**https://blog.csdn.net/pxg943055021/article/details/55189370**

**FindBugs是**基于Bug Patterns概念，查找javabytecode（.class文件）中的潜在bug，主要检查bytecode中的bug patterns，如NullPoint空指针检查、没有合理关闭资源、字符串相同判断错（==，而不是equals）等

一、**Security 关于代码安全性防护**

1.Dm: Hardcoded constant database password (DMI\_CONSTANT\_DB\_PASSWORD) 代码中创建DB的密码时采用了写死的密码。 2.Dm: Empty database password (DMI\_EMPTY\_DB\_PASSWORD) 创建数据库连接时没有为数据库设置密码，这会使数据库没有必要的保护。 3.HRS: HTTP cookie formed from untrusted input (HRS\_REQUEST\_PARAMETER\_TO\_COOKIE) 此代码使用不受信任的HTTP参数构造一个HTTP Cookie。 4.HRS: HTTP Response splitting vulnerability (HRS\_REQUEST\_PARAMETER\_TO\_HTTP\_HEADER) 在代码中直接把一个HTTP的参数写入一个HTTP头文件中，它为HTTP的响应暴露了漏洞。 5.SQL: Nonconstant string passed to execute method on an SQL statement (SQL\_NONCONSTANT\_STRING\_PASSED\_TO\_EXECUTE) 该方法以字符串的形式来调用SQLstatement的execute方法，它似乎是动态生成SQL语句的方法。这会更容易受到SQL注入攻击。 6.XSS: JSP reflected cross site scripting vulnerability (XSS\_REQUEST\_PARAMETER\_TO\_JSP\_WRITER) 在代码中在JSP输出中直接写入一个HTTP参数，这会造成一个跨站点的脚本漏洞。

**二、Experimental**

1.LG: Potential lost logger changes due to weak reference in OpenJDK (LG\_LOST\_LOGGER\_DUE\_TO\_WEAK\_REFERENCE)

OpenJDK的引入了一种潜在的不兼容问题，特别是，java.util.logging.Logger的行为改变时。它现在使用内部弱引用，而不是强引用。–logger配置改变，它就是丢失对logger的引用，这本是一个合理的变化，但不幸的是一些代码对旧的行为有依赖关系。这意味着，当进行垃圾收集时对logger配置将会丢失。例如：

public static void initLogging() throws Exception {  Logger logger = Logger.getLogger("edu.umd.cs");  logger.addHandler(new FileHandler()); // call to change logger configuration  logger.setUseParentHandlers(false); // another call to change logger configuration } 该方法结束时logger的引用就丢失了，如果你刚刚结束调用initLogging方法后进行垃圾回收，logger的配置将会丢失（因为只有保持记录器弱引用）。 public static void main(String[] args) throws Exception {  initLogging(); // adds a file handler to the logger  System.gc(); // logger configuration lost  Logger.getLogger("edu.umd.cs").info("Some message"); // this isn't logged to the file as expected } 2.OBL: Method may fail to clean up stream or resource (OBL\_UNSATISFIED\_OBLIGATION) 这种方法可能无法清除（关闭，处置）一个流，数据库对象，或其他资源需要一个明确的清理行动。 一般来说，如果一个方法打开一个流或其他资源，该方法应该使用try / finally块来确保在方法返回之前流或资源已经被清除了。这种错误模式基本上和OS\_OPEN\_STREAM和ODR\_OPEN\_DATABASE\_RESOURCE错误模式相同，但是是在不同在静态分析技术。我们正为这个错误模式的效用收集反馈意见。

**三、Bad practice代码实现中的一些坏习惯**

1.AM: Creates an empty jar file entry (AM\_CREATES\_EMPTY\_JAR\_FILE\_ENTRY)

