Java 代码检测工具 PMD 规则集解决方案合集

Basic Rules : The Basic Ruleset contains a collection of good practices which everyone should follow.

- Braces Rules : The Braces Ruleset contains a collection of braces rules.
- Clone Implementation Rules : The Clone Implementation ruleset contains a collection of rules that find questionable usages of the clone() method.
- Code Size Rules : The Code Size Ruleset contains a collection of rules that find code size related problems.
- Controversial Rules : The Controversial Ruleset contains rules that, for whatever reason, are considered controversial. They are separated out here to allow people to include as they see fit via custom rulesets. This ruleset was initially created in response to discussions over UnnecessaryConstructorRule which Tom likes but most people really dislike :-)
- Coupling Rules : These are rules which find instances of high or inappropriate coupling between objects and packages.
- Design Rules : The Design Ruleset contains a collection of rules that find questionable designs.
- Finalizer Rules : These rules deal with different problems that can occur with finalizers.
- Import Statement Rules : These rules deal with different problems that can occur with a class' import statements.
- J2EE Rules : These are rules for J2EE
- JavaBean Rules : The JavaBeans Ruleset catches instances of bean rules not being followed.
- JUnit Rules : These rules deal with different problems that can occur with JUnit tests.

- Jakarta Commons Logging Rules : The Jakarta Commons Logging ruleset contains a collection of rules that find questionable usages of that framework.
- Java Logging Rules : The Java Logging ruleset contains a collection of rules that find questionable usages of the logger.
- Migration Rules : Contains rules about migrating from one JDK version to another. Don't use these rules directly, rather, use a wrapper ruleset such as migrating_to_13.xml.
- Naming Rules : The Naming Ruleset contains a collection of rules about names - too long, too short, and so forth.
- Optimization Rules : These rules deal with different optimizations that generally apply to performance best practices.
- Strict Exception Rules : These rules provide some strict guidelines about throwing and catching exceptions.
- String and StringBuffer Rules : These rules deal with different problems that can occur with manipulation of the class String or StringBuffer.
- Security Code Guidelines : These rules check the security guidelines from Sun, published at http://java.sun.com/security/seccodeguide.html#gcg
- Type Resolution Rules : These are rules which resolve java Class files for comparisson, as opposed to a String
- Unused Code Rules : The Unused Code Ruleset contains a collection of rules that find unused code.

*PMD 规则之 Basic Rules(基本规则)

- EmptyCatchBlock: Empty Catch Block finds instances where an exception is caught, but nothing is done. In most circumstances, this swallows an exception which should either be acted on or reported.

解决方案　空的 catch 块：发现空的 catch 块没做任何异常处理的事，在大多数情形下，这会吞噬一些应该被处理或报告的异常

- EmptyIfStmt: Empty If Statement finds instances where a condition is checked but nothing is done about it.

解决方案　空的 if 表达式：发现使用 if 进行了条件判断，但是判断之后没做任何处理

- EmptyWhileStmt: Empty While Statement finds all instances where a while statement does nothing. If it is a timing loop, then you should use Thread.sleep() for it; if it's a while loop that does a lot in the exit expression, rewrite it to make it clearer.

解决方案　空的 while 表达式：发现空的 while 表达式，如果是一个定时的循环，你应该在循环体内使用 Thread.sleep()；如果是对于退出处理做的一个处理大量事情的 while 循环，重写代码使它更清晰

- EmptyTryBlock: Avoid empty try blocks - what's the point?

解决方案 空的 try 块：避免空的 try 块

- EmptyFinallyBlock: Avoid empty finally blocks - these can be deleted.

解决方案 空的 finally 块：避免空的 finally 块 - 这些是可以删掉的

- EmptySwitchStatements: Avoid empty switch statements.

解决方案 空的 switch 表达式：避免空的 switch 表达式

- JumbledIncrementer: Avoid jumbled loop incrementers - it's usually a mistake, and it's confusing even if it's what's intended.

解决方案 避免混乱的循环 **增量** - 它常常是一个错误，而且容易让人迷惑

错误代码示例：

```
public class JumbledIncrementerRule1 {
  public void foo() {
   for (int i = 0; i < 10; i++) {
    for (int k = 0; k < 20; i++) {
     System.out.println("Hello");
    }
   }
  }
  }
```

父子循环都用 i++

- ForLoopShouldBeWhileLoop: Some for loops can be simplified to while loops - this makes them more concise.

解决方案 有些 for 循环可以简化为 while 循环 - 这样可以更加简明

代码示例：

```
public class Foo {
  void bar() {
  for (;true;) true; // 没有初始化块和变化块，相当于 : while (true)
  }
}
```

- UnnecessaryConversionTemporary: Avoid unnecessary temporaries when converting primitives to Strings

解决方案　将原始类型转换为字符串类型时不必要的临时转换

代码示例:

```
public String convert(int x) {
  // 多了一个创建对象环节
  String foo = new Integer(x).toString();
  // 下面这种写法就好了
  return Integer.toString(x);
}
```

- OverrideBothEqualsAndHashcode: Override both public boolean Object.equals(Object other), and public int Object.hashCode(), or override neither. Even if you are inheriting a hashCode() from a parent class, consider implementing hashCode and explicitly delegating to your superclass.

解决方案　同时重写 equals() 和 hashCode() 方法：要么全部重写这两个方法，要么全部不重写

- DoubleCheckedLocking: Partially created objects can be returned by the Double Checked Locking pattern when used in Java. An optimizing JRE may assign a reference to the baz variable before it creates the object the reference is intended to point to. For more details see http://www.javaworld.com/javaworld/jw-02-2001/jw-0209-double.html.

解决方案　双重检查锁机制：在 JAVA 中有时候创建的对象是通过双重检查机制获取的，一个优化的 JRE 可能在真正创建对象之前先将指向这个对象的引用赋给一个变量，如 baz，需要更多细节参考：http://www.javaworld.com/javaworld/jw-02-2001/jw-0209-double.html .

示例代码：

```
public class Foo {

  Object baz;

  Object bar() {

  /** 这里当对象没创建完时也可能判断不为空，那么在多线程环境下，就可能导致某些线程使用 bar() 方法后直接返回一个未指向完整对象的 baz 引用，从而发生错误，也可以参考笔者博客：http://blog.csdn.net/jack0511/archive/2009/02/04/3862382.aspx   */

      if(baz == null) {

      synchronized(this){

        if(baz == null){

          baz = new Object();

        }

      }

     }

    return baz;

  }
```

}

- ReturnFromFinallyBlock: Avoid returning from a finally block - this can discard exceptions.

解决方案　从 finally 块中返回: 避免从 finally 块中返回 - 这会导致异常捕获后又被抛弃 .

```
public class Bar {
  public String foo() {
  try {
   throw new Exception( "My Exception" );
  } catch (Exception e) {
   throw e;
  } finally {
   return "A. O. K."; //. 这句导致 catch 到的异常直接被丢弃，强制返回“ A.O.K ”
  }
  }
}
```

- EmptySynchronizedBlock: Avoid empty synchronized blocks - they're useless.

解决方案　空的 Synchronized 块：避免空的 synchronized 块 - 它们是无用的

}

- UnnecessaryReturn: Avoid unnecessary return statements

解决方案　不必要的 Return ：避免不必要的 return 语句

示例代码：

```
public class Foo {
  public void bar() {
  int x = 42;
  return;
  }
}
```

- EmptyStaticInitializer: An empty static initializer was found.

解决方案　空的静态初始化块：发现一个空的静态初始化块

示例代码：

```
public class Foo {
  static {
  // empty
  }
}
```

- UnconditionalIfStatement: Do not use "if" statements that are always true or always false.

解决方案　非条件化的 if 表达式 : 当表达式总是为真或总为假时，不要用 if

```
public class Foo {
  public void close() {
  if (true) {
      // ...
   }
  }
}
```

- EmptyStatementNotInLoop: An empty statement (aka a semicolon by itself) that is not used as the sole body of a for loop or while loop is probably a bug. It could also be a double semicolon, which is useless and should be removed.

解决方案　非循环中不要有空的表达式：在一个非 for 循环或非 while 循环体中使用的一个空的表达式（或者称为一个分号）可能是一个 bug 。也可能是一对分号，这是无用的需要被移除的

代码示例：

```
public class MyClass {
   public void doit() {
     // 下面只用了一个；号
```

```
        ;
        // 语句结尾用了两个；号
        System.out.println("look at the extra semicolon");;
    }
}
```

- BooleanInstantiation: Avoid instantiating Boolean objects; you can reference Boolean.TRUE, Boolean.FALSE, or call Boolean.valueOf() instead.

解决方案　布尔量实例化：避免实例化一个布尔对象；用指向 Boolean.TRUE,Boolean.FALSE 或 Boolean.valueOf() 的引用代替

- UnnecessaryFinalModifier: When a class has the final modifier, all the methods are automatically final.

解决方案　非必要的 final 修饰符：注意当一个类被 fianl 修饰时，所有这个类的方法就自动变为 final 类型了

- CollapsibleIfStatements: Sometimes two 'if' statements can be consolidated by separating their conditions with a boolean short-circuit operator.

解决方案　分解的 if 表达式：有时候两个 if 语句可以通过布尔短路操作符分隔条件表达式组合成一条语句

- UselessOverridingMethod: The overriding method merely calls the same method defined in a superclass

解决方案　无用的方法重写：重载的方法仅仅调用了父类中定义的同名方法

- ClassCastExceptionWithToArray: if you need to get an array of a class from your Collection, you should pass an array of the desidered class as the parameter of the toArray method. Otherwise you will get a ClassCastException.

解决方案　toArray 时类型转换异常：如果你想从一个枚举类型中得到某个类型的数组，你应该传给 toArray() 方法一个目的类型的数组作为参数，否则你可能得到一个类型转换错误

- AvoidDecimalLiteralsInBigDecimalConstructor: One might assume that "new BigDecimal(.1)" is exactly equal to .1, but it is actually equal to .1000000000000000055511151231257827021181583404541015625. This is so because .1 cannot be represented exactly as a double (or, for that matter, as a binary fraction of any finite length). Thus, the long value that is being passed in to the constructor is not exactly equal to .1, appearances notwithstanding. The (String) constructor, on the other hand, is perfectly predictable: 'new BigDecimal(".1")' is exactly equal to .1, as one would expect. Therefore, it is generally recommended that the (String) constructor be used in preference to this one.

