《大规模C++程序设计》

图书基本信息

书名：《大规模C++程序设计》

13位ISBN编号：9787508315041

10位ISBN编号：7508315049

出版时间：2003.9

出版社：中国电力出版社

作者：John Lakos

页数：624

译者：李师贤,明仲,曾新红,刘显明 等

《大规模C++程序设计》

内容概要

在本书中，Lakos介绍了将大型系统分解成较小且较好管理的组件层次结构（不是继承）的过程。这种具有非循环物理依赖的系统的维护、测试和重用从根本上比相互紧密依赖的系统更容易且更经济。此外，本书还说明了遵从好的物理设计和逻辑设计规划的动机。Lakos给读者提供了一系列用来消除循环依赖、编译时依赖和连接时（物理）依赖的特殊技术。

《大规模C++程序设计》

作者简介

John Lakos在Mentor Graphics公司工作。该公司编写的大规模C++程序比大多数其他公司要多，并且是首先尝试真正的大规模C++项目的公司之一。Lakos从1987年起就一直使用C++进行专业编程，并于1990年在哥哥伦比亚大学开设了面向对象编程方面的研究生课程。

《大规模C++程序设计》

书籍目录

《大规模C++程序设计》

精彩短评

1、凡是说这本书垃圾的人，都是做的小项目。对他们来说，这都是奇巧淫技。主题是绝缘
2、没能看完……
3、浏览一遍先
4、cpp不适合大规模项目呢
5、很有干货，很有启迪的一本C++程序开发方面的好书。
6、蜗居中小贝看的书
7、看完这本书你就会明白c++的纠结与失败。一个现代程序设计语言不应该让程序员考虑如此庞杂的细节。
8、不管他有用没用，先看了再说
9、翻的很不通顺 有些概念有些过时 但很多还是很有用
10、看明白的东西不多, 只对书中关于程序复杂度评定的几个图有些印象.
11、帮助不大

12、学习小贝
13、对于大型系统构建讲的非常到位！
14、这本书更多的是对 C++ 的友好批判。很适合 C++ 中毒的程序员读。

15、翻译真的是很不到位，不建议买中文版。虽然书中知识的应用领域很局限，不过经验很宝贵，值得一看的。
16、非常好的书，大规模程序设计涉及到不少不常关注的问题，书里做了很好的介绍
17、深入分析c++文件的结构，布局及设计结构
18、忘了讲什么的了
19、只能说泛读过这本书，自刚刚翻这本书时候起，就想着哪天我要是当了系统架构师了，而且手头有着大项目，我再来复习这本书，那是再好不过了！
20、大规模项目中的C++,时代久远，难免有些观点过时。翻译有些不靠谱...
21、观念有点老了，还有些翻译问题。
22、看来自己开发c++的经验的确不足，看着有些吃力，虽然在很多地方也能明白作者的意图。
看这本书的初步感觉就是，c++由于天生的一些局限，导致编译和链接阶段的大量问题；这些问题的解决办法有和oo的模式有冲突。
要全面掌握C++，真不是件容易的事情
23、2006年的时候，我们开发了一个比较大的系统，开发参与人数有十几个（其实也不错），在但是的机器情况下，有时候只是动了一个头文件，会导致很长的编译时间，在这本书中可以找到答案。
《大规模C++程序设计》这本书是在2006年开发完一个相对较大的系统后读的一本书，当时看完，是少有的让我看完觉得相见恨晚的书，书分两部分内容，C++的逻辑设计和物理设计，这本书写的很早，95年左右，所以很多逻辑设计的原则在后来都在很多其它书中出现，但是物理设计其他书很少涉及，但是物理设计在大规模程序设计（平台开发）、接口设计和模块解耦上有非常重要的左右，现在很多概念上可能将之称为部署方式，随便提一句，翻译上有些是不怎么好，但基本上不影响阅读。推荐给每一位C++开发者。
24、有帮助
25、很久没接触C++了，已经有点忘了当年看这本书时的感受

