## 整洁的代码

| **清单项目** | **分类** |
| --- | --- |
| 使用可以表达实际意图(Intention-Revealing)的名称 | 有意义的名称 |
| 每一个概念只用一个词 | 有意义的名称 |
| 使用方案/问题领域名称 | 有意义的名称 |
| 类应该是比较小的! | 类 |
| 函数应该是比较小的! | 函数 |
| 只做一件事 | 函数 |
| DRY(Don’t Repeat Yourself)原则，(拒绝重复) | 函数 |
| 用代码来解释自己的做法(译者注：即代码注释) | 注释 |
| 确定应用了代码格式化 | 格式 |
| 使用异常而不是返回码 | 异常 |
| 不要返回Null | 异常 |

*\*参考自:<http://techbus.safaribooksonline.com/book/software-engineering-and-development/agile-development/9780136083238>*

## 安全

| **清单项目** | **分类** |
| --- | --- |
| 如果不用于继承，使类为final | 基础 |
| 避免重复代码 | 基础 |
| 权限限制：程序应该运行在保证功能正常的最小权限模式下。 | 基础 |
| 最小化类和成员的可访问性 | 基础 |
| 注释出安全相关的信息 | 基础 |
| 系统的输入必须检查是否有效和在允许范围内 | 拒绝服务(Denial of Service) |
| 避免对于一些不寻常行为的过分日志 | 拒绝服务(Denial of Service) |
| 在任何情况下都释放资源(流，连接等等) | 拒绝服务(Denial of Service) |
| 从异常中清除敏感信息(暴露文件路径，系统内部相关，配置)P | 私密信息(Confidential Information) |
| 不要把高度敏感的信息写到日志 | 私密信息(Confidential Information) |
| 考虑把高度敏感的信息在使用后从内存中清除 | 私密信息(Confidential Information) |
| 限制包，类，接口，方法和域的可访问性 | 可访问性的扩展(Accessibility Extensibility) |
| 限制类和方法的可扩展性(通过使它为final) | 可访问性的扩展(Accessibility Extensibility) |
| 检验输入(有效的数据，大小，范围，边界情况等等) | 输入检验(Input Validation) |
| 把从不可信对象得到的输出作为输入来检验 | 输入检验(Input Validation) |
| 为native方法定义包装类(而不是定义native方法为pulibc) | 输入检验(Input Validation) |
| 把从不可信对象得到的输出作为输入来对待 | 可变性 |
| 使public static域为final(避免调用方(caller)修改它的值) | 可变性 |
| 避免暴露敏感类的构造函数 | 对象构造 |
| 避免安全敏感类的序列化 | 序列化反序列化(Serialization Deserialization) |
| 通过序列化来保护敏感数据 | 序列化反序列化(Serialization Deserialization) |
| 小心地缓存潜在的特权操作结果 | 序列化反序列化(Serialization Deserialization) |
| 只有在需要的时候才使用JNI | 访问限制 |

*\*参考自: <http://www.oracle.com/technetwork/java/seccodeguide-139067.html>*

## 性能

| **清单项目** | **分类** |
| --- | --- |
| 避免过分的同步 | 并发 |
| 保持同步区域比较小 | 并发 |
| 知道string连接的性能情况 | 综合编程 |
| 避免创建不需要的对象 | 创建和销毁对象 |

*\*参考自: <http://techbus.safaribooksonline.com/book/programming/java/9780137150021>*