解决方案　避免在 BigDecimal 类型的构造方法中用小数类型的字面量：人们常常以为 "new BigDecimal(0.1)" 能精确等于 0.1，其实不然，它等于 " 0. 1000000000000000055511151231257827021181583404541015625 "，这种状况的原因是 0.1 不能精确的表示双精度类型，因此，传入构造器的 long 类型不等于 0.1 ，而传入 String 类型的构造器 new BigDecimal("0.1") 可以精确等于 0.1，故推荐这种情形时用 String 类型的构造器

- UselessOperationOnImmutable: An operation on an Immutable object (String, BigDecimal or BigInteger) won't change the object itself. The result of the operation is a new object. Therefore, ignoring the operation result is an error.

解决方案　对于不变类型的无用操作：对于不变类型对象（String,BigDecimal 或 BigInteger）的操作不会改变对象本身，但操作结果是产生新的对象，所以，忽略操作的结果是错的

示例代码：

```
import java.math.*;

class Test {

  void method1() {

  BigDecimal bd=new BigDecimal(10);

  bd.add(new BigDecimal(5)); // 这里违背了规则

  }

  void method2() {

  BigDecimal bd=new BigDecimal(10);

  bd = bd.add(new BigDecimal(5)); // 这里没有违背规则

  }

}
```

- MisplacedNullCheck: The null check here is misplaced. if the variable is null you'll get a NullPointerException. Either the check is useless (the variable will never be "null") or it's incorrect.

解决方案　错位的空检查：这里的空检查是放错位置的。如果变量为空你将得到一个空指针异常。可能因为检查是无用的或者是不正确的

```
public class Foo {

  void bar() {

  if (a.equals(baz) && a != null) {}

  }

}
```

```
public class Foo {

  void bar() {

  if (a.equals(baz) || a == null) {}

  }

}
```

- UnusedNullCheckInEquals: After checking an object reference for null, you should invoke equals() on that object rather than passing it to another object's equals() method.

解决方案　使用 equals() 时无用的空检查：在对一个对象引用进行完空检查后，你应该在这个对象上调用 equals() 方法而不是将它传给另一个对象的 equals() 方法作为参数

- AvoidThreadGroup: Avoid using ThreadGroup; although it is intended to be used in a threaded environment it contains methods that are not thread safe.

解决方案　避免线程组：避免使用线程组；虽然线程组可以被用于多线程环境中，但它包含的方法不是线程安全的

```
public class Bar {
    void buz() {
     ThreadGroup tg = new ThreadGroup("My threadgroup") ;
     tg = new ThreadGroup(tg, "my thread group");
     tg = Thread.currentThread().getThreadGroup();
     tg = System.getSecurityManager().getThreadGroup();
    }
   }
```

- BrokenNullCheck: The null check is broken since it will throw a NullPointerException itself. It is likely that you used || instead of && or vice versa.

解决方案　破坏空检查：如果自身抛出空指针异常空检查就会遭到破坏，比如你使用 || 代替 && ，反之亦然。

```
class Foo {
  String bar(String string) {
  // 这里应该是 &&
  if (string!=null || !string.equals(""))
    return string;
  // 这里应该是 ||
  if (string==null && string.equals(""))
    return string;
  }
}
```

- BigIntegerInstantiation: Don't create instances of already existing BigInteger (BigInteger.ZERO, BigInteger.ONE) and for 1.5 on, BigInteger.TEN and BigDecimal (BigDecimal.ZERO, BigDecimal.ONE, BigDecimal.TEN)

解决方案 BigInteger 实例化：不要创建已经存在的 BigInteger 类型的实例，（如 BigInteger.ZERO,BigInteger.ONE ），对于 JDK.1.5 以上， BigInteger.TEN BigDecimal (BigDecimal.ZERO, BigDecimal.ONE, BigDecimal.TEN)

```
public class Test {


  public static void main(String[] args) {
   BigInteger bi=new BigInteger(1);
   BigInteger bi2=new BigInteger("0");
   BigInteger bi3=new BigInteger(0.0);
   BigInteger bi4;
   bi4=new BigInteger(0);
  }
}
```

- AvoidUsingOctalValues: Integer literals should not start with zero. Zero means that the rest of literal will be interpreted as an octal value.

解决方案　避免使用八进制值：整型字面量不要以 0 开头， 0 意味着之后的值要被解释为一个八进制值。

- AvoidUsingHardCodedIP: An application with hard coded IP may become impossible to deploy in some case. It never hurts to externalize IP adresses.

解决方案　避免使用 IP 硬编码：一个应用中的硬编码 IP 将使系统在某些情况下无法发布

- CheckResultSet: Always check the return of one of the navigation method (next,previous,first,last) of a ResultSet. Indeed, if the value return is 'false', the developer should deal with it !

解决方案　检查 ResultSet ：总是需要检查 ResultSet 对象的导航方法（ next,previous,first,last ）的返回 ，事实上，如果返回 false ，开发者需要处理它

```
// This is NOT appropriate !

        Statement stat = conn.createStatement();

        ResultSet rst = stat.executeQuery("SELECT name FROM person");

        rst.next(); // what if it returns a 'false' ?

        String firstName = rst.getString(1);



        // This is appropriate...
```

```
Statement stat = conn.createStatement();
 ResultSet rst = stat.executeQuery("SELECT name FROM person");
if (rst.next())
{
   String firstName = rst.getString(1);
}
else
{
   // here you deal with the error ( at least log it)
}
```

- AvoidMultipleUnaryOperators: Using multiple unary operators may be a bug, and/or is confusing. Check the usage is not a bug, or consider simplifying the expression.

解决方案　避免使用多重的一元运算符：使用多重的一元运算符可能是一个 bug ，并且可能令人迷惑。检查确保你的用法不是一个 bug ，或者考虑简化表达

```
// These are typo bugs, or at best needlessly complex and confusing:
          int i = - -1;
          int j = + - +1;
          int z = ~~2;
```

```
            boolean b = !!true;

            boolean c = !!!true;



            // These are better:

            int i = 1;

            int j = -1;

            int z = 2;

            boolean b = true;

            boolean c = false;



            // And these just make your brain hurt:

            int i = ~-2;

            int j = -~7;
```

- EmptyInitializer: An empty initializer was found.

解决方案　　空的初始化块：发现空的初始化块

```
public class Foo {



   static {} // Why ?
```

```
    {} // Again, why ?

}
```

PMD 规则之 Braces Rules

- IfStmtsMustUseBraces: Avoid using if statements without using curly braces.

解决方案　if 块必须用括号：避免使用 if 块时不使用花括号{}

- WhileLoopsMustUseBraces: Avoid using 'while' statements without using curly braces.

解决方案　while 循环必须使用括号：避免使用 while 块时不使用{}

- IfElseStmtsMustUseBraces: Avoid using if..else statements without using curly braces.

解决方案　if...else...块必须使用括号：避免使用 if...else...块时不使用{}

- ForLoopsMustUseBraces: Avoid using 'for' statements without using curly braces.

解决方案　for 循环必须使用括号：避免在 for 循环时不使用{}

PMD 规则之 Clone Implementation Rules(克隆实现规则)

- ProperCloneImplementation: Object clone() should be implemented with super.clone().

解决方案　适当的克隆实现：对象的 clone()方法中应该包含 super.clone()实现

- CloneThrowsCloneNotSupportedException: The method clone() should throw a CloneNotSupportedException.

解决方案　克隆方法要抛出不支持克隆异常：clone()方法应该抛出 CloneNotSupportedException

- CloneMethodMustImplementCloneable: The method clone() should only be implemented if the class implements the Cloneable interface with the exception of a final method that only throws CloneNotSupportedException.

解决方案　克隆方法必须实现 Cloneable 接口：如果类实现 Cloneable 接口，clone() 方法应该被实现为一个 final 的方法并且只抛出 CloneNotSupportedException 的异常

*PMD 规则之 Code Size Rules

- NPathComplexity: The NPath complexity of a method is the number of acyclic execution paths through that method. A threshold of 200 is generally considered the point where measures should be taken to reduce complexity.

解决方案　n 条路径复杂度：NPath 复杂度是一个方法中各种可能的执行路径总和，一般把 200 作为考虑降低复杂度的临界点

- ExcessiveMethodLength: Violations of this rule usually indicate that the method is doing too much. Try to reduce the method size by creating helper methods and removing any copy/pasted code.

解决方案　方法太长：这种违例就是方法中做了太多事，通过创建辅助方法或移除拷贝/粘贴的代码试着减小方法的规模

- ExcessiveParameterList: Long parameter lists can indicate that a new object should be created to wrap the numerous parameters. Basically, try to group the parameters together.

解决方案　太多的参数：过长的参数列表表明应该创建一个新的对象包装众多的参数值，就是把参数组织到一起

- ExcessiveClassLength: Long Class files are indications that the class may be trying to do too much. Try to break it down, and reduce the size to something manageable.

解决方案　太长的类：太长的类文件表明类试图做太多的事，试着分解它，减少到易于管理的规模

- CyclomaticComplexity: Complexity is determined by the number of decision points in a method plus one for the method entry. The decision points are 'if', 'while', 'for', and 'case labels'. Generally, 1-4 is low complexity, 5-7 indicates moderate complexity, 8-10 is high complexity, and 11+ is very high complexity.

解决方案　　秩复杂性：由 if，while，for，case labels 等决策点确定的复杂度，1-4 是低复杂度，5-7 为中，8 到 10 是高复杂度，11 以上是非常高

- ExcessivePublicCount: A large number of public methods and attributes declared in a class can indicate the class may need to be broken up as increased effort will be required to thoroughly test it.

解决方案　　过多的公共成员：一个类中如果声明了大量的公共方法和属性表明类需要分解，因为想完全测试这个类需要大量的努力。

- TooManyFields: Classes that have too many fields could be redesigned to have fewer fields, possibly through some nested object grouping of some of the information. For example, a class with city/state/zip fields could instead have one Address field.

解决方案　　太多的域：类包含太多域可以被重新设计为包含更少的域，可以通过将一些信息组织为嵌套类。比如：一个类包含了 city/state/zip 域，可以用一个 Address 域组织这三个域

- NcssMethodCount: This rule uses the NCSS (Non Commenting Source Statements) algorithm to determine the number of lines of code for a given method. NCSS ignores comments, and counts actual statements. Using this algorithm, lines of code that are split are counted as one.

解决方案　　NCSS 方法代码计算：这个规则采用 NCSS(非注释代码块)算法计算给定的方法（不含构造方法）的代码行数。NCSS 忽略代码中的注释并且计算实际代码行数。用这种算法，一行单独的代码被计算为 1.（也同时忽略空行）

- NcssTypeCount: This rule uses the NCSS (Non Commenting Source Statements) algorithm to determine the number of lines of code for a given type. NCSS ignores comments, and counts actual statements. Using this algorithm, lines of code that are split are counted as one.