《大规模C++程序设计》

精彩书评

1、本书的写作背景比较特殊，面向的人群比较狭窄。满足以下条件，你可以好好看看这本书：1 在某个软件中，必须使用C++作为软件开发的主要语言。2 软件的代码规模在100万以上。也就是要有一百万以上的C++代码。有这两个条件，你就会发现书里面说的很有用。如果你觉得看不懂，觉得书里介绍的问题和解决方案，莫名其妙，很想拍转，那说明这本书不适合你，你还是别看了。
2、这两天，要改动1个基础的类型。之前参考书里的方法画了package之间的依赖图。这下方便了，顺着依赖图指示，从依赖关系少的底层包开始重构编译，逐次推进到顶层包，最后整个程序一次性编译通过。package依赖图的好处还不止于此：1，可以指出相互依赖的不合理现象。2，新增模块时，考虑对依赖图的影响，便于保持程序结构的稳定和简洁。3，有了程序的全局俯视图，平时可以做更多的调整和优化。----------之前，和大侠讨论说，如果有工具能把依赖关系细化到模块就好了。这段实践下来，应该没有这个必要。有个包里的模块关系有点搞，特别为这个包画个模块依赖图就够了。
3、其实lakos这本书讲述的问题很简单，就是包设计原则，这些原则跟OCP、DIP一样，在敏捷软件开发中都论述过，当然不是每个人都看懂了这些后面的原则；但是，在C++语言中，你找不到包这个抽象概念所对应的东西，而且C++的链接过程有太多初学者没有弄明白的内容，而恰恰是这些内容破坏了文件的物理组织，加上大多数未经训练的软件开发者都没有将文件、目录当作包组织的概念，造成了大多数项目工程中代码组织的混乱，也许他们是好运的，没有遇上lakos它们那样，一个版本要编译一周的厄运；其实lakos是想在这本书上提出一套规范，大家按照这套规范来设计具有良好物理组织的大规模C++软件，这曾经是老B为了兼容C的同时赋予自由散漫C++程序员的自由，但这些自由却成了C++程序员的噩梦；稍后在C++基础之上改进的语言中，无不悄悄的加上了物理设计的限制，例如java中：1、一个类只能放在一个源文件中，而不能像C++那样区分.h和.cpp，要是.h和.cpp的文件名还不一样怎么办，天哪。。。2、一个.java只能有一个public class，这样的要求强制开发者去严肃对待源文件的物理组织，这个物理单元提供的接口是什么？哪些作为实现？；3、没有了extern这样的链接指示符，避免了C++中两个毫无关系的obj，莫名的在链接时刻产生了依赖，java中一个import关键字的轻易搞定的依赖声明，在C++中lakos却要先保证.h/.cpp同名，然后还要禁止掉extern声明，他的idep工具才能正常工作，要是再遇到#include后面的大小写有区别。。。idep也不干活了。。。整本书下来,lakos除了苦口婆心的告诉大家依赖危害性，如何建立起一套规范从物理上来改善依赖，如何从逻辑设计上避免引入依赖，最后的最后还给了一套用来分析依赖、度量依赖和检测循环依赖的idep工具集。可惜的是，如此工程界大师的作品，却没人关注……
4、《蜗居》第24集3:30秒截图，有理由相信小贝是搞 C++ 的。 桌子上那本书放大看是《大规模Cpp程序设计》，为无数想为cpp献身的人士叹惜呀。搞了一辈子C++，结果老婆跟了宋思明。 这部电视剧深刻揭露了C++程序员的杯具性。相信这个重大发现将彻底粉碎那些少年们对程序员这个职业的向往，这个时代女生不会因为你写了一个搞笑程序而嫁给你。建议广大程序们建议你们周围的少男们看一下蜗居，如果他以你为荣，以后想当程序员。你可以告诉他，小贝就是程序员，然后告诉他宋思明是公务员。相信他就会重新考虑自己人生的选择了！！！！
5、因为有Java，所以现在大型系统会首选java。这本书所讲述的问题java都可以解决，并且很elegant!谁叫咱们已经进入2008年了呢。C++已经不在适合在大型系统中担当重要角色。仅此而以。
6、2006年的时候，我们开发了一个比较大的系统，开发参与人数有十几个（其实也不错），在但是的机器情况下，有时候只是动了一个头文件，会导致很长的编译时间，在这本书中可以找到答案。《大规模C++程序设计》这本书是在2006年开发完一个相对较大的系统后读的一本书，当时看完，是少有的让我看完觉得相见恨晚的书，书分两部分内容，C++的逻辑设计和物理设计，这本书写的很早，95年左右，所以很多逻辑设计的原则在后来都在很多其它书中出现，但是物理设计其他书很少涉及，但是物理设计在大规模程序设计（平台开发）、接口设计和模块解耦上有非常重要的左右，现在很多概念上可能将之称为部署方式，随便提一句，翻译上有些是不怎么好，但基本上不影响阅读。推荐给每一位C++开发者。
7、最近一直在看这本书，其中的一些方法之前也有用过，现在读来又有了新的收获，今天用了差不多一天的时间把当前的项目代码作了不小的重构，主要是从程序的物理结构方面，分层更明确，实现隐藏更彻底，编译耦合进一步降低，自己感觉还不错哦！