## 综合(译者注：原文中的作者把checklist和category对应的列搞错了，译文中已修正)

| **清单项目** | **分类** |
| --- | --- |
| 对可以恢复的情况使用已受检异常(checked exceptions)，对于程序错误使用运行时异常(runtime exceptions) | 异常 |
| 更多地使用标准异常 | 异常 |
| 不要忽略异常 | 异常 |
| 检查参数的有效性 | 方法 |
| 返回空数组或集合，而不是null | 方法 |
| 最小化类和成员的可访问性 | 类和接口 |
| 在pulibc类中，使用访问器方法(accessor methods)(译者注：访问器方法即我们平常用的get/set方法)而不是public域 | 类和接口 |
| 最小化本地变量的范围 | 综合编程 |
| 通过接口引用对象 | 综合编程 |
| 遵循广泛接受的命名规则 | 综合编程 |
| 避免使用finalizer | 创建和销毁对象 |
| 当你重写equals时总是重写hashCode | 综合编程 |
| 总是重写toString | 综合编程 |
| 使用枚举来代替int常量 | 枚举和注解(Annotations) |
| 使用标记接口(marker interface)(译者注：标记接口是一种没有任何行为的接口，实现它只是为了让实现类属于某种类型，如JDK中的Serializable,Cloneable等)来定义类型 | 枚举和注解(Annotations) |
| 对共享可变的数据使用同步访问 | 并发 |
| 使用executors而不是task和thread | 并发 |
| 注释中描述线程安全情况 | 并发 |
| 存在有效的JUnit/JBehave测试用例 | 测试 |

在我们关于高效代码审查的博文中，我们建议使用一个检查清单。在代码审查中，检查清单是一个非常好的工具——它们保证了审查可以在你的团队中始终如一的进行。它们也是一种保证常见问题能够被发现并被解决的便利方式。

软件工程学院的研究表明，程序员们会犯15-20种常见的错误。所以，通过把这些错误加入到检查清单当中，你可以确保不论什么时候，只要这些错误发生了，你就能发现它们，并且可以帮助你杜绝这些错误。

为了帮助你开始创建一个清单，这里列出了一些典型的内容：

### **代码审查清单**

常规项

* 代码能够工作么？它有没有实现预期的功能，逻辑是否正确等。
* 所有的代码是否简单易懂？
* 代码符合你所遵循的编程规范么？这通常包括大括号的位置，变量名和函数名，行的长度，缩进，格式和注释。
* 是否存在多余的或是重复的代码？
* 代码是否尽可能的模块化了？
* 是否有可以被替换的全局变量？
* 是否有被注释掉的代码？
* 循环是否设置了长度和正确的终止条件？
* 是否有可以被库函数替代的代码？
* 是否有可以删除的日志或调试代码？

安全

* 所有的数据输入是否都进行了检查（检测正确的类型，长度，格式和范围）并且进行了编码？
* 在哪里使用了第三方工具，返回的错误是否被捕获？
* 输出的值是否进行了检查并且编码？
* 无效的参数值是否能够处理？

文档

* 是否有注释，并且描述了代码的意图？
* 所有的函数都有注释吗？
* 对非常规行为和边界情况处理是否有描述？
* 第三方库的使用和函数是否有文档？
* 数据结构和计量单位是否进行了解释？
* 是否有未完成的代码？如果是的话，是不是应该移除，或者用合适的标记进行标记比如‘TODO’？

测试

* 代码是否可以测试？比如，不要添加太多的或是隐藏的依赖关系，不能够初始化对象，测试框架可以使用方法等。
* 是否存在测试，它们是否可以被理解？比如，至少达到你满意的代码覆盖(code coverage)。
* 单元测试是否真正的测试了代码是否可以完成预期的功能？
* 是否检查了数组的“越界“错误？
* 是否有可以被已经存在的API所替代的测试代码？

你同样需要把特定语言中有可能引起错误的问题添加到清单中。

这个清单故意没有详尽的列出所有可能会发生的错误。你不希望你的清单是这样的，太长了以至于从来没人会去用它。仅仅包含常见的问题会比较好。

### **优化你的清单**

把使用清单作为你的起点，针对特定的使用案例，你需要对其进行优化。一个比较棒的方式就是让你的团队记录下那些在代码审查过程中临时发现的问题，有了这些数据，你就能够确定你的团队常犯的错误，然后你就可以量身定制一个审查清单。确保你删除了那些没有出现过的错误。（你也可以保留那些出现概率很小，但是非常关键的项目，比如安全相关的问题）。

### **得到认可并且保持更新**

基本规则是，清单上的任何条目都必须明确，而且，如果可能的话，对于一些条目你可以对其进行二元判定。这样可以防止判断的不一致。和你的团队分享这份清单并且让他们认同你清单的内容是个好主意。同样的，要定期检查你的清单，以确保各条目仍然是有意义的。

有了一个好的清单，可以提高你在代码审查过程中发现的缺陷个数。这可以帮助你提高代码标准，避免质量参差不齐的代码审查。