解决方案　NCSS 类代码计算：这个规则采用 NCSS(非注释代码块)算法计算给定类型的代码行数。NCSS 忽略代码中的注释并且计算实际代码行数。用这种算法，一行单独的代码被计算为 1.（也同时忽略空行）

- NcssConstructorCount: This rule uses the NCSS (Non Commenting Source Statements) algorithm to determine the number of lines of code for a given constructor. NCSS ignores comments, and counts actual statements. Using this algorithm, lines of code that are split are counted as one.

解决方案　NCSS 构造器代码计算：这个规则适用 NCSS（非注释代码块）算法计算给定的构造方法的代码行数。NCSS 忽略代码中的注释并且计算实际代码行数。用这种算法，一行单独的代码被计算为 1.（也同时忽略空行）

- TooManyMethods: A class with too many methods is probably a good suspect for refactoring, in order to reduce its complexity and find a way to have more fine grained objects.

解决方案　太多的方法：类中包含太多方法可能需要重构，以减低复杂度和获取更加细粒度的对象

PMD 规则之 Controversial Rules

- UnnecessaryConstructor: This rule detects when a constructor is not necessary; i.e., when there's only one constructor, it's public, has an empty body, and takes no arguments.

解决方案　非必要的构造器：本规则检查不必要的构造器，例如：只存在一个公共的，空方法体的，无参的构造器

- NullAssignment: Assigning a "null" to a variable (outside of its declaration) is usually bad form. Some times, the assignment is an indication that the programmer doesn't completely understand what is going on in the code. NOTE: This sort of assignment may in rare cases be useful to encourage garbage collection. If that's what you're using it for, by all means, disregard this rule :-)

解决方案　Null 赋值：将 null 赋值给变量(在声明之外)常常是不好的形式。某些时候这种赋值表示程序员没有想好代码的下一步该做什么。

备注：当你需要把变量赋值为 null 提示垃圾收集器去进行垃圾收集时这是有用的，那么请忽略这个规则

- OnlyOneReturn: A method should have only one exit point, and that should be the last statement in the method.

解决方案　只有一个返回：一个方法应该有且只有一处返回点，且应该是方法的最后一条语句。

- UnusedModifier: Fields in interfaces are automatically public static final, and methods are public abstract. Classes or interfaces nested in an interface are automatically public and static (all nested interfaces are automatically static). For historical reasons, modifiers which are implied by the context are accepted by the compiler, but are superfluous.

解决方案　无用的修饰符：在接口中定义的域自动为 public static final 的，方法自动是 public abstract 的，接口中嵌套的类或接口自动是 public static 的。由于历史原因，上下文暗示的修饰符是被编译器接受的，但是是多余的。

- AssignmentInOperand: Avoid assignments in operands; this can make code more complicated and harder to read.

解决方案　在操作中赋值：避免在操作中赋值；这会使代码复杂并且难以阅读

```
public class Foo {
 public void bar() {
  int x = 2;
  if ((x = getX()) == 3) {
   System.out.println("3!");
  }
 }
 private int getX() {
  return 3;
 }
}
```

- AtLeastOneConstructor: Each class should declare at least one constructor.

解决方案　至少有一个构造器：每个类应该至少声明一个构造器

- DontImportSun: Avoid importing anything from the 'sun.*' packages. These packages are not portable and are likely to change.

解决方案　不要引入 Sun 包：避免从"sun.*"引入任何类，这些包不是轻便的而且可能更改

- SuspiciousOctalEscape: A suspicious octal escape sequence was found inside a String literal. The Java language specification (section 3.10.6) says an octal escape sequence inside a literal String shall consist of a backslash followed by: OctalDigit | OctalDigit OctalDigit | ZeroToThree OctalDigit OctalDigit Any octal escape sequence followed by non-octal digits can be confusing, e.g. "/038" is interpreted as the octal escape sequence "/03" followed by the literal character "8".

解决方案　令人迷惑的八进制转义序列：在字符串字面量中出现令人迷惑的八进制转义序列。Java 语言规范(第 3.10.6 节)讲到：在一个字面量字符串中的八进制转义序列应该包含一个反斜杠，后续可以是：八进制数字|八进制数字　八进制数字|0~3　八进制数字　八进制数字的格式。所以任何八进制转义序列后面紧跟着一个非八进制的数字就可能造成迷惑，例如：“/038”

- CallSuperInConstructor: It is a good practice to call super() in a constructor. If super() is not called but another constructor (such as an overloaded constructor) is called, this rule will not report it.

解决方案　在构造器中调用 super():在构造器中调用 super()方法是很好的做法.如果没有调用 super()，但是调用了另外的构造器，那么这个规则不会报告出来。

- UnnecessaryParentheses: Sometimes expressions are wrapped in unnecessary parentheses, making them look like a function call.

解决方案　不必要的圆括号：有时候表达式被包在一个不必要的圆括号中，使它们看起来像是一个函数调用

```
public class Foo {
       boolean bar() {
            return (true);
       }
  }
```

- DefaultPackage: Use explicit scoping instead of the default package private level.

解决方案　默认的包：使用明确的范围代替默认的包私有的级别

- BooleanInversion: Use bitwise inversion to invert boolean values - it's the fastest way to do this. See http://www.javaspecialists.co.za/archive/newsletter.do?issue=042&locale=en_US for specific details

解决方案　布尔转换：使用按位转换来转换布尔值-这是最快的方法，参考：http://www.javaspecialists.co.za/archive/newsletter.do?issue=042&locale=en_US for specific details

```
public class Foo {
 public void main(bar) {
  boolean b = true;
  b = !b; // slow
  b ^= true; // fast
 }
}
```

- DataflowAnomalyAnalysis: The dataflow analysis tracks local definitions, undefinitions and references to variables on different paths on the data flow. From those informations there can be found various problems. 1. UR - Anomaly: There is a reference to a variable that was not defined before. This is a bug and leads to an error. 2. DU - Anomaly: A recently defined variable is undefined. These anomalies may appear in normal source text. 3. DD - Anomaly: A recently defined variable is redefined. This is ominous but don't have to be a bug.

解决方案　数据流异常分析：数据流分析是跟踪本地的变量定义与否及在数据流中不同路径的变量引用。由此可以发现多种问题：1.UR-异常：指向一个之前没有定义的变量，这是 bug 且可导致错误 2.DU-异常：一个刚刚定义的变量是未定义的。这些异常可能出现在普通的源代码文本中 3.DD-异常：一个刚刚定义的变量重新定义。这是不好的但并非一定是个 bug。

注：这个规则实在有点绕，具体含义我也不是很透彻理解！

```
public class Foo {
    public void foo() {
        int buz = 5;
        buz = 6; // redefinition of buz -> dd-anomaly
        foo(buz);
        buz = 2;
    } // buz is undefined when leaving scope -> du-anomaly
}
```

- AvoidFinalLocalVariable: Avoid using final local variables, turn them into fields.

解决方案　避免 Final 类型的本地变量：避免使用 final 类型的本地变量，将它们转为类域

```
public class MyClass {
    public void foo() {
        final String finalLocalVariable;
    }
}
```

- AvoidUsingShortType: Java uses the 'short' type to reduce memory usage, not to optimize calculation. In fact, the jvm does not have any arithmetic capabilities for the short type: the jvm must convert the short into an int, do the proper caculation and convert the int back to a short. So, the use of the 'short' type may have a greater impact than memory usage.

解决方案　避免使用 short 类型：Java 使用’short’类型来减少内存开销，而不是优化计算。事实上，JVM 不具备 short 类型的算术能力：jvm 必须将 short 类型转化为 int 类型，然后进行适当的计算再把 int 类型转回 short 类型。因此，和内存开销比起来使用’short’类型会对性能有更大的影响

- AvoidUsingVolatile: Use of the keyword 'volatile' is general used to fine tune a Java application, and therefore, requires a good expertise of the Java

Memory Model. Moreover, its range of action is somewhat misknown. Therefore, the volatile keyword should not be used for maintenance purpose and portability.

解决方案　避免使用 Volatile：使用关键字'volatile'一般用来调整一个 Java 应用，因此，需要一个专业的 Java 内存模型。此外，它的作用范围一定程度上是令人误解的。因此，volatile 关键字应该不要被用做维护和移植的目的。

- AvoidUsingNativeCode: As JVM and Java language offer already many help in creating application, it should be very rare to have to rely on non-java code. Even though, it is rare to actually have to use Java Native Interface (JNI). As the use of JNI make application less portable, and harder to maintain, it is not recommended.

解决方案　避免使用本地代码：jvm 和 Java 语言已经提供了很多创建应用程序的帮助，依赖非 Java 代码应该是非常罕见的。即使如此，事实上必须使用 Java 本地接口也是罕见的。因为使用 JNI 使得应用可移植性降低，而且难以维护，所以是不推荐的。

- AvoidAccessibilityAlteration: Methods such as getDeclaredConstructors(), getDeclaredConstructor(Class[]) and setAccessible(), as the interface PrivilegedAction, allow to alter, at runtime, the visilibilty of variable, classes, or methods, even if they are private. Obviously, no one should do so, as such behavior is against everything encapsulation principal stands for.

解决方案　避免改变访问控制：getDeclaredConstructors(), getDeclaredConstructor(Class[]) 和 setAccessible(),还有 PrivilegedAction 接口，允许在运行时改变变量、类和方法的可见性，甚至它们是私有的。显然，这是不应该的，因为这种动作违背了封装原则

- DoNotCallGarbageCollectionExplicitly: Calls to System.gc(), Runtime.getRuntime().gc(), and System.runFinalization() are not advised. Code should have the same behavior whether the garbage collection is disabled using the option -Xdisableexplicitgc or not. Moreover, "modern" jvms do a very good job handling garbage collections. If memory usage issues unrelated to memory leaks develop within an application, it should be dealt with JVM options rather than within the code itself.

解决方案　不要显示的调用垃圾收集器：调用 System.gc(), Runtime.getRuntime().gc(), 和 System.runFinalization()是不推荐的。当垃圾收集器使用配置项-Xdisableexplicitgc 关闭时，使用代码可以同样进行垃圾收集。此外，现代 JVM 对于垃圾收集工作做得很棒。当开发一个应用时内存使用的影响无关于内存泄露时，垃圾收集应该交给 JVM 配置项进行管理而非代码本身。

PMD 规则之 Coupling Rules

- CouplingBetweenObjects: This rule counts unique attributes, local variables and return types within an object. A number higher than specified threshold can indicate a high degree of coupling.

解决方案　对象间的耦合：这个规则统计一个对象中单个的属性、本地变量和返回类型的数目。如果统计数目大于指定的上限值表示耦合度太高。

- ExcessiveImports: A high number of imports can indicate a high degree of coupling within an object. Rule counts the number of unique imports and reports a violation if the count is above the user defined threshold.