8、此书应该是属于pragmatics类型得书籍，还是很棒的这本书接近C语言接口和实现，可以说两本书讲的都是同一个主题，重点都在接口和实现两个主题上。虽然此书好像都在讲物理结构，恰好是作者想通过如此简单的概念去表达一个结构良好的程序设计。诚然此书是针对大型项目，但是对于小型项目也应该这样，良好的可维护性，可扩展性。。这本书的引申出来的理念值得每个程序员去掌握。
9、这本书真的很垃圾的，看完你就后悔。还说什么大规模C++的，其实连最基本的东西都没有说清楚。在指针如何使用方面根本没有说清楚。类那一块直接带过这本书真的很垃圾的，看完你就后悔。还说什么大规模C++的，其实连最基本的东西都没有说清楚。在指针如何使用方面根本没有说清楚。类那一块直接带过这本书真的很垃圾的，看完你就后悔。还说什么大规模C++的，其实连最基本的东西都没有说清楚。在指针如何使用方面根本没有说清楚。类那一块直接带过
10、这是一本定位很独特，甚至说有些奇怪的书。如果你想从该书中获得C++在实际使用中的经验或教训，你也许会失望。因为这本书里几乎都是十多行的小例子，而且这些例子并不比我们在TCPL、C++ Primer上看到的例子好多少。如果你想从该书中获得大型软件的设计经验，你也基本上会失望。因为这本书对系统设计的介绍，并没有高人一等之处。这本书，对大多数的人来说，是无用的。但是它真的一无是处吗？不是的。这本书的特色在于，它着眼于一个特殊而且狭窄的领域：C++大型系统的物理设计。它会告诉你如何组织你的文件，在.h和.cpp中分别应该写些什么，不该写些什么。它着眼于让你的系统更快的通过编译和链接，并更少的产生冗余代码和避免隐含的链接错误。并且使得“源文件”能够被尽可能方便的重用（对，你没看错，是源文件，不是类或接口之类）。我们也许曾经花过大量的时间去学习如何更好的抽象类和接口，如何使用设计模式做出漂亮的，兼顾扩展和内聚的组件关系。但是我们恐怕真的很少注意过源代码的组织、链接和编译的问题。因为我们以为那是编译器该做的事情。是的，在其它语言也许如此，但是在C/C++语言中，由于其特殊的底层操作方式，使得编译和链接问题并不像我们想象的那样轻松。稍不注意，我们就会实现出一个逻辑上漂亮，实际上运行缓慢，程序庞大的“四不象”来。这本书就让我们注意到了这一点，并且它还提出了不少有益的原则和建议。相信会对我们这些C++er们带来不小的帮助。这本书的适用层面非常狭窄，对于使用C++开发大系统的人，该书可以提供很大的帮助。对使用其它语言，或者仅用C++完成周边工作的人，该书的帮助不太大。此书的翻译并不完美，有些用词不够严谨，但应该说尚可。基本上可以体现原作者的意思。
11、有位专家推荐过，看了两百多页后没再看下去。翻译得不好，不知道谁可以再重新翻译一下。实在不行只能看英文原版了。另，也许与扫描的PDF格式的阅读感受也有一定关系。
12、还有20分钟下班，简单写几条。1，加强了内部依赖的概念。2，针对c++的特定语法，使得uml的应用比较叫明确了。3，对模块依赖性的定量计算方法，我很有一种冲动，以后新写程序时逐个模块的累加计算，随时发现依赖问题。4，印象比较深的是，哑元指针，在宿主对象里保存，还不产生依赖性，不错呀。下面说不满的地方：全书自己定了很多定义，结果把书搞得很厚。再加上翻译也不给力，看起来实在很吃力。看到后面越来越快，书到后面也越来越白。这书是从大侠那里借来的，相信物主也是这个过程。其实作者是来自企业界，完全可以写的简单点。不要这么搞得文绉绉的。真的大牛都在写条条杠杠，如effective 系列，都在写小薄书，如设计模式。自己搞一套也造成了很多的重复，如设计模式是不可回避的东西。最后说最大的收获：本书的一条优化是把data member指针化。于是我想，如果真的这样，data_member还有意义吗？是不是c++出了问题？还是用户出了问题？因为有这条启发，我在犹豫是否该给4分？

