编程的那些事儿

——— 抽象眼光看编程

By Minlearn @ http://thousandd.appspot.com/

设计才是真正的编程！
对类型的设计才是设计！
面向对象并非一切？
无论你以为上述观点是惊天大秘或不过尔尔，你都需要这本书！

目 录

第一部分

前 言

By Chenyi

眼前这本书充分体现了作者的所思、所想、所感，他用自己独特的眼光审视着计算机技术的世界，也用自己独特的思维逻辑对技术进行解读，并用自己特有的，呵呵，偶尔带有“四个逗号=一个逗号”这样的语言风格，进行着自己的诠释。创新是一种美，独立思考也是：）

学习是一件因人而异的事情，因为每个人的生活经历、教育背景、年龄、认知模型等等，都是不尽相同的，也就是每个人所处的“维度”不同，而作者有一种“建立更高层抽象的能力”，用一种特有的方法尝试着给大家建立一个学习计算机的、相对高层的构架，这样，可以在一定程度上突破个人的“维度”，使大家从与周围事物建立联系开始，一步一步的走向计算机的世界。不识庐山真面目，只缘身在此山中。确实的，在学习技术的过程中，横看成岭侧成峰，远近高低各不同，但是作者却尽力想让这高低或是远近都不同的山峰，能在我们面前呈现出一种规律、共性来，这是难能可贵的，因为这个架构的过程对思维的要求是比较高的：)

哲语有云，动身的时候到了，有的人去生，有的人去死，只有上帝知道，于是这个问题被回归到“ To Be ? Or Not To Be ”的问题，是生，是死，只有上帝知道。

但是，人类对真理的探索和对知识的追求，却从来没有因为“生死”的维度而停止过，是的，一颗崇尚真理、探寻真理的海洋之心，将从来不会因为泰坦尼克号的沉沉而消沉，它将永远绽放在人们的心中，激励着我们向更广阔、更深髓的世界，一路前行、风雨无阻：）

在这个意义上，鼓励作者的写作和思路，也是对我们自身追寻真理的一种鼓励、一种回路。是为一点小感想：）与作者分享‼

By Minlearn

对思想的认识和界定是重要的！！因为我们需要一个知识体系才能不致于困惑！！(而身处编程界，纷繁复杂的术语和概念足以让一个初学者却步）

我抓住了哪些转瞬就在我脑中消失的思想，,因为它们远比一切成书的东西都让我感到它的珍贵！而更玄的是，他们竟然真的能够被文字描述出来！！这整本书就是小说式的教学。它力求呈现出一个精致化了的术语集。以使初学者能真正理解至关重要的那些概念。

正如 Chenyi 所说，每个人都是某个维度上的人，有他自己的年龄和认知，具体到某个历史时刻，我们的人生阅历已然被格定，而这决定了你接受新事物的能力和眼界，人生在世，已经不可能脱离某种信念（也异或某种阻力和障碍）而活，当我们开始学习编程，我们永远都是用外行的眼光去看待某样东西，而当你占在巨人的肩膀上成为一个专家之后，你就需要用全局的眼光去看待曾经陌生的知识，此时你不再是个学习者，而会批评产生你自己的认知，但那毕竟是要过的第二道槛，而初学者就是那些连第一道槛都难以过去的群体。

这其中最大的拦路虎就是对术语的理解，很多书并不切合初学者的实际，从他们的角度呈现一条清楚可见的理解路线，而只是一些大部头衍生下的反复抄袭品。

给你一个术语或道理，这个道理有什么用？没用，是的，因为要给你一个情景，你才能理解它，仅仅让你去学一个知识，而知识和众多其它知识之间相似而微有不同，如果不给出它被产生时的历史和它所处的架构(这本书不但给你思想，而且给你对应的细节)，那么我们就会迅速迷惑，更遑论运用它，因为我们不是泛化主义者，形而上学者(但是的确存在超前主义学说，只是为了创立一种学说，后来才慢慢与实践相结合)，我们需要一种与自身相联系点去理解它，我们只是生活的人，我们不是高高在上的学院派高手。

一个高手必定是与常人有不同的思想级深层的东西和他自己特有的体会，因为他也走过初学者才走过来的路经历过与所有人一样的迷惑，可是往往人们都忘了归纳那些至关重要的经验，那会是什么经验呢，那些是不会出现在任何描述具体技术细节的书里的思想级的东西，那么这本书尝试的正是记录那些秘诀，如果真的想当高手，请你不要错过这本书里任何一个字眼！！如果你是高手，这本书一定与你内心深处的某些想法偶合。

本书过后，再辅于其它教科书（比如你手头上的一本 C++教材,本书后面列举了一些与本书能很好答配的推荐参考书和源码）你应该会具备基本的编程能力和编程理解能

力。本书前半部分是对思想和认知的导论，后半部分注重真实的代码控制能力的形成。

对于学习方法，有二点区别是要深刻明白的，1，认识与实践 2，思想与细节。

知识是事物之间的联系，那么实践就是强化或深入这些联系的方法，我常想，到底是什么重要，是认知还是技能，人们普遍认为实践应在任何情况下都高于认识，事实是：可能有技能但是没有认知，但却不可能有认知却没有技能，就拿学习英语来说吧，看英语报纸也是一种实践，因为它也能够加强你实际使用英语的能力，(我不是在模糊这二者之间的区别，我只是企图站在这二者之上求得一种更泛化的认识)，实践不过更侧重动手能力而已，而认知跟它并不矛盾，知识的获得与能否运用知识本身无必然因果,拥有足够的知识，再加上泛型的思维，，你就会快速得以实践，一切都是一种格物致知的过程，只有格物至知，先格物，认识到了一定程序后就会产生对事物本质的认识，也可先认识事物本质再在指导下去发展技能，但是认知可以直接传递给你(至此只是一个你所能想象得到的浅层和大概,而且除非实践，这个大概形象你也不知道它到底是正确的还是错误的，更深层的你想象不到的抽象以及关于这些认识的正确性要求实践),相比之下一本书不可能传递很多实践的东西。本书前一部分正是力求让初学者完成从认知到实践的有效过渡。

所以说实践和认知都是重要的,没有谁比谁更重要的说法，然而对于初学者来说浅层认知的重要性要高于实践,一开始就有一个好的思想和基础显然可以为未来的实践扫清障碍，这是因为学习是一个层次上升阶段，在拥有一定知识后，理解基于这些知识之上的更高层知识会很快,, 即掌握了基础再加上一定勤奋的博物广识，知识量是几何级上升的,因此一种很好的学习方法是，学习应该先吞，(在一定知识量的前提下尽可量地博物广识，即使看不懂也要浏览完,以获得浅层的认知继续下一步学习),这是学习中自然而痛苦的过程。(不是提倡光谈和光看理论，而是把理论整理成一个架构也是一项重要的工作，不是不能直接把这个认知传递给你，而是需要再找一个与你的结合点来让你认识它，因此它是一本同时讲解到认知与实践的书，不是提倡导光谈理论，而是如果事先有理论的指导，那么学习中就会少走很多弯路，学习中最怕不能理解细节，更怕以为细节就是一切，所谓一叶屏目不见泰山，更有人把学习语言作为编程的终极目标，而如果事先有人给你指导，你就会少走很多弯路，这就是下面要谈到的思想与细节的关系)

我们鼓励在实践基础上去学习，也提倡速成,我认为学习不应该提倡逐步深入，人的生命有限,染启超在渡日的般上一夜之间学会日语,这就是说他掌握了思想，细节的东西永远是后来的，只要思想是重要的，(了解足够多的细节才能泛思，但是，在学习编程中，除了一些对至关重要概念集的理解之外，，从来都不是大思想决定一切，而只是小细节，这就要求你作很多的实践)

学习应首先理解基本的框架和思想(这是泛读)，**然后是细节**(这就是对某些内容的精读)，虽然真正的学习往往是混合了这二个过程的过程，但大多数人显然不会拥有正规的理论教育(本书正是在为此努力)，**所以一开始对语言细节的学习和深刻理解永远都是学**

习编程的重头戏，如果说一些知识仅仅知其然就够了的话(仅仅是不致于迷惑和建立知识结构)，那么有一些知识却是要精通的，因为不但要知其然而且要实际拿来应用。(人月神话的作者虽然写出来的是一本思想书，但他固然精通很多细节)，但时时要提醒自己的是细节决不是一切，这就是我在前言的后半部分推荐给你的看书方法。，

问题随之而来，既然存在这二大界限，又如何突破，否则这就是一个空谈

多走弯路，学习是认识事物间联系的过程，而记忆或实践是加强这个联系的过程，，能够认识到事物之间的联系，即便是自想的联系也可加深对事实的记忆(一个程序员有他自己的知识体系是重要的)，这就是知识

编程时碰到的信息量永远是巨大的，有生之年我们不可能掌握这些信息的来龙去脉，对于程序员来说，提供一个关于它的编程参考文档可以说是掌握了此信息，因为这个文档就是这个信息的大概，实际上我们编程大部分情况下都只是用第三方的代码库来编程，这个信息用于编程所需的全部东西，对于编程来说只要掌握这些东西就行),换句话说，一些知识如果不能理解就得先放(这本书并不适合于赶考使用), 在这个信息的社会，至于信息，是撞事学事！一个程序员并不全知全能，它只提取和了解事物对于编程方面的信息。对于事物的逻辑认识，只能在对它的编程中不断掌握它，抽象是惯穿这本书的重要的思想，维度也是，我们是从学习编程的眼光来组织这本书的。也是站在初学者的角度来解释诸多概念及其关系的。

一切东西，我们应该查本究源,深入其原子世界(任何一个术语都不会简单，有它自己产生的环境与其它知识的联系，但也正是因为这样，这也决定了它的有域性，任何知识只要放在它自己的领域内去理解才能更容易被理解),, 翻译过很多文章你就知道要措词，措词跟概念有关,二个稍微相差不大的措词都会让读者摸不头脑或让他们恍然大悟。

然而千万不要走入另外一种极端, ,知识用词和技术用语没有一个标准，比如方法和函数指的是同一个东西，比如什么是线性(一次就是线性)，什么是离散(离散了的量)，这都是仁者见伍，智者见，但人们对此的理解都不会差到那里去，并且也不会影响后来的学习,这里有一个描述或形式的概念，相信大家都还记得初中学过的集合，是描述性概念，但集合其实还有一个形式概念,给定了形式就可以框死，书中力求对某些至关重要的那些概念进行形式化的描述。

而且，要知道，即使是《虚拟机的原理与设计》这本书的作者也会对别人书里的进程概念感到不解。

最后，什么是编程能力以及语言能力是不是就是编程能力的区别，比如拿编程语言来说，只要越过语言的表达这一层，我们才能用语言表达和理解事物(语言跟你要说的话就像用 C++语言去表达一个算法，方案领域跟应用领域的对应,就像穿鞋去上海， 穿上鞋只是开始，真正你要去的目标-上海还远着呢),就像口才，一个好口才的人说话时绝对

不会想到措词，因为语言已经成为一种意象,只要把一样东西思想化，才能超越这个东西而去想到别的东西而长足发展，比如面向对象，这本书将帮你解释为什么面向对象是一种科学的机制，解释的过后你甚至会觉得这是一种本来就该存在的很亲切的机制，只要超越了面向对象我们编程时，再加上一定设计模式，才能真正不会一动手编程就考虑什么是面向对象之类。。(而这些，好像都是一个高手所能做的事了。)。本书主体中的二部分就是认知和实践，思想和细节的结合，所以你要做的就是在认识的基础上作大量实践。，每天写小程序,编程能力就会日渐提高,而当你写过和分析过很多程序之后，你就会具备一眼看出的本事，过程的最后你发现自己蛹变蝶飞了，

我注意到程员序考试中多了一项标准化，的确，知识的传达也需进入标准化时代了

书中错误再所难免，望不吝赐教！！

- 别怀疑，这正是一本同时可作为入门和进阶级的书（更偏重入门）！然而真正的高手和真正的初学者都将从中有所得。
- 你还在为学不懂大量术语而烦恼吗？如果你真有这种体会，那么你可能先要看这本书再看你正在看的 C++的书，因为你仅仅缺少一根主线，而它是能让你少走很多弯路的拐棍
- 对架构的学习才是真正的学习,知识也有它的架构，然而在这本书内有架构也有细节(高手固然知道细节，然而高手也有精神空洞,因为你还需要懂得一些细节之外的架构级的东西)!
- Python blog 的从零到尾的实现，让你知道其实你一个人就可以做出流行的 web 程序！！(本书作者也自称是一个菜鸟，但是这个程序的确是他自己写的)

最后，如果说任何行为都是功利的，那么我写这本书的目的只为博你一笑。

感谢 CCTV，感谢 MTV,，噢，错了。。。

导 读

任何语言都是有门槛的

任何语言都是有门槛的，这是因为它们都面向解决编程问题，而即使在不谈到任何语言成份的情况下，编程这个小字眼实际上都是一个巨大的问题，因为它首先要解决三大问题（我们这里谈到的编程是狭义的，即利用编程语言写系统逻辑或应用逻辑）：

- 编程要能通过计算机解决现实问题，比如统一使用数据结构和算法，即问题模式。

- 编程是人，而且是大量人来进行的，所以它还必须解决语言代码模式和复用能力的问题。即软工问题。即设计模式。

- 再者，编程跟语言有关，所以它必须首先解决语言复杂度问题，即语言的语法；比如 OO，即代码模式。

而其实无论是上面问题的那一个，对于没有任何一点编程基础或者计算机应用基础的人来说（或者即使有点吧），都是门槛巨大的。

首先来说语法。

都是无穷无尽复杂的细节让初学者停滞了他们学习一门语言的脚步,在你开始满怀喜欢地按书上的例子用 C++开始写“Hello World”时却被 IDE 少引用了一个文件之类的问题折腾得一头恼怒浪费不少时间,而书上没有任何一点关于这些的描述,或者网上也只能找到零碎不成文的信息,即使一门语言内部被宣传得天花乱坠比如拥有 OO 有多少语言机制多么利于复用这种论调泛滥的的今天,在真正动手使用一门语言的时候,作为新手的他们还是会能遇到轻易就把他们击败的细节问题。

这些细节问题往往不是大思路大方向(比如什么是 OO,这个逻辑用什么来设计好,事实上，那些当然更难),反而是那些日常时常见到的工具上或语言上的陷阱，另外一种情况是，即使有时候这些细节问题可以被理解,但就是难于真正正确地应用(这就是那些被称为语言陷阱的东西，然而这是一个大问题，虽然对于个人来说不是，但对于软工这个多人共工的人类活动来说，语言上的陷阱它是),,这些都应该被尽量在语言被设计时就避免而不是人脑避免,比如 IDE 环境,比如语言细节,比如目标领域的问题,当一门语言复杂到你真正需要掌握这些才能编程的时候,编程就演变为不仅仅是简单的复用了,虽然我们以为 OO 语言就是简单的复用语言但它们明显做得不够.(语法绝对是新手最大的拦路虎,没有一门语言,即使 RUBY,也没有把语言细节弄得足够简单,使得你可以不管任何语法产生式的机制去直接构建你的应用,,在不损耗逻辑的情况下,而且 RUBY 的一些语法机制也不是不简单.需要你有深厚的系统知识去理解他们,比如协程,元编程等技术,在接确过一点 C++人的眼里,丝毫不比 C++简单)

再来说代码结构方面的问题。比如 OO，然而，OO 真的就简化了人们编程的工作吗？遗憾的是许多人并不认同这个说法。

C 用最初级的封装了汇编的逻辑写应用，C++用类来封装逻辑，如果你看过用 C 表

达的数据结构和用 C++来表达的数据结构你就会发现这二者的不同，有时看出来样子普通的一堆语句，你根本不知道它想表达什么上下文数据结构逻辑(C 太靠近描绘底层，而 C++除了用 C 的底层能力来描述数据结构它还加了一层设计和代码结构逻辑)，就 C 来说，虽然你真一开始就明白它想写什么（比如有文档的情况下）但你如果不明白数据结构在先也不会明白用 C 表达的数据结构究竟在表达什么，就 C++,Java 这样的面向对象语言来说，似乎能够整体上看起来是一个一个的类，显得意义明了，但实际上语言越接近应用问题反而代码看起来更复杂(因为 C++代码本身跟现实问题走了二个极端，你看懂了代码却看不明白代码后面数据结构方面的东西，这是因为有代码逻辑和现实问题同时在作梗)，况且，C 与 C++都是通用语言，现实生活是很难的你根本做不到绝对靠近除非你为每一个问题写一个 DSL(C 表达的向系统问题接近和 Java 表达的向现实靠近各有各的难处,作为一门工业语言，要求它看起来便于程序员理解和复用,Java 这方面是做得不错的而 C 肯定只是专家语言)，一堆有机类有时反而难于让人看出它想表达什么，而且另外一方面，类里同也是函数，也是类 C 的进程式的语句逻辑而不是封装上的类逻辑，这些语句逻辑，

Java 的库是高度 OO 经过设过的，相比 C 语言，比如它规范了流的概念，Java 的 OO 和规范的库是人们说 Java 比其它语言易的二个方面。。但是虽然熟悉编程的人可以拿来用，但是对于没有编程经验的人来说，它照样跟 C 一样难。

因此 Java 的所谓易，是相对用过 C 的这样的熟悉编程者来说的。至少使用 Java,我们照样得学好跟 C 一样的数据结构知识。。人们说 Ruby 偏向于人，但是几乎所有的语言都脱离不了数据结构，，都脱离不了底层,如果数据结构这些实现问题没有搞清，语言机制也就是一种更大的障碍了,Ruby 的 OO 只是指类的那一方面，，RUBY 的 IO，数据结构，跟其它语言是一样的，，，复杂，跟底层相关,,

所以，什么是真正最简单的语言呢？也许我们只能从某些侧面和“形式”,“唯度”去说明。我们永远做不到把握事情真相的全部。即使有全世界那么多的纸张，“什么是人生”这样简单的问题我们也坚信写不完。但我们至少可以给出下面这样一些唯度，最最古老的问题,什么是计算机,这个问题跟什么是人生一样简单但是难解,但是我们可以站在开发的角度,给出一个足够简单的答案,计算机=硬件架构+软件 OS，只要了解了硬件架构和软件 OS,那么我们就可以骄傲地说我们精通计算机了,你就可以进入第二个问题,什么是编程,编程绝对是个广泛的话题,但站在抽象的角度，我们同样可以把它归为一个知识：即,编程的根本在于抽象。

看不懂一套源程序，主要是

- 你不知道普通的语句是体现什么样的数据结构
- 你不知道普通的语句是体现什么样的算法

- .....抽象惯用法
- 如何向现实问题靠近抽象并设计的
- 现实问题的复杂性,跟语言逻辑的简单性，非 dsl 之间的矛盾(语言从来被设计成通用的)..

因此要学好一门语言解决问题，不但要学数据结构和算法，现实问题，，精通语言的语法和库，而且学习的另外一点还就在于：抽象和设计，人类软工知识。。。

什么是语言级和语言外要学习的(数据结构与代码结构)

设计最终要被体会到源程序，如果把设计作为整个软工过程，那么源程序是这个工程最好的结果最终证明，(参见《源程序就是设计》一文,你可以 Google 得到它)

暂且你不必理解什么是设计这样的东西，你只需要理解，从设计到源程序，经过了如下二个过程。

- 一是脱离了语言的那些映射，即人们通常说到的设计一词的意义(请参照我的《什么是设计》一文，这里的设计实际上是指大设计，指领域抽象，语言选择，这些东西，包括数据结构)，
- 二是结合了语言的实现映射。即人们通常说到的实现，不过人们通常把 1 作为架构师的工作，而把 2 作为程序员的工作而已。如果架构师的工作深入到类的类部，深入到详细设计，那么他实际上担当了一部分程序员的工作。但在人类的工作中，2 是受 1 控制的，

这里面其实有一个代码抽象模式和数据抽象模式的区别。

类实际上是一种数据抽象，而不是一种数据结构，，因为它将代码抽象为一个一个的“数据模式”，即将 C++这样的通用语言增加 DSL 词汇，让它成为 DSL，可以表达 class cat,class pig,猪猫，诸如这样的领域词汇，所以类是一种数据（把词汇抽象为语言的一级 first class 数据，即 UDT，ADT 这里面 D 的意义）抽象模式，也就是代码模式，而数据结构学是一种实现模式，而不是代码模式。

数据结构学与代码结构学的区别，是解决问题的问题和解决语言映射问题的区别，两者在不同抽象层次，这就是为什么数据结构可以用任何语言可以用基本流程实现也可以 C++的类来实现，因为数据结构学跟它如何用一种代码结构来抽象是没有直接关联的，前者是如何解决问题的抽象（是一种脱离了语言的映射，即我在 1 中谈到的设计），

后者是代码问题（如何面向用了类的可复用目的来进行对具体语言的映射，即我在 2 中谈到的实现,人类的活动中，2 往往处在 1 的末端）。

在编程的实现领域，从来最终都是过程编程即 Core C 的那些指使计算机如何工作的过程式动作很好表现的逻辑〔类体里面也是〕，所以，用了类，只是将面向过程抽象到了一个更好被复用的问题，并没有将如何实现这层抽象到程序员不需要理会，所以对于不懂 C 的人来说，即使它能很好理解 OO，也做不了什么程序〔类只是将实现套了一壳，从而将面向对象层次的复用维持在这壳外，壳的里面却不行，照样是过程编程〕，OO 的强大只是指它类外的那些代码抽象模式。OO 提供的最本质的意义和作用有二，1，将 C++这样的通用语言“在类层次”变成为 DSL，2，正是因为 1 的作用在先，所以普通个人也可以开发复杂的软件系统。

也是 C 程序员的能力更多在前者(数据结构)，而数据类型属于后者，设计模式，面向对象都是后者,是 C++程序员或架构师的事。

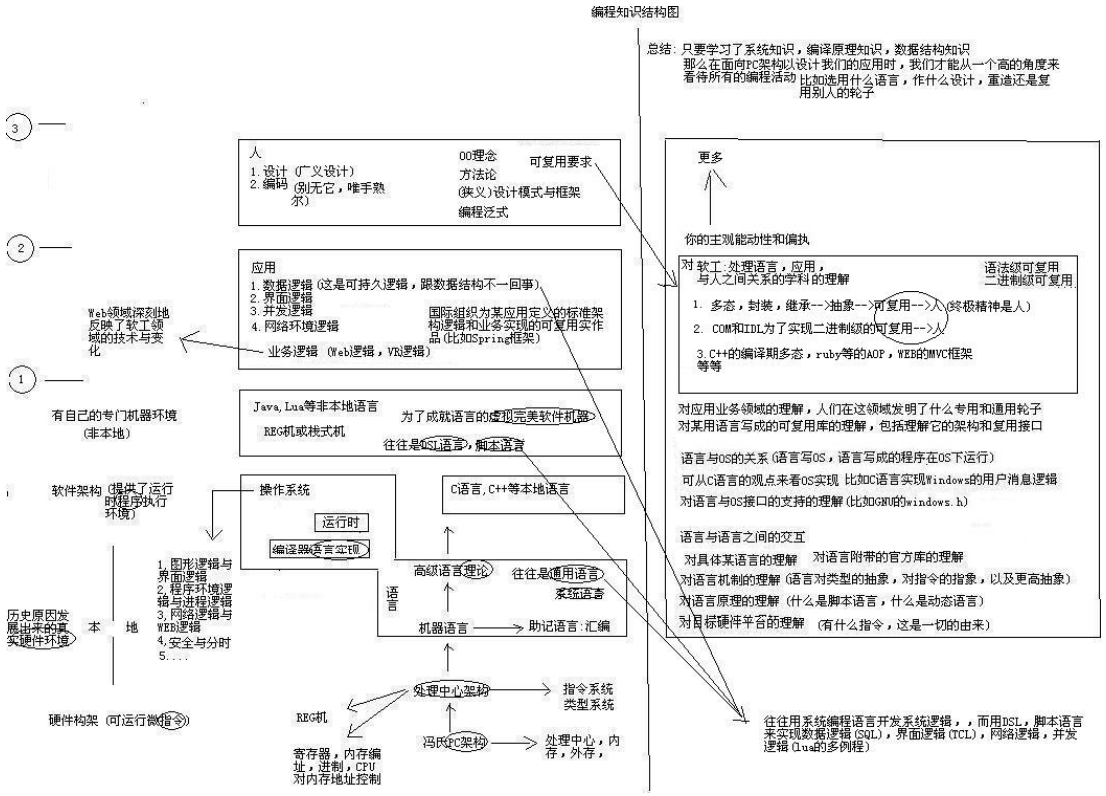
对于 2，即实现。跟具体语言有关。下面举 C 和 C++为例。

遗憾的是，C++只是对 C 的增强而非替代，如果他离开了 C，就只能有类的代码模式的复用能力，却无法保留 C 的强大而直接的实现能力（如果写数据结构这样的东西都要强制用到类，那么除非有必要为复用作目的，否则没有用 C++的必要。），所以一个 C++程序员（假设他用了 00 来实现数据结构）既是一个实现者，也是一个设计者。因为它既抽象了如何能被解决的问题，又抽象了如何能被复用的代码。

程序员对于算法，数据结构，数学实现问题这些细节的实现问题把握最重要，而不仅仅是语言能力设计，代码抽象能力,如果有架构师的工作在先，C 的程序员仅仅需要提供函数过程模块，(极度求接口化和求极度模块化，设计是需求驱动的，接口是要提供的功能驱动的，都是不必要的那是架构师的工作)三种流程控制结构加简单的类型机制,已经能将一切现实问题的解法映射给计算机(在命令式编程语言中，控制结构等等被证明为可以产生一切逻辑。。因此具备三种控制结构的语言都可以成为产生一切逻辑的语言。。),学再多的语言,再多的语言机制,不过学到了更多的映射手段,代码结构方面的东西,真正要解结的问题呢,反而被模糊了,对语言的学习应适可而止，**实际上对于程序员,数据结构学这样的实现逻辑是你应大力学习的。你的主考官都相信你会简单的面向对象写程序和复用,但就是不能确定你能不能用数据结构解结实际问题。**

编程知识结构

什么是编程要学的。首先给出一张图。图中从某些侧面描述了编程知识结构。



学习时所一开始站的难度和起点不同,后来的成就也就不同,如果不一开始学编译原理,那么以后学习编程语言中所碰到的问题都会求助于自己的经验,就会产生形而上学的错误，在类 c 语言中，对于指针等复杂易混淆的语言的细节问题的把握，只要从类型的角度去把握，你就把握了一切。还比如左值右值的区别其实是类型问题，而类型是编译原理，所以左值右值问题应该一开始存在于编译器，

这样才不会从形而上学的多重角度去理解，不但走太多弯路而且易淆，学习中最重要的是这条线.

而本书正是这样来解释指针解释左右值的.我们能想到的联系都是知识,知识是人类多年把简单的东西弄得复杂化的共工，有一些可以被证明为正确,另一些不正确或即使正确却不受人吹捧的渐渐流失而已,,,知识本多维，如果仅想从 IT 去认识 IT,，那么你会看不到全部，也会不知道很多学习 IT 需要学到的其它维度的知识(我实际上说,有些 IT 的问题不纯粹是 IT 问题,并不能直接用 IT 原有的知识来解决,就比如,JAVA 到底比不比其它语言优越,,这是软件的问题,然而它跟人有关,,跟软工有关) 就像有些知识是为了跟人结合而产生的一样(OO 这个东西不仅是技术问题而是软工问题,是为了解决编程方式跟人的关系结合产生的),是为了探索一个领域的东西却总是无法不避象涉及到另外领域的事物一样..

我们现在的编程语言和编程方式,语言提供什么样的语法机 ，人们需要以什么方式写逻辑，都不纯是狭隘的编程的知识,,而是跟程序运行环境和整个软工有关有关的,就像

必须先明白基础的编译原理抽象领域的知识,才能用更好地理解一门高级语言的语法机制一样，而编译原理本身，是跟人密切有关的(比如其形式语言理论是人的理论)

只要学习 C 语言,计算机系统知识,语言实现原理,网络原理这些系统知识（或者只是了解吧,这本书正是极力尝试让初学者达成这个目标）,才能从一个比较大的侧面去看待现实问题对于计算机的实现问题(也即编程问题),,也只有这样,只有懂 C，懂编译原理，懂计算机系统，懂网络，才能从一种大局的高度去看待并计划你的系统你的应用。。

对于计算机专业来说，为什么也才那么几门课程(高数线代离散，编译原理，C 与算法),，因为这些学科是最重要的，是人类把复杂的东西归为某一唯度的产物，比如把现实问题对于计算机开发的实现归结为专门的一门数据结构学(真正掌握了这些基础，你会发现再多后面的技术用语及其背景都是支节，比如，编译原理，操作系统，C 语言，从数据结构的眼光来看，都是同一些东西变来变去的组合罢了)

其实计算机专业学生的那些课，，数结，C，操作系统，都是无比珍贵的东西 ,,学校设立这些课而不是 C++，不是 RUBY，，是因为那些才是可以解释后来一切的底层 ,,而并非一种舍本未未的作法..

当然我并非在说编程一定要精通如上的每一门学科，有人在不明白 STL 原理的基础上照样可以能用它熟练地开发应用，注意我这里说的是会用,人们可以不懂 STL 原理和任何实现却照样可以拿来熟练使用的人大有人在。但那会是有相当局限的，至少，因为语言就是处理三个东西之间的关系,平台,语言,要解决的问题，如果你甚至都不理解编译原理是解释高级语言的实现原理，那么你就不明白类型为什么而来，也就不能进而理解 class 为什么存在，又抽象了什么样的数据本质，以及其它一切一切很多东西，仅仅满足于使用工具的人，永远都不会主动到去创新，那其实是处于一种未开化的状态。下面我浅显地一一介绍下，更多的东西你可以自行 Google

汇编基础:解释了硬件平台,即 CPU 中内置了控制内存的模块,因此要涉及到寄存器,内存地址等,

操作系统课程,如果说汇编基础解释了解硬件平台,那么这就是解决的软件平台的问题, 要学习到的跟底层相关的离散形式,,这是尤为珍贵的,比如进程,比如并发,比如异常,,

而离散数学,,就是一切离散形式,计算机和程序语言环境,和语言本身都是本质上一些离散形式,比如图灵机就是程序模型,是个离散东东,在编译原理中体现就更明显了,比如函数语言实际上就是一种高次方程的离散,

编译原理的本质是什么呢,,,如果说硬件和操作系统都是解释了平台,那么编译原理就解释了程序本身的本质,

那么 C 语言课程呢,,这解释了硬件编程的一些方面(C+大量汇编的形式广泛用于硬件编程,驱动开发),,而且,C 语言这门课程最最重要的意义还在于 C 是过程式语言的代表,它解释了一切后来的高级语言比如 Ruby,比如 Java,的那些基本语法机制,

比如数据结构,而数据结构中的"数据"二字永远是一个程序的中心因素,,从普通数值,字符串,数值到结构体到 OO 数据,体现了人们封装程序开发数据对象的复杂性的要求..而且这种发展是为了产生一种广泛深度的开发方法的,这导致了软工,JAVA 就是这样一种好的语言.

C 语言学科，一般用来作底层编程,比如那些靠近硬件层的,选用普通的 C 语言可以在 Windows 平台上开发控制底层的编程,而且还存在嵌入式 C 语言用来开发各种硬件编程，驱动，ＢＳＰ（主板支持驱序），比如一些智能手机的操作系统开发等，其实 C 语言开发也就是一种范式，一种编程习惯，它是过程式命令编程范式的代表，世界上用得最大的编程语言不是 JAVA，不是ＲＵＢＹ，不是ＶＢ，而是 C，因为 C 用来作系统编程时，它可以提供较快的运行速度，接近底层的更好控制，实现 C 语言的编译器是一种将 C 语言代码直接编译成本地码的工具，不存在任何逻辑中间层次或解释程序（比如虚拟机）因此运行速度很快 而且，C 语言提供的指针等，直接与硬件内存地址啊，这些东西挂钩，而且系统本身大都用 C 语言开发，比如 Windows，可以用 C 语言的眼光角度去解释很多 Windows 的机制，这就在你要开发的应用程序和宿主环境中提供了统一性，你可以更好控制和调用系统ＤＬＬ，不必作语言之间的ＢＩＮＤ，而且 C 语言不必涉及到ＯＯ（因为ＯＯ主要是面对人的）而 C 语言更多地是机器因素，它要求人用机器的思维方式来编程，这句话本身也就说明 C 语言是靠近机器的，因此它适合用来作系统编程，而ＲＵＢＹ等ＯＯ语言用来作面向程序员的高级脚本编程，所谓脚本语言，就是相对系统编程语言来说的，系统编程语言提供了强大的底层编程能力，而ＲＵＢＹ等脚本语言提供了调用这些系统功能的高级应用层的开发..

无论如何,一个程序员是要经常学习新知识的,如果不能快速接受(比如你的基础知识跟不上来),那么基本上你会很累，大学只是学习这个基础知识的阶段,你最好把编译原理,离散数学,汇编程序设计，操作系统，C 语言，这些基础弄得滚瓜烂熟,，并积累一些具体的开发经验,，等出了社会之后，你会发现社会上的编程知识跟你在大学学习的东西差了去了，这个时候你的这些基础知识就发挥了很重要的作用，你需要这些基础来理解软件开发中的大学问，新思想，比如设计模式，ＡＯＰ，ＯＯＰ，ＤＰ，ＤＳＬ，STL，ＵＭＬ，ＬＡＭＰ，ＲＥＳＴ，COM，Ｊ２ＥＥ， 总之钻进去了，也就是一种乐趣..好自为止，

怎么学习 C 和 C++

本书是以 C 和 C++作为全书的范例来讲解的。

鉴于这样的话题太有争执性，本文只给出一个参考意见。

就学习一门语言比如 C 来说。我们最初学习和接确的当然是它的语法语句，综观 C 语言的语句形式，就只有编译原理后期翻译的那些语句形式了，即 Core C 的那些东西。实际上你以为的 C 远远比不上真正的 C 大，你眼中的 C 可能只是真正的 C 的一小部分，

由于语言是由语法定义的，因此我们称，类型，表达式，控制语句形式，赋值语句，这样的东西为语言的要素，打开任何一门关于程序语言教学的文章，我们都可以发现这一点。。对 C 的介绍请参见第二部分第四章。

而字符串，IO，异常，标准库,,数组,与 Windows 的接口,这样的东西是语言的高级话题，和高级功能，不是语法级规定的,，比如可能是库提供的，库与语言的关系请参见以后文章

学语言，我们最终是学到了一种手段，语言的语法机制只是这个工具的规则，语义才是我们所需要的，与应用最为密切的。

C 语言加数据结构算法的方式解释了计算机产生逻辑的根本，只有深克地理解了这些，那些高级抽象就有被准确理解的可能性了,，

要知道，你想要明白 Stl 必须要理解一些 Adt 的东西，这是 C 语言算法和数据结构的一些东西,，C++并没有完全做到 Adt 完全的抽象(没有抽象到你能以一种脱离底层完全不同的眼光去看待数据结构的 Adt),，这不是 C++的错,，这是你没能学习算法跟数据结构的错。。

不要依赖太多的抽象，因为它们根本做得不彻底，你依然得学习算法跟数据结构才能理解这样的 adt.

计算机底层=C 语言加数据结构，算法,,,,这二门学科刚好完全地归纳并解释了编程对应于计算机底层的方方面面。。其实 C 语言很简单，基础的指针用法也很简单，如果学习了 C 语言版的数据结构，因为这是在学习 C 的习惯用法,，因此指针的很多抽象用法也会学到,，也就同时学会了 C 语言和数据结构,,，因此学数据结构是学习 C 语言最好的方法。。

深克地理解了算法与数据结构，你学习编程才算到了家。。因为你真正学到了用通用的编程语言解决现实问题的方法（离开了数据结构你当然可以写程序，但几乎没用，一个离开了数据处理的程序能有什么用？这就相当于学会了使剑，剑的套路，却没能学到功夫一样，这就是为什么你要学数据结构的道理。）。

用 C++进行开发要学到什么程度,需要什么知识最小集呢,当然要根本目标问题的不

同决定不同的复杂度,但语言和工具级的复杂度都是一样的

要成为某领域能实际胜任某份工作的程序员，就要做到精通四个“ldioms”(注意这是精通)

- 你要用到的语言和 IDE 的“ldioms”(一门语言，一种开发库)---编程首先就是会用一门语言和它的库

- 数据上的”ldioms”(数据结构-数据的内存模式，数据库-数据的外存模式)---编程第一个就是数据，想起 DOS 下的编程了吗，一是数据，二是代码
- 设计上的”ldioms”（面向对象，设计模式）-----编程第二个就是代码或代码框架

- 以上三条都是前提，那么这第四条就是最终的你要实际涉入的业务领域的”ldioms”---编程最终是为了为这个领域服务

以上四条是主干(最好按 1-4 的顺序精读)，而其它的都是支节。比如工具的使用啊，XML 啊，UML 啊，XP 方法啊， ANT 部署发布知识啊等等

那么真正要掌握 C++进行开发,你需要掌握那些语言级和库级的知识呢
1,至少熟练一个 IDE,make,install,,调试,等编译技术,能在编译期出现错误的时候能搞明白基本的 I 关于 DE 环境配置的错误
?2,在语言级和库级要明白的就多了出去,比如对 STL 理念和当中每个 LIB 的理解,对指针进行理解才能明白诸如函数指针,指针当返回值以返回一个数组等机制,,,当然还有很多很多 C++的惯用法,等等,这是主体部分, 3,要明白你使用的第三方库的一些知识,要了解它们 OO 的架构,一个字,要达到一种能用 C++使用它们的接口的能力就够了,这就是 OO 语言宣传它们的资本..又一次,你只会使用就够了,不必懂得库的 OO 架构,,你需要了解它们透出来供你使用的粗略架构模型和接口就行了.
4,在复用方面,要明白设计的一些知识,知道 OO,GP 这样基本思想,知道你的应用大家都用 OO 作了什么设计,你所使用的库用了什么样的封装上的设计.OO 并不仅仅是封装,封装是为了复用,,因此 OO 最终就是为了复用,封装只是中间过程..就像接口并非用来封装一样,而是用来抽象,,一切都是抽象..
5,在开发自己的库和逻辑方面,要明白应用域的一些知识,这样在设计时就能知道要构建什么样的架构.用什么模式来设计等等,用什么语言细节来实现,等等
6,要尽量熟悉以上,多练手,才能快速打出代码,,,,要记住,这个过程很自然,,就像你学好了英语的语法,再多看了一些英语的文章,那么你就可以写出英语文章来了.一切都是习惯用法,和语法游戏,,,除此之外,编程中其它的一切就是设计问题,而不是编码问题了,,,设计的问题是其它领域的问题,比如算法设计,,而不是编码问题(有些书用鸡兔同笼这样的问题来放在 C 语言的书,这对学习 C 语言本身的语言机制如流程控制有意义,数学问题的复杂性只对研究算法有用,对解释 C 语言本身无任何作用,而算法是设计通用的,不跟语言相关)

一切在于多看,多写!一开始不要写逻辑和架构过大的逻辑,在懂得设计时就可以接确了.
设计本不存在?当你快速写代码时你根本不会觉得设计的存在,,这是因为编码就是一种习惯,而设计就是一种关于要写出什么样的逻辑的设想,用什么编码方法来体现,,,设计就是对编码进行控制和计划,,这里就是编码跟设计的关系,,难的不是编码,因为所有人都可以学会语法,学会写作习惯,,,但是设计样的文章却千差万别.;.
?设计不仅是算法设计,而且是复用设计

也即,我们站在跨度太大的抽象间进行一个抽象到另外一个抽象的理解,因为这个抽象太大了,在另一个抽象发生时,那个最底层的抽象可以显得忽略不计,计算机可以记住那么多抽象,而人不能,,就像这个世界已经发明了哲学一样,难道人行事做人动不动就会求索哲学去理解事情吗?问题出现了而过不去，你会发现再高深的哲思也不过几条丑陋的信条,所以哲学有方法论,但是作为哲学的方法论是大抽象上的,并不会深入到指导每个细节,当方法论脱离细节即脱节时,就没有太大的意义

要学习编程,我觉得 C 语言是起码要学的,即使是以后学 LUA,RUBY 这样的高阶语言,最好的途径还是先学 C，比如学习编程必定最后要涉及到系统逻辑,那么是首先学习上述的计算机专业课程的知识,还是先学语言呢,,,,我觉得是先学语言,,因为理论的知识在没有实践的前担下很难掌握,,而一开始就学习语言(我这里只指 C 语言)是最好的办法,,,因为在学习的过程中,你需要一门语言来实践,,,验证各个小问题的语言实现,来掌握更多的知识，我所推荐的学习方法是先学 C，再学数据结构，C++只需要学习其 OO 和 STL 就差不多了，最好严格按照这个顺序进行。

编程能力，代码控制能力

你应该知道，真正的编程能力不是使用轮子的能力（即不是复用能力），而是将现实问题用编程语言来解决的能力(这就需要你具备设计能力，即很强的代码控制能力，能把现实问题抽象为代码而不仅仅满足于使用别人现成的库)虽然复用能力与设计能力都貌似语言能力，从语言的眼光来看，复用能力是属于设计能力的，但实际上他们还有很大不同的要求有不同的语言能力，有些人具备在不懂 STL 原理的基础上使用 STL 的能力，但它们就是不能用 STL 来实现一个 STL。。这就是上述二种能力的根本区别的证明。

（往往我们把复用能力当成编码实践能力，把设计能力当成广义的编程能力）。

有时难以理解的绝对不是代码本身(语言的语法和语句有限)，难理解的代码是其中体现出来的设计元素和应用元素..也就是所谓的算法和设计,,有时是设计模式，有时是现实问题模型的相关抽象(其实算法和数据结构也是计算机开发中历史形成的抽象,，即某

个“现实问题”，但一般将它们独立出来)。

要知道，程序的编制者跟程序的使用者有时是不一样的，这集中体现在 C++的模板的复用上，C++的模板是用来书写高抽象库的好工具，然而类型的泛化意味着通用，因为像 STL 一样的东西意味着它能并适合于提供某种抽象上的规范（接口），，我们使用 STL 就不必处处屈就通用了。我们只需要“使用”，（我们实际上也不必提供太多的轮子方面的东西因为我们正是使用轮子，因此只需开发出使用这些规范的实践，实例），就跟游戏引擎和游戏的区别一样（面向通用的库或面向实现特化的库都有）。。因此这二者涉及到的模板编程技术是有高有低的。会复用并不一定会写。当然，理解 stl 库的实现过程中使用到的模板理念显然可以帮助我们复用这些规范（接口）进行实例化（使用接口）的过程。

实际上，数据结构，设计模式，现实问题这三个东西跟任何一门程序设计语言都没有关系，因为任何一门冯氏模型决定下的开发语言都满足这些东西，，而且计算机界的一切软硬基础都是算法加数据结构的集中体现（再往人类的软工一点，就有设计模式了），

我们编程往往涉及四个能力
1，语言语法的，比如流程控制，数组，IO，字串，
2，数据结构和算法的。
3，设计模式的
4，现实问题的。

用某种语言来编程，体现的代码控制能力只有 1 是必要的，然而具备了 1 并不意味着你就能定出某种数据结构，或设计模式，，或抽象出某个现实问题，，因为那根本是另一些领域需要你掌握的东西（而不仅仅是语言能力）。所以你汇编码能力并不意味你有编程解决现实问题的能力。你有编码能力所以一定有复用能力，但你不一定有实现这些轮子的能力（以及阅读这些代码时认识到这些轮子设计的能力，这样你阅读代码的能力和眼光也会受到限制，只限于简单的复用能力，因为复用能力只要求你拥有接口复用能力，，真正的逻辑不只是接口间的简单复用，而是接口间复合形成什么样的抽象以及如何逻辑上形成这样抽象，这就不仅仅是接口复合作用了）。

所以真正的编程能力在于 2，3，4，加 1 编码能力 ,,所组成的综合编程能力;

程序员的能力模型，语言 30%,数据结构 50%,对现实事实的抽象理解能力 10%,,设计模式能力 10%。。 ==100%。

真正的编程能力是设计能力！！对现实问题，思维模型的学习！！而非对细节，对平台编程！！事物的 OO 解只是事物解空间中的一种而已！！

本书目录安排

宜交叉来读本书!!本书的架构是自成体系的，除掉第一部分(书前附录)和最后一部分（书后附录）外，中间三部分是主体，这三部分自成体系，其中的每一章的每一节都专注于一个主题，既有知识架构，也有学习方法，技术细节的描述，下面一一讲解：

第二部分《导论》

这一部分主要是关于编程的导论，这一部分过后，你基本上可以看到编程界常见的那些概念的解释。

- 第一章《系统》，这一章从抽象角度解释了 PC 系统的架构，首先指出整个 PC 系统的形成是一个不断由机器向人抽象的结果，而抽象其实是统领整个计算机领域的，在机器内部，先是图灵机作为最底层计算模型，CPU 的控制作用在整个硬件内部是最高的，然后在机器外部，OS 将硬件抽象为软件，于是所有的开发问题由机器硬平台编程转化到软平台上了，开发问题首先是解决现实问题与系统有关的那些逻辑，即系统编程逻辑，然后指出命令行和 GUI 都是系统编程中的表现抽象（是设计问题中的表现抽象，当然 GUI 也可以是应用逻辑里 standalone 的），另外，本章还介绍了操作系统与语言的关系，即系统与开发的关系 (细心的读者会发现这正是慢慢过渡到第二章对语言的专门介绍作准备)，进而提出了虚拟机，虚拟机提出的必要以及它是如何导出虚拟机语言的，虚拟机是为了成就语言的。

  这章中重点介绍了 CPU 与内存的管理作用，突出了 CPU 在整个 PC 中的控制作用及这种作用导致下的冯氏模型对开发的影响。CPU 的那些沿袭到些语言机制的特性。比如异常与跳转。

- 第二章《语言》 这一章承接第一章，如果说第一章讲解了到了计算模型和系统模型，那么这一章主要讲解基于前二者之上的语言模型和开发模型。

  编译原理，即高级语言原理，它解释了很多语言被设计出来的原理，一如既往地，在这一章中，我首先指出编译器是一种抽象，因为它使人们的编程工作变成一种面向行编写源程序的工作。

  因为同时承接了语言对于机器的实现（编译后端）和对于人的应用需求（语言的语法定义和设计，编译前端），在讲解中，我重点澄清了一些至关重要的概念而又不深入到太多编译原理的技术细节。

  比如在讲解对编译后端与编译前端的区别时。我指出语言语法机制只是抽象，

它可以抽象出 Rope，还可以是其它东西，所以它跟设计有关，编译后端才需要我指出语法前就是设计，一旦进入语法，还是设计，只是编译后，就进入了运行期了，这二个阶段是有本质差别的。这种差别使我们可以明白一些东西，比如字符串这样的东西更多地是一种抽象，C 语言可以有指针来实现，C++可以有模板来实现，模板实际上是靠近设计期的，

在讲解运行时时，

再者，指出冯氏下编程语言其实都是以类型抽象为中心动作的语言，C 是，C++是，Java 也是。然后再详细介绍了类型。最后介绍了脚本语言

- 第三章《语言之争》

这一章力求澄清初学者对语言选择方面的疑惑，从应用，人，语言这个根本的角度，首先说明了语言本不存在功能上的差别，语言功能的差别只是它们各自的表达能力范围内能解决的不同问题而已。所以语言的选择根本是受应用左右的。对各种语言的优劣比较只是在有限的范围做的狭隘的事情。

接着介绍了 C，C++语言团队之间的一些矛盾。指出矛盾的根本是源于语言间的过度抽象。使得从 C 抽象过渡到 C++抽象的各个阶层的人群大为存在。

再者，提出了第四代语言 Java，指出 C，C++没有被流行的 Web 作为开发语言，是因为使用 C 开发规模跟不上，而 C++又存在太多语言陷阱。就系统编程的眼光来看，一门语言的 I/O 最能体现它开发 Web 的能力，然而，语言对应用的适配只是狭隘的一面，Java 虽然有很强的 I/O 能力但并不是它决定了工业界采用了它，其实反过来正是 Web 开发促成的 Java 的流行。

- 第四章《最小内核语言（C)》

如果说这章主要是介绍 C 语言的，实际上不如说它是介绍整个过程式语言的。

首先介绍了 C 与 C++的根本区别在于 C++对 C 的那些增加部分，而非对 C 的保留部分。即 C++的 OO 和模板导致的泛型机制这二种机制，而它们与 C 过程式解决问题的方式迥然不同，所涉及到的语言抽象能力是处在不同抽象层次的，C++相对更靠近人，C 是用指针，过程语法机制这些底层来呈现应用逻辑的，比如字符串，但 C++是用模板，OO 等代码抽象来呈现的，这使得程序员能更好远离底层更专注他们的应用，这是这二门语言的根本区别所在。

然后指出，在利用数据结构等实现抽象来解决现实问题时，C 好像只有这么一种手段（比如它不像 C++一样能很好表现设计模式），而其它第四代语言，不但

有数据结构加算法的方式，而且还有更强大的语言内抽象机制,比如代理。闭包。

然后从类型机制开始，浅而扼要地谈了 C 语言各个方面的东西。当然，这一章作为介绍 C 语言的章节，重点介绍了指针，并指出“指针是 C 语言的一种抽象机制，而不仅仅是作为操作底层的工具”，这就将指针上升为 C 语言的一种语法机制，把它上升为跟 OO 这样的第四代语法机制的范畴。而不仅仅指针就是一个操作符，相比 C 语言其它语法机制是固定的，而指针是可以加以丰富变形形成其它语法抽象的。这样就抓住了其本质，

