**程序开发基本原则**

广州市仙海网络科技有限公司

程序部

## 做人，做事！

一分耕耘一分收获。对待事情的态度，决定了你将从中获得怎样的回报！做好人，才能做成事！

## 执行力

在其位，谋其职。程序首要责任是满足设计人员的需求，将设计人员的设计思想变为现实。

如果公司董事会决定开展一个项目，自董事长开始向下到总经理、各部门负责人、具体事物执行人的各个环节中，如果任何一层的执行力出现问题，将最终影响这件事情能否成功！所以在任何岗位，不容存在执行力差的员工！

在具体工作中，应该严格履行并实现设计文档所描述的的需求。如果对文档需求或实现方式存在疑问或异意，则及时与设计人沟通，寻求讲解或重新确定实现方式。在没有与设计人员沟通并经过设计人同意的情况下，不能违背、偏离设计文档的制作意图进行开发！对设计、需求表案的实现，必须完全完成所有的设计功能而不能只实现其中的某些部分。所有交办的工作，必须在规定的时间内完成，如果对工期有疑义，应该尽快与工作下达者沟通以便明确执行时间。在工作执行期间，如果遇到重要的客观原因而导致工作可能会延期时，必须提前与工作下达者沟通，以便让其有所准备或再进行其他安排。

技术总监在此郑重提醒：如果出现上述问题，则表明这个程序员的执行力有问题，一经发现必予以警告和开导，两次以上则予以辞退，我们需要的是认可公司价值，具有严格执行力的人！

## 优先注重的

考虑一个需求如何实现的时候，大到一个软件的架构、系统模块，小到一个函数，一个分支语句。在进行服务端以及客户端开发中，我们思考的重点优先级依次是：**稳定、性能、存储开销、传输开销、维护性**。

### 稳定

软件的稳定性决定软件一切价值的根本。

### 性能

软件的性能（承载性、效率）决定软件的最大价值容量。例如一个只能上500人的低效的游戏服务端与一个能上5000人的游戏服务端比，哪个容许的价值空间更大？同是要求高交互性的游戏，一个一屏内显示100个角色就卡得不动的客户端与一个一屏内可以显示500人的客户端比，哪个容许的价值空间大？

### 存储

存储开销包括内存存储开销以及硬盘存储开销，这方面直接决定产品的运营成本。一个需要1G内存能跑5000人的游戏的服务器程序与需要4G内存才能跑5000人的程序比，那个更节省成本？服务器的成本也是影响运营商选择游戏产品的重要因素之一。

在存储开销的优化方面客户端的要求要比服务器更苛刻，因为服务器的成本运营商可能支付得起，而客户端如果太占内存，玩家则不一定愿意为游戏而提升电脑配置，进而导致因技术原因而丧失用户！

### 传输

传输开销指在CS/BS等通信环境下客户端与服务端之间的网络通信数据包所形成的网络流量的开销，不包含通信的时间和性能开销。这部分与存储开销相类似，在游戏期间众玩家与服务器之间通信产生的流量与运营商将为此支付的费用成正比。

同样在客户端中，对传输开销的要求更高，因为大多数最终用户的带宽都是有限的，在与游戏服务器通信的同时，还要从资源服务器下载包括地图、角色、怪物、技能等游戏资源。这些资源的下载都将在服务器以及客户端双方产生通信流量，若最终用户带宽太小而造成无法及时与服务器通信或加载资源（“卡”），所导致的结果也是丢失用户！

### 持续

在采用了一些列行之有效的方法处理以上问题之后，我们必须重审代码的维护性，包括逻辑清晰性、注释完整和有效性以及可复用性。检查并设法消除代码中潜在的陷阱，以便使日后修改这份代码不容易出错。提高代码的可维护性相当于进行逻辑整理或代码重构。

### 易用

当进行开发工具与管理工具等工具软件开发的时候，我们思考的重点优先级与服务端和客户端略有差别，依次是：**稳定、易用性、效率、维护性、存储开销、传输开销**。这里比服务端和客户端的要求中多出一个“易用性”是因为实现开发工具、GM管理工具以及其他各类工具的根本初衷在于——通过工具提供易于上手、操作方便、以及节约时间和管理成本的工作方式，因此易用性就显得尤为重要。

