

3-1 代码审计分析与优化

2019年4月14日 10:27

[2 代码的错误与缺陷](#)

[3 代码复审：静态程序分析](#)

[4 性能分析：动态程序分析](#)

2 代码的错误与缺陷

FBrooks对编程的隐喻：焦油坑(TheTarPit)

- 从单一程序到软件系统过程中，所造成复杂度的快速上升，期间并需要包含不同的活动与技能，使得软件开发必须面对多样性的挑战。
- 软件系统产品(ProgrammingSystemsProduct)开发的工作量是供个人使用的、独立开发的构件程序的9倍；
 - 将程序产品化引起了3倍工作量、将构件整合为完整系统所需的设计、集成、测试又强加了3倍工作量。
 - 过去10余年的大型系统开发就犹如陷入了焦油坑，只有极少数的项目满足了目标、进度和预算的要求。
 - 各种开发问题相互纠缠和累积，团队的行动变得越来越慢，最后沉到了坑底。
- Brooks最早认识到设计程序、开发软件的差别，他以程序与系统、产品的差异，解释两者之间的不同，并以 3×3 的复杂度加以说明。
- 编程的乐趣：满足内心深处的创造渴望和愉悦是所有人的共同情感。
 - 创建新事物；
 - 开发对其他人有用的东西；
 - 将相互啮合的零部件组装成整体的过程；
 - 面对不重复的任务，不断学习的乐趣；
 - 纯粹的思维活动，完全不同于对实际物体的驾驭；
- 编程的苦恼：将做事方式调整到“追求完美”。
 - 有其他人来设定目标，而且要依靠自己无法控制的事物；
 - 编程的创造性伴随着枯燥艰苦的劳动；
 - 项目越接近结束，项目所面临的问题和困难就越多；
 - 产品在完成前总是面临着陈旧过时的威胁。

代码规范

- 形式规范：使代码“看起来很美”

- 面向代码的可读性、可维护性、可移植性
- 内容规范：使代码“运行起来很美”
 - 面向代码的正确性、可靠性、运行效率

形式规范

- 标准：简明，易读，无二义性。
- 例如：
 - 缩进、对齐
 - 空行、行内空格
 - 行宽与分行
 - 括号
 - 变量和函数命名
 - 下划线/大小写
 - 注释
 - etc

形式规范：空行

- 在源代码中适度的使用空行可以使程序结构更加清晰
- 两个空行分隔一般出现在源文件的各个节之间；
- 单个空行分隔一般出现在：
 - 方法定义之间；
 - 局部变量定义和第一个语句之间；
 - 注释之前；
 - 语句的不同逻辑分段之前；

形式规范：空格

- 一行代码只写一条语句，避免使用复杂的语句行；
- 行内空格一般出现在：
 - 关键字和括号之间；
 - 参数列表中的逗号之间；
 - 二元运算符之间；
 - 语句中的表达式之间；
 - 赋值符号之间；

```

while(true) {
    ...
}

a+=c+d;

while(d++=s++) {n++;}

printf("size is "+foo+"\n");

for(expr1;expr2;expr3)

myMethod((int)(cp+5),((int)(i+3)+1)
);

```

形式规范：分行

- 一个代码行最大长度最好控制在70~80个字符之内，超过这个范围应分成多行；
- 分行的位置应在逗号之后、操作符之前；
- 长表达式应在低优先级的操作符处拆分成新行，操作符放在新行之首；
- 拆分出的新行要进行适当的缩进(一般为8个空格)，使排版整齐，语句可读。

```

var =
someMethod1(longExp1,someMethod2(longExp2,
    longExp3));

longName1 = longName2 * (longName3 + longName4
    -longName5) + 4 * longname6;

if ((condition1 && condition2) || (condition3
&&
    condition4) || !(condition5 && condition6))
{
    doSomethingAboutIt();
}

```

```

for (int i = 1; i <= n-1; i++)
{ int t = i; for (int j = i+1; j
<= n; j++) { if ( a[j] < a[t]) {
t = j; if (t <> i) { int work =
a[t]; a[t] = a[i]; a[i] =
work; }}}}

```

```

for (int i = 1; i <= n-1; i++) {
    int t = i;
    for (int j = i+1; j <= n; j++)
    {
        if ( a[j] < a[t]) {
            t = j;
            if (t <> i) {
                int work = a[t];
                a[t] = a[i];
                a[i] = work;
            }
        }
    }
}

