**程序开发指导**



——聪明的人会将别人撞得头破血流的经验作为自己的经验，愚蠢的人只会自己撞得头破血流才叫经验。

一个员工，能给他的同事、上级带来什么样的印象，将导致他将来的职业生涯如何发展……

# 前言

从人类文明得以延续和发展的根本在于传承。现代人比古代人懂得多，而现代人并非从古代一直活到今天的。后人能够比前人懂的更多，了解的更多，其根本在于后人学习和吸纳了前人的经验，并从中总结精华！

着眼于我们现在的开发工作，新人如果能够从有经验的“前人”或“高手”身上学习宝贵的经验，将加快新人专业技能的学习速度，尽快成为高手。同时前人如果能不断的将自己宝贵的经验分享给新人，那么我们的团队将始终有新的活力补充，我们的事业才能不断的迈向新的高度。

本手册旨在规范、指导软件开发工作中需要遵循的原则，从而避免因经验不足、专业技能不足的问题而走弯路，减少因开发者思维倾向的不同而造成不必要的损失。如果暂时不能理解某些原则的意义，那么可以请你的直属上级给你讲解。

如果你有新的可以作为开发原则的体会和想法，请及时和技术总监分享，技术总监会将有价值的思想加入为新的开发指南！

目录

[前言 1](#_Toc300871692)

[程序开发指南 5](#_Toc300871693)

[态度决定回报 5](#_Toc300871694)

[执行力 5](#_Toc300871695)

[优先注重的 5](#_Toc300871696)

[稳定 5](#_Toc300871697)

[性能 6](#_Toc300871698)

[存储 6](#_Toc300871699)

[传输 6](#_Toc300871700)

[持续 6](#_Toc300871701)

[易用 6](#_Toc300871702)

[开发习惯 7](#_Toc300871703)

[可维护性 7](#_Toc300871704)

[可复用性 7](#_Toc300871705)

[一蹴而就 7](#_Toc300871706)

[创造力与破坏力 7](#_Toc300871707)

[单步所有代码 8](#_Toc300871708)

[多花一分钟 8](#_Toc300871709)

[杜绝编译警告 9](#_Toc300871710)

[语法健壮性 9](#_Toc300871711)

[善用局部变量 10](#_Toc300871712)

[多写注释 11](#_Toc300871713)

[常按保存键 11](#_Toc300871714)

[善用快捷键 12](#_Toc300871715)

[注意调试输出 12](#_Toc300871716)

[尊重用户习惯 12](#_Toc300871717)

[选择最适合的 13](#_Toc300871718)

[慎用字符串 13](#_Toc300871719)

[整数与浮点数的选择 13](#_Toc300871720)

[不使用引用（C++） 14](#_Toc300871721)

[性能优化通用法 15](#_Toc300871722)

[性能优化的原则 15](#_Toc300871723)

[重在设计 16](#_Toc300871724)

[预运算 16](#_Toc300871725)

[数据结构优化 17](#_Toc300871726)

[空间换时间 18](#_Toc300871727)

[局部变量优化 18](#_Toc300871728)

[循环优化 19](#_Toc300871729)

[索引优化 21](#_Toc300871730)

[容器选择 22](#_Toc300871731)

[查找优化 22](#_Toc300871732)

[删除优化 22](#_Toc300871733)

[分支优化 23](#_Toc300871734)

[函数优化 26](#_Toc300871735)

[减少API调用 26](#_Toc300871736)

[指令级优化 27](#_Toc300871737)

[AS代码优化 27](#_Toc300871738)

[事件监听优化 27](#_Toc300871739)

[只读数据单例化 28](#_Toc300871740)

[不使用动态类 30](#_Toc300871741)

[少用字符串 30](#_Toc300871742)

[定时器优化 30](#_Toc300871743)

[针对AS的局部变量优化 31](#_Toc300871744)

[按需初始化 32](#_Toc300871745)

[多用Profiler 32](#_Toc300871746)

[LUA代码优化 32](#_Toc300871747)

[local修饰 32](#_Toc300871748)

[合理使用表 33](#_Toc300871749)

[图形优化 35](#_Toc300871750)

[图形优化原则 35](#_Toc300871751)

[图像压缩 35](#_Toc300871752)

[位图的内存消耗 36](#_Toc300871753)

[动画原理 36](#_Toc300871754)

[Flash中的图形优化 37](#_Toc300871755)

[数据库优化 39](#_Toc300871756)

[数据库优化重点 39](#_Toc300871757)

[表和字段优化 39](#_Toc300871758)

[优化查询语句 40](#_Toc300871759)

[优化删除 40](#_Toc300871760)

[内存操作优化 40](#_Toc300871761)

[减少内存申请和释放 40](#_Toc300871762)

[使用连续的内存 40](#_Toc300871763)

[减少内存拷贝 40](#_Toc300871764)

[使用优秀的内存管理器 40](#_Toc300871765)

[比特位代替布尔 40](#_Toc300871766)

[数据结构设计 41](#_Toc300871767)

[逻辑型数据结构 41](#_Toc300871768)

[存储型数据结构 41](#_Toc300871769)

[通信型数据结构 41](#_Toc300871770)

[开发经验 41](#_Toc300871771)

[配置文件 41](#_Toc300871772)

[数据与代码分离 41](#_Toc300871773)

[时间值的使用 42](#_Toc300871774)

[使用轴时间 42](#_Toc300871775)

[默认值的选择 42](#_Toc300871776)

[惰性原则 42](#_Toc300871777)

[先取后给 42](#_Toc300871778)

[重复检查条件 42](#_Toc300871779)

[更安全的加锁 42](#_Toc300871780)

[异常处理 43](#_Toc300871781)

[同帧共时 43](#_Toc300871782)

[频率控制 44](#_Toc300871783)

[64位化 44](#_Toc300871784)

[SPID 44](#_Toc300871785)

[BUG查找方法 44](#_Toc300871786)

[断点调试 44](#_Toc300871787)

[附加调试 44](#_Toc300871788)

[远程调试 44](#_Toc300871789)

[进程转储 44](#_Toc300871790)

[内存读写断点 44](#_Toc300871791)

[排除法 44](#_Toc300871792)

[逆向追逐法 44](#_Toc300871793)

[程序员之外的 44](#_Toc300871794)

[善于学习 44](#_Toc300871795)

[注重分享 45](#_Toc300871796)

[整理桌面 45](#_Toc300871797)

[技术总监的桌面 45](#_Toc300871798)

[留给人好印象 46](#_Toc300871799)

[机会 47](#_Toc300871800)

[坚持成为习惯 47](#_Toc300871801)

[实践 48](#_Toc300871802)

[补充 48](#_Toc300871803)

# 程序开发指南

## 态度决定回报

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！先做人，再做事！

## 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段。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！所以最好的方法就是一气呵成的将代码做到尽可能的完善！

### 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。

## 性能优化通用法

程序的性能表现在诸多方面，而本节所讲的性能是指狭义上的“代码时间消耗量”，其定义为：**在相同环境下，处理等同的逻辑事物和数据，按不同的设计思想和实现方法，消耗时间越低的，效率越高**。

软件编程学术是从数学领域分支来的，可以算是数学领域的一个子集，数学的最大特点在于其在规律性。软件优化也具有一些规律可以遵循的，掌握这些规律，就可以应用在具体的开发中，提升软件的执行效率。下面将列举在我们数年的开发中总结出的优化经验和原则。

### 性能优化的原则

并非只有加减乘除等数学运算才叫运算，取类成员、取指针成员、函数调用以及条件判断等都叫做运算。**没有加快运算的方法，只有减少运算的方法！**这阐明了进行性能优化的核心思想，即不能通过加快运算速度来提升性能，而只能通过减少运算量来提升性能。性能优化的本质在于通过删除不必要的运算、不做重复的运算来减少运算的次数，从而达到性能优化的目的！

**最快的运算是没有运算！**

**最快的变量是局部变量！**

**最快的查找是直接索引！**

**最快的删除是标记删除！**

**最快的判断是直接命中！**

**最快的调用是没有调用！**

### 重在设计

检查发言中的屏蔽词，是一个非常简单的功能，他需要实现的仅仅是判断一些关键字是否存在于一串输入文字中，并且绝大多数入门的程序员均能实现这个简单的功能。这个功能不仅在游戏行业中用到，IM软件、邮箱、搜索引擎等都会用到。为了实现这个功能大部分人第一时间想到的就是在一个循环中使用字符串查找函数轮流在输入串中查找，如果找到则表明匹配了关键字。而在当前社会环境下，屏蔽词的数量一直有增无减。看着一天天增多的屏蔽词，提升关键字的匹配的效率已经成为对这个功能优化的重点，低效的字符串查找已经远远不能胜任在几万甚至几十万单位的关键字中进行匹配的需求了。那么此时就需要一种能够在海量关键字中进行快速匹配的新算法和设计方案来加快匹配效率，降低搜索开销。在wylib中的fastFT中提供了一种解决这个问题的实现并在数个产品中得以应用。如果有兴趣可以找时间阅读代码，限于篇幅的关系，本文不对其的实现原理和方法进行讲解。

性能优化的立足点在于设计上的优化。有道是“差之毫厘，谬以千里”。所以无论是设计还是编码，从动手之前我们就应该充分的将“性能”要求放进去，时刻提醒自己有没有更好的方法达到减少运算的目的。

### 预运算

**预运算的核心在于减少产生相同结果的运算次数**，保留第一次运算的结果供以后使用，直到影响这个结果的其他值发生变化时再重新计算这个结果。这个优化原则无论在设计还是编码的实际工作中都是核心的性能优化原则，掌握并熟练使用这一原则就掌握了性能优化的关键！

仔细回想曾经书写过的代码，看看有多少代码总是在进行一些结果相同的重复运算。而这些重复运算如果进行优化，那么对性能的提升将是立竿见影的。

例如，在我们目前的数款服务端中使用的“同帧共时”优化方案。方案假设在一个逻辑处理帧中，一帧的时间足够短，短到所使用的最小时间精度（一毫秒）以内，那么在帧的开始时刻我们调用获取当前系统时间的API函数，并保存这个值为全局的静态值，帧的所有展开调用的函数中，涉及到获取当前时间的操作全部直接读取这个值[[2]](#footnote-1)，而不再调用获取时间值的函数，从而减少了运算次数（函数调用也是运算，而且是很大开销的运算）。

再比如我们在客户端中，在用户技能的数据结构中本没有当前等级的技能配置对象这一成员，我们虽然可以通过增加一个属性访问器，并在访问器函数中通过技能等级在相应的配置中找到技能并获取技能等级数据，但每次访问这个属性都会引发其他的一连串的函数调用，而这这些调用的结果通常都是相同的，那么大可直接保存第一次获取的结果，以后的属性访问则直接返回这个结果。

总结来讲，能不运算的绝不运算；能只运算一次的，绝不多次运算；所需要付出的仅仅是多花一分钟：设计者多花一分钟考虑如何减少运算；实现者多花一分钟进行预运算。

### 数据结构优化

本小节所阐述的观点和原则，只适用于可编译被为机器码的语言和编译器环境，不适合任何解释型语言。

从数据结构上进行性能的优化手段往往不被重视，且数据结构一旦设计成型后，开发者大部分的时间和精力都投入到了如何解决逻辑中的各种问题中，从而难得再想起数据结构设计是否合理，是否满足优化原则。因此，必须主动出击，**在设计以存储运算参数或结果为主的数据结构时，重点关注数据结构的运算性能**。

数据结构对效率的影响体现在数据结构中各成员的数据类型以及结构成员的顺序上。不要小看如何选择成员的数据类型以及按什么顺序排列成员所带来的性能差异。如果选择不当，甚至可能要额外搭上一些不曾算过也不容易被察觉的额外开销。我们已经强调过在“设计以存储运算参数或结果为主的数据结构时，重点关注存储的运算性能”。要提高运算性能，必须正确的选择成员的数据类型。

**使用与当前处理器位数相同的整数是发挥处理器最大效率的方法**。若当前的处理器是32位的，那么使用32位的整数作为成员类型是最合适的，即使成员用不了那么大的数据类型也要选择32位的，而不应该去使用char、short等更省内存的数据类型。如果处理器是64位的，那么尽量使用64位的整形作为成员类型。在C++中可以通过宏或预处理条件定义来达到32位和64位的数据类型自动识别的功能，在Delphi中可以使用NativeInt来自动识别。

**尽力避开对字符串的使用，哪怕是字符数组指针**。凡是使用字符串的地方，是没有多大性能改善的机会的！影响的是所有使用此数据结构的逻辑处理函数。

**使用double而不是float**。double无论在性能还是数据容量（精度）上都比float有优势，且FPU硬件的浮点数处理也是以高精度浮点数进行的，唯一不比float的是它比float多占一倍内存。目前大多数常见的PC处理器的FPU都支持80位浮点数，在编译器支持的情况下可以直接使用80位浮点数来取代double。

**注意成员字节对齐**。绝大多数编译器在默认的编译选项中，会对数据结构进行字节对齐以及内存补齐[[3]](#footnote-2)。初中级程序员往往会忽略这一点从而造成不必要的内存浪费和隐性的性能开销。如果对一个数据结构使用了自定义的字节对齐大小，且因没有注意字节对齐的问题并将一个成员放在了无法被直接寻址的位置，那么每次存取这个成员都将付出额外的代价。这个例子如下图：

#pragma pack(push, 1)

class C1

{

char m\_btField1;

int m\_nField2;

}

#pragma pack(pop)