服务端程序以及客户端程序也同样需要考虑易用性，这主要着眼于软件的部署、更新和维护工作如何更容易、更方便的开展和进行。你需要考虑一下当运维人员面对几十台乃至数百台服务器时，你需要如何改善软件的操作或部署规则从而提高更新以及部署的效率。

你是否记住了你参与的开发中，应该优先注重那些要点？

## 常规开发注意事项

### 可维护性

在设计和编码任何模块、类的同时，请务必考虑这个功能如何在以后的代码维护工作中能够更容易的被修改，当有新人来阅读你现在正在写的这份代码时，如何能让他更容易理解和上手。

可维护性包括可读性、逻辑清晰性和容错性方面。**如果对一份代码的语法一致并良好，有着完善清晰的注释和详细的实现说明文档，那么我们认为这份代码具有很好的可读性；对代码各个细节的修改不容易产生新问题或隐患，或者有问题或隐患产生时能够有明显的错误或警告呈现来，那么我们认为这份代码具有很高的容错性；设计或代码中对一个细节规则的实现足够的集中，以致仅通过一两个函数就能完全实现一个规则的细节，那么其具有较高的逻辑清晰性。**具有高可维护性的代码，不仅易于阅读、理解和修改，也能够降低修改投入的成本和带来的风险。

本文中后续章节将在不同的分类中提及注重可维护性的细节。

### 可复用性

在设计和编码任何模块、类的同时，请务必考虑这个功能如何在以后的产品中被使用。如何不加修改直接使用或以更小的修改代价来使用。

在前言中我们提到了经验传承在人类文明发展中的重要性。同样，在软件开发中，一个模块的代码的可复用性（直接在其他项目中使用）就就是这份代码能否传承的重要表现。如果为每个产品的相同功能都进行一次完整的独立的开发，那么之前的产品中统一功能的代码就丧失了复用性！

从设计、编码的过程中降低模块以及代码的耦合度，才能更好的提高复用性。这方面有很多设计模式的书籍可以进行学习[[1]](#footnote-0)。而最简单有效的提高代码甚至是二进制模块的复用性的方法仍然是使用接口。

### 创造力与破坏力

软件开发是一个比较特殊的工作，程序员可以实现创意者的需求，把想法变为现实，具有很强的创造能力。而同样我们再掌握这重要创造能力的同时，也可能一不小心就发挥相反的一面——破坏力。

破坏力的直接表现是软件的BUG。没有不存在BUG的软件，只有BUG最少的软件！程序所开发出来的产品是承载着整个产品的所有相关的开发人员的心血的，我们在开发时需要更多的注意并考虑更多的情况，唯有认真负责，才可避免BUG的产生。同时我们必须在任情况下做好当BUG发生的处理准备，在最短的时间内实施有效的解决方案， 将BUG带来的损失和影响降到最低！

### 单步所有代码

当代码写完后，应将所有的代码单步调试一遍。对于各种条件分支语句的执行，则可以通过在监视窗口修改值的方法来满足条件从而使得可以执行到相应的条件分支。单步代码是保证代码质量的重要方法，也是发现最低级错误的最佳手乱。请务必认真的将写完的代码单步调试一遍，尤其是“通过修改值而执行条件分支语句的”方法，如果一旦疏忽且在一个没有调试过的分支语句中出现BUG，那么造成的影响将是不可预料的！

我曾经让一位新同事将他刚实现的一个功能模块的每句代码全部调试一遍。当我询问有没有调试那些条件不满足的情况，得到的回答是“我没有满足这样条件的数据，没有办法调试！”。于是我告诉他可以通过监视窗口或内存窗口修改出任意你想要的数据，从而满足执行条件执行任何分支代码。后来这位新同事通过这种方法进行调试后果然在那些没有调试到的分支语句中发现了数个BUG！