解决方案　过多的引入：大量的 import 表明对象有很高的耦合度。本规则统计单一的 import 数目，如果数目大于用户定义的上限则报告一个违例。

- LooseCoupling: Avoid using implementation types (i.e., HashSet); use the interface (i.e, Set) instead

解决方案　松耦合：避免使用具体实现类型(如：HashSet);用接口(如：Set)代替。

PMD 规则之 Design Rules

- UseSingleton: If you have a class that has nothing but static methods, consider making it a Singleton. Note that this doesn't apply to abstract classes, since their subclasses may well include non-static methods. Also, if you want this class to be a Singleton, remember to add a private constructor to prevent instantiation.

解决方案　使用单例：如果有一个类包含的只有静态方法，可以考虑做成单例的。注意这个规则不适用于抽象类，因为它们的子类可能包含非静态方法。还有，如果你想把一个类做成单例的，记得写一个私有的构造器以阻止外部实例化。

- SimplifyBooleanReturns: Avoid unnecessary if..then..else statements when returning a boolean.

解决方案　简化布尔量的返回：避免在返回布尔量时写不必要的 if..then..else 表达式。

代码示例：

```
public class Foo {
  private int bar =2;
  public boolean isBarEqualsTo(int x) {
    // this bit of code
    if (bar == x) {
     return true;
    } else {
     return false;
    }
    // 上面可以简化为:
    // return bar == x;
  }
}
```

- SimplifyBooleanExpressions: Avoid unnecessary comparisons in boolean expressions - this complicates simple code.

解决方案　简化布尔表达式：避免布尔表达式之间无用的比较——只会使代码复杂化

代码示例：

```
public class Bar {
 // 下面可以简化为:  bar = isFoo();
 private boolean bar = (isFoo() == true);

 public isFoo() { return false;}
}
```

- SwitchStmtsShouldHaveDefault: Switch statements should have a default label.

解决方案　Switch 表达式应该有 default 块

- AvoidDeeplyNestedIfStmts: Deeply nested if..then statements are hard to read.

解决方案　避免深度嵌套的 if 表达式：深度嵌套的 if..then 表达式难以阅读

- AvoidReassigningParameters: Reassigning values to parameters is a questionable practice. Use a temporary local variable instead.

解决方案　避免给参数重新赋值：给传入方法的参数重新赋值是一种需要商榷的行为。使用临时本地变量来代替。

- SwitchDensity: A high ratio of statements to labels in a switch statement implies that the switch statement is doing too much work. Consider moving the statements into new methods, or creating subclasses based on the switch variable.

解决方案　密集的 switch：switch 表达式的 case 块中出现很高比例的表达式语句表明 switch 表达式做了太多的工作。考虑将表达式语句写进一个新的方法，或者创建基于 switch 变量的子类。

代码示例：

```
public class Foo {
 public void bar(int x) {
   switch (x) {
     case 1: {
       // lots of statements
       break;
     } case 2: {
       // lots of statements
       break;
     }
   }
 }
}
```

- ConstructorCallsOverridableMethod: Calling overridable methods during construction poses a risk of invoking methods on an incompletely constructed object and can be difficult to discern. It may leave the sub-class unable to construct its superclass or forced to replicate the construction process completely within itself, losing the ability to call super(). If the default constructor contains a call to an overridable method, the subclass may be completely uninstantiable. Note that this includes method calls throughout the control flow graph - i.e., if a constructor Foo() calls a private method bar() that calls a public method buz(), this denotes a problem.

解决方案　构造器调用了可重写的方法：在构造器中调用可被覆盖的方法可能引发在一个尚未构造完成的对象上调用方法的风险，而且是不易辨识的。它会使得子类不能构建父类或者自己强制重复构建过程，失去调用 super()方法的能力。如果一个默认的构造器包含一个对可重写方法的调用，子类可能完全不能被实例化。注意这也包含在整个控制流图上的方法调用——例如：如果构造器 Foo()调用了私有方法 bar()，而 bar()又调用了公开的方法 buz()，这就会导致问题。

代码示例：

```
public class SeniorClass {
  public SeniorClass(){
      toString(); //may throw NullPointerException if overridden
  }
  public String toString(){
    return "IAmSeniorClass";
  }
}
public class JuniorClass extends SeniorClass {
  private String name;
  public JuniorClass(){
    super(); //Automatic call leads to NullPointerException
    name = "JuniorClass";
  }
  public String toString(){
    return name.toUpperCase();
  }
}
```

- AccessorClassGeneration: Instantiation by way of private constructors from outside of the constructor's class often causes the generation of an accessor. A factory method, or non-privitization of the constructor can eliminate this situation. The generated class file is actually an interface. It gives the accessing class the ability to invoke a new hidden package scope constructor that takes the interface as a supplementary parameter. This turns a private constructor effectively into one with package scope, and is challenging to discern.

解决方案　存取器类生成：从一个具有私有构建器的类的外部实例化这个类通常会导致存取器的生成。工厂方法，或者非私有化的构造器可以避免这个情况。生成的类文件事实上是一个接口。它赋予访问类调用一个新的隐藏的包范围的构建器并把这个接口作为补充参数的能力。这样就把私有的构造器有效地转换为一个包范围的构建器，而且是不易觉察的。

代码示例：

```
public class Outer {
 void method(){
  Inner ic = new Inner();//Causes generation of accessor class
 }
 public class Inner {
  private Inner(){}
 }
}
```

- FinalFieldCouldBeStatic: If a final field is assigned to a compile-time constant, it could be made static, thus saving overhead in each object at runtime.

解决方案　final 类型的域可以同时是 static 的：如果一个 final 类型的域在编译时被赋值为常量，它也可以是 static 的，那样就在每个对象运行时节省开支。

- CloseResource: Ensure that resources (like Connection, Statement, and ResultSet objects) are always closed after use.

解决方案　关闭资源：确保这些资源(譬如：Connection,Statement,和 ResultSet 对象)总在使用后被关闭。

- NonStaticInitializer: A nonstatic initializer block will be called any time a constructor is invoked (just prior to invoking the constructor). While this is a valid language construct, it is rarely used and is confusing.

解决方案　非静态的初始化器：非静态的初始化块将在构造器被调用的时候被访问(优先于调用构造器)。这是一个有效的语言结构，但使用很少且易造成迷惑。

代码示例：

```
public class MyClass {
 // this block gets run before any call to a constructor
 {
  System.out.println("I am about to construct myself");
 }
}
```

- DefaultLabelNotLastInSwitchStmt: By convention, the default label should be the last label in a switch statement.

解决方案　switch 表达式中 default 块应该在最后：按照惯例，default 标签应该是 switch 表达式的最后一个标签。

- NonCaseLabelInSwitchStatement: A non-case label (e.g. a named break/continue label) was present in a switch statement. This legal, but confusing. It is easy to mix up the case labels and the non-case labels.

解决方案　switch 表达式中没有 case 标签：在 switch 表达式中没有 case，是合法的，但是容易造成迷惑。容易将 case 标签和非 case 标签混淆。

- OptimizableToArrayCall: A call to Collection.toArray can use the Collection's size vs an empty Array of the desired type.

解决方案　优化 toArray 调用：调用 Collection.toArray 时使用集合的规模加上目标类型的空数组作为参数。

代码示例：

```
class Foo {
 void bar(Collection x) {
    // A bit inefficient
    x.toArray(new Foo[0]);
    // Much better; this one sizes the destination array, avoiding
    // a reflection call in some Collection implementations
    x.toArray(new Foo[x.size()]);
  }
}
```

- BadComparison: Avoid equality comparisons with Double.NaN - these are likely to be logic errors.

解决方案　错误的比较：避免 Double.NaN 的相等性比较-这些可能是逻辑错误

- EqualsNull: Inexperienced programmers sometimes confuse comparison concepts and use equals() to compare to null.

解决方案　等于空：经验缺乏的程序员有时候会迷惑于相等性概念，拿 equals()方法和 null 比较

- ConfusingTernary: In an "if" expression with an "else" clause, avoid negation in the test. For example, rephrase: if (x != y) diff(); else same(); as: if (x == y) same(); else diff(); Most "if (x != y)" cases without an "else" are often return cases, so consistent use of this rule makes the code easier to read. Also, this resolves trivial ordering problems, such as "does the error case go first?" or "does the common case go first?".

解决方案　令人迷惑的三种性：在 if 表达式伴随 else 子句时，避免在 if 测试中使用否定表达。例如：不要使用 if (x != y) diff(); else same()这种表述，而应该使用 if (x == y) same(); else diff()，大多数时候使用 if(x!=y)形式时不包含 else 分句，所以一贯地使用此规则能让代码更易于阅读。此外，这也解决了一个细节的排序问题，比如“应该是判断为 false 的代码块在前面？”，还是“判断通过的代码块在前？”

- InstantiationToGetClass: Avoid instantiating an object just to call getClass() on it; use the .class public member instead.

解决方案　通过 getClass 实例化：避免通过访问 getClass()实例化对象，使用.class 这个公共属性代替。

- IdempotentOperations: Avoid idempotent operations - they are have no effect.

解决方案　幂等性操作：避免幂等性操作-它们不起任何作用

幂等性操作，同样的操作无论执行多少次，其结果都跟第一次执行的结果相同。

代码示例：

```
public class Foo {
 public void bar() {
  int x = 2;
  x = x;
 }
}
```

- SimpleDateFormatNeedsLocale: Be sure to specify a Locale when creating a new instance of SimpleDateFormat.

解决方案　SimpleDateFormat 类需要本地参数：确保在创建 SimpleDateFormat 类的实例时指定了 Locale 参数

- ImmutableField: Identifies private fields whose values never change once they are initialized either in the declaration of the field or by a constructor. This aids in converting existing classes to immutable classes.

解决方案　不变域：识别出一旦被声明就赋值或通过构造器赋值后就从不改变的私有域，它们将存在的类变成了不变类。这样的域可以是 final 的

代码示例：

```
public class Foo {
  private int x; // could be final
  public Foo() {
      x = 7;
  }
  public void foo() {
     int a = x + 2;
  }
}
```

- UseLocaleWithCaseConversions: When doing a String.toLowerCase()/toUpperCase() call, use a Locale. This avoids problems with certain locales, i.e. Turkish.

解决方案　大小写转换时使用 Locale: 当访问 String.toLowerCase()/toUpperCase() 时使用 Locale 参数，这样可以避免某些特定的本地化问题，比如土耳其语

- AvoidProtectedFieldInFinalClass: Do not use protected fields in final classes since they cannot be subclassed. Clarify your intent by using private or package access modifiers instead.

解决方案　避免在 final 类中使用 protected 域：因为 final 类型的 class 不能被继承，所以不要使用 protected 域，通过使用 private 或包访问符来代替以修正你的意图。

- AssignmentToNonFinalStatic: Identifies a possible unsafe usage of a static field.