该例中对m\_nField2的存取是低效的，因为m\_nField2的便宜位置不是硬件直接能够存取的位置，对于这个问题的解决方案就是使数据成员的偏移位置能能够被数据成员的大小除尽。在m\_nField2的前面增加3个字节的保留成员，即可恢复对m\_nField2的存取效率。因此，**保持成员的偏移位置为该成员字节大小的整倍数，才能保证此成员的存取效率**！

### 空间换时间

原则上大部分的时间性能优化会产生额外的存储空间（内存）消耗，通常是因为存储预运算的结果需要花费存储空间，或者为了满足运算而选择更得当的数据结构成员。当然也有时间和空间都得到优化的设计方案。因此，在需要优化代码执行效率时如果需要消耗更多的存储空间，那么只要消耗的存储空间在可以接受的情况下，就应该果断的牺牲存储空间，换取效率。

### 局部变量优化

这与前面提到的“预运算”原则一样，实际上是要求在一个函数过程中，将频繁使用到的运算表达式的结果保存到一个局部变量中，则对表达式的运算就被优化为了对这个变量的访问，带来的性能提升将不仅是几倍而已，尤其是在Java、AS3、LUA等解释型语言中尤其明显。将这部分独立为一个小节单独讲解，是因为在所见到过的实际的代码中，大部分开发者对局部变量的优化做得太少，如果对这一方面加以改进，将获得十分丰厚的性能收获。

这种函数级的通过局部变量减少运算次数的措施在编译器领域称为“窥孔优化”。在一些优化能力强的编译器中通常能够在函数内识别并优化一些形式简单的重复运算。即便如此，也仅是能够在最大化优化代码的编译选项中进行简单表达式的窥孔优化，并不能做到完全识别和优化。因此这就需要开发者在实际的代码书写过程中注重这一优化原则，主动对重复的运算进行预运算优化。

局部变量优化示范

**优化前**

for (int i = 0; i < 1000; i++)

{

lookupResponse(list[i]->getRequest()->opid);

if (checkTimeOut(list[i]->getRequest()->time))

doRequestTimedOut(list[i]);

else doWaitEvent(list[i]);

}

**优化后**

int i;

CAuthInfo \*pInfo;

CHttpRequest \*pRequest;

for (i = 0; i < 1000; ++i)

{

pInfo = list[i];

pRequest = pInfo->getRequest();

lookupResponse(pRequest->opid);

if (checkTimeOut(pRequest->time))

doRequestTimedOut(pInfo);

else doWaitEvent(pInfo);

}

在本优化示例中，将循环中一些其值总是相同的重复的运算保存到了局部变量中，从而减少了运算次数。在1000次循环内，将至少3000次的list[i]下标运算函数的调用减少为1000次；将2000次getReuqst()调用减少为1000次。仅在存在有1000条数据的前提下，优化前和优化后对此函数一次调用的性能差别将是巨大的，仅仅将几个重复运算保存到局部变量就能减少3000次函数调用！

在数据存取速度的角度进行优化，也将很大程度提升代码的性能。数据存取的速度依次为：**局部变量和函数参数最快，类成员第二，全局变量第三，指针成员第四、函数调用最慢**。在这其中，局部变量和函数参数是访问速度最快的，这恰巧也能一定程度上弥补后面各种存取速度不足的问题，方法就是将其他类型的数据保存到局部变量中即可！当我们在函数中或循环中频繁的使用一个类成员或全局变量的时候，不妨将他先保存到局部变量中，并且访问这个局部变量。

在一些优化程度较高的编译器中，局部变量甚至可以被优化到寄存器中，CPU中再也没有比从取寄存器中存取值更高效的存取操作了！而当前几乎所有的处理器架构设计中，都致力于加快栈数据（局部变量的常规存储位置）的存取速度，所以请尽最大的可能使用局部变量进行优化吧。

### 循环优化

循环的使用已经是实际开发中司空见惯的方法了，尤其是for循环。但不同的循环语句写法将直接导致循环效率的巨大差异。

C++代码:

for (int i=0; i<pList->count(); i++)

AS代码:

for (var i: int = 0; i<list.length; i++)

以上写法是我们司空见惯的循环写法，就我见过的绝大多数的开发者闭着眼睛凭直觉都能写出来而且从来都不会觉得这样书写循环存在什么问题。现在，我们来分析一下问题究竟出在哪里——在for循环中，每进行一次循环都会检查一次是否满足继续循环的条件，如果不满足，那么终止循环。而以上C++代码示范中的条件语句为i<pList->count()，意味着每循环一次，这个条件表达式会运算一次，pList->count()函数也就会被调用一次！而pList->count()的返回值越大，则这个函数被调用的次数也越多，继而导致这个条件表达式被运算的次数也就越多。而在循环期间如果pList->count()的返回值不会变化（大多数实际情况下都是如此），那么我们进行的多次调用就都是白白浪费的！示例中的AS代码同样存在相同的问题，对list.length的属性访问实际上也是函数调用。正由于违背了减少运算次数的原则，因此这样的循环语句怎能高效？

这个for循环语句只需按我们减少运算量的原则，使用“局部变量优化”法进行优化，即可将效率将大大提升。

C++代码:

int count = pList->count();

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

AS代码:

var count: int = list.length;

for (var i: int = 0; i< count; i++)

在我们所常使用的开发语言C++、Java、AS、PHP等语言中，每次循环都会重新判断一遍循环继续条件，而循环继续条件表达式的复杂性和开销将直接影响循环的性能。而在Delphi、LUA中则由于语言特性而不存在这样的问题。因此，**不要在循环终止条件中使用非局部变量表达式，尤其不要使用函数调用的返回值**。

以上的示范代码使用局部变量优化法减少了运算量，从而提升了运算效率，但实际上上述循环语句还有可能进行在优化的余地。而优化的方法则是考虑能否使用降序循环。

C++代码:

for (int i=pList->count() - 1; i>=0; --i)

AS代码:

for (var i: int=list.length - 1; i>=0; --i)

使用降序循环不仅可以加快循环速度（CPU做减法指令比加法快）；也能提高代码可维护性，降低日后修改代码产生BUG的几率。使用降序循环时，不用担心在循环中从数组中移除对象，也能避免很多由此带来的问题。因此，**凡不要求数据必须按从前到后的顺序进行处理时，均应使用降序循环**。

### 索引优化

索引操作通常指代在数组中进行数据存取，既array[n]的代码形式。n在当前不同语言种类中可以是整数、枚举类型、字符串等。越高级的语言中，下标的允许类型越丰富，因此正确的辨别不同的下标数据类型的效率并选择最优的使用，将是决定对数组数据的存取效率的关键。本节所讲的索引优化仅针对所有支持数组的语言的直接数组（表）下标索引运算，而不包括具有运算符重载特性的语言中对下标运算符的重载。

常见的，在AS和LUA中数组可以使用字符串做下标来进行索引。实际上我们把这种可以通过非原子数据作为下标的数组访问看作是高级语言的一种KV表。K表示“键（Key）”，V表示“值（Value）”。K和V是一一对应的，一个K关联一个V。我们通过分析LUA中对KV表的索引实现来分析KV表的性能。LUA对KV表的实现可以算是在常见的解释语言中具有最高效率的。

在LUA中的v = t[“name”]的实现流程为：

* 对字符串”name”计算哈希值
* 通过哈希值进行下标索引值运算
* 通过索引值从t的哈希数组中取出数据值
* 将值保存在v中

实际上的实现会更复杂一些，条件也会更多一些，但这里只列举关键逻辑部分。由流程可以看出，对KV表的实现最终还是依赖原子数据下标的数组的，只是在此之上实现了一个将字符串转换为哈希值的算法，并通过算法来确定字符串对应的下标值。我们再通过CPU进行数组下标访问的角度分析一下原子类型下标数组的读取流程：

在CPU中v = a[n]的实现流程：

* 读取（数组内存起始地址 + 数组类型字节大小 x 索引值）处的内存
* 将值保存在v中

实际上机型数据索引的操作就只有一句“读取（数组内存起始地址 + 数组类型字节大小 x 索引值）处的内存”。而这句读取来煞费语气的文字实际上在CPU中只是一个移动指令MOVE EAX, DWORD PTR[ECX + 对象大小 \* n]而已！即使是在高级解释型语言中，对使用原子类型作为下标的数组的存取效率固然还是要比通过例如字符串之类的KV表要远快得多！尤其是在LUA中，语言设计者更是处于性能考虑而将表的实现划分为KV表和数组表。以便在注重性能的场合可以提供更高的效率。我们在前面的章节中也曾也讲到过要“慎用字符串”。

**在实际的设计和编码中，应当尽量使用基于整数为下标的数组存储数据**。无论是服务端用于定位对象的句柄，还是客户端用于定位物品配置数据的id值，我们从设计之初就注意到如何设计以便可以通过整数进行数组索引，可以通过整数索引的就免去了遍历查找！

### 容器选择

**优先选择数组容器，其次考虑哈希表、最后考虑链表**。数组型容器理论上可以解决任何形式的数据类型的存储，他具有极高效率的索引直接访问、添加、弹出以及遍历效率，更具有其他类型容器所无法比拟的最少内存开销——数组容器的最大内存有效利用率可达100%。在著名的VCL库中甚至只有List一种数据容器类，而这种基于数组的容器出色的完成对整个VCL库中所有数据的容纳！ 在早期的C语言中广为使用链表在近些年中出现的高级语言的类库中似乎也已经不见了踪影，而数组容器却被广泛的使用！

数组容器也并非完美的，其最明显的缺点是当从数组容器中移除一个位置的数据会引起较大的内存移动，即要将移除位置之后的所有对象向前移动以填充被移除的空间。内存移动所带来的性能开销是巨大的。但这步足以导致我们放弃对数组容器的使用，同样基于“无法提高运算速度，只能减少运算次数”的优化法则，我们无法加快内存移动速度，只能降低内存移动次数。对于从列表容器中移除操作的优化，将在后续的章节介绍。

### 查找优化

原则上，我们应当尽量将实现方法设计为可以直接通过整数下标在数组中直接索引的，请首先考虑你的方案是否有修改为按下标直接访问的形式，若目前实在难以实现索引案那么将开始考虑使用数组器存储数据，并通过何种查找算法加快查找速度。

使用数组容器时，如果不能通过外部提供的下标直接访问，那么必须为之提供一种快速查找索引的方法，如果不提供任何方法而直接从头到尾遍历查找将造成巨大的性能开销，除非容器中的数据组否少。对容器进行遍历查找的时间消耗将随数据量的增多成同比上升。

比较常用的方法为二分排序查找法（又称折半查找法）[[4]](#footnote-3)，这种排序和查找的方法在数据库领域被广泛使用，而实现起来也相对最为简单。同时二分查找算法是也可以基于多种条件来确定两个对象对比关系的结果，这完全取决于对比哈数内对象对比条件。例如财富排行榜排序的对比函数中，首要对比的条件是金钱值，若金钱值相同则对比人物名称确定排序的顺序。

即使使用各种快速的排序和查找算法，但进行排序和查找计算仍然会产生新的计算，即使没有遍历那么低效，但也是需要性能一些开销的，因此，使用排序和查找也是在是在没有方法通过索引直接定位又不愿意进行从头到尾的遍历时的折中的方法。如果尚有可能，请依然考虑如何通过索引直接存取！

### 删除优化

由于数组容器具有内存连续性，因此当从数组容器中删除一个或多个数据数时会引起内存拷贝操作。删除操作所引起的内存拷贝消耗与数组的长度、删除的索引以及删除的数量有关。当数组的长度越大，删除的起始索引值越小，删除的数量越少则引起的内存拷贝的时间消耗也越大。只有从数组中删除最后的一个对象（弹出）或最后的连续数个对象才能不引起内存移动。

若要优化数组的删除效率，减少因从数组中删除对象而造成的性能开销，则首先也必须依据“无法加快，只能减少”的原则，尽量将删除操作的次数降至最低。再而就是尽量采用删除最后的一个（弹出）或最后连续的多个的方式，避免因删除而引起的内存引动。

减少删除操作次数常见的方法为使用标记删除，即首先使用标记删除的方法将索引位置的数据标记为删除（例如将某个索引位置的数据改写为null）并且递增标记为删除的对象的数量。当标记的数量足够多的时候（例如大于等于10个）的时候，进行一次循环集中整理，将删除位置后面的数据向前移动以便填充被删除的位置，最后直接使用不会引起内存拷贝的删除末尾连续的对象的方法删除空对象。

### 分支优化

从条件分支语句中进行优化的目的在于尽快命中一个可满足的条件从而直接进入分支代码，避免进行更多的判断。这也符合“无法加快，只能减少”的优化原则。

当写下if之后就请开始考虑如何书写你的条件表达式从而可以通过布尔短路[[5]](#footnote-4)法则减少判断次数，直接命中条件。在实际的编码过程中，我们往往能够通过分析条件的逻辑关系，找出那些条件更容易命中，将容易命中的条件表达式写在前面，运算开销更低的条件表达式写在中间，耗时的条件表达式写在后面，即可在满足更优或更快的表达式的情况省去对耗时表达式的计算，从而提升效率。条件书写顺序的原则即是：**任何情况下运算开销小的条件优先写，耗时条件最后；“或”语句将容易命中的优先写，“与”语句中难以命中的优先**。

下面将列举一个使用了“容易命中的表达式优先”的原则优化过的分支代码片段，实现了*在随机输入一个正整数后，值小于3时输出”hello”、 值为从3到8时输出”fine”、 值为9时输出”good”* 的需求。

if (input >= 3 && input <= 8)

printf(“fine”);

else if (input < 3)

printf(“hello”);

else printf(“*good*”);