唯有认真负责，才可避免BUG的产生。

### 多花一分钟

在一项工具程序的开发中，开发者多花了10分钟，将一个搜索的功能做的更加易用，同时优化了搜索的性能。这直接使得公司所有使用此工具的同事，每次在进行搜索的时候节省数秒钟，而且这个工具的使用频率非常高，为使用者节约了大量的时间。

程序员多花一分钟，为产品挣得十分钟！事实上能够挣得的远非十分钟那么简单。当你在一个细节上多花一分钟，做的更人性化、更强大，那么最终用户每次使用这个功能都将从其中节约到时间。1个用户1次节约1秒，1万个用户10次节约了多少时间？

多花一分钟，消除潜在的BUG和陷阱；

多花一分钟，完善代码的注释；

多花一分钟，提升代码的效率；

多花一分钟，能做得更好……

### 杜绝编译警告

一份代码的健壮性以及含有多少潜在的BUG，与编译这份代码时输出的警告数量成正比。请不要对编译时产生的警告视若无睹，编译器既然提供警告功能，就足以说明警告的重要性。当警告出现，就足以说明代码的质量不佳，在某些场合是会出现逻辑错误的。为了防止潜在的问题，必须采取相应的代码修改措施来合理消除编译警告。

通常在默认的IDE配置下，编译时（后）均会有输出窗口或错误窗口来显示编译时的警告或错误。养成一种习惯，在编译代码的时候盯着输出窗口，每看到一个警告就双击定位代码看看是否有潜在的问题并如何消除，几次之后所有的警告都迎刃而解了。长此以往会发现自己所写的代码的警告越来越少了，代码质量也提升了。

高质量的代码必定是无警告的。

### 语法健壮性

请不要书写下列古怪代码中的任何一种，因为这不能彰显出对语言的运算符优先级掌握的有多深，只会降低代码的可维护性并增加代码阅读者的阅读成本, 我们也根本没必要在这种问题上花费思考的时间！

不要不加括号直接书写容易使人混淆优先级的运算符。

\*ptr++;

a = ~5 + 4 >> 1 + 7 ^ 3;

不要在作为函数参数或操作数的情况下使用对值产生先或后影响的运算符(++、--等)。

foo(++n);

foo(n++);

a = ++b + c;

尽量不要使代码需要拖横向滚动条才能看完一行！一行代码太长就分几行书写。一个函数的代码行数最好控制在不进行垂直滚动就能看完，如果一个函数需要下翻好几页才能看完就已经丧失可读性了。

table.foreachi( MonsterIdInScene, function( i, v ) if (v.monsterid == monId) then MonsteridIndex,MonsterName= i,v.monstername end end )

一行语句中不要出现太多括号，如果括号太多则可以考虑将一行代码分成多行，并使用局部变量保存值，最终一起运算，从而增加代码可读性。

(Class1\*)(((Class2\*)ptr)->getField())->f((Class2\*)ptr)

写switch-case语句，如果分支项多到需要下拉滚动条，那么最好调整语法层次结构不要将switch块放到其他的语法块中而使其直接位于函数的语法域内，否则很容易在日后的修改中造成花括号不匹配。

不要嵌套三元运算(a?b:c)，否则会增加代码阅读难度。

int x = arg1 ? arg2 ? arg3 : arg4 : arg5 ? arg6 : arg7;

不要使用闭包函数，这会除了降低代码的可读性，还会降低维护性以及可移植性。

table.foreachi( MonsterIdInScene, function( i, v ) if (v.monsterid == monId) then MonsteridIndex,MonsterName= i,v.monstername end end )

较多的逻辑条件层次且嵌套代码变得不易读时，应该考虑如何使代码便于维护和阅读。

**VS**

if (expr1)

{

foo1();

if (expr2)

{

foo2();

if (expr3)

{

foo3()

}

}

}

这样的书写方式，即便在很深的逻辑条件嵌套中，也具有很好的阅读性。

if (!expr1) return;

foo1();

if (!expr2) return;

foo2();

if (!expr3) return;

foo3();

