

Clean Code Model

I 基础级

- 格式
 - 横向格式
 - 纵向格式
- 注释
 - 好的注释
 - 不好的注释
- 物理设计
 - 头文件编译自满足
 - 文件设计职责单一
 - 仅包含需要的文件
 - 仅公开用户需要的接口

II 进阶级

- 命名
 - 关注点
 - 风格统一的命名规范
 - 避免在命名中使用编码
 - 名称区分问题域与实现域
- 测试
 - 风格统一的测试场景描述
 - 每个测试用例测试一个场景
 - 一组测试场景封装为一个测试套
 - 尝试使用DSL表达测试场景
- 对象和数据结构
 - 区分数据结构与对象的使用场景
 - 避免在对象中使用getter & setter方法
 - 避免在对象中暴露成员变量

- 避免在数据结构中添加行为
-

III 高阶级

◦ 函数

- 每个函数只做一件事
- 函数内语句同一抽象层次
- 尽量避免三个以上的函数参数
- 区分查询函数与指令函数
- 消除重复的函数

◦ 类

- 设计职责单一的类
- 避免方法过多的类
- 避免方法过多的类（上帝类）
- 避免过多的继承层次

◦ 系统

- 合理的对系统进行分层
 - 定义清晰的模块边界及职责
 - 分离构造与使用
 - 考虑系统性能
-

CleanCode Examples

“Clean Code That Works”，来自于Ron Jeffries这句箴言指导我们写的代码要整洁有效，Kent Beck把它作为TDD（Test Driven Development）追求的目标，BoB大叔（Robert C. Martin）甚至写了一本书来阐述他的理解。

本文是对BoB大叔CleanCode一书的一个简单抽取、分层，目的是整洁代码可以在团队中更容易推行，本文不会重复书中内容，仅提供对模型的一个简单解释，如果对于模型中的细节有疑问，请参考《代码整洁之道》。

I 基础级

基础级主要包括代码格式、注释、物理设计三部分，这三部分比较容易做到，甚至可以制定为团队的编码规范，保证团队代码保持统一风格。

1.1 格式

遵循原则：

- 关系密切内容聚合
- 关系松散内容分隔

注意事项：

- 编码时使用等宽字体
- 设置Tab为4个空格
- 使用统一的编码格式：UTF-8, GB2312, GBK
- 使用统一的代码格式化风格。例如经典风格 K&R, BSD/Allman, GNU, Whitesmiths[^{eclipse内置的四种格式}]
- 控制行宽，不需要拖动水平滚动条查看代码

1.1.1 横向格式

使用空格对内容进行分隔，使用锯齿缩紧对代码段进分隔

反例：

```
public String toString() {
    Point point=new Point();
    StringBuilder sb=new StringBuilder();
    sb.append("A B C D E F G H");
    for(point.x=0;point.x<BOARD_LENGTH;point.x++){
        sb.append('\n').append(point.x + 1);
        for(point.y=0;point.y<BOARD_WIDTH;point.y++){
            sb.append(' ').append(board.get(point).symbol());
        }
    }
    sb.append('\n');
    return sb.toString();
}
```

正例：

```
public String toString() {
    Point point = new Point();
    StringBuilder sb = new StringBuilder();

    sb.append(" A B C D E F G H");
    for (point.x = 0; point.x < BOARD_LENGTH; point.x++) {
        sb.append('\n').append(point.x + 1);
        for (point.y = 0; point.y < BOARD_WIDTH; point.y++) {
            sb.append(' ').append(board.get(point).symbol());
        }
    }
    sb.append('\n');

    return sb.toString();
}
```

