代码整洁之道

# 有意义的命名

|  |  |
| --- | --- |
| 名副其实 | 命名要精准 |
| 避免误导 | 比如accountList 如果该变量不是List类型，就会产生误导，accountGroup或accounts更好 |
| 有意义的区分 | 如果出现意思相近的变量等命名，我们要通过命名把不同之处表达出来，比如accountData和accountInfo，theMessage和Message，他们对于一个新来的读者来说，根本没有区别，只有详细的阅读了代码之后，才能知道他们俩的不同之处在哪，顾该代码缺少可读性 |
| 使用读得出来的名称 | 便于与同事之间语言上讨论分析代码 |
| 使用可搜索的名称 | 工作中，同事的代码就有这种问题，比如p,p1，等在搜索的时候，因为名称太简单且常见，搜索的结果一大堆，无法定位自己想要查找的代码行，全局修改的时候，无法精准定位，会修改到很多不想修改的位置 |
| 避免使用编码 |  |
| 不要使用成员前缀 | 1. 成员前缀对于读者来说毫无意义 2. 对于开发者，完全可以用开发工具，给成员变量上不同颜色来区分   终上所述，成员前缀就是费料，应该抛弃 |
| 接口与实现 | 接口：ShapeFactory  实现类: ShapeFactoryImpl / CShapeFactory  作者的喜好 |
| 避免思维映射 | 不应当让读者在脑中把你的名称翻译为他们熟知的名称。比如url，你本意是并非网络编程的url，你确命名为url，读者看到这个后会自动映射，产生误导 |
| 类名 | 应该尽量使用名词或名词短语，而不要用动词 |
| 方法名 | 应该是动词或者动词短语 |
| 别扮可爱 |  |
| 每种概念使用一个词 | 比如Manager或者Controller，get和select，我们应该选用其中一个并一以贯之 |
| 别用双关语 | 我们应时刻遵循一词一意 |
| 使用解决方案领域名称 | 使用计算机领域术语 |
| 使用源自所涉及问题领域的名称 | 使用所涉及问题领域的术语 |
|  |  |

# 函数

|  |  |
| --- | --- |
| 每个函数一个抽象层（只做一件事） | 每个函数在同一层抽象层中，他只做一件事 |
| 关于switch代码块的优化 | 我们可以通过抽象工厂和策略模式对switch代码块进行隐藏，让其埋在底层代码中 |
| 使用描述性的名称 |  |
| 函数参数尽量少，尽量避免3参或以上 |  |
| 一元函数的普遍形式 | 转换形式：A transfer （B b）  判断形式：boolean isTrue（B b）  事件形式：void doSomething（B b）  我们不应该设计一下函数  void transfer（B b）明明是转换函数，却没有返回值，这就非常令人费解 |
| 标识参数 | 尽量避免标识参数，这就是公然宣称自己违反了一个函数做一事的原则。  如果有这种情况，我们应该把它分成两个函数  例外情况：对于给第三方调用的函数，我们可以用标识参数，底层我们实现时则分两个函数进行实现即可 |
| 二元函数 | 1. 合格的二元函数，两个参数之间应该是有一定自然顺序的，比如求长方形面积，必然有x,y代表长和宽 2. 如果两个参数之间没有自然顺序或者逻辑关系，那么我们就要思考如何把二元函数转化为一元，什么无参函数 3. 把该方法封装为其中某个参数的内部方法 4. 分离出新的类，然后把参数以构造器的方式传入 |
| 三元函数 | 尽量避免三元函数 |
| 其他减元方法：参数对象 | 通过对参数进行逻辑分类，把部分参数封装成对象，如果x,y 封装成Point |
| 动词与关键字 | 定义方法时，应该尽量满足：  一元函数   1. 方法名为动词 2. 参数名为名词   如：writeField（String name）  二元函数  可以在1上面的基础上，把参数名加入到函数名中，比如assertExpectedEqualsActual（expected，actual）表达了参数顺序同时，也描述清楚了方法用途 |
| 尽量避免输出参数的情况 | 在面对对象的世界里，修改对象状态时，我们应该尽量避免输出参数，转而把状态封装在类中，在内部进行修改 |
| 分隔指令与询问 | 函数要么做什么事，要么就回答什么事  千万不要又做事，又回答  比如：  public boolean set(String attriburte,String value)  这个函数就非常奇怪，是代表说检查是否设置了某值，还是成功设置某值呢？同时，这样做也违反了只做一件事的原则 |
| 使用异常代替返回错误码 | 返回错误码，需要客户端立即解决，而如果抛异常，客户则可以自行定义究竟在哪层代码层去捕捉并执行该代码  由此客户端代码会变得更简洁，入侵性更小 |
| 抽离try catch代码块 | try catch代码块会搞乱代码结构，把错误处理与正常流程混为一谈  通过在增加一层函数嵌套，抽离try catch代码块至更高或更低的一层抽象层函数中 |
| 错误处理就是一件事 |  |
| 别重复自己 | 不要重复写写功能相似的函数 |
| 如何写出优秀的函数 | 没有人可以一步到位  好的代码一开始都是粗陋无需的，都是通过一步步的推敲，思考，优化，打磨，最后成为一个优秀的函数 |