解决方案　给一个非 final 的 static 类型赋值：识别出一个非安全的 static 域使用

代码示例：

```
public class StaticField {
   static int x;
   public FinalFields(int y) {
    x = y; // unsafe
   }
}
```

- MissingStaticMethodInNonInstantiatableClass: A class that has private constructors and does not have any static methods or fields cannot be used.

解决方案　在不可实例化类中缺少静态方法：一个具有私有构造器且没有任何静态方法或静态变量的类不能被使用

- AvoidSynchronizedAtMethodLevel: Method level synchronization can backfire when new code is added to the method. Block-level synchronization helps to ensure that only the code that needs synchronization gets it.

解决方案　避免方法级的同步：当为一个同步的方法加入新代码时可能发生意外。块级别的同步可以确保内含真正需要同步的代码。

- MissingBreakInSwitch: A switch statement without an enclosed break statement may be a bug.

解决方案　switch 块中缺少 break：switch 表达式缺少内含的 break 块可能是 bug。

- UseNotifyAllInsteadOfNotify: Thread.notify() awakens a thread monitoring the object. If more than one thread is monitoring, then only one is chosen. The thread chosen is arbitrary; thus it's usually safer to call notifyAll() instead.

解决方案　使用 notifyAll 代替 notify: Thread.notify()唤醒监控对象的线程。如果多余一个线程在监控中，只会有一个被选择。这个线程的选择是随机的，因此调用 notifyAll() 代替它更安全。

- AvoidInstanceofChecksInCatchClause: Each caught exception type should be handled in its own catch clause.

解决方案　避免在 catch 块中使用 instanceof:每个产生的异常类型都应该在自己的 catch 块中被处理。

- AbstractClassWithoutAbstractMethod: The abstract class does not contain any abstract methods. An abstract class suggests an incomplete implementation, which is to be completed by subclasses implementing the abstract methods. If the class is intended to be used as a base class only (not to be instantiated direcly) a protected constructor can be provided prevent direct instantiation.

解决方案　抽象类没有抽象方法：抽象类没有包含抽象方法，抽象类建议未完成的方法实现，这个方法在子类中被完成。如果类意图被用作基础类（不被直接实例化），一个 protected 的构造器就能提供对直接实例化的阻止。

- SimplifyConditional: No need to check for null before an instanceof; the instanceof keyword returns false when given a null argument.

解决方案　简化条件：在使用 instanceof 之间不需要 null 校验，当给予一个 null 作为参数时，instanceof 返回 false

- CompareObjectsWithEquals: Use equals() to compare object references; avoid comparing them with ==.

解决方案　对象相等性比较：使用 equals()比较对象的引用，避免使用”==”来比较

- PositionLiteralsFirstInComparisons: Position literals first in String comparisons - that way if the String is null you won't get a NullPointerException, it'll just return false.

解决方案　把字面量放在比较式的前面：在字符串比较时，将字面量放在前面-这种方法能够避免当字符串为空时的空指针异常，只是返回 false。

- UnnecessaryLocalBeforeReturn: Avoid unnecessarily creating local variables

解决方案　在 return 之前不必要的本地变量：避免创建不必要的本地变量

代码示例：

```
public class Foo {
    public int foo() {
      int x = doSomething();
      return x;  // instead, just 'return doSomething();'
    }
  }
```

- NonThreadSafeSingleton: Non-thread safe singletons can result in bad state changes. Eliminate static singletons if possible by instantiating the object directly. Static singletons are usually not needed as only a single instance exists anyway. Other possible fixes are to synchronize the entire method or to use an initialize-on-demand holder class (do not use the double-check idiom). See Effective Java, item 48.

解决方案　非线程安全的单例：非线程安全的单例可以导致错误的状态转换。如果可能通过直接实例化消除静态的单例.因为事实上只存在一个实例，所以静态单例一般是不需要的。另外的解决办法是同步实际的方法或者使用一个按需实例化的持有类(不要使用双重检查机制)。请参阅：Effective Java,48 章。

代码示例：

```
private static Foo foo = null;
//multiple simultaneous callers may see partially initialized objects
public static Foo getFoo() {
    if (foo==null)
```

```
        foo = new Foo();
    return foo;
}
```

- UncommentedEmptyMethod: Uncommented Empty Method finds instances where a method does not contain statements, but there is no comment. By explicitly commenting empty methods it is easier to distinguish between intentional (commented) and unintentional empty methods.

解决方案　不含注释的空方法：不含注释的空方法就是说一个方法内部没有任何程序代码也没有注释。通过明确的注释空方法能够容易的区分有潜在意向的和无意向的空方法。

- UncommentedEmptyConstructor: Uncommented Empty Constructor finds instances where a constructor does not contain statements, but there is no comment. By explicitly commenting empty constructors it is easier to distinguish between intentional (commented) and unintentional empty constructors.

解决方案　不含注释的空构造器：不含注释的空构造器就是说一个构造器内部没有任何程序代码也没有注释。通过明确的注释空构造器能够容易的区分有潜在意向的和无意向的空构造器

- AvoidConstantsInterface: An interface should be used only to model a behaviour of a class: using an interface as a container of constants is a poor usage pattern.

解决方案　避免常量接口：接口仅仅是应该用来建模类的行为的：使用接口作为常量的容器是一种劣质的用法。

- UnsynchronizedStaticDateFormatter: SimpleDateFormat is not synchronized. Sun recomends separate format instances for each thread. If multiple threads must access a static formatter, the formatter must be synchronized either on method or block level.

解决方案　不要同步静态的 DateFormat 类：SimpleDateFormat 是非同步的。Sun 公司建议对每个线程单独的 format 实例。如果多线程必须访问一个静态 formatter，formatter 必须在方法或块级别同步。

代码示例：

```
public class Foo {
    private static final SimpleDateFormat sdf = new SimpleDateFormat();
    void bar() {
        sdf.format(); // bad
    }
    synchronized void foo() {
        sdf.format(); // good
    }
}
```

- PreserveStackTrace: Throwing a new exception from a catch block without passing the original exception into the new exception will cause the true stack trace to be lost, and can make it difficult to debug effectively.

解决方案　保留追踪栈：在一个 catch 块中抛出一个新的异常却不把原始的异常传递给新的异常会导致真正的追踪信息栈丢失，而且导致难以有效的调试。

代码示例：

```
public class Foo {
    void good() {
        try{
            Integer.parseInt("a");
        } catch(Exception e){
            throw new Exception(e);
        }
    }
    void bad() {
        try{
            Integer.parseInt("a");
        } catch(Exception e){
            throw new Exception(e.getMessage());
        }
    }
}
```

- UseCollectionIsEmpty: The isEmpty() method on java.util.Collection is provided to see if a collection has any elements. Comparing the value of size() to 0 merely duplicates existing behavior.

解决方案　使用集合类的 isEmpty 方法：java.util.Collection 类的 isEmpty 方法提供判断一个集合类是否包含元素。不要是使用 size()和 0 比较来重复类库已经提供的方法。

```
public class Foo {
            void good() {
            List foo = getList();
                  if (foo.isEmpty()) {
                       // blah
                  }
      }
//下面是不推荐的方式
         void bad() {
         List foo = getList();
                        if (foo.size() == 0) {
                             // blah
                        }
               }
      }
```

- ClassWithOnlyPrivateConstructorsShouldBeFinal: A class with only private constructors should be final, unless the private constructor is called by a inner class.

解决方案　类只包含私有的构造器应该是 final 的：一个类只包含私有的构造器应该是 final 的，除非私有构造器被一个内部类访问。

```
public class Foo {   //Should be final
   private Foo() { }
}
```

- EmptyMethodInAbstractClassShouldBeAbstract: An empty method in an abstract class should be abstract instead, as developer may rely on this empty implementation rather than code the appropriate one.

解决方案　一个抽象类中的空方法也应该是抽象的：一个抽象类中的空方法也应该是抽象的，因为开发者有可能会信任这个空的实现而不去编写恰当的代码。

- SingularField: This field is used in only one method and the first usage is assigning a value to the field. This probably means that the field can be changed to a local variable.

解决方案　单数的域：域变量只在一个方法中被使用并且第一次使用时对这个域赋值。这种域可以改写为本地变量。

```
public class Foo {
    private int x;  //Why bother saving this?
    public void foo(int y) {
     x = y + 5;
     return x;
    }
}
```

- ReturnEmptyArrayRatherThanNull: For any method that returns an array, it's a better behavior to return an empty array rather than a null reference.

解决方案　返回空的数组而不要返回 null：对于任何返回数组的方法，返回一个空的数组是一个比返回 null 引用更好的做法。

- AbstractClassWithoutAnyMethod: If the abstract class does not provides any methods, it may be just a data container that is not to be instantiated. In this case, it's probably better to use a private or a protected constructor in order to prevent instantiation than make the class misleadingly abstract.

解决方案　没有任何方法的抽象类：如果抽象类没有提供任何的方法，它可能只是一个不可被实例化的数据容器，在这种状况下，更好的方法是使用私有的或受保护的构造器以阻止实例化可以让类避免带有欺骗性的抽象。

- TooFewBranchesForASwitchStatement: Switch are designed complex branches, and allow branches to share treatment. Using a switch for only a few branches is ill advised, as switches are not as easy to understand as if. In this case, it's most likely is a good idea to use a if statement instead, at least to increase code readability.

解决方案　switch 表达式带有太少的分支：switch 表达式是被设计为处理复杂分支的，并且允许分支共享处理逻辑。对于较少分支的情况使用 switch 是不明智的，因为这样代码不易理解。因此，更好的主意是使用 if 表达式来代替，这样至少可以增加代码可读性。

PMD 规则之 Finalizer Rules

- EmptyFinalizer: If the finalize() method is empty, then it does not need to exist.

解决方案　空的 finalize():如果 finalize()方法是空的，那么它就不需要存在。

- FinalizeOnlyCallsSuperFinalize: If the finalize() is implemented, it should do something besides just calling super.finalize().

解决方案　finalize 方法调用父类 finalize：如果 finalize()被实现，它应该除了调用 super.finalize()之外还应该做点别的。

- FinalizeOverloaded: Methods named finalize() should not have parameters. It is confusing and probably a bug to overload finalize(). It will not be called by the VM.

解决方案　finalize 重载：方法名是 finalize()的方法应该具有参数。因为不带参数容易令人迷惑且可能是一个 bug，那样就不会被 JVM 调用。

- FinalizeDoesNotCallSuperFinalize: If the finalize() is implemented, its last action should be to call super.finalize.

