**代码可维护性分析报告**

**团队： G17小组**

**项目： 图书交流网站**

**组长： 陈传岭**

**组员： 陈杰，周泽鑫**

目录

[1. 前言 4](#_Toc10992099)

[2. 软件维护性概念 4](#_Toc10992100)

[3. 过程语言的可维护指数 4](#_Toc10992101)

[3.1 Halstead 4](#_Toc10992102)

[3.2 Cyclomatic Complexity 5](#_Toc10992103)

[3.3 Lines of code 5](#_Toc10992104)

[4. 具体代码分析 5](#_Toc10992105)

[4.1 一般流程 9](#_Toc10992106)

[4.2 样式管理 10](#_Toc10992107)

[4.3 小结 12](#_Toc10992108)

[5. 面向对象的可维护指标 12](#_Toc10992109)

[5.1 WMC （weighted methods per class) 12](#_Toc10992110)

[5.2 DIT ( Depth of Inheritance Tree) 13](#_Toc10992111)

[5.3 NOC 13](#_Toc10992112)

[5.4 CBO 13](#_Toc10992113)

[5.5 RFC 13](#_Toc10992114)

[5.6 LCOM 13](#_Toc10992115)

[6. 软件的可分析性 14](#_Toc10992116)

[7. 代码的可读性 14](#_Toc10992117)

[7.1 表达式的格式化要点 14](#_Toc10992118)

[7.2 表达式格式化要点 14](#_Toc10992119)

[7.3 命名规范要点 14](#_Toc10992120)

[7.4 注释方法 14](#_Toc10992121)

[7.5 对齐要点 14](#_Toc10992122)

[8. 代码的可理解性 15](#_Toc10992123)

[8.1 表达式、函数、方法不能过大 15](#_Toc10992124)

[8.2 关于控制语句的书写 15](#_Toc10992125)

[8.3 函数需要写的易于识别 15](#_Toc10992126)

[8.4 降低程序之间的耦合性 15](#_Toc10992127)

[**8.4.1** 数据耦合 15](#_Toc10992128)

[**8.4.2** 数据结构耦合 15](#_Toc10992129)

[**8.4.3** 控制耦合 16](#_Toc10992130)

[**8.4.4** 临时耦合 16](#_Toc10992131)

[**8.4.5** 公共耦合 16](#_Toc10992132)

[**8.4.6** 外部耦合 16](#_Toc10992133)

[**8.4.7** 内容耦合 16](#_Toc10992134)

[**8.4.8** 注释 16](#_Toc10992135)

[9. 代码的可追溯性 16](#_Toc10992136)

[10. 软件的可改变性 17](#_Toc10992137)

[10.1 识别性 17](#_Toc10992138)

[**10.1.1** 直观的命名 17](#_Toc10992139)

[**10.1.2** 良好的注释 17](#_Toc10992140)

[**10.1.3** 慎用多态 17](#_Toc10992141)

[10.2 可分离性 17](#_Toc10992142)

[11. 软件的稳定性 18](#_Toc10992143)

[11.1 封装和数据隐藏 18](#_Toc10992144)

[11.2 数据抽象 19](#_Toc10992145)

[11.3 类型检查 19](#_Toc10992146)

[12. 小结 19](#_Toc10992147)

# 前言

很多包括自己在内的开发人员都会经常去借用开源代码进行二次开发；或者在前辈的遗留代码下，继续修修补补。这种经历往往并不像看起来那么简单——有时看懂，进而修改别人的少许代码，都会觉得无从下手，究其原因主要是代码晦涩，关系复杂，难以隔离影响等。

而这时我们或者抱怨前人代码写的不好，又或者会自惭自己编码水平太次。其实这种困境的起源除了自己原因以外，更多是因为代码的可维护性不够。

# 软件维护性概念

所谓软件的可维护性其实说简单了就是软件代码的可被修改的容易程度。如前言所说，代码反复修改的情况不可避免，这种软件的不断演化过程——具体就是修正错误；适应新环境；满足新需求——虽然貌似将软件的功能变的越发强大，但是事实上这些改变总是或多或少的有悖于当初的设计初衷，因此势必慢慢的蚕食软件的基础架构和代码质量——造成的结果是让代码越来越难看懂，健壮性越来越脆弱，修改一个bug的代价越来越大。