如果举一反三的将上面的示范应在一个循环中，那么将同样将优化方案中的return修改为continue也能够起到增强代码可读性、降低修改错误率的作用。

所有难于阅读的代码都不是由于技术能力的差距造成的，与经验的关系也是微乎甚微，如果书写代码的时不图一己之方便，用心的去书写这份代码，那么人人都能写出高质量的代码！

### 合理放置类成员

在一些具有多种继承关系的类中，如果需要添加一个类成员或接口，那么请考虑将将他们放在哪个类中更合适。绝不能将成员一概而就的全部放在最底层的类中！这样做虽然完全可以解决开发的需求、能够实现所需的功能并获得最大的兼容性，但严重违背了面向对象编程的思想。如果这个做法合理，那么我们就没有必要进行类的派生，大可直接将所有的成员、接口和函数都写在一个类中，OOP思想也不用具备继承的概念了！

面对这个问题，我们不用考虑怎样一次性的实现需求并将代码的兼容性做到好，我们大可就只实现眼前的需求即可——**将成员、接口或函数放在最直接使用它的最实际化的类中，满足这个类的使用需求即可**，当有朝一日如果其父类或父类的其他派生类也需实现相同的功能，那么我们在将这些成员、接口或函数移动到父类中也不迟，这就是在面向对象编程极为常见的“重构”。任何我们阅读过的一份结构明朗、思路清晰、关系合理的代码中，其中的类关系和成员关系必定是经过设计人员和编码人员一次又一次调整和重构后才产生的。极少有人能够一气呵成的设计或者编写出一份实现了繁杂的功能且结构非常合理的代码。

例如我们有人物、怪物和实体三个类，人物和怪物都继承与实体。我们现在有个需求是增加一个成员来判断一个人物是否是VIP，那么我们就在“人物”类中增加用于标识是否VIP的成员即可！如果有一天怪物也能成为VIP，那么我在将与VIP相关的成员、函数或接口移动到实体类中也不迟。如果一开始就把这份成员加到了实体类中，那么将由于怪物不具有VIP概念而造成额外的浪费，也使得基类更为复杂，结构也将变得不清晰！

### 合理命名

对象命名包括对类、成员、函数、常量、变量以及宏的命名，同时也包含命名风格。出于尊重大家自己习惯使用的命名风格的目的，我们不强制要求使用某一特定的命名规则。但我们要求：**相同文件中的代码必须具有统一的命名规则，后来修改的人必须遵循先前的命名规则**。既例如一份代码中，惯于使用匈牙利命名规则，那么后来修改这份代码时就不能以成员以“F”开头或以“\_”开头的命名规则，或者原有的代码中所有函数以小写字母开头，那么后来修改的人也不能以大写字母开头！

对一个对象的命名必须能够完整的、没有歧义的表达对象的作用。对象的命名不得使用中文，及时目标编译器支持中文也不要使用；名称必须尽量取正确的英文单词，若难以找到适合的单词，则可以使用汉语拼音；如果一个名称中含有多个单词或拼字，那么必须使用从视觉上即可区分单词界限的命名方式，例如驼峰命名;名称中不要使用过于抽象的名词，能够最直接和准确的表述这个对象的用途的名称是十分有意义的。

常量和定义值的宏，必须全部以大写字母书写，不同单词或拼音之间以“\_”连接！

在C++中类名称必须以一个大写字母C开头；在Delphi中类名称必须以一个大写字母T开头。在任何语言中，类名的第一个字母必须大写。例如CGameMap、TMonster、Sprite。

关于访问限制不同的类成员(被private、protected、public、published、internal等访问类型关键字修饰的成员)，我们不严格要求他们之间的书写顺序。但**必须做到相同的访问限制的成员、函数和接口写在一起**。既protected的前后都是protected，而不能将一个protected加载在一堆上下都是public的成员之中！

### 立刻考虑释放它

养成“立刻考虑释放他”的习惯将使我们的编码水平得到很好的提升，也能使得我们书写的代码更具健壮性。**凡当我们写下构造后，就当立刻去书写释放，凡当我们监听事件后，就当立刻去写移除事件监听**！