- 第五章《抽象与设计》,（时代真的不同了!！以前是手工作坊时代，现在是软工时代）在这一章中，我会写跟软工密切相关的思想和学习 IT 知识，以及任何一切知识通用的思想，，比如抽象啊，维度啊，范式啊，这对解释以后 OO，都是必须的基础，软件工程的相关概念，是编程入门的关键，对他们的理解是必须的，才能走出为程序而程序的狭小境界，，在一个更广阔的思维空间里进行考虑问题和编程。如果说《系统》是单独讲解软件环境，《语言》是讲解语言，那么这一章这就是讲解软件，语言，应用，与人的关系了，一言以概之，这就是软工，在这个大活动中，涉及到如何对多人合作的真正工程级的大系统大问题的解法，需要考虑软件作为产品的可扩展性的时候，，就需要提出设计和代码结构这二个字眼和关于它们的一系列的知识，这根本上是基于软工是为了让软件产品适合“应用”这个需求而来的，软件写出来的作为产品的特质，(界面与逻辑分开啊,数据分开啊什么的)，，这就是说“构架不是功能的要求，但却是工程的要求”(这里的构架是指代码抽象，注意这几个字) 而这是很多书本都忽略的。

- 第六章《实现抽象之数据结构》根据我们在前面“最小内核语言 C”那一章提到的，过程式语言的本质就是数据加处理数据的方法，写程序的过程就是操作这二个东西,,手工作坊的写软件时代，，就是编程=数据加算法(这句话过时了，因为过程时代，只有 Core C 的那些代码逻辑，而现在代码逻辑增多了，这句话应改成编程=数据加算法+代码逻辑),现在软工时代,编程=数据加算法加架构,其实算法是比架构更广的概念,它侧重于指"功能，解决问题",而架构侧重指"扩展功能，扩展源程序",但这二者都是设计考虑的范围,,说明白点设计就是编码前加编码后的总称，设计即源程序，源程序是最终体现,,,这也是一切高级语言在机器端的实现形式，一种语言要最终被体现成数据加语句的形式，那么创造这种形式的语言本身也要提供这二重机制,,这二重机制就是数据加操作数据的方式，数据由原来的数学，字母，这些计算机硬件能直接处理的东西，变成了 OO 的对象,而处理数据的方法，则变成了架构加算法

- 第七章《代码抽象之高级语法机制（C++）》
  这一章重点介绍了 C++，

- 第八章《代码抽象之设计模式》有人说根本不需要设计模式，有人说设计模式是 C++程序的“补丁”,，这话说错也可以，说没错也可以,,这是因为狭隘设计模式之说就是为 C++，JAVA 这样的静态语言作补丁（重构时可以很好地改变当初的设计）,， 而广泛的设计，包括设计模式所提出的那些东西，广泛的设计是一种意识内你要明白它是广义设计,，并主动引用设计模式来进行设计的动作。所以并不是一种补丁。

以上我说的都是“要懂得的部分”,,，一个程序员只有懂得这些，才算是理解了软工的常识,,，否则一味钻细节，什么都不知道，处于一种很可怜的未开化状态,，更要命的是，不知道何为设计，也就不可能主动领会别人代码或库中的设计之巧,，仅满足于了解算法实现的人是不行的，这样的人，由于一开始不考虑软件作为产品的特性,，最后写出来的东西生命力就不怎么样，事物之间的联系很重要，你在学习的时候，意识到了计算机和其它东西的联络，然后你构建了这种关系之间的系统，就自成一个学习体系了，这是有价值的地方 而且你非了解不可!！因为除非这样，你才能有效地学习,,否则迷茫于庞大的细节,，巨细不分，也就不能界定概念之间是什么关系,,，这是十分造孽,,比如《系统》这一章,“为什么要出现编程语言”,,“我们看到的编程语言到底是什么东西”“它们下面是基于什么离散数学构建的抽象”我把一切功能的达 成的中间层,，无论是软件，还是硬件，都看成抽象,我们今天能看到编程语言为什么要求我们用语法,，而不是直接在语法树上写程序为什么 Lisp 语言就能 直接在抽象语法树上写程序，而不需要一个语法,,，为什么 Yacc 是一个没有语法的词法编辑器 解释了为什么“我们要用流程控制，逻辑转移”这样的方式写程序,而很多人并不知道，他们学习编译原理，就是为了学到这个知识,这是死读书的人多么大的损失啊,,而我打算择其要点,，组成一条线,，解释语言的由来,，浅而不缺地解释很多很地道的东西,有机地讲解,，择其重点和必要的，去除过时，历史复杂性

第三部分 C，C++代码阅读与控制

如果说第一部分第七章只是从需要懂得 C++哪些方面这个层面上去阐述 C++的话，那么这整个的第三部分力求向你详细讲解 C++的几大语法机制，以及标准库。

- 第九章。(要精通所用的语言的语法语义) 我们要编程（如果说第一部分是要懂得的原理，那么这里就谈实践细节，当然要求精通），当然首先用一门语言，我打算举 C 和 C++作为例子，因此语言 和语言库这一部分重点是说到了这二大编程语言，学语言只是学语法，我们编程是用语言所带的库的，C++有 STL，BOOST，JAVA 有 JDK《J2EE,，所以库是不能不学的,,因此学一门语言=这本语言的语法+实现了这个语言的某个开发库,,而且程序=算法加数结也是不符合现状（）我接下来要谈到的《语言和语言库》这一节讲解了 C++,JAVA 和,,JFC 言和类库相关的知识，要实际编程,，掌握一门语言和库是必须的

- (要精通处理什么样的数据) 《数据结构与数据库》（编程与数据）计算机就是用

代码来处理数据，

- (要精通写什么样的代码或框架) 《算法与架构》（编程与设计）写什么样的代码，，，不再是写什么样的实现，什么样的算法的问题了，，，而且还是要体现什么设计，，构造什么架构的问题，，如果说面向过程的写代码就是注重设计算法，，，那么在 OO 这个注重可复用（而且现在这个年代，我们大部分情况下是一个开发者而不再仅仅是一个实现者，我们的确经常是使用外来库来构建应用）的软工时代，，而且要懂得设计架构，在本节的最后讲到《J2EE 大领域学》J2EE 是现在最流行的，，OO 以后的范型和问题集，因此对它的学习就代表对大多数计算机技术的学习，因为第一部分到这里为止的基础，，所以我们这里可以大致描 J2EE 的整个模型
- (要懂得待处理问题域的细节,为第四部分作准备)《游戏编程与虚拟现实》从这一部分开始讲解多媒体编程和游戏引擎开发的细节，，为下面一部分做好充足准备。。

第四部分 一个综合的例子

这部分就是讲解如何用范型来解决实际问题，最终产生一个可玩的游戏。。注重了架构的学习

《设计----GVWL1.0 开发过程》
《编码-----一个 NotWar3 核心 Demo 的实现》
.作业与重构

机器,平台与网络揭露了软工所处于的三个环境，WEB 编程越来越流行了，这就是网络上面的一层抽象(internet 是 internet,web 是 web)，编译原理其实有很多方面跟汇编有关，这就是跟机器编程有关，，平台 就是 OS，，OS 就是一个编程层，就像脚本机一样，，，，我们 native 编程是对本地编程，这里的本地就是指 OS，，而虚拟机是可以不需要操作系统就 能运行程序代码的层，，这是另外一个软件运行环境 总结起来，，，这机器，平台与网络讲解了很多“软件运行环境”，特别是我在编译原理讲解中涉及到的机器部分，是最底层的软件运行环境，对设计虚拟机也是 必须要用到的知识，，对解释 OS 的实现又是必须用到的知识，这三个环境是天然不可分割的，，站在软件的角度来看

而且，其实 C++和 J2EE 作为编译语言，是静态系统语言，因此有设计模式的出台，，，而动态语言就不需要设计模式，因为它是运行期语言，类作为数据结 构的定义在运行期可以被动态改变，，根本就不需要编译期 cast 这和《C++新思维中》“编译期策略”是一个道理，C++只能提供“编译期动态”以实作出设计模式，而动态语言由于它的性质不要做大而全的超级设计，，虽然大而全正是设计的本质意义所在,,,总而言之，设计的意义是泛义的，但是人的掌握范围之内就得因地制宜，就是多范型设计，比如类形式，，用一种类的形式综合体现了数据和操作数据的方法，，这就是 OO

注意各个小节深红色，加粗的字眼。那往往是揭示本小节主题的句子。

搞清了以上这些，那么我们现在可以进入第一章第一节了。

对你死亡级的提醒，请不断写代码唯手熟而的方式是阅读代码，成千上 W，并实践!!!

第二部分 基础篇：导论

第 1 章 系统

1.1 何谓 PC

计算机俗称电脑。我们平常谈到的计算机就是指 PC。

当然，计算机不光指 PC，还可以指服务器，大型机，我们无意去为分别这里面的区别花费时间，这纯粹是一种历史叫法。在不加特别说明的情况下，本书作为编程教学书以下部分我谈到计算机就是指 PC。我们是对 PC 编程。

PC 的概念更多地是一个 历史概念而非范畴概念,历史上第一代(注意并非第一台)PC 是指 1981 年 8 月 IBM 推出的第一代个人计算机，称为 IBM PC，它配备了 intel 的 8088 CPU 和 Microsoft 的 MS-DOS,从这里开始,Intel 确立了它 PC 行业的 CPU 霸主地位，微软也慢慢成为了软件领域的巨无霸。Wintel 组合出现。

当然，IBM 是卖电脑的，Intel 是做 CPU 的，而微软是搞软件的，这三者业务领域不一样。但电脑就是从这个时候开始进入大众生活的，在这之前是一些巨型机，科研设备机器，以及个人爱好者研制的雏形所谓“计算机”。正是这样的发展，带来了计算机向 PC 发展的态势(硬件上的，软件上的，总体架构上的)。

关于这其中的种种历史，存在有一些趣闻，比如:

- 【硬件上的】:

第一台 PC 其实是由一个叫毛利的人发明的而不是教科书中普遍谈到的那个巨大的家伙。

- 【软件上的】:

苹果图形界面拉开了计算机图形时代的到来。

50,60 年代某段时间一种叫 CP/M 的操作系统差点取代 MSDOS 而使它所属的公司成为 IBM 的合作伙伴。

MSDOS 实际上只是一层壳。它是改 PCDOS 来的。而 DOS 实际上并非微软所产，它的作者早就死了。

- 【架构上的】:

CPU 是有架构的， AMD 和 Intel 主频之争从来都没停过。

1.2 图灵机与冯氏架构

现代计算机模型的产生源于对自动控制(自动机理论)和人工智能(机器人)的研究。

图灵开始研究人工智能的时候那时计算机尚未产生，他的图灵机是对计算机领域的贡献但其实更多的是对人工智能的探测。

图灵机是一种通用自动机器的模型，这种机器有极其简单的形式（因为要能易于实现），然而只要产生足够丰富的变形它可表达并产生一切逻辑(计算函数，表达其它离散算法逻辑等)，图灵机在理论上是理想的，它有一个二端无限沿伸的纸带作为存储装置，输入，输出和状态转移函数是一个机器的三要素，这三要素组合并变形可成为一切机器的原型，可解决一切图灵机能解决的问题。因为它揭示了计算机在抽象层次运行的本质即形式计算。所以图灵因为他的这个贡献被称为计算机之父。各种形式的自动机理论随后在那个时代发展开来。

图灵机是被严格证明的。虽然它是一种抽象机难于直接说明，但正是因为这种抽象可用一套形式化的东西来表示并证明其成立(图灵机的运作原理被称为形式计算，是离散数学自动机的一部分)..通过测试的机器就叫图灵完备，，所有通过图灵完备的机器，这说明它们可以产生等价的算法。。可以解决同样的问题。

图灵机的意义是伟大的，因为它是抽象的，所以它能被实现在很多不同层次的地方，比如大到语言识别器，虚拟机模型，小到自动售货机等等。。

如果说图灵机阐述的是一种泛义上的自动机，那么冯氏模型就是专门针对计算机的自动机理论了，以前的机器(在可编程电脑出现之前)，指令是硬化的，要为某机器编程相当于重置整个机器。

冯氏机的精神在于"指令存储，顺序执行"，在冯氏模型下，执行指令的机制和存储指令的机制是分开的，要换一套程序不需要更换指令系统，因为程序是被内存存储的，取指令器是从 RAM 中随机取出（某条活动指令总是会在某个内存位置待命，随机的意思就是直接指定内存地址并从里面取码进行执行），并不用预先在内存位置固化程序，然后通过控制和地址总线交互存取到内存，这样 CPU 只负责执行，对于不同的程序，它可以只用一套指令系统。这样的架构就形成了一种执行指令加调用数据(指令数据在内存中的位置，或编码了的数值和字符)的方式，然而同图灵机一样，这种简单的抽象形式同样可形成高级的具体逻辑。

这样一来，我们编程也成了书写指令和指定供指令操作用的地址，指令内含内存地址为操作码，而如何在内存中组织数据又发展出了数据结构，用底层的观点加数据结构的观点来解释现实事物的解法就形成了算法。(当然，计算机的观点是如何寻址和执行指令，实际上在没有高级编程语言理论之前，远远还没有算法的概念，因此不能说指令是语句，也不能说地址是个变量，地址里面的东西更不能称为数据类型，CPU 能直接执行和认识的只是由 0,1 表达的指令和编码的地址以及各种基本数值和字符，只是后来以人的眼光来看可以无穷地把 0,1 编码成其它各种逻辑而已，不过在汇编语言级别，这些都不是机器能直接理解的)冯氏机以及它提出的“指令控制存储并串行顺序执行”的理念，这些都深克影响了我们这个时代的编程工作。

- 我们注意到冯氏机主要处理数据和代码，而指令是控制数据的，这使得**我们后来的编程工作的抽象焦点在于数据和语句。这是冯氏模型对编程工作最大的影响**。

- 我们注意到指令是被串行执行的，这说明冯氏模型一开始就被设计成为非并行计算机，因此在后来编程中，要实现并发逻辑需要从软件上和设计上去突破。

- 其它。。

1.3 计算机能干什么

我们要首先懂得计算机能干什么，它的功能发源主要在哪里?这样才能懂得编程的出发点和目的。这就不得不首先谈谈计算机系统的组成。

首先，CPU 是 PC 中的总控制中心，在前一节中我们谈到 CPU 执行的过程就是一个不断取指令并译码的顺序过程(所以本质上它其实并不智能，PC 只是一台高速自动机而已)，其实 CPU 不但处理它自己的事情而且是作为整个 PC 的总控制中心存在的，CPU

不但接管了内存管理，还接管了其它硬件，比如它把显存归入它的管理统一跟主内存编码，而且把硬件消息归为它的中断并把其中断处理程序统一跟内存编码。CPU 在执行中(有一个时钟发生器指导 CPU 不断地运行)不断变换其状态，输入输出则是这些二进制（因此整个 PC 做的工作都是某种意义上的 IO）。总而言之，冯氏架构中，CPU 统治了整个机器，内存二把手，其它硬件则是被它统治的。在 CPU 眼里，一切都是内存地址和指令(这就对编码工作提供了无比便利的条件，使得冯氏下的编码工作变得无比统一)。

CPU 中最主要的二个部件是控制器和运算器，计算机之所以被称为计算机，是因为它提供了形式计算，还是 CPU 里面的逻辑和算术运算呢，当然是前者。实际上运算器和逻辑器的地位就像协处理器一样，是 CPU 不必要的组件，是 CPU 扩展它的功能加的电路板。这些硬件功能固然是计算机功能的重要组成部分并影响对其的编程工作，然而并非全部。

计算机逻辑主要发源于二个地方，这决定了他的功能和对于编程的意义所在。

- 首先，计算机各个部件都是有功能的，比如浮点器可以表达浮点数并处理，计算机籍此可以直接表达硬件级的浮点抽象以及用浮点表达更大的编码了的抽象（实际上也存在软件的浮点处理器），这就为编程引入了浮点逻辑。另外一个道理，你不能编程指使 PC 为你做“给我泡一杯牛奶”之类的事情，因为它不是硬件上能提供的功能和逻辑所在（即使是抽象也得不到），只有当 PC 真正接入了一台牛奶机之后，你才能编程实现诸如泡牛奶的硬件功能。

- **计算机的功能不但在于表达硬件逻辑，而且更大的地方在于表达广泛意义上的应用逻辑。**我们可以对 PC 编程创造一个游戏，虽然图形功能是来源于显卡的，但是游戏世界的逻辑明显不是硬件的。Web 应用就是这个道理。

在机器级别编程，我们无法表达高级的应用逻辑，因为 0,1 能直接编码的逻辑是非常有限的，只有当 PC 发展到提供了操作系统，进程等高级逻辑之后，我们才能在一个新的工作起点上构造更强大的应用逻辑。

所幸的是，当编程发展到高级语言阶段之后，这些机器的细节都被完全抽象化了。

1.4 内存地址

既然冯氏架构就是将执行指令的 CPU 和存放程序的内存分开的一种架构机制，CPU 中集成了对内存的管理，在 CPU 的眼里一切都是内存地址，而且这导致了冯氏模型下编程的统一性。那么 CPU 是如何与内存发生关系的呢？学习这个有助于我们理解操作系统诸如这样的软件的系统逻辑是如何在硬件基础上包装出来的。

CPU 与内存如何发生联系被称为 CPU 的存储管理机制，CPU 管理内存的硬件是它的地址总线和数据总线(我们知道 CPU 虽然自己处理数据，但他同时还主要做着一种 IO 的动作，CPU 管理 IO 的硬件被集成在主板上一块被称为芯片组的地方)，其中地址总线是负责寻址机制的通道，而数据总线表示 CPU 能一次性处理的数据通道，地址线数与数据线性往往长度不一，这就导致了 CPU 的寻址跟处理产生了一对矛盾，地址线决定了它能看到(CPU 能使用到的)和管理到的内存总量(物理地址)，而数据线决定了它能一次性处理的数据长度和能表示的地址形式(逻辑地址),,,这就是表示(逻辑地址形式)和实际内存(物理地址)上的矛盾。

撇开其它因素，我们就事论事来讨论这个矛盾，20 位地址线可以使用 2^20 个最小内存单元即 1MB 的内存空间(这就是说使得这种 CPU 的一个任务理论上可使用至多 1MB 的线性空间，因为它只能寻址到这么大的地儿)但 16 位 CPU 中的寄存器只能表示前 16 位（CPU 是用寄存器来表示地址的，这就是说虽然 CPU 能看到整整 1MB 的空间，但他一口吃不下其中的最小一个单元，因为它首先都不能直接表达这个单元）。因此 CPU 要表达和要用尽这 1MB 的空间，不能以直接线性对应的方式来表达。除非数据线多于或等于地址线。

间接的方法就是设置另一层抽象。可令 16 位先表达一个大小为 64KB 的段，1MB 的内存刚好等于 1MB / 64KB 倍数个这样大小的段。在这种情况下，内存就不再是绝对线性的了(当然，实际上所有内存都是线性的，这里说的是分段之后的逻辑情况下，而且你不用担心 CPU 认不认识这样的划段法因为它的确认识,下面会谈到一个实模式的东西)，而是被划分成了一个一个的段(在段内才是线性的)，16 位用来表示段内的每个最小内存单元，空出的 4 位不再用来表达内存本身，可以用来表达段本身。

以上讨论的情况是 8086CPU 在实模式运行的存储管理逻辑，即从逻辑地址(CPU 要形成任务用到的地址形式)到真实物理地址的编码（实际机器有的地址），这中间要经过一个变换，CPU 负责这个转换。无论在多少长度的地址线和多少长度的数据线产生矛盾的情况下，都会由 CPU 负责这个转换，不过 32 位数据线的 CPU 转换方式要特别复杂而已(特殊的分段方式再加一个分页的方式，段寄存器不像实模式那样用来实际存储物理地址的线性表示，它只用来实现硬件级计算最终物理地址的一个中间存储)。

在 32 位 CPU 出现之后，寄存器能表示的逻辑地址早就是 4G 了，而地址总线超过了 32 位(除非地址总线少于逻辑能表示的数量才需要实模式那样的分段,然而 32 位 CPU 下并没有产生这样的矛盾因此可以以线性地址的直接表示方式来表示逻辑任务线性空间，然而 32 位 CPU 照样也实现了一种转换机制，这是因为它需要实现更强大的保护模式而不仅仅是完成寻址。

综上所述，逻辑表示是寄存器能表示的地址形式，，真实地址是系统装配的内存量，而**线性表示是 CPU 用来形成任务的任务地址。统称为任务空间。不跟硬件地址相关，也不跟逻辑表示形式相关，这完全是一种独立机制的编码**，32 位 CPU 下，一个任务的

线性空间表示总是 4G（注意总是这个词），只是一个转换机制会负责这逻辑地址到线性地址的转换，，然后又有一个转换机制负责线性地址到真实物理地址的转换，程序员完全不必为线性地址过少和这中间的转换而烦恼那是 CPU 必须管的事，，否则 32 位的汇编工作几乎要手动驱动 CPU。

明白了以上这三个概念，让我们再来看下面的章节。

1.5 分段和分页以及保护模式

32 位的 CPU 利用它的转换机制可以保证 4G 空间一直存在内存中(这就是说，实际上并没有一个实际的 4G 空间在内存中，只是 CPU 可以看到一个 4G 大的线性段，能以它为基础形成任务直接供程序使用这才是 CPU 关注的)..这样的话，对于编程来说，只要是在保护模式下，我们都可以拥有一个 4G 的编程可用空间，不必调进调出的(它本来就一直在内存中为 CPU 所看见)。。

任务调度主要是协调任务对计算机系统内资源(如内存、 i/o 设备、 cpu)的争夺使用。进程调度又称为 cpu 调度，其根本任务是按照某种原则为处于就绪状态的进程分配 cpu。由于嵌入式系统和标准系统中内存和 i/o 设备一般都和 cpu 同时归属于某进程，所以任务调度和进程调度概念相近，很多场合不加区分，下文中提到的任务其实就是进程的概念

上述 32 位 CPU 的转换机制是分段跟分页的综合作用。分段机制不管这个 4G 空间的实际调进调出，因为它不负责实际分配内存，它只负责逻辑地址到线性地址的转换过程..实际分配内存的工作由分页机制来完成，它负责线性地址最终到实际的物理地址的转换，它把这分段后形成的 4G 虚拟的空间用来调度真实的内存页，内存页就是真实的物理地址的载体，分页机制只需保证生成的页面大小总和局限在这 4G 空间中即可。。页面是随用随载，调进调出的，以较小的单位而不是段来实际使用内存，，这就加大了内存的使用率（虽然分页并非百分百做得到如此）。

要深切理解 CPU 是如何完成分段分页形成任务可用空间的过程是一个精微的学说，段选择符，门，LDT，GDT，pae （CPU 中有一个 pae 位可以保证所有 CPU 都只在概念上只看到这一大小）这些概念之间是如何相互间发生关系并维护这样一个 4G 空间。以及这个机制产生的结果和意义所在，需要翻阅相关的书籍。下面试阐述一二：

- CPU 首先检查这个选择符的 ti 字段，以决定是选择到 ldt 还是 gdt 中选择描述符，然后检查相应的索引以在 ldt 或 gdt(这些都是内存中的数据结构表，表驱动方式)找到最终的描述符。。这些符结构都在 4g 内存的高端部分，注意是在内存中..找到描述符之后，再以判断选择一样的方式在描述符中判断各个字段，主要是生成段基和

段长，段的三个权 cpl,dpl,rpl..

- 比如在如下的一个指令中 mov ds,ax, ax 存储的并不是指向 ds 的指针，，也就是说它并不实际存储 ds 的地址。而是一个选择符(注意，第一它不是段描述符，第二它段选择符是一个 32 位的数据结构，它的各个字段表明了不同的意义，组成整个段选择符的意义),,,段寄存器大小有限，CPU 中的寄存器只用来存储指针值，或者描述用的数据结构，比如在这里是一个段选择符。。

- CPU 就是靠以上方式来实现对于段机制的保护的，每一条指令都有内存读写逻辑，每一个内存读写逻辑都有这样的寻址，最终要涉及到进入段中。涉及到保护模式的一系列机制。。

- 门是 CPU 中的一种迂回机制，有调用门，任务门，中断门，异常门，这四种门的用途不一样，但都是为了实现 CPU 对内存访问进行保护的一种迂回机制(比如数据访问权限啊，任务安全切换啊，中断正常返回啊)，调用门实现了数据访问的权限机制，跟四种普通段，段描述符有关，任务门跟 tss，tss 段选择符有关，中断门与异常门跟中断或异常处理例程有关。。
  首先来谈调用门。对于一种跨段的段间调用 call 或跳转 jump，统称调用和跳转和转移,有直接转移和间接转移，，但是直接访问方式不能实现段间调用的同时转变程序的特权级，调用门的设置就是为了实现一个间接调用加同时改变程序特权的跳转方式。。
  任务切换有直接切换和间接切换，任务门跟一个 tss 段有关，跟调用门谈到的对于普通段迂回机制一样，任务门也是实现间接任务切换的手段。

  而对于中断门与故障门来说，门描述符内存储的是中断与异常处理例程的地址。。

总而言之，CPU 提供了对保护模式的硬件支持。而所谓保护，是对段，内存，这些保护模式的概念提供迂回机制，，于是发展出分段，分页，调用门，任务门，中断门，异常门这些的 CPU 机制，更多请自行研究。。

1.6 操作系统

当 CPU 能够被用来形成任务控响应中断时，操作系统这样的软件级的逻辑就可以在上面慢慢构建起来了。实际上机器往往并不需要一个操作系统也可以运作，比如电传打字机和电子打孔机，在裸机层次也可以实现其功能。

但配备了强大 CPU 的 PC 绝不是一般的电器，冯氏模式一开始就指出，它是能够换程序和可编程的，对 PC 的使用往往源于它抽象出来的强大的软件功能，而不是仅仅是

用 CPU 来加热鸡蛋(如果实际上 CPU 真的能煮熟一个鸡蛋而且业界流行这样使用 CPU 的话)。

操作系统就是这样的软件的第一层，它沟通了裸机与使用机器的用户，对于裸机来说，操作系统可以由它直接抽象出线程，进程，网络，图形等系统功能，运行于操作系统下面的各种应用软件可以使用它们进一步抽象出各种更为高级的应用逻辑(比如 Web, 多媒体)，对于最终用户来说，操作系统提供了界面使得各种高级应用成为可能，而且对于程序员用户来说，使编程工作脱离直接面向机器编程，，因为操作系统可为各种应用软件提供一个执行路径，程序员可以面对操作系统编程，这是一个巨大的成就。即操作系统不但是计算机功能新的提供者，而且是开发环境。

这样计算机系统实际上是硬件系统支持下的软件系统了。人类要直接面对的就是软件系统(这就是抽象的能力，它一方面隔断了人类并不擅长的硬件细节，另一方面提供给人们一个更为强大的软件环境)。不论对于最终用户或开发用户来说都是如此(当然对于硬件工程师来说不是这样)。

一句话，操作系统的出现是源于各层次人们普遍的需求。

操作系统提供那些对于应用和开发来说最最基础的功能和逻辑，因为它直接关联机器和 CPU 机制而直接面向初级软件需求(比如形成过程以运行程序使操作系统具有执行程序的功能)，在操作系统内核中，内存管理，进程，文件，网络，用户界面，是最先应被抽象出来和最先被解决的问题。我们知道软件即抽象，所有的计算机系统能呈现和解决的逻辑，第一步是解决系统支持的问题（这导致了对系统编程的意思所在。），第二步才是应用逻辑的表达和解法(应用编程和领域编程)。

当然，**操作系统是一个大的概念，它小到只需要包含一个内核，大到可以包括内核层，开发支持层，调用层，硬件抽象层，应用接口层这样的操作系统架构**（参见 Google 手机平台，这本质是因为软件是抽象，所谓抽象就是在不同的抽象层次完成不同的工作，操作系统作为软件也不例外）。因此讨论操作系统我们只需讨论操作系统内核便可一窥其端倪，我们只讨论巨内核的 Linux Core(相比之下有的系统内核是微内核，这种内核基本上只提供对进程，内存管理，图形，用户界面，这样抽象层次的封装和架构逻辑而不实际实现它们)。

Linux 内核实际上主要是一个定制硬件功能为软件的逻辑,怎么说呢,它一方面沟通了硬件驱动和软件,使硬件可以运作起来,为软件逻辑服务,,,这是第一,第二,它对于硬件功能的定制功能,比如进程,,,就是把 CPU 硬件资源转化为软件可以使用的进程资源,把网卡资源转化为 socket 跟进程一样是种 OS 资源, ,因此,内核实际上只是初初级的硬件功能抽象,

界面就是表现抽象，它的窗口机制,都是独立内核的(KERNEL 硬件抽象层),,如果称

内核为操作系统(实际上只是 OS 的核心一部分)的话,那么,X 协议的实作品,桌面环境,窗口管理器,该桌面上的 APP,,,都是作为应用程序来运行在内核之上的,而不是像 WINDOWS 一样直接在内核就集成了 GUI,苹果 OS 在内核集成了 X,因此图形效果很好(LINXU 桌面没有 WINDOWS 快), 这就是移殖问题产生的地方之一。,因为 UNBUT 逻辑并不是其它逻辑的泛化源而是与其它 APP 一样地位的基于内核的普通 APP。

计算机能做的事在经过 OS 之后就有了新的扩展，不光是 CPU 的取指令，译指令，执行指令，从内存中取数，CPU 用逻辑器和运算器运算，，中断处理例程，IO 设备的功能，而且还是计算机后来所有的软件逻辑包括 OS 的所有的功能都由它决定。。
由 OS 生成调用这些如上硬件资源的策略。。

1.7 Linux 系统

在操作系统方面,LINUX KERNEL 是最小逻辑,GNU LIBC 在这个时候就被集成了,很好,内核的架构是重要的,LINUX 先是硬件逻辑,再是语言作为中间层,用户程序 SHELL 等,作为用户空间,这种架构很模块很科学
Linux 内核实际上只是一个定制硬件功能为软件的逻辑,怎么说呢,它一方面沟通了硬件驱动和软件,使硬件可以运作起来,为软件逻辑服务,,,这是第一,第二,它对于硬件功能的定制功能,比如进程,,,就是把 CPU 硬件资源转化为软件可以使用的进程资源,socket 跟线程一样是种 OS 资源,这之后才出现 OS,,因此,内核实际上只是初初级的抽象,而 SDL,就是更高一级的抽象..
它的窗口机制,都是独立内核的(KERNEL 硬件抽象层),,如果称内核为操作系统(实际上只是 OS 的核心一部分)的话,那么,X 协议的实作品,桌面环境,窗口管理器,该桌面上的 APP,,,都是作为应用程序来运行在内核之上的,而不是像 WINDOWS 一样直接在内核就集成了 GUI,苹果 OS 在内核集成了 X,因此图形效果很好(LINXU 桌面没有 WINDOWS 快),,,,我们平常所谓的 UBUNTU 也只是 APP 集(在内核上面一点点的逻辑),,,在服务器版中能精简得到,这就是为什么移殖了 UBUNTU 却不等于移殖了它上面能进行的一切 APP 一样,因为 UNBUT 逻辑并不是其它逻辑的泛化源而是与其它 APP 一样地位的基于内核的普通 APP

我觉得用语言本身来作为架构的一部分,这样来实现移殖,可以解决一切移殖问题,只有该语言的 VM 被移殖了,那么该语言之下的逻辑就全部被移殖了
对一个平台的开发是促使这个平台能得于流行的基础,,,

1.8 CPU 与异常

一般谈到源程序的跨平台特性时，总是说在 p4 加 winxp 下编译通过，所以软件移殖问题最初源于硬架构的 CPU，然后才源于软架构的 OS，同时提到了 CPU 和 OS)，

CPU 与 OS 的关系该如何理解呢，如果说 CPU 控制了计算机硬件上的一切，OS 就相当于 CPU 的外层调用策略(shell 就相当 os 的外层调用接口)。它把 CPU 逻辑和 CPU 控制下的外设逻辑，存储逻辑封装为软件可用的东西。os 的许多东西，是 CPU 硬件加速下的结果。

比如说 Windows 和 linux 的内存分页机制和保护模式，,如果 CPU 没有提供三种模式(实模式，x86 虚拟模式，保护模式)中的保护模式，那么在 OS 级虽然可能可以实现一个所谓的保护模式(但却没有直接硬件的支持)

在保护模式下，CPU 中的段寄存器不用来实际存储物理地址的线性表示(而在实模式下是这样)，它用来实现硬件级计算最终物理地址的一个中间存储。。

中断是硬件级的消息，中断和异常是 CPU 直接相关的东西(CPU 有二跟中断引脚),,,很多编译器提供了异常处理机制，Windows 更是实现了一个 SEH,，但是，这些都是抽象了 CPU 的这方面的逻辑。。

中断中的硬件中断是真正意义上的中断，把它称为中断是为了跟异常区别开来，二者都指 CPU 对影响它原来指令流程的意外干预过程。(CPU 对这二者都提供了处理例程)。但是中断是硬件级的，是来自外部的，异常是来自 CPU 内部的指令执行过程中的一个出错,，是来自指令执行过程中的(所以，所谓的软中断指令其实也是异常)。

而发生中断或异常时，二者都是靠 CPU 的跳转指令来完成跳转到相应的处理例程的,，这是 CPU 直接执行指令的结果(Ruby 甚至鼓励用中断和异常来代替正常跳转，这是直接用 CPU 的指令的结果，编译器控制不好会造成很多隐患)，机器级的跳转指令是程序语言实现它的控制流的一个重要方面

在保护模式下，跳转应在各任务间正常切换，否则会引起著名的操作系统保护错误，处理例程调用完之后通过一定手段返回正常任务)，那么 CPU 就发展出一些诸如调用门，任务门之类的东西。。用来规范这些跳转。。

再比如递归，由于它每次调用实际上都产生一个新的函数，对于栈式 CPU 的计算机来说，由于它主要利用内存而不是寄存器来存储这些新函数的临时变量和压参操作,，因此这个递归过程不应有过多的调用深度，否则压参入栈方面的开销会过大（这是指空间方面的优势，栈这个段区可用来存储很多参数但是速度不快，而 reg 式计算机速度快但是没有过多的 reg 供递归存参用。因此空间方面是个劣势。)。

1.9 所谓堆栈

过程式语言跟堆栈这种逻辑密切相关，堆栈逻辑一部分体现在机器模型，一部分体现在语言模型上，

如果你知道函数对于结构化程序设计的重要性的话，那么你也会明白堆栈对于函逻辑的重要性，所以理解结构化程序设计范式的重要手段是理解堆栈这种 ADT 。而这出现在数据结构学中。

运行时是机器挂钩的概念， 非语言挂钩的概念，（但是实作品可以用语言来描述）一门完善的语言的提出，，包括提出一个针对该机器环境关于此语言版本的的代码运行时环境.（因为语言实现需要被运行，那么就需要发展出一个 c runtime dll 的东西了）称为 native runtime，因为是对机器本身，而非类 jvm 软件机器，的本地机器运行时逻辑的 C 语言封装。(我们将一种机器是属于 reg 方式来执行逻辑还是用内存 stack 方式来执行针对它的平台逻辑，，以及相关的一系列知识称为运行时)

我们知道，REG 和内存都是语句运行时(不妨改成运行空更好)的存储空间集散地，但是不同的机器，有的不具备 REG，有的侧重利用这二者的机会又不尽相同

如果机器是堆栈机(采用内存 stack 方式的运行逻辑)，那么它就是用内存来作为主要运行场所的机器，JVM 速度很慢，但是很耗内存，从这点就可以看出来了(因为它的机器中没有软件摸拟的 reg,,为了跨平台，jvm 根本不打算将 x86 的 reg 机制引入，而我们的 x86 平台是硬件的 reg，所以称为本地平台，硬件的 reg 总比软件的快，软件的 3d render machine 总没有直接硬件访问的快你可以联系这点来理解)

机器如果是寄存器机，那么主要用寄存器来执行语句逻辑，指令逻辑，完成数据的传送(语句即指令加数据,为什么任何一个逻辑正确的可执行文件为什么 CPU 能执行它并产生结果呢，这是因为一切 pe 逻辑都可反汇编译成指令加数据的语句，它能被执行，因为它同时包含了代码加数据，CPU 不能执行纯 data 或纯 text)，，因为寄存器是 CPU 的而堆栈是内存的,所以 reg 机明显比 stack 机快，但是寄存器器数量和容量有限(比如递归不好做)，所以针对这类机器设计的单个指令代码长度一般不长，，在一些方面是比不上 stack 机的

当然，intel 平台中，，CPU 和内存是共同发挥作用来实现寻址的，而且也共同(主要指 CPU 中的 REG，一方面，SP 这类 REG 发挥指针的作用指向内存，一方面，一些 REG 本身可用来直接存数据)作用构成运行时(时空=CPU 时间加内存场所)，，那么 intel 平台是 stack 机还是 reg 机呢，当然是综合机

首先，对于 x86 平台，是如何执行函数逻辑(比如反汇编的一段 C 语言的函数的附近得到的一段汇编语句)的呢，，通过探索这个过程分析那些汇编逻辑(其实就是编译器产生的平台逻辑，机器码嘛)，我们可以发现这个平台的运行时底层信息

首先，PE 载体逻辑内有一段堆栈段，被映射到内存(载入器和重定位器会完成这个工作)后形成一个堆栈区，寄存器此时高速活动，它的一个活动记录中，必定保存有栈基栈顶的一个指针(此时它们重合)

堆栈的基础概念要掌握，它是一段从高地向低地生长的空间，而且是活动空间，因为栈顶(指示当前出栈入栈位置的指示，它存在 sp reg 内，永远只拥有一个是“当前“的值，减它就是在低地址更低所以是为堆栈腾生长空间，加它是让低地址往高地不低一点，堆栈空间会变小)会因为频繁的出入栈动作而变动。这种变动性使得我们掌握堆栈的方式(求某一个成员在此活动堆栈空间的位置)只能用简单的'bp"+- 该成员对应 bp 的偏移，的方式来指定了，而栈顶的偏移永远是 sp-bp

在进行一段函数调用时，必定是存在调用方和被调用方，一定存在参数间的互传，怎么传参，调用结束后指令权如何在调用方与被调用方之间协调，平台是如何执行这些运行时逻辑的呢？

对！x86 用到了堆栈，对于 C 语言函数，参数右向左入栈(显示成汇编逻辑时是 push)，第一个参数反而在堆栈(此时是堆栈对应这个函数附近活动区域的一个帧，因此称为堆栈帧,即堆栈区的某个真正的活动堆栈)的最上面(偏向低地址方向生长)这样，相对其它参数来说，它就显得先进后出嘛，后进所以先出嘛，当然，它到底出不出，此时出不出，这跟当前 sp 没有一定联系，，因为 sp 当前位置不只是由压参影响的(甚至没有必然系，因为 sp 这东西可以人为用汇编码指定)，还受其它自动变量的影响呢

C 语言的这种规则跟 pascal 的规则完全不同，，这导致的 C 语言函数可以有变参数数量的好处

1.10 真正的保护模式

8086/8088 有 20 根地址线，寄存器为 16 位
寄存器的位数，16 位表示决定了计算机的字长，即一次性能处理的数据总长度(16 个位，始终记住，位是计算用来表示存储单元和数据长度的最小单位，无论是外存或内存都一样)，因此在程序中只能定义至多 16 位长的变量，因为只能定义 16 长的数值(变量),故(寄存器大小)它也决定了计算机能表示的数值的大小，即 2 的 16 决方(1048576),这种数值至多能用来表示的存储单位是 1048576 个,,或者说 1048576 个位，而这正是一个 16 位的数值变量所能达到的最大值，因此这种寄存器为十六位的 CPU 只能表示至多 1024kb 个内存位，虽然可能计算机本身不止这么多内存，虽然有时地址线不止寻址这么点的空

间，但 16 位的寄存器只能看见并寻址这么多的内存(因为地址线是 CPU 引脚线，是 CPU 的概念)，寄存器的位数理论上决定了 CPU 能"表示"的最大的内存范围或外存范围(当然，连 CPU 表示都无法表示的内存范围那就没有意义了)，而地址线决定了计算机实际能存取访问寻址到的内存范围，即 1M，不包括 CPU 连看都无法看见的那部分,(当然如果你的计算机都没有这么多的内存也是枉说)一个是 16 位，一个是 20 位，CPU 是怎么样产生数值用来表示地址线所能寻址到的 1M 地址的各个单元的地址的呢???，(这里以字节来说)1M 可以分为 64k 个 64b,这样，寄存器用来存放地址，80286 有 24 地址线，寄存器为 32 位

因为寄存器是存在于 CPU 中的，因此说是 CPU 的寻址，又为什么说 CPU 对内存的寻址呢，为什么我们在这里口口声声地说 CPU 对内存的寻址呢，这有什么意义呢？这当然是为了计算机能正确执行指令而准备的,这是计算机工作的本质啊(而为了执行指令，CPU 能对内存进行寻址是首先必须要办到的,因为程序最终从外存到内存中才能被 CPU 执行，CPU 和内存都是靠电工作的,CPU 提供执行指令的本领,而内存提供存储指令的本领,这是冯仪曼说的,成功完全指令必须是 CPU 和内存一起工作完全的,而外存是靠磁来工作的，CPU 只能执行内存 RAM 内的指令，外存用来永久存放文件,称为持久化),程序要从外存被加载到内存中才能形成指令

(指令在程序的 CODE 段中，EXE 文件被加载到其进程空间时------这个过程术语叫映射到其进程空间，

它的代码段就在物理内存中了，因为只有代码段才包含指令，这部分要首先映射到物理内存中来，程序的指令用到的数据------这通常表现为程序中的变量常量存在 data 中，数据段部分被开辟成堆或栈的方式被映射到 EXE 的进程空间中来(分段机制),形成 EXE 的编译器不同开辟模式不同，像 Delphi 的编译器实现的 exe 进程空间只有堆没有栈这种内存模式，堆和栈是系统级运用内存管理器进行分配内存的动作后(是系统级的作用)形成的特殊内存块，这些内存块被映射到 EXE 的进程空间，这有点像 EXE 的 DLL 模块的映射模式，dll 文件被映射到其宿主 EXE 的进程空间时，不像 EXE 一样一开始就把代码段实际加载到物理内存中去了而是在 EXE 实际调用到它时才到实际的物理内存中去(分页机制，只要你的计算机定义了虚拟内存，那么在执行大程序时，，这个分页机制就会频繁用到),跟 EXE 的 DATA 段一样属于一开始就映射到 EXE 进程空间而不实际形成内存的部分，EXE，DLL 被加载到内存后，它所占据的内存块就叫模块是静态的，而进程是一个执行体的实例是活动的，线程是一个进程的某个执行单元，所以我们说程序被映射到到其进程空间而不直接说映射到物理内存中，只是需要执行的代码段(注意此段非实模式彼段，后面会谈到)才被进入到物理内存，但不需执行的那部分不需立即加载到内存(就像 DATA 和 DLL)不得不说的是，进程空间并非物理内存，前面一再强调"程序被映射到到其进程空间而不直接说映射到物理内存中",而且更准确地来说，它们二者是完全没有关系的，4GB 虚拟地址空间整个儿压根儿就是虚拟的(这个道理就像你玩一个客户端为 2G 的游戏时，你启动客户端的时候已经把整个客户端的 2G 资源都加载到 4GB 空间去了，但是只要这 2GB 中需要当前调用的那部分资源才进入内存,分段机制开避四 GB 任务空间，分页机制把需要用到的数据动态加载到进程空间,任务空间就是进程空间，然后通过这些资源在程序中的自动变量表示离开内存)，它只是能表示 4GB 空间的虚拟地址

而已，并不是实际的物理内存，仅仅根据 32 位寄存器能表示那么多的内存来设置的那样一个机制，这种机制成全了将进程空间隔离的好处(所以四 GB 的说法是进程的一个概念通常说 4GB 是进程的空间)，而不像整个六四 KB 都可以， Windows 的虚拟内存管理器(保护模式下)会负责适当的时候把需要的，映射到进程空间的内存搬到物理内存中去(分页机制)，现在来解释"注意此段非实模式彼段，后面会谈到"这句话,在 FLAT 下，已经没有段 segment 了，,在 Flat 模式下就无所谓 code 段跟 data 段的了，因为原本实模式 CS，DS 意义下所指向的段全都在一个 4GB 大的虚拟地址空间中，实模式下段的意义不存了，是段选择子，FLAT 内存模式下， CS,DS 根本就不实际用来指向段，即不实际用来指向一段内存(而是存储一个称为段选择符的数据结构), FLAT 下说的代码段是指 EXE 头定义的段，是 RAW FILE(RAW 指没有分配实际内容的内存)里定义的段而非实模式下 CPU 段寄存器指向之"段"，模拟原本的段取而代之正是 EXE 头的节的意义，程序员无须知道这些段的实际的物理内存地址而只须知道虚拟地址，(我们知道在 32 位寄存器，在 RAW FILE 里才有节 section，有了 PE 文件的头中的这样段定义，当 EXE 被加载到内存中来，就相当一个跟内存数据段一样的数据结构，虽然平址模式下无所谓代码段数据段，但 PE 文件的格式间接实现了它，就像 XML 文件它本身就像一个数据结构一样所以它被很多程序语言的 IDE 用来持久化，被很多程序用来保存信息如 3D 网格信息也会用 xml 文件格式)

1.11 异常与流转

运行时是不是移殖的产物？比如 apache 运行时，是一个抽取最核心的运行库，比如 C 库的运行时,等
异常是什么呢?异常不是错误,,是比错误轻一级的概念,当异常(即使被捕获到了)不能(在程序中预定义的处理块)被正确处理,就可以抛出一个错误(当然，错误和异常在程序中可表现为很多种，编译型语言如 C++就有静态编译期语法和动态运行期逻辑错误之分)而调试，包括对这二部分进行调试,静态期的调试要容易得多,而运行动态期的类型在写代码时(编译前和编译中)显得易读,但是调试时十分困难(因为要从运行中调试)..这就是动,静的区别所在
(这个过程通常是,我们按照正确的程序流程来处理错误,发生错误,在程序预测到它,并不尝试恢复,,比如跳栈,如果能处理则处理之,不能则崩溃)
在传统编程中(一种语言未引进异常机制前),我们总是用 IF,,THEN 这样的判断跳转语句来提供处理异常发生时的处理逻辑,,,当一门语言内置异常机制时,此时,我们就可以用新的处理逻辑,此时,通常程序语言会作跳栈和保护现场的动作 .因为大凡大型软件,需要一种很强大的异常机制,它必须全天侯运行，不能出错就恢复不了，这种动用了程序语言原理（一切程序即函数调用,如果你反汇编过就知道了）来处理异常的方法就是真正的异常. . (异常与跳转关系如此亲密,RUBY 甚至鼓励用异常实现流转)

1.12 操作系统与语言的关系

附带参见 4.10《我为什么学习 python》第二大段“为什么学 python，另外一个原因是”起。

语言有它跟 OS 直接相关的地方，虽然形式语言的理论不跟任何 OS 相关，但是具体一个语言实现(编译系统或 IDE)都服务于一个 OS。

操作系统与语言的关系就是一个系统与系统开发的关系。这是二个独立而又相互联系的模型，native 语言服务于系统，而 vm 语言服务于虚拟机（其实像 native 是相对的概念，比如 java 相对于 jvm 就是 native,但是一般还是把 win32 加 x86 作为真正的 native）。

如果能有一种方法，使一种语言机制直接跟 OS 相应，而且一一对应，，没有其它任何第二种语言建成在这个 OS 上，换言之，跨平台问题将不存在，，因为只有一种语言产生的逻辑需要考虑移殖，

编译器的目的是为了产生码，系统语言基于平台逻辑是为了开发平台逻辑，脚本是为了描述应用

那么解决 GUI 问题有什么新的方法呢，

1.13 虚拟机与语言

虚拟机是对 CPU 的模拟(还有模拟中断机制和 IO)，，，一般提到硬件的“机”就是指冯氏的构架，

这个构架中，CPU 是首先要谈的东西。。Intel 的构架有专门一本书来讲解。汇编语言就是 CPU 的汇编语言，不同的机器有不同的指令集。。因此有不同的汇编语言，这就是移殖问题最初产生的地方。。

如果你玩过软件的虚拟机就知道了,,市面上存在很多虚拟机..

