**代码大全**

* P7 **把主要精力集中于构建活动，可以大大提高程序员的生产率。**  
  在最近的一个项目中，对于这一点，我是深有体会。我们花了很长的时间做设计，结果接下来的许多工作都在愉快的心情下完成。我觉得 P28 的那个食物链的例子更有说服力，**健康的生态环境中，海鸥吃新鲜的鲑鱼，鲑鱼吃新鲜的青鱼，青鱼吃新鲜的水蝽。这是一条健康的食物链。** 如果环境被污染了，水蝽在污染的水域游泳，那么海鸥，食物链的最后一环吃下的不仅仅是是不健康的鲑鱼体内的垃圾，还有青鱼，水蝽体内的污染物。**软件开发中，架构师吃掉需求，设计师吃掉架构，程序员，软件食物链的最后一环，消化掉设计。**如果一开始就被污染了，我们就不要指望程序员快乐了。整个软件都会具有放射性，周身都是缺陷，绝对导致程序员脾气暴躁、营养失调。在我们规模不大的团队里，一个人身兼数职，伤害更大。所以，项目一开始就决定了它能否成功。
* P7 **源代码——往往是对软件的唯一精确描述**  
  其实我们不必为没有精确的文档沮丧，不是吗？
* P13 **常见的软件隐喻**  
  好的隐喻可以让我们思考更多的问题，并走上正确的道路。我们是在 Writing Code，还是 Growing a System 还是 System Accretion 或是 Building Software ? 做不同软件有不同的方法，不要拘泥。
* P24 **避免使用错误的方法制造正确的产品**  
  往往我们在软件开发中会很强调测试。的确、测试是质量的保证。但是测试只保证有质量的代码，却不保证有质量的设计。
* P42 **需求的 checklist**  
  其实我们不必去照本宣科的写需求分析书什么的，做需求分析即使是在大脑中，口头交流上完成，也是有这么一个过程。落下文字固然是好的，但并不是重点。关键在于做不做。**是否详细定义了系统的全部输入，包括来源、精度、取值范围、出现频率。是否详细定义了系统全部输出，包括目的，精度，取值范围、出现频率，格式？是否定义了机器内存和剩余磁盘空间的最小值？是否详细定义了系统的可维护性，包括适应特定功能的变更、操作环境的变更、与其他软件的接口的变更能力？** 书中列的远比我这里列出的多，非常值得一读。定下这些是很重要的，我觉得合理的游戏开发，是有一个相对稳定的策划方案，和一些已经完成完成的美术资源。大部分的变更都留在下一版本去做。策划和美术永远为下一个版本工作，而程序可以根据相对稳定的需求做设计。这样做，即使第一个版本是不可玩的，扔掉，也是能让游戏最终成功。
* P46 **数据设计**  
  我曾经很迷惑项目文档到底要写什么？这里列举的一些东西解开了我一些疑惑。如果你选择使用一个顺序访问的列表表示一组数据，就应该说明为什么顺序访问比随机访问更好。（往往随机访问更为高效）**在构建期间，这些信息让你能洞察架构师的思想。在维护阶段，这种洞察力是无价之宝。** 后面 P58 有个更为有趣的例子：Beth 想做丈夫 Adbul 家祖传的炖肉。Adbul 说，先撒上胡椒和盐，然后去头去尾，最后放在锅里盖上盖子炖就好了。Beth 就问了，“为什么要去头去尾呢？” Abdul 回答说，我不知道，我一直这么做，这要问我妈。他打电话回家一问，母亲也说不知道，她一直这么做，这个问题要问奶奶。母亲就打了个电话给奶奶，奶奶回答说，“我不知道你为什么要去头去尾，我这么做是因为我的锅太小了装不下”:D **架构应该描述所有主要决策的动机。**
* P48 **国际化和本地化**  
  国际化常常被称为 i18n 是因为 Internationalization 这个单词太长了，I 和n 之间有 18 个字母。 同理，通常本地化简写为 l10n 。这个工作一定要在构架期想好啊，到底我们需不需要 i18n 或者支持 l10n 就够了，到底用 UTF-16 还是 UTF-8 还是 ascii 串就可以。是代码中嵌入字串，还是封装到一个类，还是可以放去一个配置文件。我们现在的项目，一开始考虑岔了，结果后来花了很多精力重构。好在发现问题不算晚，否则代价更高。引申的想一想，发现问题我们应该最早的解决，不要怕做出错误的决定，因为更可怕的是，意识到错误后，因为害怕修正错误的代价过大，而一拖再拖。代码的 bugfix 大家都知道应该立刻做，但是设计失误却容易被放过，将就着做下去。