从示例中可以看出3到8的输入范围大因此更容易被命中，所以我们将这个条件放在最前面，其次就是小于3的条件更容易命中。如果直接按需求文档的要求顺序来书写代码，那么在进行条件判断的顺序上将造成诸多不必要的开销。

下面将列举一份使用了“运算开销小的次之，耗时条件最后的”原则优化过的分支代码片段，实现了 *“在未排序的数组中找到名称为”fasten”，年龄为23的对象”*的需求。

CPerson \*pPerson;

for (i=count()-1; i>=0; --i)

{

pPerson = list[i];

if (pPerson->age == 23 && !strcmp(pPerson->name,“fasten”))

{

return pPerson;

}

}

示例中将运算开销更小的整数值对比条件写在前面，将函数调用这样的耗时表达式写在后面，如果对象的age值为23则不会调用strcmp函数。如果将判断顺序反过来，那么同是实现相同的功能，则前后性能差距将是巨大的。

“或”语句中之所以将容易命中的条件优先写，是因为一旦一个条件成立，那么后面的条件都不会被运算了！同样“与”条件中将难以命中的优先写，是因为一旦一个条件不成立，那么后面的条件都不会被运算了！

//如果a == 1成立，那么后面两个条件不会被运算

if (a == 1 || a == 5 || a == 7)

expr;

//如果b != 7不成立，那么后面两个条件不会被运算

if (a != 7 && a != 5 && a != 1)

expr;

同时对于**相同逻辑条件不同值的分支代码，不要独立开来，应该果断的使用else将他们连接**。以下示范代码阐述了这个原则的重要性。

**原始代码**

if (a == 1)

expr1;

if (a == 2)

expr2;

**优化后**

if (a == 1)

expr1;

else if (a == 2)

expr2;

优化前无论a等于1的条件成立与否，都会进行a是否等于2的判断，而优化后的代码中如果a等于1则不会再进行没有意义的a是否等于2的判断了！

我们常用的大部分语言中都会支持switch-case类型的分支语句。而C++、Delphi等将源代码编译为机器码的语言和编译器环境下，编译器会对各个case分支语句进行优化以便在代码内部建立分支表，从而通过数组下标的从分支表中直接取得分支代码地址并跳转。不知你以前是否考虑过分支语句是如何实现的，是否编译器仅仅帮助我们将case语句翻译为一条一条的if-else而已？下面让我们来揭示在绝大多数的C++以及Object Pascal编译器中是如何实现switch-case的。

**分支语句原形**

switch(n)

{

case 1: expr1; break;

case 3: expr2; break;

case 4: expr3; berak;

case 5: expr4; break;

...

case 100: expr99; break;

default: expr100; break;

}

**编译器会优化为类似如下的代码**

//将连贯的分值的代码建立到分支表中

branch\_table[98] = { expr2, expr3, ..., expr100 };

//优化条件语句以尽快命中条件

if (n >= 3 && n <= 100)

{

//以数组下标形式取得分支表中对应分支只的代码并调用

branch\_table[n-3];

}

//无法建立分支表的代码将展开为if-else语句

else if (n <= 1)

{

if (n < 1)

expr100;

else expr1;

}

因此可以总结出，加快分支命中的核心在于使代码能够被建立在分支表中，而建立分支表的条件则是我们书写的分支值必须是有规律连续递增或递减相同值的！一旦分支表被建立，那么在使用分支表的时候就几乎没有条件判断了，而是直接从分支表中直接调用代码！而我们所要做的就是**在分支语句中将连续规律递增或递减的分支值处理代码顺序书写**，以提供编译器建立分支表的机会。

switch分支语句在AS中被确信为没有进行代码优化，编译器仅仅只是进行了转换为if-else的代码替换操作，这可以通过单步调试AS代码中的switch-case语句观察行的执行顺序即可发现。而switch分支语句在Java中的优化方式目前尚不明确，如果你对此了解请告诉技术总监以便他能够想你虚心求教。

使用数组（表）优化。

### 函数优化

函数优化的核心仍然在于减少函数调用。例如在C++中通过代码内联或将一些简洁的频繁调用的函数书写为宏从而使得预处理器来将宏展开为内联代码。如果我们阅读过AES或Base64算法的C语言实现的源代码，就可以想象如果将那些十分简洁的权值计算书写为函数，那么函数本身实现的运算的代码的开销还原不及对函数调用所产生的开销。

通过对函数的参数进行优化，也可有效的提升函数的调用效率。

首先是从调用约定上优化。一个函数除非会被系统回调或者需要作为导出函数，否则不要为函数增加任何调用约定修饰，否则只会画蛇添足，影响编译器对此函数的调用代码的优化；

然后是减少函数参数的数量。编译器在开启优化的情况下通常会使用三个寄存器来传递参数[[6]](#footnote-5)，因此普通函数的参数控制在三个原子类型参数以内可以有效的避免额外的参数压栈和出栈操作，而类中的非静态函数则由于需要使用一个寄存器传递this指针，所以应当使参数不超过两个;

再者即是尽量使用符合处理器位数的原子类型数据作为函数参数。与我们再“数据结构优化”一节中讲到的思想一致，使用当前处理器位数的数据类型，可以最大化利用数据范围，并且避免传参过程因传递更小的原则类型而需要进行的寄存器高位清除操作。如果你平时细心，会发现libc库中的那些实际只需以字节为操作单位的函数参数，也被声明为了与处理器位数相同的数据类型，例如memset中的填充值参数、strchr中的搜索目标字符参数等，这其中正包含这一优化原则的思想。

最后请注意尤其不能以不带有引用修饰或非指针的类、结构体作为函数参数传递也不要反悔一个这样类型的值，这将引起巨大的性能开销[[7]](#footnote-6)，如果出现一个这样的函数参数，他对性能的破坏力将是你优化再多的地方也可能无法平衡的。

### 减少API调用

不能认为调用系统API与调用自己写的函数一样的高效。事实上绝大多数API调用的开销，相对调用自己的函数或执行自己的代码来说都是低效的。低效的原因并非API的实现效率低，API函数的内部优化肯定是做足的，而效率低的原因在于大部分API实现的功能需要在系统内核中进行，从我们书写的代码执行到系统内核的代码，涉及到用户态的切换，并且在内核中存在着较多的同步操作[[8]](#footnote-7)。

因为诸多逻辑的实现或数据的处理需要调用API，要做到完全不调用API是几乎不可能的，因此我们只能致力于减少调用的次数。对API的调用次数越少，因调用API而造成的开销就越小。

在进行服务端开发中，我们常见的API调用次数优化有：在逻辑循环的一帧中可能需要调用许多次获取时间的API、申请和释放内存的API、发送网络数据以及接收数据的API等等。我们分别使用了静态时间值（每帧更新一次）、内存池以及发送和接收数据缓冲的方式来减少了API的调用，而这些优化也十分明显的提高了程序的效率。

### 指令级优化

如果上述优化仍然无法满足性能需求，那么可以考虑使用指令级优化。指令优化的根本是直接书写最接近机器代码的汇编代码，透过编译器对高级语言的编译直接在CPU上进行运算。这需要一定的汇编基础。

在当下书写PC架构的汇编代码，我们要求必须提供32位版本以及64位版本。两种版本除了在参数压栈的规则上有较大的不同外，其他方面均大相径庭。在此需要更多注意的是指令集的兼容性，对于普通应用我们应当尽量使用兼容性高的指令，而放弃某种特定的处理器所独有的指令集，诸如Intel的SSE指令集或AMD的3DNow!指令集。

## AS代码优化

Action Script是一种解释型脚本语言。在“性能优化通用法”章节中讲到的各种优化原则和方法，同样适用于对AS代码的优化。本章中我们将基于AS语言、flashVM以及flashSDK独有的特性，谈谈如何对日常开发中编写的AS代码进行优化。

### 事件监听优化

本节所讲的内容重在如何避免对象引用关系没有合理解除从而导致对象不会被销毁而引起的内存得不到释放的问题。

事件分派和监听机制是flashVM以及flashSDK中一种十分重要的一部分。对某个对象的某个事件的处理需要首先添加监听，并在合适的时候移除监听。如果没有移除监听将导致这个对象以及监听这个对象事件的对象均不会被移除，而其中的对象所关联的其他对象也随之不会被移除，从而造成内存持续增长。在flash对事件的处理规则中，仅允许在监听对象自己的事件和监听对象自己的子对象（不包括孙对象）的事件时不移除事件监听也能销毁对象。

为了避免因监听的事件忘记移除而导致的对象不会销毁的问题。我们应在开发时注意从如下几个方面进行避免。

1. 如果要监听对象自己或对象的一级子组件的事件，那么可以直接在构造函数里添加监听；
2. 如果要监听多个自己的一级子组件的相同事件且此事件具有冒泡阶段（例如鼠标事件以及大多数的渲染和显示相关的事件），那么就不必要逐个对子组件添加事件监听，而是直接监听对象自己的这一事件并且在事件处理函数中通过事件的target属性判断是那个对象引发的事件从而进行不同的处理；
3. 如果要在显示对象中监听其他对象（非这个显示对象的子对象）的事件，那么最好的方法就是在这个显示对象被添加到显示列表（Event.AddedToStage事件被调度）的事件处理中进行其他对象的事件监听，并且在从显示列表移除（Event.RemovedFromStage事件被调度）的事件处理中移除对其他对象的事件监听；
4. 对于Event.ENTER\_FRAME事件监听，最好的方法是在对象的Event.ADDED\_TO\_STAGE事件处理函数中添加对ENTER\_FRAME事件的监听，并且在对象的Event.REMOVED\_FROM\_STAGE函数中移除对ENTER\_FRAME事件的监听。因为Event.ENTER\_FRAME事件无论显示对象是否可见或者是否添加到了舞台中，这个事件都会被调度，而通常实际的使用需求中当显示对象不显示的时候是可以不调度此事件的。所以为避免不必要的事件调度而需要按此方法来监听和移除ENTER\_FRAME事件。

### 只读数据单例化

在某些类的实现中通常很容易犯下重复构造一个完全相同且不会修改的对象的错误。例如在一个UI库中具有一种窗口类，这个窗口类使用到一些图形资源和用于描述这个图形资源的对象。如果在类的构造函数中具有构造这个图形资源的语句，那么每构造一个这个类，都会引发再次构造一份相同的图形资源。这其实完全是浪费的！这个窗口类所使用的图形资源只需实例化一份即可，以后所有的窗口类示例都使用这个份资源即可，从而大大节约内存开销。

下面将通过实现一个带有背景图片的窗口类的不同方法的代码示范，来说明只读数据单例化的优越性。

**常见的写法**

class Window1

{

//窗口背景位图资源定义

[Embed(source="WndBG1.png")]

static private const WndBGRes: Class;

private var m\_Background: Bitmap;//窗口背景位图

public function Window1()

{

//创建并显示窗口背景位图

m\_Background = new WndBGRes();

addChild(m\_Background);

}

}

**而下面的书写会节省内存开销**

class Window1

{

//窗口背景位图资源定义

[Embed(source="WndBG1.png")]

static private const WndBGRes: Class;

//使用一个静态变量来保存资源数据

static private const WndBRResBMData: BitmapData;

private var m\_Background: Bitmap;//窗口背景位图

public function Window1()

{

//如果从未初始化过资源数据则现在初始，如果初始化过了则不再重复

if (!WndBRResBMData)

{

WndBRResBMData = new WndBGRes().bitmapData;

}

//从静态资源数据创建位图并显示为窗口背景

m\_Background = new Bitmap(WndBRResBMData);

addChild(m\_Background);

}

}

通过以上代码看到对一个类中使用的数据的不同的处理方法，后者比前者更省资源。原因在于构造100万Window1，后者只产生一份窗口背景位图资源的实例，而前者则要产生100万份！因此，本节的优化原则即是：**所有只读的类成员必须设计为静态成员并只实例化一次，所有不会被修改的内嵌图像数据只允许实例化一次**。

### 不使用动态类

在实际的开发中应当尽量减少使用动态类，包括Array。针对这个细节，我们将站在AVM如何查找对象的成员的实现方法的角度进行阐述（这可能需要读者对于AVM以及AVM指令和编译原理有稍微的理解）。

在AVM中对于动态类和非动态类，其性能的显著差别在于类成员的查找方式上。对于非动态类，AS3编译器在编译期间就已经知道了成员的名称，并可以将成员的名称存储到编译后SWF文件的QName表中并确定成员名称的QName整数索引。而FlashVM虚拟机对于这个成员的查找将使用这个成员的QName整数索引减去此类的第一个成员的QName索引，从而得到其在成员表中的下标继而直接取成员。而对于动态类，由于编译器可能无法在编译时明确成员名称（例如表达式o[name]），因此无法确定QName索引，从而导致在查找对象成员的时候只能通过字符串进行，即使使用哈希或折半法进行名称查找优化，性能也远远不及使用QName直接索引。

另外，出于代码的可维护性考虑，也不提倡使用动态类。使用动态类首先无法使用到IDE的代码提示功能；其次在将值保存为动态类对象的成员和从动态类成员中获取成员的过程中，及其容易产生赋值和取值时名称写错的情况，而这一情况是不会引发编译警告或错误的；再次是动态类的逻辑结构不清晰，数据结构的构建可能被分散到多处逻辑处理中，这也很容易造成名称重复而产生冲突。