```

形式规范：命名规则

标识符：类名、变量名、函数/方法名、参数名

- 命名应当直观，可以“望文生义”，最好采用英文单词或其组合，应能反映它所代表的实体，意义明确；
- 命名的长度应符合“最小长度下的最大信息量”原则，过长的英文单词应采通用而合理的缩写；
- 一般禁止使用单字符变量名(局部循环变量除外)；
- 不要仅依靠大小写来区分相似的标识符；
- 不要出现局部变量和全局变量重名的现象；
- 变量名：使用“名词”或“形容词+名词”的形式；
- 函数/方法名：使用“动词”或“动词+名词”的形式。
- 类的命名采用以大写字母开头的单词组合而成；
- 常量名采用全大写的字母单词组合而成，单词之间用下划线分隔；
- 变量名和参数名采用第一个单词首字母小写而后面的单词首字母大写的单词组合；
- 可以使用前缀s_表示静态变量，使用g_表示全局变量，使用m_表示类的成员变量，使用p_表示指针，使用b_表示布尔类型的变量；
- 函数/方法名采用第一个单词首字母小写而后面的单词首字母大写的单词组合。

```

class CourseOffering {

    static final int MIN_WIDTH = 4;
    static final int GET_THE_CPU = 1;

    int g_numStudents;
    int m_width, m_height;

    void setValue ( int width, int height ) {
        m_width = width;
    }
}

```

```

class CourseOffering {

    static final int MIN_WIDTH = 4;
    static final int GET_THE_CPU = 1;

    int g_numStudents;
    int m_width, m_height;

    • void setValue ( int width, int height ) {
        m_width = width;
        m_height = height;
    }
    void calcAvgScore() {
        int i;
        for ( i=0; I < g_numStudents; i++) {...}
    }
}

```

形式规范：声明

- 尽量在变量声明时进行初始化，只有当初值依赖于某些计算得到的值时可以不对其进行初始化；

```
private boolean isDirty = false;
```

```

○ private boolean isDirty;
...
isDirty = getState();

```

- 变量声明仅放在代码块的开头，除了循环语句之外，一般不要在首次使用时才进行声明；

```

void myMethod () {
    int int1 = 0; //在函数块前面声明变量
    if (condition) {
        int int2 = 0; //在if语句块前面声明变量
    }
    ...
}

//for循环语句中进行循环变量的声明和初始化
for(int i=0; i<maxLoops; i++) {...}

```

- 每行只声明一个变量，尤其避免同一行内声明不同类型的变量；

```

○ int level, size;
  int foo, foo_array[];

```

- 变量名不要重复；

```

    int count;
    ...
    myMethod() {
        if (condition) {
            int count;
            ...
        }
    }

```

形式规范：语句结构

- 避免使用空的else语句和嵌套的条件判断语句
- 避免采用过于复杂的条件测试
- 尽量减少使用“否定”条件的条件语句
- 避免书写复杂的表达式

形式规范：注释

- 三种格式：
 - 文档风格：一般直接写在类、方法、函数的前面；
 - C语言风格：采用“/*...*/”格式，通常是多行注释，可以用来描述算法等多行的文字；
 - 单行注释：采用“//...”格式，简单说明该行语句的作用。
- 文档风格的注释：通常置于每个程序模块的开头部分，给出程序的整体说明，对于理解程序本身具有引导作用：
 - 程序标题；
 - 有关本模块功能和目的的说明；
 - 主要算法；
 - 接口说明：包括调用形式，参数描述，子程序清单；
 - 有关数据描述：重要的变量及其用途，约束或限制条件，以及其它有关信息；
 - 给函数和全局数据加注释
 - 模块位置：在哪一个源文件中，或隶属于哪一个软件包；
 - 开发简历：模块设计者，复审者，复审日期，修改日期及有关说明等。
- 注意
 - 程序中不能没有注释，但注释也不能过多；
 - 不必要注释含义已经十分清楚的代码；
 - 修改代码时应该同时修改注释，以保证注释和代码的一致性
 - 注释应当准确易懂，防止出现二义性
 - 注释的位置应该与被描述的代码相邻，应该写在程序代码的上方并且和代码左对齐
 - 变量定义和分支语句(条件分支、循环语句等)必须写注释，因为这些语句往往是程序实现某一特定功能的关键。