* P51 **过度工程**  
  这个问题把握好并不容易。一方面，我们希望系统健壮，如果组成系统的各个部分只在最低限度满足健壮性要求，那么整体通常是达不到要求的。**软件健壮性不取决于最薄弱的一环，而是等于所有薄弱环节的乘积。**构架应该指出每个部分，程序员为了谨慎而宁可做过度工程，还是做出简单的能工作的东西就够了。有些东西是不应该过分花精力的，这个错误我们也犯过，尤其一些一开始就知道以后很可能要重构的部分，大量的精力花在里面很浪费。
* P62 **选择编程语言**  
  我曾经也觉得 C++ 是万能的，这种想法很多 C++ 程序员也有。但是无可否认，每种语言的表达力是不同的。书在这页有一张表，如果 C 的表达能力是 1 的话，C++ 和 Java 就是 2.5 。而 perl 和 python 却有 6 。这就是我们选择游戏逻辑脚本编写的原因之一。另外对语言的熟悉程度是很影响程序员的效率的，所以我们不能独立的看语言本身的表达能力。P63 有个例子，用一群 Fortran 背景的程序员去用 C++ 编写一个新系统，结果他们编写出的是伪装成 C++ 的 Fortran 代码。他们扭曲 C++ 来模拟 Fortran 的不良特性并且忽略了 C++ 丰富的面向对象能力。我们这里有个现成的例子，一个 C++ 程序员用 C++/C 的方式写 [Lua](http://blog.codingnow.com/cloud/Lua5) ，结果可想而知。到现在我还在叮嘱他，一定要理解，再理解 [Lua](http://blog.codingnow.com/cloud/Lua5) 。lua 不是 C 。
* P68 **Programming into a Language**  
  注意，这里是 into 而不是 in 。书这里用了一个 vb 的例子来说明，恰好我也有个例子。我们现在用 C++ 构建系统，C++ 里有个相当麻烦的东西，就是单件的生存期问题。一个 singleton 到底什么时候创建出来，什么是否析构，相信很多 C++ 程序员在构建大系统的时候都头痛过。据我所知，我们公司别的项目的同事到现在还在头痛这个问题。这次我做了一个约定，禁止任何模块的代码构造静态对象，也就是说，任何在 main 函数前自动的对象构造过程和 main 函数之后的自动析构过程都是不允许的。然后我们有一整套管理单件的方法供使用，这个问题被很好的解决了。我们再也没有为某个单件什么时候构造出来的，或是为什么他提前析构了的问题烦恼过。
* P78 **管理复杂度的重要性**  
  我们做软件，就是在和问题的复杂度做斗争。有三个问题需要注意：用复杂的方法解决简单的问题；用简单但错误的方法解决复杂的问题；用不恰当的复杂方法解决复杂的问题。
* P80 **high fan-in 和 high fan-out**  
  高内聚，低耦合很容易被重视。但是高扇入低扇出有时候会被忽略。这里是说，我们应该尽量的大量的使用某个低层次上给定的类(high fan-in) 而每个类都应该尽量少使用其他的类(控制在7个之下)。
* P83 **子系统间尽量减少联系**  
  书里的例子很贴切。一个通用的规则是，如果 A 系统用到 B ，B 又用到 C ，那么 C 就不要再用到 A 了。系统层的设计图应该是无环图。
* P91 **封装不仅仅是有简化过的模型看到复杂的概念，而且同时还不能让你看到复杂概念的任何细节**  
  隐藏信息的重要性毋庸质疑。所以我们现在不仅用 C++ 的 private 隐藏信息。还用接口的方法，不在头文件暴露任何设计细节。另外，任何一个不满足现状的程序员，对自己以前的代码一定不会满意。但是复用老的不满意的代码并非坏事。我们需要做的是，重用的时候，把老的东西隐藏起来。