在设计一种 CPU 时，总是先实现一个软件的虚拟机来模拟它，这种软虚拟机被运行在一个平台上，运行在这个平台上但对虚拟机编程的编译器叫交叉编译器。在虚拟机设计完全并实现之后，这个编译环境也要移殖到虚拟机上(此时它被实现到目标硬件平台上)，，只需要用这门语言产生关于这门语言在新硬件环境下的编译器就可以了。

然后，操作系统就可以在这个基础上开始设计并实现了。。但用虚拟机方式来验证此机器，和提供一个编译器，这是首先要做的事情。。

posix 是类 unix 系统的接口标准，，为了移殖方便而提出的，因为 OS 实际上是一种系统调用例程的抽象，这个系统调用例程规定了“计算机硬件反映到软件能做的事”，是最初软件逻辑产生的地方，被经过 OS 封装之后形成了进一步的进程逻辑，sokcet 逻辑等。。对系统编程就是考虑到硬件逻辑和 OS 逻辑的一个综合过程。
C 语言作为系统编程的代表语言，它的标准库里面的函数，实际上是站在硬件逻辑和 OS 平台逻辑上面的一个通用接口。在 Windows 和类 unix 上都是这个抽象接口。支持 C 标准库的 OS 必须声明它支持这些接口。方能让标准 C 语言移殖其上。。

1.14 虚拟机与语言

通常用一门语言写出来的程序都被用在诸如ＸＳ６加ＯＳ的架构上运行(平台移殖或编译通过时,我们总是说,在某某某架构上加某某 OS 上编译通过)，当一门语言为了实现跨平台（一般是跨硬件架构，硬件掌有真正的运算资源即芯，而ＯＳ核心将硬件能力供软件服务，包括进程，等，对一台裸机编程就是汇编了）考虑时，它首先发展出一个虚拟机（只有这个虚拟机实现了跨软硬平台那么语言代码就实现了跨平台了，因为代码的运行环竟的根本是硬件架构)，这样所有用这个语言写出的的代码就由这个虚拟机来解释，这个源代码就不是本地码了（不是供Ｘ８６＋ＯＳ这样的本地生成机器码)，软件实现的虚拟机往往是一种高级机器(是一种刻意为了适应程序的运行而设置的完美环境,实际硬件上的这样的机器没有),而往往是一种虚拟机能执行的中间码,,当然,虚拟机上的语言如果有特定编译器和编译技术的支持,同样可以为操作系统和硬件生成机器码

一般情况下，软件运行环境=硬架构加软架构
那么虚拟机就是专门为了一门语言而出现的高级软件机器，，它直接在裸机上（加ＯＳ？）运行，因此去掉了ＯＳ的作用，是语言跟它运行环境的直接结合体(虚拟机是虚拟机,虚拟机里面的编译器或解释器就是它跟语言发生联系的地方)，是为了一门语言而出现的优化运行环境
虚拟机的意义远远不只是为程序代码提供一个可运行环境,它真正的意义是使编程工作完全远离操作系统跟硬件架的差别(传统的源程序总是要考虑到移殖,而且要考虑到平台本身给语言造成的复杂性,比如 C 库的 I/O),,在 JVM 和 JAVA 下编程,我们无视有一个 OS 和平台的差别,所有人都面对同样形式的代码..JVM 有了这个优势,,JAVA 因此成为了工业语言的原因之一.
通常用一门语言写出来的程序都被用在诸如ＸＳ６加ＯＳ的架构上运行(平台移殖或编译通过时,我们总是说,在某某某架构上加某某 OS 上编译通过)，当一门语言为了实现跨平台（一般是跨硬件架构，硬件掌有真正的运算资源即芯，而ＯＳ核心将硬件能力供软件服务，包括进程，等，对一台裸机编程就是汇编了）考虑时，它首先发展出一个虚拟机（只有这个虚拟机实现了跨软硬平台那么语言代码就实现了跨平台了，因为代码的运行环竟的根本是硬件架构)，这样所有用这个语言写出的的代码就由这个虚拟机来解释，

这个源代码就不是本地码了（不是供Ｘ８６＋ＯＳ这样的本地生成机器码），软件实现的虚拟机往往是一种高级机器(是一种刻意为了适应程序的运行而设置的完美环境,实际硬件上的这样的机器没有),而往往是一种虚拟机能执行的中间码,,当然,虚拟机上的语言如果有特定编译器和编译技术的支持,同样可以为操作系统和硬件生成机器码

1.15 调试器与汇编器

虚拟机与语言的关系很密码，像 jvm 跟 Java 语言的关系就很爱味，有人说，jvm 仿佛并非一个通用的软件环境，对 jvm 编程久了，会觉得像是吃饭的时候去了一家只提供一种口味的冷饮店，因为有些 jvm 的机制，比如 rmi,,只限于用 Java 语言去开发。

Java 语言属于高级语言，实际上当构造了一个虚拟机之后，第一件是给他设计一套汇编语言。除非你不打算给这种虚拟机进行开发，一旦有开发的需要，就需要实现一个汇编语言或一套高级语言。这种必须要实现一个汇编器，，，只有这样，虚拟机才能开始执行，，虚拟机本身并不需要一个汇编器才能开始执行字节码，你可以直接为虚拟机手动写字节码,,提供汇编器只是为了开发的需要。。上面说了。

同样为了开发的需要，，调试器也是需要的，一般的 IDE 都有调试器，调试逻辑来源于 CPU 逻辑，intel 中有断点中断处理例程，，中断一般来源硬件动作，，但也可在源程序中人为地给 CPU 下一个软件中断，这就是 int 指，interrput 的前三个字母，int 3 这样的指令让 CPU 产生一个陷阱状态，让机器进入调试状态，此时 CPU 并不正常执行程序，，而是一步一步地执行程序。。

调试器分硬件级的和软件级的，但都是为了让开发者或机器调试者研究程序在机器内部二进制级别的执行程序,,调试器一般向你显示 CPU 寄存器状态，符号信息，内存上下文等等。。

汇编语言实际上并不图灵完备，因为它是对指令的一种命称指代，汇编语言没有变量。因此没有编译器里面谈到的变量等符号映射为内存地址的符号化信息，，因为汇编器毕竟不是编译器，解决问题时所面向的源语言和目标语言不一样。。

运行时，从字面上理解为代码运行的环境，无论它是机器套个 OS 执行引擎，，还是虚拟机这样的软件模拟的运行时，无论如何，代码要得于运行，必须需要内存和 CPU 指令，，JVM 就是 JAVA 字节码的运行时(并不包括 JIT，JAVA 的编译器等部件，这些都不是执行字节码必须的，只是负责生成字节码的，而运行时是驱动目标代码运行的环境)，C 语言代码被编译后到 WINTEL 上执行时，Windows 操作系统加编译器的 runtime 支持就是机器代码的运行时..

1.16 命令行下编程实践

编程为什么要在命令行下而不提倡在界面下呢，，这是因为命令行下有界面，而界面下人们都往往不觉察到有命令行，而其实命令行才是本质逻辑，界面只是一种 GUI，是一种建立在本质逻辑上的用户接口和配置文件（只是一种不必要的表现抽象）。真正的逻辑可以不需要一个界面接口来展现它，但强加了界面接口的逻辑就限制了逻辑本身的表达。。因为**命令行是观念里自由的世界，是 C 这样语言开发时所面向的直接世界，是计算机逻辑发源的真实世界，所有计算机逻辑的起点是命令行的**(仅需要文字界面这样的用户接口就行了)，图形用户界面逻辑只是后来的后来需要考虑的东西。。所以开发时不免将界面逻辑滞后，而实际上，无论先后顺序还是本质特征，我们始终都要把握一个观念，即其实命令行比（图形）界面丰富有趣得多，我们始终要相信，虽然我们在 Windows 下长大，但其实命令行才是主流，才是我们的思想和开发根据地。

UNIX 下的那种 Shell 操作，命令行工具和管道理念，远远比 Windows 菜单有效率(当然，命令行最普遍它的门槛却很高)，因为本质上，命令行逻辑的一条语句 "ProduceMenu(10000)"，屏幕再大，也不可能生成 10000 个菜单供你使用。。而在命令行下，一条命令可搞定这样的逻辑，因为命令行是没有界面限制的观念世界（图形没有文字表意丰富），形成计算机逻辑的一层一层是是命令行的，，C 开发时也是命令行的因为 C 的库全是命令行的，Windows API 接口是 C 后来的产生 GUI 的一种方式，C 是面向命令行的，它面面系统底层，并不一开始就将界面逻辑纳入所有的逻辑开发中。。

我常用的 C 开发平台是 CH，一种 C 解释器和命令行模式下的交互编程环境，它用了如下的 Msys 工具，我们需要深克掌握这些 Linux 下的开发工具（它们实际上也是系统工具，因为 Linux 比 Windows 提供了更原始对开发的支持）.

make，这是 linux 下编译大型程序使用的，很多 IDE 都集成了它。

vim，，如果能熟练使用 VIM，那么你就会发现，它远远比 Windows 下的记事本，和一切 IDE 都有用。。

1.17 平台之 GUI

历史上,Sun 公司一直在操作系统方面力图占据桌面(它后来的 SUN DESKTOP SYSTEM 简单就是拉及啊),,它的 OPEN LOOK 后来败给了 MEIRF,

GUI 问题其实首先是一个设计问题。

最初的逻辑都是没有 GUI 接口的,一切都是 SHELL 跟 KERNEL,Unix 系统下,MIT X Windows 是桌面的 CORE,桌面最著名的模型,就是 B/C 消息模型,用类 C++写的桌面系统才有 CLASS,用 C 写的比如 GNOME 没有窗口类名这样的概念存在,有的只是关于窗口的指针,句柄(这些逻辑可由ＱＴ，ＧＴＫ这样的来产生，也可由编程语言来产生，比如ＧＴＫ，在库内就实现了＂对象＂这样的逻辑，因此它的窗口也有ＣＬＡＳＳ　ＮＡＭＥ这样的逻辑)等.
为什么 GUI 如此重要?

最初的第一个编译器是以很复杂的方法被写出来的，后来的编译器是有形式语言这样的理论指导下发展出来的，因此，解决开发编译器的角度不同，那么所付出的努力也会不同，就像 GUI 问题，如果你能换一种眼光去看，，那么或许我们现在所看到 GUI 就根本不是现在的 GUI，，现在 GUI 的消息机制，也许以另一种思路就能很轻松解决(比如我们将界面逻辑看成 XML 逻辑，在 XML 中提供消息机制达到的效果，或者根本不需要一个所谓的消息机制，不同思议的不是问题本身，而正是人类的大脑，任何问题都可以只是想象，然后才有解法)，不给平台任何更多的运行负担，，因为没有平台相关的逻辑存在，这样 OS 的内核就不会越做越大。

从C语言的观点来说明桌面,从C 语言的观点可以说明一切逻辑,因为计算机逻辑都是由C 开发的,而 C 本身也最接近计算机的处理逻辑,C 抽象了汇编中的一些机器困素,但是它幸好没有做到全部抽象,比如指针,实际上直接跟内存这个部件相关,因此 C 语言可用来解释很多计算机的东西,,,这是 C 语言作为教学语方的一个很好的地方.
现在流行的开发一般都分为桌面开发和 WEB 开发,但是桌面和 WEB 应该是被整合的,GOOGLE DESKTOP SEARCH,GNOME 的名字中,就有 NETWORK,甚至于 OS 内核中就有网络支持,在 GNOME 中也有 BELGEL 这样的桌面搜索,,因此网络跟桌面本来就该在一起,只是 WEB 和 HTTP 协议这特定的一块的逻辑得到了长足发展(因为跟企业应用紧紧结合),导致了流行的 WEB 开发,因此,技术都是受商业驱动的..不要以为技术是很高尚的东西

因此，GNOME 这样的图形平台标准，或者一个实现，，就影响了桌面应用，，以及桌面上的很多程序的移殖问题,,从浏览器、办公套件、邮件客户端、音乐/视频播放器、CD/DVD 刻录工具、BT 下载软件、即时通讯工具以及偏门的音频抓轨工具都一应俱全,,这本质也就是我们在前面说到的是逻辑的关联问题,,比如,浏览器所用的 HTML 逻辑,就被封装在 GNOME 中,这造成运行在它上面的 WEB 浏览器各不一样,因此 IE 核心,跟 FOX 核心是不一样的..corba 甚至用在 GNOME 实现内，就像新兴的 XML 一样也被用在ＧＮＯＭＥ内

1.18 界面的本质应该是命令行功能支持下的配置描述文件

你以为 GUI 是什么呢？

界面 GUI 这个东西，不是逻辑本身（人们通常以为 CUI 才是，CUI 中提供一些支持界面的逻辑，,然后,界面主体的本质应该是命令行功能支持下的配置描述文件,此时 GUI 不是逻辑，而是配置），但是界面也可当作逻辑本身来看，，此时，它就不应该是配置

在操作系统的设计中，那些硬件访问功能，CPU 资源转化成进程，永远是最底层，最应该被首先解决的逻辑，，那么界面呢，通常被放到后面，操作系统的设计跟一般稍微中等应用程序的设计(特别是涉及到移殖)时碰到的问题几乎是一样的多，，因为应用程序总有它的关于运行的平台逻辑（CPU+OS），，而不总是那些建立在业务领域的高层逻辑，，在完成这些底层逻辑的过程时，也要涉及到界面，，涉及到线程，，涉及到图形，，涉及到 SOCKET，，内存模式的数据结构，等，因为这就是 PC 的逻辑，除非你不是对 PC 开发（开发语言本身跟平台结合导致的一系列底层逻辑也大量出现在开发语言本身的机制和细节中，，也出现在开发语言开发出的可复用库中，），，因此只要是复用，都跟平台逻辑相关，，只有像 WEB 开发+JAVA，这样的组合，才使程序员彻底(也不是彻底，70%吧)不需要了解系统知识，，而可以进行编程(所以有人说 WEB 程序不是纯正的程序员，因为它们不懂计算机，只懂他们的工作业务，他们做的不是开发，而是逻辑配置)，你说写 XML 文件是开发吗？写 XML 机制的实现才是开发，，因为涉及到数据结构和内存。。这才是开发,,,Lua 作为配置语言的脚本语言就是这个意思,它适合写界面描述和配置数据描述这些高层的东西,不适合描述机器本身..

可见，所谓程序，就是不同的层完成不同的工作而已，，一个是靠近 PC 底层的比较难点，，一个是靠近你要做的事的高层，比较容易点，，这一切体现的精神，，就是封装和接口，，封装体现的终极精神是复用，，复用体现的终极精神是人。。

设计应该被配置，而不是被编码,设计语言应是一门非图灵完备的独立语言(要考虑到它的通用性而不是 DSL 就更好了)。

如何为一个 OS 设计 GUI 接口呢（我们以为 OS 内核都是命令接口的像 LINUX 就是这样，不像 WINDOWS 内核那样把 GUI 集进去）

首先最底层就是 LINUX，然后是 SDL 逻辑和 XML 逻辑封装进 linux 内核，作为 CUI 中为后来所用的 GUI 接口,然后在这基础上发展出一些封装 SDL 的高层库（使得以后的开发不要动不动就访问硬件），，用这个库发展出一个桌面环境，比如 argr

然后玩游戏什么的，就直接使用 SDL 就可以

其它的应用逻辑（除了玩游戏等桌面应用），比如网络应用，因为内核中有 SDL 和 XML，至少它们的界面已经解决了，其它的问题就是其它的问题了

第 2 章 系统内核之 Linux Core

2.1 Linux 的网络架构

进行编程，只要涉及到多线程，网络，几乎总要懂得这样的系统逻辑(虽然它也可以是语言逻辑)，因为即便高级语言如 python 对它进行了怎么样的封装(变成语言级的东西了)，但它最初源于基本的系统逻辑还是不变的，总得有挂起，信号量这样领域词汇要掌握。

2.2 Linux 的多线程

在操作系统和应用程序的逻辑中都会涉及到并发逻辑(特别是数据库系统中)，并发逻辑可以多线程也可多进程，甚至多线程多进程的方式进行，当然，单核环境下的并发永远是一种对并发的模拟机制。

解决多线程的问题是多解的，，源于解决方法的高层模式各有不同，，(也即，从开发的眼光来看，它首先是一个设计问题)，，有不同的相关概念，，比如锁，，，就不需要用到信号量。

当然，有时你用的语言提供了并发的支持，，而且有时候，操作系统的并发逻辑会影响到你选用一种更有效的相容于操作系统的并发模型

并发问题源于图灵模型与需要开发处理的现实问题之间的矛盾，图灵模式是控制主导数据（机器的底层问题都是 IO 问题），这使得

2.3 coming on..

第 3 章 语言

3.1 真正的计算模型

在第一章中我们谈到，整个 PC 系统是一个经不断抽象后形成的构架，在最底层是计算模型，即图灵模型，在发展出一个 OS 后，硬件转化为软件了 (也即，可供开发使用了)。如果说 OS 抽象硬件为基于 OS 之上的开发工作提供了可能，那么 OS 开发层就是进一步把这种可能具体化了 (一个 OS 的发展总是离不开 Developer 的支持)，开发层出现的抽象就是语言模型和开发模型。

可以看出，不论是对计算机本身，还是对其上的开发工作，都是一个不断由底层向人抽象的过程。

开发模型就是开发者跟计算机交流的抽象(跟 OS 抽象硬件我们就统一面对一个软件层面一样，如果开发模型这个抽象被解决了，我们就可以专注于正在写的代码和正在用代码表达的应用，因此你感觉不到它的存在，不可否认的是，高级程序员只需要在语言提供的类型上开始工作，而不用管编译器在背后为你做了哪些把类型映射为位的工作，也不用管 OS 是如何为你的程序提供一个执行路径的)。

但是正如图灵模型不是一种真正智能的机器一样，计算机作为一种机器，给计算机设计它能识别的语言只能是一种形式语言。(巧合的是，计算模型往往跟语言模型一样，都会用到抽象识别装置比如图灵装置),,

用计算机来识别的语言只能是形式语言而非我们的自然语言，自然语言是一种形式语言的超形式，因为它可以被解析为形式语言,并且带有它作为自然语言的非机器解析元素，形式语言是严格的（不符合条件就出错），自然语言是活拔的（计算机语言不可能是自然语言是由自然语言的性质决定的，因为自然语言中有太多的歧义），与自然语言一样的是，形式语言也有语法。

我们不妨可以这样理解。

语言有组成它的最小单位即词(较大的单位有句子，段落，文章，卷等等,这些更大一级的单位又是由词或其它单位构成的，,包括词在内，这些语言单位构造的语言具体或语言形式统统称为串,语言的最终目的就是得出语言的一个一个串),和语言单位进行组

合（产生串）的规则即文法产生式,任何语言最终都是具体语言材料即词汇和具体语言规则即文法的组合

语言分为语言具体和语言形式，语言具体即最终由终止符构成的串，而语言形式即不一定最终由终止符构成的串(它也可以是语言具体和语言形式的杂合体)

串是语言材料的模板（串代表一个一个的语言具体或语言形式）,，而文法产生式是语言规则的模板(文法产生式产生文法)

语言具体或语言形式的产生是通过串与串的等价替换得出来的(这是串与文法关生联系的地方与手段),，如果存在 w0->w1,即串 w0 可以替换 w1,这个串间替换的规则就是“文法产生式”,，用来产生文法

以上到底什么意思？相信现在不难理解了

3.2 开发模型与语言模型

软件界不存在银弹，是因为我们站在老百姓的立场说机器的事。永远只是模拟。而不会真正得到一棵银弹。

编译器的定义中其实少了一个方面，**编译器不仅是翻译工具，而且更重要的是形成语言规范才是这个定义的主体**。

在我的理解中，至少在图灵机的层次上，图灵机不是智能的。在我看来，智能的唯一条件是机器有自主意识。

在我们现用的 PC 上提倡英语编程(并用传统的编译理论知识去实作)，那是 SB，因为这是舍近求远的做法，我们的 PC 本来就不是语言机。我们需要对 PC 本身进行改造或重建。

为什么机器模型跟开发模型是紧紧联系的呢？而且历史上存在很多失败的计算机模型(死亡计算机协会)，只有冯氏模型活了下来。

因为语言写出来的软件就是为了系统服务，(比如结构化程序语言符合堆栈机)，所以这个系统最好就是基于某种语言的,(但冯氏模型太屈就通用的低层实现而不是高层开发抽象),语言逻辑即系统逻辑，这多好啊（从这方面看，图灵机既是计算机模型也是语言模型是很容易理解的),,我们的 PC 设计成非语言直接对应的机器，是因为开发虽然对于 PC 很重要，但做其它工作时需要 PC 提供其它通用机制，因此不能仅仅屈就开发模型（比如还屈就机器实现难度上面谈到）。

虚拟机就是在这样的历史使命下产生的，它是软件上实现的机器,是语言模型跟机器模式折中的产物，因此有些人提出它，目的更多地是为了迎合某种语言的提出(实际上虚拟机并不企图提出一套机器标准，因为硬件上很难实现这样理想的机器，我们的 CPU 只能是晶体管认识二个状态),，而且这样的机器不必屈从硬件实现(可以基于某种高层理论提出)，只要给程序执行提供一套执行路径即可，因此也需要 OS 设计中的运行时，进程，中断等等。’’

对于开发来说 PC 史的发展就是一个一直在走弯路的过程,，这个过程中又提倡什么抽象(这是由软件的根本任务决定的因为软件即抽象)，但是太晚了，PC 一开始就被设计成一个远离人类大脑的东西有它自己的底层机制（再抽象的高级语言开发抽象都受制于冯氏的数据加指令模型)。应该把 PC 设计成一开始就靠近人的东西，这样后来的开发中才不需要太扭曲的抽象。

想象一下，如果我们人也是电器化的东西,，也有控制器等等，比如一个生物硬盘，那么当人们被接入计算机时，编程的工作，也就变成了用 0,1 组成代码,，即，机器语言就是高级语言，编译原理这样的知识根本不用去学。因为用不着编译器和汇编器这样的东西。这样的话，如果我们说话表达中存在银弹的话，那么软件开发必然也存在一种银弹。

即如果让 CPU 一开始就屈从于人，而不是冯氏的屈就底层的模型，那么我们今天的开发就不会是这样的数据加语句。

我觉得所谓网络神经电脑的研究是不必要的(我这里说不必要仅仅是对于开发来说无多大影响)，因为那是变更底层，而对于软件和开发者来说，再怎么样的底层都没有用,，我们着重的是能影响开发逻辑的 PC 的高层模型（如何变更底层来实现这个 PC 级的高层模型才是重要的，如果它是能学习的就是真正的 AI 机了)。即开发中我们只需要那些能根本影响开发的 PC 高层模型（而不是冯氏屈就底层的模型这导致以后开发需抽象),,,而且这样的机器不但是“开发机”，而且也是“通用机”。

所以系统必须是基于语言的某种。。二进制，不,，机（必须基于某种低层啊，日,语言数学）

即，面向系统的语言，导致面向语言的系统出现,,因为语言逻辑就是系统逻辑，所以这种机器在产生软件时所需的逻辑很小。软件很快

3.3 联系编译原理学语言

编程的本质在于将语言机机制逻辑转化为应用逻辑，完成它们之间的变换,但是这中间经过了很多抽象很多迂回。

编译原理所涉及到的那些东西，就像 OO 一样，对于计算机来说，完全是一种迂回。

因为高级语言编译器的出现是为了迎合人类能读懂的文本源程序(面向行的编译器),所以它先提出一套语法,为了这套语法就造出了正规式,自动机,最终到语言本身这中间的诸多逻辑

而 OO 和框架(FrameWork) 也是一样的道理,,计算机直接执行的是二进制,,但人类若要组织这些逻辑,就得用 OO 思想....因为这是人类工程的需要(对于计算机来说就是迂回了)

诚然，用 C 和机器的观点来反映应用就够了，因为屏幕本来是二维的我们却要发展出一个三维游戏，,，有时复杂性仅仅是为了跟人靠近而不是屈就计算机，因此这个绕的弯子是值得的，是一种计算机逻辑向人类逻辑的变换,但最终要变换成计算机逻辑,我们只取 OO 的中间变换,,,虽然 OO 最终要变成计算机离散逻辑
因此简单类型是机器的类型,但复杂抽象类型是人的类型,,,计算机的东西是死的,但是却越来越抽象,因为抽象带来的不是复杂性,而是靠近人的简单性,这个要特别明白 ,,,即,相反的,抽象是简单化而不是复杂化

有限自动机是一种抽象的识别装置，,,往往一个文法对应一个有限状态机**(**一个编译器词法阶段**,,**语法阶段不属于这个阶段**)**，所以我们来论讨有限状态机

正则表达式（文法产生式）往往对应一个正则集合，
写一个编译器真正涉及到了设计(把编译器分阶段完成正体现了设计,,编译器理论本身就是吸取乔姆斯基这个不懂计算机的自然语言研究学者研究出的成果，以及后来的一整套编译理论，包括，图灵的机器，正规词法逻辑这些高层逻辑模型),，编码（具体编译器的实现自然离不开编码）,,算法(关于语法的 bnf 设计就是一个大算法,验证图灵机是不是会终止算法),,数据结构 (树与递归频频出现，这个层次上的数据结构还是离散数学意义的离散结构，树啊，环啊，图啊，还没有到具体用什么语言编码的层次，也就是说还是通用的理论层次),,，是一个系统工程，从 C 语言的眼光来看，开发一套编译器这所有的过程，是一套算法加数据结构实现的集中体现（当然 JAVA 也要开发编译器，只是 JAVA 有高层设计工具，比如 OO 等，对于算法加数据结构的体现没有 C 深，而 JAVA 也一般不用作开发编译器，因为 JAVA 对于编译器后端开始的工作几乎力不从心）。。

在这个系统工程前期，，编译器前端(词法，语法，中间代码)大部分理论还是基于编译理论的高级逻辑的，实现方法和图径都比较单一，因为有统一的理论和相关的 yacc.lex 工具等，在后端时(代码生成，代码优化，运行时环境，错误处理和调试)就得进入平台逻辑了。。这里才是发生分歧的地方和可以无限深入的难点所在。。

一般来说，编译原理就是指前端,，因为后端不再属于编译知识了,，而是平台处理逻辑,，至中间代码生成时，已经完成高级源程序到代码的生成了(虽然是中间代码，虽然还要经过汇编到最终的目标语言，不过这些后来的过程不做也可以，因为我们也可以发展一个虚拟机内置解释器执行这些中间代码,,，后端的动作不属于编译原理，因为视具体平台不同,，后来的过程不是统一的理论了,,比如代码优化,，那更是一个没有定论的过程。

正规式和自动机理论统一了词法分析的过程就像大家都用 bnf 来描述文法一样，它让编写编计器的工作变得科学化和合理化，有人说，nfa 是给机器看的，而正规式是给人看的。。这样说比较形象，而实际上直接写语法也是可以的。词法分析可以一点也不涉及到正规式与自动机这样对于人的迂回逻辑模型，代码控制能力强的人可直接写此类逻辑。

什么是词法分析呢，因为我们现在是以"写"作程序的手段的，是面向行(每一行语句都对应中间代码的一个三元式，每一行都是一个编译时给定行标号产生内存地址的)的，因此编译器也是面向文本的,，字符串成为编译器唯一面向的东西,，首先，从一段源程序中读入,，词法分析的任务在于得出一个一个的 lex 为语法分析所有,,，词法分析直接面向源程序文本,,语法分析面向业已分析好的词,,故前者仅仅产生词(语法单元 token),只有语法完成才能产生语言 (文法产生式),而语义给业已建立的语法和语法元素 (即树和节点)增加属性等语义信息,,语义分析过程的一个实现方法就是这所谓的语法制导翻译. 语法分析过后，如果是一遍扫描，那么中间代码几乎产生目标代码，编译几乎完成

而语法分析则采取了文法产生式(而非正规产生式,虽然都是产生串集，但正规式本身表明一个匹配模式，匹配式。。文法产生式指明如何为语言产生一个串，前者产生的语言是词会集，后者产生的是语句集,,词法过程远远没有到底语言的意义,，只有到达语法阶段了,，才能谈得上语言,，因此对于编译原理的几个阶段来说,词法处理只是一个跟语言挂不上钩的很初级的过程)，它接收的输入是词，编译原理语法分析过程产生的输出是串，串的集合即语言，串即语句，不再是词了，而是语法一级的单元,，而词是词法一级的单元,，

正如关于词的逻辑有正规逻辑一样,，关于串的逻辑就是产生式逻辑，关于产生式逻辑有一个就是上下文无关文法(这也就是当今很多语言采取的乔姆斯基的文法了),，而其实存在很多其它文法的

词法分析和语法分析过程都涉及到图灵机，有限状态自动机 dfa,无限状态自动机 nfa,是一种名叫图灵的抽象的机器模型,，它刚好与正规集形成的语言(由一个正规表达式推导出的串集形成语言，这是指词法意义上的语言)一一对应,，即，对于一个正规语言，有一个 dfa 能够处理它 (这个一一对应关系在离散数学中存在科学的证明方法)。

图灵机涉及到图灵状态与此状态相关的处理,,因为图灵机是一种关于状态与处理的

机器，有一些图灵机还提供记忆功能，这就是为什么要在 lex 工具中写一个 C 例程对应每一个状态的原因，，我们说语言是图灵完备的主要是指文法产生式的图灵处理过程.

这个过程中所有状态都可以用一种离散结构(或者称为数据结构吧，我认为数据结构都应该提供数据存储地，而图好像可以没有)图来表示和加以方便地研究。。对图灵模型的研究往往借助这种状态图

语法分析产生一段抽象树，，树这个数据结构深刻并与生俱来地与递归相关，，因此你可以看到 LR,LL,自顶向上，，自顶向下这些概念，，语法分析的过程产生抽象语法树，供下一阶段的语义分析和代码生成过程用。。

语义分析,就是将语法分析过程中产生的符号赋于语言的意义,即语义(比如此符号具有什么类型，而类型是语义上的东西),语法制导翻译（以之前分析得到的抽象语法树作为推导的翻译过程,为它当中的各个节点建立意义）不属于语法分析阶段，，即不属于上面一段的编译原理语法分析阶段，，注意翻译二字导致意义的差别。。如果说词法分析和语法分析仅仅是分析阶段的话，，那么从语法制导翻译和中间代码表示开始，，就进入了实际的翻译的实质阶段了(编译就是把高级源程序转为汇编目标语言，这个总过程称为编译，，编译，编译，有一个译字，直到这里为止，译字渐渐明朗，编译前端完成，编译完成，因为进入了代码的阶段，虽然这里是中间代码表示)，

词，串，，还有一个符号的概念，，词，串是编译原理意义上语言的概念，，符号就是程序意义上的语言本身的概念了，，对应词法分析中的词，语法分析中的终结符等概念，，，这些东西 (不光是词，终结符，而且是语句，相关的语句逻辑块比如一个 for 过程，我们知道在形成三元式时一切都规范化了，成了一行一行的符号加地址的统一形式，) 在被作为中间代码表示时，，就成了实质翻译成的语言(也许此时也不能称为语言，只能称为中间形式)的符号，，词法分析阶段和语法分析阶段绝不仅仅就是分析，，他们还生成和完善以及维护一个记号系统，，前端是一个所有动作的统称

这些符号记号在前端被不断完善它们的属性值，比如变量类型啊，，关键字 ID 啊，变量编译后的地址啊，

中间代码表示则着眼于产生的串，如何用中间形式(源程序到目标语言的中间形式)表示一个串(一个符合语法规则的句子，),,一般采用三地址格式（这是个形式化了的中间形式表示的串），比如对于算符优先的规则，，一般采用波兰后缀，或逆波兰形式。如果前面分析阶段主要是树和图的方式来说明和处理，那么这里栈式处理数据的方式在这里频频出现。、

所谓的栈式机概念在这里出现

为什么一门语言的类型机制如此重要呢，答案就在符号系统。

汇编原理说明了机器语言的逻辑。。编译原理后端的那些知识说明了语言代码如何映射到机器。。通过该语言编译后端学习此语言是学习的比较好的途径

3.4 理解编译原理中的一些至关重要的二义性

编译原理中的一些二义性

理解编译原理的时候，有一些相似但本质上其实根本不一样的东西，，我们作一个归纳和分别。

凡是正规表达式能匹配的地方，语法分析器就能对他们进行下一步的处理。也即，词法分析器是语法分析器的开路者，因为它们基于同样的过程（都是识别出可用的字符串 - 不过词法是识别出由正则表达式简单标识的“标记”，而语法是识别由“标记”本身逻辑上形成的“符号（或组）”而已，说前者是简单标识出是因为只有一个正则表达式，说后者是逻辑上形成是因为有更高级的语法产生式逻辑），而后者正可以以前者的输出作为基础（因为它也是识别嘛）。

词法分析只是简单的逻辑，它面向粗糙的文本，最终结果是导出一系列可用的“标记”；依据的是正规表达式；而正规表达式只是一种简单的标识匹配逻辑，并非一种高级的产生逻辑(实际上我们呆会会谈到，所谓的产生式逻辑也是归约移进之类的东西而已，而这样东西，其本质上也是一种标识和匹配逻辑，不过我们把它看作更为高级的产生逻辑，因为这里会出现数据结构和算法上的东西)。

语法分析也只是机器的逻辑（语义开始就是人的逻辑了），比如它把词法分析得出的“标记”进一步识别为“产生式”；依据的是语法产生式。我们说了，语法产生式不过也是标识和匹配逻辑而已，不过在这个过程中，除了匹配规则之外(当然不是正规表达式了而是语法产生式)，，还涉及到一系列数据结构和算法上的技术逻辑，所以把它看成更高级而已。

所以这二者本质上都是机器逻辑下的“识别”，是一种形式，只是简单的匹配替换逻辑。不过后者比前者要高级那么一点点儿了。当然，仅仅就因为这个“更高级一点”的说法，**这二个过程实际上就变得本质有不同了，，前者是非语言级的，而后者进入语法阶段了，**所以属于编译原理了的前端了(其实词法分析可以是编译原理的一个可有可无的过程，可以不依据正规表达式而直接写提取出串的逻辑，这就是手工写的代码控制的文本处理逻辑了，因为语法分析可不管你前面是词法分析还是什么其它过程，它需要的标记能拿到就可以了)。

而语义呢，就是不是简单的产生串的集合了。它是把串加入了逻辑(语法制导翻译)，不

过
就跟词法分析的匹配和语法分析的匹配是二个不同的过程一样，这里又存在一个二义性。

即在语法分析的阶段也出现过一次“给串不断加逻辑”的过程。

我们不妨先来严格区别“串"，“标记"，“符号”这三者的概念，串就是字符串，词法分析一开始需要面对的对象，是简单无意义的东西，标记和符号就是逻辑上的具有意义的“串"，其实它们本质上都是串或串的组合，我们能区别标记和符号是因为“标记”是词法阶段的“串”的说法，而“符号”是语法阶段的“串或串组”的说法（当然词法分析中只有串，而语法分析可能是串或串组）。在接下来所有的讨论中，我们规范说法，当我们说到语法分析就会说到与它相对而言的下一层即词法分析上的“标记"，当我们说到词法分析就会说到与它相对而言的下一层即源程序文本中的“串"。语义分析，我们最后说。

所以说，所有的标记和符号都是串或串组，不是所有的符号都是标记，但只要是标记(当然，不包括词法分析中忽略的串，比如源程序中的空格啊，什么的)，它就是符号。
我们再来看下语法分析和我们平常的英语的对比。

我们抽取英语中一个子集来讨论，
比如英语“语法”中有“主谓宾”这样的“句子"，和“主谓苹果”这样的“句子"。（注意到我把一些词加上双引号了）

因为在我们所处的年代，大家都是通过用键盘写源程序的方式来写程序的（没有人是用嘴说程序的吧？，这也就是大家说的“面向行的编译器和编译原理了"），所以当我们在源程序中写下苹果的时候，就想到有朝一天它会被词法分析器拿走，苹果这个“字符串“被词法分析器作为“标记"，此时它就是终结符。而像上面的主，谓，宾之类的就是“非终结符”了。非终结符可以表示某个中间语法产生式。

“非终结符”和“终结符”是单个的“符"，非终结符就是一种语法产生式中的左边的临时符号（相对最终语法产生式“句子”来说）而已。它代表一个由串组组成的长语法产生式

泛义上的“句子”的形成就是一个根据“语法”不断组合替换“语法产生式”的过程。
所以“句子”可以是：

主谓宾
I 谓宾
I 谓苹果
He 吃宾

He 吃苹果
(当然，英语中的句子不会像上面那样不“严格”，实际上英语中的句子要严格得多)

而“句子”这个概念是我们最终要得到的。在语法分析中，就是起始状态(终极的语法产生式，最终需要的语法产生规则)。

词法分析中的“终结符”被直接送给语法分析作为“符号”。而它产生的非终结符要被语法分析，当然，词法分析器不知道它是不是“终结符”或“非终结符”，这二个概念只能是语法分析器眼里对不同的“标记”产生的概念。

在词法分析过程中，会给“标记”加上不同的“词性”，比如它是“名词”，“副词”，还是动词（我们知道不同的词性可以做不同的句子结构），当然这只是词法层次的“串属性“，体现在后来的语法分析过程中，就会产生语法层次的不同说法的“属性”了。

主，还是谓，还是宾。这是语法分析过程中，会给串加的“语法”属性，而词性相对而言只是词法范畴会给串加的串属性。

在人类的编译器技术中，词法分析和语法分析，以及语义分析，
从来都是用“标记”符 ID 的属性来说话的，即“符号表”里的那些内容。语法分析过程维护一个符号表。每个表项都是一个“词性”，因为它需要词法分析产生的“标记”的这些相关内容。（对应于编程语言中变量的类型）

语法分析进一步给这样拥有“词性”的“符号”加上语法层次的属性。比如如果标记符 ID 是一个名词，，那么好，语义分析过程之后，给这个具体的名词又加上了什么样的句子结构属性呢，比如它是作为主语还是宾语，这个不能混淆，否则语法逻辑上就开始出现主宾不分了，更遑论在语义分析中会出现错误了。

所以，词法分析实际上是一个“文本制导匹配”，
语法分析实际上是一个“词法制导产生”

给上面一句话接个下巴，语义分析正是针对标记符在语法分析阶段的那些属性进一步给它加上语义级的属性内容。所以是“语法制导翻译”。（注意到了吗，匹配，产生，翻译这三个词在措词上的巨大区别，所以，词法是词法的，语法是语法的，语义是语义的）

什么是语义是语义的呢，比如“左值与右值”的区别，

```
int main()
{
    //语法上可以这样“写”，但你的思想里决不能这样“认为”，即语义不是跟语法一致的。//因为语法是语法的，语义是语义的。
```

```
    //为什么呢这样说呢，看 printf()的二个例子

    char * mychar = "dsfasdf";

    //首先，它是一个数组，其中每个元素都是 char *
    printf("%d\n",sizeof("dsfasdf ")); //大小为 8

    //而它是一个指针
    printf("%d",sizeof(mychar)); //大小为一个指针长，最长为 4

    return 0;
}
```

这就是语法跟语义的区别所在。。

3.5 如何理解运行时

编译前端负责一直到中间代码生成的所有动作，至此，实际上编译器已经差不多完成了。但是还差一个内容，就是代码运行环境问题的解决，如果是针对机器加 OS 本身来运行这种中间码，那么可以用运行时也可以用解释器的方法来进行。也可以发展出一个专门的虚拟机器，再把这门语言的中间码生成为虚拟机的目标码(此时再考虑是运行时还是在虚拟机内做一个解释器)。

编译后端的中心任务是给前端生成的中间代码提供一个运行环境，这导致了运行时的概念，当然，编译后端，除了给代码提供一个运行环境之外，还指代码优化之类。

所以，什么是运行时呢，就是，不跟具体编译前端有关的，跟具体平台有关的(虚拟机或本地)那些编译原理知识，给前端生成的中间代码在上述二平台中提供一个运行环境
如果给本地平台就是 C，C++这样的运行时或解释器，如果是虚拟机，就是.net,java 那样的。

(以上道理对理解模板元编程有很大的帮助，比如模板元编程就是把编译器实例化机制当成了运行时的机制，我们以后会谈到)。

编译后端会对目标平台产生代码，这目标平台包括机器和 OS，最主要还是 OS，因为 OS 是封装了系统调用（BIOS 中原子基本的几条 IO 功能接口，可供 OS 和系统进行以后的抽象）的，在语言看来，封装了的 OS 和语言的其它库一样，也是一种语言逻辑，有接口就可以调用(语言跟 OS 并非一种鸡生蛋生鸡的关系，往往是 OS 出来了，然后有

一种特定运行于这种 OS 下的语言实现即编译器，tiny 甚至可以自编译自身)，

那么如何在 OS 下运行这种语言写成的代码呢，它如何从 OS 中获得空间等运行资源，如何获得 system call IO 操作文件(比如 c stdio 中的文件操作实际是封装了 system call io 逻辑来的)呢，把运行时想象成语言后端逻辑(编译后端代码生成器及其生成的代码)跟 OS 的逻辑接口就行了。对于 OS 来讲，它解决的是 OS 如何运行它，对于代码来说。它解决的是如何从 OS 中获取资源进行运行，和提供 main()这样的接口，，当然特定编译器商甚至也包括一些语言实现 std lib dll,因为这毕竟算是开发完整一套编译器嘛。。一般会把 std lib dll 当成 crt,这是不对的和不完整的。

其它语言是没有 crt 的只有 c 有，因为 C 有标准编译器实现必须实现它,这个意义上 run time 是语言的库逻辑,run time 是编译后端，是每种语言包括无标准库的语言都需要的，这个意义上它是系统软件是语言对于系统的接口是编译器后端一套完整编译器实现必须的,（我们知道解释器一般是系统软件，作为编译后端的意义的 crt 是相当于解释器对于一门语言的意义）

从解释器的角度我们来看一下，的确是这样的，解释器不为特定目标生成代码，它只负责运行语言源代码(虽然这其中会有一个中间代码，但解释器不运行目标代码，它的目标就是解释器自己，目标代码就是中间代码),，解释器就相当于机器和 OS，它没有 c runtime,,编译环境下的 crt 相当于额外的中间层。。

再者比如.net 的通用 runtime,也是这个道理.

3.6 运行时环境

运行时其实不如叫做运行空，就是程序运行的物质环境，比如 CPU 架构（里面的寄存器，就是专门为通俗意义上-不只是一门语言的程序，，，的程序而设置的最终运行时，里面的部件全是程序专用的）,，当然，运行时，也可以是虚拟机这些软件逻辑上的东西，，其实程序不可能纯粹以硬件作为运行环境(除非电器化的微指令,或者裸体指令，运行时不是纯粹指 CPU 构架，因为单纯一个 CPU 提出来,只有指令，并没有程序的概念，必须等 OS 机制，高级语言的汇编原理出来,然后程序机制出来，才发展出运行时这个说法),还是有 OS 逻辑来封装 CPU 硬件逻辑，然后为运行时所用。

3.7 运行时

编译时是编译期的事情，运行时是运行期的事情，这里的时可以理解为期间（运行期间），也可理解为空间(运行逻辑关于使用 reg 还是 stack 的逻辑),

一门语言同时提供编译器，和这门语言实现的运行时解释器,一个编译逻辑，一个面向某平台的运行逻辑(针对本地机器的一般称为运行时，针对类 JVM 机器的一般称为解释器，所以 jvm 跟解释器是分开的，一个是平台，一个是平台下针对 Java 代码的 runtime 实现)，

在学习 C++的运行期动态 OO 对象时，我们也要学习一个称为 rtti 的东西,,一切高级的语言机制，都可在运行期探求它的平台实现细节，如何用 stack 内存和 reg 展开,。。这样才是深克理解了高级语言该机制(因为了解了某一平台下具体实现的细节，而且是最细节的汇编逻辑，因此该语言抽象也被在高层次被体现并理解了)。。

比如 C++类的成员函数，它实际上是一种变态的函数。。从他的汇编逻辑中可以看出来，，在压参时还压入一个 this 指针。从这个眼光来看。跟普通的 cdel,fast,pascal 函数都不一样
所以说，函数作为一种机制，有不同的实作品。

runtime 的意思在这里进一步明显，，实际上不只执行函数要用到 stack runtime 和 stack frame，在执行诸如堆栈队列数组链表树这些高级数据表示与组织的内存逻辑时(即运行时逻辑，这也属于运行时逻辑)，，不一定直接用到 stack runtime 机制，，虽然执行函数时的 stack 机制的是代表机器就是一种堆栈机运行时的典型。。

3.8 运行期与编译期

编译器的工作在于将高级码转化为平台（这里的平台之意主要指ＣＰＵ）目标码，但是现代的编译器和ＩＤＥ还加了一些高级单元，如汇编器，装载器，重定位器，链接器，调试器，这些的功能来直接生成可运行文件.
一个编译器，如ＷＩＮＤＯＷＳ上的ＧＮＵ，ＶＣ，ＢＣ等等，，对应于一个特定平台，编译器实现，链接器实现都是平台相关的（这里的平台主要指ＯＳ）往往有一个ＲＵＮＴＩＭＥ　库，，这个库是什么呢，，一个ＯＳ一般由 C 编成，这些 C 的库一般向程序(向编译器编译出来的，往往是用户空间的程序)提供ＣＰＵ进程，，线程，ＳＯＣＥＴ等资源，运行库就是提供这些访问功能的地方，这就是编译器跟平台发生联系的地方，一般一个编译器都有它对应于某个平台的运行时库..
那么什么是编译期呢，什么是运行期呢？什么是编译期静态，什么是运行期动态？什么是静态语法错误，什么是动态逻辑错误呢？ＲＴＴＩ与ＲＵＮＴＩＭＥ有什么联系，为什么有 new 这样的动态分配内存.
编译期静态和运行期动态主要是指类型来说的，

那么泛型编程跟模板又有什么关系,一门语言的语言机制必定可在它的编译器实现找到答案.那么模板技术在编译器中是如何实现以支持泛型编程的呢（我们会在抽象之高级语法机制那章中讲到）

我们知道，一等公民在一门程序语言中是重要的，C++不支持数组的直接拷贝和赋值，不支持对象的直接赋值，不支持范型类型，这就是因为对象和数组是 C++用其它方法抽象得到的语言机制，比如按位拷贝这种机器很擅长做的事，其实范型编程是可以用很多方法达成的，C++唯独用了编译器的＂参数化类型＂作法来实现模板再由模板实现泛型编程，而ＲＵＢＹ等语言有它自己的范型编程做法，，基于对象编程跟面向对象不同，不要以为你会写 ＣＬＡＳＳ就是面向对象，如果没有用到ＯＯ的继承和多态，你同样是在写基于对象的东西(比如 OO 不但作为一种抽象机制，它还是一种接口机制，类的成员函数是接口，类本身也是一个接口，你需要用 OO 的 private,protected 这些接口控制机制使一个类足够内敛，方使多人开发的子程序协同工作，使一个人的工作不必涉及到另一个人工作出来的类内部，比如不宜在里面直接写实现逻辑，要一层一层地下放到它的子类中，**如果你还不用 private,protected，那么你实际上在面向过程编程，因为这种情况下，class 只是一个 public struct 而已**)。。

3.9 编译与运行期分开的本质与抽象

其实字符串逻辑，数据结构逻辑都可以完全不跟内存有关，语言都可以站在一个比较高的角度来抽象，甚至如果可以，一门语言可以把字符串抽象为 Rope(当然还可以是其它东西),因为语法级是设计抽象，可以不跟运行期有任何关系，，编译期后端才负责运行的事，才有运行效益的说法，所以语法级，也就是设计期可以站在一个完全脱离运行逻辑的抽象高度上进行对字符串的抽象，因为语法是 C 语言，C++，Java 这样的高级语言支持的那些写法和映射能力，它只负责这方面的高级逻辑的事，而编译前端跟后端是可以分开的，只有到编译后端时才需要映射到汇编语言这样的机器逻辑，

比如 C 用底层来表达语法级的设计抽象，，比如它将字符串看待是指针数组，这就是 C 语言跟其它所有语言不一样的地方，，因此它产生的运行期抽象跟语言级抽象最接近，虽然是运行抽象但几乎等同设计，因此可用在内存有限的地方，因为它抽象小，比如手机等特殊平台上，但即使是这样，C 的抽象能力也是巨大的，它可以抽象 OOP，抽象结构，抽象 Windows，抽象。。。

而相比之下，C++直接在语法级支持更大的抽象，它沿袭了 C 的 Core（流程控制什么的，C 被称为最小内核语言）,但在比较大一点的抽象上它沿袭了 C 的指针和预编译这二大抽象模块，并自己发展出一个运行期的 OO，接着面向运行期的模板可以产生 STL 这样的泛型抽象集，面向受限编译期的模板可以产生 Boost Mpl 库这样的元编程抽象集，而其它的第四代语言比如 Java，Ruby，Lua，Python，它们在语法级直接支持的设计抽象就更多了，因为它们不需要像 C 一样处处屈就运行期，，而是屈就人，想怎么样设计就怎么样设计，至于运行期，，完全是虚拟机的性能和对语法机机制的实现支持问题。。

以上可以很好地解释 C++的模板泛型和元编程。我们将在以后的章节中讲到。

3.10 面向类型化的设计和面向无类型泛化的设计

以数据抽象动作为中心的设计，以类型为中心的语言。这是冯氏下编程的特点

有没有注意到呢，编程首先是将概念进行数据化，以至于将那些不是实体的抽象概念都数据化 — 比如设计 STL 时将迭代器 Template Class 化，，还比如指针其实是一种企图把数据和代码整合成统一的数据的手段。即 code handle 数据类型，脚本语言的变量中，数值可直接表示科学运算用到的元素，其它 UDT 和 ADT 可表其它高级的 DSL 抽象表示。。也即类型。

类型就是数据（的模板），编译语言严格这个过程，脚本语言提倡类型即数据。我们可由类型定义出子类型，，也是一种数据类型。。比如 Typedef int myint,,,这在 C,C++中广泛使用。

类型即内存中编码了的“计算机数据”，即设计者眼中的“数据”(UDT)，（即“ＯＯ设计”实际上=“用ＵＤＴ表达设计，用数据描述设计”）在很多地方，类型化正是设计的死敌，因为它规定了目标对象的产生模式，在整个设计中充满着错综复杂的对象头文件引用。数据间的复合进入设计一旦被固化，用数据表达的设计容易造成偶合。

而这正是范型产生的最佳替代。，范型将设计中出现中的数据默认为一个空的占位符ＵＤＴ。无论是ＯＯ还是泛型开发，都是以数据封装动作为中心的。而过程化设计是以函数封装为模块为中心的。

即第一点：ＯＯ以数据封装为中心，这会造成偶合。

第二点：OO 容易产生过度 OO 化的负影响，因为极容易将那些不显式的设计元素封装为对象，这反而带来了负面影响。

第三点：而 OO 的三重机制中的继承，更是

在抽象类型作为设计手段的方法上，OO 和模板是差不多的，然而不同的是模板是“泛化”类型 OO 是纯粹只用“类型化”来实现设计（对即逻辑的抽象逻辑，我们知道，实现中也离不开设计，设计库的设计就更离不开设计了），，所以 OO 的设计手段不足，而设计模式是一种新的设计理念，语言并不直接支持，，C++用 STL 作为语言的数据抽象，，用 LOKI 作为语言的设计能力（OO，模板相比来说是小规模的设计手段）。

注意，OO 的类型化绝对不能说成对象化，OO 定义成面对对象是极其错误的译法，，OO 是用于设计的，，所以 O 这个字眼只能是“类型化”的“类型”，而不能译成对象，，因为对象是实现里面，不是 OO 设计的原义所在。

3.11 语言的类型系统

高级语言引进类型机制，其意义是重大的，一方面，类型直接抽象了现实生活，比如数值，字符，这使得计算机可直接用来作科学运算，另一方面，广义的类型是程序员用来操作内存的规则，变量是程序员用来操作内存的手段而不管编译原理那门课中的东西，而变量是建立在类型机制上面的。构建在类型上的数据结构就抽象了计算机存储机制和它能表达的现实事物。程序员可用这种抽象来产生更深层的逻辑和解决计算机能解决的问题(算法是针对数据结构的撒，如果说数据结构是计算机的结构，那么算法就是用来解决计算机问题的，数据结构和算法是计算机的而不是编程语言的科学，这个说法就来源于这里)。

对类型的进一步抽象是很多高级语言在做的事情，比如动态类型语言，比如类，范型，数据结构等。。特别是类的提出，是一种通用类型，这使得类型不仅仅只表示数值和字符，还可以表示更抽象的概念，

编译原理告诉我们，类型机制是编译期的一切，，关系到变量表示这种类型，关系到编译期类型安全和转换机制，关系到一门语言组织更大数据抽象的能力。关系到与它相关的表达式的属性。

类型跟变量的区别不要混淆.. 类型是类型,是编译器的概念.. 变量是你从类型搞出来的,,是程序的概念

动态类型语言不是没有类型，而是其变量在运行时可以自由转型。范型是建立在为一个通用类型上的操作的手段。类是一种定义类型的手段 UDT。。。因此除了简单类型之外，还有自定义类型。。否则仅仅靠建立在基础类型之上的复合类型还是不够(C++提供 Class 这种 UDT 并导致了基于对象和面向对象，这是 C++区别于其它语言最尤为可以拿来类比的特征)。

我们可以从一个很初级的逻辑来讲解类型与变量,在没有计算机出现之前，是用算盘这样的工具来计算，但算盘不能处理，一个世界跟一个猪如何交互，它只能处理 1 加 1 等于 2。。这就是类型产生的必要。。因为计算机不仅能抽象数值，还能处理人类想象出来并能通过计算机表达的其它抽象。。当然，这一切都是抽象了的。。但算盘根本无法抽象。。所以在计算机开发领域，对类型的抽象是必要的。

1,动态语言是指语言的运行时,其运行环境是动态的,,新的函数可以被引进,,等等

2,动态类型语言,是指类型可以运行期被改变的语言,,一般来说,类型系统是一个语言的特征之一,如果它都可以是在运行期是动态的,那么该语言就是在运行期动态的

3,弱类型语言,,,语言有类型,,但是类型之间的转换并非严格,,,字符串指定可以转型后用作数值型

4,无类型,,,无类型是指类型是不需要显式在写代码时声明给它一个类型的语言,,在运行时视给它的值确定类型,而且还可以再变动,即所谓的 duking type,,,因为没有类型,,所以就没有变量(因为变量就是类型的代表撒),...没有变量只有值,,,值决定一切,,给它一个鸭子走的动作,,如果它像,,那么它就是一只 duck

数据结构与算法是属于计算机的，而不是程序设计语言的。更多在出现在计算机系统实现上。比如操作系统实现，编译系统，计算机图形上。

3.12 流程控制

大多一门语言都提供了流程控制,,形成了各自的语言关于流程语句的语法要素(更高级的跳转机制有异常，等),,语法是一种形式(所以称语言是形式语言),,语义是一种意义,,,

其实循环,跳转这些东西来自于汇编语言,,,高级语言接纳了这些理念(因为汇编语言几乎就是机器逻辑,高级语言提供类汇编语言这样的机制是为了符合平台逻辑,,,况且高级语言最终要编译成汇编语言和平台逻辑,,循环语言要最终被还原成汇编语言的形式,这些处理速度就可大大加快),,,发展出了它们关于循环,跳转封装了的语言机制..