凡当我们写下new后，就应该立刻去写delete的代码；凡当我们申请一份内存后，就应该立刻去书写释放内存的代码；凡当我们写下TClassName.Create后经应当立刻去写Free语句；凡当我们写下addEventListener的时候，就应该立刻去写removeEventListener的语句！做到这样的习惯，能写出含有内存泄露代码的概率就微之甚微了！如果你现在已经具备了这样的习惯，那么请坚持并将它带给你身边的人。如果你还没有养成这样的习惯，那么请再工作和学习中注意培养这一习惯，**这是成为一个合格的程序开发人员的最基本的习惯！**

### 善用局部变量

在不会对局部变量初始化且在编译时没有警告的IDE或语言中书写代码（例如VC），最好在变量声明后就给予一个默认值。一个局部变量如果用做循环下标，那么请不要在循环外出于废物利用的想法再使用这个变量。一个局部变量只用于一种用途，不要为了降低局部变量开销或者图省事而将一个局部变量做多次不同意义的用途。大可直接无视局部变量的内存和性能开销。局部变量的命名最好有显著的字母差别，防止因为变量名称太像而造成书写或粘贴错误。下面将列举一些经典的例子：

int i;

for (i=0; i<count; ++i)

{

exprs;

}

s = i;//这种用法极易产生隐患

int lv = actor1.level;

exprs;

lv = actor2.level;

exprs;

foo(lv);//以后没看清代码，极有可能还以为lv的值是actor1.level！如果单独谁用一个局部变量保存actor2.level则此问题就不会发生了

int nEffect = expr;

void \*pEffect = getEffInst(nEffect);

if (nEffect)//事实上这里是要判断pEffect不为空,而这个书写错误很难被发觉

expr;

以上不提倡的行为都会降低代码维护性，为日后修改代码埋下地雷。

### 多写注释

当自己回头看一份半年前你亲自写的代码，如果没有文档，没有注释，则要花很长的时间来找回当年的思路，更何况如果将这份裸体代码给其他人看呢。只有含有良好注释的代码，才能够具有更好的维护性，更便于被他人阅读和领会，对产品的可持续性带来很大的促进作用。

对注释的书写，应当能够禅明所要表达的意图，在书写关键部位（重要逻辑思想、解决过BUG的地方、容易产生BUG的地方）注释之前，请设想一下，如果有其他人正在看这份代码，那么要如何书写注释才能让他人更容易理解并且不产生歧义。

为了加强代码的可读性和可维护性，在我们的规范中，要求每个类、结构、函数、成员、枚举值、常量都要书写注释。这是技术部硬性的制度规定，**写注释是开发工作中的一项职责，是工作的一部分**！技术总监和项目主程也将不断抽查代码，对于忘记写注释的地方会提醒并督促书写。

### 常按保存键

大多数开发软件中的“保存”按钮的快捷键是Ctrl+ S，请经常按下一下这个快捷键，写完一点，保存一点。可以避免因软件崩溃、死机、断电等不可预料的情况造成的数据丢失！如果有“全部保存”功能，那么按下全部保存按钮会更稳妥，通常全部保存的快捷键是Ctrl+Shift+S。

### 善用快捷键

工欲善其事，必先利其器。我们要想用好一个软件，那么是必须要掌握他的快捷键的。使用快捷键完成一个功能的效率要比使用鼠标通过菜单或按钮来进行操作要高效得多！例如当你看到一个按钮或菜单的文字后面带有”(K)”这样的符号，那么Alt+括号中的这个字母就是这个按钮或菜单项的快捷键。通常打开一个模态的对话框后，按ESC键均能将其关闭；这些快捷键规则已经形成潜规则了。

了解一款软件的快捷键的最佳方法是阅读其帮助文件中的快捷键章节，并且在实际的使用过程中谷时常使用这些快捷键从而形成习惯。

### 注意调试输出

许多开发者由于调试环境受限或个人习惯的原因，喜欢使用打印内容的方式输出一些信息用于进行调试分析。这不失是一种好的做法，但无论如何请别忘记在功能调试通过后删除这些输出！一个大的工程往往是由一个团队中的数个人参与的，如果没人都留一点无用的调试输出，那么最终这个软件在运行的时候就是不断的在输出调试内容，而当遇到问题要去看调试输出什么样的值的时候，就如同大海捞针，找半天也未必找得到有意义的输出的内容。