调用putNextEntry()方法写入新的 jar 文件条目时立即调用closeEntry()方法。这样会造成JarFile条目为空。 2.AM: Creates an empty zip file entry (AM\_CREATES\_EMPTY\_ZIP\_FILE\_ENTRY) 调用putNextEntry()方法写入新的 zip 文件条目时立即调用closeEntry()方法。这样会造成ZipFile条目为空。 3.BC: Equals method should not assume anything about the type of its argument (BC\_EQUALS\_METHOD\_SHOULD\_WORK\_FOR\_ALL\_OBJECTS) equals(Object o)方法不能对参数o的类型做任何的假设。比较此对象与指定的对象。当且仅当该参数不为 null，并且是表示与此对象相同的类型的对象时，结果才为 true。 4.BC: Random object created and used only once (DMI\_RANDOM\_USED\_ONLY\_ONCE) 随机创建对象只使用过一次就抛弃 5.BIT: Check for sign of bitwise operation (BIT\_SIGNED\_CHECK) 检查位操作符运行是否合理 ((event.detail & SWT.SELECTED) > 0) If SWT.SELECTED is a negative number, this is a candidate for a bug. Even when SWT.SELECTED is not negative, it seems good practice to use '!= 0' instead of '> 0'. 6.CN: Class implements Cloneable but does not define or use clone method (CN\_IDIOM) 按照惯例，实现此接口的类应该使用公共方法重写 Object.clone（它是受保护的），以获得有关重写此方法的详细信息。此接口不 包含 clone 方法。因此，因为某个对象实现了此接口就克隆它是不可能的,应该实现此接口的类应该使用公共方法重写 Object.clone 7.CN: clone method does not call super.clone() (CN\_IDIOM\_NO\_SUPER\_CALL) 一个非final类型的类定义了clone()方法而没有调用super.clone()方法。例如：B扩展自A，如果B中clone方法调用了spuer.clone（），而A中的clone没有调用spuer.clone()，就会造成结果类型不准确。要求A的clone方法中调用spuer.clone()方法。 8.CN: Class defines clone() but doesn't implement Cloneable (CN\_IMPLEMENTS\_CLONE\_BUT\_NOT\_CLONEABLE) 类中定义了clone方法但是它没有实现Cloneable接口 9.Co: Abstract class defines covariant compareTo() method (CO\_ABSTRACT\_SELF) 抽象类中定义了多个compareTo()方法，正确的是覆写Comparable中的compareTo方法，方法的参数为Object类型，如下例： int compareTo(T o)  比较此对象与指定对象的顺序。 10.Co: Covariant compareTo() method defined (CO\_SELF\_NO\_OBJECT) 类中定义了多个compareTo()方法，正确的是覆写Comparable中的compareTo方法，方法的参数为Object类型 11.DE: Method might drop exception (DE\_MIGHT\_DROP) 方法可能抛出异常 12.DE: Method might ignore exception (DE\_MIGHT\_IGNORE) 方法可能忽略异常 13.DMI: Don't use removeAll to clear a collection (DMI\_USING\_REMOVEALL\_TO\_CLEAR\_COLLECTION) 不要用removeAll方法去clear一个集合 14.DP: Classloaders should only be created inside doPrivileged block (DP\_CREATE\_CLASSLOADER\_INSIDE\_DO\_PRIVILEGED) 类加载器只能建立在特殊的方法体内 15.Dm: Method invokes System.exit(...) (DM\_EXIT) 在方法中调用System.exit(...)语句，考虑用RuntimeException来代替 16.Dm: Method invokes dangerous method runFinalizersOnExit (DM\_RUN\_FINALIZERS\_ON\_EXIT) 在方法中调用了System.runFinalizersOnExit 或者Runtime.runFinalizersOnExit方法，因为这样做是很危险的。 17.ES: Comparison of String parameter using == or != (ES\_COMPARING\_PARAMETER\_STRING\_WITH\_EQ) 用==或者!=方法去比较String类型的参数 18.ES: Comparison of String objects using == or != (ES\_COMPARING\_STRINGS\_WITH\_EQ) 用==或者！=去比较String类型的对象 19.Eq: Abstract class defines covariant equals() method (EQ\_ABSTRACT\_SELF) 20.Eq: Equals checks for noncompatible operand (EQ\_CHECK\_FOR\_OPERAND\_NOT\_COMPATIBLE\_WITH\_THIS) equals方法检查不一致的操作。两个类根本就是父子关系而去调用equals方法去判读对象是否相等。 public boolean equals(Object o) {   if (o instanceof Foo)     return name.equals(((Foo)o).name);   else if (o instanceof String)     return name.equals(o);   else return false; 21.Eq: Class defines compareTo(...) and uses Object.equals() (EQ\_COMPARETO\_USE\_OBJECT\_EQUALS) 类中定义了compareTo方法但是继承了Object中的compareTo方法 22.Eq: equals method fails for subtypes (EQ\_GETCLASS\_AND\_CLASS\_CONSTANT) 类中的equals方法可能被子类中的方法所破坏，当使用类似于Foo.class == o.getClass()的判断时考虑用this.getClass() == o.getClass()来替换 23.Eq: Covariant equals() method defined (EQ\_SELF\_NO\_OBJECT) 类中定义了多个equals方法。正确的做法是覆写Object中的equals方法，它的参数为Object类型的对象。 24.FI: Empty finalizer should be deleted (FI\_EMPTY) 为空的finalizer方法应该删除。一下关于finalizer的内容省略 25.GC: Unchecked type in generic call (GC\_UNCHECKED\_TYPE\_IN\_GENERIC\_CALL) This call to a generic collection method passes an argument while compile type Object where a specific type from the generic type parameters is expected. Thus, neither the standard Java type system nor static analysis can provide useful information on whether the object being passed as a parameter is of an appropriate type. 26.HE: Class defines equals() but not hashCode() (HE\_EQUALS\_NO\_HASHCODE) 方法定义了equals方法却没有定义hashCode方法 27.HE: Class defines hashCode() but not equals() (HE\_HASHCODE\_NO\_EQUALS)  类定义了hashCode方法去没有定义equal方法 28.HE: Class defines equals() and uses Object.hashCode() (HE\_EQUALS\_USE\_HASHCODE) 一个类覆写了equals方法，没有覆写hashCode方法，使用了Object对象的hashCode方法 29.HE: Class inherits equals() and uses Object.hashCode() (HE\_INHERITS\_EQUALS\_USE\_HASHCODE) 子类继承了父类的equals方法却使用了Object的hashCode方法 30.IC: Superclass uses subclass during initialization (IC\_SUPERCLASS\_USES\_SUBCLASS\_DURING\_INITIALIZATION) 子类在父类未初始化之前使用父类对象实例 public class CircularClassInitialization { static class InnerClassSingleton extends CircularClassInitialization { static InnerClassSingleton singleton = new InnerClassSingleton(); } static CircularClassInitialization foo = InnerClassSingleton.singleton; } 31.IMSE: Dubious catching of IllegalMonitorStateException (IMSE\_DONT\_CATCH\_IMSE) 捕捉违法的监控状态异常，例如当没有获取到对象锁时使用其wait和notify方法 32.ISC: Needless instantiation of class that only supplies static methods (ISC\_INSTANTIATE\_STATIC\_CLASS) 为使用静态方法而创建一个实例对象。调用静态方法时只需要使用类名+静态方法名就可以了。 33.It: Iterator next() method can't throw NoSuchElementException (IT\_NO\_SUCH\_ELEMENT) 迭代器的next方法不能够抛出NoSuchElementException 34.J2EE: Store of non serializable object into HttpSession (J2EE\_STORE\_OF\_NON\_SERIALIZABLE\_OBJECT\_INTO\_SESSION) 在HttpSession对象中保存非连续的对象 35.JCIP: Fields of immutable classes should be final (JCIP\_FIELD\_ISNT\_FINAL\_IN\_IMMUTABLE\_CLASS)  The class is annotated with net.jcip.annotations.Immutable, and the rules for that annotation require that all fields are final. . 36.NP: Method with Boolean return type returns explicit null (NP\_BOOLEAN\_RETURN\_NULL) 返回值为boolean类型的方法直接返回null，这样会导致空指针异常 37.NP: equals() method does not check for null argument (NP\_EQUALS\_SHOULD\_HANDLE\_NULL\_ARGUMENT) 变量调用equals方法时没有进行是否为null的判断 38.NP: toString method may return null (NP\_TOSTRING\_COULD\_RETURN\_NULL) toString方法可能返回null 39.Nm: Class names should start with an upper case letter (NM\_CLASS\_NAMING\_CONVENTION) 类的名称以大写字母名称开头 40.Nm: Class is not derived from an Exception, even though it is named as such (NM\_CLASS\_NOT\_EXCEPTION) 类的名称中含有Exception但是却不是一个异常类的子类，这种名称会造成混淆 41.Nm: Confusing method names (NM\_CONFUSING) 令人迷惑的方面命名 42.Nm: Field names should start with a lower case letter (NM\_FIELD\_NAMING\_CONVENTION) 非final类型的字段需要遵循驼峰命名原则 43.Nm: Use of identifier that is a keyword in later versions of Java (NM\_FUTURE\_KEYWORD\_USED\_AS\_IDENTIFIER) 验证是否是java预留关键字 44.Nm: Use of identifier that is a keyword in later versions of Java (NM\_FUTURE\_KEYWORD\_USED\_AS\_MEMBER\_IDENTIFIER) 验证是否时java中的关键字 45.Nm: Method names should start with a lower case letter (NM\_METHOD\_NAMING\_CONVENTION) 方法名称以小写字母开头 46.Nm: Class names shouldn't shadow simple name of implemented interface (NM\_SAME\_SIMPLE\_NAME\_AS\_INTERFACE) 实现同一接口实现类不能使用相同的名称，即使它们位于不同的包中 47.Nm: Class names shouldn't shadow simple name of superclass (NM\_SAME\_SIMPLE\_NAME\_AS\_SUPERCLASS) 继承同一父类的子类不能使用相同的名称，即使它们位于不同的包中 48.Nm: Very confusing method names (but perhaps intentional) (NM\_VERY\_CONFUSING\_INTENTIONAL) 很容易混淆的方法命名，例如方法的名称名称使用使用大小写来区别两个不同的方法。 49.Nm: Method doesn't override method in superclass due to wrong package for parameter (NM\_WRONG\_PACKAGE\_INTENTIONAL) 由于错误引用了不同包中相同类名的对象而不能够正确的覆写父类中的方法 import alpha.Foo; public class A {   public int f(Foo x) { return 17; } } import beta.Foo; public class B extends A {   public int f(Foo x) { return 42; }   public int f(alpha.Foo x) { return 27; } } 50.ODR: Method may fail to close database resource (ODR\_OPEN\_DATABASE\_RESOURCE) 方法中可能存在关闭数据连接失败的情况 51.OS: Method may fail to close stream (OS\_OPEN\_STREAM) 方法中可能存在关闭流失败的情况 52.OS: Method may fail to close stream on exception (OS\_OPEN\_STREAM\_EXCEPTION\_PATH) 方法中可能存在关闭流时出现异常情况 53.RC: Suspicious reference comparison to constant (RC\_REF\_COMPARISON\_BAD\_PRACTICE) 当两者为不同类型的对象时使用equals方法来比较它们的值是否相等，而不是使用==方法。例如比较的两者为java.lang.Integer, java.lang.Float 54.RC: Suspicious reference comparison of Boolean values (RC\_REF\_COMPARISON\_BAD\_PRACTICE\_BOOLEAN) 使用== 或者 !=操作符来比较两个 Boolean类型的对象，建议使用equals方法。 55.RR: Method ignores results of InputStream.read() (RR\_NOT\_CHECKED) InputStream.read方法忽略返回的多个字符，如果对结果没有检查就没法正确处理用户读取少量字符请求的情况。 56.RR: Method ignores results of InputStream.skip() (SR\_NOT\_CHECKED) InputStream.skip()方法忽略返回的多个字符，如果对结果没有检查就没法正确处理用户跳过少量字符请求的情况 57.RV: Method ignores exceptional return value (RV\_RETURN\_VALUE\_IGNORED\_BAD\_PRACTICE) 方法忽略返回值的异常信息 58.SI: Static initializer creates instance before all static final fields assigned (SI\_INSTANCE\_BEFORE\_FINALS\_ASSIGNED) 在所有的static final字段赋值之前去使用静态初始化的方法创建一个类的实例。 59.Se: Non-serializable value stored into instance field of a serializable class (SE\_BAD\_FIELD\_STORE) 非序列化的值保存在声明为序列化的的非序列化字段中 60.Se: Comparator doesn't implement Serializable (SE\_COMPARATOR\_SHOULD\_BE\_SERIALIZABLE) Comparator接口没有实现Serializable接口 61.Se: Serializable inner class (SE\_INNER\_CLASS) 序列化内部类 62.Se: serialVersionUID isn't final (SE\_NONFINAL\_SERIALVERSIONID) 关于UID类的检查内容省略 63.Se: Class is Serializable but its superclass doesn't define a void constructor (SE\_NO\_SUITABLE\_CONSTRUCTOR) 子类序列化时父类没有提供一个void的构造函数 64.Se: Class is Externalizable but doesn't define a void constructor (SE\_NO\_SUITABLE\_CONSTRUCTOR\_FOR\_EXTERNALIZATION) Externalizable 实例类没有定义一个void类型的构造函数 65.Se: The readResolve method must be declared with a return type of Object. (SE\_READ\_RESOLVE\_MUST\_RETURN\_OBJECT) readResolve从流中读取类的一个实例，此方法必须声明返回一个Object类型的对象 66.Se: Transient field that isn't set by deserialization. (SE\_TRANSIENT\_FIELD\_NOT\_RESTORED) This class contains a field that is updated at multiple places in the class, thus it seems to be part of the state of the class. However, since the field is marked as transient and not set in readObject or readResolve, it will contain the default value in any deserialized instance of the class. 67.SnVI: Class is Serializable, but doesn't define serialVersionUID (SE\_NO\_SERIALVERSIONID) 一个类实现了Serializable接口但是没有定义serialVersionUID类型的变量。序列化运行时使用一个称为 serialVersionUID 的版本号与每个可序列化类相关联，该序列号在反序列化过程中用于验证序列化对象的发送者和接收者是否为该对象加载了与序列化兼容的类。如果接收者加载的该对象的类的 serialVersionUID 与对应的发送者的类的版本号不同，则反序列化将会导致 InvalidClassException。可序列化类可以通过声明名为 "serialVersionUID" 的字段（该字段必须是静态 (static)、最终 (final) 的 long 型字段）显式声明其自己的 serialVersionUID：   ANY-ACCESS-MODIFIER static final long serialVersionUID = 42L; 68.UI: Usage of GetResource may be unsafe if class is extended (UI\_INHERITANCE\_UNSAFE\_GETRESOURCE) 当一个类被子类继承后不要使用this.getClass().getResource(...)来获取资源