《大规模C++程序设计》

章节试读

1、《大规模C++程序设计》的笔记-第18页

声明与定义定义：一个声明将一个名称引入一个程序；一个定义提供了一个实体（例如，类型、实例、函数）在一个程序中的惟一描述。一个声明就是一个定义，除非：
它声明了一个没有详细说明函数体的函数；
它包含一个 extern 定义符并且没有初始化函数或函数体；
它是一个包含在一个类定义之内的静态类数据成员的声明；
它是一个类名声明；
它是一个 typedef 声明一个定义就是一个声明，除非：
它定义了一个静态类数据成员；
它定义了一个非内联成员函数。

2、《大规模C++程序设计》的笔记-第563页

在实现中使用基本类型
1.在已知安全的情况下，才可以允许使用short。
2.在实现中，也尽量不考虑用unsigned
在设计一个包，函数，组件时使用最简单有效的技术。

3、《大规模C++程序设计》的笔记-第186页

恩，在接口上并且只在名称上使用，这个概念总结得真好啊，可以消除一种形式的物理循环依赖。

4、《大规模C++程序设计》的笔记-第367页

绝缘，我不知道翻译是不是准确，我的理解就是C++中，声明与实现完全分离。
对于一个组件的实现不进行绝缘的决策，可能是基于该组件不是很广泛使用的认识。绝缘轻量级的，被广泛使用的并且通过值返回的对象，会显著降低系统的运行时性能。对于大型系统来说，大型的，被广泛使用的对象，要尽早进行绝缘。

5、《大规模C++程序设计》的笔记-第48页

每次你增加一个特征去取悦一个人，你就扰乱和潜在地骚扰了你的客户库的其他人。曾经发生过这样的事情：原本是轻量级和很有用的类，经过一段时间后变得过于臃肿，不但不能做好每件事情，而且毫不夸张地说，它们已经变得每件事情都做不好了。
说的多好 = =
贪心必须死

6、《大规模C++程序设计》的笔记-第252页

“ 私有数据成员是类的一个实现细节 ”，即它暴露了类的一部分实现给其使用者，如果想作彻底的封装，可以采用针对接口编程的模式，接口中除声明的虚函数接口外无其他。

7、《大规模C++程序设计》的笔记-第115页

C++中没有包的概念，但程序员心理要有这个形，表现在物理形式上就是目录结构的安排，

整个程序用到的基础库，逻辑事务层，中间驱动层，UI层。我习惯先有逻辑上的设计，再有物理上的分层，在每个层次的物理分层中又细分：把组件接口的头文件和其他头文件和实现文件分开，以便以组件的单独发布。