C 语言最终要编译成汇编语言和平台逻辑,,循环语言要最终被还原成汇编语言的形式,,这就是调试的由来,,,调试分语法级调试和语义级运行期的错误.

对语言机制的理解，应该始于编译后端，再终于其语法对应的语义。为什么控制结构如此重要呢，因为它代表语法和语义方面的要求.比如布尔条件式，语义允许短路的语言会直接编译出条件判断后的结果。。有时如果语法语义允许，语法上布尔运算也可等同算术运算。。因为布尔结果用数值可表示。。C 语言就是这样

C 语言字符串有大量指针这说明了与 C++的抽象字符串不一样，说明 C 是用底层来描述问题和设计的。

这些说明，要更好地了解一门语言,,最好要上升到他的语法规范。联系编译原理知识来了一解一些东西。以及编译实现时是如何满足这些规范的。

for 的条件表达式可能短路,,它的 update 部分可以是任何算式，甚至不要 updata 部分一般把最常发生 case 的情况放在最后面，如果你学过汇编就知道，编译器是从后到前搜索的

注意汇编是没有逻辑运算符的，只有移位运算符(而往往 C++中把它们同用)

有三种,,逻辑（与，或，异或，取反），移位(无符号左移,,右移,,高位补 0 等等),

在汇编中

contine 和 break 的区别，contiune 放在循环语句的某一具体层中，当满足条件时，继续执行下一个当前循环(即重新判断条件)，而 break 可以放在一个循环的任何地方，是跳出当前循环(注意因为循环可能是嵌套的，所以这个当前循环是指 break 所在的那一个嵌套层次),,然后继续执行这个循环外的第一条语句(即结束循环)。

C 语言中的信号是什么东西呢，,

是一种高级的跳转机制，语言的流程控制结构

3.13 所谓函数

函数机制往往被作为过程式语言的代表，但其实无论是 C 这种纯正的过程式语言，还是 RUBY 这种 OO 式语言，，与其说函数它是一种语言机制，倒不如说函数是一种接口，更多地来说是一种过程式非过程式语言通用的机制,比如 class 里面其实也可以是过程式编程的函数调用，一种具体的构造逻辑的接口形式（一种单入口多出口，而且可被其它函数无限调用的接口,因为它是一种最接近计算机单路离散算法的逻辑，，这也就是 C 语言采取的自顶向下的形式，,所以实际上这种简单的形式可构造一切计算机逻辑,就是这种机制，，决定了函数实际上可构造一切逻辑，包括被 C++用来构造类机制，注意函数有一特点就是它的返回并不决定函数一定就结束），，虽然这些语言谈到的函数都不是一个意思(C 的函数是通常意义上我们说的函数,语言直接支持的第一等公民，而 RUBY 的函数是另一种运行时支持下的函数变形，比如虚函数，成员函数，COM 接口，RUBY 并不提倡用函数为主体进行编程)

从源程序到机器码经过了编译和汇编，，这二个过程都是单向过程，对于纯编译语言来说，不可能将机器码还原为源程序，，对于类 Java 的半解释半编译语言，，有一些工具跟据 jvm 解释器的逻辑，可从中间代码还原出源程序的形式

反汇编的过程是一个模拟还原的过程，这是一个静态过程(可模拟得出 pe 载体的机器码，全部函数级算法逻辑，注意，这个过程并非 hex 查看,,hex 查看是打开 pe 载体，以十六进制形式查看这个可执体进行修改)，,有的工具在反汇编方面做得很好，如果 ida 内置的反汇编器甚至可得出函数名级的标识，而 dumpbin 做为另一个反汇编工具，仅仅做了初级的这方面工作，比较它们对同一段程序的反汇编结果就可看到，，不同的汇编器对同一个东西得出不同的结果说明反汇编是一个模拟的过程，，影响 PE 逻辑的最终还是 hex 修改 pe 本身(PE 载体在外存磁盘上会有一个载体，因为它内部按外存线性地址划分成了几大块，所以被 map 到内存中时，这些块的相对偏移还是不变的)

如果说调试器是静态过程(分析了一次就形成了结果)，那么调试的过程是运态模拟运行的结果(调试器维护一个活动的逻辑过程环境，可根据你的输入和程序逻辑作出反映)，这种方式相比反汇编来说，，是一种更加好和互补的寻求 PE 逻辑的方式

无论是二种方式中的任何一种，可看到，都用了函数作为十六进制级反工程逻辑的最高代表，所以说函数是分析一切机器逻辑，计算机语言形式体现的计算机逻辑的最好接口模块，，当然也是构造的最好方式，如文章开头所说，，

函数有三个部分，，一原型，二类型，三，参数，对一个函数的把握要从以下几方面进行中

首先，原型部分，各个编译器在编译并汇编相应语言的同一段代码时，会产生相同的反汇编序列(这就是 progo 原型，这成为在汇编逻辑中发现一个函数开头部分并借此发现这是一个函数逻辑的方法)

我们知道基本的原型有 C 的自右向左和 pascal 的自左向右方式(其中，由于 C 函数原型方式下，将执行权交由调用代码来完全，这使得被调用函数中可出现可变参数)

3.14 所谓流程

高级语言在封装机器汇编逻辑上损失了一定的灵活性(因为只有汇编才是与机器逻辑一一对应的)

而高级语言的流程控制，分支逻辑等，，终归是某种抽象品，，只能提供有限的 If Else 形式，这些封装了的高级语言关于流程的逻辑(其实判断，循环都是流程控制逻辑)

这就是封装，抽象，与原来可获得到底层灵活性的矛盾所在

第一个 if 往往是最基本的条件逻辑,else 是一种变相的 if 逻辑，是针对于已提出的 if 的反面，，是 if 正好相反的条件,而其它的 if,在一个 if 存在的条件下，，相当于 else if

一定要明白这里的区别，，这些语言逻辑产生的对应的汇编码的绝对性决定我们得明白这些细微的差别

理解这一类汇编逻辑时，我们得理解 intel 平台的逻辑，条件指令逻辑，，和分支逻辑这二大部分
典型的有，高级语言的条件逻辑转化为汇编逻辑时是它的倒装形式，而且 else 部分是放在所有分支逻辑前面的。

明白这些，将有助我们理解高级语言汇编出来的逻辑，从而更好明白高级语言的这些关于流程的语言机制。。

3.15 数据类型和数据结构是二种不一样的东西

一切语言机制都是为了抽象,,,抽象真的有那么重要吗??

对数据的抽象必要吗,,,

什么是真正的数据 ,,什么是抽象了的数据,,,,数据类型就是数据的模板 ,,计算机和语言用了什么样的形式语言的形式去指代和控制他们?

数据类型 Type 是高级语言对汇编直接操作位的“位”的抽象,，而这句话中的“操作”,，也被高级语言抽象为施加在这些类型上的操作，比如，对于整型数，有相加，但是不能对整型数执行浮点数的运算，虽然程序员方面是直接使用 Type,但在编译器方面，其实对应的还是位，编译器为程序员隐藏了内部逻辑。使我们的编程工作能维持在使用类型而不是机器语言“位”的高阶抽象层次。

一切都是抽象，数据类型的提出本身就是一种抽象，而至于提出什么样简单类型也是一种抽象，数组，指针，都是 C 的复合类型，实际上 C 只有 int,float,char 这三种类型，高级语言提出这三种类型是因为这三种类型抽象了我们的现实事物。比如 int,float 对应数值意义的抽象，char 对应字符意义的抽象。。

而算法和数据类型是建立在 type 和施加在 type 上操作的更高级抽象。。是语言，大凡具有类型机制的高级语言通用来的，用来设计应用和解决问题的方法。。

算法提供了一种用语言来进行设计的手段，是实现抽象，当你不知道如何实现一个程序时，先找到数据结构，自然就找到了算法，编码之后程序就实现了,，就是这个道理，所以说数据结构和算法是通用的语言用来进行设计的抽象惯用法。如果说数据结构是非语言级的设计抽象(它也是一种实现相关的设计抽象)，那么高级语法机制就是语言相关的设计抽象(是一种代码相关的设计抽象)，而我文章中最后一部分谈到的设计就是工程相关的设计抽象(相对实现相关，代码相关，这似乎可以称为结合了二者的综合的设计抽象了。)。。

首先，数据结构在 type 的基础上进行抽象,，它看不到汇编的位，只是考虑如何将 type 翻来复去地变换形式进行使用，而算法，看不到汇编的指令，只是考虑如何用高级语言的语句来操作这些数据结构,“即算法是对如何操作数据结构的抽象”因此数据结构和其上的操作称为 adt,

一句话，type 和 type 上的操作是抽象汇编的,，那么，数据结构和其上的操作是高级语言站在 type 和 type 操作上的抽象，一者是高级语言面向汇编，一者是高级语言面向高级语言的 type.

函数是这里最能体现这种抽象的机制，首先，函数接纳实参或数据结构这些抽象，，函数体内的代码是“施加在实参上的操作的整合体”这样一种抽象。。

3.16 为什么需要变量这些东东

Java 中一切对象都是引用，引用即地址引用，操作对象只要操作对它们的引用就可以影响对象本身，因为引用本来就是对象本身存储在内存中的物理表示嘛

地址是变动的，因为一块内存在不同时刻可以存储程序中用到的各种数据，因此称这些数据为变量，数据的本质是类，构造函数就是实际为某种数据分配内存的过程，

在程序中我们常常需要作数据的移动，复制，比如函数的形实演绎

实际上，我们根本不需要操作数据本身(在面向对象的范畴里，类对象就是唯一的数据，比类的成员更能代表数据)，因为往往有时候这样的代价太大了，传值的方式就是直接复制一份原来对象的新的对象(需要给产生这个对象的类一个复制的构造函数)，这样就会产生一个临时的复制体，现代的应用中，一个应用中有成千上万个对象是很常见的(因为往往是用某种大型库写的)，，给系统巨大的时空开销负担，而且很不安全，在没有提供垃圾收集的语言中，涉及到二次删除的问题

而引用方式就不需要作频繁的复制，因为从抽象上来讲，，引用的确可以代表对象本身（因为对象是程序的，而引用是计算机的，引用本质上就是对象嘛），

另外一个方面是，当我们用逻辑操作符比较二个相同的对象时(一个对象和它的复制体)，返回的结果居然是 False,这是因为实际比较的是它们的引用，，，由于第二个比较对象是原对象的复制体，因此它的地址即引用是与原来的对象不同的，因此比较结果会是假‼

GameObject& rgo = *pa; // rgo 的静态型别是 GameObject，
// 动态型别是 Asteroid。
为什么说动态呢，因为=赋值时就经过了一个隐式的转换
dymic cast 就是对动态型别的转换
注意此时 rgo 就是 gameobject 类型，，而不是 asteroid 类型，，也即，，一个变量的类型永远是它的静态类型而非动态类型
这是什么呢？ rgo 被声明为一个指向 GameObject 的引用(引用就是引用，，而非指针，引用就代表对象本身，取地址操作符不是引用，这二个东东根本不一样，虽然可以通过对一个对象取地址就可以将它变为指针)，，但其实它就是一个对象_____即 rgo 是一个 GameObj 对象，因此说 rgo 的静态型别是 gameobj
point to someobj 跟 ref to someobj 是不一样的

refcount 就是实现对引用的计数的，
由于引用是变量的别名，，所以，引用并不是一个地址值，而是变量名，，，，注意
必须对一个即存变量进行它的一个引用声明，而不能对一个常数进行引用声明

用计算机的眼光来看是内存地址，用人的抽象来看是索引工作。。而无论是顺序数组，还是链式，索引，散列，都是通过某种抽象形式(下标，，全局表，函数)来最终寻找到内存地址。。。
这个道理就像，我们通常不用&变量的形式来获得指针的意义，而是用*，因为&是面向内存的，而*是面向用户的抽象.

引用就是解决指针由无义数字到有义名字的问题，是一种由底层向人的方向抽象的过程

3.17 脚本语言

net 的虚拟机就是把所有的语言不直接放到 CPU+OS 这个二次本地上了(纯 CPU 是一次),而是放到了同一个虚拟机内这个三次平台之上.因此各种语言规范写出来的代码一者相对另一者来说是 native 的.可以共用一个类库,而 Ruby,Java 由于是从 C 发展而来的,它们只视 C 为 native,如果是其它代码要为它们所用,必须改变别的语言中的函数入栈压栈规则,,,通过一种 bind 的技术改造..IT 界的整合与分离无处不在
要知道什么是脚本语言,就必须知道什么是脚本程序,脚本程序简称脚本,是一种程序代码,它是由某个系统(如操作系统)或服务器(如 Web Server)或应用程序(如 AutoCAD, MS Office 等)环境提供的能自动化操作该环境的功能的指令执行序列.而为了合法地编写出脚本程序所制定的形式语法和语义的规范,原则,定义等等总和形成了脚本语言.脚本语言实质上其实是一种用户接口.现在用户接口这个词,多指图形用户接口(GUI),即指软件的视觉和使用设计必须符合用户的习惯.但用户是有层次的,一部分是普通用户,他们通过软件提供的类似菜单选项,工具条选项,以及使用说明和简单配置等功能,来完成他们的日常工作和任务的,这种用户也被认为是最终用户.而另一部分的用户不仅使用这些一般的功能,而且希望软件随时能按照他们的意愿定制使用在菜单工具条选项中不能提供的功能,或者他们能在这个软件平台上做二次开发,将开发后的产品再出售给自己的客户或者自己内部的其他部门.为了能够满足这部分用户的需求,软件开发商就推出一种编程语言,让这部分用户通过编程的方式来使用软件内部提供的功能.这种语言就被称作脚本语言.使用它的这些用户被称为高级用户.
????脚本语言赖以生存的软件环境被称作是宿主环境(host environment).宿主环境可以是操作系统,服务器程序,应用程序,而开发这些宿主环境的程序语言(如开发操作系统一般使用 c 语言,开发 WebServer 一般使用 c 或 Java 语言,开发应用程序一般使用 C++/Java/c#语言)被称作系统开发语言,或用一个更贴切的说法是---宿主语言(Host Language).在软件产业发展初期,软件没有 GUI 接口,软件供应商提供一些使用该软件的 API(应用程序接口),而这些接口一般采用的编程语言是宿主语言.由于宿主语言是功能强大但也复杂的语言,因此使用该软件的用户也是专业性较强的用户.但随着硬件的快速发

展,软件业逐渐渗透到其他产业以及用户群体的不断扩大和 GUI 的出现,逼迫软件开发商必须提供一种比宿主语言功能较弱,但使用简洁,方便的语言来给一些非专业程序员用户使用.这就是脚本语言产生最根本的原因.目前世界上有数以千计的脚本语言形式.在操作系统领域,Linux 上有 bash, Windows 上有 WSH(Windows script host),而 web 上有 perl, jsp, php, asp, VBscript, JavaScript. 在应用程序领域, AutoCAD 上是 AutoLisp, MS Office 上有 VBA. 3ds MAx 上有 MAXScript. 各种各样的脚本语言极大地丰富了其宿主程序的功能,使宿主程序能满足不同客户的个性化需求.

????因此,目前大多数相当出名的软件都提供有脚本支持,我们国内的软件开发商是否也能考虑以这种形式来提供二次开发的接口呢?因为脚本语言的简单性能够降低二次开发的成本.如果软件是使用 Java 开发的,提供的二次开发语言也是 Java 或干脆直接提供源码进行二次开发,会极大地增加开发和维护的成本.而且当今软件行业的竞争日趋激烈,谁能快速满足不同客户的个性化需求,谁就能在竞争中占据有利位置,因此脚本语言的地位也日益突出.可以这样下一个定论,如果软件产品中不提供脚本支持,该软件产品必死无疑.

????脚本语言有多种分类方法,但按照使用方式来划分,脚本语言可被分为两种,一种是独立型(或称宿主型)(stand-alone)脚本语言,另一种被称为嵌入型(embedded)脚本语言(也称作嵌入式脚本语言).独立型脚本语言顾名思义,是指所构建的应用程序的主体程序全部或绝大部分是由脚本语言来编写的,即使用到了系统设计语言,也是非常少的(主要存在于库中),脚本语言与宿主语言的接口也只是由脚本语言调用宿主语言编写功能的单方向调用,极少反过来由宿主语言调用脚本语言功能.由于完全是用脚本语言来编写,因此就可以摆脱宿主语言的束缚,定义出符合需求的脚本类型,如脚本语言中所定义的对象的内存布局不必要与宿主语言中所定义的对象相同(比如 Python 对象和 C++对象在内存布局方面就是不同的,两者要进行通讯必须经过一定地转换).甚至可以忽略具体的类型,设计弱类型语言.独立型脚本语言的代表有 Python,Ruby,perl 等.嵌入型(也称嵌入式)脚本语言是指构建的应用程序的主体结构由宿主语言(C/C++/C#/Java)来编写(主要是为了性能和效率方面的考虑),但为了增加灵活性和二次开发性,在应用程序内部嵌入一种脚本语言来灵活地操控宿主语言编写的功能,并且宿主语言功能和脚本语言功能之间的双向调用是非常频繁的,而且也是对等的(即指双向调用的机会是相同的),由于是嵌入到宿主系统中,所以要受到宿主语言的一些约束,无论是数据类型还是内存布局,应尽量与宿主语言保持一致,如脚本语言的对象和宿主语言的对象最好能够在内存布局上保持一致,以便两种对象能非常直观快速地相互通讯,而无须进行费时而又冗长的相互转换.当然独立型与嵌入型脚本语言也并不是绝对分得非常清楚的,许多脚本语言既是独立型脚本语言又可作为嵌入型脚本语言使用.如 Python 和 Lua.

3.18 系统编程 Or 脚本编程?

系统编程第一要考虑的问题不是方便性和灵活性,而是安全性,特别是类型的安全性,

C++有运行期多态,这集中体现了它的面向对象,,是用虚表来实现的,这就是面向对

象的精粹,,

在一门 dukingtype 语言如 Ruby,Lua 这样的应用语言而非系统语言中，C++那样的虚表机制实现运行期多态是不需的，因为 duking type 这字眼已经是多态了,

基于对象是什么意思呢,,这个基于对象不是指"不能从中定义出对象的类本身这种 ADT",,,这里的基于对象,,就是指通过运态语言中通过"接口"实现的对象多态这种编程泛式,是动态语言中除了 OO 范式之外的另一种更为高级的编程泛式..

什么是基于原型呢,，就是说，事物是什么样的就是什么样的，如果它动起来像一只 Y 子，那么它就是一个 Y 子,，于是我们就提出一个原型,，比如 Y 子动的动作，（这就是二进制级的接口，对象行为集，而通常情况下，基于类的 OO 语言中都是单根继承，语义上的，实际上这种继承是不符合现实的，因为“类别”都是人为加上的概念,class 这个字眼与其说是一种 ADT,，事物原型，不妨说它更像是一种偏离原型的分类即 classification），实际上运行期的行为集接口实现的原型才是事物的本质,,，而编译期事先由单根定好的继承，这种早就分好类，并在运行期泛出来的对象反而是不符合事物原来本质的。。

类型在运行期可以被动态改变,,像 Y 子一样灵活地编程,那么在编译期类型就得不到被检测,,这极大地放宽了一门语言对于类型的检测机制,要知道如果是对于系统编程的话,类型机制是多么多么地重要,虽然动态类型语言编程灵活,但是其安全性没有一个保障 .. 这就是安全性和灵活的矛盾之处.

上面的基于对象,实际上也就是基于原型 ,,C++只有运行期的面向对象,却没有类似 RUBY 的基于对象开发泛式了吗?用 C++的模板技术可以办到‼

"C++模板实现的基于对象"是十分可贵的,第一,它提供了类 RUBY 的基于对象编程的机制(提供了类型多态,),,保证了编程的灵活性,第二,这种基于对象的多态是在编译期被栓测的,,一方面又保证了语言的类型安全性

一个是二进制运行时级的多态,一个语义级的多态

那么这样说,C++的运行期多型可以被抛弃了(我觉得作为系统语言的 C++只需要基于对象和模板就够了,虽然 C++可同时作为应用开发语言,但是它的运行期实现的 OO 实在是给系统问题引入了过多的人类 OO 思维,使系统问题变得多解,很多时候,我要求去掉 C++的 OO,当我只是希望用 C++开发系统相关的逻辑的时候),,因为基于对象的方式提供了编译期多态就够了..而且同时提供了类型在编译期多变了灵活性和安全性

第 4 章 语言之争

4.1 语言与应用与人(1)

在前面我们不断提到，很多语言机制,当你做到用抽象的眼光去看待它时,它才真正算是被你理解了,即如果你能从现实的应用开发这个高度和需求角度,突然想到你可能会需要用到 C++的模板(假设你以前未碰过它或只是听说过),那么你才会真正开始明白关于模板的一些东西而不是从学习 C++的语法书开始(要从人的角度到语言的角度再到人的角度)。比如你的应用逻辑中存在一个循环你需要一个循环,而 C 语言的语法刚好提供了循环语句,那么你会主动去翻 C 的这种流程控制进行学习它,但这是例外的情况,很多情况下,绝大部分的应用逻辑并不直接对应到一套语法机制,可供你直接采用,完成这个应用到语言的映射。因为语法机制实在有限,更多的逻辑被体现到了这种语言的库中,或其它第三方库中（语法产生库逻辑,语法产生各种变形逻辑来映射应用，语法与库的关系请参照其它章节）。

即应用才是最终的目的现实问题,我们永远要先行解呈清和解决它,**最难的编程问题是认识你要解决的问题的问题,在编程解决一个问题的整个过程中**。语法问题的解决处在编程的末端（这就是所谓的编码和实现问题）,应用开发更多地跟语义直接相关,而非语法。无论你看了哪本语言方面的语法书,负责的都会告诉你只学会了基础,的确,比起对应用的理解来说,以及如何用语法表现应用,学语言语法永远都是基础.明白语法只是第一步,你得明白语义,明白语义抽象到现实应用的那些部分。但是有人会说了,应用问题在一定意义上 — 比如提供了复用库,就转化为语言问题了,此时应用问题本身还是要搞清吗？要注意,语言会跟应用发生联系,但应用绝对不会跟语言发生联系,所以应用问题是独立语言和语言问题的要单独搞清的,这里我们谈到了语言与应用的关系,那么我们该如何理解 C++的一些常见语法机制呢并联系到应用呢？

在 C++的诸多语法机制中。类和模板都是代码结构,是面向人(解决复用和软工问题,处在人和语言之间)而不是面向机器(像数据结构处在计算机和现实问题之间,解决的是现实问题而不是人的问题),类提供了一个数据化“代码”的统一手段,使编程工作复用维持在这个高度上,但是正如上面所说,它没有解决实现问题。这也就是实现与抽象的区别所在。数据结构解决实现问题,代码结构解决抽象(到语言)问题。

4.2 语言与应用与人(2)

一门语言提供什么机制，，是由它能表达的应用逻辑来决定的，历来是语言适应并决定应用，什么语言开发什么样的逻辑（人，应用逻辑，语言及其机制永远是三个相互依存的矛盾，应用逻辑要求语言提供什么样的逻辑，而人开发应用逻辑也要求语言提供靠近人的机制比如 OO,，因此用该语言开发的逻辑都可以用这语言本身的机制来解释），比如 JAVA 适合做 WEB，LUA 适合做游戏(因为它的协程机制)，XML 适合做数据(因为它提出的元数据理论解决了浏览器文本交换的异构，并一度深入到数据库领域)，比如 Windows 是用 C 开发的，那么 C 的那些语言机制就可以用来解释 Windows 原理的一切，，

再举一个例子，我们将语言要面向开发的逻辑称为“问题本身”，“逻辑领域”，WEB 领域是最能体现这三者关系(称为软工)的领域，因为这里面存在关于这三者的诸多矛盾和他们的解决之法。再比如虚拟现实领域，，一个“虚拟世界”这个说法是用 OO 来表达呢，，还是用 LUA 的具体语言机制来表达呢还是用语言实现的某个库逻辑来表达呢(LUA 毕竟还是属于语言，可能在库级逻辑提供对游戏的支持，并不直接在语言机制里，，即它的语法支持里支持游戏逻辑，，因为它还是属于不同于 SQL 这样的高度面向领域的 DSL 语言的)，那么“虚拟世界”是用语言语法来表达呢，还是用库来表达呢，，是用语言的 OO 机制来表达呢，，，还是由语言的一种更好实现的机制来表达呢？？（诚然，OO 作为语言机制能够表达一切，比如“虚拟世界”，但那是代码结构侧面的，而且并不总是直接的好方法，有时对于一个特定应用逻辑，LUA 可能提供了一种更好更快速的语言机制，或库级逻辑，，反正要记住，OO 并非一切）这就是语言机制基于应用的要求。。

4.3 学编程之初，语言之争

其实就语言本身来说,并没有汇编,C,C++ 和 Java,Ruby 这几个语言之间哪个语言更强大一点的说法,大凡用其中一方能实现的功能,用一方都完全能够抽象得到,Java 所关注的 Web 编程领域,C++完全可以提供同样的功能实现，只有抽象完成,整个 Windows 系统都可以用 Java 来写,这就是说,在软件的抽象里,任何事情都可以以抽象叠成的方式来完成.但是显然地,在 Wintel 上装个 Jvm,再用 Java 实现个 Windows,这是个傻瓜行为(舍近求远而且有应用的瓶颈问题存在).

(C 的抽象在于抽象底层机器因素,提供程序员可见的因素和机制,,这就是抽象,,对于人的靠扰,C 的接口在语言级的细微度接口主要还是函数级的接口,,编译器级有 .out 文件,,,底层=算法加数据 ..接口提供了功能模块如何交互和复用的机制)

再比如 JAVA,,提出了很多语言机制,,也是为了能更好地远离本地发展抽象,,,它把什么都 OO 化了,这很利于对开发人员的抽象,但反而对一些迫切需要底层能力(比如指针和设备驱动开发)的应用没有利用之处,,,

就像在理解 Delphi 为什么能提供过程内过程一样，为什么函数语言可以避免不用设计模式,为什么函数范式的语言可提供 eval 函数(流程控制，等都是语言间通用的),为什么动态语言适用于 WEB 开发和 XML 结合，线程和异常．等等，这是因为语言也是抽象品，在它上面可以构建其它高级抽象，但是如果能了解一种语言抽象的得来过程，那么许多问题也就不问自知了(因为抽象跨度很小,中间很多的透明抽象现在可见了)．学习一门语言的好方法是学习它的抽象由来．．虽然这是个巨大的学习过程，涉及到很多其它非程序语言编制领域的抽象．．

当然,如果仅仅是想开发一个应用,那么不必了解这么多,有人不理解 STL 的设计原理和架构,照样都可以用 STL 技术快速开发,有人对 JAVA 不必作如上我们所涉及到的分析,是因为每个人都有自己维度上或科学或不科学的理解(有人甚至因为 JAVA 是编译器哈),正确的思想级的理解对开发不是一定必要的,所谓死读书也是有用的,不一定要理解到那个层次,能学会使用就够了,细节永远是重要的(有一句很出名的话:别无它,唯手熟耳),与开发应用最结合紧密,但是思想和学习方法也是重要的,我以前看过一本参考书,因为把所有的知识来源都理了一遍,抽象达成从不大的跨度说了一遍,比纯粹的一堆细节性的教科书更能让我明白很多东西,也即,细节不是一切,只是关键,思想仅仅只是重要的.

在 C 的时代,我们以为语言细节和系统知识才是难点,,在提供了 VM 式 OO 语言的时代,我们却发现设计更难.应用更难,如何开发被别人复用更难,如何复用别人的逻辑更难,一个时代有一个时代的问题

在 C 时代(也就是汇编时代),CPU 内置了操作码指令码,又提供了对内存和寄存器的控制,因此程序=数据+操作(换言之,计算机再怎么样,在其内部都是用操作码操作数据的机制).以 C 中对数据还进行了结构体这样的封装,后来数据经过了 CLASS 抽象和封装,一样都可以最终解释到这个机器过程.

给二套代码,一套过程 C 式,一套 OO 式,让读者发现他们之间的不同（一般 C 提供的接口都是函数，，一个库应尽量提供少的逻辑，接口并不总是虚的迂回接口，，比如 C++的抽象函数，，也有干实事的函数，，作为可复用的接口应是抽象的，，像上面提到的这些接口都是作为应用接口的，面向应用的，是不首先考虑作为复用接口的，如果是面向复用考虑而设计的接口，一般是干虚事的抽象函数）
第三个问题,什么是语言,我该选择什么语言,又回到可爱的语言之争了,谁也不想浪费精力花费时间却发现学了一门不太讨好的语言.但是千万要相信我,没有通用永远有前途的语言,永远没有最好，只有此时此刻刚刚好，没有通用，否则就是骗自己,很遗憾给了你一个几乎没有用的答案,我们在这里不宜讲解(请在本书进入一个阶段之后看 p178)

把 C++的语言机制分个类,,,有 1,指针,间接访问者实现 OO 也可,,2 ,OO,3,访问控制 COSNT,STATIC 等
深入分析一下全部的 C++语言机制,会发现他们都面向抽象,C++没有接口,却轻易能用指

针和纯虚类实现 JAVA 的接口
本书将从计算机这个编程环境,语言是什么

4.4 C 与 C++之争

C++之父在他的一本书中透露说，1/3 的人反对在 C++中加入 OO(OO 引进了编程向人的思维靠近的理念，是一种信仰和理念)，1/3 的人积极推荐他向 C++中加入这些功能，最近一次， Linux 之父跟一帮 C++的拥护者在一个论坛上吵得可以开交。

那么这两帮人争的究竟是什么呢，难道就是传说中 C，C++孰优孰劣这样的语言间存亡的问题吗？知识分子们当真会这么粗鲁？

首先， C++是必定离不开 C 的， C 对于 C++的作用应该说在这个话题的最前面。如果 C++一开始没有用 C 作为基础来进行 ++,那么在那个人们只会 “ 函数 +过程 ” 的年代,C++就只能是个早产而死的婴儿了。

实际上,函数和类都是代码结构,只不过函数和三种控制结构要简单,比类要简单得多,但其实他们都跟解决实际问题没有必然联系,伪码图,C 函数,C++类,都能实现数据结构,只有当问题要被编程实现时,我们左手有 C 而右手有 C++却反而陷入二难一样。而如果只有 C,大家都只会 C,谁都会明白用 “三种流程,一个函数 ” 的写法去写，这样的争端根本不会存在。

C 的程序员在用 C++表达数据结构与算法时，出现了二种情况： 要么写出来的东西虽然被套一个 Class 名字，实际上还是过程编程，要么虽然能类化相关概念做到真到的面向对象，但是感觉不如 C 的三种流程来得直观，感到 OO 很别扭。（参见 STL 的写法和对 STL 的使用，还有你能找到的 OO 实现的数据结构库）

业界对于数据结构，实际上人们已经习惯于用三种流程来表达数据结构，跟类比起来，三种流程加简单数据类型的语言机制已经足够。如果再强制他们使用 C++来表现(当然，我们这里指的是真正会用 00 来写程序的人)。那么他们实际上做了二件事： 对数据结构和算法能解决的问题做了解决（数据结构） 对代码进行了抽象（代码结构），使他们刚用 C++呈现的数据结构与算法的实现如何能被下一位程序员更好复用，我们知道 OO 是 C++带给程序员的抽象机制，对于了解这层抽象的人来说，他可以很灵活地写出质量好，逻辑清楚的类(比如 STL 的作者，虽然那是 Template class 类，泛类，但我们这里还是把它作为 OO 的类来看待)，但是如果回顾对 STL 的学习过程，大多数人会觉得诸如迭代器,Type Traits 榨汁器这样的东西是晦涩的东西。

所以，简单的语言机制都能解决的问题，为什么硬要套上一个 Class，还要造出诸如迭代器之类的东东？

面向对象程序员之间很容易写出质量参差不齐的程序，有的人写的程序用了大量迭代器之类的抽象概念，有的程序员写的程序全是 Class pig,Class cat 之类的实体概念。对于前面程序员写的程序，如果不能理解，基本上很难复用(即使有 class public member 在那里)。后面程序员写的程序，有时又会显得太繁琐(比如他会把一些不该抽象的东西抽象成为猫爪子，而实际上并不需要这层抽象)。

这就是过度抽象给程序员带来的麻烦。对于了解它会使用它的人来说是利器，对于不会使用它的人来说实际上是障碍。抽象的好处是某一个层面上的相对容易性（比如 OO 将复用维持在 public member,protected member 等接口上，当然 OO 远不只这些。），但是在另外一些层面会对不理解它的人造成更多层面上的迷茫(OO 的缺陷不来源于 OO 本身，而是来源于对 OO 的滥用)。

抽象给人带来的好处和坏处并存，对于多人合作的软工项目来说，一些语言的机制是禁忌，比如模板，他越抽象，就会造成越来越大的混乱，OO 虽然统一了代码形式为类，但并没有统一设计和抽象的方法(在同一个应用中，有人会写迭代器，有人会写阿猪阿猫)。所以 OO 并不是银弹，银弹是那些能统一人类思想，形成契约文化，经验的东西（比如我们写小说的那些套路），而不是简单的 class 这种面向复用的小技俩。 而在软工界，除了 OO，实际上还存在很多很多其它的过度抽象。

所以，总结一下，C 与 C++之争的关键的关键在于：并不是所有人都会 C++,会 C++的不一定会用类,用了类的不一定在面向对象,用了对象的一部分只会写 Class Pig 这样的东西,另一部分却开始写 Class iterate 了，这就是问题产生开来的关键。

C 和 C++的争论高潮就在这里，系统逻辑本来就是固定的，死的离散，而 C++的 OO 对他们作了极为灵活的封装变换，因此造成即使对系统的理解，，也变得在人们之间不够透明，，这造成的可复性反而会降低。

实际上，脱离平台逻辑的那些领域问题，基本上人人都会有一个解法，宜在这里采用 OO，OO 最应该出现在脚本语言中，并且面向对象和基于对象也是二种几乎性质的东西，虽然都是设计期，都是对运行期的设计，但其底层支持下的 RTTI 有性质上的差别，C++作为底层开发语言，它提供的“基于对象开发”还是比较接近 C“封据抽象”的风格的，但是它的面向对象，就有点脱离系统编程语言向 DSL 发展了，一般来说像 RUBY 这样的语言才提供 OO 和其它一系列语言机制。因为这些语言机制都是脱离系统编程的高层问题所需要的。而给底层开发提供一个 OO，反而让人觉得封装过头，，抽象得太象了，，太象人的思想了，反而不能够让人家了解系统编程语言作为描述底层的透明性要求。。

所以，对于有些人说的 OO 是种鸡肋和遗憾，到此如果你能得出自己的见解，那么你才算理解了这篇文章

4.5 观点一：我为什么选择 C 而不是 C++及其它语言

首先 C 是最不容易过时的语言。

因为它跟硬件和汇编相对接近最近，可以向后发展，而其它语言只能向后发展却不能向前发展。。即使是在冯氏结构过时之后，C 语言的地位也是最好的。嵌入式就说明了这个道理。。

其次 Core C 的语法机制简单。

C 语言能用简单的语法解决那些对于计算机实现要迫切解决的问题，而且这种做法已经成了惯例，比如算法，比如内外排序，当要排序的对象远远大于内存时，需要设计一种好的算法来解决其对于计算机的实现问题，，

而 Java 语言对应用的设计，解决的是站在人类逻辑角度的“设计算法”问题，比如如何进行架构设计才能更好复用以利于软工的开展，这二者所处的维度不同，决定了他们解决问题的方法根本不一样。。

当要设计机器的时候，Java 的那些设计思想根本派不上用处(是一种过于迂回的办法)。。所以 C 跟 Java 的理念是二码事，前者解决实现，后者更多地面向人解决复用。在下面一节会进一步谈到。

4.6 C,C++与 Java

C 语言重点不是为了来开发应用的，而是为了开发跟计算机本身(远离应用业务)那一层面的逻辑。C 不能满足大规模的开发,因此出现了引进了 OO 的 C 变体 C++,C++几乎是 C 的再造但更多的是增加，虽然它对 C 的兼容性是天生的，但是 C++又有很多陷阱,这些陷阱一部分来自对 OS 平台的逻辑,一部分来自 C++语言本身的复杂细节,(C++的库并没有把平台复杂性封装得让开发者可以不管它，而且语言细节带来的复杂性比如指针是用 C++这样的语言进行开发永远不可避免的)平台逻辑和语言本身细节带来的复杂性是阻碍开发者前进的二个拦路虎因此出现了 Java,JVM 这个统一平台使 Java 开发不需要了解任何我们真实的环境(系统知识，比如硬件和 OS).而且 Java 提供的库已经把 JVM 的进程资源,Socket 资源调用逻辑封装得稍微会点 OO 的人都可以拿来用了。

在这个新技术日新月益的今天，C 的开发早过时了，系统开发现在有点过时了，现在是高层逻辑开发时代和 Web 编程。现在的 Web 开发（这些架在逻辑上的逻辑（ TCP, HTTP, WWW, SOAP 一层一层而来））,,,提出了一系列新语言,新的框架,,设计哲学,,这些都是远离计算机底层的(甚至远离数据结构和算法这样的通用实现逻辑),,更像是一种逻辑的配置,,然而逻辑的配置比开发更难,,,,这个道理就像:这个时代 GUI 远远比 CUI 常见,,但反而 CUI 才是主流

现在再也不是早是不是要用 OO 还是不要用 OO 的问题了(这个问题已经早过时了)，因为 JAVA 和 DOTNET 出现了，它们的出现表明，，不但要用 OO，而且还要用虚拟平台上的一套 OO，（也即所有的应用应该从平地再加一层逻辑，加到虚拟平台上去，以前我们是从汇编到 C，这是对 CPU 编程转到对 OS 编程的改变，从 DOS 到 WINDOWS 的编程平台转变，，对网络编程和对本地编程平台的转变等等。因为我们 OO 编程时总是用一套 OO 库来进行我们的开发嘛，以前我们是直接用本地平台上的 OO 库，现在好了，我们被迫用虚拟的 OO 库，因为它建立在平台之上，同平台独立，因为这层抽象最大的作用就是 1.使我们利用 JAVA 或 DOTNET 开发出的程序是真正的 OO 程序 - 因为 JAVA 和 DOTNET 提供的虚拟编程 OO 库都是严格组织和科学划分的，专门为 OO 而设计，而且更重要：2.它们独立平台,这样编程的时候我们的眼光就会不再局限平台而专注跟平台无关的通用应用领域，问题域,如 J2EE）

这就是说,抽象远离了计算机底层,,在高层又变得异常灵活和难于掌握,,

其实 Java 这样的语言因为面向 Web,面向业务,,,它要学习的东西反而更多更难,,,学习 Java,如果不学 Web,这几乎是件可笑的事情,可是 Java 易学,Web 就不易学了

但是 C 语言要解决的问题永远是底层,系统永远是简单和形式的,大不了把数结与算法学精.学透,把系统本身学透,,C 的开发就到家了

但是 JAVA 后面的应用问题永远有新的学习需要出现,,因为 JAVA 后面的东西是异常灵活的高层应用而非固定的底层...面临的问题永远有新的解法,WEB 领域永远有新框架新语言的提出..

所以我说,,,其实 Java 比 C 难..!!!!!

这就跟 C 一个道理,,C 易学,,但 C 面向的系统问题涉及到大量数据和算法,又显得很难,,,如果学 C,却不是为了解决系统,这又是一件可笑的事情,,,永远不要以为学好一门语言就是全部目标了

似乎有人嫌 Java 过于简单，而且非 C++不能体现他们钻研到底层的精神，要知道语言级的形式才是羁绊，现实问题才是复杂和急需解决的，如果抱这种习惯久了,就出不来了

不要去干追逐技术的蠢事，，你只是用户，，只需学会一门工具开发，但是也不要走入另外一个极端,就像我崇沿 Ubuntu,我并不会动则就把 Windows 和 Ubuntu 作一个彻底的分离，并一有时间就抵制 Win 并使用 Ubuntu 一样

学好 C 语言在于将 C 语言包括指针在内的所有语言机制习惯用法，跟数据结合和算法的结合,,C 用于实际问题才是重点要学习的

懂了 C，却没懂系统底层一样没有用，不懂数据与算法也没有用，因为 C 就是面向并开发这些的

其实,Java 和 C 都可以发展很多后来的抽象,,Java 和 C 容易学习,是因为它们入口处小，(Java 和 C 有有限的语言机制和学习成本)学习和控制就很容易了,,但抽象应该长足发展(Web 问题和系统问题是一个不断可以深入的话题,需要新的语言机制的支持),，Java 和 C 解决的问题永远没有一个头,,永远都可以是新的话题..因为应用无形而语言有形

应该相信一些简单的道理，因为它们是道，道统一所有理

无论是什么语言,一门语言写出的程序总是程序=抽象+接口，

但是,JAVA 不仅是一门语言,它是一个平台,这指的是它的 JVM,所以除了 Java,它上面还可以发展出其它的语言。比如 JPython(C++却不是一个可移殖的语言,如果要移殖,它的库也不允许,关于 C++的库有太多涉及到本地 OS,C++与 OS 藕断丝连，这在一方面导致它并不适合ＷＥＢ开发，C++与ＯＳ挂钩，JAVA 代码与ＪＶＭ持钩，而ＪＶＭ本身是可移殖的，因此这层抽象也就说明 JAVA 代码是可移殖的),关于 JAVA 的 WEB 应用是发展在这个平台上的,JAVA 变为 WEB 开发语言不仅仅是因为 JAVA 代码可以一次运行,到处执行,历来不是语言成就应用,而是应用成就语言,就像 ROR 成就 RUBY 一样,人们发现 J2EE 框架库的出现适合 WEB 开发于是就导致了 JAVA 作为 WEB 语言的流行,而且关于 JAVA 的库和先进的框架越来越适应ＷＥＢ开发，由于这个历史，天时地利原因，JAVA 最终成为ＷＥＢ开发的主流.
后来ＸＭＬ出现了，ＸＭＬ与ＷＥＢ的结合可谓是无孔不入，ＸＭＬ甚至渗入到桌面开发的ＯＦＦＩＣE中（，当然，这是ＸＭＬ不作为文档意义上另一个维度上的抽象，其实软件开发到最后，桌面与ＩＮＴＥＲＮＥＴ或者ＷＥＢ的抽象间隔会越来越小），WSDL 被提交给 W3C,JAX,AJAX 等等,这些都说明现在的 WEB 开发和布薯都趋向于与ＸＭＬ紧密相联
XML+Ｊ２ＥＥ终究太复杂，在证明ＲＯＲ这种框架比Ｊ２ＥＥ优秀之后(开发周期短)，ＲＵＢＹ语言作为开发ＷＥＢ的最佳选手身份就被成全了，RUBY 是动态语言,它提供的处理 HTML 的能力使它优于 JSP 在网页中嵌入 TAG 的方式,然而这些都不是最最重要的方面，PHP 也有这个很强大的处理文本的能力

C 跟 JAVA 是二种完全不同理念的语言,因为它们的运行环境不相同,产生的目标码根本不同,前者是为机器加 OS 这个本地产生目标码,而 JAVA 为 JVM 产生目标码,由于这二个平台的不同导致的理念是根本的差别,JAVA 不能称为系统编程语言,因为这个系统指代 OS,C 是编译成本地目标码的,而 JAVA 码只能称为 JVM 的系统编程语言.VB5 之后有编译为本地码的特征,因此是半系统编程语言.

C 跟 JAVA 是二重天,因此对它们的开发理念完全不同,一个是面向本地,一个是面向人,可以不管 OS 平台的任何差别(这就是说不用学习程序运行环境给语言带来的机制问题和复杂细节,不用学习关于 JVM 的任何知识来进行 JAVA 编程,人们只需关注那些他们要完成的逻辑,而平台逻辑可以忽视不顾,因为 JVM 被移殖到任何地方时,人们将面对毫无差别的 JVM 和 JAVA 代码,代码不会因为平台不同而产生新的要解决的问题,,这是指移殖,,而且,要写 JAVA 代码,也不必了解任何 JVM 的知识,,因为根本没有必要为 JVM 编程,,JVM 虽然能给你进程,,给你 SOCKET 资源,但是在库中已经被妥善地封装了,人们对它固定了一个认识,,不必从 JVM 中再对它们经过原始理解,再引申为 JAVA 所用,因为关于这些资源的接口足够简单了,减少了人们对系统的理解,一谈到虚拟机的语言,就要明白这跟 C,跟 C++这样的编译语言是二重天).人们说 JAVA 并非好的教学语言,因为 JVM 毕竟不是我们真实的系统,,它是一种架空的虚拟的完美的人工运行环境,而 OS 加硬件架构这样的本地才是我们要认识的本地,

Java 绝对是很好的软工语言,但 c 不是
虽然都是通用语言,但在“语言级”能直接提供的最小复用单元上,Java 可以提供类,接口这些的抽象,但函数只能是一种初步的调用接口,不是数据也不是类型,甚至不能直接表达实体,既使能也只能靠拙劣的抽象手段得到(就跟西加加的元编程一样)
数据化,唯有数据化,才是计算机和人共同理解并加以利用的东西,才是软工要首先统一的目标所在
typedef int myinttype
typedef myinttype (*myfuntype)()
struct,pointtype,简单类型
你指望 c 的这些“数据类型”成为软工中大量的开发人员理解并大量使用的东西吗,并以此为基础抽象大量复杂的领域逻辑?那恐怕是专家作的事吧,怪不得有本书叫《C 专家编程》呢，而 Java 是平民语言‼仅仅懂得设计就可以拿来配置出逻辑的语言。

面向对象最大的功劳不是它的三重机制,而是它桥接了现实和语言之间的映射,即使是大众也能理解并独自开发内含大量逻辑的软件，所以,如果大学是培养软件人才的地方,Java 为什么没有比 c 很多的理由成为教学语言呢

c 有全部的离散而 RUBY 相对来说就少点,没有指针这个离散因此 C 产生 RUBY，而 RUBY 只能 BindC,RUBY 不能写我们现在能看到的 WINDOWS(因为它关于 C 的底层用到指针),而 C 可以出 RUBY。

什么样的应用用什么样的语言,需要达到什么样的逻辑就要选择好语言,因为 C 几乎能控制计算机内部所有离散(而应用就是计算机离散的向后发展，C 语言的强大在于它是抽象的源端，可以向后发展。并可兼顾前面的应用开发，而 Java 早就抽象到后面去了),因此 OO 反而没有 C 的直接底层强大(这相对计算机能完成的事来说的,在开发上,OO 自然会快点好点)

所以，业界一般使用二种语言，一种作为实现语言即系统编程语言，一种作为设计语言，即脚本语言。

1,OS 用 LINUX 内核+Ruby shell,比如 Ruby,底层开发一律用 C,比如驱动程序,游戏底层硬件渲染,RUBY 可方便调用 C,而且直接用于硬件编程时十分科学,比如 embed c,C 的运行也十分快
有人会问了,当 RUBY BIND C 时,那难道 C 的模块不也成非本地代码了?其实不然,当 C 模块被 BIND 时,它只提供一个接口给 RUBY 使用(只是在 RUBY 这边产生了一个使用逻辑),真正的执行体(被调用者)还在 C 模块中,还是 native code.
2,RUBY 与 C 混然天成,当作高级非系统编程时,应把它变为一种 SHELL 加通用编程语言..让 RUBY 也同时成立软工时代标准语言

C 面向底层,很好地解释了学习一切 C 泛化出来的其它语言的底层知识,比如字符串,比如 IO,比如进程,线程,对学习是极为有利的,而 RUBY

C++终于还是属于一种复杂的语言,它的机器因素太多了,多得初学 C++的人迈不开脚,,所以要去的地方还很远,,,,用一门语言来描述要解决的问题,C++只完成了最最初级的抽象,虽然它提供了 OO,但是它导致的因为 C++的 OO 机制也很让人头痛..