**四、Correctness关于代码正确性相关方面的**

1.BC: Impossible cast (BC\_IMPOSSIBLE\_CAST) 不可能的类转换，执行时会抛出ClassCastException 2.BC: Impossible downcast (BC\_IMPOSSIBLE\_DOWNCAST) 父类在向下进行类型转换时抛出ClassCastException 3.BC: Impossible downcast of toArray() result (BC\_IMPOSSIBLE\_DOWNCAST\_OF\_TOARRAY) 集合转换为数组元素时发生的类转换错误。 This code is casting the result of calling toArray() on a collection to a type more specific than Object[], as in:  String[] getAsArray(Collection<String> c) {   return (String[]) c.toArray();   } This will usually fail by throwing a ClassCastException. The toArray() of almost all collections return an Object[]. They can't really do anything else, since the Collection object has no reference to the declared generic type of the collection.  The correct way to do get an array of a specific type from a collection is to use c.toArray(new String[]); or c.toArray(new String[c.size()]); (the latter is slightly more efficient).  4.BC: instanceof will always return false (BC\_IMPOSSIBLE\_INSTANCEOF) 采用instaneof方法进行比较时总是返回false。前提是保证它不是由于某些逻辑错误造成的。 5.BIT: Incompatible bit masks (BIT\_AND) 错误的使用&位操作符，例如(e & C) 6.BIT: Check to see if ((...) & 0) == 0 (BIT\_AND\_ZZ) 检查恒等的逻辑错误 7.BIT: Incompatible bit masks (BIT\_IOR) 错误的使用|位操作符，例如(e | C) 8.BIT: Check for sign of bitwise operation (BIT\_SIGNED\_CHECK\_HIGH\_BIT) 检查逻辑运算符操作返回的标识。例如((event.detail & SWT.SELECTED) > 0)，建议采用!=0代替>0 9.BOA: Class overrides a method implemented in super class Adapter wrongly (BOA\_BADLY\_OVERRIDDEN\_ADAPTER) 子类错误的覆写父类中用于适配监听其他事件的方法，从而导致当触发条件发生时不能被监听者调用 10.Bx: Primitive value is unboxed and coerced for ternary operator (BX\_UNBOXED\_AND\_COERCED\_FOR\_TERNARY\_OPERATOR) 在三元运算符操作时如果没有对值进行封装或者类型转换。例如：b ? e1 : e2 11.DLS: Dead store of class literal (DLS\_DEAD\_STORE\_OF\_CLASS\_LITERAL) 以类的字面名称方式为一个字段赋值后再也没有去使用它，在1.4jdk中它会自动调用静态的初始化方法，而在jdk1.5中却不会去执行。 12.DLS: Overwritten increment (DLS\_OVERWRITTEN\_INCREMENT) 覆写增量增加错误i = i++ 13.DMI: Bad constant value for month (DMI\_BAD\_MONTH) hashNext方法调用next方法。 14.DMI: Collections should not contain themselves (DMI\_COLLECTIONS\_SHOULD\_NOT\_CONTAIN\_THEMSELVES) 集合没有包含他们自己本身。 15.DMI: Invocation of hashCode on an array (DMI\_INVOKING\_HASHCODE\_ON\_ARRAY) 数组直接使用hashCode方法来返回哈希码。 int [] a1 = new int[]{1,2,3,4}; System.out.println(a1.hashCode()); System.out.println(java.util.Arrays.hashCode(a1)); 16.DMI: Double.longBitsToDouble invoked on an int (DMI\_LONG\_BITS\_TO\_DOUBLE\_INVOKED\_ON\_INT) 17.DMI: Vacuous call to collections (DMI\_VACUOUS\_SELF\_COLLECTION\_CALL) 集合的调用不能被感知。例如c.containsAll(c)总是返回true，而c.retainAll(c)的返回值不能被感知。 18.Dm: Can't use reflection to check for presence of annotation without runtime retention (DMI\_ANNOTATION\_IS\_NOT\_VISIBLE\_TO\_REFLECTION) Unless an annotation has itself been annotated with @Retention(RetentionPolicy.RUNTIME), the annotation can't be observed using reflection (e.g., by using the isAnnotationPresent method). . 19.Dm: Useless/vacuous call to EasyMock method (DMI\_VACUOUS\_CALL\_TO\_EASYMOCK\_METHOD) While ScheduledThreadPoolExecutor inherits from ThreadPoolExecutor, a few of the inherited tuning methods are not useful for it. In particular, because it acts as a fixed-sized pool using corePoolSize threads and an unbounded queue, adjustments to maximumPoolSize have no useful effect. 20.EC: equals() used to compare array and nonarray (EC\_ARRAY\_AND\_NONARRAY) 数组对象使用equals方法和非数组对象进行比较。即使比较的双方都是数组对象也不应该使用equals方法，而应该比较它们的内容是否相等使用java.util.Arrays.equals(Object[], Object[]); 21.EC: equals(...) used to compare incompatible arrays (EC\_INCOMPATIBLE\_ARRAY\_COMPARE) 使用equls方法去比较类型不相同的数组。例如：String[] and StringBuffer[], or String[] and int[] 22.EC: Call to equals() with null argument (EC\_NULL\_ARG) 调用equals的对象为null 23.EC: Call to equals() comparing unrelated class and interface (EC\_UNRELATED\_CLASS\_AND\_INTERFACE) 使用equals方法比较不相关的类和接口 24.EC: Call to equals() comparing different interface types (EC\_UNRELATED\_INTERFACES) 调用equals方法比较不同类型的接口 25.EC: Call to equals() comparing different types (EC\_UNRELATED\_TYPES) 调用equals方法比较不同类型的类 26.EC: Using pointer equality to compare different types (EC\_UNRELATED\_TYPES\_USING\_POINTER\_EQUALITY) This method uses using pointer equality to compare two references that seem to be of different types. The result of this comparison will always be false at runtime. 27.Eq: equals method always returns false (EQ\_ALWAYS\_FALSE) 使用equals方法返回值总是false 28.Eq: equals method always returns true (EQ\_ALWAYS\_TRUE) equals方法返回值总是true 29.Eq: equals method compares class names rather than class objects (EQ\_COMPARING\_CLASS\_NAMES) 使用equals方法去比较一个类的实例和类的类型 30.Eq: Covariant equals() method defined for enum (EQ\_DONT\_DEFINE\_EQUALS\_FOR\_ENUM) This class defines an enumeration, and equality on enumerations are defined using object identity. Defining a covariant equals method for an enumeration value is exceptionally bad practice, since it would likely result in having two different enumeration values that compare as equals using the covariant enum method, and as not equal when compared normally. Don't do it. 31.Eq: equals() method defined that doesn't override equals(Object) (EQ\_OTHER\_NO\_OBJECT) 类中定义的equals方法时不要覆写equals（Object）方法 32.Eq: equals() method defined that doesn't override Object.equals(Object) (EQ\_OTHER\_USE\_OBJECT) 类中定义的equals方法时不要覆写Object中的equals（Object）方法 33.Eq: equals method overrides equals in superclass and may not be symmetric (EQ\_OVERRIDING\_EQUALS\_NOT\_SYMMETRIC) 34.Eq: Covariant equals() method defined, Object.equals(Object) inherited (EQ\_SELF\_USE\_OBJECT) 类中定义了一组equals方法，但是都是继承的java.lang.Object class中的equals(Object)方法 35.FE: Doomed test for equality to NaN (FE\_TEST\_IF\_EQUAL\_TO\_NOT\_A\_NUMBER) This code checks to see if a floating point value is equal to the special Not A Number value (e.g., if (x == Double.NaN)). However, because of the special semantics of NaN, no value is equal to Nan, including NaN. Thus, x == Double.NaN always evaluates to false. To check to see if a value contained in x is the special Not A Number value, use Double.isNaN(x) (or Float.isNaN(x) if x is floating point precision). 36.FS: Format string placeholder incompatible with passed argument (VA\_FORMAT\_STRING\_BAD\_ARGUMENT) 错误使用参数类型来格式化字符串 37.FS: The type of a supplied argument doesn't match format specifier (VA\_FORMAT\_STRING\_BAD\_CONVERSION) 指定的格式字符串和参数类型不匹配，例如：String.format("%d", "1") 38.FS: MessageFormat supplied where printf style format expected (VA\_FORMAT\_STRING\_EXPECTED\_MESSAGE\_FORMAT\_SUPPLIED) 但用String的format方法时实际调用了ＭｅｓｓａｇｅＦｏｒｍａｔ中干的格式化方法而引起格式化结果出错。 39.FS: More arguments are passed than are actually used in the format string (VA\_FORMAT\_STRING\_EXTRA\_ARGUMENTS\_PASSED) 使用String的format方法时有非法的参数也经过了格式化操作。 40.FS: Illegal format string (VA\_FORMAT\_STRING\_ILLEGAL) 格式化String对象语句错误 41.FS: Format string references missing argument (VA\_FORMAT\_STRING\_MISSING\_ARGUMENT) String的format操作缺少必要的参数。 42.FS: No previous argument for format string (VA\_FORMAT\_STRING\_NO\_PREVIOUS\_ARGUMENT) 格式字符串定义错误，例如：formatter.format("%<s %s", "a", "b"); 抛出MissingFormatArgumentException异常 43.GC: No relationship between generic parameter and method argument (GC\_UNRELATED\_TYPES) This call to a generic collection method contains an argument with an incompatible class from that of the collection's parameter (i.e., the type of the argument is neither a supertype nor a subtype of the corresponding generic type argument). Therefore, it is unlikely that the collection contains any objects that are equal to the method argument used here. Most likely, the wrong value is being passed to the method. In general, instances of two unrelated classes are not equal. For example, if the Foo and Bar classes are not related by subtyping, then an instance of Foo should not be equal to an instance of Bar. Among other issues, doing so will likely result in an equals method that is not symmetrical. For example, if you define the Foo class so that a Foo can be equal to a String, your equals method isn't symmetrical since a String can only be equal to a String.  In rare cases, people do define nonsymmetrical equals methods and still manage to make their code work. Although none of the APIs document or guarantee it, it is typically the case that if you check if a Collection<String> contains a Foo, the equals method of argument (e.g., the equals method of the Foo class) used to perform the equality checks.  44.HE: Signature declares use of unhashable class in hashed construct (HE\_SIGNATURE\_DECLARES\_HASHING\_OF\_UNHASHABLE\_CLASS) A method, field or class declares a generic signature where a non-hashable class is used in context where a hashable class is required. A class that declares an equals method but inherits a hashCode() method from Object is unhashable, since it doesn't fulfill the requirement that equal objects have equal hashCodes. 45.HE: Use of class without a hashCode() method in a hashed data structure (HE\_USE\_OF\_UNHASHABLE\_CLASS) A class defines an equals(Object) method but not a hashCode() method, and thus doesn't fulfill the requirement that equal objects have equal hashCodes. An instance of this class is used in a hash data structure, making the need to fix this problem of highest importance. 46.ICAST: integral value cast to double and then passed to Math.ceil (ICAST\_INT\_CAST\_TO\_DOUBLE\_PASSED\_TO\_CEIL) integral的值转换为double后使用了Math.ceil方法 47.ICAST: int value cast to float and then passed to Math.round (ICAST\_INT\_CAST\_TO\_FLOAT\_PASSED\_TO\_ROUND) int 类型的值转换为float类型之后调用了Math.round方法 48.IJU: JUnit assertion in run method will not be noticed by JUnit (IJU\_ASSERT\_METHOD\_INVOKED\_FROM\_RUN\_METHOD) 在JUnit中的断言在run方法中不会被告知 49.IJU: TestCase declares a bad suite method (IJU\_BAD\_SUITE\_METHOD) 在一个JUnit类中声明的一个suite()方法必须声明为 public static junit.framework.Test suite() 或者 public static junit.framework.TestSuite suite()的形式。 50.IL: A collection is added to itself (IL\_CONTAINER\_ADDED\_TO\_ITSELF) 集合本身作为add方法的参数，这样会引起内容溢出。 51.IL: An apparent infinite loop (IL\_INFINITE\_LOOP) 方法的自调用引起的死循环 52.IM: Integer multiply of result of integer remainder (IM\_MULTIPLYING\_RESULT\_OF\_IREM) 和整数余数进行乘法运算。