* P99 **预料不同程度的变化**  
  好的设计人员应该对以后可能变化的部分很敏感。这点很有体会，我自己就是这样一步步过来的。做一个决定之前，想想如果他做错了的后果。估计未来改变的成本，以作出合适的设计，非常的重要。系统为了适应每种变化而做过度设计又是不合适的。到底怎样设计，需要良好的感觉。
* P101 **耦合的种类**  
  松散耦合是每个系统设计人员所追求的东西。但是其标准往往把握不准。举个简单的例子，不一定恰当。在我最早的设计里，系统把坐标这个东西封装成一个叫做 point ，以后参数传递都传 point \* ，而不是 x,y 。这看使很合理。但是，这的确增加了耦合度。因为每个类都需要知道 point 的细节。很多情况下，用简单类型做参数传递反而更合适。(到底传 point \* 还是 x,y 依旧要根据实际情况靠量) 参数过多也会导致耦合度的增加，从这个角度看，x,y 是两个参数， point \* 是一个参数。关于耦合程度的问题，没有绝对唯一的标准。书里的阐述和总结非常值得一看。
* P103 **查阅常用的设计模式**  
  设计模式这个东西，给程序员带来的最大的好处就是增加了交流的便捷。一个人可以思考的深度取决于他用于思考的语言掌握的词汇量。设计模式也可以给程序员带来这种便捷。其实常用的设计模式，即使你没有看过《设计模式》这本书，只要你是一个经验丰富的程序员，这些估计大多考虑过。但是，给设计模式起个名字，却可以加快伙伴间的交流。不过必须警惕一个陷阱，那就是为模式而模式。强迫代码去使用某个模式是很危险的。
* P106 **避免失误**  
  失败的经验比成功的经验重要。如今网游开发尤甚。我们的游戏能成功，是因为我们在失败后吸取教训，小心谨慎。我们的代码未必质量高很多，但是很多业内同仁做出的东西吸取了许多人家成功的经验却失败了，正是因为他们缺少对失败教训的学习。
* P111 **自上而下和自下而上的设计方法**  
  很多人都赞同自上而下的设计方法，把问题逐步分解，再分而治之。而我喜欢至下而上的设计方法。先把绝对要做的模块做了，再考虑怎么把他们搭起来。但是我在实际操作的时候往往是，自下而上的做，自上而下的思考。书这里对两种方法的优缺点的总结非常精辟。我特别同意最后的结论，**这两种方法并不是互相排斥的——你会受益于二者的相互协作。**
* P114 **开发人员不把原型代码当作可以抛弃的代码。**  
  这个问题很严重，我多次看到过了。做原型真的是一个好方法，但一定要明白，有些代码写出来就是为了以后扔掉的。
* P118 **使用数码相机**  
  设计文档不一定要详细的写成规范的文档，重要的是我们做了设计并保留下来，而不是写出漂亮的文档（去蒙投资？）数码相机似乎是一个好东西，这样可以把草稿纸上的乱涂乱画保留下来。我一直觉得铅笔是设计师最好的伙伴。开会的白板也是，虽然现在已经有电子化的白板，但是，昂贵的东西不是所有开发人员都需要的。
* P131 **不要让 ADT 依赖于储存介质**  
  这点早就意识的到，类里面最好不要有 readfromfile ， writetofile 这样的方法。但是为了方便，往往又会加上这些方法。最终的结果是，依赖文件带来的不便总是比便利要多。同理，依赖文件名也是不恰当的。可悲的是，有些错误犯一次往往不够。意识到这样做的不好是很容易的，真正杜绝它是另一件事。
* P143 **在万不得已时通过 private 继承来实现“has a”的关系**  
  private 继承的主要原因是让外层的类能够访问内层被包含类的 protected 成员。根据我自己的经验，让类有 protected 成员本身就不是一件好事。我刚学 C++ 的时候，很多类都有大量的 protected ，但现在，只剩下了 private 和 public 。虽然偶有 protected ，但是我相信，那些都是可以通过改良设计然后去掉的。我不喜欢教条，如果有教条说，不准用 protected ，我会很反感。而实现上，我的感觉会排斥 protected 的使用。