在进行C++开发的时候，推荐的方法是包含wylib库中的\_ast.h并使用TRACE宏来输出调试内容，这样的内容只在有调试器附加的时候才会直接输出到调试器的输出窗口。而控制台窗口则留给重要内容输出用。但即便如此，也请将调试通过的暂时不再有意义的输出语句注释掉，以防干扰他人的调试输出。

### 尊重用户习惯

曾有一位负责游戏数据统计后台开发的同事将后台进行了一番彻底的升级，升级后实现了一些让人兴奋的新特性支持。而升级造成与原存在版本在部分细节上的不兼容，并且导致访问地址也要修改。为此在更新前我特意要求保持地址不变，至少保持旧的地址可以跳转到新的地址中。所有使用这个后台的用户都是将页面收藏到浏览器的收藏夹，并且已经使用了相当长的一段时间，其中有我们公司的也有众多合作公司的用户，如果修改了访问地址将导致必须通知所有人地址变更了并且使用者要重新收藏页面。

尊重用户的行为习惯和操作习惯，不要随意修改一个已经向最终用户发布的功能的操作方法，除非新的操作方法经过研讨后被确认具有更好的操作性。作为开发者可能认为一些细小的修改不会引起大的反应，但事实上这个功能的使用者们从对这个细微操作的长期使用以来形成使用习惯，这就是用户习惯。一旦将之改掉，将使这些使用者重新适应新的操作规则！

尊重用户操作尤其是客户端开发中更为重要，在操作性问题上最好不要抱有“创新”的的思想，在操作上的创新的代价很高，而且收获也不高。顺应用户的行为习惯，用户才愿意了深入的解我们的产品。例如我们在“善用快捷键”一章中见到的常用快捷键的“潜规则”，如果某一天你接触到的一个软件，他的快捷键规则不遵循这些潜规则，那么你能否使用得顺手？而当你用不顺手的时候这个软件的下场就是被卸或删除，因为他违背你的惯性思维习惯，而且不值得你去为它改变你的习惯。

### 选择最适合的

条条大路通罗马，实现一个功能的方法有很多种，但最优的方法往往只有一种。牛刀固然可以用来杀鸡，但用杀牛的刀来杀鸡可能要花更多的力气，也仅仅只是把鸡杀死了而已，如果直接用专门用来杀鸡的刀来杀鸡，鸡同样会被杀死而且不用枉费力气。在实际的开发中，我们常常会走入使用牛刀杀鸡的误区却难以自知。在实际的开发中存在着一些方法可以在广度上解决很多问题，但是往往也会到来额外的开销，在对一个需求有明确的想法知道该如何做之后，请再考虑一下有没有更好的方法，会不会有额外的开销！选择对需求的实现方案时要充分考虑：**能否解决问题，是否没有新的负面问题产生的，是否是额外开销的**。

### 一蹴而就

在进行实际的设计和编码时，最好的方法是一气呵成完成一个系统或系统中某个功能的开发。在实际编码的时候，请将代码、注释、优化一次性完成，在一边写代码完成功能的同时，一边注意书写注释，一边考虑实现的方式是否是最优的，并且消除所有编译警告，最后单步通过所有的代码并删除那些没用的调试输出。千万不要抱着“我先实现功能，然后再……”的心态！因为大多数人把一个功能实现并被验收之后，几乎不可能会主动再去优化这份代码，或者再去回头写注释，甚至都懒得回去删除无用的调试输出。

### 慎用字符串

能用整数的地方绝不使用字符串！字符串的确具有很广的适用性，可以用于描述和存储绝大多数的数据。但使用字符串的内存开销以及性能开销都比使用原子类型（char、short、int、double等）要大得多，当你打算将一个类的成员的数据类型或者函数的参数设计为字符串的时候，请再三考虑能否使用原子类型替代。