鉴于这个矛盾，Martin Fowler提出的(refactor)代码重构主要就是从代码编写角度出发，提高代码的维护性，以便能更好适应软件演化。那么接下来的一个问题是：软件的可维护性有无标准的评测方法？学院派早都就此问题给出四个定义——：可分析性；可改变性；稳定性；易测性 。此刻先别去追究这个几个形而上学的术语 —— 后面我会就各点进一步展开，谈谈自己的看法。再次之前，我们先来看看定量评价可维护性的方法 (其实本文重点，不在于此，你完全可跳过以下两节)。

# 过程语言的可维护指数

首先来来谈谈面向过程语言的可维护性计算：这里有一个更貌似深奥的可维护性指数：

Maintainability Index（MI) = MAX(0,(171 - 5.2 \* ln(Halstead Volume) - 0.23 \* (Cyclomatic Complexity) - 16.2 \* ln(Lines of Code))\*100 / 171)

它看似一个对维护性定量分析的精确的数学公式。其实不然。这个公式无论是系数或者是运算项都是来自于经验规则（你千万别想着去推导它）

## Halstead

是测量代码计算复杂度, 具体如果一个程序有N个操作数和运算符，n个不同的的操作数和运算符，那么halstead ＝ N ＊ Log2(n) ; 总之程序中的运算符和操作数越少越利于提高MI。

## Cyclomatic Complexity

是代码的逻辑复杂度，程序的每个可能的执行分支（if, while，for 等）都为该指标贡献1个点。 该值的建议范围<10 ,最多不超过20。

## Lines of code

是代码行数。

注意，需要说明的是每个模块，每个文件都可计算维护性指数，甚至每个函数都可计算，而程序的MI则是其所有的平均值。

如果你想测试一下你代码的MI，这样的工具开源的倒是有一些（http://www.chris-lott.org/resources/cmetrics/）。这里给大家推荐一个在Linux下的工具：pmccabe（apt-get install pmccabe即可安装)可用于计算Cyclomatic Complexity，line of code等 。可惜目前计算halstead的开源工具我还没有发现，不过好像vs 2008的新特性里支持代码复杂度计算，其中的各种指标还比较详细，建议大家可试用。

这个来自经验的MI的值越高越好，说明维护性越强。但是毕竟是经验规则，就如同我们有很多经验说如何买股票能挣钱，或者如何打仗能打胜，但是却绝对没有谁说一定这么作就能挣钱或者打胜仗。所以MI更合理的使用场合的是作为代码重构时的参考，便于发现那部分代码可能需要重点考虑。而且最好的评价方式是进行纵向比较，也就是说将你重构前的代码MI和重构后的比较，从而判断重构的积极意义。

# 具体代码分析

例：首页部分代码：

<?php

include 'head.php';

include 'connect.php';

?>

<aside class="jcsp">

<h3>图书资讯</h3>

<ul>

<ul>

<?php

$query1 = mysql\_query("select \* from product where lanmu like '图书资讯' limit 0,12 ");

while($row = mysql\_fetch\_array($query1,MYSQL\_ASSOC)){

echo '<li><a href=more.php?id='.$row['id'].' class="title">'.$row['title'].'</a></li>';

}?>

</ul>

</ul>

</aside>

<article class="gsjs1">

<h3>关于我们</h3>

<dl>

<dt> <a href="#"><img src="images/about.png"></a></dt>

<dd>

大学生在校园中有着太多诱惑，网游、电影、手游…这些诱惑充斥着我们的大学生活。高中期间你也许还会捧着一本书慢慢的看，但是在现在的大学生活，阅读渐渐消失在人们的视野中。大部分人没有时间，而有时间的人又不知道读什么书好。再或者当你读完一本好书后，急切的想要分享，但周围的人却没有兴趣，冷淡的对待。志同道合的朋友才长久，爱玩游戏的朋友好找，爱看书的朋友可不好找。