基于以上弊端分析，**在实际开发中应当尽量杜绝使用动态类以及JSON**,JSON语法实际上也是再构造一个动态类。在本节一开始提及的Array也是动态，同样是不提倡使用的，凡是使用Array的地方请全部考虑使用高效明朗的Vector代替。

### 少用字符串

请参阅[慎用字符串](#_慎用字符串)。

### 定时器优化

实现某些功能往往需要用到定时器的支持。在FlashSDK中提供了Timer类和flash.utils.setInterval函数来直接提供定时调用函数的功能支持。而后者是对前者的一个易用性的封装。但在使用FlashSDK4.5以及更高版本的编译的时候，会产生不推荐使用setInterval函数的警告，这说明对于Timer类和setInterval函数，Adobe官方是不再赞同使用setInterval函数的。

AVM对于定时器的处理相对来讲还是需要很大开销的。尤其当Timer对象的实例很多的情况下，资源开销将成指数增长。因此，在实现相同功能的情况下如何减少Timer对象的数量就成为本节优化介绍的重点。

本节针对定时器(Timer)类的优化原则与“只读数据单例化”原则类似。实现一个类并且其中含有定时调用功能时，大多数人会为这个类增加一个Timer成员并在构造函数或符合条件的逻辑域中构造这个Timer并开始计时。这与“只读数据单例化”中的示例一样，构造100万个这个类同样会产生100万个Timer类。而如果为这个类提供一个静态的Timer对象和一个静态的Vector对象，Vector中存放所有的本类的实例并在Timer的更新事件中遍历这个Vector中的所有实例并调用更新函数，那么即使构造无数个对象也只有一个Timer对象！

使用Event.ENTER\_FRAME、Event.RENDER、Event.EXIT\_FRAME事件也可以达到定时调用的功能，这也是实际开发中较常用的一种功能，使用起来也较为方便。但基于渲染事件的处理的频率精度取决于刷新率以及某帧的事件开销程度。在精度上没有Timer精准，但仍然比为每个对象使用一个Timer进行更新要好得多。

另外，即使是调用周期不同的计时需求，也可以使用同一个Timer来驱动，这也是服务器中所使用的定时器的方法，不仅开销小效率高，其实现也比较简单且。

### 针对AS的局部变量优化

在通用优化法中我们已经介绍了一些局部变量的优化方法。但出于AS语言以及SDK的特性，仍有一些针对性的针对局部变量应该采取的优化方法可以实施。原则是**将被频繁调用的函数中需要构造的局部变量声明为静态似有变量**。在频繁调用的函数中的局部变量需要构造的情况下，可以将局部变量声明为静态似有变量，从而避免每次调用函数都构造一份局部变量对象的实例，从而减少内存占用以及运算量。下面将用一份代码范例来进行对比。

**优化前**

private function foo(disp: DisplayObject; x: Number, y: Number)

{

var matrix: Matrix = new Matrix();

matrix.translate(x, y);

bitmapData.draw(disp, matrix);

}

**优化后**

static private var foo\_matrix: Matrix = new Matrix();

private function foo()

{

foo\_matrix.identity();

foo\_matrix.translate(x, y);

bitmapData.draw(disp, matrix);

}

优化前每调用foo函数一次均会构造一个foo对象，而此对象仅需在此函数中临时使用一下。而优化后无论调用函数foo多少次都只会有一个Matrix对象被实例化。

### 按需初始化

按需初始化对象能够显著的减少资源占用和性能开销。通常一个功能较多的类，其中的部分成员是仅在满足某些条件时才会使用到，甚至是绝大多数情况下都不会使用到这些成员。对这些不是任何时刻均会用到的成员应该采取在第一次使用时候再初始化的方式，而不是在类的构造函数中直接初始化，这样既可避免在成员不使用期间因存在这个成员的实例而产生不必要的开销。

遵循按需初始化成员的原则在UI方面所带来的资源节约和性能优化最为明显。例如游戏中的某些窗口，会第一次需要打开的时候进行创建和初始化，因为有很多不常用的窗口在本次游戏期间可能都不会打开并使用。

由于部分UI窗口中的图像、动画或声音等媒体资源是动态加载的。因此如果能对含有动态加载资源的UI窗口进行按需初始化将避免在进入游戏之初发起太多的资源加载请求从而增加带宽消耗并降低性能。如果UI窗口中含有动态加载的媒体资源且难以对这个UI窗口应用按需初始化，那么无论如何也要将窗口中的各种媒体资源进行按需初始化优化，例如在窗口第一次显示的时候在再初始化这些媒体资源。

### 多用Profiler

FlexBuilder/FlashBuilder中提供了一个功能十分强大的性能分析工具，它可以帮助开发者分析内存的消耗以及函数级的代码性能消耗，是其他语言开发者十分羡慕和渴望的功能之一。通过Profiler可以分析出哪些对象需要进行优化以及那些对象可能存在泄露。因此，每当完成一个系统或功能的开发后，开发者必须在Profiler完整的运行几遍这个系统中的所有功能，以便及时的分析其中是否存在内存泄露或者是否有需要优化调整的地方。

## LUA代码优化

LUA是一种解释型脚本语言。在[性能优化通用法](#_性能优化通用法)章节中讲到的各种优化原则和方法，同样适用于对LUA代码的优化。本章中我们将基于LUA语言独有的特性，谈谈如何对LUA代码进行优化。

### local修饰

在LUA中对一个变量进行local修饰后此变量将被放在所属闭包的局部变量表中。LUA存储有两种存储变量的场合——全局和局部。全局变量存储场合在一个LUA虚拟机中只有一个，每个函数或闭包都具有一个独立的局部变量表。当进行值的读取和写入的时候，LUA首先从局部表中查找此名称的变量，如果找到则对这个变量进行读写，如果没找到则再从全局表中查找。不仅是变量才可以被local修饰，LUA中可以对任何类型使用local修饰，包括函数、表等。由此可以看出：如果想让一个变量或函数被更快的找到，那么就使用local修饰它，从而使得LUA将它放到局部变量表中。以下为用于证实这一原则的示范代码，请分别将代码运行两次，一次不对变量v使用local修饰，另一次对v增加local修饰，然后对比两次运行的输出结果。

v = 0

--记录开始计算的时间按

local ts = os.time()

--循环计算5千万次

for i = 1, 50000000 do

v = v + 1

end

--输出计算消耗的时间

print(“计算耗时：”..os.time() - ts)

需要注意的是，LUA对局部变量表的实现也是有限制的。每个局部变量表最多只能存储250个局部变量。这在普通函数中通常足够使用了。但作为整个脚本顶层闭包来说则可能显得不够用。当出现后者情况的时候LUA会报expression too complex错误；此外，被local修饰后的变量或函数将无法被脚本宿主程序访问。

**只要变量或函数不会被宿主程序使用，那么就对它使用local修饰**。凡是没有进行local修饰的变量或函数，LUA均默认将其放在全局变量中，这将严重损失变量的查找效率！

针对每个局部变量表只能存储250个变量的问题，我们可以考虑在出现这个问题的情况下，使用一个被local修饰的表作为一个新的变量容器，这个表中又可以存储无限个变量了！切记这个方法请在local变量超过250个并且引起报错的情况下再使用，如果没有出现这个情况请不要使用此方法，否则只会降低效率。

### 合理使用表

LUA中的表是非常灵活和便于使用的，是一种非常优秀的设计思想的结晶！同时如果能够合理的掌握表的使用原则，也将对降低内存开销并提升代码性能有莫大的帮助。

请注意LUA中的纯数组表的下标是从1开始的，因此当使用tb[0]或者tb[1.5]的时候相当于使用tb[“0”]和tb[“1.5”]！也就是在使用KV表而不是数组了，只要表的下标不是一个大于0的正整数，那么值的存取方法将被采用使用字符串的Key进行索引！而KV表和数组的效率差异在“容器选择”一节中已经分析过了。

当我们使用表来提供配置数据的时候，不要使用tb = { [1] = 2 }这样的写法，这与前面讲的tb[0]产生的结果一致。tb = { [1] = 2 }和tb[1] = 2的结果是不同的！前者相当于tb[“1”] = 2，而后者则是纯粹的高效数组操作！通常我们书写tb = { [1] = 2 }这样代码的初衷是为了提高可维护性，便于阅读和日后修改。但达到提高维护性目的的方法有很多，例如使用合理的分段和注释，而提升配置性能的方法往往只有这一种！为了便于更好的理解，下面给予一个对比的范例：

不推荐的写法

ConfTable =

{

[0] = {...},

[1] = {...},

[2] = {...},

...,

[10] = {...},

[11] = {...},

}

推荐的写法

通过注释还是可以方便的定位下标的

ConfTable =

{

//0-9

{...},

...,

//10-19

{...},

}

LUA中表的另一大有点在于可以随时向其中补充和删除数据，这是非常灵活的一点。我们随时可以使用tb.key = value的方式对表设置一个值，也可以随时使用tb.key = null的方式移除一个值。结合上一节讲到过LUA中全局变量以及局部变量的性能差异以及局部变量的数量限制，我们可以**将所有不会被通过名称调用的函数或变量保存在同一个表中，从而减少全局变量和局部变量的数量，起到加快变量查找的速度**。接下来我们将通过一个代码示范来阐明什么样的函数是不会被通过名称调用的函数，并且如何实施这一优化方法。

--物品使用回调函数表

local ItemFnTable = {}

--使用物品回调函数

function SysUseItems(actor, itemid)

local ifd = ItemFnTable[itemid]

if ifd then

ifd.fn(actor, itemid)

end

end

--使用Item1的回调处理函数

local function itemFnOfItem1(actor, itemid)

return true

end

--将回调函数设置为使用id为100的物品回调

ItemFnTable[100] = itemFnOfItem1

以上代码中，实现了处理对id为100的物品的使用回调。玩家使用物品时，逻辑服务器会调用这个脚本中的SysUseItems函数并传递相关参数。通过代码我们看到， itemFnOfItem1**函数被关联为了其他表的成员并且只用于处理特定的唯一的事物且不会被通过名称进行调用，这样的函数称为不会被通过名称调用的函数**。由于不会被通过”itemFnOfItem1”这个名称进行调用，所以为此还浪费了一个local修饰，但如果不加local修饰又会使此函数变成全局函数从而增加全局变量的数量导致降低了全局变量和函数的查找速度。当下我们需要一种能够将以上两个问题全部解决的处理方案，而下面的代码示范则演示了解决这个问题的关键：

--物品使用回调函数表

local ItemFnTable = {}

--使用物品回调函数

function SysUseItems(actor, itemid)

local ifd = ItemFnTable[itemid]

if ifd then

ifd.fn(actor, itemid)

end

end

--设置使用id为100的物品回调

ItemFnTable[100] = function (actor, itemid)

return true

end

由修改后的代码可见，我们直接将函数写为表的某个值，从而既不会产生一个局部函数，也不会产生一个全局函数，这个函数只存在ItemFnTable表中。

## 图形优化

读者可能需要具备基本的计算机图形学基础，懂得不同的颜色位数、位图的存储格式等最基础的概念。

### 图形优化原则

同样，我们不能加快图形渲染或呈现的速度，我们只能减少图形呈现或绘制的次数来起到减少时间开销的目的。

### 图像压缩

大部分常见的图像文件格式往往是出于减少图形文件的体积从而减少网络传输消耗以及硬盘存储消耗的目的而设计的。而减少一个文件的大小的通用做法就是对其进行压缩。这也是为什么你会发现使用各种压缩工具压缩某些图形或视频文件后并没有获得太高压缩比的原因，因为这些文件本身已经被压缩过了，很难以再用常规的压缩算法再压缩。

图像或视频的压缩与其他类型文件的压缩不同，其压缩方式通常可分为有损压缩和无损压缩。经有损压缩算法压缩后的图像在还原后相对原始图像会损失一些数据，丢失的数据将导致图像丧失一些细节的表现，而不同的压缩算法和强度带来的图像损失也不同。常见的有损压缩位图文件格式有jpg、png、gif、dds等，而常见的无损压缩文件格式通常有tga、rle等。

压缩后的图像文件或数据均不能直接用于显示或绘制，必须要经过对应的解压缩算法解压后才能用于显示，这就如你将一个电子文档压缩后，需要解压才能使用是一样的道理[[9]](#footnote-8)。因此请不要误以为使用文件大小更小的压缩位图也能够更省内存或显存。图像在内存或显存中最终还是要还原为和位图相同格式的数据才能被硬件使用。

### 位图的内存消耗

在前一节中我们提到过图像压缩的基本知识并说明了压缩可以很好的节约网络传输量和硬盘存储量但并不能节约最终使用时的内存，并且要位图形的压缩和解压额外付出性能成本。因此，本节将重点描述如何计算位图的内存消耗量。

位图的内存消耗与位图的颜色格式以及位图的像素尺寸成正比。在现阶段硬件通常使用32位的位图进行呈现，我们也常使用32位的位图进行绘画。32位位图中每个像素占32个比特位，也就是4个字节。计算一副32位图位图的最小内存字节开销的公式为：4 x 宽 x 高。一张1024x768的32位位图的内存开销为4 \* 1024 \* 768 = 3M。只需333张1024\*768的位图即可消耗掉1G的内存，由此可见位图的内存开销相比其他数据来说是巨大的！而唯一能够减少内存开销的方法就是减少位图数量或者缩小位图的尺寸。

### 动画原理

“动画”顾名思义就是在“动”的“画”。图画本身是不会动的，但将多张不同内容的图画在一个较短的时间连续更换显示，那么这些图像中描绘的对象“看起来”就是“动”的。如果你见过冲洗后的电影胶片或者小时候看过通过一手捏着书脊一手快速翻动从而看到书里的角色在动的效果，那么你将很容易理解这一概念。

