*valueOf*(v2);

BigInteger res = *intRangeCheck*(b1.add(b2));

**return** res.intValue();

}

**public** **static** **int** subInt(**int** v1, **int** v2) **throws** ArithmeticException {

BigInteger b1 = BigInteger.*valueOf*(v1);

BigInteger b2 = BigInteger.*valueOf*(v2);

BigInteger res = *intRangeCheck*(b1.subtract(b2));

**return** res.intValue();

}

**public** **static** **int** multiplyInt(**int** v1, **int** v2) **throws** ArithmeticException {

BigInteger b1 = BigInteger.*valueOf*(v1);

BigInteger b2 = BigInteger.*valueOf*(v2);

BigInteger res = *intRangeCheck*(b1.multiply(b2));

**return** res.intValue();

}

**public** **static** **int** divideInt(**int** v1, **int** v2) **throws** ArithmeticException {

BigInteger b1 = BigInteger.*valueOf*(v1);

BigInteger b2 = BigInteger.*valueOf*(v2);

BigInteger res = *intRangeCheck*(b1.divide(b2));

**return** res.intValue();

}

}

### 避免除法和取模运算分母为零

要避免因为分母为零而导致除法和取模运算出现异常。

if (num2 == 0) {

// handle error

} else {

result1= num1 /num2;

result2= num1 % num2;

}

## 类和方法操作

### 数据成员声明为私有，提供可访问的包装方法

攻击者可以用意想不到的方式操纵public或protected的数据成员，所以需要将数据成员为private，对外提供可控的包装方法访问数据成员。

### 敏感类不允许复制

包含私人的，机密或其他敏感数据的类是不允许被复制的，解决的方法有两种：

1. 类声明为final

final class SensitiveClass {

// ...

}

1. Clone 方法抛出CloneNotSupportedException异常

class SensitiveClass {

// ...

public final SensitiveClass clone() throws CloneNotSupportedException {

throw new CloneNotSupportedException();

}

}

### 比较类的正确做法

如果由同一个类装载器装载，它们具有相同的完全限定名称，则它们是两个相同的类。

不正确写法：

// Determine whether object auth has required/expected class object

if (auth.getClass().getName().equals(

"com.application.auth.DefaultAuthenticationHandler")) {

// ...

}

正确写法：

// Determine whether object auth has required/expected class name

if (auth.getClass() == com.application.auth.DefaultAuthenticationHandler.class) {

// ...

}

### 不要硬编码敏感信息

硬编码的敏感信息，如密码，服务器IP地址和加密密钥，可能会泄露给攻击者。

敏感信息均必须存在在配置文件或数据库中。

### 验证方法参数

验证方法的参数，可确保操作方法的参数产生有效的结果。不验证方法的参数可能会导致不正确的计算，运行时异常，违反类的不变量，对象的状态不一致。

对于跨信任边界接收参数的方法，必须进行参数合法性校验

private Object myState = null;

//对于修改myState 方法的入参，进行非空和合法性校验

void setState(Object state) {

  if (state == null) {

    // Handle null state

  }

  if (isInvalidState(state)) {

    // Handle invalid state

  }

  myState = state;

}

### 不要使用过时、陈旧或低效的方法

在程序代码中使用过时的、陈旧的或低效的类或方法可能会导致错误的行为。

### 数组引用问题

某个方法返回一个对敏感对象的内部数组的引用，假定该方法的调用程序不改变这些对象。即使数组对象本身是不可改变的，也可以在数组对象以外操作数组的内容，这种操作将反映在返回该数组的对象中。如果该方法返回可改变的对象，外部实体可以改变在那个类中声明的 public 变量，这种改变将反映在实际对象中。

不正确的写法：

public class XXX {

private String[] xxxx;

public String[] getXXX() {

return xxxx;

}

}

正确的写法：

public class XXX {

private String[] xxxx;

public String[] getXXX() {

String temp[] = Arrays.copyof(…); // 或其他数组复制方法

return temp;

}

}

### 不要产生内存泄露

垃圾收集器只收集不可达的对象，因此，存在未使用的可到达的对象，仍然表示内存管理不善。过度的内存泄漏可能会导致内存耗尽，拒绝服务（DoS）。

## 异常处理

### 不要忽略捕获的异常

对于捕获的异常要进行相应的处理，不能忽略已捕获的异常

不正确写法：

class Foo implements Runnable {

public void run() {

try {

Thread.sleep(1000);