解决方案　finalize 没有调用父类的 finalize:如果 finalize()方法被重新实现，它最后一个动作应该是调用 super.finalize();

- FinalizeShouldBeProtected: If you override finalize(), make it protected. If you make it public, other classes may call it.

解决方案　finalize 应该是受保护的：如果你覆盖 finalize()，使他是 protected 的，如果作为 public 的，其它类就可以调用了。

- AvoidCallingFinalize: Object.finalize() is called by the garbage collector on an object when garbage collection determines that there are no more references to the object.

解决方案　避免调用 finalize:Object.finalize()是由垃圾收集器发现没有引用指向这个对象的时候调用的，应尽量避免人为调用

PMD 规则之 Import Statement Rules

- DuplicateImports: Avoid duplicate import statements.

解决方案　重复的引入：避免重复的 import

- DontImportJavaLang: Avoid importing anything from the package 'java.lang'. These classes are automatically imported (JLS 7.5.3).

解决方案　不要引入 java.lang:避免从'java.lang'包引入任何东西，它里面的类是自动引入的

- UnusedImports: Avoid unused import statements.

解决方案　未使用的 imports:去掉不使用的 import

- ImportFromSamePackage: No need to import a type that lives in the same package.

解决方案　从同一个包引入：不需要从同一包引入类型

- TooManyStaticImports: If you overuse the static import feature, it can make your program unreadable and unmaintainable, polluting its namespace with all the static members you import. Readers of your code (including you, a few months after you wrote it) will not know which class a static member comes from (Sun 1.5 Language Guide).

解决方案　太多的静态引入：如果滥用静态引入特性，会使你的程序不具有可读性和可维护性，你引入的太多的静态成员污染

*PMD 规则之 J2EE Rules

- UseProperClassLoader: In J2EE getClassLoader() might not work as expected. Use Thread.currentThread().getContextClassLoader() instead.

解决方案　使用合适的类加载器：在 J2EE 中 getClassLoader()方法可能不会按照期望工作。使用 Thread.currentThread().getContextClassLoader()来代替。

- MDBAndSessionBeanNamingConvention: The EJB Specification state that any MessageDrivenBean or SessionBean should be suffixed by Bean.

解决方案　消息驱动 bean 和会话 bean 命名规则：EJB 规范表示任何消息驱动 bean 和会话 bean 的命名应该以'Bean'结尾。

代码示例：

```
/* Proper name */
            public class SomeBean implements SessionBean{}

/* Bad name */
            public class MissingTheProperSuffix implements SessionBean {}
```

- RemoteSessionInterfaceNamingConvention: Remote Home interface of a Session EJB should be suffixed by 'Home'.

解决方案　远程会话接口命名规则：会话 EJB 的 remote home 接口命名应该以'Home'结尾。

代码示例：

```
/* Proper name */
            public interface MyBeautifulHome extends javax.ejb.EJBHome {}

/* Bad name */
            public interface MissingProperSuffix extends javax.ejb.EJBHome
{}
```

- LocalInterfaceSessionNamingConvention: The Local Interface of a Session EJB should be suffixed by 'Local'.

解决方案　本地接口会话命名规则：会话 EJB 的本地接口应该以 'Local' 结尾。

代码示例：

```
/* Proper name */
            public interface MyLocal extends javax.ejb.EJBLocalObject {}
```

```
/* Bad name */
        public interface MissingProperSuffix extends
javax.ejb.EJBLocalObject {}
```

- LocalHomeNamingConvention: The Local Home interface of a Session EJB should be suffixed by 'LocalHome'.

解决方案　本地 Home 命名规则：会话 EJB 的本地 home 接口应该以'LocalHome'结尾

代码示例：

```
/* Proper name */
        public interface MyBeautifulLocalHome extends
javax.ejb.EJBLocalHome {}

/* Bad name */
        public interface MissingProperSuffix extends
javax.ejb.EJBLocalHome {}
```

- RemoteInterfaceNamingConvention: Remote Interface of a Session EJB should NOT be suffixed.

解决方案　远程接口命名规则：会话 EJB 的远程接口应该没有后缀。

代码示例：

```
/* Bad Session suffix */
        public interface BadSuffixSession extends javax.ejb.EJBObject {}

/* Bad EJB suffix */
        public interface BadSuffixEJB extends javax.ejb.EJBObject {}

/* Bad Bean suffix */
        public interface BadSuffixBean extends javax.ejb.EJBObject {}
```

- DoNotCallSystemExit: Web applications should not call System.exit(), since only the web container or the application server should stop the JVM.

解决方案　不要调用 System.exit:web 应用不该调用 System.exit(),因为只有 web 容器或应用服务器才能停止 JVM.

- StaticEJBFieldShouldBeFinal: According to the J2EE specification (p.494), an EJB should not have any static fields with write access. However, static read only fields are allowed. This ensures proper behavior especially when instances are distributed by the container on several JREs.

解决方案　静态 EJB 域应该是 final 的：根据 J2EE 规范(p.494),EJB 不应该有任何具有写入访问权的静态域，然而，只读静态域是允许的。这样能够保证合适的行为，尤其当实例分布存在于多个 JRE 的容器中

- DoNotUseThreads: The J2EE specification explicitly forbid use of threads.

解决方案　不用使用线程：J2EE 规范明确禁止使用线程。

备注：意思是已经由 J2EE 规范和成熟类库帮你封装了线程处理，自己尽量不要用线程。

PMD 规则之 JavaBean Rules

- BeanMembersShouldSerialize: If a class is a bean, or is referenced by a bean directly or indirectly it needs to be serializable. Member variables need to be marked as transient, static, or have accessor methods in the class. Marking variables as transient is the safest and easiest modification. Accessor methods should follow the Java naming conventions, i.e.if you have a variable foo, you should provide getFoo and setFoo methods.

解决方案　Bean 成员应该是可序列化的：如果类是一个 Java bean，或者被一个 bean 直接或间接地引用，它就需要序列化。类的成员变量需要被标记为 transient,static,或具备访问方法。将变量标记为 transient 是最安全和最简单的改动方法。访问方法应该遵循 java 命名规范，例如：如果有一个变量 foo,应该提供 getFoo()和 setFoo()方法。

- MissingSerialVersionUID: Classes that are serializable should provide a serialVersionUID field.

解决方案　缺少序列化版本 ID：可序列化的类应该提供 serialVersionUID 域

PMD 规则之 Junit Rules

- JUnitStaticSuite: The suite() method in a JUnit test needs to be both public and static.

解决方案　Junit 静态套件：在 Junit 测试中 suite()方法需要是公共的和静态的

- JUnitSpelling: Some JUnit framework methods are easy to misspell.

解决方案　Junit 拼写：一些 Junit 框架方法容易拼写错误。

- JUnitAssertionsShouldIncludeMessage: JUnit assertions should include a message - i.e., use the three argument version of assertEquals(), not the two argument version.

解决方案　Junit 断言应该包含提示信息：Junit 断言应该包含提示信息-比如你应该使用三个参数的 assertEquals()方法，而不使用两个参数的。

- JUnitTestsShouldIncludeAssert: JUnit tests should include at least one assertion. This makes the tests more robust, and using assert with messages provide the developer a clearer idea of what the test does.

解决方案　Junit 测试应该包含断言：Junit 测试应该至少包含一个断言。这能让测试更健壮，而且使用附带提示信息的断言能让开发者清楚的了解测试到底做了什么。

- TestClassWithoutTestCases: Test classes end with the suffix Test. Having a non-test class with that name is not a good practice, since most people will assume it is a test case. Test classes have test methods named testXXX.

解决方案　测试类没有用例：测试类以 Test 作为后缀。非测试类的命名含有 test 作为后缀不是好的方式，因为大多数人会以为它是一个测试用例。测试类中的测试方法命名方式是 testXXX。

- UnnecessaryBooleanAssertion: A JUnit test assertion with a boolean literal is unnecessary since it always will eval to the same thing. Consider using flow control (in case of assertTrue(false) or similar) or simply removing statements like assertTrue(true) and assertFalse(false). If you just want a test to halt, use the fail method.

解决方案　不必要的布尔断言：包含一个布尔字面量的测试断言是不必要的，因为它总是计算同样的值。如果考虑流程控制(一旦出现 assertTrue(false)或类似字眼)或仅仅移除类似 assertTrue(true)和 assertFalse(false)等表达式。如果你只是打算测试挂起，可以使用 fail 方法。

- UseAssertEqualsInsteadOfAssertTrue: This rule detects JUnit assertions in object equality. These assertions should be made by more specific methods, like assertEquals.

解决方案　　使用 assertEquals 替代 assertTrue：本规则检查 Junit 断言在对象值上的相等性。这些断言应该由更多特定的方法组成，例如 assertEquals。

- UseAssertSameInsteadOfAssertTrue: This rule detects JUnit assertions in object references equality. These assertions should be made by more specific methods, like assertSame, assertNotSame.

解决方案　　使用 assertSame 替代 assertTrue:本规则检查 Junit 断言在对象引用上的相等性。此类断言应该由更多特定方法组成，比如：assertSame，assertNotSame

- UseAssertNullInsteadOfAssertTrue: This rule detects JUnit assertions in object references equality. These assertions should be made by more specific methods, like assertNull, assertNotNull.

解决方案　　使用 assertNull 替代 assertTrue：本规则检查 Junit 断言对象引用的相等性，此类断言应该由更多特定方法组成，比如：assertNull，assertNotNull

- SimplifyBooleanAssertion: Avoid negation in an assertTrue or assertFalse test. For example, rephrase: assertTrue(!expr); as: assertFalse(expr);

解决方案　　简化布尔断言: 避免在 assertTrue 或 assertFalse 方法测试时使用反向表达，比如：将 assertTrue(!expr) 换成 assertFalse(expr) 来表达

PMD 规则之 Jakarta Commons Logging Rules

- UseCorrectExceptionLogging: To make sure the full stacktrace is printed out, use the logging statement with 2 arguments: a String and a Throwable.

解决方案　　使用正确的异常日志：保证打印出完全的异常堆栈，记录或抛出异常日志时使用包含两个参数的表达式：一个参数是字符串，一个是 Throwable 类型

- ProperLogger: A logger should normally be defined private static final and have the correct class. Private final Log log; is also allowed for rare cases where loggers need to be passed around, with the restriction that the logger needs to be passed into the constructor.

解决方案　合适的日志记录器：日志记录器一般应该被定义为 private static final 的，而且应该有正确的类。Private final Log log;也被允许使用在需要传递的这种极少的情况中，具有这种限制日志记录器就需要被传入构造器中。

PMD 规则之 Java Logging Rules

- MoreThanOneLogger: Normally only one logger is used in each class.