因此无论是Flash动画还是电影，都可以称为动画，其本质就是将内容不同的画面按一定时间间隔轮流显示。

以上对于动画的概念的诠释隐含了二维的概念。严格的来讲上面所说的动画特指二维动画，也是是动画的最基本表现形式。同样在现实中普遍存在并易于理解的还有三维动画，三维动画的概念要比二维动画稍微复杂一些，为避免混淆，本文不再做详解，但以目前的计算机图形呈现原理和技术，三维动画最终依然要转换为二维动画来呈现。

### Flash中的图形优化

#### 选择开销最小的方法

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。同样是显示一份内容，不同的实现方法所付出的代价和产生的开销均不一样。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。明确了显示的需求后，首先应该是考虑用什么样的方法更省、更快，避免用牛刀杀鸡。

正如我们目前看到的，大多数的游戏开发团队都不使用mx做UI库，反而大多使用轻量级的第三方ＵＩ库或自己开发ＵＩ库。虽然mx库已经足够完善和易用的，但其中包含了太多臃肿的东西，这些不需要的内容是不可选的，你必须无条件的为这些你所不用使用的特性支付使用的代价——内存开销和CPU占用。

我们可以理解为，显示任何对象的最终方式是使用位图呈现，任何内容丰富的显示对象的最终是要被绘制到位图中进行显示的，虽然你没有亲手做这个操作，但FlashPlayer对图形的呈现确实如此。

#### 使用位图绘制

在显示一份内容或者实现一个组合的显示对象的时候，我们应该尽量考虑能否直接使用位图显示。如果可以使用位图显示，那么显示的开销将会降到最低，性能将被提到最高！例如，很多进行了数年的AS开发的人，也只懂得用TextField来显示文字，“凡是一个地方需要显示文字，那么马上使用TextField”这就是他们的惯性思维并且从来不会对此产生怀疑。TextField不仅提供了文本显示的功能，他还含有富文本处理、文字输入等高级的功能，还还有对各种鼠标键盘的事件处理……这些功能全是不可选的，哪怕只是想显示一个字母也要支付对其他功能的开销。针对刚才讲到的“显示任何对象的最终方式是使用位图呈现”，我们可以直接使用位图来显示文本，从而将开销降到最低并使将性能最大化。方法就是使用一个静态的TextField用于临时显示文字，再将此TextField通过BitmapData.draw方法画到一幅位图中！所以在实际的开发中，我们一直都要求**在Flash中使用位图来显示文字，而不是TextField**。在开发中可以使用CommonUI.baseUI.text.TextFactory类将文本绘制成位图来显示。

如果使用矢量图，那么同样请先考虑能否使用位图代替，因为矢量图也是要绘制到位图中的，如果直接使用位图就一步到位了。如果没法用位图代替，那么在矢量图内容更新不频繁的情况下，将cacheAsBitmap属性设置为true将强制FlashPlayer对此矢量图使用位图缓存。

如果一份显示内容的更新比较频繁，则不应当对其使用位图绘制优化法。使用位图显示内容可以获得很好的呈现效率，但如果显示的内容需要变化的时候需要重新绘制位图，这将产生很大的性能开销。如果显示内容更新频率不频繁，那么我们可以放心的使用位图绘制法呈现内容，尤其是内容一旦确定就不再更改的情况。但如果内容更新频繁，那么我们就应该慎重的选择是否仍然再适合使用此方法。例如，当实现下雨、下雪等内容更新频繁的画面时，我们就不应该使用位图绘制法，那只能为了支付画面更新而带来的高昂的时间消耗而降低性能！

另外，当在项目中大量的使用位图绘制的方式来呈现内容的时候，请务必认真注意由此而带来的内存开销情况。因为每个位图数据都需要独立的消耗一份内存的！例如在CommonUI中实现的窗口类中，窗口的背景是用的是九宫格拼接的方式实现的。九宫格拼接的呈现支持使用位图绘制的方式一次性显示，也支持使用九张位图排列并拉伸显示的方式来显示背景。这就要开发者在具体使用中，视项目中的窗口数量而决定使用何种方式来显示。这也是优化原则中“空间换时间”的又一选择性体现。

可以使用位图呈现的方式进行显示优化的方面还有很多，这需要在实际的开发过程中进行总结并选择开销最小的方法！同样，我们前面介绍了动画的实现原理，那么如果我们只是想要实现动画的播放，并不一定要选择使用MovieClip或SWF，因为它们的开销还是颇大的，基于动画原理我们可以很用以的以最小开销的位图来实现一个动画播放的组件。例如wylib.Packs.BitmapPack.MDPAnimation。

#### 减少显示数量

使用开销小、性能优的显示方法可以降低显示成本，同时还要注意如何减少显示数量。这其实是不容忽视的一个很大的方面，在实际开发中会产生很多“可以不要”的或者可被“合并优化”的显示对象。这一原则具有几个常见的规则可以遵守。

**将不连贯的文本连贯显示**。例如要显示文本“点击可领取经验奖励，今日还可领取**4**次”。其中次数（4）是可变的。部分开发者在做的时候可能会采取使用3个位图或者TextField来显示这串文字，其中“4”前面的作为一份文字显示，“4”作为一份显示，“4”后面的又单独显示。这样当剩余次数改变的时候只需修改第二个显示对象的内容，确实会方便一些，但为此付出的是要使用3个显示对象来显示这串文字！在实际的开发中不允许这样做，可取的方法是使用一份显示对象，在次数变更的时候把正行内容更新一次，从而从量上对显示的性能进行优化。

**减少滤镜的使用。**为显示对象使用滤镜会很大程度的降低显示的性能。使用的滤镜越多、滤镜的强度和精细度越高所要突出的性能成本也越高。按照我们实际进行开发的实际经验来统计，使用最多的滤镜是对文本应用外发光滤镜以呈现文本描边的效果，继而增加文本在各种背景下的可读性。而无论是使用位图还是TextField来呈现文本，要为他们都增加一个外发光滤镜，所带来的性能开销还是颇大的。因此，当需要为文本增加滤镜的时候，应该在将文本绘制到位图前，指定TextField的滤镜，并将应用了滤镜的TextField连同滤镜一起绘制到位图中，从而不用再额外使用滤镜。在TextFactory中提供的各种绘制文本到位图的函数中均提供的滤镜参数，支持将滤镜绘制到结果位图中从而优化显示性能。

**移除对象而不是隐藏对象。**当一个显示对象不再需要被显示的时候，请将它从parent中移除，而不是使用visible将他隐藏。隐藏只是导致对象不可见并且不可交互而已，其各种渲染调度事件仍然会被调度，如果对其相关渲染事件添加了事件监听，则虽然对象看不见，但这些事件监听函数仍然被调度，而通常在实现中是不需要被调度的。

#### 减少资源开销

存在较多不同的资源实例或者相同的资源被实例化太多次，都将影响显示性能并增加内存开销。尤其是相同资源被多次实例化的情况，在实际开发中必须杜绝。我们在AS代码优化一章中的“只读数据单例化”中已经介绍过相同资源只实例化一次的意义。所以再开发中我们**必须保证相同的资源对象只被实例化一次**，保证其实例不超过一份！

对于不同的图像、动画、声音等资源，我们必须**在实现需求的前提下，以能少则少、能小则小的原则使用资源**。首先考虑实现这个功能是否能用公共资源，或者是否能够使用之前已经用过的存在的资源；再次要从自身避免从将相同的资源文件按复制多次并嵌入为不同的资源对象；其次针对其他部门提交的媒体资源，看看大小是否合适，能否再小一些，如果还有减少大小的余地，则必须要求资源提供者对资源进行优化。

请注意，一张图片被嵌入到SWF中成为资源后，这个份资源的大小不一定和我们在系统的资源管理器中看到的文件大小一致。尤其是使用png文件时，由于其具有很高的压缩比，而嵌入SWF后会被转换为位图，从而在资源管理器中看起来只有1K的图像文件，嵌入后会远大于1K！**如果一个图像作为背景等不需要不规则边缘时，那么使用JPG格式，仅当图像需要呈现不规则边缘的时候，才使用PNG格式**，并且使用的ＰＮＧ格式要要求美术人员尽量降低颜色位数。

## 数据库优化

本章所指的数据库是狭义上的支持结构化查询语言（Structured Query Language）的数据库。特指部分MySQL（侧重）、SQLServer以及Access数据库的部分功能，因为作者的数据库经验仅限于此，并且也认为对以上数据库功能的使用只限于解决现实问题中所接触到的部分，而没有进行过深入的涉猎。如果对文中所提及的观点或方法有异议请及时联系技术总监以便研究并更正。

### 数据库优化重点

按以往游戏开发所采用的数据存取方式，游戏数据库通常更注重写入的效率。优化核心在于优化写入查询的速度并减少查询次数。insert、update、delete、alter等会产生数据改变的操作统称写操作，所有的写操作几乎都会涉及到对表的锁定，这是为了保证数据一致性，在一个写操作语句在未处理完成之前，所有的与写操作相关的表的select、show等读操作都要被挂起，直到此前与表相关的写操作全部完成。因此提升写操作的效率即是提高数据库整体性能的关键所在。而影响写操作产生的消耗的因素则与表中的数据量、更新字段的数量、语句条件复杂度以及写操作发起的次数有关。针对影响数据写操作消耗的因素的优化就是对数据库整体性能优化的原则。

### 表和字段优化

通过减少表之间的关联性，可以提高数据查询的效率。这就要求尽可能的减少实现一个功能所需要的表的数量。当大多数需要查询或更新的数据在一个表中时，即可不必要进行多个表的主键或索引关联，并且不用从多个物理文件中读取数据。

如果一个表的数据量足够大，那么将数据分散到多个结构相同但名称不同的表中能够有效的减小查询以及更新的效率。例如在日志服务器的数据库中，每天都会为每个逻辑服务器创建一个日志表，从而将海量的游戏操作日志分散了许多个表中。

在设计表的时候，应当尽量避免使用太小的数据类型做独立的字段。例如不使用tinyint、byte、short等过小的数据类型，而是将几个小的字段组合为一个大的字段。例如可以将4个tinyint组合为一个int字段，在查询和更新时可以使用位操作选择性的选取其中的部分。这种方式可以减少表中的字段数量，从而提高查询和更新的效率。

合理的对表使用主键和索引是表优化的核心所在！一个表存在索引和不存在索引的查询效率是远远不同的。合理的索引能够提高整个数据库的操作效率。在索引设计中，索引字段应挑选重复值较少的字段；在对建有复合索引的字段进行检索时，应注意按照复合索引字段建立的顺序进行。

### 优化查询语句

对SQL语句进行优化的原则适合使用性能优化通用法中的[分支优化](#_分支优化)方法。核心思想在于加快查询语句的命中速度。在实际的数据库使用经验中我们总结出，在完成一条带有条件的更新和查询语句时，大部分时间花在了条件匹配上。表中的数据量越大，在进行数据匹配上所花费的时间也越长。如果查询的条件无法省略，那必须加快条件的命中速度！除了适用[分支优化](#_分支优化)原则中的方法外，在优化SQL语句中的条件语句时还应当优先使用主键或索引的字段作为条件，并且将这样的字段条件书写到条件按表达式的前面来。

### 优化删除

根据我们在数个项目中的数据库使用经验来看，作为游戏数据库，部分表的数据量会很大，且伴随有频繁的删除-插入操作。而删除操作通常是所有常见的写操作中开销最大的操作。因此，依据“无法加快，只能减少”的优化原则必须使用行之有效的方法来减少删除的次数从而减少因删除造成的开销。最终采用的是将需要删除的数据的索引字段update为0，每1分钟删除一次表中所有索引为0的数据。由于update语句的开销比delete小得多，所以优化后性能得到明显的提升。

## 内存操作优化

程序的性能优化不仅体现在代码的优化上，同样在内存管理的方法和思想样也将不同程度的影响程序的逻辑处理开销，从而对程序的运算效率产生影响。在此，我们将内存视为逻辑处理中的“数据”，而对从内存方面所作的优化的原理视为是对数据的输入、输入以及运算方式的改善。

### 理解内存碎片

首先我们需要简单的阐述一下“内存块”的概念。在保护模式的操作系统中，进程具有独立连贯的虚拟内存空间，在32位下最多可以使用4G，而在64位下则要大得多。我们每次产生一个对象或者申请一份内存，都将从这连续的内存中“挖”一块出来，用于存储这个对象或提供内存的空间。“挖”出来的这个块内存成为一个内存块，内存块的数量随着我们申请内存的次数越来越多。由于虚拟内存是连续的，所以任何一块内存的地址都是不可改变的，凑则就会与其他地址重复。所以，当我们释放了内存块后它将被放回虚拟内存的“原处”。

由于操作系统提供的默认的内存管理器需要对虚拟内存进行管理，当内存释放时，内存管理器会尝试合并连续的已经释放的内存块，从而获得一个更大的连续的空闲内存块。同时，在申请内存时，如果最后一个空闲块的大小不足，内存管理器会也会遍历所有的空闲内存块以试图找到一个空间足够的空闲的块并进行分配。因此，太多的内存块将会形成内存碎片，影响释放内存时内存块的合并速度以及申请内存时大内存块的查找速度，从而增加运算开销，降低程序效率。尤其在进行服务端方面的开发时，如果不采取一些内存优化措施，将会产生极多的内存块，形成严重的内存碎片，从而每当申请内存以及释放内存的操作发生时，都会造成太多的时间开销。