# 注释

|  |  |
| --- | --- |
| 注释不能美化糟糕的代码 |  |
| 应该尽量用代码进行表达，其次才是注释 | 因为注释也需要维护，也有可能是错误的 |
| 好的注释就是你想办法不去写的注释 | 好的注释，读者无需看其他地方的代码，即可准确获取所有信息，注释要解释到位，讲得仔细，如果读者看完还是云里雾里，还不如不写这个注释 |
| 用于表达意图的解释 |  |
| 用于记录警示 |  |
| 利用好todo，记录未来应该做的事 |  |
| 公共API需要javadoc |  |
| 不应该每个参数都加参数，这都是废话 |  |
| 能用函数或变量时，就别用注释 |  |
| 归属/署名 用Git记录即可 | 如果用注释，随着时间推移，归属和署名和作者将越来越没有半毛钱关系 |

# 格式

|  |  |
| --- | --- |
| 垂直顺序 | 调用函数在被调用函数上面 |

# 对象和数据结构

|  |  |
| --- | --- |
| 面对对象和面向过程两种思想是对立的 | 面对对象：增加数据结构（类）容易，增加函数却难，因为多态，所有相关类都需要修改  面对过程：增加函数简单，增加数据结构（类）却很难，一旦增加，全部函数都改改 |
| 最少知识原则 | 只适用于对象  不适用于数据结构  应用于对象就是尽量把对象细节进行隐藏，客户端直接通过对象就能获取到想要信息，无需通过获取内部成员，然后再调用成员的API获取某些结果 |
| 简单的数据结构真的需要get/set方法？  这值得思考 |  |
| 混杂对象 | 混杂对象：一半是对象，一半是数据结构，  即有函数的同时，还用get/set暴露了私有域  应该尽量避免这种对象，这是乱七八糟的设计 |
| 学会隐藏数据结构 | 比如：当你从一个对象获取一个内部数据结构或成员域，我们就要思考，我们拿这个数据最终是要什么事，这件事能否交给该对象去做？  最好的结果是，我们新定义一个函数，把这件事给做了 |
| 数据结构不应该增加业务函数 | 一旦增加业务函数，就成了一个混杂的对象，如果真有这种业务场景，我们可以通过组成，新定义一个类，来隐藏该数据结构，同时，提供相对应的函数 |

# 错误处理

|  |  |
| --- | --- |
| 使用异常而非返回码 |  |
| 对于应用级，应该使用非受控异常 | 受控异常意味着，一旦加入，如果不catch处理，上层所有API都要加上throw签名，违反了开闭原则 |
| 依调用者需要定义异常类 | 我们通过封装第三方API，对第三方该API抛出的异常进行转化，把抛出的异常封装成我们想要的样子 |
| 特例模式 | 处理必须立即catch做处理的异常 |
| 别返回null值 | 用特例模式解决 |
| 别传递null值 | 尽量不要定义可以传null的方法，这样调用者就可以跟调用者约定，如果传null就必定存在问题 |
|  |  |
|  |  |

# 边界

|  |  |
| --- | --- |
| 利用自定义边界类，对边界接口（第三方API）进行封装 | 由此把应用程序代码和边界接口进行隔离 |
| 如果不想长期绑在旧版本上，我们在引入第三方API时，应该做学习性测试 | 好处：   1. 学习型测试，测试需要的功能同时，固化需求功能的测试用例，利于分析版本升级后功能是否有影响 2. 测试过程中，提高我们对API的理解 3. 可以把测试结果，迁移移植到我们边界类，降低对开发进度的影响 |
| 使用尚不存在的代码：预先定义边界类，后期在通过适配器模式进行实现 |  |
| 1. 自定义边界类 2. 适配器模式   都是非常有效的解耦客户端代码和边界接口（第三方API）的手段 |  |
|  |  |
|  |  |
|  |  |
|  |  |