内容规范

- 代码设计规范不光是程序书写的格式问题，而且牵涉到程序设计、模块之间的关系、设计模式等方面。
- 几个例子：
 - 函数：现代程序设计语言中的绝大部分功能，都在函数(Function,Method)中实现，关于函数最重要的原则是：只做一件事，但是要做好(单一责任原则)。
 - 错误处理：80%的程序代码，都是对各种已经发生和可能发生的错误的处理，如果你认为某事可能会发生，这时就要用错误处理。
 - 参数处理：所有的参数都要验证其正确性，使用“断言”(Assert)；

类的编码规范

- 类
 - 使用类来封装面向对象的概念和多态 (Polymorphism) 。
 - 避免传递类型实体的值，应该用指针传递。换句话说，对于简单的数据类型，没有必要用类来实现。
 - 对于有显式的构造和析构函数，不要建立全局的实体，因为你不知道它们在何时创建和消除。
 - 只有在必要的时候，才使用“类”。
- Class vs Struct
 - 如果只是数据的封装，用Struct即可。
- 公共/保护/私有成员Public、Private和Protected
 - 按照这样的次序来说明类中的成员：public、protected、private
- 数据成员
 - 数据类型的成员用m_name说明。
 - 不要使用公共的数据成员，要用inline访问函数，这样可同时兼顾封装和效率。
- 虚函数VirtualFunctions
 - 使用虚函数来实现多态 (Polymorphism) 。
 - 只有在非常必要的时候，才使用虚函数。
 - 如果一个类型要实现多态，在基类 (BaseClass) 中的析构函数应该是虚函数。
- 构造函数Constructors
 - 不要在构造函数中做复杂的操作，简单初始化所有数据成员即可。
 - 构造函数不应该返回错误（事实上也无法返回）。把可能出错的操作放到HrInit()或FInit()中。
- 析构函数
 - 把所有的清理工作都放在析构函数中。如果有些资源在析构函数之前就释放了，记住要重置这些成员为0或NULL。

- 析构函数也不应该出错。
- New和Delete
 - 如果可能，实现自己的New/Delete，这样可以方便地加上自己的跟踪和管理机制。自己的New/Delete可以包装系统提供的New/Delete。
 - 检查New的返回值。New不一定都成功。
 - 释放指针时不用检查NULL。
- 运算符（Operators）
 - 在理想状态下，我们定义类不需要自定义操作符。只有当操作符的确需要时。
 - 运算符不要做标准语义之外的任何动作。例如，“==”的判断不能改变被比较实体的状态。
 - 运算符的实现必须非常有效率，如果有复杂的操作，应定义一个单独的函数。
 - 当你拿不定主意的时候，用成员函数，不要用运算符。
- 异常（Exceptions）
 - 异常是在“异乎寻常”的情况下出现的，它的设置和处理都要花费“异乎寻常”的开销，所以不要用异常作为逻辑控制来处理程序的主要流程。
 - 了解异常及处理异常的花销，在C++语言中，这是不可忽视的开销
 - 当使用异常时，要注意在什么地方清理数据。
 - 异常不能跨过DLL或进程的边界来传递信息，所以异常不是万能的
- 类型继承（ClassInheritance）
 - 当有必要的时候，才使用类型继承。
 - 用Const标注只读的参数（参数指向的数据是只读的，而不是参数本身）
 - 用Const标注不改变数据的函数。

3 代码复审：静态程序分析

代码复审(Review)

代码复审：代码是否在“代码规范”的框架内正确地解决了问题

名 称	形 式	目 的
自我复审	自己 vs. 自己	用同伴复审的标准来要求自己。不一定最有效。因为开发者对自己总是过于自信。如果能持之以恒，则对个人有很大好处
同伴复审	复审者 vs. 开发者	简便易行
团队复审	团队 vs. 开发者	有比较严格的规定和流程，用于关键的代码，以及复审后不再更新的代码。 覆盖率高——有很多双眼睛盯着程序。但是有可能效率不高（全体人员都要到会）