例如：i % 60 \* 1000 是进行(i % 60) \* 1000运算而不是 i % (60 \* 1000) 53.INT: Bad comparison of nonnegative value with negative constant (INT\_BAD\_COMPARISON\_WITH\_NONNEGATIVE\_VALUE) 保证非负数和负数进行比较 54.INT: Bad comparison of signed byte (INT\_BAD\_COMPARISON\_WITH\_SIGNED\_BYTE) 比较有符合数，要先把有符号数转换为无符合数再进行比较 55.IO: Doomed attempt to append to an object output stream (IO\_APPENDING\_TO\_OBJECT\_OUTPUT\_STREAM) 宣布试图在对象的输出流处添加元素，如果你希望能够添加进一个对象的输出流中必须保证对象的输出流处于打开状态。 56.IP: A parameter is dead upon entry to a method but overwritten (IP\_PARAMETER\_IS\_DEAD\_BUT\_OVERWRITTEN) The initial value of this parameter is ignored, and the parameter is overwritten here. This often indicates a mistaken belief that the write to the parameter will be conveyed back to the caller. 传入参数的值被忽略，但是对传入值进行了修改，并返回给了调用者 57.MF: Class defines field that masks a superclass field (MF\_CLASS\_MASKS\_FIELD) 子类中定义了和父类中同名的字段。在调用时会出错 58.MF: Method defines a variable that obscures a field (MF\_METHOD\_MASKS\_FIELD) 在方法中定义的局部变量和类变量或者父类变量同名，从而引起字段混淆。 59.NP: Null pointer dereference (NP\_ALWAYS\_NULL) 对象赋为null值后 没有被重新赋值 60.NP: Null pointer dereference in method on exception path (NP\_ALWAYS\_NULL\_EXCEPTION) A pointer which is null on an exception path is dereferenced here.  This will lead to a NullPointerException when the code is executed.  Note that because FindBugs currently does not prune infeasible exception paths, this may be a false warning. Also note that FindBugs considers the default case of a switch statement to be an exception path, since the default case is often infeasible. 空指针引用上调用去除引用方法，将发生空指针异常 61.NP: Method does not check for null argument (NP\_ARGUMENT\_MIGHT\_BE\_NULL) 方法没有判断参数是否为空 62.NP: close() invoked on a value that is always null (NP\_CLOSING\_NULL) 一个为空的对象调用close方法 63.NP: Null value is guaranteed to be dereferenced (NP\_GUARANTEED\_DEREF) There is a statement or branch that if executed guarantees that a value is null at this point, and that value that is guaranteed to be dereferenced (except on forward paths involving runtime exceptions). 在正常的null判断分支上，对象去除引用操作是受保护的不允许的 64.NP: Value is null and guaranteed to be dereferenced on exception path (NP\_GUARANTEED\_DEREF\_ON\_EXCEPTION\_PATH) There is a statement or branch on an exception path that if executed guarantees that a value is null at this point, and that value that is guaranteed to be dereferenced (except on forward paths involving runtime exceptions). 65.NP: Method call passes null to a nonnull parameter (NP\_NONNULL\_PARAM\_VIOLATION) 方法中为null的参数没有被重新赋值 void test(){ String ss = null; sya(ss); } public void sya(String ad){ ad.getBytes(); } 66.NP: Method may return null, but is declared @NonNull (NP\_NONNULL\_RETURN\_VIOLATION) 方法声明了返回值不能为空，但是方法中有可能返回null 67.NP: A known null value is checked to see if it is an instance of a type (NP\_NULL\_INSTANCEOF) 检查一个为null的值是否是想要的类型对象，而不是由于粗心或者逻辑错误引起的 68.NP: Possible null pointer dereference (NP\_NULL\_ON\_SOME\_PATH) 对象可能没有重新赋值 69.NP: Possible null pointer dereference in method on exception path (NP\_NULL\_ON\_SOME\_PATH\_EXCEPTION) A reference value which is null on some exception control path is dereferenced here.  This may lead to a NullPointerException when the code is executed.  Note that because FindBugs currently does not prune infeasible exception paths, this may be a false warning. Also note that FindBugs considers the default case of a switch statement to be an exception path, since the default case is often infeasible. 在异常null值处理分支调用的方法上，可能存在对象去除引用操作 70.NP: Method call passes null for nonnull parameter (NP\_NULL\_PARAM\_DEREF\_ALL\_TARGETS\_DANGEROUS) 方法参数中声明为nonnull类型的参数为null void test(){ String ss = null; sya(ss); } public void sya(@nonnull String ad){ ad.getBytes(); } 71.NP: Store of null value into field annotated NonNull (NP\_STORE\_INTO\_NONNULL\_FIELD) 为一个已经声明为不能为null值的属性赋值为null。 72.Nm: Class defines equal(Object); should it be equals(Object)? (NM\_BAD\_EQUAL) 类中定义了一个equal方法但是却不是覆写的Object对象的equals方法 73.Nm: Class defines hashcode(); should it be hashCode()? (NM\_LCASE\_HASHCODE) 类中定义了一个hashCode方法但是却不是覆写的Object中的hashCode方法 74.Nm: Class defines tostring(); should it be toString()? (NM\_LCASE\_TOSTRING) 类中定义了一个toString方法但是却不是覆写的Object中的toString方法 75.Nm: Apparent method/constructor confusion (NM\_METHOD\_CONSTRUCTOR\_CONFUSION) 构造方法定义混乱，保证一个标准的构造函数。 例如： SA(){ } void SA(){ } 76.Nm: Very confusing method names (NM\_VERY\_CONFUSING) 混乱的方法命名，如getName和getname方法同时出现的时候 77.Nm: Method doesn't override method in superclass due to wrong package for parameter (NM\_WRONG\_PACKAGE) 方法因为取了不同包中的同名的对象而没有正确覆写父类中的同名方法 import alpha.Foo; public class A {   public int f(Foo x) { return 17; } } ---- import beta.Foo; public class B extends A {   public int f(Foo x) { return 42; } } 78.QBA: Method assigns boolean literal in boolean expression (QBA\_QUESTIONABLE\_BOOLEAN\_ASSIGNMENT) 再if或者while表达式中使用boolean类型的值时应该使用==去判断，而不是采用=操作 79.RC: Suspicious reference comparison (RC\_REF\_COMPARISON) 比较两个对象值是否相等时应该采用equals方法，而不是==方法 80.RE: Invalid syntax for regular expression (RE\_BAD\_SYNTAX\_FOR\_REGULAR\_EXPRESSION) 对正则表达式使用了错误的语法，会抛出未经检查的异常，表明正则表达式模式中的语法错误。 81.RE: File.separator used for regular expression (RE\_CANT\_USE\_FILE\_SEPARATOR\_AS\_REGULAR\_EXPRESSION) 使用正则表达式使用了错误的文件分隔符，在windows系统中正则表达式不会匹配’\’而应该使用'\\' 82.RV: Random value from 0 to 1 is coerced to the integer 0 (RV\_01\_TO\_INT) 从0到1随机值被强制为整数值0。在强制得到一个整数之前，你可能想得到多个随机值。或使用Random.nextInt（n）的方法。 83.RV: Bad attempt to compute absolute value of signed 32-bit hashcode (RV\_ABSOLUTE\_VALUE\_OF\_HASHCODE) 此代码生成一个哈希码，然后计算该哈希码的绝对值。如果哈希码是Integer.MIN\_VALUE的，那么结果将是负数（因为Math.abs（Integer.MIN\_VALUE的）== Integer.MIN\_VALUE的）。 在2^ 32值之外字符串有一个Integer.MIN\_VALUE的hashCode包括“polygenelubricants”，“GydZG\_”和“，”DESIGNING WORKHOUSES “。 84.RV: Bad attempt to compute absolute value of signed 32-bit random integer (RV\_ABSOLUTE\_VALUE\_OF\_RANDOM\_INT) 此代码生成一个随机的符号整数，然后计算该随机整数的绝对值。如果随机数生成数绝对值为Integer.MIN\_VALUE的，那么结果将是负数（因为Math.abs（Integer.MIN\_VALUE的）== Integer.MIN\_VALUE的）。 85.RV: Exception created and dropped rather than thrown (RV\_EXCEPTION\_NOT\_THROWN) 此代码创建一个异常（或错误）的对象，但不会用它做任何事情。例如：if (x < 0)   new IllegalArgumentException("x must be nonnegative"); 这可能是程序员的意图抛出创建的异常： if (x < 0)   throw new IllegalArgumentException("x must be nonnegative"); 86.RV: Method ignores return value (RV\_RETURN\_VALUE\_IGNORED) 该方法的返回值应该进行检查。这种警告通常出现在调用一个不可变对象的方法，认为它更新了对象的值。例如：String dateString = getHeaderField(name); dateString.trim(); 程序员似乎以为trim（）方法将更新dateString引用的字符串。但由于字符串是不可改变的，trim（）函数返回一个新字符串值，在这里它是被忽略了。该代码应更正： String dateString = getHeaderField(name); dateString = dateString.trim(); 87.RpC: Repeated conditional tests (RpC\_REPEATED\_CONDITIONAL\_TEST) 该代码包含对同一个条件试验了两次，两边完全一样例如：（如X == 0 | | x == 0）。可能第二次出现是打算判断别的不同条件（如X == 0 | | y== 0）。 88.SA: Double assignment of field (SA\_FIELD\_DOUBLE\_ASSIGNMENT) 方法中的字段包含了双重任务，例如：   int x;   public void foo() {    x = x = 17;   } 这种为变量赋值是无用的，并可能表明一个逻辑错误或拼写错误。 89.SA: Self assignment of field (SA\_FIELD\_SELF\_ASSIGNMENT) 方法中包含自己对自己赋值的字段。例如： int x;   public void foo() {     x = x;   } 90.SA: Self comparison of field with itself (SA\_FIELD\_SELF\_COMPARISON) 字段自己进行自比较可能表明错误或逻辑错误。 91.SA: Self comparison of value with itself (SA\_LOCAL\_SELF\_COMPARISON) 方法中对一个局部变量自身进行比较运算，并可说明错误或逻辑错误。请确保您是比较正确的事情。 92.SA: Nonsensical self computation involving a variable (e.g., x & x) (SA\_LOCAL\_SELF\_COMPUTATION) 此方法对同一变量执行了荒谬的计算（如x&x或x-x）操作。由于计算的性质，这一行动似乎没有意义，并可能表明错误或逻辑错误。 93.SF: Dead store due to switch statement fall through (SF\_DEAD\_STORE\_DUE\_TO\_SWITCH\_FALLTHROUGH) 在swtich中先前的case值因为swtich执行失败而被覆写，这就像是忘记使用break推出或者没有使用return语句放回先前的值一样。 94.SF: Dead store due to switch statement fall through to throw (SF\_DEAD\_STORE\_DUE\_TO\_SWITCH\_FALLTHROUGH\_TO\_THROW) 在swtich中因为出现异常而忽略了对case值的保存。 95.SIC: Deadly embrace of non-static inner class and thread local (SIC\_THREADLOCAL\_DEADLY\_EMBRACE) 如果是一个静态内部类。实际上，在内部类和当前线程有死锁的可能。由于内部类不是静态的，它保留了对外部类的引用。如果线程包含对一个内部类实例的引用，那么内外实例的实例都可以被获取，这样就不具备垃圾会回收的资格。 96.SIO: Unnecessary type check done using instanceof operator (SIO\_SUPERFLUOUS\_INSTANCEOF) 在进行instanceof操作时进行没有必要的类型检查 97.STI: Unneeded use of currentThread() call, to call interrupted() (STI\_INTERRUPTED\_ON\_CURRENTTHREAD) 此方法调用Thread.currentThread（）调用，只需调用interrupted（）方法。由于interrupted（）是一个静态方法， Thread.interrupted（）更简单易用。 98.STI: Static Thread.interrupted() method invoked on thread instance (STI\_INTERRUPTED\_ON\_UNKNOWNTHREAD) 调用不是当前线程对象的Thread.interrupted()方法，由于interrupted（）方法是静态的，interrupted方法将会调用一个和作者原计划不同的对象。 99.Se: Method must be private in order for serialization to work (SE\_METHOD\_MUST\_BE\_PRIVATE) 这个类实现了Serializable接口，并定义自定义序列化的方法/反序列化。但由于这种方法不能声明为private，将被序列化/反序列化的API忽略掉。 100.Se: The readResolve method must not be declared as a static method. (SE\_READ\_RESOLVE\_IS\_STATIC) 为使readResolve方法得到序列化机制的识别，不能作为一个静态方法来声明。 101.UMAC: Uncallable method defined in anonymous class (UMAC\_UNCALLABLE\_METHOD\_OF\_ANONYMOUS\_CLASS) 在匿名类中定义了一个既没有覆写超类中方法也不能直接调用的方法。因为在其他类的方法不能直接引用匿名类声明的方法，似乎这种方法不能被调用，这种方法可能只是没有任何作用的代码，但也可能覆写超类中声明。 102.UR: Uninitialized read of field in constructor (UR\_UNINIT\_READ) 此构造方法中使用了一个尚未赋值的字段或属性。 String a; public SA() { String abc = a; System.out.println(abc); } 103.UR: Uninitialized read of field method called from constructor of superclass (UR\_UNINIT\_READ\_CALLED\_FROM\_SUPER\_CONSTRUCTOR) 方法被超类的构造函数调用时，在当前类中的字段或属性还没有被初始化。例如： abstract class A {   int hashCode;   abstract Object getValue();   A() {     hashCode = getValue().hashCode();     }   } class B extends A {   Object value;   B(Object v) {     this.value = v;     }   Object getValue() {     return value;   }   } 当B是创建时，A的构造函数将在B为value赋值之前触发，然而在A的初始化方法调用getValue方法时value这个变量还没有被初始化。 104.USELESS\_STRING: Invocation of toString on an array (DMI\_INVOKING\_TOSTRING\_ON\_ANONYMOUS\_ARRAY) 该代码调用上匿名数组的toString（）方法，产生的结果形如[@ 16f0472并没有实际的意义。考虑使用Arrays.toString方法来转换成可读的字符串，提供该数组的内容数组。例如： String[] a = { "a" }; System.out.println(a.toString()); //正确的使用为 System.out.println(Arrays.toString(a)); 105.USELESS\_STRING: Invocation of toString on an array (DMI\_INVOKING\_TOSTRING\_ON\_ARRAY) 该代码调用上数组的toString（）方法，产生的结果形如[@ 16f0472并不能显示数组的真实内容。考虑使用Arrays.toString方法来转换成可读的字符串，提供该数组的内容数组 106.UwF: Field only ever set to null (UWF\_NULL\_FIELD) 字段的值总是为null值，所有读取该字段的值都为null。检查错误，如果它确实没有用就删除掉。 107.UwF: Unwritten field (UWF\_UNWRITTEN\_FIELD 此字段是永远不会写入值。所有读取将返回默认值。检查错误（如果它被初始化？），如果它确实没有用就删除掉。