4.7 通用语言与 DSL

像 C++，C 这样的语言都是被设计为通用的，要通用，因此往往基于某种靠近计算机底层的离散形式,而 DSL 实现特定领域事情的语言,(相对 C++,C 来说)不强大，不深入底层，不能控制计算机干任何通用事情,因此往往基于高层模型,,

因此,C++,C 这样的语言必须要涉及到汇编原理里面的东东,而 DSL 可以以任何高层的形式被体现,比如不需要编译的 UML 图都是,POWERPOINT 代码都是 DSL,根本不需要编译器这样的图灵完备装备
这就是脚本语言比不上编译语言这样的语言对计算机编程方面的功能强大.因为脚本语言的虚拟机往往是高级机器,根本不像我们的硬件机器那么底级,图灵模型对应我们的硬

件机器和架构,而虚拟机往往跟硬件架构差别过大,因此脚本语言和系统语言是为二个不同的机器设计他们干的事,,,,而一般虚拟机作了高级逻辑,比如 GC 等,而 X86 不可能在硬件架构级就用了 GC,如果 CPU 芯片可以用硬件加速的方法直接支持语言的垃圾回收机制就好了,这要求语言跟 CPU 一一对应,,而且 OS 也提供了大局方面的 GC(并不仅仅针对某个语言,比如 WINDOWS 的操作系统级的资源释放)

C 跟 C++到底有什么区别呢,我觉得第一个加号是一种理念上的叠加,第二个加才是语言要素上的改变,C 跟计算机离散和底层接近,解决的问题是如何实现,专注于计算机对问题的实现,因为 C 语言就是机器的观点

C++跟人接近,解决的是如何更好地复用,关注于问题本身,脱离了实现逻辑如平台逻辑,人们如何更好地写代码服务工业化,OO 就是人类的观点,C 是实现域扩展,C++则完成了一个从实现域到问题域的一个维度变换而已,

C 和 C++解决问题时,,站在二个根本不同的维度而已.

4.8 .NET 与 JVM

.NET 语言的公共语言运行时就相当于 JVM,它们为一种语言或多种语言的代码提供运行的平台(比如运行时为它们分配内存,,普遍认为在.NET 的运行库支持下可以运行多种语言的代码,在 JVM 下可以运行 JAVA 原生代码

但是要知道,原生不原生是相对的概念,如果能在 JVM 上实现一个 Ruby 的解释器,那么 Ruby 代码也就是原生的 Java 代码,

OS 跟虚拟机的关系,比如用 C(更严格地说是 C 的一个编译器实现比如 MSVC,BC,GNUC)写出来的代码就是直接在操作系统上运行的(由一个叫运行时的抽象层向 OS 请求内存时空资源比如 CLS 的托管内存说法),,这相对 OS 来说,C 代码就是原生代码,但是当为一种语言发明一种虚拟机运行平台时,这个抽象就大了,我们不再称这个抽象跨度为原生,而是过度了的原生,也就是说,不是原生,而是相对虚拟机的原生,比如 JAVA 代码之于 JVM 的关系

实际上编写虚拟机是编写 OS 的技术之一(在一台裸机上写出一个虚拟机才能调试代码和执行代码),并且直接在一个业已存在的 OS 上抽象出一个虚拟机实现也是可以的,,,因为这样可以独立很多平台执行这种代码,,这样做的目的(在业已存在一个 OS 的情况下)就倾向于"为了语言而创建一个运行平台"也即一定程序上"JVM 是为了 JAVA 而出现的",而本来不需要一个 JVM 就可以直接在 OS 上写 JAVA 语法的代码的

?那么 JVM 与 JAVA 解释器的关系又是什么呢?一门语言的最高级公民(first class)往往存在于栈内,比如函数啊,OO 体啊,但是 JVM 又不是 JAVA 解释器,不属于运行时抽象也不属于 OS 抽象,而是编译原理抽象,学习的过程中,我们必须格定这种"抽象所属",才能

4.9 你为什么需要 Ruby

困此我们说 C,和 C++是一派,因为它们没有一个虚拟执行环境，面向真正的计算机环境，C 可以做 C++做的一切事情,但是 C 明显不能大规模开发需求,当面向本地编程时,当逻辑过大时，一定需要 C++这样的语言作辅助.C 和 C++被禁固在本地上，C++是本地编程的极致了，提倡对本地编程只需要 C 就够了而排除 C++的这种论调是极端的，而 C++也并不是做得完美，除非对 C++本身进行改造，增加如ＲＵＢＹ那样的新语言的新机制，让 C++成为容纳一切范型的多范型，否则它并不能直接实现很多ＲＵＢＹ能干的事情(RUBY 构建在ＲＵＢＹ虚拟机上的ＯＯ机制和其它语言机制多了去了，比C++简单但比C++的那些范式强大，比如 C++的模板，虽然可以让 C++带来ＧＰ，但是ＲＵＢＹ的ＧＰ要科学和有效得多，还比如 JAVA 用反映和代理技术去模拟ＲＵＢＹ的动态语言性质), JAVA 却根本不擅长操作本地,因为这是它的设计目的导致的(JAVA 就是为了独立平台,使得开发时可以不管复杂的平台逻辑,而且即使是对 JVM 的平台逻辑,JAVA 的那些库也作了很好的封装)

那么你到底需不需要一个ＶＭ式的语言呢，首先如果你是作本地开发，虽然ＶＭ式的语言可以通过ＪＮＤＩ这样的本地接口和ＢＩＮ技术来进行访问，但终归是一种绕之又绕的方法，而且ＢＩＮＤ和改造技术也不是万能，有时你并不需要全部的原语言的逻辑，此时你需要定制ＢＩＮＤ来调用你需要的部分逻辑，这是第一，第二，带了ＶＭ作执行环境的语言过于宠大，当一旦有这种语言写成的源程序当被执行时，其执行效立造成的影响不容小观（比如游戏开发，在一些实时碰撞和消息互发时要求不要有迟缓，ＷＥＢ除外，ＷＥＢ的平劲不在ＩＯ）

第三点，要搞清ＲＵＢＹ等这些语言的目标，ＲＵＢＹ被设计成通用编程语言，但是更接近一种ＤＳＬ，它就比较适合搞ＷＥＢ，

那么你到底需不需要一个动态语言呢？?动态语言就是在写代码的运行前编译期不需要为变量指定类型，因此类型宽松，写作起来不需要考虑类型之间的静态期转换，只需像对待普通简单类型变量一样对待抽象 CＬＡＳＳ类型（变量），源程序可读性也很强（因为没有很多关于类型约束的逻辑和信息），但就是因为类型信息在运行期，因此在编写时ＩＤＥ不能获取到正确的类型信息，这给编写造成了一定的麻烦.

那么到底需不需要一个解释语言呢？解释语言就是边翻译边运行，程序可以被实时调用，这几乎是脚本语言的一个通用特征（一般脚本的意思就是通过一个ＳＨＥＬＬ调用，ＳＨＥＬＬ就是解释器，脚本这个说法相对于系统来说的，实现比较简单的ＤＳＬ方面的东西，系统编程语言是通用的，那么脚本往往实现某个领域的逻辑，比如文本处理，但ＲＵＢＹ和 JAVA 走的都是通用脚本编程语言的路子),类逻辑可以先存在于持久化状态，再被实时加载，每一句新写的源程序都可以被即时执行.

那么到底需不需要一个类 RUBY 语法机制的语言呢，如果没用到协同线程，那么 Ruby 中的 Coroutine 你就用不到．如果这些都刚好对应上你需要的功能，而且上面提到的其它方面都基本满足，那么ＲＵＢＹ还是比其它语言要适合你的.

因为脚本就是要实现一种ＤＳＬ的东西，所以ＲＵＢＹ即使很强大，Ｂlizaard 还是愿意用ＬＵＡ这样的东西，因为ＬＵＡ的一些比如协同线程很适合游戏开发，而且ＬＵＡ不必通用，而且游戏就是要轻量的ＶＭ，脚本语言的本质决定了它不好被发展成为一种通用语言.

没有一种通用最好的系统编程语言+通用的最好的脚本语言，不要去寻找它，因为历来

都是应用决定语言，而不是语言成就应用. .

4.10 我为什么学习 Python

在本书第一部分第五章《真正的函数指针》中。最后一句“这样的语言留之何用？其实我的本意在于表达：“C 做做系统还是可以的，然而处理高层编程，比如 web 就不太好了”

因为 C 是文言文，python 是白话文，而我们是人

事情总是往高级的方向和抽象的方向上发展 (以前是网络即计算机 ,,现在是程序即动态 web 页面 ,network is pc,application is dyamic server pages,即所谓的云计算的一部分), 因为我们是人，不能和机器处理细节的能力相提并论。就像文言文一样，它灵活和绕之又绕的用法就像极了孔乙己说的“茴香豆的茴有三种写法”，而白话文只提倡一种通用的用法，即 python 所提倡的“任何一个事情最好只有一种解法”c 太拘泥于语言细节和平台细节，而 python 极力专注于高级的应用（在后面的介绍中你可以不断看到，在 python 语言中有很多高级语法机制，甚至设计模式），而现在的 web 开发，需要语言提供直接而显式的白话表达方式

为什么用 python 呢？另外一个理由是移殖问题，我们知道在一个架构中（比如 google 手机平台），应用往往搭建于软件的机器上 (visual machine)，而往往非 real machine(cpu 架构 + os) 上，才会具有最大的移殖性 (因为在 vm 上实现，只须要一层，而直接实现在 real machine 上，有二层移殖障碍需要考虑，而 real machine 是必须存在的不能发展出一个所谓的“python 机“来，因为 OS 是必须需要的，一个机器 OS 不仅仅是像 vm 这样的东西就可以的，所以解决开发导致的移殖问题，就发展出一个 VM 出来)，就拿 mysql 来说，你当然可以移殖 C 的 mysql 到一个手机平台上，然而你也可以移殖 python 过去，再移殖 pysqldb 过去。显然后者具有更强的移殖能力。其实这也就是 java 和 jvm 产生的初衷。

生命苦短，为了实务起见，我用 python!!!

4.11 类 VB，DELPHI 类 RAD 语言分析

vb 在 Windows 平台上是一个流行的开发环境 ,在排名上很靠前 ,VB 所体现的理念就是比尔向开发者靠近的态度 ,一个 OS 的发展离不开 DEVELOPER 的支持 ,比尔算是深刻地做到了这点

并非没有指针的语言功能就一定很弱 ,并非没有 OO 的语言功能就一定弱 ,运用 VB, 你照样可以用 VB 开发游戏 ,系统软件等 ,解释器并非虚拟机 ,VB 带了一个解释器和运行时库 ,

但是没有带一个软件机器性质的运行时虚拟机,,因此效立也不是很差.
我们关注的是 VB 的快速开发理念,它的最大特点就是"封装的可视性"和"复用接口的简单性",,,可视性是指逻辑单元总是被发布和开发成一个可视的控件,简单性是指每个控制都有有限和简单的几条属性和方法,,语言做到这个程度已经十分简单了,不需要我们掌握学习 C++要学习到的一系列其它细节复杂性,VB 把这个年代的程序员更多地做的是一种直接使用别人的库的事情这个理念算是做到家了,只要在上游统一了简单的形式,那么在下游就可以用这种简单性开发一切逻辑,,我们知道,复用就是逻辑组合,逻辑总有接口,对象的属性和方法都是接口(但一般把数据属性封装起来,因为透露数据的接口是危险的接口),构建应用就是组合接口兼容的逻辑单元，但是这个过程明显比不得现实生活中的电脑组装,这种事情.所幸 VB 也有其另外一种过程式的范式,并不要求用户一定要用这种方式开发.
因为应用是复杂的,软件开发永远不可能做到比 VB 还简单(永远不可能像现实生活中的找零件配电脑这种过程),,,这是因为应用永远是复杂的,而且有它自己的专门性,,,除非别人发布的库就是为你的应用定制的,,又或者你的逻辑不简单,需要复杂的逻辑(而别人没有提供类似可拿来用的逻辑,或者即使有相似的,但是为了适应你的应用而进行改造,那么复用工作就是纯粹的开发工作了(相对来说就不是复用了,而是开发了)..如果改造工作代价过大,或者对逻辑和接口的调整根本不能进行，那么ＶＢ的这种编程理念也是没用意义的.
这就是为什么 VB 永远不能跟 C,C++这样的语言功能强大一样,,毕境,,有限的形式 1 可视 2 接口简单并不能直接产生很多复杂可产生的计算机逻辑(就像其它语言能提供的众多范式一样),,,这个道理就像 WINDOWS GUI 跟 UNIX CUI 之于开发谁更复杂谁更科学谁更简单一样,WINDOWS 对 GUI 做多了文章,,使得在某些人眼中利用 GUI IDE 开发根本没有 UNIX 下 SHELL 开发简单和自然,甚至于一个 VI 就可以当成整个 IDE..有时候,提供的形式越少,少得简单得最最初级的编程者都知道用控件来搭软件,那么这 2 种向"可视化"发展的简单形式反而不能描述更更复杂的逻辑,只能将逻辑做到控件层

所以 VB 这样的东西还是局限性很强的,,不是一门真正的系统编程语言,,像是一种介于系统语言和 WEB 语言这样的 DSL 之间的语言,,,往往用于 RAD,,,通用性不强
因此 VB 从来都不是标准,,只是像 E 语言,DELPHI,一样,只能往 RAD 的方向发展
正像我所说的,,,如果你想为你的应用计划一个科学的设计
最好是理解整个 PC 逻辑,,硬件的,软件的,,语言的,,库的,,应用的,,软工的
要注意 VB 并非通用 OS 编程语言 ...因此它的局限性很大
历来都是应用成就语言,,,有了 RAD 所有就有了 VB
离开了快速开发的需求,没有人为 VB 提供复用控件,VB 就没有太多用了..因为下不能入底层,上不能跟 VB.NET 一样进入高层的 WEB 开发,是鸡肋了.

第 5 章 语言最小内核(C)

5.1 C 与 C++是二种不同的语言

C 语言的最初的版本是起源于 B 语言的，后来经过发展，又经过了多次标准化（第一次民间标准化是 C 语言的二个创始者 K&R 写的一本书，C 是最初作为 UNIX 专用系统编程语言出现的），分裂成了二支，在 C89 标准上面加上 OO,加上 Template,就形成了 C++(C++是 C89 的超集，当你手中拿的 C++教材没讲基于过象面向对象和模板并且你需要的不仅是一本 C 的教材时，你可以现在就扔了它)，第二支继续独立发展,发展出了 C99，目前最新的 C 标准是 C99,支持并实现了它的编译器市面上比比皆是。

Unix 被产生时，是用汇编写的，汇编有移殖问题（汇编是硬架构上的 CPU 专用语言），因此 Unix 的作者又创始了 C 语言，并改写 Unix,所以历史上 Unix 上是产生 C 之前的，C 的移殖性超好，这个作者又写了一本书，形成了 C 语言的第一次标准化。几乎是 Unix 产生之后，类 Unix 系统不断出现,,为了规范这些类 Unix 的兼容性，提出了一个 Posix,对系统调用例程接口进行了规范，因此产生了 Posix api 库.

最初的 C++语言的实现(就是 C++作者写的一个编译器)实际上本质上是一个 C 编译器，他在 C 编译器的前端额外加了一个过程，把 C++代码映射为 C 语言作为目标语言。再把此目标语言映射为机器语言。。

关于 C89 与 C99 方面的差异有一些是根本性的，因为实现的原理根本不一样（是编译器方面的差别，而不是库级逻辑上的差别），比如 C++用 Template 技术实现的数组和 C99 版本的数组。

所以，C++除去 C89 的那部分才是 C++的主体(所以说学习 C++是学习 C 语言和 C++语言这二个过程，我们知道++是 C 家族语言中特有的运算符，C++这个表达式表示，第一次的运算过程只首先取 C 的值，第二次及以后的运算过程才在保留原值的基础上开逐次始加 1，并形成 C++，当然，C++是可以没有 C 而独立的，可以独立被学习的，它也有自己的 stdc++lib,比如 string 等，IO 等，并且还有 stl)，C++为了成为多范型语言，先是杂合了 C 的很多特性，提供了指针，位，函数，流程这些基于过程的开发支持，为了成为基于对象和面向对象语言，又实现了一个运行期 OO – 其中有运行期多型机制（这是 C++之父" 发明" 的），再后来是在没有任何原 C 的编译技术的支持的情况下发展出了一个 Template（也是 C++之父所在的研究小组" 发明" 的），是 C++首创的独立编译技术（并由此" 发现" 了 C++的编译期多态和实现了 Loki，Boost MPL 库等），STL 就是用了 C++的 Template 而出来的东西，STL 是面向运行期的，而 Boost MPL 库是面向编译期的。

C99 的数组跟 STL 的数组就是大不相同了，，一个是 C 上面的，一个是 C++的模板技术上面的，有着不同的实现原理，编译支持基础．．

在 C 中，数组就是原生 Built-in 的，，数组就是连续的同类型数据在内存中的分布,,,注意,1,连续空间 2,同类型数据,,如果不是连续的空间就不是数组,,比如用指针 link 起来的就是链表,而不是"数组表",,注意"表这个说法",,数组是一种表,,因为它用索引下标索引数据对象,是一种 key:value 对,而链表也是 2 同类型数据,,

在 STL 中，模拟数组比如 Vector,List 是用模板来实现的．．这二者之间明显存在差别，静态数组一般直接用 C 的数组，，但是正因为 C 的原生数组是没有边界检查的，编译器根本不保证这个栓查全部交给程序员来处理，，而 STL 版本的模拟数组就会非常严格．．

这就是二者之间的差别

任何语言都有算法+数据结构之说的设计，但 C 更为突出，因为 C 只有这种设计范式，相对其它语言众多的语言机制来说（除了基础流程，类型系统那些东西不说），这更像是 C 的设计全部，为 C 而生的，在用语言机机制表达应用方面，C++有“类”,有设计模式（但设计模式用 C 来体现，似乎没有人作过尝试），有模板，有编译期多态，有范型，当然也有“算法加数据结构”，但 C 仿佛只有这种““算法加数据结构””,,，是设计的非常原始阶段和手段。。

在 C 中提供了很多支持数据结构和算法的元素，比如数组，链表，指针，struct,typedef,相比其它语言，c 还提供了比其它语言的更好的支持数据结构的语言机制。。特别是，C 中的一些语言机制，如位移(其实大都机器提供的位指令并不直接操作位，而是对整个字节进行位操作，进而间接控制所需要的位，我们常常通过对原字节进行位移的方法得出 bit mask，再将这个 bit mask 跟原字节进行与，或，异或，最后得出需要的效果，比如提取 1 位，消去 0 位，颠倒特定位)，跟搜索算法中特定一些算法直接相关，，相比之下，其它高级语言的数据结构都是基于高级语法结构的.甚至于像 lisp 这样的语言就将 adt 内置为其一级类型。

5.2 C 的类型系统与表达式

任何语言都有一个“类型系统”， BCPL 作为 C 语言的始祖(就连 32 位汇编语言都引进了变量和类型,类型是一门语言的基础，用来表示为程序所用的内存的抽象，即多少内存，可以在这个类型上执行什么操作)，它是没有数据类型的，实际上一切都是抽象，C 语言中的基本类型正是抽象了“以何种方式使用内存(更确切地说是使用内存的位,因为这是汇编语言和机器逻辑最终要考虑的问题，在高级语言中得抽象一下不能直接用位组)”，使用内存的方式就成了基本类型，这是一种抽象。。将基本类型发展为 Struct 聚合

类型又是一种抽象，，将数据类型封装为 ADT(同时封装了可施加其上的操作)就更是一种抽象了。。基本类型中为什么会有数值，字符就知道了，这完全是一种趋向现实的抽象，这是每一种语言的每一个程序，每一个现实问题最能广泛体现到的抽象，当然，基本类型完全也可以是函数，如 Lisp 语言，，只要是一门语言直接支持的类型，那么它必定在内存中有一个位存储模式，汇编器规定了什么样的操作可以施加在这个位组合上。，

比如 C++直接支持类(UDT)为它的 First Class，，Lisp 这样的语言支持直接传递函数指针(因为函数是它的类型)，，，就 C 来讲，，它支持 primitive types，即普通的 Char,Char *,Int,Int *,Float,Float *,Void 类型。。

到底什么是类型呢，C++ template 的 type 跟 class 有什么区别呢。。

C 语言中有一种 void 类型，，无类型指针。。我们知道变量是有三个要素的，1 变量的名字，2，变量的地址，3，变量的类型，其中 3 是最重要的，它影响了 2，，这也就是说，对于一种变量来说，一谈到这种变量，它的大小就清淅了，，变量的最重要的特性是它类型影响下的大小。。。这是变量的本质

所以，对于变量的指针这种类型来说，，在使用它之前，一定要知道这指针是指向什么变量，，这样编译器才知道从这个地址，以多少大小的范围去操作它，否则 void 类型的指针是不能被解引用的，只能供调用参数用。。

使用 void 来作为函数参数时，主要出于这种目的，比如我们想让一个函数处理多型数据，就可传递这种数据的指针，因为是多型，我们不直接定义 int *,float *,这样多的类型，而代于 void 作参，这样处理时带来 void 解引用后的结果(当然，得 cast 成 int*或 float*这样实际的指针变量才能使用得到它指向的数据)。

而 cast 是一种什么样的过程呢，C 语言的隐式转换是编译器的动作，手动转换是程序员的事，但是，无论是这二转换的那一种，都不改变原指针值，它只是在内部作了一份拷贝(这个道理就像传指针也是传值，不过传的是指针值，因此另外一种意义上来说就相当于传地址，，它跟直接传值一样，也是作一份拷贝)，你可以将这个拷贝值看成为 C++新增的指针语法：变量别名。。

5.3 C 的数组，指针与字符串

(我这里讲的都是 C 标准)

C 只直接支持简单的数据类型，只有简单的 Int,Float,Char,Void,这样的数据类型可以直接定义使用(并可作为函数参数传送,但数组就不能直接被传送,需要转成它的指针形式 – 一个 32 位长整型数被传送,,因为它不是 First Class),,其它的高级数据表示比如字

符串,数组,都是在简单数据类型之上模拟得到的,,,是库级的逻辑,,编译器并不直接支持..(C 编译器没有提供对字符串 String 的 Built-in 支持,只有 Char 和 Char *，C 把它作为库级逻辑的 String 来抽象)

C99 在库级新增了_Image，_Complex，_String 这些数据抽象(相对 C++来说)．．

所以 C 的指针类型几乎跟数组，字串这样的逻辑同胞共母，字串本质是 Char * 数组，，数组是内存相续的同型数据，用指针变量来表达数组和字串几乎浑然天成(数组名就是指向第一个数组元素的指针，类型为:"(基类型)指针变量名"这样的形式，，而且字串直接被定义成 char[]的形式，，数组大小一定要程序员方面保证容纳字串大小+一个/号的大小,字符串函数大量涉及到指针比如 strcat，C 有一个标准的字串实现，cstring.h，C++有一个 STL 版的字串，C++还有一个原生版的 string.h，特定的编译器也有特定的字串实现)，，，C 就这样把所有的东西统一于简单的"简单类型"．．

当然 C 也提供了 union,struct 这样的结构体来表达比简单的类型来抽象一级的类型，其实 C 也可以表达ＯＯ（不过它并不承诺一定要实现运行期多态，继承等ＯＯ语言的标准技术，，有一本＜＜C 支持面向对象技术＞＞的书讲解到了相关的技术）

5.4 C 的输入与输出流

C++版本有个 Iostream(Cin，Cout 就在里面，实际上 C++的 IO 库很大，是一个复杂的基于对象的模板系统),C 也有它的 Stdio.h(有 InputChar()和 Printf()等这样的函数),这是 C 语言的标准（库）独立于所有的硬件和软件环境，是语言本身的因素，，,一门语言的 IO 机制是很重要的，比如，JAVA 的 I/O 封装得很完善，，它细分成了多个ＯＯ表达的"流"，极大地方便了程序员。

I/O 绝非仅仅输入输出这四个字这么简单,首先这是一种语言机制(就跟语言提供异常处理,这样的语言机制的地位一样重要),是关于这种语言对向它输入和由它输出的所有抽象的全部集合,它涉及到一门语言如何处理与系统的交互，如果将用户输入转成该语言写成的程序所用，，如何看待 OS"文件"的概念．并发展出一系统概念,如 input steam,file steam,等等

在 PC 的架构中，语言是高于 OS 内核的(一般在一个系统的架构中，开发层是高于内核层的)，Windows 是用 C 写的，C 有它的 I/O，是先于 Windows 的，一门语言并没有具体的数据类型，，比如 unsigh,sign 这样的区别是来源，起决定作用的还是硬件，，C 语言只是对它们提供了一些名称指代,,比如 int 在有些机子上是 32 位在有些机子上是 16 位，这对考虑一种程序的可移殖性是十分重要的。

在 C 的眼光中,一些输入输出都是某种流,语言接受用户输入,或者程序进行输出,先以内存作为根据地进行缓种,，缓冲分为二种，一种是缓冲文件系统(这是 C 标准的文件

系统,,即不是一次性输入用户数据到程序中,而是输入一次缓冲一次)，另一种是非缓冲文件系统(每次输入就完成一次绝对的 i/o)，这是，C 把显示器，打引机像成标准输出，把用户输入键盘，想象成标准输入，，并把它们看成 " 标准输入出流 " ,,,把 OS 的文件想象成 filesteam.

5.5 所谓指针：当指针用于设计居多时

指针是 C 语言中的一种“语言机制”，它导致的差别在于：如果用得一般,指针就是一种普通的工具,仅仅在给函数传地址以改变实参,数组的定位本质是指针,,这些课题上达到顶点，而如果 C 语言的指针用得好，C 语言就会是另外一种语言。

那会是一种什么语言呢,那会是一种 Advanced Pointer C Lanuage(增强型指针 C 语言，指针使 C 变为设计语言就跟 C#高级语言一样，而不再仅是普通意义上拥有指针作为底层机制的中间语言)

因为指针是 C 语言唯一的"抽象语言机制",我们这里提出"抽象语言机制",说明可用于设计,,,比如 C++有"OO","范型"等等(很多书上讲解 C++没有讲解这是对的,因为 C++的语言机制中,只有 OO 和范型是它自己的,而指针几乎是 C 语言唯一的抽象语言机制)

指针被用于设计时，，它的用法有哪些？？ 这就是学 C 的最高境界，

首先指针是一种底层实现和设计通吃的语言机制汇编语言中也有指针，比如
mov eax dowrd ptr [某一地址]
mov edx dowrd ptr [某一地址]

这样的结构，，说明指针在这方面是一种内存地址的指针，，然而当指针发展到 C 的指针和 C++的引用时，，又形成了更高层的逻辑，，，

5.6 指针成就的 C 语言

学 C,,要学透就学指针,,,要学全,就要学 C 的标准库

指针不单是控制底层的工具,,,也是 C 的抽象(应用)形成语言机制或库逻辑的工具。

1) 字符串,C 的字符串跟指针密切相关.C 根本就是用内存来控制字符串的.而 C++的 lostream.h 你去看看,,全是抽象了的模块,你根本不需要深入内存,二者根本不可同日而语,,这是因为 C 就是站在底层去构造字符串逻辑的,,而 C++隐藏了这些,,不让程序员知道..所以 C 最适合当系统编程语言,而 C++控制系统的能力却没有 C 强,,,一个很好的例

子就是可嵌入开发，C++根本不行，因为 C++更多地是一种应用开发语言

2)数组,C 的指针几乎（我说几乎）等同数组，前期的 C 根本不能把数组当参数传递，，后来的 C 标准支持了这个观点，，因为数组在 C 的观点里就是内存地址．因此用指针完全可以控制数组，，函数可以返回一个指针以返回一个数组，，，数组也可以用指针而不用索引来定位并操作它的元素

３）位操作，，就是所谓的 bitwise 了，，这个跟指针的关系就更不用说了

综上所述，，C 是靠指针来进行设计它自身的，，，而且，C 是靠指针来解决 C 能解决的程序问题的．．因为 C 不仅是一种语言手段，，而且是一种语言抽象．．．

网上有用指针实现 C++的ＯＯ的文章，，读完它，你就发现 C 通过指针机制还可成为设计语言，，，什么是设计语言呢，，C++，ＶＢ，ＲＵＤＹ，ＬＵＡ，ＤＥＬＰＨＩ都是设计语言，而 C 就是底层语言．

C 是系统编程语言,,一般谈到对系统编程,,,就是数据结构加算法（当然，系统编程的真正概念是指：socket,graphics,IO，gui 这些平台支持逻辑相关的层面）,,,编译器前端的构造就是一个大算法,,,运行时就是一个大数据结构,,,操作系统作为语言运行的环境,,考察它的实现过程,也是数据结构加算法的集中体现,

如果你真要学透 C 语言,,,,第一要学习 C 语言是如何来的,,这就是编译后端了(动态运行期的类型信息),,,第二要学习语言本身的机制,,当然重点是指针..因为指针不单是内存地址,,,而且是一种语言的抽象机制,,,C 语言只有指针这种抽象机制,,就像 C++的 OO,范型一样

数据结构加算法是系统的观点,,所以容易跟 C 语言结合,,,C 语言也因此被用来系统编程

而 VB 是 RAD,,抽象了太多底层的东西,无所谓数据结构加算法,,所以被局限于代码组合和快速开发...因为它的接口就是控件属性和方法,,,它的功能模块就是控件..

其实 OS 不需要一个语言，语言只是为了开发应用而出现的，OS 可以不要它(语言编译器也属于系统软件)，更确切来说并不需要一个编译器，，它只需要一个

编译器的后端，比如运行时，或者仅仅一个解释器。有了这二个就可以执行源程序了，这二者之间的接口要明白,因为解释器可以直接执行编译器前端生成的中间代码

而且 VB 的运行时面向的根本就不是真实机器,,VB 是一种架空的语言..因此它的语言机制也是构空的,,,不是面向系统的.因此无所谓数据结构加算法

5.7 用抽象的眼光解读指针

指针是一种语言的抽象机制,而不是仅是语言用来控制低层的手段(指针指向变量,而变量是内存地址)
指针有什么用呢,,指针用"指针本身"来指代"它所指的东西本身"(虽然实质不同，一个是指针变量一个是指向的变量本身,二者除了意义上指代与被指代这层联系之外,其它方面没有任何联系,从变量的三个方面来看,
1.变量的型别,,每个指针变量都是一个指向某种型别的指针变量,它首先是变量,然后是指针变量,再然后是指向某某某型别的指针变量,第四是指向某具体变量的指针变量,被指的变量可以是其它类型,但指针变量都是一个 32 int
2,变量的作用域,指针变量只为索引被指代的变量,如果被指代物被释放了,而指针本身没有被释放,就成了野指针.
3,变量的所占的空间,不用说了吧
虽然存在极大的不同,但是,但是指针的精神要求你把它们二者建立"他们是相同的"这样的认识,这是基于抽象的考虑要求的
比如 int* pointoaint;????? //pointoaint 就是一个关于 int 类型的指代,,,
那么这种指代有什么用呢,(相比之下,引用比指代更能体验这一点,引用=它所指的对象本身,它所引用的对象本身,虽然这二个概念不等价,但指针这层抽象的意义就是让我们人脑把指针作这样的理解,)
在单根继承ＯＯ语言中(就是所有的语言级的 first class OO 对象都是从一个 Tobject 这样的东东继续而来,用户定义的 OO 对象也需要从这个根对象定义而来一样)实现泛编程的容器中，，比如 JAVA 的ＬＩＳＴ中，一般用指向ＴＯＢＪＥＣＴＳ的引用再填充这个ＬＩＳＴ，这样容器里的对象不是对象实体，而是指向他们的索引对象，，，
很显然，JAVA 的ＬＩＳＴ中的对象都是索引他们的引用变量，而非对象本身，是假对象.
另一方面，，，人们用索引来看待变量的方法，，这种行为自身也表明，，引用只是一种靠近人脑的抽象，，，是一种语言的抽象机制，，人们说指针是 C 语言的灵魂，，说的就是这个道理. .

5.8 指针的赋值：左值与右值

形如以下的一个表达式，char *p = &somechar;

我们知道 p 是指向字符的指针变量，作为一个变量，它的内容为 somechar 的地址，它的指向对象为 somechar，然而，p 不是不可以指向其它的字符，比如我们 p=&anotherchar;这同样成立。

我们说，在赋值表达式 char *p=&somechar;中，等号右边的 &somechar 是一个右值，

等号左边的 p 是左值；

也可这样说，对于 char *p=&somechar;显然等号左边也为一个表达式，在这个表达式中，p 是左值，而*p 是右值。

左值与右值的概念与区别最初来源于编译原理，大致是指表达式两边的左右值是有严格区别的，分左右的。

在 C 中，左值就是可以改变其值(被赋值)的一方，所以要求赋值表达式的左边是一个"变量"(不管右值是什么，左值接受右值的内容，此例中为&somechar，是一个 value,然而语义上可以看作为指针)；

而右值可以是变量，也可以是常量，或者临时值。(不管怎么样，它有值提供给左值就行)

归纳起来就是，右值把它的“值”(是个值或地址值)赋给左值所指的“地址”（必须是个有地址的量,比如不能对常量赋值）

所以在 C 中，左值一般指能变动其内容的，有地址的变量，而右值就是有值，可变动或不变，但不一定要求其有地址的量。

但不能单纯的其值变动或不变动来判断，比如 char *p = &somechar;的表达式中，显然地，p 和*p 都不固定可以变动其内容。

所以判断的唯一依据是：赋值中，是哪方把哪方的值赋给了哪方地址所在的空间。

在 C++中这两个概念经过了改动。因为 C++引入了引用这个东西，还有引用到常量。

形如 int& max(int a,int b)的函数表达式中， max 因为本质上是一个返回 int 的函数引用“类型”，

函数引用也是一种“语言类型”，而且它有一个地址（函数都有地址），而且这里还有&,所以它是一个左值。

而形如 int* max(int a,int b)的函数表达式中，max 因为本质上是一个返回 int 的函数指针“类型”，

函数指针类型也是一种“语言类型”，而且 max 必有一个地址，但它表明的是一个函数值,所以它是一个右值。

试归纳之，具名（显式给出了其名字）对象就是左值，不具名对象就是右值。

所以，对于 C 和 C++来说，总的判断步骤可以如下进行：

1，分解认识出表达式（左，右，或整个表达式）内含的“量”的类型；

2，因为任何类型都是语言的类型，所以可能有地址，有值。

3，具名且有地址的就可以是左值，不具名而且有值的就可以是右值。

5.9 C 抽象惯用法

诚然，C 只有有限的语言要素，它的语法只是稍微抽象了汇编,它适合描述系统底层，而不适用表达过高的抽象，比如它没有直接站在 OO 角度上去描述除系统底层问题之外的其它问题，C 的那些语法机制是 C 在语言级能用来表达所有问题的唯一方式(即使这样，C 还不失为一种系统编程和应用编程的综合语言，它的表达能力还是巨大的而且 C 的优点是易于学习进门但进阶难，不像别的语言连进门都难比如 OO 语言，它的 OO 机制一时半会是很难被初学者理解的)C 虽然没有语言级的 OO，不能直接用 C 的某种语法机制进行 OO 思维写 OO 代码，但 C 的一些简单语法机制同样可以实现模拟了的 OO 的抽象(作为库级抽象来进行 OO)。。这导致“如何用 C 达成高度抽象语句或程序”这样的问题的产生(不只是 OO，C 也可以实现 C++的 template 各种各样的抽象来间接解决现实问题而不满足于用 C 本身来直接解决问题,这里，间接解决现实问题定义 OO 等库逻辑，就是抽象的意思所在)。
这就是说，抽象并不用 C 的语法机制直接写应用，而是在进行某种设计，在C 模拟的 OO 中，它企图先在库级完成 C 模拟的 OO，再去解决它要解决的最终问题(是一种跟 C++的元编程一样的手法，是一种拙劣的抽象迂回行为)，当然这只是 C 抽象的一种，C 的抽象主要表现在四个方面，

1. 用C 来解释的数据结构问题本身就是某种抽象和设计,在一些教材讲解到的 C 版本的数据结构教学中，可以大量看到 C 写抽象的方式

2，C 的指针是 C 的主要抽象手段(这样说是因为一些其它 C 的语法要素也跟指针一样用于设计导致高抽象，)，C 的指针可直接产生很多复杂的思想

3 跟上面一开始谈到的一样，我们想用 C 构建某种靠近现实问题的抽象,比如我们想用 C 来实现 OO，或者 template,这样 C 的这些逻辑就成了 dsl 了。再来解决问题。

4,我们不想先实现一个 OO，template 这样的通用解决问题的 C 抽象，这样的通用抽象称为范式，而是直接面对要解决的问题，用 C 的语法语言要素去产生一个或一套习惯用

法的抽象。相比 OO，template 这些范式是小规模的抽象惯用法,下面我们来讨论一下 C 高抽象的习惯用法。。我们还有另一章专门讲解 OO 范式.

1),,用指针,struct,typedef,,这样的东西可以声明一个链表的结构本质，这在数据结构中大量看出。指针是“通过地址来操作变量”的抽象，struct 是建立在简单数据类型上复合数据类型的抽象，typedef 是 type redefine 的抽象。。这些抽象分别使用，或者组合使用可表示很多其它高级抽象。。
2)形参定义为指针，实参向它传地址，实质上是传值然而意义上是传送地址，这种指针的抽象用法，可用来影响调用函数中的实参。
3)void 这种类型可作为形参，实现一种多态机制，，void 函数也是如此.
4)C 语言数组的本质是地址，多行数组是按行优先描述的一维数组，因此其元素地址都可由数组名抽象得到.
5)字符串是一种变长字节，操作字符串的那些函数，也可操作内存块，这就把字符串抽象为一种数据结构的东西。
6)C 用 void (foo *)(int) 这样的结构来表示返回 void,以 int 为参的函数指针 foo.这遵守屏看原理.关于函数指针和数组指针还有很多抽象的变形。。来实现回调函数等抽象。
7)C 的指针在链表中，实现了一种"指针即变量自身"的指代作用，比如 stack->next=stack->next->next，这种指针的赋给，实际上就是指针所指变量在意义上的直接替换。。这也就是 C++中为指针新增的引用语法。。
8)typedef 实现了换名和子集定义抽象，用旧类型定义出新类型，新类型就是旧类型的别名，或者是旧类型的一个子集，这样的抽象语义。。特别是 define 实现了一种变量替换式的函数抽象。
9)用 const 或双层 const 进行防修改。
10)还有很多很多。。

5.10 C 的观点：底层不需要直接代码抽象

抽象是为了简单化,抽象=对机器的简单性=对人的复杂性

C++在语言语法级集成的抽象太多了,而且它的库级抽象也太多了,BOOST,编译期的泛型抽象,运行期的 OO 泛型抽象等,,OO 抽象等,,无一不表示,C++更适合当一门靠近应用的语言(只有高层应用要求抽象,底层开发就要不得太多抽象,因为机器本身就是机器,C 就是对应机器的,无抽象必要),,,这造成的结果是要全面掌握 C++要求人们掌握的知识太多了,C++的这么多因素使 C++变成了多范型,完全密封远离低层(然而你要知道,抽象意味着对底层的迂回,对人类思维的靠扰,这迂回多了,对底层的访问就少了),,,,其实 C++也明显没有损耗 C 操作底层的能力(我这里仅指它 OO 范型和模板等高级语法机制),,,,但是它提供高级的语言级和库级的抽象反而对人们要求掌握它的成本变得过高了,这就是说,C++对于开发人员提供的"形式,工具"过多(至少比 C 多).,提出了很多抽象,.在提供的抽象方面,C 只有指针,函数指针,结构体,这些有限的东西,而 C++呢,有 OO,有

TEMPLATE(语法级),有 STL,有 BOOST(库级)这么多
在开发人员这端,C++代码复杂难解,因为一个会读 C++源码的人首先要理解语言本身,才能理解作者要想在代码中描述的现实事物(机器逻辑通过 C++对于应用逻辑和现实事物的变换),,,而这里面,抽象太多了(甚至过度抽象,过度设计)
用 OO 表现的现实事物,相对于 C 用结构和函数表达的现实事物来说,,,后者更接近机器.C 本身的那么有限的语法机制,即使离开了语法级的 OO,C 也可直接用函数级的第一类型(组成的模块)和结构体数据抽象,和指针,,这些东西描述应用领域,比如 Windows 用 C 表达消息,,jxta 用 c 表达管道,端点这些概念,,,
语言的选择永远离不开应用,要开发底层,,C 比 C++好,WEB 开发是那些远离底层的逻辑,用 JAVA 可以,但是 C 其实也是 OO 的,它的结构体,本身就是一种数据抽象,比如我要描述一个学生,我就选择性的定义一个 struct student,里面加上学分啊,身高啊这些数据,(这是最好的方法,比 C++的类都要好,为什么呢,因为 C 语言的类型机制就是计算机的类型机制,,这些定义数据抽象做到了机器跟应用域事物的一一对应,非常地好)

而定义关于数据的抽象是冯氏语言模型的根本，我们知道可以用设计数据的方式抽象 DSL 概念词汇，系统编程支持词汇。

JAVA 明显不好用来开发底端,,这就是语言适配应用的道理,,但是计算机终久是计算机,如果你想学习计算机本身而不仅仅是想通过掌握 JAVA 开发 WEB,那么 C+LINUX 永远是最好的选择,因为开源的东西一般出现在 LINUX 下.而且通常用 C 开发的,世界上开源项目最多的就是 C 了.
一句话,C 比 C++简单,如果想开发图形啊,网络啊,这些跟计算机本身有关的东西,这样的一个想靠掌握语言马上编程的初学者最好去学习 C,而不是 C++

5.11 真正的 typedef

typedef 是类型替代名(typedef=type dfine 嘛)，在理解时可以按以下的方法进行

typedef 是 C 语言不够丰富的类型再造类型的一种手段(是一种拙劣不直接的抽象手段)。

1.typedef 是定义一种类型的子集，，，比如 typedef int INT;(这意思就是说，INT 是 int 的一个子集)

2.类 extern 的这样一种声明(也即仅仅是对类型的向外的一种再声明)，如 extern int myint;(这意思就是说，作为变量的 myint 这里作为一种“新的类型”，而且是一种“int 类型”,,,这是一种当已有类型的再声明)

也即，extern 就是相对于变量，typedef 就相对于类型(注意它们二者并不实际等价，也即不会有 myint 这个实际变量被声明出来)

注意，一般 myint 要大写，但是这里为了说明还是采用它的小写形式

由第二种理解方式可以导出很多，如 typedef int (*myint)(),,,可理解为 extern int (*myint)(),,这里的 myint 也是一种类型，这种类型表示“指向一个返回 int 类型的函数的指针类型”

5.12 指针与指针类型

一般来说，我们为某块内存中的对象设立它的指针，只为能使用它，,，因此，指针就是关于此对象的使用抽象，中间层，，我们不必知道这个对象的具体情况就可以操作它(即使不知道它是一个什么变量的情况下)，，这个指针便提供了操作它的全部接口,对这个对象的访问与修改操作都是通过这个指针而来的，,，明白这个道理有什么呢
比如有如下一个程序

```
int var1;
int * var2 = new int;
var2=&var1;
```

为了使用变量 var2,,我们当然可以直接通过 var1 变量(也即 var1 名称本身)来访问到它,但是有什么我们还需要定义使用它的中间层，这就是指向 var1 的指针(换言之，我们并不想通过 var1 变量名称本身来访问变量)
一个对象的指针是关于这个对象到它的使用级的第一层间接，指针的指针就是二层间接，，
也即，为一个变量定义一个指针等价于计划通过这个指针去访问对象，（这个指针也可实现访问控制）
如上所示，这第一层间接和第二层间接都是通过 type*的形式出来的，这样就保证了与原对象的充分脱钩
也即 int * var 完全可以和原对象脱节(var2 这是第一层间接的其中一个指针，因为可以为同一对象声明多指针)
这样到底有什么好处呢？
有时我们想访问 var1 的同时不想改变它，这固然可以变 int var1 为 const int var1,然而这样改变了 var1 的属性(也即我们需要在不改变 var1 的属性下实现“访问 var1 却能保证这个访问过程并不会改变 var1”)
于是，我们可以用 const 修饰修饰指针指向的对象，

```
int var1;
const int * var2 = new int;
```

(这样*var2 就是 const int 了,因此 var2 指向的是一个不能改变的 int 值，虽然 var2 的值依然是 var1 的地址,,但我们的确实现在使用层就保证使用过程也不会改变原 var1 的值)

var2=&var1;
对于 var2 来说(虽然我们不知道还有多少指针会跟 var2 一样指向对象),,它作为一个指针出现以访问和操作它指向的对象,,,在 var2 的层面，我们只能通过这个指针去修改对象,,,可是，var2 被定义为 const int*,,,因此并不能使用它来修改指向的对象
即，除了类型之外，指针的声明可以与它指向并要操作的对象无关（比如关于操作的 const,,,它就可以仅仅并施行于 var2 上，而并不要求它的指向对象为 const,,因为 var2 只是关于原对象的使用逻辑而已）
总结：指针是关于一个对象的间接访问层及访问控制逻辑集合点，那么什么是指针作为类型呢？

type*
这个形式就表示一种类型,,,,如果说 type 本身也是一种类型的话，那么在 type 后加一个星号也表示一种类型,,这种类型叫“指向这种 type 的指针类型(重要的是最后的指针类型这四个字)”,,你可以联想 dephi 中的 ptype 类型
所以你就可以这样定义东西
type* someobj;
(*可向 type 靠近或 someobj 靠近，C 程序员偏向于向 someobj 而 C++程序员偏向于向 type 靠近,这样的话就有二种等价意义， type *someobj 表示*someobj 是一种 type 变量,type* someobj 表示 someobj 是一个指向 type 的指针)
someobj 就是一个指针变量，它代表指向此 type 的的指针变量,,因此在这里 type*是类型(是一种指针意义上的数据类型)，someobj 是这种类型的一个变量
根本上引用只是一个指针的别名，因此你可以由一个指针得到一个或多个引用(即关于该指针指向的对象的引用)
指针本质上是一个 32 位 long int(因此在 Generic progamming 领域有“用整型代替类型”的说法),定义一个指针时可定义 void*型指针(因为指针并不一定出生时就要指向一个对象或一块内存,而引用要求有一个初始值，因为引用本质是指针的别名)，因此它可以被指定为 0,即空指针，也可以被重新赋值(此时它就不指向它原来指向的对象或内存了),,换言之,,多个指针可以同时指向一个对象,你可以为一个对象定义多个指向它的指针(但是它并不拥有这一内存，也就是说，当它指向的对象发生了变化，指针仅仅作为指向这个对象在内存中的位置的意义就会失效或过时,它只拥有对象的 adress 值而非 value 值)，而你可以通过这些指针或引用来操作该对象
一般来说,引用常常跟 const 在一起(加上 const 只是为了强制跟保险)，那是因为定义出来的引用常常不会改变

5.13 真正的函数指针

C 和 C++都不允许一个真正的函数作为参数,,将函数作为参数传送的办法是将它用一个指针去指向这个函数,,,,

而且，通过函数指针和函数模板，可以很有效地实现业务逻辑跟界面逻辑分开(你看，设计模式能显式解决的问题---虽然设计模式也不是表现得很好，在C里面要靠拙劣的函数指针这种本来就表函数指针的东西旁敲侧击地体现，所以，C从来都不是设计语言，而是专家和怪才的语言)

学习函数指针完全是一种挑战，你能分别出以下的吗？

int *(*(*foo)(int))[5];

这个就他妈的表示：foo 是一个函数指针(这就是括号带星号的结果-即把*foo 括起来的那个括号)，它指向一个函数(以指向它的指针 foo 命名的函数)，它带一个 int 参数(即函数指针 foo 紧靠右的那个括号和括号内的 int)，，并返回一个指向数组的指针

所以，这样的语言留之何用？？只能做做系统！！

5.14 真正的句柄

句柄一般有三种意思
1,Window 的资源句柄

一个语言跟操作系统的关系是什么？？

一个操作系统可以用一种语言写成，，因此如果语言出现在前(历史原因)，那么操系统（或其一个小节）会用这种语言来描述，，，比如 Windows 的对象，，就是说如果 Windows 是用 C++来实现的，那么这个对象就是 CUI（**—对象

根本历史，，一方用另一方的东东来描述的关系

比如文件，，操作系统普通把文件作为硬盘上的东西，，可是文件在语言的观点里根本就不是这么一回事，，，文件只是 data flow,,或者一个 file map(也即在内存中作引象),,,,

2,包装被指对象为其值的智能指针
3,形如 type**的双层指针
4,其实通用意义上的 Handle 就是句柄的意思，句柄还有一种用处出现在编译原理中
5,如下

如果 B 是 A 的一个子类，，有如下代码

(A)new B

那么以上可以理解为，，，上面产生了一个对象，，这个对象指向 B（指针），，，但是其句柄为 A

即 a handle of A pointing
to a B,

也即第四种指针是用父类来操作其子类对象的一种机制

5.15 真正的 static

在 C++中，一个结构也有它的构造函数，这个构造函数（接口类不应包含构造函数）往往作为初始化该结构的初值使用，因此一般将它们作为 static 来定义，这就不得不说说 static 到底给语言增加了什么？
static 的成员也不是静态的，它也可以进行自加自减这样的操作，
static 是在编译期发生的，因此不能在运行期改变它
在一个函数体内定义的 static 数据，，相对整个程序来说都是全局的(直到被销毁)
因此 static 提供了一种把局部自动变量转化为暂时全局性的东东，，比如对于一个在函数体内的常量来说

static 也被作为类级的私有成员数据或行为(类属函数)，因为类只是定义对象的规范，因此它应该没有一块内存实际存在用于调用它的成员（一般方法是具体化出它的一个对象然后调用对象在类中定义的成员），但是 static 提供了一种“全局静态”的概念(类级的全局)，比如 Java 中 systme 名字空间的 API 就是全局的 API（包级的全局），可以直接拿来用

函数退出这个常量就失效了，解决的方法是用 static 或 const
static 的作用就在于“保值”时空限度为定义它的整个模块，，这是相对静态局部变量来说的，，还有静态全局变量
要注意，一个函数的原型和它的具体实现是分开的，你可以 overroad(复写)一个基类的 private 的函数声明，但却不能调用它的实现
也即，只能复写声明，而不能调用定义（实现）

在 C 和 C++中，声明和定义是分开的，如 int i=9;就是一种定义（因为连初值都赋过了这说明分配了内存），而 extern int i;只是一种声明而已（向外界宣告它的存在），而 Java 中就没有声明跟定义的差别了。上面是对于变量来说的（特别是对动态对象—也即对象变量这种表现更加明显），，对于函数的声明和定义的区别就更加明显了。

5.16 真正的数组索引

指针经常跟数组在一起，因此它也经常跟索引相关
因为字符串也是一个数组(更准确来说是可以用数组来实现的线性表，只不过它的最后一个字符是/0 而已)，所以形如 char*的字符串也一定跟数组有关
如果一个指针指向一个 new type[num]形式开辟的数组，那么经常有下面的形式出现，
数组名=指针=数组的索引 1 的地址=用双引号括起来的一串字符串
char* myvar = “iloveu”是用字符指针指向字符串第一个字符在内存中的位置，char[] myvar2=”iloveu2”也是成立的(字符串往往以字符指针或这种字符数组来表达)。
指针加 1 等价于索引也加 1，但是这其中发生的本质是不一样的，，指针值加了 sizeof(type)的值，，而索引只是向后一个索引递进而已
指针只能在堆上存在，，，而不能在一个函数的栈帧上存在,alloca 可在一个栈上声明
“同一”与“等价”的区别就在这里出现了
实际上数组的地址的确等于它的第一个元素的地址，，然而数组的地址并不是它的第一个元素的地址，而是这个完整数组的地址开头
mfc 的消息机制就是一种表驱动

5.17 类型和屏看原理,以及近看原理

要深克地理解 C 语言中的“屏看”原理，比如下面二个例子

```
void (int *p1,int *p2)
#define mysub(x) x^3/x
```

在第一句中，参数是 p1,而不是 int，即不是*p1,我们优先看到 p1,p2 这样的变量，即参数是变量

在第二句中，define 的是 mysub(x)这个整体，而不是 x,我们优先看到 mysub(x)，即 define 语句的整个前半部分

一些名理:

命令行就是观念里自由的世界(编程时从观念里出来，去除表现逻辑和其它逻辑,就事论事,不必输出到 gui 上，只直接面向输入输出到标准,这些语言级而非应用逻辑级的东西)，GUI 就是强加了面具做人的不自由

能图灵处理的问题，就是计算机能解决得了的问题，可通过程序手段反映出来。

我们称最小接口为原子操作，因为由它们可以演化得到更高级的接口。

一个源程序模块或二进制模式能运行，一定同时包含了指导 CPU 的指令和待处理在内存中的数据。。

解释器就是直接运行中间码，不需要完成到目标码的转换。。在编译前端，它还是比较跟编译器一样的。。

近看就是当有多个 const 在后面的时候，此时容易混淆 const 与它修饰的对象，而可以作如下判断：修饰词 const 往往与它修饰的变量最向内靠近，就像 code block 的花括号一样。

5.18 位操作与多维数组指针与元素

首先 C 语言提供位一级的操作是为了迎合汇编语言，，回忆一下 CPU 的指令集，，有专门的位操作指令，C 只是稍稍地将它们进行了下抽象（这样的话，C 有了位操作再加上指针，它就成为系统和控制语言了，抽象不但隔断不必要的细节，而且在另一层维度上，提供了对于人来说更为强大的功能，这就是抽象的二大特征），对比一下我们就知道它们的差别并不大，，CPU 还有 OF，和 CF 这二个寄存器来表示移位和运算时发生的进位和溢出现象。。

我们知道，在存储位的时候(更确切地说是一个字节级的东西)，Intel 的 CPU 是按高位在前(书写或打印时显示在左)，低位在后存储的(在右)。因此位往左边移的时候（因为左边是高位，位到了左边就占了一个高分位,整个二进制会按基数越来越大），就是乘以 2 的幂，体现在它的十位制表示会越来越大，C 语言规定，一个类型的值向左移位的时候(右边会溢出不用)，在它的右边低位加 0,一直补满这个类型应有的位数。。而这个值向右移位的时候，位到了低分位，整个二进制按基数 2 越来越小，，移了几次位就相当于除以 2 的几次幂。。移位的时候，左边补 0,如果原数左边是 0,是个正数，那么还是补 0，如果原数左边本来是 1,是个负数，那么根据不同的编译器会采取不同的动作，有的编译器把它看成“简单算术右移”，最左边那个位还是补 1，有的编译器会把它看成“逻辑右移”，，会把最左边空出来的位加 0..