</a></dd>

</dl>

</article>

<aside class="jctj">

<h3></h3>

<section id="demo" >

<article id="indemo">

<article id="demo1">

<?php

$query1 = mysql\_query("select \* from product where lanmu like '最新图书' ");

while($row = mysql\_fetch\_array($query1,MYSQL\_ASSOC)){

echo '

<dl>

<dt> <img src=upload/'.$row['pic'].' ></dt>

<dd> <a href=more.php?id='.$row['id'].' class="title">'.$row['title'].' </a></dd>

</dl>

';

}

?>

</article>

<article id="demo2"></article>

</article>

</section>

<script>

<!--

var speed=20;

var tab=document.getElementById("demo");

var tab1=document.getElementById("demo1");

var tab2=document.getElementById("demo2");

tab2.innerHTML=tab1.innerHTML;

function Marquee(){

if(tab2.offsetWidth-tab.scrollLeft<=0)

tab.scrollLeft-=tab1.offsetWidth

else{

tab.scrollLeft++;

}

}

var MyMar=setInterval(Marquee,speed);

tab.onmouseover=function() {clearInterval(MyMar)};

tab.onmouseout=function() {MyMar=setInterval(Marquee,speed)};

-->

</script>

</aside>

<?php

include 'foot.php';

?>



上述代码为红框内容

上面图片中包含：top.php，left.php，index.php



Top.php



Left.php

## 一般流程

采用“拼接”的方法，主页加载top页面，链接数据库内容，提取数据库数据，显示left和index页面的信息。

这样的方法更容易维护，并且容易寻找错误的出处：

例如：

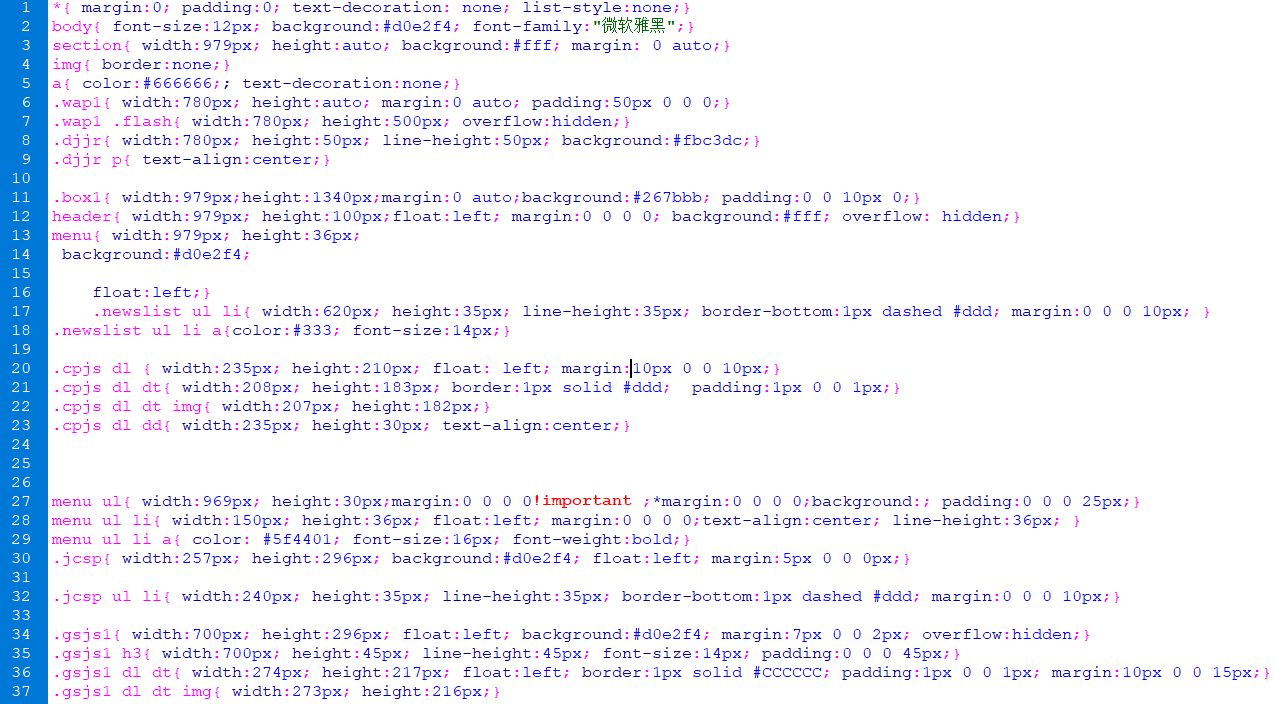


在此页面下，很容易得出网页的错误是由数据库链接问题产生的。

## 样式管理

例（css.css）

用于定义和管理网页的样式



同样在用户登录界面：