代码复审的目的

- 找出代码的错误：
 - 编码错误，比如一些能碰巧骗过编译器的错误。
 - 不符合项目组的代码规范的地方。
- 发现逻辑错误，程序可以编译通过，但是代码的逻辑是错的。
- 发现算法错误，比如使用的算法不够优化。
- 发现潜在的错误和回归性错误——当前的修改导致以前修复的缺陷又重新出现。
- 发现可能改进的地方。
- 教育(互相教育)开发人员，传授经验，让更多的成员熟悉项目各部分的代码，同时熟悉和应用领域相关的实际知识。
- 代码复审中的提问与回应能帮助团队成员互相了解，就像练武之人互相观摩点评一样。在一个新成员加入一个团队的时候，代码复审能非常有效地帮助新成员了解团队的开发策略、编程风格及工作流程。

代码复审的步骤

- 在复审前——
 - 代码必须成功地编译。
 - 程序员必须测试过代码。
- “你这样修改了之后，有没有别的功能会受影响?”
- “项目中还有别的地方需要类似的修改么?”
- “有没有留下足够的说明，让将来维护代码时不会出现问题?”
- “对于这样的修改，有没有别的成员需要告知?”
- “导致问题的根本原因是什么? 我们以后如何能自动避免这样的情况再次出现?”

缺陷检查表

- 编程规范：按照具体编程语言的编码规范进行检查，包括命名规则、程序注释、缩进排版、声明与初始化、语句格式等。
- 面向对象设计：类的设计和抽象是否合适；是否符合面向接口编程的思想；是否使用合适的设计模式
- 性能方面：
 - 在出现海量数据时，队列、表、文件在传输、上载等方面是否会出现问题，是否控制如分配的内存块大小、队列长度等
 - 对Hashtable、Vector等集合类数据结构的选择和设置是否合适
 - 有无滥用String对象的现象
 - 是否采用通用的线程池、对象池等高速缓存技术以提高性能
 - 类的接口是否定义良好，如参数类型等应避免内部转换

- I/O方面是否使用了合适的类或采用良好的方法以提高性能（如减少序列化、使用buffer类封装流等）
- 同步方法的使用是否得当，是否过度使用？
- 递归方法中的迭代次数是否合适（应保证在合理的栈空间范围内）
- 如果调用了阻塞方法，是否考虑了保证性能的措施
- 避免过度优化，对性能要求高的代码是否使用profile工具
- 资源释放处理
 - 分配的内存是否释放，尤其在错误处理路径上（如C/C++）
 - 错误发生时是否所有对象被释放，如数据库连接、Socket、文件等
 - 是否同一个对象被释放多次（如C/C++）
 - 代码是否保存准确的对象引用计数
- 程序流程：
 - 循环结束条件是否准确
 - 是否避免了死循环的产生
 - 对循环的处理是否合适，应考虑到性能方面的影响
- 线程安全
 - 代码中所有的全局变量是否是线程安全的
 - 需要被多个线程访问的对象是否线程安全，检查有无通过同步方法保护
 - 同步对象上的锁是否按相同的顺序获得和释放以避免死锁，注意错误处理代码
 - 是否存在可能的死锁或是竞争，当用到多个锁时，避免出现类似情况：线程A获得锁1，然后锁2，线程B获得锁2，然后锁1
 - 在保证线程安全的同时，注意避免过度使用同步，导致性能降低
- 数据库处理：
 - 数据库设计或SQL语句是否便于移植（注意与性能会存在冲突）
 - 数据库资源是否正常关闭和释放
 - 数据库访问模块是否正确封装，便于管理和提高性能
 - 是否采用合适的事务隔离级别
 - 是否采用存储过程以提高性能
 - 是否采用PreparedStatement以提高性能
- 通讯方面
 - Socket通讯是否存在长期阻塞问题
 - 发送接收的数据流是否采用缓冲机制
 - Socket超时处理和异常处理
 - 数据传输的流量控制问题
 - 31代码评审分析与优化