有人会问了，移位会不会把周围内存的位给挤了，不会的，因为所有的移位操作，只局限于这个类型的这个值，跟内存中其它值(只要它们都不参与此次位移运算)都没有关系。。如果二个不同类型的值用一个二目移位符连接时，编译器会进行对齐操作以使它们有相同的位数。。

位定义了几种运算符，有与(实际上不是逻辑乘，但我们按乘的原理可以得出它的计算方式)，或(逻辑加)，异或，取反，各种运算用在不同的目的下，与可以用一个屏蔽字

屏蔽给定值特定位，就是把它们置 0,,,,或可以用一个屏蔽字屏蔽给定值特定位，，就是用 1 去保留它们。。异或可 s 以反转特定位。。

这样所有的工作就成了找一个需要的屏蔽字(根据原值，，面向要作什么样的屏蔽要求，这二个因素)。。

在单维数组中，，指针跟"&数组名[索引]"或"数组名+索引"的方式有点相似，，实际上这是 C 语言语义的二义性，，这是为了让 C 语言变得灵活而保留的。。而在多维数组中，&数组名[索引]"或"数组名+索引"往往并不是一回事。。这一切都是因为 C 语言中的多维数组其本质还是单维数组，，因此用单维数组的索引或指针方式用在多维中，就会出现很多语义。。

我们来分析一下，可以得出以下结论：

首先，我们知道数组名并非指针，它只是一个意义上的等同，实际上存在数组名作为指针，，但是不存在"&数组名[索引]"这样的元素，，它表示一个地址(从...开始的地址)而非表示一个元素。。尤其是在多维数组中要分明白。。

5.19 形象理解 C 语言中的一些至关重要的二义性

对于多人共工的软工来说，一门语言的灵活性反而是它的缺点所在，C 语言的灵活性很大一部分在于它的二义性(语言的或者非语言的都有)，甚至多义性，只要掌握了二义性，那么 C 语言就是一门不灵活的语言了，下面试举几例：

C 语言中没有字符串这种“型”，C 的字符串实际上是字符数组。只有字符才是“型”；

C 的布尔也是数值(当然，新的 C99 有真正的布尔了)

C 中的字符是数值（编码的），所以可以跟数值型作转换或计算。因为字符就是“字节”，所以像 memcpy()这样的函数一方面是字符中函数，另一方面却又是内存数据操作函数。

C 的赋值是一种表达式，而非语句，所以可以跟其它表达式进行无穷有意义的复合。

C 的多维数组根本就是一种本质上由一维数组变来的，C 在语言上只支持 built-in“单维数组”；

C 的“可变数组”实际上并非“动态数组”，动态数组指数组的大小可以通过增删元素来

变其大小，而可变数组光指数组的维的大小。

而且所谓可变，，也并非真的在运行期可变（否则就是运态数组了），而是编译期可为维数大小指定一变量。

在用了数组作为函数的情况下，对函数原型的声明可以省略变量名写出来。

C 的传指针给函数，实际上还是传值，不给传的是指针值。虽然语义上正确，但语法上没变。

C 中，一看到数组名就要把它跟数组第一个元素的地址相提并论(但它毕竟是相等，不是同一)，不能产生其它想法。这造成了对数组元素表达上的多义性。

Int myarrary[10];int *ptr=arrary+5;

在这种情况下， ptr[2]是错误的定法。因为不能对一个不是数组的 ptr 取下标。

屏看和近看原理。比如

int *p 实际上正是 int* p;

int const a 实际上正是 const int a;

int const *ptr 跟 int * const ptr 不同，前者为指向“指向 const int 的 ptr 指针”，后者为“指向 int 的 const 指针 ptr”

如果指针这种东西，跟数组和函数联系起来，情况那就更复杂了，我们一步一步来了解其复杂性

Int (*myfun)(int arg1,int arg2);

这就表示， myfun 为一“函数指针”，它所指的函数以 arg1,2 为参，并返回一个 int。

而 Int * (*myfun)(int arg1,int arg2);就表示，myfun 为一函数指针，它所指的函数以 arg1,2 为参，并返回一个指向 int 的指针。

当然，这里不存在“指针函数”之说（返回指针的函数还需要说？？？-_-;）。
但对于数组来说，就既存在“数组指针”和“指针数组”了。

Int * myarray[];就表示， myarrary 为一个 int *数组，即一个由 int *为内容构成的数组，

是“指针数组”。

Int (*myarrary)[]；就是数组指针了。Myarrary 是一个指针，它指向一个 int [];
是“数组指针”；

如果把二者结合起来，情况就更复杂一点

Int (*mywhat[])(int arg1,int arg2); ---与函数结合讨论的情况下，[]是不会跑到括号外的。

（这样我们就可以根据上面谈到的第一种情况开始，结合后来的三种情况来推断）这就表示，mywhat 是一个函数指针，不过有多个这样的函数指针构成一个[]，因此 mywhat 是一个函数指针的数组(所以 mywhat 是 myarrary)，其中的每一个函数指针，它指向的函数以 int arg1,2 为参，返回一个 int。

那么 Int *(*mywhat[])(int arg1,int arg2);呢？？不用我说了吧

#define 实际上只是一种编译前的替换，所以#define someformdim char*; someformdim a,b； 实际上定义了 char *a,b；
（当然还有其它，比如经替换产生的声明语句会导致在它出现的位置无效）

总而言之，C 的二义性正是它的优点也是缺点，C 正是借助这些具备多义的东西达到一种紧凑的效果。

第 6 章 抽象与设计

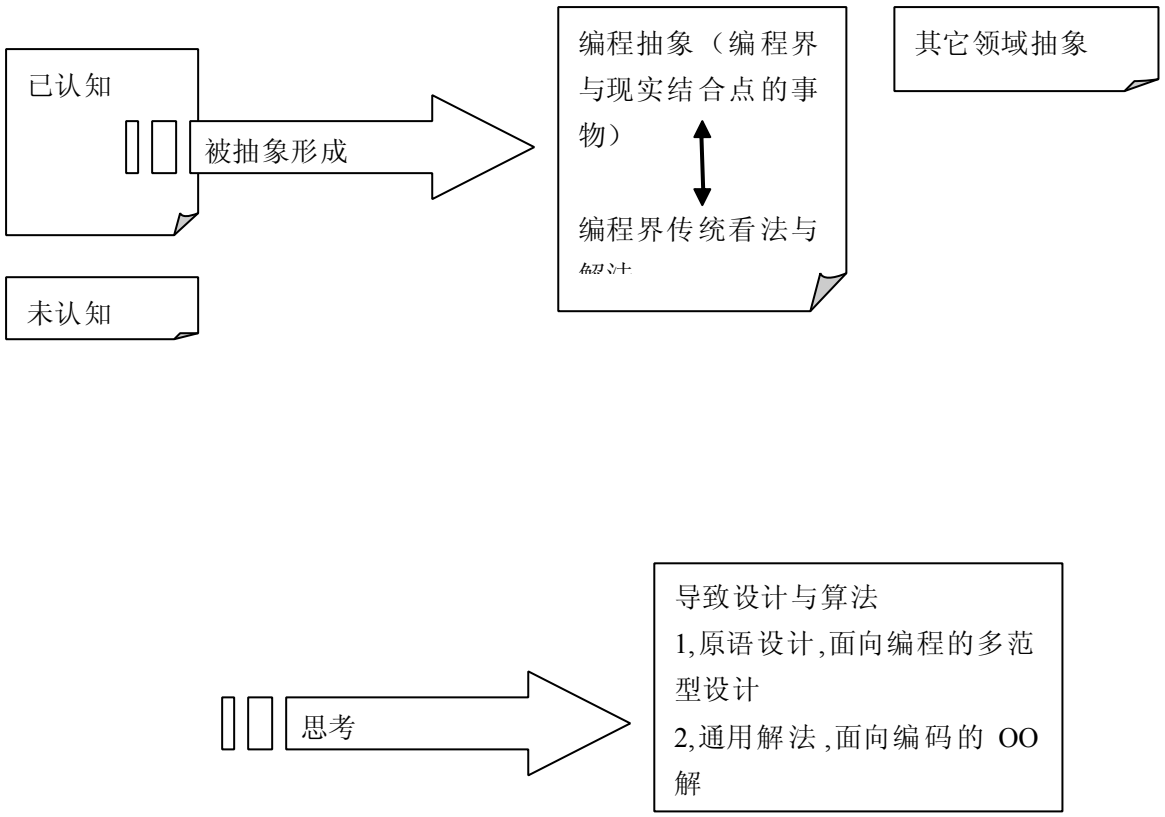
大师的禅语并非无所指，一切都只是因为我们并没有拿到那个水晶球而已!!

思想本非也不该是高高在上的东西，只是我们没有勇气跟它平起平坐而已，而且求知的你只是缺少一个求知切合点而已（这就是每个人心中的“实践认知”,，某个特别的维度认识）。

因为这三个思想实在太重要了而它们又独立于所有知识，更并且，它们不被任何其它书籍提到并组织，所以在这专门作为一章来讲解：）

现实世界(问题或事物原语空间)

经过领域抽象的原语(总看法和解法)



编程与思想

6.1 什么是抽象，软件即抽象

某某牛人说过，增加抽象是解决一切编程问题的方法!!

对于能解释抽象的概念，在编程界一抓一大把。从本书序言开始，我们就不断地提到这个词。足见理解它对理解编程的重要性。(这本书主要是从开发的角度抽象地讲解一系列专业概念)

那么，为什么需要抽象呢?

首先，人脑往往不适于长辐记忆或直接面对复杂的二进制底层，人们在面对根本无法控制的事情时，往往把它们转化为另外一件可控的事，抽象正是这样一种方法，它可以隐藏低级层面的复杂性，而在另一个层面上提供新的更为强大的能力。再在这里抽象上构建更为高层的抽象，即抽象只是把问题变了个形式（顾名思义，抽象抽象，抽取事物形象的一面），抽象完成了之后，只要不是过度抽象，那么所有后来的事情都是另外一回事了，比如抽象了数据类型，那么关于数据的逻辑都成了数据结构学了，再比如 OS 对硬件的封装就把所有硬件上的工作搬到软件上了。再比如语言的类型用来抽象表达一

些数据(类型)或事物，我们就在高级语言上工作了。

其次，从问题到解决不是一步而就的，所以需要建立中间层，先完成这诸多中间层，当中间的逻辑被解决的时候，事情自然就变得简单了，比如 OO，一个大型程序可能包含成百上千个子逻辑，如果是一个人，他可能理得清但是可能写不完，对于多人共工的软件来说写得完但是不一定大家能对这个系统共享一种认识，所以一层一层通过 OO 来建立抽象，先类化小逻辑，另外，OO 还是一种对数据和代码进行统一抽象的方式。。这样代码不再是数据加语句，而是统一的类了。你看，事情越来越清了。

抽象是解决移殖问题最好的方法，只有把接口拉高，向高层抽象，那么就可以忽视平台逻辑(实际上只是正面绕过)，比如 Linux 下一个底层函数是 a(),windows 下一个底层函数是 b(),为了在我的库中实现一个跨 linux 和 windows 的 ab(),我就需要在 ab() 内封装它们，这样我的库只需要 ab() 这个高阶接口，因为这个抽象，我却可以跨平台。

在架构中，抽象也是很常见的，比如七层模型，只有解决了前面的问题，才能着手下一层更高级的问题，这是目的也是原因，，是起点也是下一个终点。

我们知道抽象的本质在于对人的简单性，正如上面所说，比如 OO 的三重机制制造的抽象就在于统一数据和代码，于是产生了复用效益。抽象的本质在于远离问题，从靠近人的一个高层角度去解决更高级的问题。 但是抽象的优点正是它的限制性，它可能带来再大的复杂性，一般抽象到了某个程度，为了获得计算机作为底层的冯氏能力，，就不应该再抽象下去了。开发模型不需要再变了，数据抽象到数据结构级就是顶级了再抽象就不是开发问题了，现在的虚拟机的提出，都是基于已有的模式，从来没有那个虚拟机，其内部结构不是图灵模型，因为如果那样的话，它上面的开发模型将不再是数据加代码的方式。从来没有人突破过这个创新。仅仅因为大部分人没有想过，或根本无法尝试。

其次，抽象是抽取对象的可用部分，比如 OO 化一个概念，现实生活中一只猪有毛，毛色，毛还有化学成份，猪由猪毛，猪皮，猪肉，猪骨组成，这样东西在编程时都需要统统被考虑被抽象吗？不。我们从来都是抽取事情对于我们的可用部分，所以设计时千万不能做大而全的抽象。

实际上软件的设计哲学是可以用来解释一切的,因为它是真正的哲学,而真正的哲学并不仅适用软件开发(软工和计算机是二个完全不同的抽象,虽然没有人提出过计算机抽象到底是什么,软工抽象到底里面有哪些抽象存在,我们仅能站在某个或某些维度上给出一个描述性的概念而不是有限集,这也就够了,如果能站在一个大全的维度上说明到软工的全部抽象,虽然这是不可能的,但我们还是给得出的这个结果取个名字,叫范式,范式在意义上是大全而的抽象,然而人类的范式总表现为某些维度上的产物(只是我们在一本无论多么长的书里都写不完而已,如果有那么一种生命力和那么一本书存在,我们也坚信任何一个道理都不可能写完)。下面详细介绍这个唯度的概念。

首先我们来问个问题，程序如何分类呢，从算法和数据结构的角度看我们可以发现，数据结构加算法等于程序。因为数据结构源于从一套相似的算法中找出操作对象的共性这个现实，而从复用来看呢，，又可以产生设计和接口就等于程序这种说法，因此这完全是不同事物的不同唯度而已。。根本没有可比性。（至少二者都可以产生程序这个概念，于是，程序=机器加电也是正确的）抽象把事物的复杂度换化到另一层面，实际上也是另一唯度。

这就是抽象与唯度.

6.2 具象与抽象

抽象源于一个简单的事实，把事物从逻辑上分开，这样就会解偶它们之间的联系。

抽象要适中，不能抽象得太像了，为程序员复用的相对低阶接口变成了为脚本程序员的高阶接口。

复用由二次开发的程序员决定（这个事实决定了你将向它们提供什么大层次上的功能比如 ogre 的复合模式，大的逻辑或小的接口），所以复用是程序员面向的，纯 dp 的设计方案要求二次开发的程序员也要了解 DP。
对 api 的调用在此设计的最下层，由二次复用者提供。（即数据由它们提供，代码逻辑由我设计）

其实你可以将复用抬高到场景这样的层次（这就是领域逻辑，当然设计也可不深入到这么高的境界，这就是代码模式，设计模式和现实模式之间的整合设计所在），这是由复用面向决定的即需求决定的。而不仅仅是单方面的设计。

首先来对游戏进行划分。

6.3 应用与抽象

所有的术语都可以被重新定义，，游戏是什么，，其实火星人可能也在玩一种叫＂游戏＂的东西，，我们的ＢＢＳ论坛也可以和魔兽世界一样被归为网络游戏，，，这就是重新看待一个领域的抽象，，给一个术语重新格定它的含义的范围，如果性质相同或相似，，就整合它们，再发展形成一个架构，将它们发展成此架构下的分支实现，，比如 facebook,它以技术的形式统一了很多 web2.0 的应用，，，这就是说，应用这个东西，，是可以以技术的形式被统一的，，，从架构 web2.0 的眼光来看，，blog,视频点插，，，诸多 web2.0 应用，，都可以被看作为 web2.0,,,

这种分离与整合现象其实在ＩＴ界每天都在发生，,以上 facebook 是个例子，还有 adobe 的 Java developer ide? ecillpe??,Java?平台开发库将内存流，网络，本地文件都看成＂流＂，，这就是对一个术语重新进行定义，格定它的范围，，而这是合理的，，因为我们对一个术语的定义本来就是历史现象，当历史发展了，，一个术语要么被增加新的内容（量变），要么被完全演变，，成为一个新的术语（独立发展成一个新东西，虽然原来的那个术语也有效）

?我们所看到的概念，，如果重新被设计，，会产生很多新的抽象(在另外的维度上甚至会产生更多抽象，只不过我们是人，有限的生命不能允许我们同时或异时站在多个维度去想东西)，，和由此而生很多应用,,这就是 sun 所玩的游戏，，比如它提出一个"xml",,实际上 xml 的最高境界就是＂文档互换的标准＂，，由于 xml 的成功流行，这由于它是符合应用的，它就成了标准，实际上如果随着历史发展，xml 就会过时(Ｊnos 出现了)，，xml 只不过是人类知识的临时品，,，总会有它的代替品出现，xml 相对"文档交互的标准"这个说法是个实现，而＂文档交互标准＂这个说法是个思想，一种思想反映在ＩＴ界，可以用代码实现（细节级的)，，也可以用构架来形成一个观念上的应用规范，，比如ＸＭＬ规范，，这种思想一定要理解，

火星人也把他们玩的一种东西称为＂游戏＂，，任何到现在为止我们能耳听目见的东西，，其实都不像我们想象的一样简单，，当你学习西红柿的单词时，如果你不能了解到它其实是一种外来词，，这种现象，，这种对一个术语的＂历史抽象＂，，那么你就不能有效地学习它，只能说片面了解了它，，而这对学习是不利的?

应用越来越接近人了,比如 web2.0,3.0 的出现,这是指软件产品的应用,实际上在软件被作为产品被产生出来时也作了靠近人的调整,编程领域的三个东西,问题域,方案域,人都在相互影响,这种影响产生了一些技术,导致了一些编程的变革,并最终与人们的观念结合,比如 00,比如设计模式,这也将导致架构变成软件的功能性"实现"要考虑的,在某个维度上加深了复杂度,然而却在另外一些维度上提供了简单的形式和更有效的应用

互联网的最初灵感来自对学术自由和开放的向往，而现在它已成为由企业和运营商控制的商业平台。企业应用与网络的发展密不可分,这二者相互发展,成为影响软件工程界的二大主力

多维这个字眼本身就提倡从多个方面(可见多维就是多方面,当我站在某个维度为我自己说话时,我将同时失去另外其它的维度)

某些东西越来越统一和规范了,这加大了学习的门槛,比如 xml 出现,就统一了文档交互的格式,并导致了很多逻辑知识,产生了一些新的逻辑,需要被学习,但这是合理的,因为形式更加简单了统一了,并改变了一些应用的形式,比如软件分发 delopy 的统一形式等,

另外一趋势,应用越来越分布了和趋向 web,这实际上是很多年前某些大公司的战略,总有那么一群人(有些人研究应用形成架构,有些人研究编程低层形成架构和思想),先知

先觉地认识到一些东西,比如.net 的出现,网上的资源服务器越来越变成一般应用服务器,富客户端的 flex,silverlight 等等,只是它们是慢慢被民间所识所学习.

一切技术都是面向被应用,因此人无论如何都是主导.将反过来最终影响技术的被利用形式而隐藏了低层实现,一些离最终应用跨度太大的低层实现不必知道其原理,靠近人的一端要提供尽量简单的形式,比如 xml,比如 oo,面向机器的一端永远有它的实现.

应用往往是分布的（现在的网络服务器主要是应用服务器而非资源共享服务器），面向对象领域的很多东西都越来越趋于 WEB 这个分类粒度，Web 使我们的应用分布于网络，使 WEB 往往成为一个分布式的并发 OS，(网络最初是纯粹的通信网，后来才更名为计算机网络，是因为出现了 NetWork OS,今天，面向对象和 RMI,RPC,Corba,DCOM，使对象可以在网络上不同的计算机间交互，)

越来越趋于 WEB 这个粒度(粒度是我们作原语设计时用到的一些辅助用词)这个事实表明，分布式计算和面向构造软件设计的企业部署需要被学习.
Dcom 是微软用来抗争 Corba 的东东,，微软是一个很有自己理念的公司，Java 等的崛起，使虚拟平台呈现多元发展

Java 和.NET 是真正的 OO 语言，独立平台，然而它不是本地平台，一方面，不由本地 OS 直接分配内存，另一方面，它们是动态成型的语言，而不是编译期静态语言，因此速度上会比 Native 普通程序慢好多(虽然也有 JIT 技术的支持)，但是据称,，JAVA 速度越来越接近于 C++(不知道是本地 C++还是 C#,这里说的 JAVA 是指 JAVA 库和 JVM 的综合)

6.4 软件活动的特点

软件需要你考虑每一个细节，几乎每个中大型软件的设计都可以是以真正工程规模来较其大小的活动。因为这不比盖房子,，做软件需要你从大量小逻辑促成应用(每一个部分都可能需要你造轮子)。

而在我们的方案域用于表达应用的抽象元素只有类是最大的，库是最大的，这种细小性决定要去的地方，要达到的应用还很远。软件活动是一个真正体现人类心智的地方。是一个真正工程级的活动，个人几乎不能独立完成(程序员作为实现者可以不懂大设计和系统级的设计，但设计师要管)。

而且现实生活中软件活动通常都是变化的：代码在变化，设计在变化，连需求都在变化。别空想设计，设计也是源于迫切要解决的问题(设计跟需求分析有关)，你说的设计需要可能是一个空想的禁固。。

从更大的范围讲，，任何一种设计都是自蚕自缠，，当然，适合当前应用的设计总是存在的

设计的合理性只能是相对的，我们只能做一种“目前最科学”的有限设计，而且这种设计方案也受到问题本身的影响，并不是所有所有目前合理的设计都要拿来用在同一个设计上

C++为什么不是单一 OO 泛型而是多泛型的呢，因为设计就是多选题，你无法找到其单一性，这就跟 policy based design 一样的道理。

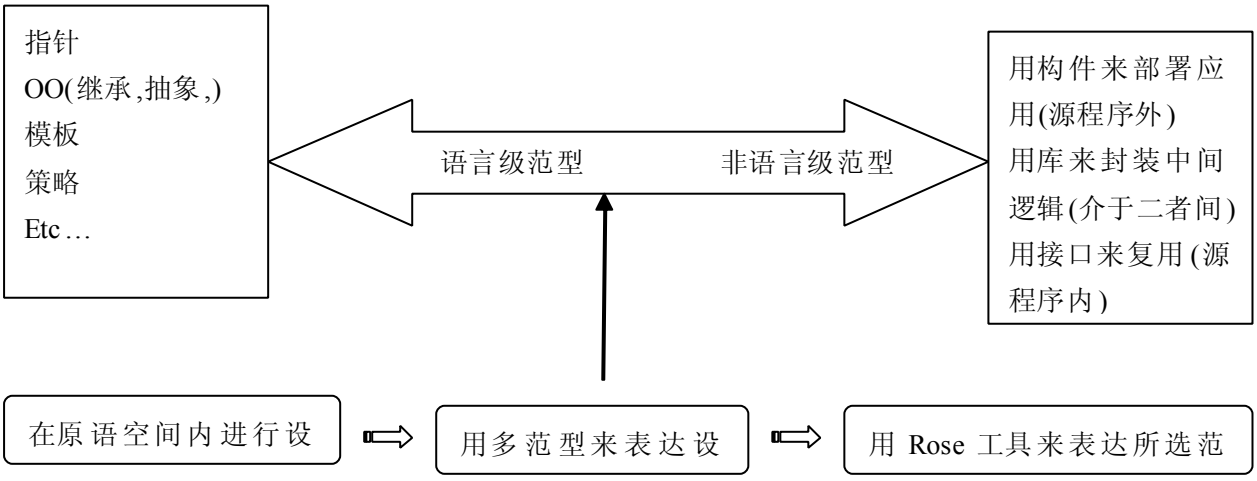
实际上，软件的设计可以不具有任何需求的成份，，只是在人类的具体活动中，往往需求会影响设计而已。。

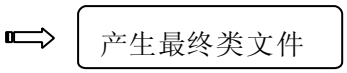
模式分为三种，问题模式(比如数据结构)，设计模式（应该算是结合了问题和代码模式的综合模式了），代码模式（比如 00，高级语言机制），MVC 是一种表达窗口的现实问题模式，此时用 mvc 复用设计模式就最好表达了。而 mvc 被 policy based design 实现了，所以它首先也是一种代码模式了。

设计期聚集于程序员，运行期聚集于用户，，比如当设计深入应用时，语言的设计和运行功能都要很强，，设计时不可能预测用户点了什么菜单。

6.5 大设计

我们曾在本书开始的时候就谈到什么是编程(它的狭义)，编程的广义就是软工，软工的广义就是设计，而设计的广义是大设计。





狭义编程与范型（设计）

范意上的设计是广泛的,不仅限于计算机的,也不限于软工抽象，我们必须明白，设计这个词首先是一个表示人类活动的词，比如有建筑设计，艺术设计，，而程序设计只是一种。它可以解读为用程序设计语言提供的基本语法和高级语法，以及独立于程序设计语言但跟语言有关的那些手段（比如 Corba,设计模式），来进行对方案域到目标应用域的一个逻辑映射（一种能产生最终产品但并不以产生最终产品为终止条件的人类活动过程，它包括选择语言的考虑，应用设计，模型抽象，维护各个部分的整个综合过程，也即设计等于软工）。故涉及到不止程序设计语言一个方面。下面我们将就这些方面一一进行解说。

- 因为程序设计是人的活动，所以它首先是一种对人类活动本身的设计。设计模型的目的正是在于规范程序设计的各个过程。使开发过程可控化，这就是程序设计模型要解决的问题(XP 编程啊，瀑布模型啊)。

- 再次，设计需要对代码逻辑的控制，因为是由人来写代码而且也是由人来维护代码的，所以选择一种能切合人类层次又靠近语言支持的方式是必要的，过程式使人类能以发号施令的方式写计算机逻辑。面向对象力求单方面站在人类抽象的角度将计算机逻辑封装为一个一个的对象。并不需要考虑对计算机的实现过程(当然在每个 class 内部也可以有过程式)，相对面向过程来说，面向对象是更抽象的（当然，仅仅在“这个层次”上我们才能获得 00 的复用好处，实际上我们也只取它的这个好处）。当然，这些都是大的代码逻辑控制手段，和高级语法机制，小的可以小到“流程控制，循环”这样的 Core C 语法课题。

- 在语言选择方面，如果是通用编译型语言，因为类型是语言表达客观事物或客观逻辑事物的机制（使之成为语言的数据使语言成为一个 DSL），所以代码逻辑的抽象首先是一种类型抽象的过程，比如把科学计算机能用到的数值抽象为 INT，FLOAT 这样的语言 Primate 类型，进一步地，我们还可以对他们进行代码层次的抽象，C++中有 OC（面向类化），模板范型这样的对类型进行抽象的机制，如果是通用脚本解释型语言，它将类型动态化，这样就能更好地表达领域词汇，DSL 抽象。相比通用编译型语言来说，这二者都存在设计的抽象和实现（因为都存在类型），但显然脚本语言更靠近实现，，这二者的唯一的区别在于，编译型语言用了严格类型的抽象来表达 DSL 逻辑，而脚本型语言采用了活动类型的方式来抽象 DSL 逻辑。

- 程序设计还是一种对应用的抽象过程，因为任何软件的编制在于解决目标问题，

面向一个目标领域，无论是架构师还是程序员方面，都需要面对程序设计要解决的问题和领域进行抽象(当然，程序员并不考虑全局应用的抽象)，，提供一个科学组织的抽象框架以更好利用复用或更下一步的抽象，软件产品的用户有三，最终用户，最终脚本用户，程序员用户。其中，2，3 是能有机会进入到程序逻辑内部的用户。如果是写代码的那个人来维护程序，实际上就是下一步的抽象，下一步的抽象和复用不过一个词说法的二个方面。

设计并不仅仅面向于创新,有时是形式的重组,而不是内容的创新,应用形式的改观,设计出的产品,要源端是面向人的,因此要提供足够简单的使用和访问形式,在目标端是要达到足够丰富的应用逻辑(比如 XML 统一文档交换,但可由 node,root 这些形式导致足够丰富的深层功能,深层这里是指上层),因此越复杂越大而全越好(但是如果没有足够人力,我们应考虑设计出别人想不到的商机),应用形式和应用逻辑作为设计中应主要考虑到的问题,有很多人想到用一种形式来统一既存的东西,比如在 jvm 搭建 Windows 和其它一切软件,然而这是多么无聊的事啊？

明白了原理跟实践,细节与思想之间的关系,我们就会有选择性地去学习细节,,一定要研究细节，因为细节是重要的，你得用这种技术细节实现某个东西,不一定要研究细节，还是因为你要不要用到它，然而思想是重要的,明白了这个思想,你就明白有些知识是技术细节,有些思想却是根本,当然,没到考虑思想的水平,技术细节永远也还是重要的,不要企图去钻研一切细节,因为很无聊,除非你是在满足自己,一生学习细节,这又是多少无聊的事

明白了抽象,我们明白一个东西,应做个明确这个东西的抽象所属,即它所在的人类范式.

在设计软件时,我们主要用 UML 工具,但是这东西是静态语言用的

6.6 什么是设计

设计即逻辑的逻辑，而且是面向人的控制逻辑的抽象即设计的第四种意思是让抽象以人.需要的方式进行组织，即人影响抽象的能力，

实际上设计无所不在，从你写第一行控制流程代码开始，你就在设计了，只不过是隐式的，机器并不会流程控制，是语言赋于你能以人类看得明白的流程设计逻辑向机器发令的。如果说流程控制设计是细小的，那么 OO 是一种显式化的设计，当你用 OO 来写代码时你就在设计，虽然你写的是实现，设计与实现间无明显分界，，，你照样是在设计，因为你用到了 OO 的三重思想，继承，泛化，封装，，这就是控制逻辑的逻辑，，是设计因素。。一句话，设计无所在不在，即使在 C 那样的紧实现的语言中也存在设计。。

上面谈到的设计是代码抽象，代码结构，也是设计的一种。

但其实泛义的设计就是软工，我们上面说到设计首先是一种对应用的抽象过程，然而，虽然抽象过程并不需要预先考虑软件复用能力，但是科学组织了的，松偶合，紧实现的抽象过程显然可以为整个设计中的所有阶段带来好处。

综上所述，无论如何，设计的最终目的是使软件产品具有更强大的生命力，对其内要做到科学抽象和组织，对其外要做到利用复用，易于重构。

设计跟编码之间无法精确定界，因为设计可以仅仅是提出一个用于实际编码所用的架构，也可以是考虑了所有编码细节的详尽设计，设计的最终目标可以是一张 UML 图，也可以仅仅是一张草图或一堆小卡片(Wildcard 说法即来源于此,当然，并不一定要求设计要成档，但是将它具现化表达出来还是很好的行为)，然而归跟结底，我们设计最终是想产生一堆有机的类文件(UML 图也是，卡片也是)，也即我们在进行设计时，我们的目标(Dest)是产生“编程领域对于现实事物或问题抽象的 OO 解(OO 解只是解空间的其中一种而已,单纯使用 OO 的三重机制这只是初级和简单的范型，但已经是非常强大的范型，它产生的类文件已经可以实现出一个很大的架构，然而，结合了模板和策略的范型就更加是功能强大的范型了,可以更为灵活地产生出一个架构)”，这往往就是一系列的 Class 文件，而设计的源(Src)则是“经过了编程抽象的现实事物或问题”，设计模式就是用来描述这个过程的，然而它又跟算法不同，算法体现的是大领域内对于某个事物的解决方法(这就是为什么算法也可以用计算机来表达的原因所在)

什么是重构呢，重构并非重写，重构是面向为已有系统增加新功能或完善已有功能的需求下，在原有系统已经存在的情况下，变更其设计方式（注意设计这个词），修补式地加进一些新的设计元素或去掉一些旧的设计元素(故往往用到设计模式，因为设计模式可以很好解决此类问题，所以在介绍设计模式的一本书里，总是会有这样的问题和需求发生在先，然而才导出一个个的设计模式)，重构往往是解决代码维护问题的一个重要方面，当然，正如上面提到的，软件产品在最初被设计和产生时也要考虑到一个科学设计的过程，这样才有利于后来产生新需求和新问题的重构过程。

这里首先有一个实现与抽象的区别所在(人们往往把设计和实现对立而模糊了对实现的理解，其实，抽象和实现才是一对对立体，找准了这个才能正确地理解实现，就像对脚本语言的理解，如果你一开始知道它是系统编程语言的对立体，那么你就不会产生脚本语言是不是一定是解释语言这样的疑问，因为它一开始是非系统编程语言，其使用解释或编译是它第二个要解决的问题所在)，但设计和实现无法分界，这首先是因为语言没能为它们提供一个有效区分的机制(Delphi 的单元文件 pas 中有接口和实现这样的关键字，C++默认将 h 文件和 cpp 文件分开，然而这都不是严格的抽象与实现分开的机制和行之有效的方法，这根本是因为这二者根本就是具体问题具体分析的事，无法在形式上进行区别，但提供 h,cpp 这样的机制也并非错误无效的方法---请参照选读中的“形式主义”)，OC 可用于设计中的抽象，也可以用于在先前设计抽象基础上进行的下一层抽象（实现）。

那么是不是可以将抽象分层呢，一部分是 DSL 类，一部分是字串这样的计算机实现类，将前部分看成设计，后一部分看成实现，，但实际上这样也是不行的。因为无法精确给每一种这样的抽象分层，而且即使分层，也是有很大局限性的。

比如在库的设计和利用库进行实现这二个过程中，都可以写函数（以过程范式编码），显然地，库的设计是高度面向复用的，而实现是一种函数调用，然而为什么库的设计中的函数才是设计，而实现中包括了函数调用的函数体就不是设计呢？他们同样是有函数原型的接口的啊。这就说明了语言本身并不能严格化这个区别。能严格化的就是具体问题中的具体区别方法。

这就说明，存在接口的地方就存在设计（因为可供再被可复用，而设计的一个意思就是为二层用户提供复用逻辑考虑的过程，比如接口，我们在比较大的层次举个例子吧，比如用 C 实现的工具库和抽象库）。我们应该严格在源程序中标志出哪些接口是作为最终工具使用的接口，哪些是能再供抽象使用的接口。

因为抽象最终要形成源程序，要在源程序中反应出来，而源程序是由自己或别人查看的，所以说抽象对设计是至关重要的，因为抽象是择其事物一方面或某维度进行抽象，所以设计不必大而全，那些“理想”靠近现实事物模型的抽象反而不是最好的抽象（比如严格用 00 设计一个游戏的图形世界，把每一个树叶都封装为对象，还把导演，这样的概念抽象出来），因为这样的抽象往往太大了，或不必要带了太多错误的臆想。抽象只能是现实事物的一个或某些方面的变形。甚至带有逻辑模型抽象，而不全是实体抽象。

6.7 C++中用于设计的语法机制和库逻辑

所以存在三种设计方式在 C++中，OO，模板和 LOKI。

模板的能力在于抽象类型，所以设计的一种意思是类型控制和抽象能力（动态类型更侧重实现谈化设计所以它首先去掉强类型，而提供弱类型或无类型或动态类型），模板只能只做类型控制，然而 LOKI 可以做到控制逻辑级。这就是单纯的模板用于设计和 LOKI 用于设计时的区别。。

设计的唯一特征是站在高阶的泛，这就成为区别实现与设计的根本，而不是大小问题的逻辑包含互饰关系（因为这样的话，设计与实现只有模糊分界），而泛成为区别这二者清楚的界限。

为什么 template 仅有 type as type 一项就足于跟 OO 匹敌呢，因为类型就是语言的一切，提供关于类型的机制就是一门语言的设计能力的一切。。

动态类型语言也有类型，，不过它的类型不需要一个设计期的 type 来确定指定而已，，其本质也是某类型的某个 value.

而动态类型没有 type，所以没有对数据的泛化，也即无法抽象数据，也就无法抽象代码和控制（设计里的）逻辑，所以无法设计（设计的一种意思是类型控制和抽象能力，，动态类型语言不存在对类型的抽象，所以至少这种意思的设计在动态类型语言是不存在，因为动态类型是 DSL 语言不是通用语言无需考虑类型及类型抽象上的设计）。它只能在线控制逻辑的运作。

所以动态类型语言是实现语言。。应该称动态类型语言为在线类型语言。模板是离线类型语言。

我们知道设计是复用导向的，最终用户（领域内用户，脚本用户）和程序员用户（本语言内的复用,语言非通用的）是二级不同的面向级，前者用动态实现语言即脚本语言，而后者用静态语言为宜。

解决可复用银弹问题的根本可以改造开发模型，，也可以将设计逻辑尽量多地沿伸到实现，而且是脚本用户级。因为此时没有编程工作，只有配置工作，银弹问题根本不存在

6.8 设计能力和程序员能力模型

设计能力第一个是结合了数据抽象和代码抽象以及语言映射能力到现实问题的解法的能力，第二个是构架架构的能力（提出一种规范一种抽象模型，就像 Java 规范和七层模型一样，相比第一种设计能力来说，这是高级的能力）。

设计还有一个方面是面向复用的工程设计。就是开发库的设计意义所在，

理解设计对理解 C++新思维那样书里的观点有效。

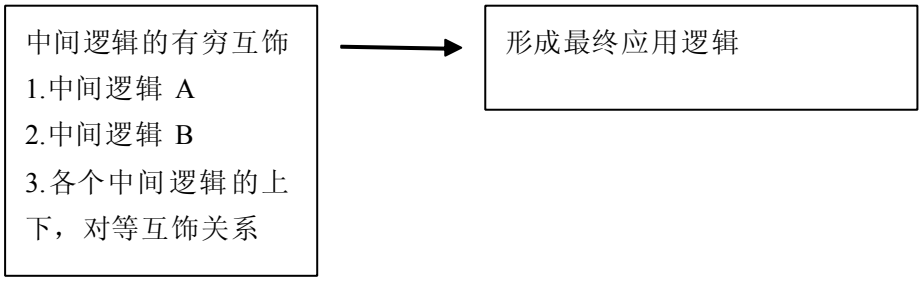
冯氏模型的局限和优点同时体现在数据结构和设计模式上，它使人无论如何都要最小在这二种子抽象上工作，，优点是，统一了开发抽象，，这也使软件逻辑变得统一。这对软工是尤为重要的。

6.9 设计方法论

“在原语空间内进行设计”前应该是现实世界问题域，在“产生最终类文件”后应该是编程域的 OO 解(或其它范型解)，即设计的 FrontEnd 面向的是用户，设计的 BackEnd

面向的最终是计算机，这中间的“原语空间设计”，“多范型表达设计”，“ROSE 工具实作范型”都是“设计演化”，即设计是一种从用户到机器的抽象过程（它包括前面提到的三个主要过程,原语设计是从上到下，泛型设计是从下到上,明白这个道理有什么用呢？这至少可以解释为什么好的架构可以扩展出足够丰富的未端实现，因为从架构设计到功能实现是互通的，这二者不是矛盾的相反是统一的，明白这个道理还有什么用呢，这也可以解释为什么 Yake 的 BaseGeneric 不是包含架构逻辑的 Generic,而是一组与平台 native 本地有关的 DLL 引入逻辑，数学函数，Logging 机制什么的，这是因为需要先提供这样一些实现，才能独立平台，而这是 Yake 首先要解决的问题，因此只能把这层逻辑放到最低层再慢慢发展其它抽象，另外，YAKE 使用的库中，比如 OGRE 和 ODE 就用到了数学函数（并非所有的问题都能靠提供架构和中间抽象来达成并解决，因为有些问题不是要不要封装和不封装的问题，而是能不能实现的问题(能不能用 OO 来表达跟能不能在算法等级实现是二个不同领域的问题，一个是软件的设计，一个是算法，前面提到了这二者之间的区别)，比如一种算法，什么是架构什么是实现，这里是一个很好的区分例子），底层必须先解决并提供这些抽象，以我来看，Yake 真正的主体不是 BaseGeneric,它是基础而不是主体，它的主体是对其它库的引入逻辑，这才是 Yake 的架构逻辑,因此说，架构逻辑有时仅仅是被体现，而没有并封装成 DLL，也即，架构是否被体现与它是不是要被封装成为一个库是没有必然关系的，哪些没有表现为库的中间逻辑也可以是架构逻辑，明白这个还有什么用呢？原语设计是不受限的面向用户的设计，然而当进入多范型设计时慢慢转入用计算机的观点来看待设计，因此像 Yake 这种与平台息息相关的表现逻辑必须在底层就解决平台抽象和数学抽象，而 LogicGeneric 就根本不用考虑这些，因此多范型设计相比原语设计来说，它是从下到上的)，而实现是一种从机器到用户的过程(实际上我在这过多强调设计与实现的差别是不对的，因为这二者无法精确定界，然而如果所有中间逻辑都被封装为库，这二者差别就很明显，库作为中间逻辑可以参与进来以缩小这二个过程差距，把应用架构称为设计，把中间逻辑封装为库(架构也可以表现为库)，实际上在这里，中间逻辑与最终实现才是对立的说法的二方，设计与编码才是另外二个对立物(设计就是原语设计而编码就是多范型设计)，

设计演化（从问题到类)，，实现演化（从类到问题)，，前者是从人到机器，后者是从机器到人



逻辑互饰构成的逻辑的巨集组合，就是一个越来越接近应用总逻辑的大逻辑，(上下互饰

就是谁更接近应用逻辑的道理,至于最终应用逻辑前面的逻辑，都可称为相对的中间逻辑)

然而设计与中间逻辑不是没有关系，把库外的末端逻辑称为实现，在这种说法下，基于库之上的实现跟设计共享同一些中间逻辑，库使这二者有机结合不产生缝隙，当然作为泛义的库是缩小任何二个架构之间差距的机制)，其实在“ROSE 工具实作范型”之后还存在一个“设计载体与设计方法”，即 UML 图，或卡片啊(设计载体)，设计方法主要是“找事物的共同点与不同点(就是多范型设计那本书的作者提到的)”，还有就是 UML 教学中出现的“给出一个句子，找出主语谓语等”(其实这些方法归纳开来就是做列举题和判断题，列举出一些细节，再判断它应属于那个接口中,这在第四部分“确定 GameGeneric 应提供什么样的高阶接口”那一节有清析的讲解)

实际上我的思想和说法比他们还要超前和规范一点，原语设计三种思想（抽象，原语，组合）就包括上述的说法(找事物的不变点就是指抽象出事物的本质,这是设计过程中一个很重要的能力)

在观察者模式中，

6.10 自上而下设计和自下而上设计

基本概念

抽象在软件开发中的重要性是不言而喻的。如果一个系统有了正确的抽象，那么这个系统就更容易理解，更容易维护，开发起来也更为高效，最为重要的是也更容易把事情作对。Grady Booch 甚至认为抽象是应对软件复杂性最为有效的手段。在面临一个复杂的系统时，往往只要再提升一层抽象层次（当然要是正确的抽象），那么该系统就会立即变得清晰、简单，理解、开发、维护起来也更为容易一些。不过，值得注意的是，虽然有了一个正确的抽象后，可以大大降低理解、开发和维护的难度，但是要想得到一个正确、合适的抽象却是非常困难的。

提起代码自动生成，可能大多数人立即会想到这样一种情形：只要获取了系统的需求，那么就会自动地从这些需求生成系统的代码。这种想法固然很好，但是在目前的科学发展水平（这种技术不单是软件技术的问题，它还和人思维的生物基础有密切关系）下却还无法实现。需求和能够自动转化为代码的精确的形式化规范之间有一个巨大的鸿沟，目前这项转换工作还只能由人来独立地完成。因此，这种技术在目前来说只能是一个神话。我们在本文中所指的代码自动生成是针对比较局限的领域而言的，即需求已经被正