# 单元测试

|  |  |
| --- | --- |
| 整洁的测试 | 测试用例骨架应该是：  构造（封装测试数据）  -  操作（执行收测代码）  -  检验（检验结果） |
| 双重标准 (测试重于可读性) | 测试用例和开发代码标准不一样，要高效，在大多数情况，代码可读性比性能更重要 |
| 测试断言应尽量少，每个测试只测试一个概念 |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |

# 类

|  |  |
| --- | --- |
| 函数的编排：自顶向下原则 | 公共静态变量  私有静态变量  私有变量  公共函数-1  被调用函数1-1  公共函数-2  被调用函数2-1 |
| 类应该短小  如何判断：通过类的责权去判断  怎么算短：符合单一权责原则的算短 | 我们的系统应该有许多小的类组成，而不是少量的大类组成。  前者就好像有许多小抽屉，我们分门别类把工具放进这些小抽屉中  后者就像少量的大抽屉，然后把工具随便怼到这些大抽屉中 |
| 内聚性要高 | 理解内聚  类中每个变量都被每个方法所使用，这种内聚性就最大 |
| 保持内聚性必然会得到很多短小的类 | 当变量在扩展过程中变多，而只有某一部分的方法集使用了某一个新变量，这个时候，就考虑是否需要分离出新的类，以维护类的高内聚性 |
| 类与类之间的组织 | 应该利用好多态，通过组合类实现的接口进行组织，依赖抽象，而不应该依赖具体实现  符合开闭原则，同时利于测试 |
|  |  |
|  |  |
|  |  |

# 系统

|  |  |
| --- | --- |
| 模块与模块之间，要以单一职责去划分 | 比如：工厂模块、和启动模块、业务模块 |
| 模块自身，要利用AOP，横贯切分，分成多个切面进行维护，并且在切面越发复杂的时候，对切换进行模块化 | 比如：缓存，日志，数据预备，等等 |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |

# 迭进

|  |  |
| --- | --- |
| 迭进的原则   1. 有测试且运行所有测试 2. 重构：提升内聚性、降低耦合度、切分切面、模块化系统性切面、缩小类和函数的长度、选用好的名称   目的：消除重复、保证表达、缩短类和函数，提高系统可维护性   1. 不可重复 2. 表达力 3. 尽可能少的类和方法（优先级最低）   测试、消除重复和表达力才是最重要到，在这个基础上，我们可以根据实际情况，尽可能减少类和方法 |  |

# 并发

|  |  |
| --- | --- |
| 并发是一种解耦策略，把做什么（目的）和怎么做（解耦） |  |
| 并发防御原则   1. 单一权责原则 2. 限制数据作用域（特别是共享数据） 3. 使用数据副本 4. 线程尽可能独立，通过设计本地变量，而无需从共享变量中获取 |  |
| 利用好单一权责原则  把线程代码和非线程代码进行解耦 |  |
| 提供可插拔式的多种多线程的方案，便于测试和策略变更 |  |
| 自动化技术，插入线程异动代码，进行测试，提前暴露多线程代码的BUG |  |
|  |  |
|  |  |
|  |  |

# 逐渐改进

|  |  |
| --- | --- |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |

蓝色：查阅spring源码，看看是如何应用的

不同概念的代码，比如对象和数据结构，对象与对象之间，数据结构与数据结构之间，尽可能不要有直接关系，如果非要建立关系，优先考虑通过组合建立关系，而非上来就把代码写在一起，从而建立起强耦合关系

编码过程涉及的元素实体与关系实体：

变量层

1. 成员变量

方法层

1. 函数
2. 错误的处理

代码层

1. 注释
2. 格式
3. 对象
4. 类
5. 过程（比如Util类，通过入参出参去做某些事情）
6. 数据结构
7. 对象与数据结构的关系
   1. 代码要相互隔离并用组合来进行组织
   2. 对象要始终通过定义函数来隐藏数据结构
8. 对象与对象的关系

最少知识原则

1. 数据结构和数据结构的关系
2. 过程与数据结构的关系
   1. 双方通过入参出参形成关联关系
3. 单元测试

系统层

边界