### 减少申请和释放

针对内存的优化原则，核心上与对代码的优化原则相同。同样是无法加快内存操作的速度，而是只能减少操作的次数，从而避免不必要的时间开销。

减少内存块的根本在于减少内存申请和释放的次数，这通常要从设计时开始做起。

为类添加一个结构化成员且这个成员需要和类一起初始化时，应该尽量添加为对象成员，而不是指针成员。这样在类的内存被分配或保留的时候会将此成员所需的内存一同分配或保留。如果将成员设计为指针类型，那么必须亲自去实例化这份指针，将造成额外的一次内存申请和释放。当这个类被构造一万次，那么也省去了一万次内存的申请和释放。

当一种数据结构在运行期间会产生很多份，并且具有很高频率的申请和释放操作，那么应该考虑使用对象池的技术来优化申请和释放。对象池的核心在于一次性申请一批数量的对象，从而将零散的对象申请与释放的操作优化为从池中直接取空闲对象，当对象销毁时也只是将对象放回池中。申请和释放操作的过程就不设计到具体的内存申请和释放，从而减少操作的开销。

养成一种习惯，当写下new或malloc的时候考虑下这个申请操作能否进行优化。

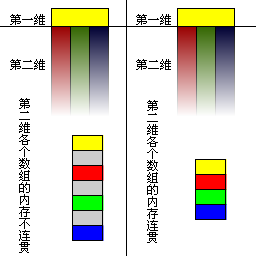
### 使用连续的内存

使用连续的内存可以显著的加快内存的读取和写入的速度。在硬件方面，CPU通常至少具有两个缓存，一级缓存和二级缓存。一级缓存包含一级是指令缓存以及一级数据缓存，用于缓存目前执行位置的代码以及其前后位置的代码以及代码涉及的数据地址（以Intel P4处理器为例）。二级缓存通常作为一级缓存的缓存。CPU在读写内存前往往要从指定内存对应的物理内存地址将数据装入缓存中。如果读写的内存是连贯的内存，那么可以减少CPU更新缓存内容的次数，从而加快处理效率。这其实也是我们在“性能优化通用法”一章中讲到的[局部变量优化](#_局部变量优化)原则的根本依据。如果读写的内存不连续，那么将导致CPU不断的从不同的物理地址载入缓存并将旧的缓存写回物理内存。

在前一节中讲到的方法中，将类成员作为对象成员使用而不是指针成员的方法，以及对象池的方法，也起到了很好的增加内存连续性的作用。

使用多维数组时要保持他们在同一块内存中。例如当要实现一个长度不变的二维数组时，我们不应该为每一个第一维的成员指针申请一块第二维数据的内存！这样会不仅会增加内存碎片，还会造成内存的不连贯性。可以采用使用一整块的内存，并划其中的各个段位第二维数据的方式来实现，不仅减少了申请和释放内存的开销，还能使得内存更为连贯。

下图阐述了两种不同的的内存结构的差别。



并不是只有在数据结构的设计上才能够发挥“使用连续内存”的优化措施。在具体的函数内的代码编码方法中也存在内存连续使用的优化。具体的表现通常为：当函数中存在针对不同对象的成员存取时，对不同成员的存取顺序的连贯性。通常要求我们**在两个连续的对象成员存取表达式之间不要加插其他对象的成员存取**。将相同或者具有地址关联性的对象存取操作的代码写在一起。下面将通过一个具体的实例来阐明这一原则。

**不连贯的内存存取**

void foo(CA \*pa, CB \*pb)

{

pa->m1 = 10;

pb->m1 = 5;

pa->m2 = 3;

pb->m2 = 6;

}

**连贯的内存存取**

void foo(CA \*pa, CB \*pb)

{

pa->m1 = 10;

pa->m2 = 3;

pb->m1 = 5;

pb->m2 = 6;

}

优化后的代码仅是调整了两个赋值语句的顺序，首先要保证的是语句顺序的调整不会对逻辑运算的结果产生影响。优化前由于第二条语句要存取pb的内存，CPU在执行完第一条语句后缓存中存储的时候pa的内存，pb可能不在缓存中，因此CPU此时将要进行缓存更新。而当调整语句顺序后，使得pa以及pb的成员存取集中了，CPU在处理完对pa->m1的赋值后不用更新缓存，因为缓存中当前已经存储者pa的内存了！

### 减少内存拷贝

我们应当从设计之初就考虑到如何最大化的避免内存拷贝操作。过多的内存拷贝会引起额外的开销。需要注意的是，不仅仅是显示的调用内存拷贝函数才会引起内存的拷贝，大部分字符串的操作也会引发内存拷贝，例如字符串赋值[[10]](#footnote-9)、字符串拼接、字符串格式化等。这也是之前的章节中要求慎用字符串的原因之一。

我们在进行高效率的网络通信编程中，通常会用到拼包和拆包的技术。对于数据包的拼接和拆解，是通过内存缓冲区进行的。但在实现内存缓冲区的时候我们应该注意从缓冲区中提取数据后，应当将读取指针调整为新的位置，而不能缓冲区中剩余的数据向前拷贝。否则将可能产生滚雪球效应，导致严重的性能瓶颈。

### 使用优秀的内存管理器

在内存管理方面有很多优秀的分配和释放的算法。有些算法适用于不同类型的内存使用需求。但大多数都致力于优化小块内存的申请和释放，并使用池或缓存的技术加以优化，从而减少内存碎片数量，提高申请和释放的效率。因此，选择一个优秀的内存管理器将对内存操作的优化起到十分重要的意义。

在C++没有统一和广泛使用的内存管理器标准，针对过去实际开发中所总结的经验中，我们提供了CBufferAllocator、CObjectAllocator以及CSingleObjectAllocator类提供内存管理的优化。CBufferAllocator的实现原理在于将释放的内存进行缓存，并在申请内存的时候检查缓存中是否存在合适的内存块，如果存在则直接使用，不存在时再从系统中申请。而CSingleObjectAllocator和CObjectAllocator是两个不同需求的对象池的实现，原理是通过大块内存池的方式提供连续的小块内存分配。这些简易的内存管理器只是实现了最基本和简单的内存优化功能，整体功能并不非常完善，但大多数情况下足以满足日常开发的内存优化需求。

在Delphi中存在着许多的十分优秀的内存管理器。例如FastMM以及Delphi原生的BorlandMM。这些内存管理器不仅提供了十分优秀的内存操作的优化，同时还提供了内存越界、内存泄露等内存错误的检查和定位。所以在Delphi中进行开发的时候，及时没有亲自做内存操作上的优化，仍不会对内存操作的性能造成影响的原因就是得益于其含有优秀的内存管理器。

直接使用Delphi的内存管理器，编译一个动态库提供给C++使用也是一种不错的选择。但目前已知的所有Object Pascal编译器都只有32位的稳定版本，内存管理器也缺乏相应的64位兼容版本，所以如果使用这种方法来为C++提供内存管理器，那么仍只能使用32位的版本。

### 比特位代替布尔

一个布尔实例的大小为1字节，我们使用其来描述一个真假值时实际上值用到其中的1个比特位，其他的7个比特位都是浪费的。如果使用比特位为单位作为真假数据存储，那么同样是1个字节，却可以存储8个真假值，而且没有在运算时额外带来开销，何乐而不为呢？

使用比特位代替布尔类型在编译为机器语言的开发语言中是不错的内存优化方案，在常见的PC处理器、移动处理器中，CPU的运算都是以比特位为单位的，因此使用比特位运算不会有额外的性能开开销，反而是恰如其分的使用处理器预算。尤其是在服务器的开发中，将一个类中的32个布尔成员优化为1个uint成员可以节省28字节，而当对象数量多的情况下，省下的内存空间将越发明显！此方案不适合在Java、AS3、LUA等需要虚拟机的解释型语言或者不支持位运算的语言中使用。

## 数据结构设计

我们从数据结构的设计用途上将数据结构分为三种类型：逻辑运算型、数据存储型和通信型。针对不同具体用途的数据结构将从不同的出发点为设计指导依据。这部分的内容更适合使用C++、Delphi等能够编译为机器码的语言。

无论是出于哪种用途或者目的而设计的数据结构，我们有必要先了解结构的内存对齐的概念。在默认的情况下，VC、GCC、C++ Builder以及Delphi在结构的内存对齐的规则为：

1. 原子成员对齐到成员字节大小的边界上；
2. 结构成员对齐到结构中字节数最大的原子成员的边界上；
3. 结构的大小被补齐到结构中字节数最大的原子成员的边界上；

struct

{

char m1;

//这里会被编译器插入1个字节（遵循规则1）

short m2;

//这里会被编译器插入4个字节（遵循规则2）

struct

{

double m1;

double m2;

};

int m3;

//这里会被编译器插入4个字节（遵循规则3）

};

我们以可直接寻址和不可直接寻址来区分一个内存地址的寻址有效性。可以简单的理解为：当一个内存地址可以被CPU的位数字节大小整除时，这个地址就是可以直接寻址的。例如32位的处理器中，位数的字节大小为4，那么如果一个内存地址可以被4整除，则可以被CPU直接寻址。而当一个内存地址不可被直接寻址时，CPU可能首先读取该地址所处的第一个可直接寻址的地址的值，并舍去不属于这个地址的那些字节位，然后从后读取该地址所处的第二个可直接寻址的地址的值，并舍去不属于这个地址的那些字节位并与第一次读取的值组合，才能读取出这个地址的值。



在上图中，CPU是32位的，黄色的每个块表示一个可被CPU直接访问的内存地址，因为他们可以被4整除。红色的块表示一个int值的地址，他的地址为5，不能被4整除，也就无法被CPU直接寻址。而CPU为了读取这个值所要做的，首先是读取4到8之间的（第二个）内存单元并丢弃值中的第一字节，然后读取8到12之间的（第三个）内存单元并丢掉后3个字节，最后将两次读取的值组合，取得了我们所要读取的值。

有些硬件甚至要求要存取的内存地址必须是处理器位数的带宽倍率上。因此编译器出于对各种处理器的兼容度以及改善内存存取效率的目的，会在红色的块之前填补3个字节，使得其对齐在了一个独立的内存单元中可以被直接寻址，从而不丧失内存存取原本性能。

### 逻辑型数据结构

逻辑型数据结构指设计用来进行数据运算和逻辑处理的结构或类。在设计这些成员的时候往往更注重数据结构能够发挥很好的效率并且不额外的占用内存。因此，在进行逻辑数据结构的设计时，重点应该遵循在“性能优化通用法”中介绍的[数据结构优化](#_数据结构优化)法。

此外，由于编译器会对数据结构进行内存对齐，为能够做到内存空间的最大化利用，**在设计逻辑型数据结构时，应当自行书写代码完成结构成员的内存补齐**，从而可以在以后想要增加更小的成员的时候，从现有的保留字节直接调整为需要的成员，从而不引起新的内存开销。例如有类：class C1{ char m1; int m2; }，良好的做法是在成员m1和m2之间增加成员char reserved[3];来填充即将被编译器补齐的内存空间，而日后如果我们需要给C1类增加一个小于3字节的成员时，就可以直接修改reserved数组的大小，腾出新的空间来容纳新的成员。

### 存储型数据结构

以节省为主，合理选择数据类型。注意预留空间供日后使用。

### 通信型数据结构

结构成员内存字节对齐、内存补齐问题，合理选择数据类型。

## 开发经验

### 参数类型

书写一个函数的参数时，我们应该对参数的类型选择“合适”的类型。对参数使用合适的类型可以增加代码的可维护性，同时使得函数更加“易用”。

例如，当实现一个函数将一个可显示的对象绘制到一个位图中并返回此位图数据，那么这个函数传递用于绘制的对象的参数类型应该使用DisplayObject，而不应该是某种具体的显示类型。如果使用参数类型为Sprite，那么将无法传递Bitmap对象，也无法传递TextField进行绘制，唯一的方法就是找到所有支持绘制的类型的共有的基类。

这一原则同样适用于类成员的类型选择上。例如，我们实现的一个窗口类中含有一个按钮，那么我们应该尽量将此按钮成员的类型书写为CustomButton，而不是具体的实现了不同按钮皮肤的Button1或Button2类，因为这样我们可以在初始化这个按钮成员的时候将此按钮从任意实现了皮肤的按钮类型中创建，并且可以随时修改按钮成员实例化的具体类型而不用修改成员声明中的类型，这带来了更良好的代码可维护性。

另一方面，当我们明确一个函数的某个参数，只能接受某一层级的类型以及派生的类型时，我们就不能将参数的数据类型书写为更原始的基类。例如，在服务器中我们提供的一个将玩家踢下线的函数中，玩家的参数的类型就应当是一个玩家类，而不能是其基类，因为基类可能不具有网络连接属性，从而无法关闭玩家的连接。

**使用可以实现所需功能的最底层的类型来作为类成员或函数参数的具体类型**。可以提高代码的可维护性，并增加代码的易用性。

### 配置文件

在进行服务端、工具开发时，我们提倡从配置文件开始入手。这可能和部分开发指导中提及的先从数据结构开始有所不同。但据实际的工作经验来看，从配置文件的设计开始入手，可以具有更多的优势。当配置文件设计完成后，该系统的框架结构和数据结构也清晰可见了，甚至通过配置文件即可直接书写出数据结构了！