确的理解并由人转化为解决方案领域中的抽象模型。代码自动生成并不是为了替代人去完成系统的软件开发的，它只是一种支援抽象的工具而已。

领域特定语言（DSL）

其实，大家在每天的软件开发中都在不经意的使用着这项工具。当我们在使用面向对象语言进行软件开发时，其实我们是在一种抽象的层面上进行工作。有了这层抽象，我们所编写的软件就会更有表现力，更为简洁。比如：当我们写下下面的代码行时：class Cat extends Animal。我们想表达的是 Cat is a Animal，面向对象语言为我们提供了强有力的支持。而这个抽象的实现细节则由编译器来完成，我们无需关心。这样我们就能够在一个更高、更直接、更易于理解抽象的层面进行开发和交流，同时也大大降低了出现错误的机会。当我们使用支持泛型或者 AOP 的语言进行软件开发时，其实我们是在另外一种抽象层面上工作。比如：当我们写下如下代码时：

```
template <typename T >

T Add(T, T)
```

我们想表达的是求两个类型为 T 的变量之和，而不管 T 到底是什么具体类型。想想看，如果语言不支持这种表达规范，我们要写多少个雷同的 Add 方法。有了这种表达规范，我们就可以直接、简洁地表达出我们的意图，而具体的转换工作就有编译器代劳了。还有，如果我们在支持 AOP 的环境中进行软件开发，那么我们只要使用该环境提供的 AOP 语言规范定义出我们希望的横切关系（其实就是一种抽象），剩余代码的编写和插入工作就由该环境帮我们自动完成了。虽然编译器或者开发环境在底层生成的实际上也是大量重复的代码，但是这些代码的抽象规范却只有一份，而人们开发、维护、沟通所基于的正是这唯一的一份抽象规范，底层的重复实现细节对开发者来说是不可见的，并且是自动和抽象规范保持一致的。可以说，在开发者的头脑中，是没有重复的。从而有力的支持了“once and only once”和 DRY 原则。试想，如果语言种没有提供这种描述规范，那么我们要编写多少晦涩、难懂、重复的代码才能描绘我们想要表达的概念。

上面提到的抽象是一些比较通用的机制，因此一般都是语言内置支持的。也正是其通用性使其有效性范围受到了限制。一般来说，上面的一些抽象机制是仅仅针对开发者群体而言的，并且使用这些抽象机制进行的表达也是由编译器来自动生成底层执行代码的。但是，还有一种抽象表达更为重要，它的作用是在开发者和客户之间进行沟通、交流。说它重要是因为它和所要开发的系统是否能够真正满足客户需要密切相关。这种抽象表达更贴近具体的问题领域，因此也称为领域相关语言（Domain-Specific

Language（DSL））。比如，如果我们开发的是一个金融系统，那么如果我们能够使用一套金融术语及其关系来刻画金融领域中的一些业务逻辑，那么不但表达起来会简洁、直接得多，更重要的是客户也更容易理解，和客户沟通起来也更为容易。再如，如果我们要开发一个科学计算系统，那么如果我们拥有了一套描述科学计算领域的词汇，那么在表达该系统时不但会容易、自然很多，而且也更加高效。有了这套 DSL 之后，剩下的工作就是要自己实现一个编译/解释器，来把 DSL 自动生成为目标语言。由于这个 DSL 一般都局限于某个特定领域，因此其编译/解释器实现起来也不会有多大困难。

敏锐的读者一定会发现，我们在前面列举的支持面向对象（OO）、泛型（GP）或者面向方面（AOP）的语言，其实也是 DSL 的一种。只不过它们所针对的是更为通用的，和软件要面临的实际问题领域无关的领域。它们的作用是为了提高一些通用问题描述的抽象层次，并且也为构建更贴近问题领域的抽象提供了基础。

自上而下 还是 自下而上？

写到这里我突然想起一个非常有趣的问题，那就是大家常常争论的自上而下开发和自下而上开发。一般认为，自上而下的开发方法更具目的性一些，也更为自然一些。其实自上而下的方法是有很大风险的，一是往往很多预先的设想很可能本身就是错的，一是这些预先设想落实到底层时要么无法实现，要么无法很好地适配。其结果就是生成一个具有大量冗余、丑陋粘合层代码的系统。而采用 DSL 思想的自下而上方法则具有很多你可能没有想到的好处。你可以先实现一些 DSL 中的单个单词，然后再实现一些更大的单元，并试着把这些元素组合为更大的领域逻辑，按照这种方法实现起来的系统往往更加简洁、清楚、干净，甚至更加易于重用。一般来说，如果你想采用 DSL 的开发方式，动态语言大有用武之地（Python、Ruby 都是不错的选择，在 www.martinfowler.com/bliki 中，Martin Fowler 对动态语言和 DSL 的关系进行了深入、有趣的描述）。

自下而上的做法实际上是在改变、扩充开发语言，使其适合于所面临的问题领域。当你进行软件系统的开发时，你不仅仅只是把你所构思的程序直接映射到实现语言，同时你还不断对语言进行增强，使其更加贴近你所构思的程序，并且表达起来能够更加简单、更加直接。比如：你会想语言中如果有了这个操作符，表达起来就更加清楚、简洁了，那么你就去构建它。这样，语言和程序就会一同演化，直到二者能够形成一种完美的匹配关系。最后，你的程序看起来就会像是用专门为它设计的语言开发的。而当语言和程序能够很好地相互适合时，所编写的代码也就会更清晰、更少、更有效。

值得注意的是，自下而上设计相对于自上而下设计来说，并不意味着用不同的顺序来编写同样的程序。当采用自下而上设计时，常常会得到一个和自上而下设计不同的程序。所得到的不会是一个单一的单片机（monolithic）程序，而是一个具有更多抽象操作符的更“大”的语言和一个用该语言编写的更“小”的程序。此外，这个语言的抽象操作符也很容易在其他的类似的程序中得以重用，所编写程序也更加易读、更加易于理解。

还有一点值得提出，那就是自下而上的开发方法可以尽快地得到来自代码的反馈，可以及时进行重构。我们大家可能都已经知道，设计模式一般是重构的目标，这里我想特别指出的是：DSL 往往也是很好的重构目标。

抽象、库和 DSL

C++之父 Bjarne Stroustrup 经常强调的“用库来扩充语言，用库来进行思考”（http://www.artima.com/intv/elegance.html 有近期对 Bjarne Stroustrup 的采访，他再次强调了这个问题），其实就是在强调 DSL 以及自下向上开发的重要性。库就是 DSL 的一种形式，Bjarne Stroustrup 所列举的科学计算库的例子就是科学计算领域的 DSL。构建库的过程其实就是在朝着更加贴近问题领域抽象的方向迈进。明白了这一点，我们就不难理解 Bjarne Stroustrup 一直强调的泛型在构建一个优雅、高效库的方面的重要性了，因为泛型为构建这种抽象提供了一个坚实的基础。

不仅 C++如此，其他一些语言（比如：Java）也拥有众多的库。这些库在我们进行软件开发时，为我们提供了强大的支持。不过，这些库往往都只实现了它所针对领域的一些通用基础性的问题。因此，在某些特定问题上，可能无法直接地使用这些库来进行表达。此时，你就应该考虑建立一个特定于自己问题的特定库了。

结论

作为开发者的我们，该如何做呢？正如本文开始所说的，抽象固然很好，但是要找出适合于自己手边问题的抽象是一件很困难的事。它应该是一个不断从代码编写中得到反馈，然后再修正，接着再编写代码的循环反馈的过程，这也是我为何认为自下而上开发更为有效的原因。我们不应该局限于仅仅会使用某一种工具，某一种开发环境，而应该多想想这些工具、开发环境背后蕴涵的思想。我们应该注重于培养自己发掘抽象的能力，

这方面能力的培养既需要很好的和客户沟通的能力，同时还需要有坚实、高超的软件能力。

此外，近期热的不能再热的技术 MDA，其核心思想其实就是 DSL，不过我对于其试图使用一种语言来刻画所有领域模型的做法并不看好。与其使用它，倒不如逐步演化出一个特定于自己问题领域的 Mini DSL 和一个 DSL 编译/解释器，并基于这个 DSL 来进行开发，这样或许更为直接一些，更为轻便一些，更为清晰一些、更为有效一些，更加具有针对性一些，也更为经济一些

6.12 大中型软件和复用与逻辑达成

我们知道一个库的发布总是带一个 samples 目录，然而那是小例子，，大中型程序是怎么开发出来的呢，，你该如何着手写一个大型的程序呢？一般来说是找轮子和写实现。

如果开发中存在造轮子的工作，那么很多逻辑都在这里面，，就拿开发游戏引擎来说吧，引擎部分总是要考虑进很多逻辑，有时是复用别人的库，有时是很多预考虑的逻辑，有时是设计模式方面的逻辑，考虑 OGRE 就知道了。

就复用逻辑（比如一个使用了 OGRE 的游戏的实际代码）来说，，它往往是 OGRE 中的 samples 演示小程序，如果是一个用 yake 写成的大型游戏，那么起码可能还存在造其它轮子的过程（比如你还用到数据结构，而不想用 STL，那么你需要自己设计—这是在造轮子了，或直接复用 GLIBC），，逻辑间的大量复合，和与其它维度上逻辑的结合（比如不仅是图形方面的，还有网络方面的代码等），就会构成一个大型软件了。

然而幸好有一个 YAKE 的程序考虑进了几乎你能用到的很多库，，当然如果你想用其它的库替代，或者加进其它的库逻辑，或者自己在复用中会想到自己开发自己的库逻辑，也是完全可行的。但是我们一般不改造 YAKE（因为如果它不提供 SRC 我们就无法改造，其它很多库都是这样的）。而是拔插使用其内组件库。。

所以大中型软件的形成，，有的是库逻辑和实现逻辑，即设计能力，，有的是复用逻辑。。即编码能力。。因此那些说程序员是简单的复用员的人是可笑的，，因为一个程序员在编程中总会涉及到设计能力（即那么比如设计一种数据结构库的能力）。。。

所以什么是设计能力就出来了，它泛指一种真正的编程能力，即将现实问题映射为语言逻辑的能力，（找轮子是作设计的一个步骤，），那么设计能力要求你有什么样的能力呢，，当然，现实问题的模型你是要清楚的，而且语言逻辑也是你要清楚的，复用的库你要从语言的观点去搞清每一个接口的真正意思（无论用什么语言，，数据结构，设计模式和语

言本身这三者的逻辑是不变的，，其它的就是具体问题映射到语言的能力了），在整个设计过程中（映射中），，你必须清楚每一个步骤，，因为软件设计真的不像拼零件（强烈鄙视这种用在小打小闹模式上的接口拼起来的软件上），，大型的软件接口太多，，你必须熟悉每一个功能模块，，和预见每一个功能接口。。这种细节的无限性决定了软件的复杂性和设计实现时的清楚性，，否则就是编译不过，不成软件。。那意味着项目失败

架构师的能力就是设计能力，，，程序员的能力就是编码能力（然而在每个小问题上每个小模块上也存在设计，所以程序员也是一个设计师，不过他不直接面对复杂的大系统而已）

6.13 架构与应用

因为功能是逻辑叠成的产物,越到底层,它对高层的逻辑依赖应尽量少，因为高层往往是应用逻辑，而底层往往是功能逻辑和业务逻辑(高层和低层通过一个隔离层意义上的逻辑来达到功能上的完成但又不致于增加这二个层次上的复杂性，使得在这二个层上的工作可以分别完成)，因此底层核心架构应尽量简小精短（比如 linux 的 core，但其中的应用可以无边无际），但是简小的同时却要提供最大程序上，或全部程序上的可扩展性，这样在完成了整个 core 后，后面的功能慢慢趋向应用时，就可以不致于改变到 core,也即这个 core 可适应一切小，中，型的平台，不必做重复工作．或者即使需要被改变时，涉及到这个 core 的修改量也尽量小．
对底层的逻辑应尽量提供迂回用的接口,迂回即抽象,增加了迂回即增加了另种一种维度上的抽象可能性,而维度永远无穷无尽,因此抽象以任何一种姿态被创建都可以是一种新的抽象,我们用代码的形式表达我们所需要的功能,即用计算机能处理代码的本质维度摸拟人的思想功用的维度,这就是"原语设计"的概念所在
逻辑的抽象粒度永远有它关于具体的复杂性，我们不能定义一个具体的逻辑块，指明它为 core 或者还是应属于 core，但是正如这句话说过的，问题有它自身的复杂性，因此我们应具体问题具体分析
这是哲学所指明的，哲学不是人的东西（虽然它是人发现的），？
底层完成了（底层往往是一些首先要解决的逻辑或者在设计意义上具有优先产生其它事物的概念体），其上的应用可以无边无际，但应用应尽量追求形式简单，一个合理的架构和其上发展的应用对于人来说应越来越简单,,,这才是合理的,我们不能掌握的复杂度应尽量隔离.因此我选用没有历史复杂性的 open jdk,gnu c,拥护开源的 linux

6.14 抽象与接口

多范型开发让你在高于面向对象的范畴里看待软件工程,所以真正的编程学习过程是要明白设计在先，这是基础的思想，然后去学习编码，，（以上二点都是方案领域）然后再去研究编程领域对现实问题的解法（这就是应用领域）

而且如果你知道可复用和可扩展对于软件工业来说是多么重要的一件事情，，，你就会知道面向对象是多么好的一种机制了(面向对象和面向构件，面向构件可以不用对象的方法，，但是很明显面向对象和面向构件都有一个共同点那就是它们都提供了可扩展，而这就促进了可复用)

数据接口是另一个很重要的概念

接口实际上就是将要用的部分作为某个接口提取出来(就是定制要使用的实现的某个子集而已)，而所有实现就是接口下的所有代码,即数据本身，故其实数据与数据接口是分离的

设计模式不致于使我们声明对象的过程变为硬编码，，，这样就使得整个软件的对象生成是动态的，这个过程就如同动态分配内存，寻求一种好的方式来组织类的过程称为设计模式或策略

接口这个概念其实无比自然，无论是为了隐藏实现的安全考虑，还是为了更好地让这些实现为用户所用这个方面来说，接口都是必需的，

接口几乎改变了现今我们开发软件的方式
举个例子，在现实生活中，经理人可以卖票，，个人可以卖票，国家也可以卖票，，在面向对象的范畴里(注意这里并非指某个具体的面向对象设计)，我们一般是把国家，个人，经理人都各声明为一个对象，再为它们各自添加一个“卖票”服务，，但是事实上，我们需要的仅仅是卖票这个服务，(如果这个服务被包含在一个发布的第三方代码库中为我们所用)，我们只需提取这个可用部分，，而不是要知道提供这个服务的各个提供者的细节(还好上面只是列举了三个对象提供了这个服务，，然而现实生活是复杂的，还存在成千上万个对象也可以提供这个服务)，，因此，对于复用者我们来说，我们需要的仅仅是卖票这个服务，而不需要知道卖票服务这个背后的情况(而且，对于发布这个第三方代码库的第三方来说，这可以更有效地隐藏它的实现细节)

接口就是把我们所需要的对象部分封装起来提供给我们，而把我们压根不需要知道的关于对象的细节隐藏，，，也就是说，，接口是关于一个对象如何能被使用的形式的封装者(一个对象可以有多种形式被使用，因此可以一个对象提供多个接口)，是真正的对象(实现)和外界复用的包装器和桥梁，也称适配器(即接口就是直接面向使用的中间件,或封装了给二个不同架构提供适配作用使它们能协作运转的中间逻辑)

应该如何设计接口呢,OO=抽象+接口,,那么其体现又是怎么样的??只有深克地分析了应用和计算机本身，你才能分析出通用和最精简的接口机制,而 OO 正是这样一种机制..因为类=数据加函数,,数据加函数既对应计算机世界,又是解释现实事物的一种机制 ...当然.OO 不单是这种理念,还要学习它的运行期多态,继承等(语言语义)..

学高级语言应不仅学它语法,而且要学它语义..因为被翻译后的代码才是代码最初被产生的地方,是最应该被研究的..

接口应尽量窄,提供尽可能少的东西,,但在概念上一定要广,,以备后来的扩展需要.这样的接口,对于复用来说,也是可以花最少的精力来学习的.什么是抽象呢,,,就是抽取对于人来说象的部分....把多种特征的东西在概念上整合为一个认识,,,抽象出接口就是这个意思...其实,抽象正是为了简单,,而不是复杂化(它只是极力把复杂化的低层细节隐藏,透露给人们一个通用的,可认识的抽象概念,即接口)

(一个接口应尽量简单，提供它自己应提供的逻辑，其它多的都是多余)，这些逻辑都是基本的，比如一个 socket 库只需提供诸如 transport(int),transport(float)这样的接口，，那么所有的逻辑都可以建立在它上面了(比如发送大件数据的后来逻辑只需建立在 transport()这样的函数基础上就行了)，

deepest concept,lowest interface = min learning

对接口编程即，将需要维持不变的东西，即设计，维护在接口层，即变化的东西，实现，维持在接口层以外的地方。

如何从以上二种机制理解 OO 呢,,,OO 为什么这么流行,,因为 OO“很好很强大”..

6.15 构件与接口，软工

UML,IDL,MDA,COM.CORBA,WEBSERVEICE,XML 软工新潮流
关于接口,也发展出一种语言叫 IDL,,语言跟应用的关系是什么呢??因为语言可以接上计算机处理跟人们的应用需要之间的接口,,所以语言和编写语言下的程序成为扩展计算机也扩展自己的手段,,甚至还有 DSL 为了解决特定问题而产生的一种语言,语言的实现即编译器,语言规范等,,
而 MDA 是不同的概念,MDA 是给定一个领域的描述,,然后写出一个依赖于接口的 xml 格式的 web services.
这直接促成了构件的产生,,,在 SOA 中,因为要集成新老系统,当构件作为一种比对象还要大的软工逻辑粒度时,,它们共同需要都需要接口,,但是构件
不要小看了这个构件？？它几乎可以是一场软工变革，跟 OO 有相平之处

这个道理就像接口和抽象、、

产生了抽象，提供了接口，但是接口却是大口径的，它隐藏了实现，接口并非刻意去隐藏实现，而是这是它的性质决定的，客户程序不能通过接口去访问接口后面的实现，否

则只有它是这个接口的客户，，它就至少无法通过接口向前访问，，它只能利用接口逻辑，去发展后面的东西。。

6.16 真正的 interface

正如对象的索引才是真正的对象一样(C++中提倡使用引用优于指针)，对接口编程才是真正的编程工作，一个程序员大部分情况下只是一个接口粘合者(因为我们不需要重新发明轮子，发明轮子-中间逻辑的工作才是真正的对实现进行编程),发布时我们也是发布接口库和其说明文档，大多数情况下,我们都是利用第三方代码库编程（这是语言之外的接口，程序员也在语言内部编制函数等接口）

接口设计是一个编程工作中常常要考虑到的问题,要考虑提供这个接口的实现会在什么地方会用到(因此它跟需求分析密切相关)，以此来设计接口的参数信息,一个接口不单单是一个函数，虽然函数的声明部分在大部分意义下作为接口的意义,,,,

所以 delphi 的单元中有实现和接口这二个字 ,,,接口的集大成者是 COM，所以 borland 以它的 IDE 很好地支持接口而著名
这是语言之外的，

设计一个面向需要程序设计语言永远是不可行的，需要永远是可变的，程序设计语言只能遵守一个固有模式而提出，有固有模式去表达和创立新东西(面向对象就很不错)，编程是人的动作，人力对工程（一个大软件就是一个工程）的控制应该尽量简化，

特别是要掌握对象(或称工件与产品)与接口-组件（对象接口与组件接口是不一样的）概念所在，前者是语言内部的，后者是语言外部的
在内部
纯数据对象称为死对象,(可用但无用)纯实现对象称为工具对象，不可用
在外部
纯接口对象称为抽象对象,(可用但无用)纯实现对象称为工具对象，不可用
面向接口，(一个一个的接口，函数称为函数接口)接口归接口，实现归实现,实现不是函数
软件就是包装器,就是一大堆接口的有机组合,不提供继承机制

一个类型的数据可以独立构成一个数据结构
对数据结构的描述包括它的类型，它的结构，它的存取规则

软件设计是一种什么样的过程？
软件就是对象组合,这些对象通过他们的接口按一定逻辑组合成可工作实体
在语言内部的继承是对象，在语言外部的继承是组件

对象的概念，包括函数对象，变量对象，数据也是对象(作为一个元出现在一个特定的数据结构里)，因此数据结构也是对象，操作实现也是对象，实现称为处理器，数据或实现的组合也可称为对象,注意，对象的组合只能是被称为对象组合，只有那些能在一起工作的对象组合(compent view)才能称为软件，这里引入工作逻辑的概念(就是 ROSE 中的 logic view)

在内部
纯数据对象称为死对象,(可用但无用)纯实现对象称为工具对象，不可用
在外部
纯接口对象称为抽象对象,(可用但无用)纯实现对象称为工具对象，不能由它派生出实例

面向接口，(一个一个的接口，函数称为函数接口)接口归接口，实现归实现,实现不是函数
软件就是包装器,就是一大堆接口的有机组合,不提供继承机制

一个类型的数据可以独立构成一个数据结构
对数据结构的描述包括它的类型，它的结构，它的存取规则

数据是对象，逻辑也是对象，操作也是对象，一切皆对象的概念，在机器内部一切皆比特，在用户眼中，一切皆对象，因此数据库的数据二字是有通用意义的，，，因此会有面向对象的数据库

至此只是软件内部，那么在软件外部有 develpoment view

数据组合接口形成一个数据结构

对象和(包括接口和实现)构成一个组件 DLL 或 LIB,一组对象和一组接口就是一个 DLL(称为一个产品)，需要一个调用协义接口

.

6.17 真正的对接口进行编程

不存在一个“对接口编程”的真实过程(虽然这是一种过程定义)，我们只是说，要为”实现”定义一系列使用它的接口(这样实现才能更好地被使用和被修改)，，这种行为才是对接口编程行为,而接口本身是什么还是没有说哈哈

一个为未来扩展而写出的工程中，大部分代码只是框架(抽象的思想模型，也就是为了扩展需要--也是为了使用需要，而定义的一层又一层的抽象),真正的实现部分(调用 API 啊,

用某个算法啊,某个完全具体工作的实体对象，或某个完成某个业务过程的会话对象)很分散，而且分散得很有规律性(因为被抽象接口经过了再组织，所以变得有规律地分散),,这样的分散机制就像把真正的实现逼到最尾端，而最高层往往是使用这些尾端要如何被使用的应用逻辑--被抽象成了一个或某些使用的统一接口形式，而且是高级逻辑，(即接口实际上是关于如何使用这些实现的隔离层,,中间层)

这样抽象也称为为客户调用(或使用)的协议定义

很多时候，在一个工程中，所有的实现都可以由一个 Demo 直接写出来(写成 CONSOLE 形式,也可以是一些对象集上面说了)，然而,真正形成产品时，我们需要再组织这些实现，让它们最终形成的产品出现(因为一个真正的产品，必须要考虑到未来修改的需要啊)这往往是一个比写实现还要难的过程，因为我们在写“如何使用这些实现，如何把这些实现分散封装到末端”的接口逻辑，而这个逻辑，往往有时比写实现本身还要难!!

6.18 过度抽象

C 和 C++只是抽象多少的问题(对硬件架构和软件系统，构成的系统编程环境进行抽象，以靠近人脑开发)，linux 之父还跟别人争得起烟,,在 Windows 的 GDI 中,用 C 实现了一个对象系统,连 C 都抽象了桌面对象,还有什么不能抽象的,,,只不过 C++在语言级实现了一个 OO 而已（而ＷＩＮ的桌面环境对象系统是在逻辑库级），
一个问题出来了,，抽象显然是为了简单化，但是这是对系统编程环境的简单化,，在高级层次会造成更大的复杂，但至少脱离了大量需要原来人脑去维护的细节，　人们只需要掌握大概就可以去复用了(接口),,，这是一个很大的进步,,，而且有些被实现了被开发过的东西和轮子可以直接被拿来复用，C 代码却要基于平台的考虑修改大量细节，这种修改的代价超过了太大，就没有可复用性了,,，所以无论如何，C++比起 C,，使人们掌握细节和系统的机会减少了，但需要人们掌握设计和复用上的知识多了（人们更专注应用领域）. 而且在这个时代,由于 JAVA 语言这样的语言的出现,语言标准得于统一,因此可复用性首先不用考虑语言本身,更而且,JDK 统一,SUN 的 J2EE 规范统一,,可复用性就成了人们之间的契约文化了,可复用性的地址会越来越下降. C 的难点就在于系统本身,,系统细节和语言细节都过多,，造成人们掌握它的成本过大，但 C 的代码没有绕ＯＯ的弯子，跟系统逻辑几乎一致，因此效率很好。

什么是过度抽象呢？如果抽象不是提出一个简单模型，而是在封装的基础上提供了一个很复杂的模式，甚至超过了人们学习 C 和系统的那些知识，那么这种ＯＯ模型（或称呈现的供可复用架构）就是不成功的，

设计模式是 OO 范式刹车的标志，因为它拉大了计算机与要解决的问题的沟隔,怎么说呢?OO 以后的 OOP 范式太多了,有设计模式,框架等,其实计算机内部的离散形式是死的,但用编程语言解决的目标问题才是巨大的(实际上 OO 加重了编程的负担和新手的入

门难度,用 OO 来描述现实世界只能做到"对象"层次,如果向模式,向框架发展就越来越庞大和力不从心了),只能越做越复杂,有了 OO 就应该适可而止了,不要再发展 OO 以后的东西了,比如设计模式,比如框架,因为 OO 以后的东西其本质还是计算机内部的离散形式,这二者南辕北辙,造成的结果是适配这二者的负担太重了,叠在裸机上的逻辑太多了,有 OS,有虚拟机,有运行时,有 PE 载体逻辑,有解释器,这只能给机器越来越重的负担,这是第一,第二,计算机底层的离散形式本身就是固定和简单有限的,自从有了 OO之后,编程逻辑发展为远离计算机底层(向人靠近)的高级逻辑,而现实问题用 OO 来解只会越来越复杂,不如用固定的编程范式,,比如 C 语言规范来表达,这最大的证明就是单例,RUBY 可以用极为简单的形式来表达单例,而 JAVA 要用一大堆 OO 的思想,,

6.19 架构不是功能的要求，但却是工程的要求

什么是架构,架构是设计的上上层部分..比如网络七层模型,甚至 PC 的冯氏模型..这些设计的上上层部分深刻影响了计算机后来的东西.不可轻易更改.

不可思议的不是代码，不是编码，而正是人类的大脑，而正是设计,,

在设计一种平台时,语言和 OS 是一个架构的基础部分(在 OS 内部更多地存在设计)，应用再在这个上面慢慢搭建.

如果你看过 google 的手机平台设计，就会发现语言这个图灵装备往往是架构的最基础部分。如果说机器语言或汇编语言是严格基于平台逻辑的,那么高级语言基于严格的语言机制和语言语法,具有严格的图灵装备,,负责产生后来的逻辑和用户级的应用逻辑,本来 OS 是不需要语言的

语言尽可能小，直接面向真实平台不要虚拟机,因为加了虚拟机执行方式会很慢,,而且,语言应作为 OS 的内核一部分.就像 WINDOWS 把 GUI 接口加入 core 一样,这样的语言执行效益很快.这样的语言也就称为系统开发语言..与之对应的是应用开发语言,或称脚本语言.

在底层,汇编是一种几乎对应机器语言的东西,因为汇编语言无语法.只是名称指称，助记符..C 作为中间语言,汇编只需要提供接口给 C，C 还应在编译期提供一个编译期多态

为什么脚本语言多称为语言粘合剂呢..因为脚本语言往往提供对多种系统编程语言.数据类型的内存模型的封装.提供了对这些语言的多种接口,,,但更主要的,脚本语言被称为脚本语言,,,更在于它面向用户的其它特域相关和特定工作相关的语言机制(比如 Lua 的多协程常被用来游戏,,而这显然不是通用语言应在语法级应提供的)也许脚本语言只需要提供与其它语言的粘合接口就可以了

在上层,我们应该提供一种"无语法"语言的 DSL 脚本(比如 SQL,好像脚本语言跟解释型语言几乎同意),,这种语言最好还是无类型的,,这样做的目的是方便程序员,但不好用来开发平台逻辑(因为平台语言,要求严格的类型机制)

我们看到的大多数语言都是图灵完备的,而我觉得 C++应改编成一个解释执行、无语法的语言,因为应用语言往往需要做成即时修改的脚本语言

6.20 你需不需要一个库

IT 开发中,,只有属于底层开发的,,,一般才称为开发,,发明轮子,,,而复用成风的今天,JAVA 这样的语言体现的是一种高级逻辑配置式的开发(高层的编程如 web 编程，都是配置式的),当然也算开发，，所谓库，是一种面向通用和复用的中间逻辑，，接口逻辑，而非终极的应用逻辑本身，,，库面向应用复用提供接口，而应用逻辑面向应用本身．语言的功能和可复用性，，一个很重要的方面是除了语言自带库之外，还有没有第三方为它开发大量的开源库

所以,如果你不是专门为了通用的目的考虑,就根本不需要开发一个库

6.21 可复用与可移殖的区别

在相关书籍中，存在很多相似但其实有很大区别的概念，比如可移殖与可复用，接口与实现，接口与抽象，下面试区别之。

在一个大型软件系统中，抽象是分层次的，，粗略地来说，有的抽象是系统平台相关的抽象，有的是对于目标问题领域的抽象。注意这个区别只是粗略的绝不是精确的（所以也可说是无层次的）。

设计中经常将实现和抽象分开并各自集中，如果抽象中过多地混入了细节考虑（即有硬编码和实现出入的地方），那么它必将在以后的扩展过程中产生麻烦，因为对于一个庞大的软件，其内部逻辑复杂，牵一发而动全身，语言给于实现和抽象形式上的划分方法只有头文件和 CPP 文件这样的初级方法。实现和抽象的分离从来都高度掌握在源程序的作者手中，我们知道，抽象不全是为目标领域作抽象，有一部分抽象是为接口作抽象，也就是为可复用的有效形式作抽象，即接口是抽象的一个部分，是抽象的简单形态，，其目

的是为了给使用它的客户提供一个复用（或实现）的原型和规范，比如库，函数 API，这里的客户是程序员用户用户。(但是像虚函数那样的语言内接口，又不完全是为了面向人的复用，而是为了面向程序内逻辑客户的实现。这个客户跟据这个接口产生出一个关于这个接口的 model)，如果不是库，则不需要提供接口设计，而实现是未端抽象（这就要求设计者具有良好的对系统的可复用考虑的设计能力）。更多的关于接口与实现的区别在文尾有述。

在一个复杂的软件系统中，可移殖逻辑主要集中在那些与系统编程相关的逻辑中，而不是对于问题的领域设计逻辑（虽然如果对问题的领域设计，，抽象得不得体的话，这样同样会导致不可复用问题，但决不会产生不可移殖问题）。比如对某语言密切有关的字串逻辑的依赖，对某平台密切有关的某个 socket..鉴于对不可移殖问题的考虑，，我们往往将它与领域逻辑分开。所以可移殖问题只是可复用问题的一部分，二者绝不是同一意思。

所以，应该怎么样做呢？这样才能同时达到可复用，又最大程度地可移殖。(当然，只能是最大程度地这样。)

编程涉及平台支持和目标领域问题。一个用编程语言写就的，用 OS 运行的，“软件系统”中，必将大量存在这样的“平台编程逻辑实现”，相比之下领域逻辑少得多，(我们将由领域逻辑主导实现逻辑。)。
一种方法就是广为谈到的“抽象与实现分开”，我们需要将各个“实现”按文件物理地分开放置（此文件将会是引用的末端，不被“设计”直接作为头文件引用，而是作为最终可用可弃的实现末端，由它引用目标领域逻辑）。当然这个过程中，我们应注意模块化（所有的编程范式都是模块化的），然而正如上述所说，模块化不能复合，不能高下相互引用，比如“设计”引用“实现”
。（因为设计中有自上而下和自下而下，故也不存在各个模块之间平等不相互引用的情况）但是如果不考虑最终软件系统的实现的话，光就设计来说，确实也存在各个设计模块之间平等绝不相互引用的情况存在。

除了上述不可复用问题来源于不可移殖之外，还存在以下几点不可复用问题产生的源头：

- 解决不可复用问题的方法是增加迂回（一层抽象接口），将实现逼回底层，这个动作出现在二个过程中 1，对于目标领域的抽象过程中，2，对于重构时的过程中。

  然而，所谓的迂回，其实也是系统中的抽象，，也会对可复用产生障碍，一定意义来说，系统中抽象层次过多，，或数量过大，都会直接对可复用性产生麻烦，这为了解决不可复用问题而设计的另一层抽象正加大了某种程度上的不可复用性。所以是一种以毒攻毒的方法。解决问题的方法正是产生问题方法的来源。

- 不幸的是，语言机制也会造成不可复用，比如模块就没有函数来得可复用性强，然而复用从来都是相对的，存在一个比较时所采用的最小考虑单元，仅在 C++语言内部而言，模块是可复用性很高的，在所有语言面前，函数接口和过程式开发无疑是复用性最大的。。这就是 linux 之父跟别人争吵的源头所在。这也就是说，C++的抽象性能反而带来了不好的地方，越抽象的东西越会阻碍可复用性。
- 当然，最后，抽象的方法不同，产生的抽象结果不一，由此产生的不可复用问题是最严重的。因为复用者一需要理解你的设计抽象，，在理解了之后，才能进行复用。如果你的抽象过于复杂，复用者不会有太多兴趣。

编程能力就是学会如何面向可复用考虑去进行抽象。当我们设计自己的系统时，为了提高它的最大可复用性，我们将它设计为与语言无关，与 OS 无关，与复用的库无关。这种工作是相当难的。设计中的目标问题抽象永远是自己的。那么如何将这些如上的“实现”逼到末端呢？

首先，要想做到与语言无关，就要用那些最初步的语法机制和开发范式，比如函数过程式。三种控制结构（事实证明它们可以产生一切逻辑）。或者自己开发一套自己的语言，在自己的语言领域之内作“固步自封的复用”。但我们知道，这（当其它语言不存在）实际刚好阻碍了其它语言对其的复用。

其次，要做到与 OS 无关，当 OS 不存在吧，不要引用 OS 的任何东西，，照样是在自己的语言的基础上发展自己的 GUI 库，等(这实际上也是很多语言提出可移殖理念最初的出发点)，一系统的系统平台相关的库。

要做到与可复用库无关，只有自己开发功能相当的库了。即一切轮子自造，包括语言。然而你可以改造语言，却不能改造英语编程语言，那是一个未知领域，你更不可能在冯氏模型之外发展这样的语言，你不能改造 PC 模型，也不要指望改造 OS，更不要指望改造电脑的能源为光脑。

所以，一切好自为之吧。既然轮子的创造是一个无底洞，，何不直接就复用别人的库呢，用大家都用的语言呢？更重要的，作大家都在做的领域设计方案。这样别人才能理解你的设计。

==========

接口是实现的原型，，一般谈到实现，就是一个复杂系统中的终极末端逻辑，意指其不必为复用留有余地(因为复用就是利用接口作进一步的抽象，复用跟接口密切相关，所以实现也指不必为接口设计作余地)，一般谈到接口，意指为下一步的抽象提供统一的原型和形式，即为如何复用提供设计，

6.22 再谈可复用

可复用不是指修改而用，而是倾向于不用修改也可以用，

可复用问题的由来一是实现的不能可复用(还记得在 C 语言中写上大量的预处理代理来实现跨平台逻辑吗?)，二是架构逻辑的不可复用，,,，实现指的就是跟计算机离散相关的平台逻辑，，架构就是人为为程序的可扩展性加上的设计逻辑，，大多数是 OO 之后的东西，，我觉得提出二门语言，一门 C，一门类 RUBY 的脚本语言来进行程序的编写,,,,在底层用 C,在高层用 OO 脚本,这样的办法很好...因为你在低层不需要架构,,而只要在高层考虑架构问题.,,这样一来,可复用性就是二个阶段的事,前一阶段只管实现,不考虑设计,,这种过程是承接性的,只要先实现了,才能被设计得架构上更科学.

模块化也是一个重要方面，设计中的抽象过程固然要求模块化，但这决不是抽象的终点，，因为有时虽然模块化了，但是模块之间却偶合了。所以，科学的对于问题的抽象应该是模块化加松偶合的过程。

6.23 真正的可复用

可复用到底追求一种什么样的效果，又能最终达到什么样的效果？运行期的效率或重构期的不可复用和窄扩问题（这是二个并非绝对统一的东西），一切都可归究到设计期的问题。

编程界的可复用主要是面向对象和构件复用和设计模式和设计复用，库也是语言内部的可复用(就跟你拥有库的源文件一样.因为有头文件也是一样的，因为你还至少清楚库的构架,这也就跟理解并应用一个库只需了解其 API 就行了但不需要了解其 SRC 级的实现一个道理.ode 的头文件集却是一个例外),COM 的复用就是纯粹的二进制的复用，因为有真正的接口的隔离作用(此时你根本不知道库的构架)，在库定义的接口中，你必须透过接口才能深入接口更下面的逻辑(可能是另一个库的实现)，因此接口一方面提供了方便性，另一方面也增加了屏蔽性，这是一对矛盾,接口的定义是为了引入某种架构或桥接二种架构使其配合工作，而这种机制在提供了方便性的同时也增加了理解和使用该接口的复杂性和运行时空的代价。

用 OO 来表达世界的观点,,,物体可以组成世界（所有其它的东西，比如物体之间的关系也是另一种意义的物件）,,，因此编程抽象了 OO,那么编程就可以用来用计算机模拟世界，这种思想是成立的。

库的组合=功能的组合(类库设计是一种跟语言同级的设计)，当然这种逻辑在使用同一种语言下是成立的(不同语言时也可以用 Swig 等技术来改造或 Bind)，然而库作为中间逻辑的封装者(库让你跳过库的实现即中间逻辑这些细节而直接面向大逻辑大架构编程,只

要引用它们就可以在自己的程序中实现它们)，可以一直细化接近最终实现,诚然单逻辑的一个类也可以被封装为一个库但是往往不样做，一个库封装了一套互饰的中间逻辑的有机组合，,这里的中间二字是相对最后的应用逻辑来说的，往往把最终的应用逻辑称为实现，这就是一种实现逻辑了而不再是中间逻辑了(这就是说库可以是一种内含高抽象的架构逻辑或具体的工具函数的实现逻辑，或基于其它库之上的架构逻辑或实现逻辑)，库可以直接深入到实现细节，但是我们要控制这种过程，一方面是中间逻辑与最终应用逻辑不可精确定界，另一方面是因为设计与封装是个无底洞，不必做这种深入，第三方面是有其它的库可以 plug 进来然后在这些“轮子”上实现（库应只提供 BaseGeneric 这个库构架(此时库本身是一组架构逻辑而非实现集)和对一些其它外来支持库的引入接口(一般接口需实现，逻辑需继承,此时其它库可按需进行 plug in 或者 out)，这就是库引用库，这种情况下有一些未端的实现是不应该加入中间封装的,比较好的作法是用一个库定义架构和基本工具函数集，以及对其它未端工具库的接口逻辑(此时先前定义的那个库就是主库，其它的库是可选的辅库,比如 Yake 的实现就是这样），实现就是最后一节提到的几个 Demo(作为基础的逻辑已经被库封装起来，其它的就是实现了)

像 Yake,它提供了一个 Base core 和很多构架上的接口逻辑，每个接口逻辑都实现了一个对外来库的引用，Base core 是工具函数集(也有一些接口逻辑),这是 Yake 的主体部分,而接口逻辑(Yake 也在这里实现了一些工具函数库比如)和对其它库的引用逻辑(也是一些 Adapter)才是 Yake 的重要部分(Yake 包括它的 base 和对其它库的引入逻辑这二大部分,当然还有它的一些工具实现，这样算起来 Yake 有三大部分).

接口是可复用中一个很重要的概念。

6.24 真正的设计与编码

软件开发过程分设计和编码，组织这二者(以及对其它细节的工程可控性要求)的工程化方法就是 XP 啊，UPS 啊什么的，设计阶段的工作是可以涵盖整个软件开发过程的，而且可以跟编码同步(一旦已经在写代码，那么你就开始在设计了)或异步（最好还是先想好设计的每个细节）进行，

与设计模式相比，算法体现的是一种更泛化的问题解决方法的说法(或者说它更测重于说明如何实现能不能实现，而设计体现是如何设计，要设计出什么架构来表达什么逻辑达成过程)，在《领域数学》节中提到的算法是数学界对于问题的 Solution(VC7 以上的工作空间被称为 Solutiom).

设计的严格意义是广泛的，，
我们这里说的设计是指定义某种思想上的架构（软件架构往往是一种思想构架，然而必须最终在代码上体现出来，，设计模式也可称为架构模式（实际上设计模式可用在大架构或小逻辑上都可）或逻辑模式，)，然后在这种架构下编码构建应用(世俗的眼光里好像编

码的地位一直要次于设计^^)，这不是一种泛思想的活动（虽然严格意义上的设计的确是泛义的），而是面向产生类文件的设计，但我们正对事物或问题进行设计，所以在源端是非常不确知的高构，在目标端是一定要产生类文件，

因此编程界(注意这三个字)的设计有三种

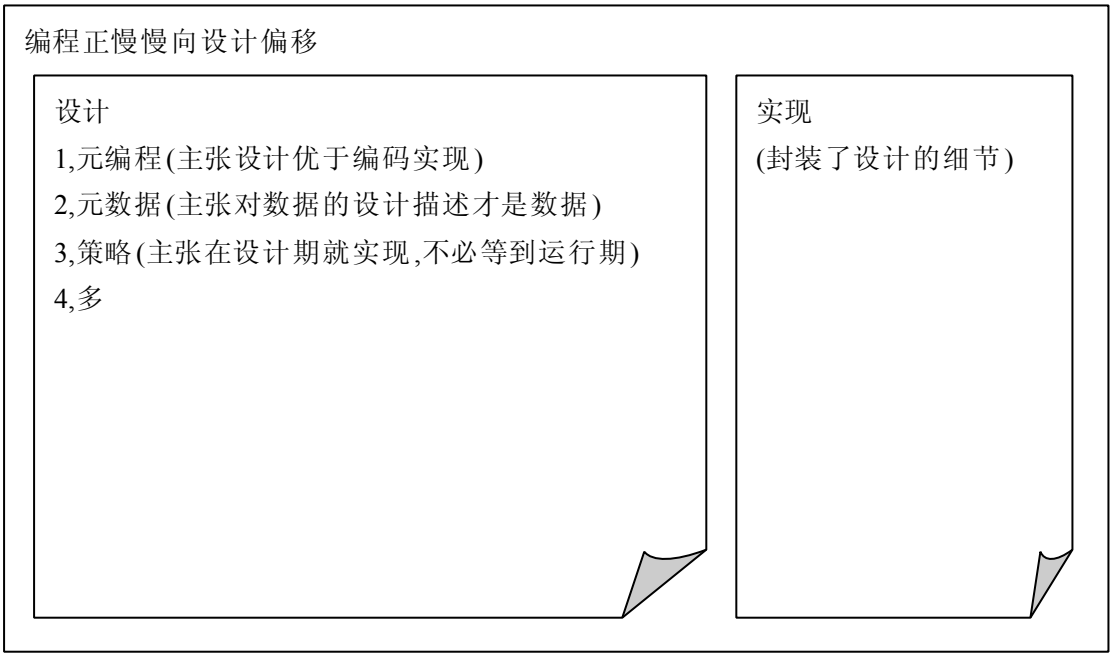
1. 基于通用设计传统的设计,比如采用一切语言级和非语言级的多范型设计
2. 一开始并不直接面向编码的原语设计，后来才转为面向编码的范型设计
2. 技术上用策略来实现的设计
3. 混合了你自己创建出来的一些架构思想的设计

用原语设计去主导你的设计，再用修饰过的原语设计去主导你的多范型设计(有效的设计应分分这二步)，最后才产生类文件。。

现在很多领域都向原语发展（人们正在更好地认识这个世界),，比如下面的一张图，左边是设计（偏重思想），右边是实现（编码），左边在不断地靠近右边

1.类是对对象的设计，,，模板是对类的设计，

很多现象表明，编程正慢慢向设计偏移，细节正慢慢向它所属的大领域靠拢



6.25 基于构件的开发

构件是运用了组合的思想之一，然而组合是一种更大的思想，有些时候，组合的个体之间并不需要接口，，只是当它们运作起来，它们便突然构成了这个璀璨的世界。就像这个世界，水，火，大海，高山并不是某个神预谋为了某种美感而创立的，神只是分别创立

它们（没有留专门的接口吧？），然而它们各自存在而已，然而突然有那么一天，神创建的人类突然发现这简单的组合也是这么美和合理。（请原谅我用了这么一个不严格的神话来描述这种思想，然而作这种泛化思维有好处的）

在编程领域，库跟构件严格上是二不同概念，然而大体上等价，这里不区分二者。

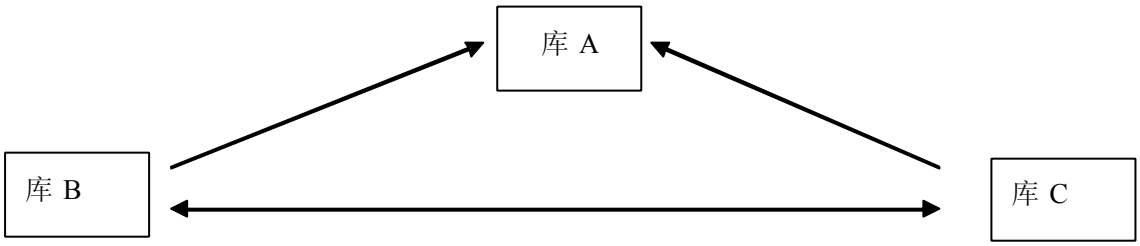
按使用形式分构件一般有五种

1. C 的库，可能是运行时库，语言函数库，用户发布的应用库，OS 级的 API 库，等等，这些库由于往往只有函数级的接口，因此只要明白参数调用或入栈规则就可以了
2. C++的库，这种库也可以是上述四种库，这些库由于是用 C++写成，可能采用了 OO，因此其内可能含有某种架构，我们必须懂得其内置的架构才能更好地使用它们。
3. 接口库，比如 COM，这种构件就是严格意义上的构件，因为它直接内置接口定义，需要懂其内置的架构才能更好使用它们，更高级的还有 DCOM 等
4. 构件套，比如 ActiveX(主要用于网络分发)
5. 类 JavaBean 直接为编程环境服务的组件。

这几种库一般都用 DLL 来实现，DLL 是一种装载机制(它可以在运行期动态装卸，比起静态库的另外一个优点来说就是可以避免双份的库引入)，而构件是构件，这种关系要明白

一个很重要的思想，千万不能让库的架构主导你的原先的架构，，你永远有你自己的大智慧（在你的应用中你有自己的架构逻辑），也不要做严格的学院派高手去研究某个库的接口实现（设计一个好的库的架构也是一项大工程），除非你是为了学习，否则不要去探究它的实现，而且库的架构不必复用到你的架构中，你永远只是库挑剔的顾客与使用者。

按组合关系来分库存在有平等关系，上下关系，多对一关系，比如有三个库



如图，库 A 与 B，C 是上下引用,也是一种一对多的引用(单向箭头)，库 B 与库 C 是独立平等的关系(可能 BC 并不引用各自的逻辑，说它们平等只是相对要引用它们的 A 来说的)

这种构架就是我们呆会在第四部分提到的总架构,库 A 是 GameGeneric,库 B 是 ShowGeneric,库 C 是 LogicGeneric,当然在每个库下面还有很多库引用关系

什么是上下关系呢？就是说谁更靠近应用谁就是上，平等关系就是它们作为中间逻辑靠

近应用的程度是一样的（这主要是因为它们为共同一个库提供逻辑，而那个库更接近应用，比如这里的 B，C 对 A 的关系）

按性质来分有架构库和工具库(库都是中间逻辑的封装者,然而这中间逻辑也有“是架构”还是“非架构”之分),非架构逻辑就是或工具函数（实现逻辑），架构逻辑就是对一种思想模型的描述，，对其它库的引用逻辑，使用逻辑（封装逻辑），通过这种方法，将利用外来库的行为封装为自己应用的架构

我们在这里反复提到架构与实现，那么到底什么是架构呢，广义的架构就是中间逻辑互饰以构成最终应用的过程中，出现的一切中间逻辑间的关系(无论这些中间逻辑有没有被封装成为库)这就是架构，，，狭义的架构就是指反映设计中某种物体模型的逻辑层次(比如我提到的 GameGeneric 由表现和世界逻辑组成云云)

比如上图中，中间逻辑的有穷互饰就会形成最终应用逻辑，但是中间逻辑这个说法永远都是相对（于最终应用逻辑）说法，越靠近最终应用的中间逻辑越是高级逻辑

要知道，，抽象达成到最终应用中，，需要设计的接口（这里主要指函数级的）精度和量度都是巨大的（可以用单逻辑的类文件，也可用库来集成一些逻辑作为二进制的中间逻辑，但是如果是使用别人的库，，那么往往需要作对它的引入逻辑，也即对这个库的使用逻辑，，也即你自己的接口才是你的应用主体，别人的只是拿来用的（并形成你的应用的接口），，比如 Yake 对其它的库的引入逻辑），为了未来的更好复用性考虑，越远离应用逻辑的接口，，对它的设计应尽量考虑接口上的放大化