拼接了top，left和admin\_login页面的内容



以及单独创建的connect页面，用于网页和数据库的连接：



用于每个页面的基本调用。



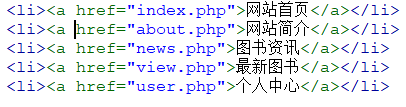
Admin管理页面

同样，管理员页面同样采用分割的方式。

## 小结

网页本身的易维护性相比与建立在pc和移动端的应用，网页具有明确分割的特点，样式和内容的分离，框架与结构的分离，为程序员提供了良好的环境。

例如，当主页需要增加新的一栏项目时，只需要修改head中的下图这一列代码：



增加需要的新栏目，并且创建与之相连的php页面，在新的页面中同样加载head，connect等页面，可以提高维护的速度，网页结构的完整性也不会被破坏。

# 面向对象的可维护指标

对于面向对象语言而言，可维护性的指标则更发散，更复杂一些。毕竟面向对象代码的基础单元是对象，而对象的封装，继承，多态这些特性无不影响代码的可维护性。下面介绍几个面向对象语言使用的通用指标。

## WMC （weighted methods per class)

基于类的加权方法，说直接点就是给定类中的方法数目，方法越多则该类WMC越大——其实对于过程语言这点也可借用，类相当于过程语言中的文件（我们一般都称其是模块），所以说文件中的函数越多，则维护性越难—— 我曾经参与过一个操作系统开发项目，其编程规范要求每一个函数一个文件。虽然有些极端，但也不失为一种维护策略。

## DIT ( Depth of Inheritance Tree)

继承树深度度量，继承是面向对象语言的优点之一，可提高代码重用性。可是层层继承也势必带来风险——牵一发而动全身，父类的改动对子类影不可避免，因此过深的继承关系对代码维护性会产生不良影响。

## NOC

(Number of children)子女数，一个类的子女数应该适当，而不能太多。太多时应再重新划分子类——这里要强调一点接口或者抽象类是我们推荐的方式；而采用继承则需要扩充父类方法，也就势必增加了模块的耦合性，削弱了灵活性。

## CBO

(Coupling Between Object Classes)类之间的耦合系数。我们都知道避免耦合性的开发原则——耦合会降低代码重用，增加单元测试难度和代码维护性。耦合系数其实就是给定类依赖于其他类的数目——所谓代码依赖有方法调用中，属性访问，继承，参数，返回值，以及异常等。CBO我认为是一个很重要的度量，一定要尽可能降低耦合性。

## RFC

（Response for a class) 类响应度量，指当类的对象接受到一个消息时，执行的方法总数。显然该值越大，则越难理解，调式和测试。

## LCOM

(lack of cohesion in Methods) 欠内聚量度，统计完全没有共享属性的方法个数。简单的计算方法就是用没有共享属性的方法个数，减去所有有共享属性方法个数，结果为负数则按0计算。 内聚值越高说明方法之间的关联性不强，因此可以继续差分子类，如果LCOM很低则说明类封装的很内聚——这点可以说是强内聚的标准定义。