1.1.2 纵向格式

使用空行对内容进行分隔，函数或类的方法不要太长，尽量能一览无余

此处插入纵向格式代码对比

1.2 注释

遵循原则：

- 尽量不写注释，尝试用代码自阐述

注意事项：

- 擅用源码管理工具
- 提交代码时，日志要详细
- 确认编译器支持“//”
- “//之后有空格，“//之前有空格

1.2.1 好的注释

1. 法律、版权信息
~~此处插入版权信息代码~~
2. 陷阱、警示
~~此处插入陷阱、警示代码~~
3. 意图解释
~~此处插入意图解释代码~~
4. 性能优化代码
~~此处插入性能优化代码代码~~
5. 不易理解代码
~~此处插入不易理解代码代码~~

1.2.2 不好的注释

1. 日志型注释 -> 使用源码管理工具记录
~~此处插入日志型注释对比代码~~
2. 归属、签名 -> 源码管理工具自动记录
~~此处插入归属、签名对比代码~~
3. 注释掉的代码 -> 删除，使用源码管理工具保存
~~此处插入注释代码代码~~
4. 函数头 -> 尝试使用更好的函数名，更好参数名，更少参数替换注释
~~此处插入函数头对比代码~~
5. 位置标记 -> 删除，简化逻辑
~~此处插入位置标记代码~~
6. 过时、误导性注释 -> 删除
~~此处插入过时、误导代码~~
7. 多余、废话注释 -> 删除
~~此处插入多余、废话代码~~

1.3 物理设计

遵循原则：

- 头文件编译自满足（C/C++）
- 文件职责单一
- 文件最小依赖
- 文件信息隐藏

注意事项：

- 包含文件时，确保路径名、文件名大小写敏感
- 文件路径分隔符使用“/”，不使用“\”
- 路径名一律使用小写、下划线或中划线风格
- 文件名与程序实体名称一致

1.3.1 头文件编译自满足(C/C++)

对于C/C++语言头文件编译自满足，即头文件可以单独编译成功。

~~此处插入头文件编译自满足代码~~

1.3.2 文件设计职责单一

文件设计职责单一，是指文件中对于用户公开的信息，应该是一个概念，避免把不相关的概念糅合在一个文件中，文件间增加不必要的依赖。

~~此处插入头文件编译自满足代码~~

1.3.3 仅包含需要的文件

1. 文件设计时，应遵循最小依赖原则，仅包含必须的文件即可。

~~此处插入头文件编译自满足代码~~

2. 特别的，对于C++而言，可以使用类或者结构体前置声明，而不包含头文件，降低编译依赖。该类依赖被称为弱依赖，编译时不需要知道实体的真实大小，仅提供一个符号即可，主要有：

- 指针
- 引用
- 返回值
- 函数参数

~~此处插入头文件编译自满足代码~~

1.3.4 仅公开用户需要的接口

1. 文件设计时，应遵循信息隐藏原则，仅公开用户需要的接口，对于其他信息尽量隐藏，以减少不必要的依赖。

~~此处插入头文件编译自满足代码~~

2. 特别的，对于C可以使用static对全局变量、函数等进行隐藏，对于支持面向对象语言则使用其封装特性即可。

~~此处插入头文件编译自满足代码~~

II 进阶级

进阶级主要包括命名、测试设计、数据结构及对象设计，该部分要求编码时关注到更多细节，从语义层次提升代码的可理解性。

2.1 命名

命名是提高代码表达力最有效的方式之一。每一个命名，我们都应该抱着谨慎的态度，像给自己孩子取名字一样，为其取一个好名字。好的名字，总能令人眼前一亮，令阅读者拍案叫绝，但好的名字往往意味着更多的思考，更多的尝试，体现着我们对代码的一种态度。随着我们对业务的进一步了解，发现名字不合适时，要大胆的重构他。

遵循原则：

- Baby Names，宁停三分，不强一秒
- Min-length + Max-information
- 结构体/类名用名词或名词短语
- 接口使用名词或形容词
- 函数/方法使用动词或动词短语