**五：Performance关于代码性能相关方面的**

1.Bx: Primitive value is boxed and then immediately unboxed (BX\_BOXING\_IMMEDIATELY\_UNBOXED) 对原始值进行装箱，然后立即取消装箱。这可能是在一个未要求装箱的地方进行了手动装箱，从而迫使编译器进行立即撤消装箱的操作 2.Bx: Primitive value is boxed then unboxed to perform primitive coercion (BX\_BOXING\_IMMEDIATELY\_UNBOXED\_TO\_PERFORM\_COERCION) 对原始值进行装箱然后立即把它强制转换为另外一种原始类型。例如： new Double(d).intValue()应该直接进行强制转换例如：(int) d 3.Bx: Method allocates a boxed primitive just to call toString (DM\_BOXED\_PRIMITIVE\_TOSTRING) 仅仅为了调用封装类的toString()而对原始类型进行封装操作。比这种方法更有效的是调用封装类的toString(…)方法例如： new Integer(1).toString()    替换为   Integer.toString(1) new Long(1).toString()    替换为   Long.toString(1)  new Float(1.0).toString()    替换为   Float.toString(1.0)  new Double(1.0).toString()    替换为   Double.toString(1.0)  new Byte(1).toString()    替换为   Byte.toString(1)  new Short(1).toString()    替换为   Short.toString(1)  new Boolean(true).toString()    替换为   Boolean.toString(true) 4.Bx: Method invokes inefficient floating-point Number constructor; use static valueOf instead (DM\_FP\_NUMBER\_CTOR) 使用new Double(double)方法总是会创建一个新的对象，然而使用Double.valueOf(double)方法可以把值保存在编辑器或者class library、JVM中。使用存储值的方式来避免对象的分配可以或得更好的代码性能 除非类必须符合Java 1.5以前的JVM，否则请使用自动装箱或valueOf（）方法创建Double和Float实例。 5.Bx: Method invokes inefficient Number constructor; use static valueOf instead (DM\_NUMBER\_CTOR) 使用new Integer(int)方法总是会创建一个新的对象，然而使用Integer.valueOf(int)方法可以把值保存在编辑器或者class library、JVM中。使用存储值的方式来避免对象的分配可以或得更好的代码性能 除非类必须符合Java 1.5以前的JVM，否则请使用自动装箱或valueOf（）方法创建Long, Integer, Short, Character, Byte实例。 6.Dm: The equals and hashCode methods of URL are blocking (DMI\_BLOCKING\_METHODS\_ON\_URL) 使用equals和hashCode方法来对url进行资源标识符解析时会引起堵塞。考虑使用java.net.URI来代替。 7.Dm: Maps and sets of URLs can be performance hogs (DMI\_COLLECTION\_OF\_URLS) 方法或者字段使用url的map/set集合。因为equals方法或者hashCode方法来进行资源标识符解析时都会引起堵塞。考虑使用java.net.URI来代替。 8.Dm: Method invokes inefficient Boolean constructor; use Boolean.valueOf(...) instead (DM\_BOOLEAN\_CTOR) 使用new方法创建一个java.lang.Boolean类型能够的实例对象是浪费空间的，因为Boolean对象是不可变的而且只有两个有用的值。使用Boolean.valueOf()或者Java1.5中的自动装箱功能来创建一个Boolean实例。 9.Dm: Explicit garbage collection; extremely dubious except in benchmarking code (DM\_GC) 在代码中显式的调用垃圾回收命名，这样做并不能起作用。在过去，有人在关闭操作或者finalize方法中调用垃圾回收方法导致了很多的性能浪费。这样大规模回收对象时会造成处理器运行缓慢。 10.Dm: Use the nextInt method of Random rather than nextDouble to generate a random integer (DM\_NEXTINT\_VIA\_NEXTDOUBLE)  如果r是一个java.util.Random对象，你可以使r.nextInt(n)生成一个0到n-1之前的随机数，而不是使用(int)(r.nextDouble() \* n) 11.Dm: Method invokes inefficient new String(String) constructor (DM\_STRING\_CTOR) 使用java.lang.String(String)构造函数会浪费内存因为这种构造方式和String作为参数在功能上容易混乱。只是使用String直接作为参数的形式 12.Dm: Method invokes toString() method on a String (DM\_STRING\_TOSTRING) 调用String.toString()是多余的操作，只要使用String就可以了。 13.Dm: Method invokes inefficient new String() constructor (DM\_STRING\_VOID\_CTOR) 使用没有参数的构造方法去创建新的String对象是浪费内存空间的，因为这样创建会和空字符串“”混淆。Java中保证完成相同的构造方法会产生描绘相同的String对象。所以你只要使用空字符串来创建就可以了。 14.ITA: Method uses toArray() with zero-length array argument (ITA\_INEFFICIENT\_TO\_ARRAY) 当使用集合的toArray()方法时使用数组长度为0的数组作为参数。比这更有效的一种方法是 myCollection.toArray(new Foo[myCollection.size()])，如果数组的长度足够大就可以直接把集合中的内容包装到数组中直接返回从而避免了第二次创建一个新的数组来存放集合中值。 15.SBSC: Method concatenates strings using + in a loop (SBSC\_USE\_STRINGBUFFER\_CONCATENATION) 在循环中构建一个String对象时从性能上讲使用StringBuffer来代替String对象 例如： // This is bad   String s = "";   for (int i = 0; i < field.length; ++i) {     s = s + field[i];   }   // This is better   StringBuffer buf = new StringBuffer();   for (int i = 0; i < field.length; ++i) {     buf.append(field[i]);   }   String s = buf.toString(); 16.SS: Unread field: should this field be static? (SS\_SHOULD\_BE\_STATIC) 类中所包含的final属性字段在编译器中初始化为静态的值。考虑在定义时就把它定义为static类型的。 17.UM: Method calls static Math class method on a constant value (UM\_UNNECESSARY\_MATH) 在方法中使用了java.lang.Math的静态方法代替常量来使用，使用常量速度和准确度会更好。 以下为Math中的方法产生的值。 Method Parameter  abs -any-  acos 0.0 or 1.0  asin 0.0 or 1.0  atan 0.0 or 1.0  atan2 0.0 cbrt 0.0 or 1.0  ceil -any-  cos 0.0  cosh 0.0  exp 0.0 or 1.0  expm1 0.0  floor -any-  log 0.0 or 1.0  log10 0.0 or 1.0  rint -any-  round -any-  sin 0.0  sinh 0.0  sqrt 0.0 or 1.0  tan 0.0  tanh 0.0  toDegrees 0.0 or 1.0  toRadians 0.0 18.UPM: Private method is never called (UPM\_UNCALLED\_PRIVATE\_METHOD) 定义为Private类型方法从未被调用，应该被删除。 19.UrF: Unread field (URF\_UNREAD\_FIELD) 类中定义的属性从未被调用，建议删除。 20.UuF: Unused field (UUF\_UNUSED\_FIELD) 类中定义的属性从未被使用，建议删除。 21.WMI: Inefficient use of keySet iterator instead of entrySet iterator (WMI\_WRONG\_MAP\_ITERATOR) 当方法中接受一个Map类型的参数时，使用keySet的迭代器比使用entrySet的迭代器效率要高。

