# 编码基本功训练-编码流程方法工具规范

## 简介

采用的编码流程可分为五步：

|  |  |  |  |
| --- | --- | --- | --- |
| 步骤 | 方法 | 用途 | 操作要点 |
| 1 | 设计改善 | 改善设计，目的是提升代码的模块性，让代码更容易理解，更容易调试，测试，修改，复用，扩展。本步骤输出函数的原型定义 | 1. 数据隐藏；  2. 换位思考，化身用户，编写demo，根据demo改善API设计； |
| 2 | 单元测试 | 在编写函数实现代码前，先编写函数的单元测试代码，利用单元测试来调试函数，降低函数调试的难度和工作量 | 1. 测试驱动开发，先写单测代码；  2.每个单测案例只测试一种情况，减少调试定位错误的范围；  3. 案例完善，把各种情况测试完全；  4. 结果检查完善，保证过程中如果出现错误，可以自动发现 |
| 3 | 覆盖率分析 | 在函数编写实现代码之后，运行单测，产出覆盖率数据，根据覆盖率数据衡量单测质量，增补新的单测用例 | 1. 覆盖率达到80%以上，越高越好2. 没覆盖到的代码行，要分析是否可以通过补充用例进行覆盖，如果可以，则补充用例，重新运行 |
| 4 | valgrind检测 | 对于C/C++，单元测试无法发现内存泄漏、溢出、线程类问题，利用valgrind对单测运行过程进行检查，排除这几类问题 | 1.valgrind是动态调试工具，无法发现未运行代码中的问题，所以要和单测、覆盖率分析配合，保证尽可能多的逻辑都运行过；  2. valgrind无法100%发现溢出、泄漏类问题，只能是辅助工具 |
| 5 | codereview | 对于设计类问题，以及只在特定异常场景发生的错误，单测、valgrind检测都无法完全消除，通过代码review，认为排除这一类的问题 | 1. 尽量先通过单测、valgrind检测先排除逻辑类问题；  2. 关注设计结构类问题；  3. 关注特定异常场景的处理措施； 4. 关注模块化，可调试、可测试、可修改、扩展、复用 |

编写函数实现代码发生在第二步“单元测试”和第三步“覆盖率分析”之间。

## 详解

### 1. 设计改善

单元测试是提升代码质量的最有效方法，但是要保证单测的效率和效果，需要从设计开始下功夫。

设计做得不好，API职责不够单一，解耦不完全，会导致函数依赖环境，依赖其它模块，难以进行单元测试，需要打桩，需要构造复杂的数据。

复杂的单测案例，测试效果更差，需要大量时间编写维护，而且后续设计变化后，需要跟着修改，很难持续使用。

设计的改善以API为中心，API，即一个模块对外暴露的使用接口。设计得好的API，可以对外屏蔽模块内部实现细节，呈现出一个易于使用，易于定位错误的使用界面。

API足够的稳定，能适应需求的变化，意味着针对这些API编写的测试用例就无需频繁改动。

API的改进，可以采用demo作为检验工具，对API的语义、使用细节进行检验，以修正原始设计上存在的不足。

完成API的设计初稿后，转换自己的视角，以用户的身份角度，利用这些API实现一个简单但典型的场景功能。在实现过程中，分析是否能对API进行简化，并尝试找到一些灰色地带，把API的语义明确下来，比如内存到底什么时候释放，参数的取值范围，前后置条件，等等。

### 2. 单元测试

单元测试不仅仅用来测试验证代码正确性，还可以作为一种高效的调试手段，替代单步调试定位BUG的落后方法。

所谓测试驱动开发（TDD），就是这种通过单元测试来驱动开发流程，改善代码设计、实现、调试等系列过程的一种方法。

单步调试定位BUG，是一种基本的代码调试方法，能够清晰观察代码执行过程，分析BUG准确位置。为什么还说它落后？这里主要有几个原因：

* 单步调试很难用于大范围的bug定位；
* 单步调试无法用于高性能场景的问题定位；
* 单步调试会改变程序执行逻辑，难以定位某些多线程，多进程同步问题；

单步调试一般用于大致定位问题所在后，分析BUG的准确位置和成因。

用单元测试进行错误调试，有几个好处：

* 单元测试每次只运行某个场景，可以减少BUG出现时需要排查的代码量；
* 单元测试可以自动报告问题爆发的原因和位置，通过这些信息比较容易反推出错误引入点；
* 单元测试具有幂等性，复现bug具有比较高的确定性，可以反复运行，配合单步调试精确定位问题引入点；

单元测试要产生效果，必须保证：

* 测试场景覆盖完全，代码逻辑覆盖完全；
* 结果检查充分，不漏过任何一个问题；
* 错误自动报告，不依赖人工分析；

### 3. 覆盖率分析

单元测试要产生效果，需要保证场景覆盖完全，代码逻辑覆盖完全。评审是一种保证测试案例质量的方法。

除了评审，还有一种方法可以显著提升单元测试案例的质量，即，覆盖率分析。这种方法相比评审，有个特别的好处：可以用于自检，不依赖项目组其它人员的介入。

编译的时候加入生成覆盖率数据的选项(比如编译时加-g -fprofile-arcs -ftest-coverage,链接时加-lgcov)，运行单元测试之后，可以用gcov或者lcov生成覆盖率数据。

分析覆盖率数据，可以发现有些代码行从来没有运行过，简单分析原因，可以判断这是由于案例缺失造成，或者场景难以构造需要打桩造成。

根据这些数据，可以做到有的放矢，精确补充用例。

### 4. valgrind检测

逻辑上的错误，检查输出数据，即可发现。但内存泄漏和溢出等错误，属于意外错误，没有可移植的方法自动检测。在linux下，有一个工具叫valgrind，可以检测出程序运行过程中发生的大部分内存泄漏问题和指针非法操作问题。

valgrind采用虚拟运行原理，能精确指出出现问题的代码位置，有了这些信息，很容易反推错误引入的代码位置。从而快速定位消除bug。

### 5. CodeReview

单元测试加valgrind检测，可以消除大多数逻辑问题。但是它们没办法识别设计/结构类问题，没办法发现设计隐患。

这些问题，基本只能靠经验老道的编码人员肉眼识别。

这也是CodeReview无法被完全替代的原因。它可以作为一种兜底手段，解决其它手段无法完全消除的各种终极问题。