在AS3编程中，由于没有“枚举(enumerate)”的概念且flash SDK中倡导使用字符串常量作枚举定义值，事件名称的定义也使用字符串，因此潜移默化的对大部分AS3程序员形成了优先使用字符串的影响。但这往往不能（完全）胜任性能要求高的需求，通过我们数年来的实际开发的经验和一贯的坚持，在此请注意：能使用整数或浮点数描述的绝对不使用字符串！

在C/C++中进行开发，需要使用字符串时应当首先考虑使用字符数组或字符指针来传递、存储字符。尽量少用或不用字符串类。

更少的字符串，将更大的提升程序的性能并降低内存开销！

### 整数与浮点数的选择

整数和浮点数都属于原子数据类型，在PC处理器中他们是直接由硬件支持的。但就目前硬件对整数和浮点数的处理效率看，CPU进行整数预算远比FPU进行浮点预算要快得多。因此，绝大多数情况下我们均会使用浮点数。而在运算结果要求浮点数的场合，参与运算的数据类型则应当使用浮点数，而不能出于降低内存开销的目的而使用整数。使用整数参与浮点运算，则只会降低性能！

class Foo

{

int m\_Field1;

double calcValue(int arg1)

{

return 0.3 \* m\_Field1 / arg1;

}

};

以上代码例演示了一种不合理的使用整数的方法，正确的做法是成员m\_Field1以及arg1均应该使用浮点数。通过代码我们看到，函数的返回值的类型是浮点数而使用整数参数浮点运算，只会降低性能。

### 避免立即数

“立即数”可以通俗的理解为“常数”，既我们日常开发中在代码中实际书写的最基本的源数据。例如100、0.3、”字符串”都属于立即数。

**凡是为实现一个功能相同的数据运算而书写的立即数中，不允许在代码中出现一次以上值相同的立即数**，而应该将其书写为常量。遵守这一原则对于提高代码的可维护性具有重要意义。

示例代码

p1 = 2 \* 3.1415926 \* (r1 \* r1);

p2 = 2 \* 3.1415926 \* (r2 \* r2);

以上代码中的立即数3.1415926应该书写为一个常量PI。

wnd1.x = (1000 – wnd.width) / 2;

wnd1.y = (600 – wnd.height) / 2;

wnd2.x = (1000 – wnd.width) / 2;

wnd2.y = (600 – wnd.height) / 2;

以上代码中的立即数1000和600都应该被书写为常量。

## C++开发注意事项

C++语言是目前使用最广泛的语言，并且由于具有可将此语言直接编译为机器代码的编译器，从而使得使用C++开发的软件具有硬件级的效率，再加上语言的开放和部分优秀编译器的免费性，使得语言能够广泛的在世界各地被开发者使用。但由于其语言的灵活性以及标准库的缺陷性，导致在诸多方面的开发中没有形成向Java那样具有广泛的行业统一的设计和开发标准，形成几乎每个技术团队都有开发的独特性的情况。同样也由于语言过于灵活，语言的设计和核心库中存在诸多不严谨的规则，从而形成了诸多开发中的“陷阱”，这些陷阱几乎被所有的C++开发者所诟病，即使在富有经验的老牌C++程序员，也时常陷入这些陷阱中。由于缺乏行业标准的框架和通用库，导致诸多应用层与系统层之间衔接的部分都要独立开发的情况。

为了在实际的开发中，充分发挥使用C++的种种优势并尽量避开各种“陷阱”，将对我们在数年的开发中总结的宝贵经验整整理成章，作为公司C++语言开发交流和学习的资料，并作为规范实际开发的指南和标准。

### 不使用引用

C++语言由于诸多语言设计上的原因，包含很多被大家诟病的“陷阱”。即便是再富有经验的老手，往往也不能全部避开这些陷阱，时不时的也掉进去一下。

引用与指针相比，带来的最直接的好处仅仅是在写代码的时候的差别，访问指针成员需要”->”运算符而访问引用成员则只需使用”.”。在带来这样的便捷性的同时也带来了最容易在高手和新手之间出现问题的，因此应该尽量避免使用引用，取而代之使用指针来传值。