} catch (InterruptedException e) {

// 此处InterruptedException被忽略

}

}

}

正确写法：

class Foo implements Runnable {

public void run() {

try {

Thread.sleep(1000);

} catch (InterruptedException e) {

Thread.currentThread().interrupt(); // Reset interrupted status

}

}

}

### 不允许暴露异常的敏感信息

没有过滤敏感信息的异常堆栈往往会导致信息泄漏，

不正确的写法：

try {

FileInputStream fis =

new FileInputStream(System.getenv("APPDATA") + args[0]);

} catch (FileNotFoundException e) {

// Log the exception

throw new IOException("Unable to retrieve file", e);

}

正确的写法：

class ExceptionExample {

public static void main(String[] args) {

File file = null;

try {

file = new File(System.getenv("APPDATA") +

args[0]).getCanonicalFile();

if (!file.getPath().startsWith("c:\\homepath")) {

log.error("Invalid file");

return;

}

} catch (IOException x) {

log.error("Invalid file");

return;

}

try {

FileInputStream fis = new FileInputStream(file);

} catch (FileNotFoundException x) {

log.error("Invalid file");

return;

}

}

}

### 不允许抛出RuntimeException, Exception,Throwable

不正确的写法：

boolean isCapitalized(String s) {

if (s == null) {

throw new RuntimeException("Null String");

}

}

private void doSomething() throws Exception {

//...

}

正确写法：

boolean isCapitalized(String s) {

if (s == null) {

throw new NullPointerException();

}

}

private void doSomething() throws IOException {

//...

}

### 不要捕获NullPointerException或其他父类异常

不正确的写法：

boolean isName(String s) {

try {

String names[] = s.split(" ");

if (names.length != 2) {

return false;

}

return (isCapitalized(names[0]) && isCapitalized(names[1]));

} catch (NullPointerException e) {

return false;

}

}

正确的写法：

boolean isName(String s) /\* throws NullPointerException \*/ {

String names[] = s.split(" ");

if (names.length != 2) {

return false;

}

return (isCapitalized(names[0]) && isCapitalized(names[1]));

}

## 多线程编程

### 确保共享变量的可见性

对于共享变量，要确保一个线程对它的改动对其他线程是可见的。

线程可能会看到一个陈旧的共享变量的值。为了共享变量是最新的，可以将变量声明为volatile或同步读取和写入操作。

将共享变量声明为volatile：

final class ControlledStop implements Runnable {

private volatile boolean done = false;

@Override public void run() {

while (!done) {

try {

// ...

Thread.currentThread().sleep(1000); // Do something

} catch(InterruptedException ie) {

Thread.currentThread().interrupt(); // Reset interrupted status

}

}

}

public void shutdown() {

done = true;

}

}

同步读取和写入操作：

final class ControlledStop implements Runnable {

private boolean done = false;

@Override public void run() {

while (!isDone()) {

try {

// ...

Thread.currentThread().sleep(1000); // Do something

} catch(InterruptedException ie) {

Thread.currentThread().interrupt(); // Reset interrupted status

}

}

}

public synchronized boolean isDone() {

return done;

}

public synchronized void shutdown() {

done = true;

}

}

### 确保共享变量的操作是原子的

除了要确保共享变量的更新对其他线程可见的，还需要确保对共享变量的操作是原子的，这时将共享变量声明为volatile往往是不够的。需要使用同步机制或Lock

同步读取和写入操作：

final class Flag {

private volatile boolean flag = true;

public synchronized void toggle() {

flag ^= true; // Same as flag = !flag;

}

public boolean getFlag() {

return flag;

}

}

//使用读取锁确保读取和写入操作的原子性

final class Flag {

private boolean flag = true;

private final ReadWriteLock lock = new ReentrantReadWriteLock();

private final Lock readLock = lock.readLock();

private final Lock writeLock = lock.writeLock();

public void toggle() {

writeLock.lock();

try {

flag ^= true; // Same as flag = !flag;

} finally {

writeLock.unlock();

}

}

public boolean getFlag() {

readLock.lock();

try {

return flag;

} finally {

readLock.unlock();

}

}

}

### 不要调用Thread.run（），不要使用Thread.stop（）以终止线程

### 确保执行阻塞操作的线程可以终止

public final class SocketReader implements Runnable {

private final SocketChannel sc;

private final Object lock = new Object();

public SocketReader(String host, int port) throws IOException {

sc = SocketChannel.open(new InetSocketAddress(host, port));

}

@Override public void run() {

ByteBuffer buf = ByteBuffer.allocate(1024);

try {

synchronized (lock) {

while (!Thread.interrupted()) {

sc.read(buf);

// ...