注意事项：

- 避免使用汉语拼音
- 避免使用前缀
- 避免包含数据结构
- 避免使用数字序列
- 擅用词典
- 擅用重构工具

2.1.1 关注点

- 文件夹 | 包
- 文件
- 函数 | 类方法 | 类
- 参数 | 变量

2.1.2 风格统一的命名规范

社区有很多种类的命名规范，很难找到一种令所有人都满意，如下规范仅供参考：

```
| Type | Examples |
| namespace/package | std, details, lang|
| struct/union/class | List, Map, HttpServlet |
| function/method | add, binarySearch, lastIndexOfSubList |
| macro/enum/constant | MAX_ERAB_NUM, IDLE, UNSTABLE |
| variable | i, key, expectedTimer |
| type | T, KEY, MESSAGE |
```

团队可以根据实际情况进行改动，但团队内命名风格要一致。

2.1.3 避免在命名中使用编码

在程序设计的历史中，在命名中使用编码曾风靡一时，最为出名的为匈牙利命名法，把类型编码到名字中，使用变量时默认携带了它的类型，使程序员对变量的类型和属性有更直观的了解。

基于如下原因，现代编码习惯，不建议命名中使用编码：

- 现代编码习惯更倾向于短的函数、短的类，变量尽量在视野的控制范围内；
- 业务频繁的变化，变量的类型可能随之变化，变量中的编码信息就像过时的注释信息一样误导人；
- 携带编码的变量往往不可读
- 现代IDE具有强大的着色功能，局部变量与成员变量容易区分

由于历史原因，很多遗留代码仍然使用匈牙利命名法，修改代码建议风格一致，新增代码建议摒弃

~~此处插入匈牙利命名对比代码~~

~~此处插入成员变量前缀对比代码~~

~~此处插入类、接口前缀代码~~

2.1.3 名称区分问题域与实现域

1. 现代程序设计期望程序能很好的描述领域知识、业务场景，让开发者和领域专家可以更好的交流，该部分的命名要更贴近问题域。
~~此处插入DSL代码~~
2. 对于操作实现层面，尽量使用计算机术语、模式名、算法名，毕竟大部分维护工作都是程序员完成。
~~此处插入计算机术语、算法、模式对比代码~~

2.2 测试

整洁的测试是开发过程中比较难做到的，很多团队把测试代码视为二等公民，对待测试代码不想工程代码那样严格要求，于是出现大量重复代码、名称名不副实、测试函数冗长繁杂、测试用例执行效率低下，某一天发现需要花费大量精力维护测试代码，开始抱怨测试代码。

遵循原则：

- F.I.R.S.T原则
- 测试用例单一职责，每个测试一个概念
- 测试分层(UT, CT, FT, ST...), 不同层间用例互补，同一层内用例正交
- 像对待工程代码一样对待测试用例

注意事项：

- 擅用测试框架管理测试用例
- 选择具有可移植性测试框架
- 尝试业务表达力更强的测试框架
- 关注测试用例有效性
- 关注测试用例执行速度

2.2.1 风格统一的测试场景描述

1. Given-When-Then风格

- (Given) some context
- (When) some action is carried out
- (Then) a particular set of observable consequences should obtain
此处插入 Given-When-Then 代码