下面列举一些常见的因使用引用而带来的问题，都是非常难以发现的。

class Class1

{

Class1 foo() //这里本意为Class1& foo()，但写漏了&

{

this->calcTime = time();

return \*this;

}

}

int main()

{

Class1 c1;

foo(c1);

}

上面的例子中很难发现foo的参数中少了一个”&”符号，带来的问题是每次调用foo函数都会构造并拷贝一个Class1的实例，并在函数调用后再析构这个临时的对象。这里的代码确实不会产生运行时错误或者语法错误，甚至不会产生警告。这份问题代码可能存在并持续使用数年都不会被发现！

Class1 &c1 = arg1;

for (;condition();)

{

c1 = next();

c1.calcTime = time();

}

上面的代码片段中存在这样一个问题：代码书写者写下c1 = next()语句的本意是将next()的返回的对象保存在变量c1中，继而对这个值进行修改和运算。但实际上c1 = next()的结果是将next()的返回值拷贝到c1（也是arg1）中了！接下来的对c1的修改影响的都是arg1而不是next()的返回对象！

如果在实际开发中使用指针代替引用，则以上问题均不会发生！

### 注意字符串拷贝

出于节省内存以及加快运算效率的目的，我们在项目开发中常使用字符数组来做字符串使用。而使用字符串时常会遇到字符串拷贝的操作，而在向字符数组中拷贝字符串内容的时候往往忘记检查源字符的长度与目标字符串数组的内存长度，从而造成将较长的字符串拷贝到缓冲区较小的字符数组中造成的内存越界，从而造成数据被破坏，产生不可预料的结果。

解决数组内存拷贝越界的问题重点在于规范化这一操作的方法，既然很难时时刻刻想到要检查数组缓冲区，那么最好的方法就是养成使用一种可以自动检查缓冲区并防止越界的拷贝方式的使用习惯。**在向行字符缓冲区拷贝字符串时，应使用\_asncpyt宏**；宏中实现的字符串拷贝功能会检查字符缓冲区的长度并保证不会越界且保证字符缓冲区是0结尾的。**向数组缓冲区拷贝数据时，应使用\_arrncpy宏**。这些宏在游戏通用代码库的ShareUtils.h中。

### sizeof技巧

sizeof运算是一个编译级运算，即sizeof的结果在编译期间就确定的，最终会被编译器便以为一个立即数而不会在运行期间产生任何运算开销。在诸多的示范代码、书籍中给出的含有sizeof操作的代码有很多都是sizeof(类型)的写法。甚至有开发者认为sizeof的操作数必须是数据类型，但事实上sizeof运算的操作数不仅支持数据类型，还支持变量、常量甚至任何结果不为void的运算表达式！

以下sizeof表达式都是被编译器支持的

sizeof(int);

sizeof(this);

sizeof(\*this);

选择使用哪种sizeof的书写方式影响最大的体现在代码的可维护性上。从上面的示例中看出后两种sizeof的写法比第一种写法具有更高的可维护性。我们将通过下列代码示范阐明这一原则的重要性以及带来的可维护性。

const int len = 100;

int \*ptr = new int[len];

ZeroMemory(ptr, sizeof(int) \* len);

**VS**

const int len = 100;

int \*ptr = new int[len];

ZeroMemory(ptr, sizeof(\*ptr) \* len);

请注意上述代码示范中的前后两种写法中的不同在于sizeof的操作数一个使用类型，而另一个使用表达式。如果这份代码就此使用1年，之后需求变更，要将ptr的类型从int修改为short。那么第一份代码将产生内存写越界，因为可能只是将ptr的类型从int\*修改为short\*，而忘记将下一行语句中的sizeof(int)修改为sizeof(short)。但第二份代码将不会因ptr的类型变化而产生问题，因为其中的sizeof(\*ptr)语句具有很强的适应性，运算的结果会随ptr类型的不同而自动改变！

所以，**sizeof运算的操作数应当使用变量名称或表达式，避免使用类型**。从而提升代码维护性，降低日后修改代码而产生的风险。

1. 请上网搜索关键字“C++设计模式”或“Java设计模式”。 [↑](#footnote-ref-0)