}

}

} catch (IOException ie) {

// Forward to handler

}

}

public static void main(String[] args)

throws IOException, InterruptedException {

SocketReader reader = new SocketReader("somehost", 25);

Thread thread = new Thread(reader);

thread.start();

Thread.sleep(1000);

thread.interrupt();

}

}

### 相互依存的任务不要在一个有限的线程池执行

有限线程池指定可以同时执行在线程池中的线程数量的上限。程序不得使用有限线程池线程执行相互依赖的任务。可能会导致线程饥饿死锁，所有的线程池执行的任务正在等待一个可用的线程中执行一个内部队列阻塞

## 输入输出

### 程序终止前删除临时文件

### 检测和处理文件相关的错误

Java的文件操作方法往往有一个返回值，而不是抛出一个异常，表示失败。因此，忽略返回值文件操作的程序，往往无法检测到这些操作是否失败。Java程序必须检查执行文件I / O方法的返回值。

不正确的写法：

File file = new File(args[0]);

file.delete();

正确的写法：

File file = new File("file");

if (!file.delete()) {

log.error("Deletion failed");

}

### 及时释放资源

垃圾收集器无法释放非内存资源，如打开的文件描述符与数据库的连接。因此，不释放资源，可能导致资源耗尽攻击。

try {

final FileInputStream stream = new FileInputStream(fileName);

try {

final BufferedReader bufRead =

new BufferedReader(new InputStreamReader(stream));

String line;

while ((line = bufRead.readLine()) != null) {

sendLine(line);

}

} finally {

if (stream != null) {

try {

stream.close();

} catch (IOException e) {

// forward to handler

}

}

}

} catch (IOException e) {

// forward to handler

}

## 序列化

### 不要序列化未加密的敏感数据

序列化允许一个对象的状态被保存为一个字节序列，然后重新在稍后的时间恢复，它没有提供任何机制来保护序列化的数据。敏感的数据不应该被序列化的例子包括加密密钥，数字证书。

解决方法：

1. 对于数据成员可以使用transient ，声明该数据成员是瞬态的。
2. 重写序列化相关方法writeObject、readObject、readObjectNoData，防止被子类恶意重写

class SensitiveClass extends Number {

// ...

protected final Object writeObject(java.io.ObjectOutputStream out) throws NotSerializableException {

throw new NotSerializableException();

}

protected final Object readObject(java.io.ObjectInputStream in) throws NotSerializableException {

throw new NotSerializableException();

}

protected final Object readObjectNoData(java.io.ObjectInputStream in) throws NotSerializableException {

throw new NotSerializableException();

}

}

### 在序列化过程中避免内存和资源泄漏

不正确的写法：

class SensorData implements Serializable {

// 1 MB of data per instance!

public static SensorData readSensorData() {...}

public static boolean isAvailable() {...}

}

class SerializeSensorData {

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

ObjectOutputStream out = null;

try {

out = new ObjectOutputStream(

new BufferedOutputStream(new FileOutputStream("ser.dat")));

while (SensorData.isAvailable()) {

// note that each SensorData object is 1 MB in size

SensorData sd = SensorData.readSensorData();

out.writeObject(sd);

}

} finally {

if (out != null) {

out.close();

}

}

}

}

正确写法：

class SerializeSensorData {

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

ObjectOutputStream out = null;

try {

out = new ObjectOutputStream(

new BufferedOutputStream(new FileOutputStream("ser.dat")));

while (SensorData.isAvailable()) {

// note that each SensorData object is 1 MB in size

SensorData sd = SensorData.readSensorData();

out.writeObject(sd);

out.reset(); // reset the stream

}

} finally {

if (out != null) {

out.close();

}

}

}

}

### 反序列化要在程序最小权限的安全环境中