设计配置文件的格式时，在考虑如何实现的同时还应充分站在最终使用者的角度考虑问题，并且注意以下原则：

**易用性**，在设计配置文件的结构时，如何调整配置文件的设计、如何安排各个数据项的默认值、在配置中的顺序以方便最终编辑配置文件的人，提高他们编辑配置文件的效率。

**简单性**，做到更简单、更容易上手以便降低理解和学习的成本。

做到了易用性和简单性，则可以从从学习和使用中节省整个开发团队的时间，提高工作效率。

大多数情况下，我们提供手动配置的配置文件的初衷在于让他人进行数据的录入和配置工作。那么易用性和简单性将直接影响配置或数据录入的效率，甚至影响到配置数据录入者出错的概率。

### 数据与代码分离

由于在实际的开发过程中，一份设计需求的最终验收需要经过不断的逻辑规则和配置参数的细微调整，每当设计人员发现一些细节需要调整时都要与技术人员沟通并实施调整，因此在这个过程中往往会付出大量的沟通成本和修改成本。为了加快开发效率，我们将从技术人员实现设计需求的环节进行规范，从而降低修改的沟通成本。

通常在实现一种功能的时候，往往可以从需求中分离出配置数据和逻辑代码。配置数据与代码分离后，可以方便的实施**技术人员负责逻辑，设计人员负责配置**的工作方式，从而减少在配置修改上所花费的时间和成本。

因此，在实际开发中，应当极力避免在代码中写进行逻辑运算和处理所需要的实际数据。遵循**代码只实现逻辑，数据由外部提供**的开发和设计原则。将被分离出的数据配置应当独立成为配置文件，并实现对配置的读取和使用。

当使用LUA开发时，同样需要遵循数据与代码分离的原则。但出于LUA的特性和通常实际需求的实现的灵活性，允许在LUA数据配置中包含少数函数成员以进行特例化的条件检查和操作。这种特例在任务系统的配置中被较多的使用。

### 时间值的使用

1. 使用下次时间；
2. 先修改时间，再进行操作，防止异常发生导致无法修改时间并持续操作。

### 使用轴时间

动画、进度等等……

### 默认值的选择

### 惰性原则

### 先取后给

先设标记后发奖励。

### 允许的条件

在进行设计和开发的时候，如果一个功能的处理规则涉及到对数据的筛选条件，那么应当尽量在条件规则的阐述语句中使用“哪些条件允许”的表达方法， 而不要使用“哪些条件不允许”的表达方法。因为通常在设计一个规则的时候，能想到的允许的条件是有限的，而不允许的条件往往是无限的。为了避免因提供的规则中有遗漏，也应当列举这些有限的“允许的条件”。

在一次开发中，测试人员发现了游戏中的宠物药品无法放入到快捷栏中，于是立刻要求程序进行了修改。通过这个例子我们可以看到，如果开发人员新加的一个物品不是可被放到快捷键中的物品，那么他能够想到把他放入到快捷栏中吗？这需要一定的逆向思维才能做到，通常是很困难的！

从可维护性的角度考虑，提供“允许的条件”往往比提供“不允许的条件”具有更高的可维护性。在基于原有的规则添加新的数据后，开发人员往往会立刻测试新添加的数据是否可再原规则上使用。如果不可被使用，那么会立刻被发现并作为新的“允许的条件”添加。而当添加不允许操作的数据时，若要发现问题必须以逆向思维的角度站在相反的立场来进行测试，而这通常是你我都难以做到的，所以相比下来，提供允许的条件具有很高的可维护性。

### 重复检查条件

### 更安全的加锁

在多线程编程中，加锁是在多个线程之间传递和同步数据的常用方式。而因加锁、解锁的操作却带来了诸多性能以及死锁的问题。同时，含有锁操作的代码也是日后修改中最容易出问题的代码。

具有锁的代码的维护隐患在于，在函数开始之处进行加锁，并在函数返回前解锁，则当函数足够复杂，且逻辑层次较深时，一旦忘记解锁就使用return，则会造成其他使用此锁的线程被死锁！当为了修改某个已实现功能的细节而修改这个函数并因为这个问题而造成了死锁，在测试时往往也比较满意被发现。因为针对小的细节的修改，开发者以及测试人员往往只关注这个细节有没有生效，而忽略了为了修改这个细节而可能产生的问题。而BUG就此埋下。

针对此问题的一种行之有效的解决防范是在使用锁的函数中使用安全自解锁(wylib::sync::lock::CSafeLock)，通过类析构的方式主动进行解锁。编码时不必书写加锁和解锁的代码，只需用CSafeLock声明一个局部成员，传递需要加、解的锁对象指针即可。这样可以避免日后修改这份代码时可能产生的问题。

加锁请注意尽最大可能减少因当前线程获得了锁而对其他线程的影响。最好的做法是在存取需要进行独占存取的操作的代码前加锁，并在存取数据后立刻解锁，而不要在加锁后做一些与锁起到的保护无关的操作。此外，如果有循环，请将加锁操作和解锁操作放到循环外，并优化循环内的代码，减少与锁保护数据无关的代码。

推荐

void foo()

{

...;//与锁保护数据无关的代码

{

CSafeLock(&m\_Lock);

readDataExpr();

}

...;//与锁保护数据无关的代码

}

不推荐

void foo()

{

CSafeLock(&m\_Lock);

...;//与锁保护数据无关的代码

readDataExpr();

...;//与锁保护数据无关的代码

}

加、解锁的操作是由操作系统内核进行的，因此效率也会比较低，所以尽可能的降低加解锁的频率，可以提高性能。

### 异常处理

### 同帧共时

在我们目前的数款服务端以及客户端中均有使用“同帧共时”优化方案。方案假设在一个逻辑处理帧中，一帧的时间足够短，短到所使用的最小时间精度（一毫秒）以内，那么在帧的开始时刻我们调用获取当前系统时间的API函数，并保存这个值为全局的静态值，帧的所有展开调用的函数中，涉及到获取当前时间的操作全部直接读取这个值[[11]](#footnote-10)，而不再调用获取时间值的函数，从而减少了获取当前时间的API函数调用次数。

这个方案通过在多款产品的实际使用，尚未表现出因在一帧或多帧中的时间误差而产生明显的问题，因此这一方案在大型项目系统中值得推广使用。

### 频率控制

### 数据库对象命名

每一个数据库对象，包括数据库、表、字段、存储过程等都需要一个名称。我们必须明确，由于MySQL数据库对名称大小写的敏感性取决于操作系统是否对文件的名称具有大小写的敏感。这因为MySQL中的数据库和表都是在目标操作系统中作为文件存储的，文件的名称与数据库或表在创建时提供的名称完全相同。在Windows操作系统中，文件的名称是不区分大小写的，而在Unix、Linux等操作系统中文件的名称是严格区分大小写的，Abc和abc是分别指代不同的文件的。所以再这种情况下，MySQL对数据库和表名称的匹配也是大小写敏感的。

为了使得我们的SQL语句无论在哪种操作系统的MySQL中，都具有良好的兼容性，我们要求**所有的数据库对象的名称必须使用小写字母**，包括数据库名称、表名称、字段名称以及存储过程的名称。

虽然可以通过严格的进行大小写匹配从而在名称中使用大写字母，以使得对象的名称看起来更工整、清晰和易读，但我们很难保证开发团队中的所有人对名称的命名都具有相同的思维习惯，也难以保证团队的所有人都可以在任何时候严格的遵循名称的大小写匹配！除此之外，大小写不匹配而能够引发的错误要在非Windows系统下的MySQL中才生发生，而我们通常的开发环境都是Windows的，这意味着我们在开发和测试期间难以察觉这一问题，而最终到了向正式的Linux服务器中的MySQL应用的时候，错误才会呈现出来！相比之下，让团队中的每个人遵循”所有的数据库对象的名称必须使用小写字母”的原则更容易遵守并且没有潜在隐患。

### 创建表语句

**所有涉及到创建表的SQL语句中，禁止书写丢弃表的语句！**对于许多缺乏经验的开发人员来说，大部分人可能会有一种不经意习惯，那就是先丢弃现有表，再创建新表的方式来书写一个创建表的SQL脚本，这种情况是可以十分完美的解决表存在的情况下无法创建新表的问题，既简单又方便！但这是一种十分不良的习惯，更可能是噩梦的开始。

请设想一下，如果某一天程序或运维人员不小心错误的在一个已经部署好并运行许久服务器中又执行了一遍数据库初始化的脚本，那么脚本中的drop table语句将会把表连同表中的数据一起丢弃，这一错误可能要导致这个服务器需要从备份的数据中进行回档，并要从数据库的日志中恢复数据，从而引起诸多不必要的连锁反应！而这一且都可以从初始化数据中的SQL文件中的这条drop语句中开始避免！

如果你很细心的话，可能会发现，无论是SQLServer、MySQL或者Oracle数据库的相关工具，都具有导出初始化数据库的SQL脚本的功能，而无一例外的是这些工具导出的初始化数据库的脚本中的创建表的语句都不包含drop table语句，而都是以create table *表名* if not exists这样的形式生成创建表的语句！这也可以充分说明使用什么样的语句创建表更安全，更能减少隐患。

### 使用限制语句

**凡对数据库的更新、删除操作，请考虑加入limit语句进行限制**。这是一种保障数据安全，并将可能发生的问题的影响降低至最小的良好习惯。当书写update或delete语句的时候，如果语句足够复杂或者在我们书写语句时出现各种粗心问题，都有可能导致我们写错甚至干脆忘记写下语句中本应存在的where条件部分，这将导致那些本不应该给更新的数据被更新，或者本不应该被删除的数据被删除了！谁能保证这种事情不会发生呢？起码我自己就干过很多次，况且我也亲眼见过别人在犯这个错误。但从我第一次犯下这个错误之后，我就吸取了深刻的教训，凡日后书写更新或删除语句时，我都会再按下回车之前想想能否加入limit限制，以便即使我的语句有错，也能尽量的将受影响的数据降至最低！而每次当我因为写错where语句或者忘记写where部分的时候，我都在庆幸多亏我加了limit 限制了被更新或删除的数据的数量，接着就只消考虑如何恢复被修改的数据即可。

### 二进制字段类型

进行数据库开发的时候，有时我们需要向数据库中存储纯粹意义的二进制数据。可以存储二进制数据的字段类型有很多，char、varchar、binary、varbinary、blob等等。如果我们仅仅只存储二进制数据，那么我们就不要使用char或varchar作为字段类型。虽然使用它们作为字段类型也可以存储二进制数据并且没有使用上的影响，但是这两个类型的更广泛的用途是被用作存储纯粹意义的字符串！当我们看一个已存在的表达的结构，或者阅读一个表的初始化语句的时候，如果设计这个表的人以varchar类型存储二进制，你是否会很容易理解为这个字段是存储字符串的呢？我相信大部分人的回答都是“是”！造成这种歧义的原因不在读者是否按惯性思维理解这个字段的用途，而是设计表的人没有尊重这种惯性思维。如果以binary或blob作为存储二进制的字段类型，那么没有人会以为这个字段是存储字符串的，都会不言自喻的明白这里是存储二进制数据段，因此也就免去了不必要的理解和解释成本！

### 64位化

这是时代的趋势……

### 使用语言包

### 价格与SPID

凡是涉及到价格的地方，请充分考虑多运营支持。

## BUG查找

### 断点调试

### 附加调试

### 远程调试

### 进程转储

### 内存读写断点

### 排除法

### 逆向追逐法

错误显示为什么、哪里能够显示这样的错误、什么条件导致显示这样的错误。

# 程序员之外的

## 善始

什么样的开始，将决定什么样的结果！如果没有一个良好的开始，那么即使在中途拼尽全力的去改变，恐怕连事倍功半也难以做到。

在我们的事业中，大部分工作是需要团队协调配合的完成的，而同一份细小的工作往往也是先后会由多个团队中的成员来完成和修改。后参与的人总是遵循先前的人工作的方式和方法来延续这个细小的工作的做法，这可能也是一种对前人的尊重。

在我们早前开发的一款游戏的服务端中，我们配套的的提供了一款脚本编辑器。编辑器中具有自动完成以及代码提示的功能（与VC或Delphi的代码提示相同的功能），而这些功能的使用需要事先将函数的描述录入到脚本编辑器的函数库中。早前的人在开发的时候每增加一个脚本函数，都会录入到脚本编辑器中并且书写完整清晰的函数介绍，后来接手开发的人也继续的坚持了这一好的行为。这带来的好处是在进行脚本开发的时候可以方便的使用代码函数自动补齐以及函数提示功能，极大的增加了脚本的开发、修改和阅读的速度，也使得开发时不可能打错函数名称。新人上手脚本开发的速度也非常快。而在这款游戏之后的一款服务端中，使用的仍旧是相同的脚本编辑器，但是却没有人将脚本函数录入到脚本编辑器中！这就导致进行脚本开发的时候必须亲自到脚本导出的头文件里翻函数名称和说明，而写出的脚本给新人看时，却没法使用编辑器的代码提示功能，增加了上手的难度。同时修改脚本时使用的各种函数名称也很容易出现名称错误或大小写匹配错误!