* P174 **对于超过 200 行代码的子程序来说，没有哪项研究发现它更够降低成本和/或降低出错率**  
  这是个谁都懂的道理，但是错误我自己也犯过。早几年，我为大话西游编写了一个图文混排的函数，可以在聊天或是其他屏幕文字中同时显示不同的文字，有不同的颜色，状态，效果，还可以显示动画图标。当时图一时方便，还有一些效率原因，我写了一个几百行的函数。再接下来的几年里，我为这个函数多次头痛过，出过不只三次的 bug 。直到在梦幻西游的项目里，这些模块得到全部重写才舒了一口气。长达千行的子程序我也在一些地方见过，那也是非常可怕的。并非所有程序员可以真正认识到这样做的可怕程度吧。
* P176 **使用 C++ 中的 const 关键字来定义输入参数**  
  能够正确充分的使用 const 是合格的 c++ 程序员评判标准之一。
* P176 **不要把子程序的参数用做工作变量**  
  不要因为想当然的效率因素使用输入参数做工作变量，编译器会帮你优化的。如果这样做，至少在我们公司内部的 codeview 上是要被严重警告的。
* P210 **确认留在代码中的错误消息是友好的**  
  读这段的时候想到同事给我讲的一个故事：他以前的同事因为敲出脏话被开掉了，起因是他的老板（也是一个程序员）在调试代码的时候被某个模块的出错信息骂了一通 :D
* P220 **通过伪代码编程过程创建子程序**  
  并非所有的代码编写前都要有伪代码的。更古老一点的编程方法里，要求编写程序前先画框图。这种方法我也学习过，但始终不觉得自然。伪代码如果也有条条框框，说不定我也不会去用。但是，实现复杂算法时，这种方法依旧被我使用的最多。一个附带的好处就是，先有了注释，然后删掉多余的；而不是先有代码，再加上注释。
* P238 **数据认知测试**  
  哈哈，我得了 28 分（或者是 27.5) ，而且没有被作者的陷阱框住。
* P240 **隐式变量声明对于任何一种语言来说都是最具危险性的特性之一**  
  早年，我们用 [Lua](http://blog.codingnow.com/cloud/Lua5) 的第 4 版写脚本，结果在这个问题上吃过几次大苦头。好在 lua 在第 5 版后增加了 metatable 可以有办法避免这个问题。那就是给 \_G 定义一个 \_\_newindex 的 method 。例子在 lua 包里的 test 目录下可以找到，[[External Link]pil](http://www.lua.org/pil) 里也应该有提到。
* P244 **Rob Pike 建议使用 0xDEADBEEF 这一常量来填充内存，因为在调试器里很容易识别它。**  
  其实还可以用 0xBADF00D 也很有趣，可惜是 7 个字母。有个朋友用这个做网名，我还蹭过一顿饭。:D
* P246 **尽可能缩短变量的存活时间**  
  这是一个浅显但实用的道理。p248 里还提到，全局变量的存活时间最长，就凭这一点，我们也应该避免使用。使用全局变量和不使用，是关易写代码和易读代码的区别；也是“方便性”和“智力可管理性”的理念区别。这一节还有个被量化的概念，变量的跨度。把一个临时变量重复使用，在增加了存活时间的同时也增加了其跨度，所以出现了很糟糕的味道。这一点在 P255 又谈了一次。
* P253 **绑定时间**  
  一般说来，变量的绑定时间越晚，灵活性越好。这里的关于绑定时间的总结很不错。分别为：
  + 编码时绑定 (使用 magic number)
  + 编译时绑定 (使用命名常量)
  + 加载时绑定 (读注册表，配置文件等)
  + 对象实例化时绑定 (创建对象时读入)
  + 即时 (每次操作时读入)