**六：Internationalization关于代码国际化相关方面的**

Dm: Consider using Locale parameterized version of invoked method (DM\_CONVERT\_CASE) 使用平台默认的编码格式对字符串进行大小写转换，这可能导致国际字符的转换不当。使用以下方式对字符进行转换 String.toUpperCase( Locale l ) String.toLowerCase( Locale l )

**七：Multithreaded correctness关于代码多线程正确性相关方面的**

1.DL: Synchronization on Boolean could lead to deadlock (DL\_SYNCHRONIZATION\_ON\_BOOLEAN) 该代码同步一个封装的原始常量，例如一个Boolean类型。 private static Boolean inited = Boolean.FALSE; ...   synchronized(inited) {      if (!inited) {        init();        inited = Boolean.TRUE;        }      } ... 由于通常只存在两个布尔对象，此代码可能是同步的其他无关的代码中相同的对象，这时会导致反应迟钝和可能死锁 2.DL: Synchronization on boxed primitive could lead to deadlock (DL\_SYNCHRONIZATION\_ON\_BOXED\_PRIMITIVE) 该代码同步一个封装的原始常量，例如一个Integer类型。 private static Integer count = 0; ...   synchronized(count) {       count++;      } ... 由于Integer对象可以共享和保存，此代码可能是同步的其他无关的代码中相同的对象，这时会导致反应迟钝和可能死锁 3.DL: Synchronization on interned String could lead to deadlock (DL\_SYNCHRONIZATION\_ON\_SHARED\_CONSTANT) 同步String类型的常量时，由于它被JVM中多个其他的对象所共有，这样在其他代码中会引起死锁。 4.DL: Synchronization on boxed primitive values (DL\_SYNCHRONIZATION\_ON\_UNSHARED\_BOXED\_PRIMITIVE) 同步一个显然不是共有封装的原始值，例如一个Integer类型的对象。例如： private static final Integer fileLock = new Integer(1); ...   synchronized(fileLock) {       .. do something ..      } ... 它最后被定义为以下方式来代替：private static final Object fileLock = new Object(); 5.Dm: Monitor wait() called on Condition (DM\_MONITOR\_WAIT\_ON\_CONDITION) 方法中以java.util.concurrent.locks.Condition对象调用wait（）。等待一个条件发生时应该使用在Condition接口中定义的await()方法。 6.Dm: A thread was created using the default empty run method (DM\_USELESS\_THREAD) 这个方法没有通过run方法或者具体声明Thread类，也没有通过一个Runnable对象去定义一个线程，而这个线程出来浪费资源却什么也没有去做。 7.ESync: Empty synchronized block (ESync\_EMPTY\_SYNC) 该代码包含一个空的同步块：synchronized() {} 8.IS: Inconsistent synchronization (IS2\_INCONSISTENT\_SYNC) 不合理的同步 9.IS: Field not guarded against concurrent access (IS\_FIELD\_NOT\_GUARDED) 域不是良好的同步访问--- 此字段被标注为net.jcip.annotations.GuardedBy，但可以在某种程度上违反注释而去访问 10.JLM: Synchronization performed on Lock (JLM\_JSR166\_LOCK\_MONITORENTER) 实现java.util.concurrent.locks.Lock的对象调用了同步的方法。应该这样处理，对象被锁定/解锁时使用acquire（）/ release（）方法而不是使用同步的方法。 11.LI: Incorrect lazy initialization of static field (LI\_LAZY\_INIT\_STATIC) 静态域不正确的延迟初始化-- 这种方法包含了一个不同步延迟初始化的非volatile静态字段。因为编译器或处理器可能会重新排列指令，如果该方法可以被多个线程调用，线程不能保证看到一个完全初始化的对象。你可以让字段可变来解决此问题 12.LI: Incorrect lazy initialization and update of static field (LI\_LAZY\_INIT\_UPDATE\_STATIC) 这种方法包含一个不同步延迟初始化的静态字段。之后为字段赋值，对象存储到该位置后进一步更新或访问。字段后尽快让其他线程能够访问。如果该方法的进一步访问该字段为初始化对象提供服务，然后你有一个非常严重的多线程bug，除非别的东西阻止任何其他线程访问存储的对象，直到它完全初始化。 即使你有信心，该方法是永远不会被多个线程调用时，在它的值还没有被充分初始化或移动，不把它设定为static字段时它可能会更好。 13.ML: Method synchronizes on an updated field (ML\_SYNC\_ON\_UPDATED\_FIELD) 对象获取一个可变字段时进行同步。这是没有意义的，因为不同的线程可以在不同的对象同步。 14.MSF: Mutable servlet field (MSF\_MUTABLE\_SERVLET\_FIELD) 一个web服务一般只能创建一个servlet或者jsp的实例（例如：treates是一个单利类），它会被多个线程调用这个实例的方法服务于多个同时的请求。因此使用易变的字段属性产生竞争的情况。 15.MWN: Mismatched notify() (MWN\_MISMATCHED\_NOTIFY) 此方法调用Object.notify（）或Object.notifyAll（）而没有获取到该对象的对象锁。调用notify（）或notifyAll（）而没有持有该对象的对象锁，将导致IllegalMonitorStateException异常。 16.MWN: Mismatched wait() (MWN\_MISMATCHED\_WAIT) 此方法调用Object.wait()而没有获取到该对象的对象锁。调用wait（）而没有持有该对象的对象锁，将导致IllegalMonitorStateException异常。 17.NP: Synchronize and null check on the same field. (NP\_SYNC\_AND\_NULL\_CHECK\_FIELD) 如果代码块是同步的，那么久不可能为空。如果是空，同步时就会抛出NullPointerException异常。最好是在另一个代码块中进行同步。 18.No: Using notify() rather than notifyAll() (NO\_NOTIFY\_NOT\_NOTIFYALL) 调用notify（）而不是notifyAll（）方法。 Java的监控器通常用于多个条件。调用notify（）只唤醒一个线程，这意味着该线程被唤醒只是满足的当前的唯一条件。 19.RS: Class's readObject() method is synchronized (RS\_READOBJECT\_SYNC) 序列化类中定义了同步的readObject（）。通过定义，反序列化创建的对象只有一个线程可以访问，因此没有必要的readObject（）进行同步。如果的readObject（）方法本身造成对象对另一个线程可见，那么这本身就是不好的编码方式。 20.Ru: Invokes run on a thread (did you mean to start it instead?) (RU\_INVOKE\_RUN) 这种方法显式调用一个对象的run（）。一般来说，类是实现Runnable接口的，因为在一个新的线程他们将有自己的run（）方法，在这种情况下Thread.start（）方法调用是正确的。 21.SC: Constructor invokes Thread.start() (SC\_START\_IN\_CTOR) 在构造函数中启动一个线程。如果类曾经被子类扩展过，那么这很可能是错的，因为线程将在子类构造之前开始启动。 22.SP: Method spins on field (SP\_SPIN\_ON\_FIELD) 方法无限循环读取一个字段。编译器可合法悬挂宣读循环，变成一个无限循环的代码。这个类应该改变，所以使用适当的同步（包括等待和通知要求） 23.STCAL: Call to static Calendar (STCAL\_INVOKE\_ON\_STATIC\_CALENDAR\_INSTANCE) 即使JavaDoc对此不包含暗示，而Calendars本身在多线程中使用就是不安全的。探测器发现当调用Calendars的实例时将会获得一个静态对象。 Calendar rightNow = Calendar.getInstance(); 24.STCAL: Call to static DateFormat (STCAL\_INVOKE\_ON\_STATIC\_DATE\_FORMAT\_INSTANCE) 在官方的JavaDoc，DateFormats多线程使用本事就是不安全的。探测器发现调用一个DateFormat的实例将会获得一个静态对象。 myString = DateFormat.getDateInstance().format(myDate); 25.STCAL: Static Calendar (STCAL\_STATIC\_CALENDAR\_INSTANCE) Calendar在多线程中本身就是不安全的，如果在线程范围中共享一个Calendarde 实例而不使用一个同步的方法在应用中就会出现一些奇怪的行为。在sun.util.calendar.BaseCalendar.getCalendarDateFromFixedDate()中会抛出ArrayIndexOutOfBoundsExceptions or IndexOutOfBoundsExceptions异常。 26.STCAL: Static DateFormat (STCAL\_STATIC\_SIMPLE\_DATE\_FORMAT\_INSTANCE) DateFormat 在多线程中本身就是不安全的，如果在线程范围中共享一个DateFormat的实例而不使用一个同步的方法在应用中就会出现一些奇怪的行为。 27.SWL: Method calls Thread.sleep() with a lock held (SWL\_SLEEP\_WITH\_LOCK\_HELD) 当持有对象时调用Thread.sleep（）。这可能会导致很差的性能和可扩展性，或陷入死锁，因为其他线程可能正在等待获得锁。调用wait（）是一个更好的主意，释放对象的持有以允许其他线程运行。 28.UG: Unsynchronized get method, synchronized set method (UG\_SYNC\_SET\_UNSYNC\_GET) 这个类包含类似命名的get和set方法。在set方法是同步方法和get方法是非同步方法。这可能会导致在运行时的不正确行为，因为调用的get方法不一定返回对象一致状态。 GET方法应该同步。 29.UL: Method does not release lock on all paths (UL\_UNRELEASED\_LOCK) 方法获得了当前的对象所，但是在方法中始终没有释放它。一个正确的示例如下：   Lock l = ...;     l.lock();     try {         // do something     } finally {         l.unlock();     } 30.UL: Method does not release lock on all exception paths (UL\_UNRELEASED\_LOCK\_EXCEPTION\_PATH) 方法获得了当前的对象所，但是在所有的异常处理中始终没有释放它。一个正确的示例如下：   Lock l = ...;     l.lock();     try {         // do something     } finally {         l.unlock();     } 31.UW: Unconditional wait (UW\_UNCOND\_WAIT) 方法中包含调用java.lang.Object.wait（），而却没有放到条件流程控制中。该代码应确认条件尚未满足之前等待;先前任何通知将被忽略。 32.VO: A volatile reference to an array doesn't treat the array elements as volatile (VO\_VOLATILE\_REFERENCE\_TO\_ARRAY) 声明一个变量引用数组，这可能不是你想要的。如果一个变量引用数组，那么对引用数组的读和写都是不安全的，但是数组元素不是变量。取得数组的变量值你可以使用java.util.concurrent包中的数组的原子性特性 33.WL: Sychronization on getClass rather than class literal (WL\_USING\_GETCLASS\_RATHER\_THAN\_CLASS\_LITERAL) 实例的方法中同步this.getClass()，如果这个类有子类集合，那么子类集合中的对象将会在这个类的各个子类上进行同步，这不是我们想要的效果，我们只要同步当前的类对象而不包含它的所有子类，可以同步类名.getClass()。例如，java.awt.Label的代码：      private static final String base = "label";      private static int nameCounter = 0;      String constructComponentName() {         synchronized (getClass()) {             return base + nameCounter++;         }      } Label中的子类集合不可能在同一个子对象上进行同步，替换上面的方法为：     private static final String base = "label";      private static int nameCounter = 0;      String constructComponentName() {         synchronized (Label.class) {             return base + nameCounter++;         }      } 34.WS: Class's writeObject() method is synchronized but nothing else is (WS\_WRITEOBJECT\_SYNC) 这个类有一个writeObject（）方法是同步的，但是这个类中没有其他的同步方法。 35.Wa: Condition.await() not in loop (WA\_AWAIT\_NOT\_IN\_LOOP) 方法没有在循环中调用java.util.concurrent.await()。如果对象是用于多种条件，打算调用wait()方法的条件可能不是实际发生的。 36.Wa: Wait not in loop (WA\_NOT\_IN\_LOOP) 这种方法包含调用java.lang.Object.wait（），而这并不是一个循环。如果监视器用于多个条件，打算调用wait()方法的条件可能不是实际发生的。