关于如何使用这些度量没有标准法则， NASA有一个建议值：WMC>100,CBO>5,RFC>100,RFC>5\*NOM ,NOM>40 (NOM是类中方法个数），一旦有2个以上指标达到上述阀值，则需引起重视。

最后，就耦合问题——也就是依赖问题，我们重点说一下： 无论是过程语言程序，或者是面向对象程序。模块之间（类，或者是文件）或者包之间（对过程语言没有包的概念，但我们更喜欢将某个功能集合放在一个文件架下，如linux内核各种子目录，可以将其类比成包）的关系具有两种关系：依赖别人，或者被别人依赖 。比如工具包（math,pthread,rt，std容器等）一般都不依赖别人，而且设计抽象（后面我们会进一步解释抽象），这些包属于稳定包，不能经常改动，因为细微的改动则会影响依赖其的所有组件。而处于接近用户的模块或者包则多依赖于稳定包，它们不被其他模块或者包依赖，这些包可以被经常被改动，而影响不会扩散。

另外还需要特别注意，尽量不要循环依赖或者相互依赖，即你依赖我，我依赖你。这种相互依赖，危害很大，任何一个包改变都会引起连锁反应。因此依赖应该是单向的——可以参见层次设计，如VFS等。 另外从模块接口设计角度讲，引用外部方法时，可借助回调算子（callback函数）模式,将外部方法作为变量传入本函数作用域，再以使用。这样做降低了模块代码间的赖性，有利于阅读，修改和调试。

我们依照学院派的思路划分——在逻辑上有其可取性——来依次展开讲述软件维护性的几个评价维度。

# 软件的可分析性

首先是软件的可分析性——它包括可读性，可理解性和可追溯性，这是软件开发首先应该遵从的要求，该要求不高，只是需要养成好习惯。

可读性这里指的比较狭义，它强调的是编码风格：如格式，命名，对齐，注释等。 而可理解性和可追溯性则是在软件设计层面的要求。 可理解性强调代码编写的应遵循的约定俗成的模式，莫要将代码写的太个性化，太特立独行；可追溯性强调代码各部分依赖性的情况，依赖越少，隔离性越强，则越容易追溯。

# 代码的可读性

先来谈谈可读性，格式问题其实就是一些“美学上”的约定罢了。总结如下：

## 表达式的格式化要点

二元操作符号都应以空格和前后相隔，不需要空格的是括号、标识符和一元操作符。比如 n = （time.tv\_sec – diff) % (3600 \* 24) 。另外使用括号划分功能和限制运算顺序仍是正道——不但容易让人读懂，也防止了运算符优先级混淆。

## 表达式格式化要点

控制流关键字（if,while）等,和后面判断表达式单元以空格隔开

## 命名规范要点

自己不要发明命名规范了——C程序员就去参考Linux内核代码的命名规范，或者GNU的规范，java的遵循java规范。

## 注释方法

格式要求去参考Doxygen的要求；注释内容等见下一节

## 对齐要点

代码的对其（包括换行，缩进）对于代码可读性至关重要，相关说明很多，这里不再赘述。给大家一个辅助工具帮助调整代码对齐——indent命令来对原代码做优化。如indent -npro -kr -i8 -ts8 -sob -l80 -ss –ncs（参数说明：-npro或--ignore-profile 　不要读取indent的配置文件.indent.pro；-kr 　指定使用Kernighan&Ritchie的格式；-i8 　--indent-level 设置缩排的格数为8；-ts8 设置tab的长度；-sob或--swallow-optional-blank-lines删除多余的空白行；-l80 代码超过80换行；-ss或--space-special-semicolon若for区段只有一行时，在分号前加上空格；-ncs或--no-space-after-casts 　不要在cast之后空一格）