解决方案　多于一个日志记录器：一般而言一个日志记录器只用于一个类中。

- LoggerIsNotStaticFinal: In most cases, the Logger can be declared static and final.

解决方案　日志记录器不是 static final 的：大多数情况下，日志记录器应该被定义为 static 和 final 的

- SystemPrintln: System.(out|err).print is used, consider using a logger.

解决方案　SystemPrintln：如果发现代码当中使用了 System.(out|err).print,应考虑使用日志记录代替

- AvoidPrintStackTrace: Avoid printStackTrace(); use a logger call instead.

解决方案　避免使用 PrintStackTrace:避免使用 printStackTrace();使用日志记录器代替。

*PMD 规则之 Migration Rules

- ReplaceVectorWithList: Consider replacing Vector usages with the newer java.util.ArrayList if expensive threadsafe operation is not required.

解决方案　用 List 替换 Vector：如果不是在必须线程安全的环境下，考虑使用 List 代替 Vector

- ReplaceHashtableWithMap: Consider replacing this Hashtable with the newer java.util.Map

解决方案　用 Map 替换 Hashtable: 如果不是在必须线程安全的环境下，考虑使用 Map 替换 Hashtable

- ReplaceEnumerationWithIterator: Consider replacing this Enumeration with the newer java.util.Iterator

解决方案　用 Iterator 替换 Enumeration:考虑用 Iterator 替换 Enumeration

- AvoidEnumAsIdentifier: Finds all places where 'enum' is used as an identifier.

解决方案　避免用 enum 作为标识符：找到所有使用 enum 作为标识符的地方

- AvoidAssertAsIdentifier: Finds all places where 'assert' is used as an identifier.

解决方案　避免使用 Assert 作为标识符：找到所有使用 assert 作为标识符的地方

- IntegerInstantiation: In JDK 1.5, calling new Integer() causes memory allocation. Integer.valueOf() is more memory friendly.

解决方案　Integer 类型实例化：对于 JDK1.5,调用 new Integer()会造成内存分配，而 Integer.valueOf()具有更好的内存友好性。

- ByteInstantiation: In JDK 1.5, calling new Byte() causes memory allocation. Byte.valueOf() is more memory friendly.

解决方案　Byte 类型初始化：对于 JDK1.5,调用 new Byte()会造成内存分配，而 Byte.valueOf()具有更好的内存友好性。

- ShortInstantiation: In JDK 1.5, calling new Short() causes memory allocation. Short.valueOf() is more memory friendly.

解决方案　Short 类型初始化：对于 JDK1.5,调用 new Short ()会造成内存分配，而 Short.valueOf()具有更好的内存友好性。

- LongInstantiation: In JDK 1.5, calling new Long() causes memory allocation. Long.valueOf() is more memory friendly.

解决方案　Long 类型初始化：对于 JDK1.5,调用 new Long()会造成内存分配，而 Long.valueOf()具有更好的内存友好性。

- JUnit4TestShouldUseBeforeAnnotation: In JUnit 3, the setUp method was used to set up all data entities required in running tests. JUnit 4 skips the setUp method and executes all methods annotated with @Before before all tests

解决方案　Junit4 测试时应该使用 Before 注解：在 Junit3 中，我们使用 setUp()方法设置运行测试时需要的值，然而 Junit4 忽略 setUp 方法在所有测试前先执行被@Before 标注过的方法。

- JUnit4TestShouldUseAfterAnnotation: In JUnit 3, the tearDown method was used to clean up all data entities required in running tests. JUnit 4 skips the tearDown method and executes all methods annotated with @After after running each test

解决方案　Junit4 测试时应该使用 After 注解：对于 Junit3,tearDown()方法被用来清除测试执行后的数据。Junit4 忽略 tearDown 方法然后在执行每个测试方法后执行所有标记为@After 的方法。

- JUnit4TestShouldUseTestAnnotation: In JUnit 3, the framework executed all methods which started with the word test as a unit test. In JUnit 4, only methods annotated with the @Test annotation are executed.

解决方案　Junit4 测试应使用 Test 注解：对于 Junit3，测试框架执行每个以 test 单词打头的方法作为单元测试。而对于 Junit4，只有标记为@Test 注解的方法才能被执行。

- JUnit4SuitesShouldUseSuiteAnnotation: In JUnit 3, test suites are indicated by the suite() method. In JUnit 4, suites are indicated through the @RunWith(Suite.class) annotation.

解决方案　Junit4 测试套件应使用 Suite 注解：对于 Junit3，测试套件表现为 suite()方法，而对于 Junit4，测试套件表现为@RunWith(Suite.class)注解

PMD 规则之 Naming Rules

- ShortVariable: Detects when a field, local, or parameter has a very short name.

解决方案　短变量：检测出域或参数的名字命名非常短。

- LongVariable: Detects when a field, formal or local variable is declared with a long name.

解决方案　长变量：检测出域或参数的名字命名非常长。

- ShortMethodName: Detects when very short method names are used.

解决方案　短方法名：检测出方法命名太短。

- VariableNamingConventions: A variable naming conventions rule - customize this to your liking. Currently, it checks for final variables that should be fully capitalized and non-final variables that should not include underscores.

解决方案　变量命名约定：变量命名规则-根据你的喜好调整。当前规则检查 final 类型变量应该全部大写而且非 final 变量应该不包含下划线。

- MethodNamingConventions: Method names should always begin with a lower case character, and should not contain underscores.

解决方案　方法命名约定：方法命名应该总是以小写字符开头，而且不应该包含下划线。

- ClassNamingConventions: Class names should always begin with an upper case character.

解决方案　类命名约定：类名应该总是以大写字符开头

- AbstractNaming: Abstract classes should be named 'AbstractXXX'.

解决方案　抽象类命名：抽象类应该命名为‘AbstractXXX’

- AvoidDollarSigns: Avoid using dollar signs in variable/method/class/interface names.

解决方案　避免美元符号：在变量/方法/类/接口中避免使用美元符号。

- MethodWithSameNameAsEnclosingClass: Non-constructor methods should not have the same name as the enclosing class.

解决方案　方法和封装类同名：非构造方法不能和封装类同名

- SuspiciousHashcodeMethodName: The method name and return type are suspiciously close to hashCode(), which may mean you are intending to override the hashCode() method.

解决方案　令人疑惑的 hashCode 方法名：方法名和返回值类似于 hashCode()，将令人误解为你试图覆盖 hashCode()方法。

- SuspiciousConstantFieldName: A field name is all in uppercase characters, which in Sun's Java naming conventions indicate a constant. However, the field is not final.

解决方案　令人疑惑的常量字段名：一个字段的名称全部用大些字符表示是 Sun 的 JAVA 命名规则，表示这个是常量。然而，字段不是 final 的。

- SuspiciousEqualsMethodName: The method name and parameter number are suspiciously close to equals(Object), which may mean you are intending to override the equals(Object) method.

解决方案　令人迷惑的 equals 方法名：方法名和参数近似于 equals(Object)，可能让人迷惑为你想覆盖 equals(Object)方法

- AvoidFieldNameMatchingTypeName: It is somewhat confusing to have a field name matching the declaring class name. This probably means that type and or field names could be more precise.

解决方案　避免属性和类同名：属性和类同名易造成误解，这可能意味着类型或属性名可以命名的更精确

- AvoidFieldNameMatchingMethodName: It is somewhat confusing to have a field name with the same name as a method. While this is totally legal, having information (field) and actions (method) is not clear naming.

解决方案　避免属性和方法同名：属性和方法同名易造成误解。即使这是完全合法的，但信息(属性)和动作(方法)都没有清晰地命名。

- NoPackage: Detects when a class or interface does not have a package definition.

解决方案　没有包：检测到类或接口没有定义在包里面。

- PackageCase: Detects when a package definition contains upper case characters.

解决方案　包的大小写：检测到包的定义中包含大写字符

- MisleadingVariableName: Detects when a non-field has a name starting with 'm_'. This usually indicates a field and thus is confusing.

解决方案　令人迷惑的变量名：检测到非字段类型以 m_开头，这通常表示这是一个字段所以这种做法让人迷惑。

- BooleanGetMethodName: Looks for methods named 'getX()' with 'boolean' as the return type. The convention is to name these methods 'isX()'.

解决方案　返回布尔类型的方法命名：发现返回布尔类型的方法被命名为’getX()’，而惯例是命名为‘isX()’的形式。

PMD 规则之 Optimization Rules

- LocalVariableCouldBeFinal: A local variable assigned only once can be declared final.

解决方案　本地变量可以是 Final 的：本地变量只被赋值一次可以声明为 final 的

- MethodArgumentCouldBeFinal: A method argument that is never assigned can be declared final.

解决方案　方法参数可以是 Final 的：传入方法的参数从不被赋值可以声明为 final 的

- AvoidInstantiatingObjectsInLoops: Detects when a new object is created inside a loop

解决方案　避免在循环中实例化对象：本规则检查在循环的内部用 new 创建对象

- UseArrayListInsteadOfVector: ArrayList is a much better Collection implementation than Vector.

解决方案　使用 ArrayList 代替 Vector：ArrayList 是比 Vector 更好的集合类实现

- SimplifyStartsWith: Since it passes in a literal of length 1, this call to String.startsWith can be rewritten using String.charAt(0) to save some time.

解决方案　简化 StartWith：字符串中截取长度为 1 的字串时，可以用高率一点的 String.charAt()代替 String.startsWith()

- UseStringBufferForStringAppends: Finds usages of += for appending strings.

解决方案　使用 StringBuffer 来进行字串 append 操作：查找使用+=来连接字串的不良方式

- UseArraysAsList: The java.util.Arrays class has a "asList" method that should be used when you want to create a new List from an array of objects. It is faster than executing a loop to copy all the elements of the array one by one

解决方案　使用 Arrays 类的 asList 方法：java.util.Arrays 类有一个 asList 方法，当你试图将一个对象数组转换为一个 List 时应该使用这个方法，这比循环从数据拷贝元素快得多。

- AvoidArrayLoops: Instead of copying data between two arrays, use System.arraycopy method

解决方案　避免数组循环：拷贝数组用 System.arraycopy 代替循环拷贝数组元素

- UnnecessaryWrapperObjectCreation: Parsing method should be called directy instead.

解决方案　不必要的包装对象创建：解析方法应该被直接调用

- AddEmptyString: Finds empty string literals which are being added. This is an inefficient way to convert any type to a String.

解决方案　加空字符串：发现+“”操作。这是将其他类型转换为字符串的低效的做法

PMD 规则之 Strict Exception Rules

- AvoidCatchingThrowable: This is dangerous because it casts too wide a net; it can catch things like OutOfMemoryError.