- JAVA对象处理：
 - 对象生命周期的处理，是否对象引用已失效可设置null并被回收
 - 在对象传值和传参方面有无问题，对象的clone方法使用是否过度
 - 是否大量经常地创建临时对象
 - 是否尽量使用局部对象（堆栈对象）
 - 在只需要对象引用的地方是否创建了新的对象实例
- 方法（函数）
 - 方法的参数是否都做了校验
 - 数组类结构是否做了边界校验
 - 变量在使用前是否做了初始化
 - 返回堆对象的引用，不要返回栈对象的引用
 - 方法的API是否被良好定义，即是否尽量面向接口编程，以便于维护和重构
- 异常处理：
 - 每次当方法返回时是否正确处理了异常，如最简单的处理是记录日志到日志文件中
 - 是否对数据的值和范围是否合法进行校验，包括使用断言
 - 在出错路径上是否所有的资源和内存都已经释放
 - 所有抛出的异常是否都得到正确的处理，特别是对子方法抛出的异常，在整个调用栈中必须能够被捕捉并处理
 - 当调用导致错误发生时，方法的调用者应该得到一个通知
 - 不要忘了对错误处理部分的代码进行测试，很多代码在正常情况下
 - 执行良好，而一旦出错整个系统就崩溃了？
- 安全方面：
 - 对命令行执行的代码，需要详细检查命令行参数
 - WEB类程序检查是否对访问参数进行合法性验证
 - 重要信息的保存是否选用合适的加密算法
 - 通讯时考虑是否选用安全的通讯方式
- 其他
 - 日志是否正常输出和控制
 - 配置信息如何获得，是否有硬编码

代码复审的核查表：以子程序为例

- 总体问题：
 - 如果把程序中的某些部分独立成子程序会更好？
 - 是否用了明晰清楚的动宾词组对过程进行命名？
 - 是否是用返回值的描述来命名函数？
 - 子程序的名称是否描述了它的所有功能？

- 子程序是不是只做一个事情，且是较强的功能内聚性？
- 子程序间是否松散耦合？
- 子程序的长度是否它的功能和逻辑决定？

- 参数传递问题

- 形式参数与实际参数匹配吗？
- 子程序中参数的排列合理吗？与相似子程序中的顺序一致吗？
- 接口是否进行了说明？
- 子程序中参数个数是不是7个或者更少？
- 是否只传递了结构化变量中子程序用得到的部分？
- 是否用到了每一个输入参数和输出参数？
- 是否在所有情况下它都会返回一个值？

- 防错性编程

- 是否使用了断言？
- 子程序对于非法输入数据进行防护了吗？
- 子程序是否能很好地进行程序终止？
- 子程序是否能很好地处理全局变量修改情况？
- 是否不用很麻烦地启用或去掉调试帮助？
- 是否信息隐蔽、松散耦合，以及使用输出数据检查？
- 子程序是否检查返回值？

走查(Walkthrough)

- 定义

- 由设计人员或编程人员组成一个走查小组，通过阅读一段文档或代码，并进行提问和讨论，从而发现可能存在的缺陷、遗漏和矛盾的地方。
- 类型：设计走查、代码走查。

- 走查过程

- 与评审过程类似，即先把材料先发给走查小组每个成员，让他们认真研究程序，然后再开会；
- 与评审的区别：评审通常是简单地读程序或对照错误检查表进行检查；走查则是按照所提交的测试用例，人工模仿计算机运行一遍，并记录跟踪情况。

- 走查是开发者的一次友好的会议，需要仔细规划，并有明确的目的、日程、持续时间和参与人员，许多小组以星期为单位走查。

- 走查前几天：召集人将收集的一些要在会上审查的材料(模型、文档、程序代码等)分发给参与者，参与者研究这些材料并在会议之前提交意见。
- 会议期间：召集人提出大家的意见并对每一项进行讨论。
- 会议时间比较短，一般23小时。

- 会议的目的是查明问题，而不是干扰开发者。
- 会议的思想是确认问题的存在，甚至不必去谋求问题的解决。
- 会后：将问题分发给相应人员进行解决。

使用静态代码分析发现潜在bug

- 静态代码分析(staticcodeanalysis)：在代码构建过程中帮助开发人员快速、有效的定位代码缺陷；
 - 无需运行被测代码，仅通过分析或检查源程序的语法、结构、过程、接口等来检查程序的正确性，找出代码隐藏的 errors 和缺陷，如参数不匹配，有歧义的嵌套语句，错误的递归，非法计算，可能出现的空指针引用等等。
- 30~70%的代码逻辑设计和编码缺陷是可以静态代码分析来发现和修复的”
- 主要技术：缺陷模式匹配、类型推断、模型检查、数据流分析等。
- 常用的Java静态代码分析工具：
 - Checkstyle
 - FindBugs
 - PMD