# 代码的可理解性

代码的可理解性顾名思义，是说代码设计中一个重要思想——简单就是美！ 不要复化你的代码。为此你需要注意如下点：

## 表达式、函数、方法不能过大

易读的代码函数一般都在10-20行左右，不要将多个事情放在一个函数里完成，时刻要注意一个函数只完成一件事情，这种设计无论是从代码重用性（越是功能单一的代码越容易重用），或者可读性上都有很大的好处，对于单元测试也大有益处。当然操作硬件初始化的函数有可能很大，或者复杂的逻辑函数也可能较大，因此不强求要将函数代码变短，但是写成小函数是一个基本的编程取向。如，Linux内核代码建议：函数最大不要超过300行——超过了认为几乎不可读，内部局部变量不要超过10 ——过多也造成函数不可读。

## 关于控制语句的书写

控制语句中的状态判断（if XX,while XX），和相应的处理语句都应该力图简单明了。如果状态判断表达式样很复杂，则应该单独抽象成函数或宏，以便理解；同样要是处理逻辑很复杂，则同理应该抽象成函数完成。

## 函数需要写的易于识别

写函数应该按照约定俗成的方式来写，比如链表的写法，下一个个元素使用next指针，使用for循环遍历等都是大家约定的方式 。这种方式应该遵从，无论是变量命名也好，或者是控制就结构也好，莫要特立独行，让人误解。

## 降低程序之间的耦合性

代码耦合可进一步细分下面种类：

### 数据耦合

一个函数将数据传递给另一个函数接续处理，这种耦合对代码理解性无伤大雅。

### 数据结构耦合

将一个结构大的数据结构传递给一个函数，而这个函数 只需要该结构中的一部分。比如，有些程序为了避免使用全局变量，因此所有的数据都通过参数传递，就会有这种情况——由于数据结构耦合允许被调用程序读取或者修改不属于其操作的数据，因此这种行为是危险，应当避免的。

### 控制耦合

一个函数传递给另外一个函数的参数会影响被调用函数的执行控制流程，其主要问题在于输入参数不同，执行代码代码就不同，因此代码很难理解。 解决方法是将函数进一步拆分。

### 临时耦合

和函数调用顺序相关的逻辑设计，应该遵循约定的调用方式和命名方式，如采用 1 .construct /open /acquire 2. use 3. destruct .finalize cloes ,dispose 等。

### 公共耦合

两个函数使用了同一个全局变量。全局变量的使用是可理解性的大敌，因为对于程序中的全局变量，我们都需要遍历所有代码才能确定该如何操作该变量。如果全局变量多了，且在代码中出现处频繁，则将是一个难以容忍的任务 。所以尽量减少全局变量，至少让全局变量不要出自己模块——使用staitic等方式将其隐藏自己模块中。

### 外部耦合

两个模块隐式的共享设备接口，通讯协议，或者数据结构。这种情况在协同开发中很是常见，大家必须事先约定好之间的信息接口，才可进行各自独立开发。这个沟通过程可往往是最耗成本的环节，我们尽量在设计中减少这种耦合吧。

### 内容耦合

一个模块修改了另一个模块的内部数据，或者一个模块依赖于另一个模块的内部数据。这种耦合最严重，可以说这种设计是失败的，即便你的代码全部是你自己用，也是搬起石头砸自己脚。

### 注释

代码注释对可理解性很重要，一个有用的注释应该传递准确信息，比如函数定义，应该指明该函数作用、入口参数、返回值、前提条件、后续结果。另外对于流程式的程序，应该有里程碑式的标记注释，以表明程序的结构和功能，这种注释多出现在程序初始化流程中；另外对于数据声明的注释则应该紧邻数据生命，以方便定位、查询。

另外有几个特殊注释，需要提醒大家：

TODO: 表示需要实现，但目前还未实现的功能

XXX: 勉强可以工作，但是性能差等原因

FIXME:代码是错误的，不能工作，需要修复

# 代码的可追溯性

首先是位置问题，也就是说变量的使用和其定义的位置应该尽量靠近，不要都放到函数入口后定义变量——虽然这样看似很整洁。因为我们看到一个变量被用时，自然会去寻找其定义，因此最好靠近放置。（我们的搜索顺序必然是代码块内，同意方法或者函数里，同一类或者文件中，同一个包或者目录里的其他文件，其他项目中）