8、《大规模C++程序设计》的笔记-第503页

尽量避免返回const的值。比如const int func(); 函数的自定义类型参数尽量用引用，避免用指针和直接传值（直接传值造成对象拷贝，造成性能开销）。避免用指针是因为指针可以没有初值，而引用必须有值。永远不要企图删除一个通过引用传递的对象。如果无论何时，一个形参通过引用或指针传递其实参给一个函数，改函数既不修改也不存储它的可写地址，那么这个形参就应该声明为const。

避免将通过值传递给一个函数的形参声明为const。

```
void f(const int x){    //bad idea
  x++;                  //compile-time error
}
```

在接口中使用基本类型

1.避免在接口中使用short，改为使用int
2.避免在接口中使用unsigned整数，应该改为int整数。
3.避免在接口中使用long，应该使用int或自定义的大整数类型
4.考虑在接口中对于浮点类型只使用double，除非有强制性的原因才使用float或long double

9、《大规模C++程序设计》的笔记-第258页

同层次的循环依赖，可能导致升级为一个更高层的潜在问题。

在庞大的低层次的子系统中，循环物理依赖将最大程度的增加维护系统开销，也就是越低层的模块组件，应该越独立可靠才对。

如果同层的组件产生循环依赖，应该考虑把公有的依赖功能抽象为一个单独的，较低层次独立组件来共享。----------代码降级。
把一个系统分解为更小的组件，既可以使他更灵活，也可以使它更复杂，因为要与更多的部件（组件）一起协同工作了。对回调的需求可能是系统不良整体设计的症状
根据以上，异步的东西更难理解是正常的，有什么办法避免？

10、《大规模C++程序设计》的笔记-第90页

预处理宏
在.c或.cpp文件中预处理宏的好处才超过它们带来的问题。

除非是作为包含卫哨，否则在头文件中应避免使用预处理宏。.h文件中最好不要有文件作用域范围的enum typedef extern const之类的变量定义，也不要有函数定义，即使是内联函数也不要轻易在头文件中定义。

11、《大规模C++程序设计》的笔记-第472页

1. 避免在函数接口中使用short， c++语言要求在表达式前装short提升为int;

2. 避免在函数接口中使用unsigned整数，c++语言要求在一个有unsigned参加的双目操作中，先要将另

一个整数转换成ungigned；

3. 避免在函数接口中使用log, 用int或自定义的大整数来代替;

4. 避免在函数接口中使用float, long double, 使用double;

12、《大规模C++程序设计》的笔记-第270页

把私有成员函数单提出来，放到一个helper class中，并且这个helper class放到实现文件中去来对使用者隐藏。这个方法very goog!

13、《大规模C++程序设计》的笔记-第167页

测试
对于整个设计的层次结构（类的层次结构）进行分布式测试，比只在最高层接口进行测试有效得多。A调用B，B调用C，那么A,B,C都要测试。不要仅仅测试A。
在一个设计良好的模块化子系统中，可以对许多组件进行隔离测试，也就是模块独立性越高越好，可以降低与系统集成的风险。

层次图
每一个有向非循环图都可以被赋予惟一的层次号，一个有循环的图则不能，所以，一个好的层次设计，最好的树状的。在大多数真实情况下，大型软件的层次结构如果要被有效测试，那么它们必须是可层次化的。分层测试就是为每个组件添加一个测试驱动程序。
上图就是可划分层次的层次设计，没有循环依赖。可以有效测试。

14、《大规模C++程序设计》的笔记-第326页

封装过程接口：当我们为一个c++dll提供c接口的时候，其实在就是在封装过程接口

15、《大规模C++程序设计》的笔记-第326页

谁申请，谁释放

16、《大规模C++程序设计》的笔记-第156页

这个翻译的，唉，有的地方真是“ 书读百遍，其意自现 ”

17、《大规模C++程序设计》的笔记-第600页

打完收功！

18、《大规模C++程序设计》的笔记-第25页

不必要的编译时依赖
一个大型的C++程序通常会比同样的C程序有更多的头文件。一个文件包含不必要的头文件，是造成C++中过多耦合的原因。不管这些不必要的头文件是否会用到，过多的包含伪指令（#include），不仅会增加编译时的开销，而且也可能增加由于较低层次上改变了系统而必须重新编译整个程序的可能