解决方案　避免 catch Throwable 对象：这是危险的因为覆盖的范围太广，它能够 catch 类似于 OutOfMemoryError 这样的错误

- SignatureDeclareThrowsException: It is unclear which exceptions that can be thrown from the methods. It might be difficult to document and understand the vague interfaces. Use either a class derived from RuntimeException or a checked exception.

解决方案　具体声明抛出的异常：不确定方法中能够抛出什么样的具体异常。为模糊的接口提供证明并理解它是很困难的。抛出的异常类要么从 RuntimeException 中继承或者抛出一个被检查的异常。

- ExceptionAsFlowControl: Using Exceptions as flow control leads to GOTOish code and obscures true exceptions when debugging.

解决方案　异常用作流程控制：使用异常来做流程控制会导致 goto 类型的代码，且使得调试的时候发生的真正的异常含糊化。

- AvoidCatchingNPE: Code should never throw NPE under normal circumstances. A catch block may hide the original error, causing other more subtle errors in its wake.

解决方案　避免捕获空指针异常：在正常情形下代码不应该捕获 NullPointException。否则 Catch 块可能隐藏原始错误，导致其他更多微妙的错误。

- AvoidThrowingRawExceptionTypes: Avoid throwing certain exception types. Rather than throw a raw RuntimeException, Throwable, Exception, or Error, use a subclassed exception or error instead.

解决方案　避免抛出原始异常类型：避免抛出特定异常类型。与其抛出 RuntimeException，Throwable，Exception，Error 等原始类型，不如用他们的子异常或子错误类来替代。

- AvoidThrowingNullPointerException: Avoid throwing a NullPointerException - it's confusing because most people will assume that the virtual machine threw it. Consider using an IllegalArgumentException instead; this will be clearly seen as a programmer-initiated exception.

解决方案　避免抛出空指针异常：避免抛出空指针异常——这会导致误解，因为大部分人认为这应该由虚拟机抛出。考虑用 IllegalArgumentException 代替，这对于开发者定义异常来说比较清晰。

- AvoidRethrowingException: Catch blocks that merely rethrow a caught exception only add to code size and runtime complexity.

解决方案　避免重复抛出异常：catch 块仅仅重新抛出一个已捕获的异常只会增加代码量和程序运行的复杂度。

- DoNotExtendJavaLangError: Errors are system exceptions. Do not extend them.

解决方案　不要继承 java.lang.Error:Error 是系统异常，不要继承它。

- DoNotThrowExceptionInFinally: Throwing exception in a finally block is confusing. It may mask exception or a defect of the code, it also render code cleanup uninstable. Note: This is a PMD implementation of the Lint4j rule "A throw in a finally block"

解决方案　不要在 finally 块中抛出异常：在 finally 块中抛出异常易造成迷惑。它将掩盖代码异常或缺陷，也促使代码不稳定。备注：这是 PMD 从 Lint4j 原则衍生实现的

- AvoidThrowingNewInstanceOfSameException: Catch blocks that merely rethrow a caught exception wrapped inside a new instance of the same type only add to code size and runtime complexity.

解决方案　避免抛出同类异常的新实例：catch 块仅仅重新抛出一个同类型捕获异常的新的实例只会增加代码量和运行复杂度。

PMD 规则之 String and StringBuffer Rules

- AvoidDuplicateLiterals: Code containing duplicate String literals can usually be improved by declaring the String as a constant field.

解决方案　避免重复的字面量：代码包含重复的字符串常常可以重构为将此字符串声明为常量

- StringInstantiation: Avoid instantiating String objects; this is usually unnecessary.

解决方案　字符串初始化：避免初始化字符串对象；这是不必要的。

- StringToString: Avoid calling toString() on String objects; this is unnecessary.

解决方案　String.toString():避免对字符串对象调用 toString()方法，这是不必要的

- InefficientStringBuffering: Avoid concatenating non literals in a StringBuffer constructor or append().

解决方案　低效的 StringBuffering: 避免在 StringBuffer 的构造器或 append()方法中连接非字面量类型

- UnnecessaryCaseChange: Using equalsIgnoreCase() is faster than using toUpperCase/toLowerCase().equals()

解决方案　不必要的大小写转换：使用 equalsIgnoreCase()比将字符串大小写转换一致后再比较要快。

- UseStringBufferLength: Use StringBuffer.length() to determine StringBuffer length rather than using StringBuffer.toString().equals("") or StringBuffer.toString().length() ==.

解决方案　使用 StringBuffer 的 length()方法：使用 StringBuffer 对象的 length()方法来计算 StringBuffer 对象的长度，而不是使用 StringBuffer.toString().equals("") or StringBuffer.toString().length() ==.等方法

- AppendCharacterWithChar: Avoid concatenating characters as strings in StringBuffer.append.

解决方案　用 char 类型连接字符：在使用 StringBuffer 的 append()方法连接字符时，避免使用 string 类型。

- ConsecutiveLiteralAppends: Consecutively calling StringBuffer.append with String literals

解决方案　连续的字面量连接：连接字符串时连续的调用 StringBuffer 的 append()方法

- UseIndexOfChar: Use String.indexOf(char) when checking for the index of a single character; it executes faster.

解决方案　使用 indexOf(字符)：当你检测单个字符的位置时使用 String.indexOf(字符)，它执行的很快。不要使用 indexOf(字符串)

- InefficientEmptyStringCheck: String.trim().length() is an inefficient way to check if a String is really empty, as it creates a new String object just to check its size. Consider creating a static function that loops through a string, checking Character.isWhitespace() on each character and returning false if a non-whitespace character is found.

解决方案　低效的空字符串检查：用 String.trim().length()来判断字符串是否空是低效的做法，因为它会创建一个新的字符串对象然后判断大小。考虑创建一个静态的方法循环 String，用 isWhitespace()检查每个字符如果遇到非空白字符就返回 false

- InsufficientStringBufferDeclaration: Failing to pre-size a StringBuffer properly could cause it to re-size many times during runtime. This rule checks the characters that are actually passed into StringBuffer.append(), but represents a best guess "worst case" scenario. An empty StringBuffer constructor initializes the object to 16 characters. This default is assumed if the length of the constructor can not be determined.

解决方案　　不充分的 StringBuffer 声明:如果不能在事前声明合适大小的 StringBuffer 容量可能导致运行期不断地重新分配大小。本规则检查字符事实上传递给 StringBuffer.append(),但是表明了在最坏情况下的最好的预测。空参数的 StringBuffer 构造器默认将对象初始化为 16 个字符的容量。这个默认情况是在构造长度无法确定的情况下假定的。

- UselessStringValueOf: No need to call String.valueOf to append to a string; just use the valueOf() argument directly.

解决方案　　无用的 valueOf 方法：调用 append()方法时不需要把参数用 valueOf()转换一次，直接将非 String 类型的值作为参数放在 append()里面。

- StringBufferInstantiationWithChar: StringBuffer sb = new StringBuffer('c'); The char will be converted into int to intialize StringBuffer size.

解决方案　　StringBuffer 使用字符初始化：StringBuffer sb = new StringBuffer(’c’); 字符 c 会转换为 int 值，作为 StringBuffer 的初始化大小参数。

- UseEqualsToCompareStrings: Using '==' or '!=' to compare strings only works if intern version is used on both sides

解决方案　　使用 equals 方法比较字符串：使用‘==’或‘! =’比较字符串大小只是比较两边的常量池的引用。

- AvoidStringBufferField: StringBuffers can grow quite a lot, and so may become a source of memory leak (if the owning class has a long life time).

解决方案　　避免在类中使用 StringBuffer 属性：StringBuffer 类型变量可以变得非常庞大，所以可能造成内存泄漏。（如果宿主类有很长的生命期）

*PMD 规则之 Security Code Guidelines

- MethodReturnsInternalArray: Exposing internal arrays directly allows the user to modify some code that could be critical. It is safer to return a copy of the array.

解决方案　　方法返回内部数组：暴露内部数组直接允许用户修改的代码会是非常危险的，返回一个数组的 copy 是安全的做法

代码示例：

```
public class SecureSystem {
  UserData [] ud;
  public UserData [] getUserData() {
      // Don't return directly the internal array, return a copy
      return ud;
  }
}
```

- ArrayIsStoredDirectly: Constructors and methods receiving arrays should clone objects and store the copy. This prevents that future changes from the user affect the internal functionality.

解决方案　　数组被直接存储：构造器和方法接收数组应该 clone 对象并保存副本，这会阻止用户将来的改变影响内部的功能。

代码示例：

```
public class Foo {
 private String [] x;
  public void foo (String [] param) {
      // Don't do this, make a copy of the array at least
      this.x=param;
  }
}
```

PMD 规则之 Type Resolution Rules

- LooseCoupling: Avoid using implementation types (i.e., HashSet); use the interface (i.e, Set) instead

解决方案　　松散耦合：避免使用实现类(如 HashSet)，使用接口(如 Set)代替

- CloneMethodMustImplementCloneable: The method clone() should only be implemented if the class implements the Cloneable interface with the exception of a final method that only throws CloneNotSupportedException. This version uses PMD's type resolution facilities, and can detect if the class implements or extends a Cloneable class

解决方案　存在克隆方法类必须实现 Cloneable 接口： clone()方法只有在类实现了 cloneable 接口且伴随一个 final 方法只抛出 CloneNotSupportedException 异常的情况下才要被实现。这个版本使用了 PMD 的类型分析工具，能够检测类是否实现或继承了一个可克隆的类

- UnusedImports: Avoid unused import statements. This rule will find unused on demand imports, i.e. import com.foo.*.

解决方案　无用的 import：避免无用的 import 语句。这个规则按需查找无用的 import 语句。

- SignatureDeclareThrowsException: It is unclear which exceptions that can be thrown from the methods. It might be difficult to document and understand the vague interfaces. Use either a class derived from RuntimeException or a checked exception. Junit classes are excluded.

解决方案　具体声明抛出的异常：不确定方法中能够抛出什么样的具体异常。为模糊的接口提供证明并理解它是很困难的。抛出的异常类要么从 RuntimeException 中继承或者抛出一个被检查的异常。

*PMD 规则之 Unused Code Rules

- UnusedPrivateField: Detects when a private field is declared and/or assigned a value, but not used.

解决方案　未用的私有变量：检测到私有变量被声明或被赋值，但是未使用

- UnusedLocalVariable: Detects when a local variable is declared and/or assigned, but not used.

解决方案　未用的本地变量：检测到本地变量被声明或被赋值，但是未使用

- UnusedPrivateMethod: Unused Private Method detects when a private method is declared but is unused.

解决方案　　未用的私有方法：检测到已声明但未使用的私有方法

- UnusedFormalParameter: Avoid passing parameters to methods or constructors and then not using those parameters.

解决方案　　未用的常规参数：避免传递给方法或构造器不使用的参数