另外要尽量减少代码模糊性。比如前面说的内容耦合，任何一行代码改动都可能引起其他代码的错误。为此应当尽量采用私有属性等方式，防止不必要的暴露。还需要注意多态机制其实也是造成模糊的一个方面，因为多态让我们很难知道到底在使用那个方法。

# 软件的可改变性

软件的可改变性是说对软件作出修改的容易程度。主要从两点分析：

1 找到修改点的难度；

2 修改是否会对软件的其他部分造成影响。

总之一个是识别性，一个是软件的隔离性问题。

## 识别性

就识别方式而言，我们要摸是自上而下认识代码，或者是自下而上认识代码。自上而下的方法，需要我们对整个系统架构足够熟悉，才能找到需要修改的子系统，然后继续寻找到具体的修改位置。这种方式对于复系统代价很大；相反自下而上的方式则需要直接深入到具体代码片中，当然这种方式需要依靠启发思维和直觉来帮忙（想想，如果一个新人来到公司，让其去解决系统bug——假设这个系统维护性很差——首先定位问题这个环节，就可能让其在初期陷入手忙脚乱，狼狈不堪的境地）。

提高软件的可识别性的方法有：

### 直观的命名

直观的名称有助我们识别他们。比如Linux的系统调用都以sys\_前缀，那么修改莫个系统调用则直接可通过搜索函数定位到。

### 良好的注释

注释中有时需要记载引用的规范（尤其硬件初始化的注释，往往需要知名手册的页码，行数等），算法（重要算法需要用伪代码注释），那么通过该搜索注释，可帮助找到对应修改点。

### 慎用多态

多态的优点是提高代码重用性，但同时带来的缺点是很难定位负责给定功能的代码片。因此不能滥用抽象，比如只有一个子类的类或者模板相关性很强，只能针对少数类型时，则慎用抽象。

## 可分离性

下来说说分离性问题。

代码分离性是说基于需求变更，代码修改的扩展性。对于分离性的保证需要前瞻性设计——将此后的变更尽量集中在系统的不稳定单元——即那些需要依靠其他单元的单元，而自身又很少被其他单元依靠。具体的方法有：

提高抽象设计、增强内聚、减少重复代码和硬编码。其中最重要的方法就是——高度抽象+强内聚。

先说抽象，抽象最常用的是接口抽象，比如VFS系统和实际存储引擎之间的结构抽象，则很好的隔离了具体数据持续化过程，和上层的控制逻辑。在抽象接口不变的情况下，存储引擎的开发，维护都是独立的，并不影响VFS的工作逻辑；而对于增加注入用户访问限制等逻辑，修改VFS层的控制流即可。这两者互相不干扰，各自隔离。——这种接口抽象对于framework程序、库程序等尤其重要，一旦接口定义好了，变化部分就处于实现了，而接口本身休要轻易改变。

再说内聚，所谓内聚就是减少单元之间的耦合。原则是“只和你最亲密的朋友交谈”，不要试图去通过一个对象，访问另一个对象的方法，那样就以意味着你和你朋友交谈了。另外可借设计抹额是减少程序耦合性，比builder、factorey method、chain of responsibility、interator 、mediator、memento、strategy、template、method、visitor等模式；对于内聚设计还有一个原则上文已经提及，就是一个方法只有一个逻辑，不要再里面做两件事。

避免重复代码，就是尽量不要在程序中重复实现同样逻辑的代码片；避免使用硬编码，就是使用宏等代替硬编码。这两点都容易理解，这里不再赘述。

# 软件的稳定性

软件的稳定性是说在我们的代码演化过程中，修改局部代码所引起的连锁反应不应引起过多的不良后果。最主要的手段有封装数据结构和数据隐藏、分离组件和服务、以及数据抽象和进行类型检查。