**八：Malicious codevulnerability关于恶意破坏代码相关方面的**

1.EI: May expose internal representation by returning reference to mutable object (EI\_EXPOSE\_REP) 返回一个易变对象引用并把它保存在对象字段中时会暴露对象内部的字段描述，如果接受不守信任的代码访问或者没有检查就去改变易变对象的会涉及对象的安全和其他重要属性的安全。返回一个对象的新副本，在很多情况下更好的办法。 2.EI2: May expose internal representation by incorporating reference to mutable object (EI\_EXPOSE\_REP2) 此代码把外部可变对象引用存储到对象的内部表示。如果实例受到不信任的代码的访问和没有检查的变化危及对象和重要属性的安全。存储一个对象的副本，在很多情况下是更好的办法。 3.FI: Finalizer should be protected, not public (FI\_PUBLIC\_SHOULD\_BE\_PROTECTED) 一个类中的finalize（）方法必须声明为protected，而不能为public类型 4.MS: Public static method may expose internal representation by returning array (MS\_EXPOSE\_REP) 一个public类型的静态方法返回一个数组，可能引用内部属性的暴露。任何代码调用此方法都可以自由修改底层数组。一个解决办法是返回一个数组的副本。 5.MS: Field should be both final and package protected (MS\_FINAL\_PKGPROTECT) 一个静态字段可能被恶意代码或另外一个包所改变的。字段可以放到protected包中也可以定义为final类型的以避免此问题。 6.MS: Field is a mutable array (MS\_MUTABLE\_ARRAY) 一个定义为final类型的静态字段引用一个数组时它可以被恶意代码或在另其他包中所使用。这些代码可以自由修改数组的内容。 7.MS: Field is a mutable Hashtable (MS\_MUTABLE\_HASHTABLE) 一个定义为final类型的静态字段引用一个Hashtable时可以被恶意代码或者在其他包中被调用，这些方法可以修改Hashtable的值。 8.MS: Field should be moved out of an interface and made package protected (MS\_OOI\_PKGPROTECT) 将域尽量不要定义在接口中，并声明为包保护 在接口中定义了一个final类型的静态字段，如数组或哈希表等易变对象。这些对象可以被恶意代码或者在其他包中被调用，为了解决这个问题，需要把它定义到一个具体的实体类中并且声明为保护类型以避免这种错误。 9.MS: Field should be package protected (MS\_PKGPROTECT) 一个静态字段是可以改变的恶意代码或其他的包访问修改。可以把这种类型的字段声明为final类型的以防止这种错误。