2. Should-Given-When风格

此处插入 Should-Given-When 代码

2.2.2 每个测试用例测试一个场景

好的测试用例更像一份功能说明文档，各种场景的描述应该职责单一，并完整全面。每个测试用例一个测试场景，既利于测试失败时，问题排查，也可以避免测试场景遗留。

此处插入 测试职责单一 对比代码

2.2.3 一组测试场景封装为一个测试套

所有测试用例不应该平铺直叙，在同一个层次，可以使用测试套将其分层，便于用例理解与管理。

此处插入 测试套封装 代码

2.2.4 尝试使用DSL表达测试场景

尝试使用DSL描述测试用例，领域专家可以根据测试用例表述，判断业务是否正确。测试DSL可能需要抽取业务特征，设计、开发测试框架。

此处插入 测试 DSL 代码

2.3 对象和数据结构

此处不讨论面向对象与面向过程设计范式的优劣，仅区分对象与数据结构使用场景与注意事项。

遵循原则：

- 对象隐藏数据，暴露行为
- 数据结构暴露数据，无行为

注意事项：

- 数据结构与对象不可混用
- 避免在对象中使用getter/setter方法
- 避免在对象中暴露数据
- 避免在数据结构中添加行为

2.3.1 区分数据结构与对象的使用场景

面向对象主要关注“做什么”，关心如何对数据进行抽象；数据结构主要表示数据“是什么”，面向过程主要关注“怎么做”，关心如何对数据进行操作。二者都可以很好的解决问题，相互之间并不冲突。

在使用场景上：

- 若数据类型频变，可以使用面向对象
~~此处插入数据类型频变代码~~
- 若类型行为频变，可以使用数据结构
~~此处插入类型行为代码~~

2.3.2 避免在对象中使用getter & setter

使用面向对象较面向过程的一个很大的不同是对象行为的抽象，较数据“是什么”，更关注对象“做什么”，所以，在对象中应该关注对象对外提供的行为是什么，而不是通过getter&setters暴露数据，通过其他的服务、函数、方法操作对象。如果数据被用来传送（即DTO，Data Transfer Objects），使用贫血的数据结构即可。

~~此处插入getter & setter代码~~

2.3.3 避免在对象中暴露成员变量

面向对象为外部提供某种服务，内部的数据类型应该被封装，或者说隐藏，不应为了访问便利，暴露成员变量，如果需要频繁被调用，请考虑为DTO，使用数据结构。

~~此处插入暴露成员变量代码~~

2.3.4 避免在数据结构中添加行为

数据结构表示数据“是什么”，承载着数据的特征、属性。为数据结构增加一些“做什么”的行为，让数据结构变的不伦不类，也会增加设计的复杂度，不知该封装该数据结构，为其提供一些行为，还是直接调用它的方法。对于特殊的构造函数或者拷贝构造函数、赋值操作符除外。

~~此处插入数据结构中增加行为代码~~

III 高阶级

高阶部分涉及到函数、类、系统的遵循的一些设计原则和一些基本的实现模式，以提升代码的可理解性。

3.1 函数

遵循原则：

- 别重复自己（DRY）
- 单一职责（SRP）
- 同一函数所有语句同一抽象层次

注意事项：

- 避免规定函数的长度

- 避免打着性能的幌子拒绝提取函数
- 避免函数名名不副实，隐藏函数真正意图
- 避免一开始就抽取函数，可以先完成业务逻辑，逐渐重构

3.1.1 每个函数只做一件事

每个函数只做一件事，做好这件事，是单一职责在函数设计中的体现。只做一件事最难的是要做的到底是哪件事，怎么样的函数就是只做一件事的函数，提供如下建议：

1. 函数名不存在and,or等连接词，且函数名表达意思与函数完成行为一致
2. 函数内所有语句都在同一抽象层次
3. 无法再拆分出另外一个函数
~~此处插入单一职责代码~~

3.1.2 函数内语句同一抽象层次

函数内语句在同一抽象层次或相同抽象层次。抽象层次是业务概念，即函数内业务逻辑在同一层级，不能把抽象与细节进行混杂。遇到混杂的函数可以通过提取函数(Extract Method)或者分解函数(Compose Method)的方法将其拆分。

~~此处插入同一抽象层次对比代码~~

3.1.3 尽量避免三个以上的函数参数

函数参数最好无参数，然后是一个参数，其次两个，尽量避免超过三个。太多参数往往预示着函数职责不单一，也很难进行自动化测试覆盖。遇到过多参数函数，考虑是否可以拆分函数或把一些强关联参数封装成参数对象。