代码静态分析工具

- Checkstyle：通过对代码编码格式、命名约定、Javadoc、类设计等方面进行代码规范和风格的检查。
- FindBugs：通过检查类文件或JAR文件，将字节码与一组缺陷模式进行对比从而发现代码缺陷，完成静态代码分析。
- PMD：通过其内置的编码规则对Java代码进行静态检查，主要包括对潜在的Bug、未使用的代码、重复的代码、循环体创建新对象等问题的检验。
- Jtest：能够按照其内置的超过800条的Java编码规范自动检查并纠正这些隐蔽且难以修复的编码错误，同时还支持用户自定义编码规则，帮助用户预防一些特殊用法的错误。
- 微软VisualStudio中的代码分析工具可以检查代码中是否存在一些常见缺陷和违反良好编程习惯的情况。
- Lint是一种静态程序分析工具，目前已形成了一系列工具。它侧重于代码的逻辑分析，发现代码中一些潜在的错误，如数组访问越界、内存泄漏、使用未初始化变量等
- JSHint (www.jshint.com) 是一款JavaScript代码验证工具，用于检测代码中的错误和潜在问题。
- CSSLint (csslint.net) 是一款CSS代码检查工具，可以进行基本语法检查以及使用一套预设的规则来检查代码中的问题。
- HTMLHint (htmlhint.com) 是一款基于JS开发的静态扫描组件，支持所有浏览器和Nodejs平台，可集成到IDE环境或编译系统中。

4 性能分析：动态程序分析

程序性能分析

- 性能分析(performance analysis, 也称为profiling): 以收集程序运行时信息为手段研究程序行为的分析方法, 是一种动态程序分析的方法
- 这种方法与静态代码分析相对, 主要测量程序的空间或时间复杂度、特定指令的使用情形、函数调用的频率及运行时间等。
- 性能分析的目的在于决定程序的哪个部分应该被优化, 从而提高程序的速度或者内存使用效率。
- 性能分析可以由程序的源代码或是可执行文件进行, 一般会使用称为性能分析工具(profiler)的工具进行。

程序性能分析的两种方法

- 抽样(Sampling)
 - 当程序运行时, 分析工具时不时看一看这个程序运行在哪一个函数内, 并记录下来, 程序结束后, 工具就会得出一个关于程序运行时间分布的大致的印象。这种方法的优点是不需要改动程序, 运行较快, 可以很快地找到瓶颈。但是不能得出精确的数据, 代码中的调用关系(CallTree)也不能准确表示。
- 代码注入(Instrumentation)
 - 将检测的代码加入到每一个函数中, 这样程序的一举一动都被记录在案, 程序的各个效能数据都可以被精准地测量。这一方法的缺点是程序的运行时间会大大加长, 还会产生很大的数据文件, 数据分析的时间也相应增加。同时, 注入的代码也影响了程序真实的运行情况。
- 一般的做法是, 先用抽样的方法找到效能瓶颈所在, 然后对特定的模块用代码注入的方法进行详细分析。

关于性能分析

- 性能测试->分析->改进->性能测试
- 先Debug, 再性能分析
- 先保证正确性, 再提高效能
- 是否会用性能分析工具来提高程序质量是一个优秀程序员的标志之一

性能分析的工具

- 不同的编程语言有不同的性能分析工具;
- http://en.wikipedia.org/wiki/List_of_performance_analysis_tools进行了一个很好的汇集;
- 常用:
 - Google提供的gperftools, 面向C/C++等
 - VisualStudioTeamSystemProfiler
 - TestandPerformanceToolsPlatform(TPTP), Eclipse/Java

- cProfile,profile和pstats, Python
- 以gperftools(C/C++)为例
 - <http://code.google.com/p/gperftools>
 - Google提供的一套工具，用于分析程序性能，找到程序的性能瓶颈。
 - mallocfree内存分配器tcmalloc;
 - 堆内存检测和内存泄漏分析工具heapprofiler和heapchecker;
 - CPU性能监测工具CPUprofiler。
- 以TPTP(JAVA)为例
 - <http://www.eclipse.org/tptp>
 - EclipseTest&PerformanceToolsPlatform, 作为Eclipse的插件。
 - 重点关注其中的Tracing&ProfilingTools。
 - 内存分析(BasicMemoryAnalysis);
 - 执行时间分析(ExecutionTimeAnalysis);
 - 代码覆盖(MethodCodeCoverage);