性。如果忽略编译时依赖，那么有可能使得系统中每一个编译单元几乎包含每一个头文件，最终使整个工程项目编译变得跟爬行一样慢，这样会随着项目规模暴增。
不必要的全局空间污染
尽量别把typdef这样的类型定义放到类外，尽量放到类内，否则维护者不好找其定义。这样一来，声明新类型就可以：className::NewType object了。

遵循以上规则，就可以使冲突降到最低，同时使得在大系统中寻找逻辑实体更加容易。
物理设计
好的物理设计不仅仅是被动地决定系统的现有逻辑实体应如何划分。物理设计常蕴含要支配逻辑设计的输出。一个好的物理设计是一个没有循环的图。小项目物理设计不太重要，但是大型项目，物理设计往往是成功的关键。

19、《大规模C++程序设计》的笔记-第423页

包，不知道是不是翻译问题，其实书中说的就是一个Library，一个.lib,.a,或者.dll，.so。或者是更加广义抽象的一个大型库（框架）。

所以书中才会说，给包增加命名空间，和类名字函数前缀。避免包之间的循环依赖，使包层次化。

包绝缘
包呈现了一种比组件更高层次的抽象，在设计包时，应该把它的输出头文件数量最小化，可以提高可用性。
main程序
一个大型系统，应该是由很多main函数组成的，并不只是唯一一个main函数。系统并没有单一的顶部。比如Vsiual Studio IDE，有很多bin文件（编译器前端，后端，程序错误报告器等），各分其职。

20、《大规模C++程序设计》的笔记-第215页

这个管理类让我想起了设计模式中的Facade的模式，它对高层使用者隐藏了子系统的复杂性，对外提供了统一的操作接口。其实这种模式大家经常自觉不自觉地使用。

21、《大规模C++程序设计》的笔记-第128页

依赖关系
如果组件y.c编译时需要x.h，那么就说明组件y对组件x的编译时依赖。

如果对象文件y.o(编译y.c产生的)，包含未定义的符号，在链接时，需要x.o来辅助解析这些外部符号，那么就说明组件y对组件x的链接时依赖。

一个编译时依赖，几乎总是隐含一个链接时依赖。

组件的依赖关系具有传递性。

分析依赖包含图
假如系统编译成功的话，基本上能由#include的包含关系，就可以推断出系统的物理依赖。

组件包含指导方针-------只有当组件x确实并实质性的使用了定义在y中的函数或类，才应该#include y.h。不然就必须删掉。如果不删除，会造成编译时耦合，拖慢编译速度。

22、《大规模C++程序设计》的笔记-第31页

作者描述了一个对开发者（类的开发者）和客户（类的客户，即使用类的其他开发者）都有用的组织（类成员布局）。此布局细分为三组：创建函数（CREATOR)、操纵函数（MANIPULATOR）和访问函数（ACCESSOR），其中前两者是非常量成员函数，后者是常量成员函数。作者说明对于像 包装器（wapper） 这样的非常大型的类来说，其他组织方式可能更实用。
大致布局如下代码所示

```
class Car {
// ...
public:
    // CREATORS
    Car(int cost = 0);
    Car(const Car &amp;amp;car);
    ~Car();

    // MANIPULATORS
    Car&amp;amp; operator=(const Car &amp;amp;car);
    void addFuel(double numberOfGallons);
    void drive(double deltaGasPedal);
    void turn(double angleInDegrees);
    // ...

    // ACCESSORS
    double getFuel() const;
    double getRPMs() const;
    double getSpeed() const;
    // ...
};
```

有些人会设法把成员函数分组为 get/set 对，如图 1-6 所示。对某些用户来说，这种风格是概念误导的结果，认为一个对象不过就是一个有数据成员的公共数据结构，每一个数据成员必须既有一个“ get ”函数（访问函数）又有一个“ set ”函数（操纵函数）。这种风格本身可能（有时候）会阻碍真正封装接口的产生，在真正封装的接口中，数据成员没有必要透明地反映在对象的行为中。图 1-6 所示的代码：

```
class Car {
    double d_fuel;
    double d_speed;
    double d_rpms;

public:
    Car(int cost = 0);
    Car(const Car &amp;amp;car);
    ~Car();

    double getFuel() const;
    void setFuel(double numbersOfGallons);

    double getRPMs() const;
    vodi setRPMs(double rpms);
```

```
    double getSpeed() const;
    void setSpeed(double speedInMPH)
};
```