~~此处插入过多参数例子代码~~

3.1.4 区分查询函数与指令函数

从数据的状态是否被修改，可以将函数分为两大类：查询函数和指令函数。查询函数不会改变数据的状态；指令函数会修改数据的状态。区分二者区别，需要注意如下：

1. 查询函数考虑使用is, should, need等词增强其语义
2. 查询函数往往无参数或仅有入参，考虑使用const等关键词明确查询语义
3. 忌在查询函数语义函数体内修改数据，造成极大迷惑
4. 指令函数忌用查询语义词汇，查询函数忌用指令操作词汇(set, update, add...)
~~此处插入查询与指令的代码~~

3.1.5 消除重复的函数

函数的第一个遵循原则就是短小，但短小是结果，不是目的，也没有必要刻意追求短小的函数。消除代码中的重复，自然会得到可观长度的函数。重复可谓一切软件腐化的万恶之源，是否是否识别重复、消除重复也是我们软件设计的一项基本功。

~~此处插入重复对比代码~~

3.2 类

面向对象程序设计一个比较大的优势程序的扩展性，本节不会涉及太多关于扩展性建议，主要关注类设计的整洁、可理解性。

遵循原则：

- 别重复自己 (DRY)
- S.O.L.I.D原则

注意事项：

- 避免公开成员变量
- 避免类过多的方法（上帝类）
- 避免过多的继承层次
- 避免将父类强转为子类
- 区分接口实现与泛化

3.2.1 设计职责单一的类

单一职责是类设计中最基本、最简单的一个原则，也是最难正确使用的原则。职责单一的类必然是一些内聚的小类，内聚的小类进一步简化了类与类之间的依赖关系，从而简化了设计。软件设计在一定程度上就是分离对象职责，管理对象见依赖关系。

那么什么是类的职责呢？Bob大叔把它定义为“变化的原因”，职责单一的类即仅有一个引起它变化的原因的类。如何判断一个类是职责单一呢？给出一些建议：

- 类中数据具有相同生命周期
- 类中数据相互依赖、相互结合成一个整体概念
- 类中方法总是在操作类中数据

3.2.3 避免方法过多的接口

接口隔离原则（ISP）就是避免接口中绑定一些用户不需要的方法，避免用户代码与该接口之间产生不必要的耦合关系。接口中虽然没有数据，但是根据其行为职责及用户的依赖，将其进行拆分。清晰的接口定义不但可以减少不必要的编译依赖，还可以改善程序的可理解性。

3.2.3 避免方法过多的类（上帝类）

方法过多的上帝类，预示该类包含过多职责，类之间存在着大量的重复，不便于设计的组合，需要对该类进行进一步抽象，拆分成更多职责单一，功能内聚的小类。

3.2.4 避免过多的继承层次

继承关系包括接口继承（实现）、类继承（泛化）两种，接口继承即依赖于抽象，方便程序的扩展；类继承便于复用代码，消除重复，但是设计中过多的继承层次，往往导致设计逻辑的不清晰，建议继承层次不要太深，另外，可以考虑使用组合方案替代继承方案（比如使用策略模式替代模版方法）。

3.3 系统

遵循原则：

- 分而治之
- 层次清晰

注意事项：

3.3.1 合理的对系统进行分层

一个设计良好的系统，必然是一个层次清晰的系统。分层方法可以参考业界常用方法，比如领域驱动设计（DDD）将其分为：表示层、应用层、领域层、基础设施层。

3.3.2 定义清晰的模块边界及职责

分层结构还依赖于层与层之间边界、接口、职责清晰。

3.3.3 分离构造与使用

分离构造与使用，即将对象的创建与对象的使用分离，是降低软件复杂度的常用方法，对应领域驱动设计（DDD）中使用工厂（Factory）创建对象，使用仓库（Repository）存储对象。

3.3.4 考虑系统性能

系统性能是不同与功能的另一个维度，在软件设计过程中，不宜过早的考虑性能优化，但是切忌设计中带来明显的性能劣化。