绑定越晚，在增加灵活性的同时，也增加了耦合度。

* P256 **避免让代码具有隐含含义**  
  这里有个例子其实我曾经也类似的干过，变量 pageCount 在非负的时候表示已打印纸张的数量，否则，-1 表示有错误发生。这在技术领域里被称为“混合耦合”是应该避免的。pageCount 客串了一个 boolean 类型。如果真的想节省一个变量的空间的话，我觉得使用 union 可能能好些。

union {

int pageCount;

struct {

unsigned int unused:31;

unsigned int pageError:1;

};

};

或许这也不是个更好的方案。

* P298 **在程序生命期中尽早决定国际化/本地化策略**  
  关于这个问题，我们吃过的苦头足够说明问题。另外，MS 倡导的 \_T() 宏不一定是一个优秀的方案。如果要支持 unicode ，UTF-8 是个不错的选择，虽然不总是最好的。UTF-8 表示汉字时 50% 超出的储存空间好好考虑一下。
* P301 **更安全的做法是使用 strlcpy() 或 strcpy\_s()**  
  很高兴看到这句话是在译注里列出，可见译者是用了心的。类似的译注，我在侯杰的译书中见过。不过 strlcpy 和 strncpy 的行为还是有区别的，strcpy 往往可以简单的用 strncpy 取代，但是 strlcpy 却没有这个效果。不过我也倾向于用 strlcpy ，它同 strcpy 一样的高效，而且的确也更安全。strncpy 在参数为 NULL 时会出错。
* P305 **把 enum 的第一个元素留做非法值**  
  这是这本书读到现在碰到的第一个以前没想过的技巧。一直我都是把非法值定义成最后一个 enum 值，或者定义成一个很大的特殊数字。书这里的道理很充分，因为一些没有合理初始化的变量往往是 0 ，把 0 作为非法值更容易捕捉到错误。
* P309 **Monty Python 为 20 世纪 60 年代英国经典电视连续剧，python 语言由此得名。**  
  还是译注，很有趣 :) 我以前并不知道 [[External Link]python 的名字是怎么来的](http://www.google.com/search?client=opera&rls=zh-cn&q=monty+python+%E8%AF%AD%E8%A8%80&sourceid=opera&ie=utf-8&oe=utf-8)。
* p329 **简化复杂的指针表达式**  
  上学的时候，我经常写 p->q->r->s.data 这种表达式，当我工作后，被这种表达式坑过几次后，**用额外的指针变量来提高代码清晰度**(p327) 同样成了我的信条。另外 p329 介绍的**画一个图**的方法也是被经常使用。
* P330 **分配一片保留的内存后备区域**  
  用样来保证程序崩溃的紧急处理所需要的内存是个简洁的方法。
* P364 **滥用case语言**  
  把不同的控制结构混在一起用的确是非常糟糕的习惯。不过有时候的确可以带来高一些的效率。我们可以不用它，但是不能读不懂它。这让我想起上次去跟同事一起出差，在火车上他谈到的一个例子：

int n = (count + 3) / 4;

switch (count % 4) {

case 0: do {

dosomething0();

case 1:

dosomething1();

case 2:

dosomething2();

case 3:

dosomething3();

} while (--n>0);

}

* P374 **在 while 循环更适用的时候，不要使用 for 循环**  
  道理早就明白，可是就是老不遵守 :( 我想还是因为懒。
* P376 **一个循环只做一件事情**  
  我在自己的书中谈性能时也提到过这一点，其实，合并几件事情在一个循环做不一定可以得到更高的性能。
* P377 **避免出现依赖循环下标最终取值的代码**  
  还是一个早就明白的道理，但是错误一犯再犯。
* P384 **for 循环变量的作用域**  
  VC 中的语义跟 C++ 标准定义的不一样。MS 推荐一种解决方法：

*#define for if (0); else for*

* P397 **如果为我工作的程序员用递归去计算阶乘，那么我宁愿换人**  
  说的好。如果我带的新同事在 C 语言里用递归计算斐波纳契数列，估计我也会教育一下的。
* P399 **让代码不包含 goto 并不是目的，而只是结果，把目标集中在消除 goto 上面是于事无益的。**  
  p408 还有一句，如果程序员知道存在替换方案，并且也愿意为使用 goto 辩解，那么用 goto 也无妨。
* P408 **软件开发这一领域是在限制程序员对代码的使用中得到发展的**  
  这里举的例子，允许根据行数或者标号调用子程序，是我早年用 basic 时梦寐以求的。不过那个时候 basic 过于简陋，期待这种功能是迫不得已。实在不行的时候还要自己扩展 basic 解释器。
* P429 **最后是去找一个好的方案而且同时避免引发灾难，而不要试图去寻找最佳的方案。**   
  我想补充，如果你知道一种更好的方案但是不能完全避免灾难的时候，请注释这种更好的方案。
* P442 **这项建议（在等于表达式中的常数写在前面以避免把 == 错误的敲成 = 的问题）与按造数轴排列的建议相冲突。我个人偏向于使用数轴排序法，让编译器来告诉我有没有无意写出的赋值语句。**   
  我也不希望把常数写在前面，但老说不清楚原因。