**十：Dodgy关于代码运行期安全方面的**

1.XSS: Servlet reflected cross site scripting vulnerability (XSS\_REQUEST\_PARAMETER\_TO\_SEND\_ERROR) 在代码中在Servlet输出中直接写入一个HTTP参数，这会造成一个跨站点的脚本漏洞。 2.XSS: Servlet reflected cross site scripting vulnerability (XSS\_REQUEST\_PARAMETER\_TO\_SERVLET\_WRITER) 代码直接写入参数的HTTP服务器错误页（使用HttpServletResponse.sendError）。表达了类似的不受信任的输入会引起跨站点脚本漏洞。 3.BC: Questionable cast to abstract collection (BC\_BAD\_CAST\_TO\_ABSTRACT\_COLLECTION) 在代码投把一个集合强制类型转换为一个抽象的集合（如list，set或map）。保证该对象类型和将要转换的类型是一致的。如果你只是想要便利一个集合，那么你就不必将它转换为Set或List。 4.BC: Questionable cast to concrete collection (BC\_BAD\_CAST\_TO\_CONCRETE\_COLLECTION) 代码把抽象的集合（如List，Set，或Collection）强制转换为具体落实类型（如一个ArrayList或HashSet）。这可能不正确，也可能使您的代码很脆弱，因为它使得难以在今后的切换指向其他具体实现。除非你有特别理由这样做，否则只需要使用抽象的集合类。 5.BC: Unchecked/unconfirmed cast (BC\_UNCONFIRMED\_CAST) 强制类型转换操作没有经过验证，而且不是所有的此种类型装换过的类都可以再强制类型转换为原类型。在代码中需要进行逻辑判断以保证可以进行这样的操作。 6.BC: instanceof will always return true (BC\_VACUOUS\_INSTANCEOF) instanceof测试将始终返回真（除非被测试的值为空）。虽然这是安全，确保它是不是说明一些误解或其他一些逻辑错误。如果你真的想测试是空的价值，也许会更清楚这样做的更好空试验，而不是一个instanceof测试。 7.BSHIFT: Unsigned right shift cast to short/byte (ICAST\_QUESTIONABLE\_UNSIGNED\_RIGHT\_SHIFT) 无符号数右移后进行转换为short或者byte类型时可能会丢弃掉高位的值，这样的结果就是有符合数和无符号数无法区分（这取决于移位大小） 8.CI: Class is final but declares protected field (CI\_CONFUSED\_INHERITANCE) 这个类被声明为final的，而是字段属性却声明为保护类型的。由于是final类，它不能再被继承，而再声明为保护类型的很容易造成混淆。为了从外部能正确的使用它应该把它们声明为private或者public类型。 9.DB: Method uses the same code for two branches (DB\_DUPLICATE\_BRANCHES) 此方法使用相同的代码，以实现两个有条件的分支。检查以确保这是不是一个编码错误。 10.DB: Method uses the same code for two switch clauses (DB\_DUPLICATE\_SWITCH\_CLAUSES) 他的方法使用相同的代码来实现两个switch的声明条款。这可能是重复代码的情况，但可能也显示出编码的错误。 11.DLS: Dead store to local variable (DLS\_DEAD\_LOCAL\_STORE) 该指令为局部变量赋值，但在其后的没有对她做任何使用。通常，这表明一个错误，因为值从未使用过。 12.DLS: Useless assignment in return statement (DLS\_DEAD\_LOCAL\_STORE\_IN\_RETURN) 本声明把一个局部变量放到方法的返回语句中。这对于方法中局部变量来说是没有意思的。 13.DLS: Dead store of null to local variable (DLS\_DEAD\_LOCAL\_STORE\_OF\_NULL) 把一个本地变量赋值为null值，并且再也没有对这个变量做任何的操作。这样可能是为了垃圾回收，而是Java SE 6.0，这已不再需要。 14.DMI: Code contains a hard coded reference to an absolute pathname (DMI\_HARDCODED\_ABSOLUTE\_FILENAME) 此代码包含文件对象为一个绝对路径名（例如，新的文件（“/ home/dannyc/workspace/j2ee/src/share/com/sun/enterprise/deployment”）; 15.DMI: Non serializable object written to ObjectOutput (DMI\_NONSERIALIZABLE\_OBJECT\_WRITTEN) 代码中让一个非序列化的对象出现在ObjectOutput.writeObject()方法中，这样会引起一个错误。 16.DMI: Invocation of substring(0), which returns the original value (DMI\_USELESS\_SUBSTRING) 此代码调用了subString(0)方法，它将返回原来的值。 17.Eq: Class doesn't override equals in superclass (EQ\_DOESNT\_OVERRIDE\_EQUALS) 子类定义了一个新的equals方法但是却不是覆写了父类本省的equals()方法。 18.FE: Test for floating point equality (FE\_FLOATING\_POINT\_EQUALITY) 此操作比较两个浮点值是否相等。由于浮点运算可能会涉及到舍入，计算float和double值可能不准确。如果要求值必须准确，如货币值，可以考虑使用固定精度类型，如BigDecimal类型的值来比较 19.FS: Non-Boolean argument formatted using %b format specifier (VA\_FORMAT\_STRING\_BAD\_CONVERSION\_TO\_BOOLEAN) 使用%b去格式化Boolean类型的值不正确的但是它不会抛出异常，任何非空的值都会输出true，任何为空的值都会输出false 20.IC: Initialization circularity (IC\_INIT\_CIRCULARITY) 在引用两个相互调用为环状static方法去初始化一个实例时是错误的。 21.ICAST: integral division result cast to double or float (ICAST\_IDIV\_CAST\_TO\_DOUBLE) 整形数除法强制转换为double或者float类型。 int x = 2; int y = 5; // Wrong: yields result 0.0 double value1 =  x / y; // Right: yields result 0.4 double value2 =  x / (double) y; 22.ICAST: Result of integer multiplication cast to long (ICAST\_INTEGER\_MULTIPLY\_CAST\_TO\_LONG) 整形数做乘法运算结果转换为long值时如果采用 long convertDaysToMilliseconds(int days) { return 1000\*3600\*24\*days; } 结果会因为超出整形的范围而出错。 如果使用： long convertDaysToMilliseconds(int days) { return 1000L\*3600\*24\*days; }  或者： static final long MILLISECONDS\_PER\_DAY = 24L\*3600\*1000; long convertDaysToMilliseconds(int days) { return days \* MILLISECONDS\_PER\_DAY; }  都可以避免此问题。 23.IM: Computation of average could overflow (IM\_AVERAGE\_COMPUTATION\_COULD\_OVERFLOW) 代码中使用x % 2 == 1的方法去验证运算是否存在余数的情况，但是如果出现负数的情况就不起作用了。使用x & 1 == 1, or x % 2 != 0来代替 24.INT: Vacuous comparison of integer value (INT\_VACUOUS\_COMPARISON) 整形数进行比较结果总是不变。例如：x <= Integer.MAX\_VALUE 25.MTIA: Class extends Servlet class and uses instance variables (MTIA\_SUSPECT\_SERVLET\_INSTANCE\_FIELD) 这个类扩展从Servlet类，并使用实例的成员变量。由于只有一个Servlet类的实例，并在多线程方式使用，这种模式有可能存在问题。考虑只使用方法的局部变量。 26.MTIA: Class extends Struts Action class and uses instance variables (MTIA\_SUSPECT\_STRUTS\_INSTANCE\_FIELD) 类扩展自Struts的Action类并使用这个实例的成员变量，因为在Struts框架中只存在一个Action实例对象并且使用在多线程的情况下很可能会出现问题。 27.NP: Dereference of the result of readLine() without nullcheck (NP\_DEREFERENCE\_OF\_READLINE\_VALUE) 对readLine()的结果值没有进行判空操作就去重新赋值，这样的操作可以会抛出空指针异常。 28.NP: Immediate dereference of the result of readLine() (NP\_IMMEDIATE\_DEREFERENCE\_OF\_READLINE) 对readLine()的结果立即赋值，这样的操作可以会抛出空指针异常。 29.NP: Possible null pointer dereference due to return value of called method (NP\_NULL\_ON\_SOME\_PATH\_FROM\_RETURN\_VALUE) 方法的返回值没有进行是否为空的检查就重新赋值，这样可能会出现空指针异常。 30.NP: Parameter must be nonnull but is marked as nullable (NP\_PARAMETER\_MUST\_BE\_NONNULL\_BUT\_MARKED\_AS\_NULLABLE) 参数值在任何情况下都不能为空，但是有明确的注释它可以为空。 31.NS: Potentially dangerous use of non-short-circuit logic (NS\_DANGEROUS\_NON\_SHORT\_CIRCUIT) 代码中使用（& or |）代替（&& or ||）操作，这会造成潜在的危险。 32.NS: Questionable use of non-short-circuit logic (NS\_NON\_SHORT\_CIRCUIT) 代码中使用（& or |）代替（&& or ||）操作，会引起不安全的操作 33.PZLA: Consider returning a zero length array rather than null (PZLA\_PREFER\_ZERO\_LENGTH\_ARRAYS) 考虑返回一个零长度的数组，而不是null值 34.QF: Complicated, subtle or wrong increment in for-loop (QF\_QUESTIONABLE\_FOR\_LOOP) 确定这个循环是正确的变量递增，看起来，另一个变量被初始化，检查的循环。这是由于for循环中太复杂的定义造成的。 35.RCN: Redundant comparison of non-null value to null (RCN\_REDUNDANT\_COMPARISON\_OF\_NULL\_AND\_NONNULL\_VALUE) 方法中包含一个不能为空的赋值还包含一个可以为空的赋值。冗余比较非空值为空。 36.RCN: Redundant comparison of two null values (RCN\_REDUNDANT\_COMPARISON\_TWO\_NULL\_VALUES) 方法中对两个null值进行比较 37.RCN: Redundant nullcheck of value known to be non-null (RCN\_REDUNDANT\_NULLCHECK\_OF\_NONNULL\_VALUE) 方法中对不为空的值进行为空的判断。 38.REC: Exception is caught when Exception is not thrown (REC\_CATCH\_EXCEPTION) 在try/catch块中捕获异常，但是异常没有在try语句中抛出而RuntimeException又没有明确的被捕获 39.RI: Class implements same interface as superclass (RI\_REDUNDANT\_INTERFACES) 子类和父类都实现了同一个接口，这种定义是多余的。 40.RV: Method discards result of readLine after checking if it is nonnull (RV\_DONT\_JUST\_NULL\_CHECK\_READLINE) readLine方法的结果不为空时被抛弃 41.RV: Remainder of 32-bit signed random integer (RV\_REM\_OF\_RANDOM\_INT) 此代码生成一个随机的符号整数，然后计算另一个值的。由于随机数可以是负数，所以其余操作的结果也可以是负面的。考虑使用Random.nextInt（int）方法代替。 42.SA: Double assignment of local variable (SA\_LOCAL\_DOUBLE\_ASSIGNMENT) 为一个局部变量两次赋值，这样是没有意义的。例如： public void foo() {     int x,y;     x = x = 17;   } 43.SA: Self assignment of local variable (SA\_LOCAL\_SELF\_ASSIGNMENT) 局部变量使用自身给自己赋值 public void foo() {     int x = 3;     x = x;   } 44.SF: Switch statement found where one case falls through to the next case (SF\_SWITCH\_FALLTHROUGH) Switch语句中一个分支执行后又执行了下一个分支。通常case后面要跟break 或者return语句来跳出。 45.SF: Switch statement found where default case is missing (SF\_SWITCH\_NO\_DEFAULT) Switch没有默认情况下执行的case语句。 46.Se: private readResolve method not inherited by subclasses (SE\_PRIVATE\_READ\_RESOLVE\_NOT\_INHERITED) 声明为private的序列化方法被子类继承 47.UCF: Useless control flow (UCF\_USELESS\_CONTROL\_FLOW) 没有任何作用的条件语句。 if (argv.length == 0) { // TODO: handle this case } 48.UCF: Useless control flow to next line (UCF\_USELESS\_CONTROL\_FLOW\_NEXT\_LINE) 无效的条件控制语句，注意if (argv.length == 1);以“;”结尾，下面的语句无论是否满足都会运行。 if (argv.length == 1);         System.out.println("Hello, " + argv[0]); 49.UwF: Field not initialized in constructor (UWF\_FIELD\_NOT\_INITIALIZED\_IN\_CONSTRUCTOR) 字段从来没有在任何构造函数初始化，对象被创建后值为空。如果该字段未被定义就重新赋值会产生一个空指针异常。 50.XFB: Method directly allocates a specific implementation of xml interfaces (XFB\_XML\_FACTORY\_BYPASS) 方法自定义了一种XML接口的实现类。最好是使用官方提供的工厂类来创建这些对象，以便可以在运行期中改变。例如： javax.xml.parsers.DocumentBuilderFactory javax.xml.parsers.SAXParserFactory javax.xml.transform.TransformerFactory org.w3c.dom.Document.createXXXX