个人觉得这段话最后一句翻译有问题，不过看这段话应该也能够明白作者的意思。
类的数据成员是类的实现，类需要的是合理的接口，而并非生搬“ 每个数据成员需要成对的访问器函数 ”，这样有时候会阻碍开发者找到真正合适的接口。比如 图 1-6 代码的类接口明显不如开始所示的类接口（汽车 一般是耗油和加油，而不可能直接设置油箱里的剩余油量ヽ（〃＿〃）ノ。最后，还有在哪里放置数据成员的问题。完全封装好的类没有公共数据。从逻辑角度来看，数据成员只是类的实现细节。因此，许多人都愿意把类的实现细节（包括数据成员）放在类定义的末尾，如图 1-7所示。图 1-7代码如下所示：

```
class Car {
public:
    Car(int cost = 0);
    Car(const Car &amp;amp;car);

    // ...
private:
    double d_fuel;
    double d_speed;
    double d_rpms;
};
```

虽然这个组织对于不成熟的客户来说可能更具可读性，把实现细节藏在类定义末尾的尝试掩饰了它们并未被隐藏的事实。在头文件中实现细节的存在会影响编译时耦合的程度，这种耦合不会因为在类定义中重新放置这些细节就简单地消失。
因为本书要研究物理的和组织的设计问题，我们始终把头文件中的实现细节放在公共接口的前面（部分原因是要强调它们的存在）。在第 6 章中，我们会讨论像这样的实现级的混乱如何才能被整个地从一个头文件中移走，从而确实对客户隐藏。所以，慢慢看到第 6 章吧，看作者的方法是什么。
不过，个人觉得极有可能就是使用 pimpl 模式（或 pimpl机制）来在接口（头文件中）隐藏真正的实现（简单来说是在接口类中包含一个真正实现类的对象的私有指针，所有的实现最终都是由这个对象去完成，而接口类本身来说，就相当于一个纯接口层）。ヽ（￣▽￣）ノ
关于 pimpl 模式的一篇文章链接：PIMPL 模式的实现及应用

23、《大规模C++程序设计》的笔记-第263页

恩，学到了一个方法：对于声明成私有的helper function，可以消除掉，以减少头文件中的不稳定因素，从而增强了绝缘度。但是这里的helper function，指的是那些在实现中仅会利用到此类的公共接口的helper function，而这种情况比较少。

24、《大规模C++程序设计》的笔记-第105页

逻辑设计只研究体系结构问题；物理设计研究组织问题。物理设计集中研究系统中物理实体以及它们如何相互关联的问题。大多数传统的C++程序中，系统中每一个逻辑实体的源代码都必须驻留在一个物理实体中，一般称之为一个文件。基本上每一个C++程序都可以描述为一个文件的集合，这些文件有一些是头文件，而有一些是实现文件。对于小型系统足够了，对于大型系统就需要利用额外的结构来生成可维护，可测试和可重用的子系统。

一个组件就是物理设计的最小单位。

组件不是类。在结构上，组件是一个不可分割的物理单元，没有哪一部分可以独立地用在其它的组件中，一个组件的物理形态是标准的，并且独立于它的内容。一个组件严格地由一个头文件（.h）和一

个实现文件（.cpp）构成。（配对）。

一个组件一般会定义一个或多个紧密相关的类和被认定适合于所支持的抽象的任何自由运算符。
根据vczh的经验，他一般喜欢以文件来划分功能，那么我想可以等价于以组件来划分功能，一个功能划分为一个组件可能比较合适。
一个组件的物理接口就是它头文件中的所有东西。
物理设计规则
在一个组件内部声明的逻辑实体不应该在组件之外定义。对于一个可重用的组件来说，它必须合理地自我包含。