## 封装和数据隐藏

数据封装的目的在于让数据或者方法的作用范围尽可能收敛，具体做法是：

1.变量声明进限于其作用域中（在C总我梦经常在某个条件下的{}内声明和使用变量）。

2.声明类成员并给予最小的可见性（类方法的访问控制符号可帮助我们，权限越小，使用的范围越窄，对他的修改可能引起的问题则越少，因此优先选择priviate）。

3.相关类尽量封装到一个模块内。java使用package,C++使用namespace。C没有专门的概念，但可以看成文件，那么模块内并非所有的类，方法都需要和外界交互，只需要给予那些提供外部接口的内一外部可见的权限，而将其他封装起来，减少外部可见接口，提高稳定性。

4.使用组件和分离进程：

在开发复杂系统时，尽可能的将组件或者服务单独实现，以便隔离相对独立自系统。这点很是重要，化整为零的策略不但似的系统功能解耦合，而且整个系统越发容易开发，调试，测试，因而也越发稳定。比如Berkelery DB其锁机制等都是独立的子系统，完全独立于整个系统；另外Linux的打印服务程序也被分开成几个独立的自任务（进程）——有提供用户界面的程序lpr,有提交打印神情的lp,有查看打印队列的lpr程序，这种隔离必然提高了整个服务程序的稳定性，因为减少了突发或者恶意交互，提高了系统容错性。错误的定位和隔离都大大优于单模块程序。

## 数据抽象

数据抽象和前述的封装相得益彰。数据抽象的本质就是提取数据结构的本质特征，创建出个数据类型，并创建接口方法来操作这些类型，向外提供给定功能，而外界不用关心这些 数据类型的具体组成。

我们常常在项目中看到的utils 或者common目录下的很多工具类（如容器等）都属于这种抽象。比如glib库中抽象出的指针数组，或者链表，hash等容器则是典型的数据抽象——它已经被抽象成可以容纳任何类型的对象的数据组织结构了；将抽象再提升一个层次，如果我们的项目需要实现自己的一个map容器，则可以提供一个统一的map容器方法（添加，删除，查询）等，而内部的实现则或者可用glib的容器，也可采用boost的容器。由于高度抽象了map接口，最后的使用者则不需要关心具体实现，就好比你使用的虚拟机在linux宿主上，或者在windows宿主上跑，你根本不用关心。

## 类型检查

稳定性最后一点谈类型检查，程序的类型检查的任务交给了编译器，我们不要绕开它。这种编译期检查往往是发现我们错误的好帮手，莫要绕过它。不过，粗暴的程序员常在碰到类型不匹配警告时，采用强制转换来消除警告——而强制转换中，由通用数据类型（void\*）转换成特定类型时,需要谨慎，一定要确保数据类型的正确性，否则很难发现，即使到了运行期，也很难跟踪。

另一种检查是编译期间的断言：主要用于核实编译环境，比如检查，编译器，操作系统

#ifdef MACH

[]

#elif defined(\_\_NetBSD\_\_)

[]

else

#error OS unsupported

#endif

或者监察程序配置是否正确

#if(FAST\_NFS\_PING \*MAX\_ALLOWED\_PING)>=ALLOWED\_MOUNT\_TIME

#error : sannity check failed in ...

最后一部检查是运行期间断言，这种应该出现在调试版本中，在运行期对程序条件作出判断，弥补编译期断言的不足。（运行期需要关闭NDEBUG标识的情况下才有效,因此在调试期可方便使用）。

除了上述各点外，对于软件维护性而言还有易测性等要求。不过由于大家基本已经对软件的是的重要性和基本方法早已取得了共识，切资料丰富、实践充分，因此我在这里就不谈它了。

# 小结

总之，当前软件规模化开发以后，编写代码时刻要提醒自己，自己写的代码是要给别人读，要被别人修改的，而他人对代码必然没有你自己熟悉，因此要将代码写的易于读懂，便于修改。