当接手修改一份代码的时候，为了遵循同一个文件的风格一致性，我们会按照原先代码的风格进行命名、新修改；当我们实现一个新的功能的时候，我们会参考系统中现有类似功能模块的实现方式和注意事项，来实现这个新的功能。当我们作为一个新人加入一个新的团队并刚开始投入工作的时候尤其会这样做。编写代码如此、程序的架构如此、设计的思想和做人的思想也如此。这就意味着大部分情况下新人进行的修改或者实现的新的功能或者进行的新的设计会被局限在前人的思考范围内！前人的做法将成为后人的参考和标准，只有极少数的人能够突破，并且能够突破的人往往需要将原来的工作推翻并且重新建立。这样是一种非常优秀的做法，一切思想的技术的进步都是由这样的人来推进的。但是这样的人在新人中很难找到，大部分都是熟悉了工作并且参与了很长的时间，加上自己本身能够有着建立更合适的方法来取代现有方法的能力才能做到。

当我们着手开始实现某事之前，我们应当进行充分的考虑，并且以最严格的方式要求要求自己。这样当后来的人接手这份工作，或者参考它的实现方式和思想的时候，也会自然而然的按先前的思维进行考虑，并且也会严格的要求自己，那么事情从头到尾都会是健康的，都会是可以持续发展的。

**好的事情会越来越好，糟糕的事情会越来越糟！**重点取决于这件事情的开始是良好的还是糟糕的。

## 善于学习

不善于学习的人，将不能与时俱进，注定无法跟上被时代发展的步伐！

## 注重分享

在善于通过学习提升自己能力的同时，也要善于同其他人分享你的知识和宝贵经验。这样我们的团队才能互相学习，共同进步，我们一起为之奋斗的事业才能有更高的收获！

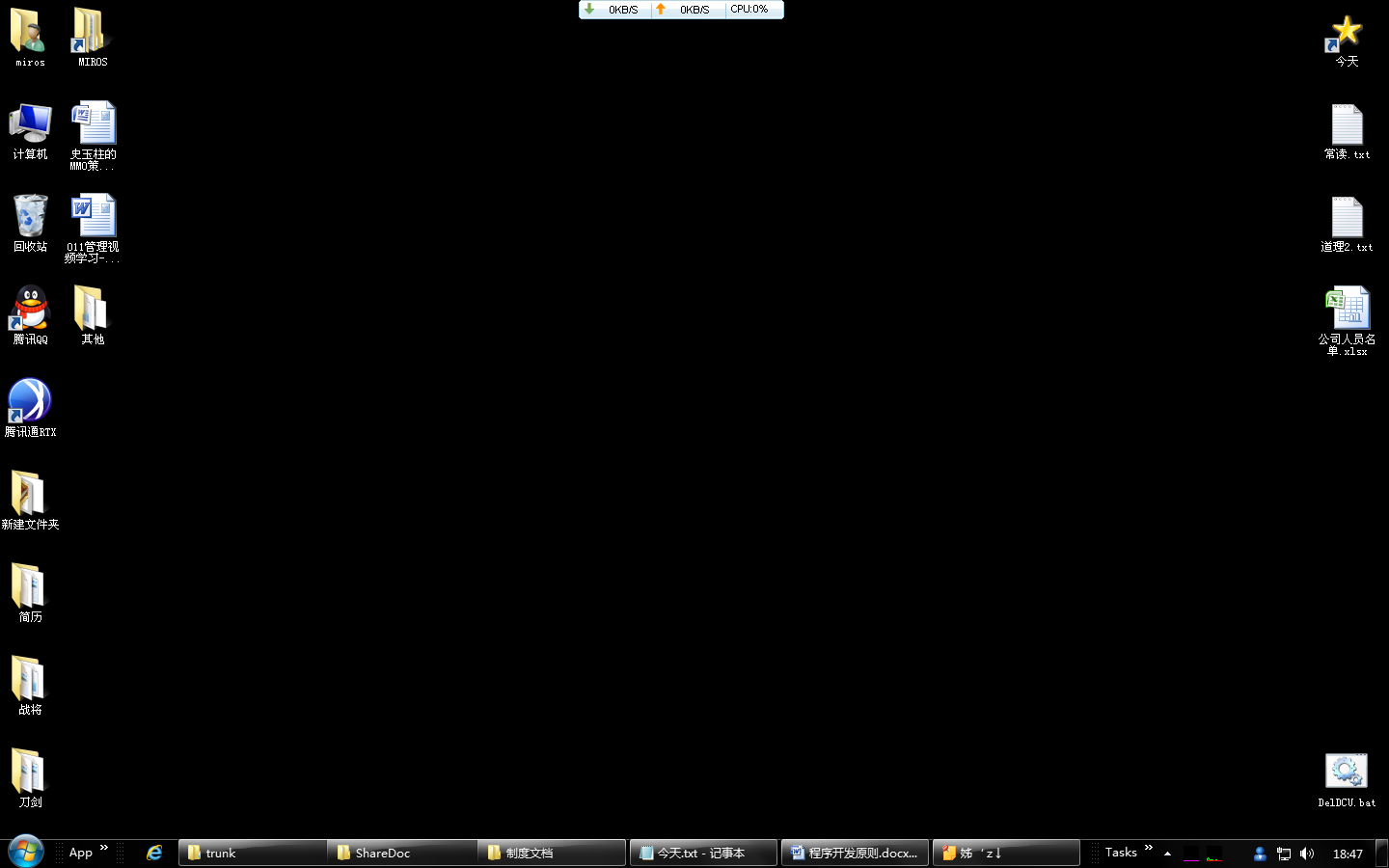
## 整理桌面

你工作电脑的桌面，是整洁还是堆满了各种文件，从一定程度上反映出个人是否善于将凌乱的事情整理得井井有序。如果你每天开机第一眼看到的是一张铺满各种图标的桌面，你今天的思绪又如何整理清楚。而当你习惯将文件丢到桌面，当你面对桌面几十个图标，如何能迅速快捷的找到一个文件？

古语有云：一屋不扫何以扫天下。同样如果连桌面都不会用心收拾，那么你的工作将如何做好，如何理顺？一位同事曾在我的建议下整理了他那看起来铺满图标的凌乱的桌面，不久很多看到他桌面的人发出过这样一句感叹：“哇！这么干净！”。可以看出这位同事之前凌乱的桌面不仅影响了他自己的工作心情，同样也影响到了别人的的心情以及这位同事留给别人的印象！

时常整理你的桌面，将各种内容的东西归类放到不同的文件夹中，将常用的放到快速启动中，久而久之会提升你对问题进行归类的能力，能够帮助从设计上将一个模块划分的更为清晰合理，能在代码结构上写出清晰易读的代码，并且在你生活中的其他方面也带来好的影响！

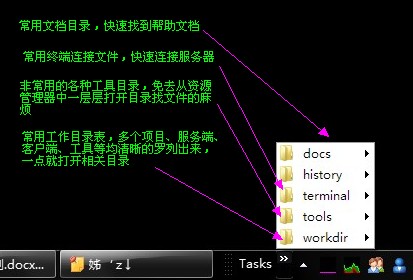
## 技术总监的桌面



桌面左侧为常用工作分类目录，右侧为各种不常用文件。各种文件清晰分类，要找什么文件直接凭感觉到相应的位置打开目录然后找到文件。工作直觉化了，那么效率就提高了。



将经常使用的软件放在快速启动栏中，不用去开始菜单里找，不用在桌面找，更不用一层层的从资源管理器中打开文件夹去启动。



合理的利用工具栏，将常用的文档、工具、工作目录都整理到工具栏中，点一下就打开比在资源管理器中一层层点开找到一个文件要高效得多！

## 留给人好印象

作为一个程序员，给你的同事、上级带来怎么样的映像，将导致你将来的职业生涯如何发展。

对本职工作的完成情况是给他人对个人最直接和坚固的印象。能够出色的完成自己的本职工作，降低BUG率，工作的结果满足公司要求，具备高的可维护性、可复用性以及完善的文档、注释，将给上级留下深刻的映像，赢得的将是能够更好的发挥你价值的机会！

而如果工作结果频繁的出现问题，且这些问题是因为没有很好的阅读或领会此文档中所强调的方面而引起的，那么只会加深别人对你的不良印象！长此以往，不仅影响你个人在薪水、职位方面的进步，更加会养成不良习惯从而制约人生的发展！

## 机会

我曾与一位在外打工的同乡小王聊天。这个小伙20出头，辍学后来着大城市里闯荡。小王向我说道他老板给他太多的事情做，一些本来不该由他做的或者已经不再工作时间的事情也总是安排他去做。他觉得老板认为他很老实所以总是想额外的利用他，有很多的人不使唤，反而每次都是找到他。所以他现在下班就赶紧跑出来了，手机也关掉，就是不想被老板使唤。

听到他说后，我反而在为小王的想法而担忧。首先毋庸置疑的是他的老板或许不该占有他的非工作时间。但考虑回来，老板在有事情做的时候会首先想到的是他，老板能形成“有事，让小王做”的这种惯性思维对小王来说是莫大的机会！在有事情需要做的时候会想到让小王做，而当有加薪或升值机会的时候，老板想到的是谁？

在当下这个竞争激烈的社会中，我们必须善于抓住那些能够历练和提升我们自己的机会，让上级和老板记住你的名字的最好方法就是按时按质的完成任务并且不出错误，如果能做得更快更好则更能加深对你的好印象，得到晋升和提拔的机会就此而来！这就是“名利”一词中何以将“名”放在前，“利”放在后，名和利是存在先后和因果关系的。

## 坚持成为习惯

经常做的事会形成习惯，包括好习惯和坏习惯。坚持任何好的行为以及严格的要求，时间长了就会形成习惯！好的习惯将是未来成功的莫大助力！注意观察身边的人，如果觉他在某一方面做的比较好，那么分析他再这方面有什么好的习惯，并向他学习可以形成这种习惯的方法，不久，在这方面你也会做的和他一样好，甚至比他好。

凡是入门的程序员，懂基本的语法，学会了使用常用的基类库，都是能够完成一份开发需求的。但要做好，满足实际应用，尤其是大型应用则未必是人人都有能力做好的。原因在新手比高手缺乏经验，没有注意到一些和原则的重要性——吸收经验、注意细节加上注重如何做的更好的态度，就是个人技术能力能够有多高、产品的高度能够有多大的根本！

吸收前人的经验、注意细节，再有一颗认真的执着的心，这样的人注定成功！

## 实践

有的人看过一篇觉得不错的文章之后会在心里感慨一句：嗯，不错！之后就没了下文。另有一种人看过觉得不错的文章之后会尝试做些什么……两种人的差距就从看过文章之后的不同行为而拉开！

仔细总结我们不难发现，所有可以提升技术能力或把事情做好更完善的都是在一些简单而零散的细节中，做这些细节做与不做带来的差别可能是细微的，但如果这个细节处在关键的地方，以及我们忽略了足够多的细节，那么汇聚起来的就是不同产品之间的差异！有时甚至我们能够想起这些细节但却没有辅助实施，短期来看对产品不利，而长期来看则更使个人形成不良习惯从而制约个人的能力和事业发展。

观察一下身边的“高手”或者那些令你佩服的人，看看他们是如何对待事情，如何对待工作的，是不是从每个能够注意的细节开始的？所以在任何时候保持对自己严格的要求，并从自己所做的工作和事情开始认真的对待这些细节，成功就在不远处了！

## 补充

对以上任何原则或说法存在异意或不清楚的地方，请随时与你的上级或技术总监沟通，让我们大家一起以去伪存真的态度完善这份手册，以至在现实中对我们的工作有更好的促进作用！

如果你有新的可以作为开发原则的体会和想法，请一定要和技术总监分享，技术总监会将有价值的思想加入为新的开发指南，让我们一起分享经验，完善这份指南！

1. 请上网搜索关键字“C++设计模式”或“Java设计模式”。 [↑](#footnote-ref-0)
2. 严格上讲仍有极少的值具有很高的精度要求，是需要调用获取时间的API的。 [↑](#footnote-ref-1)
3. 如果不了解字节对齐以及内存补齐，请自行上网搜索关键字“字节对齐”以及“内存补齐”，限于篇幅，本文不再赘述。 [↑](#footnote-ref-2)
4. 若不了解此算法，请自行上网搜索关键字“二分算法”。 [↑](#footnote-ref-3)
5. 如果不曾了解布尔短路，则请上网搜索关键字“布尔短路”。 [↑](#footnote-ref-4)
6. 64位编译器可以支持使用寄存器传递更多的参数，通常出于通用性我们尽量还是以32位为标准，这样无论在32位和64位下都能很好的体现这一优化原则意义。 [↑](#footnote-ref-5)
7. 在Delphi编译的Object Pascal语言中，编译器会对传递结构体的调用函数以及被调函数进行优化，如果传递的结构体没有在被调函数中修改，那么编译器会在编译时将参数优化为指针。对于函数返回结构体也有优化，具体表现为直接在调用函数中保留一份栈空间并将空间地址传递给被调函数，被调函数则在调用函数传递的空间中写入值，因此也没有产生返回值的内存拷贝。但即使如此，我们还是应该注意这一问题并在代码中显示的使用引用或指针参数。 [↑](#footnote-ref-6)
8. 如需了解更多，上网搜索关键词：“保护模式”、“用户态”、 “Windows线程调度”、“内核IO同步” [↑](#footnote-ref-7)
9. 虽然有些压缩和解压缩软件支持直接在压缩文件中查看或运行文件，但实际上也是解压到了系统的临时目录中后从临时文件中打开的。 [↑](#footnote-ref-8)
10. 关于字符串的赋值：C++中目前没有普及的引用型字符串类，因此常用的字符数组、string类在字符串赋值的时候都会引发内存拷贝。在wylib中我们提供了RefString类来改善这一问题。而Delphi、AS3由于语言以及编译器内置了string类型，当一个字符串被赋值给另一个字符串的时候不会引发内存拷贝，编译器对他的实现相当于只是将目标字符串的指针指向源字符串，并增加源字符串的引用计数。 [↑](#footnote-ref-9)
11. 严格上讲仍有极少的值具有很高的精度要求，是需要调用获取时间的API的。 [↑](#footnote-ref-10)