组成一个组件的.cpp和.h文件的名称应该严格匹配。比如stack.h stack.cpp

每个组件的.cpp文件都应该将包含它自己的.h文件的语句作为其代码的第一行有效的语句。

客户程序引用第三方库（组件）的时候应该包含直接提供类型定义的头文件，除非私有继承，应避免依赖一个头文件去包含另一个头文件。
对于上面，我的理解是，比如我们依赖库A（A.h,A.cpp），我们有个组件B也依赖库A，所以B.h中包含了A.h,但同时，我们的另一个组件C，它也依赖库A，它应该直接包含A.h，不应该为了图简单方便，直接包含B.h(B包含A)。可以看成，我们需要什么组件就直接用它的头文件，不要间接引用。

25、《大规模C++程序设计》的笔记-第166页

记不清具体的时间了，当时被灌输了一个概念：任何问题，都可以通过多加一层来解决掉。当时理解的不够深刻，后来在程序设计过程中慢慢也用到，这个章节讲的正是这个内容。

26、《大规模C++程序设计》的笔记-第5章 层次化

第5章主讲层次化，以前多数的设计讨论集中在逻辑设计上，这里则关注于物理设计，是大型项目的必修课。层次化的意义包括: 1. 消除循环依赖，提高设计的可维护性。2.达到分层测试，以提高可测试性。 3. 有利于重用代码。在作者介绍的技术者，更贴近于实践，设计概念在其它的设计模式资料都有介绍，有一些手法还是有点旧的，但它的核心思想是借助CCD(累积组件依赖)来衡量系统的依赖性，然后再进一步进行优化。

CCD就不介绍了，一般的大型项目里会对层次、目录的依赖性有明确的定义，主要以消除不必要的循环依赖为目标。比如Google Chromium, 对目录的依赖就有明确的定义, 比如Blink定义的组件化目标就对各个目录做了明确的定义,详情参考:Core/Module Componentization。其中web为对外的功能接口，其它层次使用这里定义的接口来使用Blink。core从字面就可以理解它是最为核心的实现。platform则是以委托的形式要求客户端实现的。而modules则是将分散于core中某个业务相关的逻辑抽离出来的组件。
上图是目前它们间的依赖关系，线上的数字代表了不同方向存在依赖的代码编译单元个数
。Chromium正在努力进行优化。

下面简单介绍一下作者提到的手法:
1 升级
如果同层次存在循环依赖，一是将一部分升级到高层次，比如Blink里的modules。 二是建立一个更高层次的类做为门面(Facade), 比如Chromium CAW层次中的AwContents。

2 降级
公共类、基础核心的代码可以降低到更低层次，以便于供上层各个模块重用。比如Blink中的platform，或者Chromium中的base。
3 d-pointer (Opaque Pointer)或者(dumb data)
即对象访问另一个对象时，不要提供对象的调用，可以通过增加一个接口提供另一个对象的指针或引用。个人认为这一条用在层次化上并不合适，存在反向依赖的问题。如果在层次内使用是倒是可以。
4 冗余
有时为了重用，反而会引入耦合。比如日志输出依赖于某些库提供的文件IO功能，而这个日志输出功能期望能在不同项目中使用，这个依赖也许就太重了。可以通过对IO的简单实现来代替，这时有一部分重复实现的冗余也是值得的。
5 回调
以回调函数的形式解除依赖，其实使用依赖倒置的方法更为恰当。
6 管理者类
将一些低级对象的操作集中起来，也可以进行适当的抽象。就是各类的XXXManager。
7 分解
一些比较复杂的对象可以通过分解的方式，从中提取出层次。
层次的概念，可以对比<<敏捷软件开发 模式 原则 实践>>中关于包的设计原则的说明。

层次的概念大家都很清楚，关键还是如何给以清晰的定义，努力克制自己走捷径的想法，多从结构的合理性考虑设计。用一些工具可以帮助整理这些依赖关系。比如SciTools Understand。

27、《大规模C++程序设计》的笔记-第383页

“ 最小化修改之后的源代码的编译时间，可以显著地减少开发开销 ”

《大规模C++程序设计》

版权说明