GameGeneric 向你透露面向游戏和脚本级开发者的那些接口(包括它自己的一些逻辑，和一些对库 B，C 的封装逻辑也即入接口逻辑，库 A 向用户封装了 B，C 的细节而提供一个大接口给用户使用，当然库 A 可能也有它自己的逻辑和架构),这些接口是直接面向高阶应用和开发的需要而创立的，既然 GameGeneric 建立在 B，C 之上，那么在引用 A 的时候(利用 A 进行编码实现或在其上面架构更高层逻辑的时候)，B，C 是不是变得无可访问呢？不是，我们其实还是可以在这个步骤中访问到库 B，与C 的内节的(库的组合逻辑绝非一方屏蔽一方而是一方对一方的归纳简化和扩展导出)。

这是为什么呢？因为大架构是供你使用架构下的细节的（这后半句话才是重点），，大架构只是逻辑归约以更好面向程序员，它是逻辑简化而非逻辑替换，，而程序员最终要引用的逻辑是架构下的细节逻辑。。

是的，，明白这些有什么用呢？？但是反过来想一想，，如果连这都不懂，，，那么那才是你的损失之处呢^^（因为这是实践思想，，很多书都不会讲到）

6.26 大逻辑与小逻辑

SOA,COM,框架,这样的术语着眼于解决大逻辑间的开发,而不像 OO 语言本身只能将逻辑提供到类这一层次(虽然宠大的类也可称为大逻辑,但我们这里仅指有限属性和行为的类逻辑)

6.27 什么是范式

范式就是编程习惯，比如函数编程法，，OO 编程法， AOP 编程法
这些编程法是由计算机处理数据的方法和内部逻辑所决定的，，比如函数语言就是由 lamda 演算得来的，函数语言可以用一种其它范式的语言不能用的 eavl()过程
计算的原理是一种图灵机的离散形式，，，因此是命令式的,,,只需一个入口，和一堆要处理的数据，就能串行得到另一个结果(这就是串行形式算法,计算机的功能就是状态的改变,未来出现的并行计算和多核会导致编程出现并行范式)，这个结果可以被另一个串行处理作为中间结果,,,计算机(堆栈机)或虚拟机(JVM 的软件机器)是构建在这上面的另一层抽象,在这种开发方式下，要注意很多算法，，，离散的算法，和数学的算法,拿递归跟迭代来说,迭代就是计算机处理的方式，，，是行动导向的，而递归偏重于目标，，偏重于人的思维,,比如函数语言中，，就用迭代比较好， 而 OOP 中的语言中，，就用递归比较好因为计算机离散形式处理迭代好，迭代需要一个循环变量而递归不需要

在最开始，可将范式想象成一种特别聪明、能够自我适应的手法，它可以解决特定类型的问题。也就是说，它类似一些需要全面认识某个问题的人。在了解了问题的方方面面以后，最后提出一套最通用、最灵活的解决方案。具体问题或许是以前见到并解决过的。然而，从前的方案也许并不是最完善的，大家会看到它如何在一个范式里具体表达出来。尽管我们称之为“设计范式”，但它们实际上并不局限于设计领域。思考“范式”时，应脱离传统意义上分析、设计以及实施的思考方式。相反，“范式”是在一个程序里具体表达一套完整的思想，所以它有时可能出现在分析阶段或者高级设计阶段。这一点是非常有趣的，因为范式具有以代码形式直接实现的形式，所以可能不希望它在低级设计或者具体实施以前显露出来（而且事实上，除非真正进入那些阶段，否则一般意识不到自己需要一个范式来解决问题）。范式的基本概念亦可看成是程序设计的基本概念：添加一层新的抽象！只要我们抽象了某些东西，就相当于隔离了特定的细节。而且这后面最引人注目的动机就是“将保持不变的东西身上发生的变化孤立出来”。这样做的另一个原因是一旦发现程序的某部分由于这样或那样的原因可能发生变化，我们一般都想防止那些改变在代码内部繁衍出其他变化。这样做不仅可以降低代码的维护代价，也更便于我们理解（结果同样是降低开销）。为设计出功能强大且易于维护的应用项目，通常最困难的部分就是找出我称之为“领头变化”的东西。这意味着需要找出造成系统改变的最重要的东西，或者换一个角度，找出付出代价最高、开销最大的那一部分。一旦发现了“领头变化”，就可以为自己定下一个焦点，围绕它展开自己的设计。所以设计范式的最终目标

就是将代码中变化的内容隔离开。如果从这个角度观察，就会发现本书实际已采用了一些设计范式。举个例子来说，继承可以想象成一种设计范式（类似一个由编译器实现的）。在都拥有同样接口（即保持不变的东西）的对象内部，它允许我们表达行为上的差异（即发生变化的东西）。合成亦可想象成一种范式，因为它允许我们修改——动态或静态——用于实现类的对象，所以也能修改类的运作方式。在《Design Patterns》一书中，大家还能看到另一种范式：“继承器”（即 Iterator，Java 1.0 和 1.1 不负责任地把它叫作 Enumeration，即“枚举”；Java1.2 的集合则改回了“继承器”的称呼）。当我们在集合里遍历，逐个选择不同的元素时，继承器可将集合的实施细节有效地隐藏起来。利用继承器，可以编写出通用的代码，以便对一个序列里的所有元素采取某种操作，同时不必关心这个序列是如何构建的。这样一来，我们的通用代码即可伴随任何能产生继承器的集合使用。

第 7 章 实现抽象之数据结构

7.1 算法+数据结构的本质

数据结构的讨论是在抽象了数据类型之后才出现的(因此数据结构的准确含义是“计算机开发领域中的数据抽象学”)，汇编不需要变量是因为程序员包揽了内存分配，而高级语言提供变量，变量的内存分配由编译器或运行时完成，因此可在这个基础上发展基于靠近人的抽象数据结构，而 OO 既是对数据的一种抽象(当然，它跟数据结构对数据的抽象是站在不同角度的)，也是一种对代码的抽象

算法不是设计，（算法更多的不是代码逻辑的设计，用最小内核的流程控制比如 C 都可以实现算法跟数据结构），

一种算是通用的解决问题的方法,不限于编程开发
一种算法是算法设计方法,如递归
一种算法是计算机的执行方法,图灵算法,证明算法可行性
一种算法是跟某具体语言结合的数学问题的解答,如鸡兔问题等
一种算法是设计算法,架构逻辑
一种算法是综合设计算法,

算法并非代码逻辑，而只是附属于语言和数据结构学交界的那些东西（算法是从属数据结构的），只有设计模式才是代码逻辑和代码抽象学。。

从开发（者）的观点，我们可以把 OS 看成是提供 API 的软件抽象机制，同样从计算机开发领域的角度看，我们可以把算法看成是数据结构的附属，，因为数据结构是源于算法的，而数据结构是开发中的数据抽象，，,,因此作为从开发眼光来看的算法是数据结构的附属(因此人们说算法和数据结构是数学，计算机软件，计算机硬件三门学科之间的交叉学科，)。

7.2 函数增长与算法复杂性分析

从这一节开始，我们就慢慢进入算法了，，为什么把算法放在这里而不是放在数据结构那一章呢，，因为算法是属于数学和计算机的，，跟它们的结合要更前于数据结构(虽然也严重与数据结构有关，但是数据结构是进入到计算机以后的抽象，算法是它以前的概念)

，，先讲讲什么是算法，，它的分类与有关证明
递归算法，，,分支定界，，二分法，，贪婪法，等都是用来描述算法的(它们本身不是算法，，，只是用来设计算法的方法)
程序正确性

算法的有效性包含二部分,算法的正确性和算法的复杂性，,这里讨论算法的正确性，,复杂性在前面 2.2 节函数增长部分讨论过了

对算法的正确性的研究用到逻辑规则，证明技术(如数学归纳法)，算法

不考虑语法等其它因素，,,除非测试了所有可能的输入，，程序都给出正确的答案，这就是程序正确性的概念

如何把以上概念分解为形式定义使它具有可操作性呢？

在有输入的情况下，只要第一部分证明：若程序终止，则获得正确的答案，，第二部分证明：程序总是终止，就可以断言这个程序是正确的(二者之一成立是部分正确)

第三章讨论算法的正确性，这里讨论算法的复杂性
在假定输入值一样的条件下，，求算法的空间或时间复杂性
由于空间复杂性跟数据结构有关而本书不涉及太多的数据结构
因此在这里主要讨论算法的时间复杂性，，即步数
运算次数的单位是整数加法，减法，，等基本运算

7.3 数据与数据属性

研究数据结构我们目的从来不是那些底层的东西，基础的数据结构，，而是抽象的比如树，图，甚至是树图之上的，优先队列，多重集等。

在数据结构中，我们一般是从最普通的情况谈到施加了各件条件和限制的情况，比如有向图是无向图的一种特殊情况（施加了有向这个条件，而无向图是一种更一般的图，因此图论中一般是谈无向图及遍历等操作，再谈有向图及其特性。最后特别是有向无环图 DAG)。

逻辑上，数据结构的最高境界是集合（集合一不要求它的元素同型即是否 typed，甚至可以重复，二不要求元素间存不存在 ordered 或 sorted 关系，即没有施加任何条件和限制，或者说对这些限制不予考虑，无所谓这些条件与限制），所以通常用集合来形式上定义其它的逻辑数据结构。数据结构包括数据元素本身和数据元素之间的关系这二方面。所以讨论时通常用数据元素构成的集合和数据元素之间的关系构成的集合为基础来定义和导出其它的逻辑数据结构。

理解数据结构前理解什么是数据和数据节点，以及它们的属性，这是很重要的，但很多人忽略了它，以至于有些教科书都混用Ordered与Sorted，实际上，对这两个概念的理解至关重要，对它们的澄清工作是一本数据结构方面的书最应该首先完成的。它直接关系着数据结构学中诸多基础概念的导出。那么什么是数据和数据结点呢，数据节点又是体现了什么样的重要属性呢?这些属性(比如刚谈到的ordered,sorted)影响了这些逻辑数据结构在以后被拿来讨论时的哪些方方面面呢？

数据结构绝非仅仅数值数据结构。数据结构不仅用来研究数值，节点数据还可以是任何类型type。或ADT(从拥有类型的语言眼光来看，一切用代码结构能编码的计算机数据都可以是节点数据)，数据结构研究的数据是施加了一定限制的数据而绝非广义的数据，这主要体现在数据的二个属性上1.存不存在order还是sorted,2.有没有key value的分别，还是根本就无所谓这二种属性。

首先，数据结构能处理的数据节点是逻辑上分存不存在order的，如果无所谓order，那么集合就是这样一种逻辑数据结构，如果有order，那么就可以是线性list，层次tree(我们呆会会谈到order为何跟tree是有紧密联系的)这样的逻辑数据结构，

另外，计算机能处理的数据节点是分存不存在Key Value区别的，如果无所谓key value，那么其中每个节点都是数据实体，如果存在是key还是value的区别，那么这种逻辑数据结构中的数据可以表达为一个key,可能是一种关联式的逻辑数据结构。这就是计算机能处理的“数据‘和它能形成数据结构学这样的科学所加的二个属性，下面我们详细阐述之。

Ordered 表示一个接一个，顺序不可改变，如果被改变(比如一个元素后接的元素不再是以前那个了)，就不再是以前那个 ordered list 了，但它依然是 ordered 的(它的意义在于这里)。所以如果一个 unosorted 表被 sorted 过了，它肯定不再是以前那个 ordered list 了，只是因为它当中有些元素的 ordered 位置换动了，所以称它是一个新的 ordered 且 sorted 的表。

不妨把 Ordered 严格称为循序(而不称为顺序)，而把 Sorted 称为排序，那么我们就从字眼上对这二者进行了很好的区分。(当然你可以这么做，但为了统一起见，下文一律用顺序代替循序的说法。)

什么是 key value 呢？键可以是一个记录中的某个字段，或字段的组合，单字段的以这个字段值作为整个键。。关键字相当于字典中的单词，，而数据项相当于这个词的词义，造句，等全部的词条信息，这样说你一定不会明白，说实话我第一次看到这样的说话也没能明白。。实际上就是说，我们要查的是关于某个词的词条义，音等，，但我们是通过某个单词找到该单词的词条义，音的。。在字典中单词和其音义是一同被存入字典的，都是(某记录的)数据项，但单词是作为键的数据项。

明白了以上这样我们就可以理解一种特别的数据结构了，拿典型的hash table来说吧,首先就其key value属性来说，如果存在key value的区别，就是hash set，hash map之类的东西(因为hash set,hash map是关联式的数据结构)，，这跟属性中的第二点有关。。。第二，它的hash，是针对数据结构中的数据来说的，hash一定要是hash个数据结构出来，而hash数据结构中的数据是无所谓ordered还是不ordered的(hash table,map,set它们根本就是uniform的)，而一般数据结构是非uniform的，这满足第一个属性.

真正的数据结构其实只有二种，表和树，因为按第一点属性来看，前者是线性 order 序，后者是层次 order 序（我认为只有这二者才是划分和形成概念的标准），如果说硬有第三种，那就是无所谓 order 序的，即集合，(传统中一般把逻辑数据结构分为线性表，树，图，hash)但我觉得集合才是取代图的概念，而不是图，图只是节点间的逻辑关系居先，一般谈到 ordered 就是前驱后继的由来，而节点间的连通关系根本不是 ordered。

为什么树的 ordered 会成为划分树所属的逻辑派别的依据呢？比如二叉排序树它并非二叉有序树，为什么要进行这种区别呢(你可以在下一节找到答案)

7.4 线性表，树与 ordered

单一的一个 List 字眼表示数据结构的“表”，此时并不知道它是线性表还是什么其它表（比如链表）

在不同的场合，它关联着如下三种意义中的某一种：

1. Linked List
2. Vlist,即广义表，也即 Vlist = Lists(复数形式表广义表)
3. (Ordered) List

其中 3 就是普遍讲到的“有序线性表”，是 List 最广泛的表意所在。普通意义下的 List 表即 typed 的,ordered 的线性表。

狭义的 List 指 Linked list，隐喻就是它区别于 Sequence List = Sequence Ordered list（以顺序结构存储的线性表）

线性表除了有序线性表这种形式之外，还有 Sorted ordered list.即排序(线性)表。。除它之外，数组，集合，队列，堆栈这样的 Adt 都可以是 List

就树来说，如果将树某个节点的从左到右的子树看成有序的（有左右的 order 之分），那么就是有序树，即ordered 树，但却不是 sorted 的，因为只有二叉排序树 binary sorted tree 才是排序树（不过它不是左右子树之间的 sorted，而是左右子树与它们的根之间的 sortness）。

就特征的逐步放宽而言，存在，二叉树（order），二叉查找树(sorted)，N 叉树(un ordered),多叉树(之所以不把 N,多叉树这二者等同，是因为我们特别强调 N 叉树子树间不 order,而多叉树根本不考虑这一点,根本不考虑左右子树之间的关系)

线性表的线性“序 order”和树的层次“序 order”这二者有什么意义呢，这种意义决定了对于它们的遍历逻辑。搜索逻辑主要跟 sortness 有关，而不是 orderness 有关，线性表决定了它的元素顺序是有序 ordered 的(虽然并不一定经过排序 sorted)因此遍历可按前驱的方向或后继的方向但不一定搜索一定要严格按这个顺序因为那是由 sortness 决定的，因此树的搜索方向可先（根）序，后（根）序。而树的遍历其实任何访问都是从根开始，，所谓中序，其实也是从根开始，只不过它处理数据的顺序是不跟它的访问顺序一样的而已。而且它是只有二叉排序树才有的。

查找表 table 是同型数据 typed 的无序(sorted or unsorted)有限序列。二叉查找树也叫二叉顺序树。
二叉树遍历的本质在于动态地将非线性树转化为线性表。,

一维数组(向量)
二维数组(包含行向量与列向量的矩阵)

表 table 也是二维数组，它的行是记录，列是字段

线性是针对逻辑数据结构来说的，数据结构可按逻辑结构和存储结构分类，当数据结构按逻辑来分时，有一种是线性的，因为数据结构总有结点，线性数据结构中的结点存在线性关系，即每一个结点都有且只有一个前驱和后继，这种现象可用一种关系数学的表达符来表达(关系数学是关系数据库的数学基础，而计算机的数据结构学跟数据库学又是相通的)。。当然，头结点和尾结点除外，与线性对应的当然就是非线性了，比如树，除了根节点作为没有前驱的之外，包括根节点的任何节点却都在逻辑上（作为树的逻辑）有二个后继，还比如图，任何一个节点都在逻辑上可以包括多个前驱和后继，，，逻辑上都有前后之分只是线性的是一个对一个，而非线性的是一对多，或多对一。。

(下面我们用结点来表示逻辑数据结构中的独立项，用节点来表示存储结构中的独立项,用记录表达索引逻辑的独立项，这是三个完全不矛盾的词,)，

当数结按存储来分时，有一种是顺序的，即它申请了一块连续的内存来构见自己(至于它是用这内存实现了线性的还是非线性的逻辑上的数结我们不知道)，，数组用存储结构的相对位置来表达它的逻辑结构（这就是索引），，数组可以独立成为一种抽象数据结构(这就是普通的以数字为下标索引的数组)，，也可以用来实现更高级的抽象数据结构 (比如树)。。

按存储来分时，还存在链接型数组结构，索引数据结构，离散(散列)数据结构，其中顺序的当节点没有使用完全部空间，它就显得有点浪费，而其它的还行。。下面一一介绍。。

链式的很普通，索引式的就是系统(你的应用逻辑)维护一张索引表，里面有各个记录(我们称要索引才会有记录)的记录，比如一个记录的长度，存储位置,等，通过查找表，就能找到记录。

而散列的，系统维护一个函数而非一张表，函数体的逻辑计算出“记录的存储位置”和"key(KEY 就是索引逻辑面向用户的一层)"之间的对应关系。。

所以无论是顺序，链式，散列，索引表，我们都是在进行一种最终到记录的存储位置的索引工作。。数组名[下标]这样的一个形式，用计算机的眼光来看是内存地址，用人的抽象来看是索引工作。。而无论是顺序数组，还是链式，索引，散列，都是通过某种抽象形式(下标，，全局表，函数)来最终寻找到内存地址。。。
这个道理就像，我们通常不用&变量的形式来获得指针的意义，而是用*，因为&是面向内存的，而*是面向用户的抽象.

7.5 数据结构之数组

数组是编译期就指定的(数组大小固定，因此每个成员都被硬编码为内存地址，动态数组并非运行期数组，只是指定数组大小的是一个变量，实际上从编译器的眼光来看还是一种静态数组,STL 中的所谓动态数组 Vector 其实是用一般的静态数组来模拟的)，数组的特点就是可以下标索引，，抽象了对内存地址的需要作的干涉(只需操作索引就可操作数据)，从数组中增加删除一个数据是可能的，但数组本身业已变化，数组增删一个元素都要经过大量的重新对齐和移动操作(因为索引得视增加删除的位置重排)，，要形成一个新数组，从这个意思上看，，数组是根本运行期不可变的。它的变化只能发生在它的一个复制品上。 因为在静态期被固定了

数组的另一特点是它只存储同型 typed 数据

链表是动态运行时构造的，数组和链表都是一种存储机制，而堆栈队列数据结构是一种数据抽象机制(抽象了如何访问删除数据成员的通用接口)，数组可以摸拟这二者，当然链表也可以。。(这个道理表明，数据结构分存储数据结构和逻辑数据结构，中心是数据成员本身处理，不可能有离开数据存储地谈数据结构的道理(所以每一种抽象数据结构，必有存储机制

而链表则不一样，因为它内置了指针，没有抽象对内存地址需要做的干涉工作(对于程序员来说，这都是他们的工作，这样可以动态分配，开辟，增加删除成员)，，对数据的读写访问（我们知道，这是数据结构的中心存在意义所在）没有抽象上的索引可用，，只能操作指针和底层。。。

对链表的插入和删除是很方便的。因为只需往往改动一个指针，，所有的改变都是在这链表上发生并自更新，

树是一种可用链表和数组摸拟的抽象数据结构，，因为它内含了大小逻辑，，因此可以索引(左右索引，而不是一般数组的前后索引)，，也可指针索引

线性数据结构就是逻辑上有头有尾的一对一的数据结点组成的结构,,当它用于表时，就是 linear list,线性表用顺序存储的方法来存储就是顺序表 sequences list 了,注意线性表只指出结点有序(表的意义所在)一一对应(线性的意义所在)，至于如何有序它没有作规定，而数组是一种用下标 index 来体现’有序’的方式， indexied 的线性表即 index list(与 linked list 对应)=vector，因此是一种顺序表的方式(当然 index list 就字眼上说并没有完全地指出它应有的限制，准确的说法是 index linear list)。是顺序表最简单最基本的形式(注意数组是一种抽象数据结构)。与链式存储的单链表一样，是最基本的，因此常有 sequence stack,link

stack 的对比（而其实数组有多维数组，链表有循环链表等）。当然这二种表都是属于逻辑上的 linear list.

数组是一种顺序数据结构，它本身独立作为数据结构来用时，也是一种线性结构，然而它用来模拟树时，或用来实现其它数据结构时，那么这些形成的数据结构就谈不上是线性结构了(当然作为底层的数组它还是顺序的)。。

7.6 Vector 的观点

一般教科书把广义表跟（多维）数组放在一起讲解，是因为有时泛意义上的数组就是广义表的一种。

vector 就是数组的数组(STL 中 vector 是动态数组，我们不取那种说法)，我们知道逻辑上的多维数组必须转成一维数组的方式被存储，C 中的一维数组是地地道道的数组，，这个意义上的数组是实实在在的，，因为它们是按内存线性地址存储的，，所以明显地，，C 语言中的其它泛数组的 adt 都是抽象的，，从 C 用一维数组实现多维数组这个意思上，可以看出 C 语言的确是面向底层的，，它企图用底层解释一切。由于它的这种作为，导致了表示方式上的一些多义性，比如多维数组的指针跟元素的表示混淆不错。

C 中泛数组有普通多维数组,动态数组和 vector,其中又以 vector 最为典型，我们知道，多维数组要转为逻辑等价的一维数组，，有三种方式，，1，row 列优先，这就是 C 的方式，，比如 foo[3][4],它有三行 4 列 row,它先每一行排 3 个，再 4 列,,2,行优先，这也就是 pascal 的方式，但是这二种方式都有局限，因为它们只能实现为方形规则的数组，要突然这种局限就是 vector 的事情了，

3,,就是 vector 方式，，C 标准库中倒是没有一个 vector,我们讲 c++的 stl 中的 vector,一门语言要模拟多维数组，必须要达到一种“多维数组就是数组的数组”这样的抽象，而 vector 就是这种思想的本质，，因为 3 维数组是 2 维的，这就是说，3 维数组是 2 维数组的数组，而由 3 到 2,就是把 3 维中的其中一维标准化了,,缩为 1 了，，这正是 vector 的思想，不要把线性代数中的矢量跟这里的混淆了，但在线性代数中这样的比较有巧合性，比如矩阵(对应多维数组)是由矢量(某维缩为 1 的矩阵）所组成的，，矢量就是数组中的数组了。。比如 foo[3][4][5],可以是:

(1)一个下标为 3 的一维数组，，每个元素都是[4][5]的二维数组；
(2)..
(3)..

7.7 数据结构初步引象(2)

数据结构中也有文件的概念，实际上当这个文件(数据结构一般处理数据在内存中的模型)放在外存中(被持久化)就成了数据库，数据结构中也大量用到字段，记录，键等数据库概念，，在计算机领域，数据库和数据结构本来就是同根共源的，数据结构的抽象模式就是内模式，，关系数据库与其它数据库的区别在于关系数据库会返回一个集合，而不是一些特定记录，在数据结构的讨论中，频频用到关系代数的概念，比如排序线性表 sorted list.它按照某种顺序来组织数据，比如<，这实际上是一种偏序关系。

如 binary search tree,= binary sort tree

查找表(lookup table?search table?)是一种无序的集合,这使得只能以顺序比较方式操作，因此我们必须对它们强加一些关系，形成 sorted 的,动态查找表必须动态变化以维持它的表序(即它是一种动态查找表，表是自变化的以维持某种利于查找的顺序)。。

（一般）树，森林，二叉树之间可以相互变换，，树可以通过对节点的限制或修饰发展出多种抽象，比如平衡树，AVL 树，B+(B_ 的一种变体)，红黑树。。。当然还有很多，因此数据结构实际上是一门可以无限深入的科学

如果说树是一种层次关系(树是严格分父子的)，那么图就是一种松散关系，在任何节点间都会产生关系，因此最符合现实模型。

对图的学习涉及到关系代数，线性代数中很多知识

其实要注意各种数据结构的存储结构，图的多重链表，树的最小带权路径，，图的多路查找等等知识，这些都是高级话题。。

7.8 数据结构初步引象(3)

双端队列是栈和队列的一种泛化（因为它的二头都可以出或入）。栈和队列跟其它数据结构遍历不同的是，对于栈和队列的内部是不可访问的，只有栈顶和队列头这几个元素可以遍历到。

查找表是一种集合，是同类型数据的集合，，因为集合是一种无结构的无序松散排列（表是一对一更，树是一对多，图是多对多，那么集合对这些都没有规定，实际上有表，图，集合三种大数据结构，树是表的推广，图是树的推广，但表和树都有作为自身存在的意义和作为树和图的双重意义存在，在所有数据结构中，反而只有图才是用得最为广泛的。。

)。。

用位向量可以表示集合（这有点像用邻接矩阵表示图），其它方法也可，比如链表等，，但位向量表达法在提供了位操的机器上产生的效益是很高的。

图也有根，还有无序链表 unsorted 还是 unordered？？连通和强连通，，，回路和路径，，环弧边，无向图是无向还是双向图，这些都是微妙差别的概念。。

7.9 数据结构初步引象(4)

线性表是最常见的数据结构和逻辑结构，然而只有图才是最常用的数据结构和逻辑结构。。

其实树状结构有天然的转化成线性结构的优势，因此也有作为线性表的树的存在
(因此说树是树性表的一种推广,教学的时候完全可以从线性表引伸到树,,
树有作为线性表存在的树，树也有作为树本身意义存在的树，和作为树图意义存在的树，这三种情况都需要被讨论。)
此时从树的眼光来看这种原来是树的数据结构它还是存在父子关系，从线性表的眼光来看是变换了的平等关系.

7.10 树与图初步引象

树是图的一种特例（树用图的眼光来定义是：节点和边，不过它的边是有向边，即树是 sparse 图, 没有回路的连通图，而图的边的关系仅仅是连通关系，实际上树的边含义要丰富一些，比如它还包括层次的隐喻），图的树的区别在于，树是严格分层次的(它的前驱是父，后继只能是子，因此产生出父子，节点的高，宽度，树的高，宽度等概念)，而图是任何顶点间都有可能发生关系(即存在边，存在边就叫邻接),除了树，图之外，还有森林（相比树对图的定义来说。它少了一个条件，森林是没有环，可能不连通的图,一棵树也可以是林林，但森林不一定是一棵树），,二叉树与一般树的区别在于 1，一般树不可以无节点，而二叉树可以没有节点，2，二叉树的某个节点至多有二子节点，即二叉树不但分层次，而且子树也分顺序，因此对于一般树的边历过程有三种,,先（根）边历，中（根）边历，后（根）边历，，但是正是因为二叉树有左右子树之分，因此中序边历是二叉树特有的。

树和图都可以作为数据结构(主要作用在于存数据,附带一定逻辑的数据)或逻辑结构（主要作用在于表逻辑），节点可存任何东西，比如一个条件，一个值或一个结构都可以。边也可以加权表逻辑。。比如最小生成树是最小加权生成树的简称，，注意，树有最小子树，图也有最小生成树。

如何存储呢，因为用图可以用来解释树，因为决定图的特征在于它的节点，因此它也可以用跟图一样的存储结构来表示自己。但是因为树是图的稀疏表示，一般不用邻接数组存储节点的每个所有特征，,引线二叉树才这样做(因为它赖此达到一种维护它访问的能力)。

7.11 树初步引象

(1)
树为什么称为递归的呢，就是因为它的子树也是一棵树，，而这棵树的子子树也是一棵树，因此被称为递归定义的。

树的高度就是深度，因为根的深度就是高度。（因为递归定义中非叶节点就是子树，所以非叶节点的高度就是该子树的高度宽度等特征，即用根节点特征来代替子节点高度等特征）

兄弟是同一个节点为父的子节点，而堂兄弟是父节点在同一层的子节点。

通路跟回路，七桥问题解决的是欧拉通路而非回路，

最小生成树是最小加权生成树的简称，但是存在另外一种最小非负加权生成树的东西，，就是最小代价 cost 生成树了。

在树中，有 height balanced tree 也有 weight balanced tree,这个 weight 就是一个层上节点的最大值。。

作为图的树的深度优先有时要求回溯，深度优先遍历时，它从一个节点开始往下遍历时，在到达叶节点时需要重新向上“溯树”。
这就需要额外维护一个栈来记录已经访问过的节点。当访问到一开头那个节点时就“标记已访问并存入栈”，下次从这个节点的兄弟作下步向下遍历完成并作回溯时需要将其弹出堆栈，，，以此类推完成遍历（因此它是一个由树的递归特性决定的递归过程）。。

图的广度遍历用到了队列逻辑。

字符串这么平常，然而却需要涉及到不浅的数据结构和I0，，这使语言表达字串方面成为学习这种语言的一个重要方面。

7.12 B 减树

B-tree 就是 B 减树（因为存在一个 B+树所以会导致这样的误解，实际上并没有 B 减树这个概念存在，-号只是一个连字符而已也可写成 B_），也不要跟二叉树 binary tree 混淆了。

那么什么是 B 减树呢，一般说它是 Bayer’s tree 的缩写（bayer 是这种树的创建者之一的名字，另外一种最常见的命名方法是 b=broad,bushy,因为这种树所有“外部叶节点”在同一 level 上聚集）。它适用于树的内部节点被频繁访问，但又一般不常访问这些节点以下节点的情况，比如计算机的二级存储器硬盘等设备，常以结点为目录，叶节点为文件，，一般我们频繁访问目录但又不常深入最终每个文件。

实际上 b 减树也是要求一定的 balance 的(有人也说 B 减树的 B 代表 balance,不过这样的话就与普通的平衡树相重意义了所以并不常采用这种),,它是一种动态查找树，因为需要在每次 update 后都动态调整它的节点情况。只不过它并不需要作特别频繁的 rebalancing 动作，因为 B 减树将外部节点维护在同一层次上这个特点使它只需要只很少的调整便可保持 balanced 。

那么 B 减树到底具体是如何一种数据逻辑呢？它到底如何使保持平衡只需很少的调整呢？更重要的是，B 减树到底有什么用呢？

首先，这种树的某个（或每个）“内部非叶节点”的下层子节点(直接子树)是数量可变的，而且在一个区间里变动（我们呆会再来讨论这个所谓的区间），这个非叶节点维护
一堆 elements 和 pointer,其中 elments 用来区别各个子树的取值范围，比如这个非叶节点有 3 个子节点为根的子树，那么它需要维护 2 个(子节点数-1 个)elments，假设为 key1,key2，那么第一个子树的所有值都小于 key1,中间子树的所有值都在 key1 和 key2 之间，最右边子树的所有值都小于 key2(当然，这是 N 叉树，3 叉树，这里不是说二叉树的左右)，很显然地，key1,key2 这些 elments 是 sorted 有序线性表，那么 points 部分呢，它指向每个子树..有几个子树(子节点)就有几个 pointer.

所以，每个内部节点维护（该节点所拥有子树数-1）个的 elements 和（子树数）个的 pointer.叶节点在同一层上，没有 element 也没有 pointer.，不带任何信息。

现在来讨论那个所谓的区间，以 m 为 order 的 B 减树(称为 B-tree of order m)，再设 n 是一个内部节点允许的最少节点数，那么那么区
间就是[n 上界,m 下界]，以这种区间为表征的 B 减树一般呈现出以下特性
（1） 每个内部子节点（只要不是作为该子树的根或叶子）都至少有 m/2 个子节点；
注意，最多 m,最少 m/2;此时 n 相当于 m/2
（2） 每个内部子节点(如果它就是作为子树的根并且不是叶子)那么最少可以有 2 个子

节点；
这说明，该内部节点最大子 elments 数可以为 m 或 m-1,最少 elements 数可以为 m/2 或 m/2-1(n=m/2 或 n=m/2-1);
(3)每个内部子节点(如果它是叶子),,，那么不带任何 elemnets 或 pointers

所以，“子树数-1 个 elements”,“子树数个 pointer”,“叶节点不带任何信息”，“n-m 的区间所体现的上面三点”,，这些都说明了什
么呢，这些特征都赋于了这种树什么样的特性和能力呢？

我们可以看到，如果内部节点不是作为最终叶节点，那么它(要讨论的这个内部节点)的子节点(它的下一级子节点)个数至少是半满 half full 的，这
意味着，在[m,n]区间内(或称[m/2,m])二个半满的内部节点可以进行合并形成一个合法的新内部节点。一个全满的内部节点可以分离成二个合法的
新内部节点(只要父节点中可以容纳这些新子节点就可以)，从这层意义上，的确不需要太多的高度上的平衡。

更多的关于这种树的删除，插入算法可以从这层意义上引申而来.

7.13 图初步引象

线性表是数据项平等的集合(抽象数据的最高境界最一般情况是集合,因此在对各种 ADT 进行定义的时候都涉及有集合)，无论对其进行什么操作，只有维护一个线性关系而不管它是无序还是有序的，就是线性表，而树的各数据项是有层次 level 的，无论对其进行什么变换，只有作为树的眼光从根向下来看，它总存在逻辑意义上的父子，这层逻辑意义是作为树的意义存在的，在对树的各种操作中都考虑进去了作为处理时的因素的),

因此从逻辑意义上来说，树是线性表的推广,,,图却不是树的推广,,,因为虽然它也处理数据项集，但它处理的是数据项间连通关系而不是地位关系，在其各种存储表示和后来高级逻辑中都会加入邻接考虑。

如果说其它数据结构只有数据项集，那么图不但有数据项集，而且有顶点连通关系集。参见对其 ad t 的定义就知道了,,而显然连通关系并非前驱后继关系,，在关系代数中,，存在有全序偏序对称自反等关系。。
也即在图中，结点的地位关系我们不考虑，不存在线性平等也不存在树型父子，而是不考虑这样的地位关系,这一点上它像集合，(集合，是不考虑结点之间地位
关系也不考虑连通的)，图只考虑一种称为连通关系的不伦不类的关系，这一点上它区别所有的数据结构，它的二个顶点间可以连接，这一点跟树相同，但是顶点
并不经常用作存储数据用，这一点又跟树不同，树的二顶点存在的是地位关系，而图的二顶点要处理的是连通不连通关系。。

因此我们规定，逻辑数据结构只有二种线性和非线性，只要不是线性，那就是非线性，而不管它是节点地位关系导致的非线性，还是其它关系，比如连通不连通导致的非线性（其实这二者无关，因为图也可用矩阵来存，，而矩阵，虽然是一种非线性结构，但以某种眼光看，它又是线性的）。

数据结构也可是逻辑结构，树可作为数据处理结构也可作为逻辑表达结构比如流程，因此图不但是一种数据结构而且还是一种组合数学中的离散结构(比如拓朴学就是关于有向图的)。。。.

7.14 树的平衡与旋转

树是递归定义的，所以树是用它的根节点来定义和代表的，子树就是以那个子节点为根的树(如果它存在的话)。所以我们常说，某某树的左节点(以递归的树的眼光来看，就是某某树的左子树)，某某树的右子树的右节点。如此一直递归。

搞清了这些(子树，树或子树的跟节点这几个概念等价，所以我们说树深高就是以那个树的根节点的深高为表征的)，我们再来看树的深高。

首先，深高(注意我没有提到树)都是发生在一个到两个节点间的(深是两个，高是一个，所以这两个概念中所谓的节点只有后来被体现到根上，才演变为树的深高)

两个节点间的深是唯一的，因为深就是路径，而两节点间的路径是唯一的，所以体现到根上，就是树的深为零(因为是根到根的路径)，在树中取一个固定节点，越往树根的位置处的某个另外节点，深就越高。所以把远离树根的方向看成深度方向。

某个节点间的高，这个概念中，以这个节点为固定节点，尽可能往深度方向的某个节点靠拢，找到深度最大的那个点，这两点间的路径那就是高。(所以，所谓高，也是两节点间的，而且也是两节点间的路径，即变相了的深度，也即，深度中的另外一个节点是不固定的)当然，为了把握概念，我们说高度是发生在一个节点上的。

这样的话，有一件事应该很清晰了，给你在纸上画一颗树，其平不平衡，不平衡在哪，一眼就能看得出来，

那个产生不平衡的节点的高度差其实一眼就可以看出来，因为高度差就是深度方向上(与)的差嘛，这就是不平衡子树

实际上 avl 树只是自平衡二叉树的一种，高度差为一是这种树的最古老定义（ self

balanceing 并不仅仅是 height balancing 动作。而 AVL 仅仅是 balanacing the height)，还存在其它不以 height 的调整为其平衡方式的树，比如红黑树，但是只有 AVL 树是严格用平衡因子去定义的(因此是“平衡”的最正宗意义所在)，而红黑树不是，因此红黑树不是平衡二叉树，实际上它故意允许一定程序上的不平衡。

红黑树不是平衡二叉树但它“保证”2*O(logN)的最坏情况下的平衡度（人们往往根据这个原因把它归为跟 AVL 一类），AVL 树是 1.44*O(logN)，这二种树只“保证”（注意只是保证）最差情况下的下界为 1.44*O(logN)或 2*O(logN)，称它们为平衡树不是因为其内部是不是极力在保持平衡还是不是，重点在这里（最坏情况下保证了多少的平衡度）,，综合考虑插入，查找，删除等操作来说，红黑树的效益要好于 AVL，因为 AVL 是严格平衡的因此只对查找 intensive。查找时只需向上查找 1.44*O(logN)个节点就行(经过 AVL 平衡的树只需对 height bounded at 1.44*O(logN)个节点进行处理)。。

一个节点的平衡因子是左右子树根节点深度之差(注意并非高度之差)，因此不要跟这个节点所处的深度本身搞混了，一个只有左子树无右子树的树，它的树根的深度是 0(树根的高度越到根越大，树根的深度(相对整树来说)永远是 0 空树为-1,这个深度并不影响树根的平衡因子)，平衡因子是 1(只跟左右子树的深度有关)，因为右子树不存在，深度为-1,（而左子树深度为 0）,因此它们差的绝对值为 1,,即平衡因子为 1.

在建立 AVL 树（向一棵普通意义下的 binary sort tree 进入插入一系列 key 码）时，边插入边判断（插入点的平衡因子是否导致了不平衡），在发现因为插入动作导致的不平衡现象时进行 rotation.，以维护其平衡性，直到所有的数据插完，这树依然维持 binary sort 性质（因此称 sort tree 为查找树，，即查找特性 intensive 的，作为数据结构的树，其实它的本名最开始是 sort tree 然后才是 search tree，search tree 一般有二种，第一种是每个节点都含 key 和 value 的，所以它的每个节点都包含数据，而有些 search tree 只有叶节点含数据，其包括根在内的内部节点都用来 search,只包含有 key）和 height balanced 性质。那么怎么样进行判断并在需要的时克 rot(根据情况不同有时是二次 rot)呢？ 在这之前，我们需要规范一些概念。1.rot 所涉及到的主体（插入点和支点 privot）2,最小不平衡子树。3, 扁担原理。

最好的方法是举个例子。
比如我们将考虑把{20,35,40,15,30,25}制造为一棵 binary sort 并 height balanced tree，，

这样我们就解决了开篇提到的“扁担原理”，综上所述，我们讨论了 RR 的情况（如何判断违规以及如何作重平衡处理），，下面谈到的是 LR 的情况了。请自行理解。.

7.15 完全与满

Vector 之所以称为 index list，，我们知道 index 是索引式存储，list 是逻辑结构，于是四种存储结构和四种逻辑结构可以组合到 16 种组合方式。而 index list 正好是这其中的一种。

满二叉树的概念容易弄懂，而完全二叉树（二叉树都是从左到右有序的）这个概念实际上并不突出“编号”，它表现的是这样一种树：如果在某一层上，左边的节点为空，那么（在这个节点）右边就不能有节点，无论是这个节点右边的节点是个兄弟节点还是堂兄弟节点。所以满二叉树一定是个完全二叉树，而完全二叉树不一定是个满二叉树。。

树的遍历分先序，中序和后序，又分递归遍历法和非递归遍历法，故有 6 种遍历方法(也许只有前序才能递归？因为它是根，只有根才有递归意义？)。 当把对树的递归遍历转化为非递归遍历时需要一个辅助栈外加几个循环
(我们知道加栈是递归算法转化为非递归的通用手段之一)。

完全二叉树中的完全跟完全 (无向)图中的完全以及完全有向图的完全意思是不一样的，前者并不是一种规则形状(比起满二叉树来)而后两者是规则情况。

因为连通图对无向图有意义但对有向图却没有意义，因此对有向图引入强连通分量的概念。

7.16 多路 234 树与红黑树的导出

数据结构间都是有联系的，比如 234 树其实可以导出红黑树也可导出 B 树。

红黑树也是一种二叉排序树，因此一方面保证了查找效率（中序遍历时可以快速到达根部），另一方面，它也保证“最深的深度不大于最浅的深度的 2 倍”，这使得红黑树有一定程度的 balance 特性，因此对于查找之外的另外操作，它也有很不错的效率。

红黑树主要是通过为每一个节点着色来达到以上特征的

1. 首先一个节点要么是红要么是黑只有二色可以被用来着色。
2. 根节点默认为黑色
3. 对于叶节点来说，不管它的父节点是黑色还是红色，一律着色为黑色。（每一个叶节点我们都可以假设它是一个内部节点因此有有二个不存在的子节点设为 NULL）
4. 对于任何一个节点来说如果它是红色，那么它的二个子节点都为黑色。(从每个叶子到根的所有路径上不能有两个连续的红色节点，因为从叶子到根的路径是唯一

的，也即它就是从根到这个叶子的路径，因此它是这个叶子的深度)

5. 高度方向上（注意这里不谈到具体的高度，而是指高度方向上），从任一节点到其每个子孙叶子的所有路径都包含相同数目的黑色节点。

以上五条特性导致了

深度上，“最深的深度不大于最浅的深度的 2 倍”， 从根到叶子的最长的可能路径不多于最短的可能路径的两倍长。

我们来推导一下上述性质，并规范一下“从节点到根，从根到节点，从节点到叶节点”这些说法的准确含义。(注意有时候高度就是深度，当一条边的层次决定它有多少个点附着上面的时候，这也就决定了它的高度和深度具有同一性).

7.17 真正的 ADT

在讲解算法与数据结构的教科书中（特别是 C++用来表达数据结构与算法），有一种语言抽象机制屡屡被谈到，这就是 ADT。

类是真正的数据，，比类的成员数据(POD,C 的 plain old data，数据结构学中，我们从来不是大量需要数组，链表这样的底层 ADT，而是需要更抽象的比如堆栈队列，红黑树，hash map 这样的更为抽象的东西)更能代表数据的意义，我们的程序就是一个一个的数据，类是用用户的观点来表达现实生活中出现的各种各样的数据的最好方式,因此会有面向对象数据库的出现(数据库技术中，关系模式的下表达的数据给人的引象似乎是它是为专门的数值数据而建立的)

ADT 就是指封装了事物属性跟作用的指代物本身，它的属性和作用是 ADT 之所以为 ADT 的意义所在。即数据类型不再是基本类型，而是结合了数据跟代码的抽象了的模型（是一种数据类型）

相比线性表 O(n)的复杂度来说(输入的元素个数是最主要的影响因子)，对二叉树的各种操作（插入，查找删除）效益直接受制于高度，即 h = O(logN)，即树提供了 O(logN)的复杂度，其中 N 是节点数，（二叉树中）我们并不直接把 N 作为影响操作效益的因素，h 才是，因此需要对 height 进行 balancing 才能控制操作效益。

BST 的构造是严格取决于待输入系列的，当待输入系列本身就是一个 sorted 的系列时，那么当这些序列被用来构造 BST 时，它就是会退化成对应的“链表”。在操作上会失去

树的优势。

7.18 数据结构的抽象名字

算法源于用计算机去解决实际问题，对由研究算法过程中出现的数据结构问题的研究也是一个很重要的工作，特定数据结构和作用于其上的算法们，就称为一个 adt..

有时，上层的抽象 adt 可以有自己的一套算法独成一种 adt,与它们的子 adt 一样，,,，有时，adt 之间进行某种意义上的组合形成新的变种 adt,,,又有自己的一种算法，如果组合了 2 种 adt,那么就是一种带有 2 种意义的 adt,有时，一种 adt 以另一种 adt 作为实现，自己作为一种抽象，,这样的方式演化成一种新的 adt.我们必须明白这所有抽象叫法之间的关系，明确他们其实所指的东西有很大的不一样。。而且要明确这些叫法之间的特指与泛指的层次关系。。谁由谁通过谁抽象得来。

关联数组包括普通数组，关联数组是一种可以被称为广义数组的数组，我们知道关联数组是 key value 索引对，只是它的 key 可以是任何类型，而普通数组只是 interger,,,关联数组有很多抽象，甚至只是别名，比如在 smalltalk,objectivec,.net,Python,realbasic 中被称为 dictionaries，在 perl 和 Ruby 中被称为 hashes，在 C++和 Java 中被称为 maps,在 common lisp 和 Windows powershell 中被称为 hashetables,在 php 中存在关联数组，只是索引被限制成整型和字符串，这就变成普通数组和字典了，在 Lua 中唯一只有关联数组这种数据结构，被称为 table,这也是关联数组比较正统的称法之一，

我们知道集合也是一种关联数组，不过它把 key value 对中的 value 给忽略了，把 key 作为 value,从有 keyvalue 对这一点来说它像关联数组，另外，它的索引就是 1 到 n 的某个子集，从这一点来说又像普通数组，联系一下 bit vecotr.

数组跟向量这个说法的关系是什么呢，其实向量 vector 一般被实现为动态数组 dymic array，我们知道静态数组的空间是在编译期被分配的，那么动态数组就是用 malloc 函数在运行期构成起来的数组逻辑，我们知道数组能线性时间随机访问，但是不好重建和插入，因为需要移动插入点后的所有元素，比起 linked list 来它的索引字段并非指针，而是直接对应内存位置的硬编码索引，因此比不上 linked list 在发生插入时可以仅仅通过转变指向的方式就可以实现数据结构内部的重建。。而动态数组就是这样一种在运行期有优化了的重建能力的数组逻辑。。

vector 的学名叫向量，也即数组 index list 的一种,linked list 也即链表而非索引表,跟数组有关的不只向量，还有队列，堆栈关于数组的表示，甚至还有树，图,矩阵(一种多维数组，C 语言中把 foo[m][n] 看成是 foo[m*n]的一维数组)，关联表等，除了数组是实现之外，,

其它的叫法，它们有些不是指同一个东西(比如向量跟矩阵是不一样的东西，数学上的向量是矩阵中某维为 1 的情况下的子矩阵，是构成矩阵的量，向量是向量空间的概念，而矩阵是线性空间的概念，不过他们同构，于是在运算上涉及到了一起，数据结构上，一般 vector 就是一维数组，而 matrix 是多维数组)，，有些是一层一层的抽象关系(比如 vector 是 index list 的一种， index list 又是顺序列表的一种,图有一种形式是树，而树又是 bst 的基础)。都是建交在 C 数组这个语言要素上的抽象 (前者是实现，是存储抽象，后者是高层抽象叫法).

7.19 真正的数据结构

数据结构是什么？它是组织内存中对象或基本类型数值(primtive types)的形式，为了更好地组织和使用这些对象而慢慢发展起来的固有形式，惯用法(idioms),

数据分类与 ADT

很多地方都用到到向量，，在学汇编时，，中断向量表就是一种向量,为什么 STL 没有 arrary 而只有 vector 呢？因为数组一般都用来实现 vector，而且数组是语言的内含类型,一定程序上不提供太多的接口(根本因为是作为数值形式的数值没有被 wrapper 成一个 first class 因此不能作为函数的参数—因为没有复制构造函数，只能作为指针形式间接来操作它，也因此不能成为一个函数的返回值)，而 Vector 可以提供跟数组类似的结构（也有多维数组）和比数组更高级的使用接口(一般数组类型只能是静态的不可伸缩的，而 Vector 可作为动态数组动态改变大小)

实际上多维数组并不是在内存中是一个标准的矩阵(学过线代就知道，任何一种矩阵都可以化为它的等价三角矩阵)，比如 C 或 C++就用一种行或列优先的方式来索引其元素。

数组是一个静态配置的空间，在命名方面 JAVA 做得比 C++好（个人看法），比如 INT MyVar[10];作为数组名的 MyVar 带了序列，而JAVA 中 INT[] MyVar;这就有点相当于 type *是指向某种 type 的指针类型，而 type[]是某种大小的数组类型一样好记

下面来阐述几个易混淆的概念

顺序线性表，，有序线性表(orderd list)，hasp map（可用关联容器来表达）都是列表，比如字典顺序，也即字母顺序的一种关联机制

线性表即通俗意义上的“线性数据结构”，线性的意思是什么呢，它一定要满足几个规则(1，有唯一的第一个元素和最后一个元素，2 除了第一个元素都有一个前驱，，3，除了最后一个元素，都有一个后缀)

上述的元素就是数据结点的意思，数据和数据结点之间的关系可表达为 B=(K,R)的二元组,满足关系的二个元素一个称为前驱一个为后继

计算机存储数据的方式只有二种，顺序存储与离散存储(又有链式，索引式，Hash 式)，这是面向计算机端的存储结构，本向用户端的逻辑结构有集合(set),,哈希(hash map 是一种 map),,表(table),数组(array),,向量(vector),,图,树,map 映射,线性表，节点链式表 linked list,多维数据结构(matrix)等等，这种关系就有点像数据库的内外模式之分。逻辑结构间也有高级的演变方式，比如用 vector 来实现矩阵和 table

map 是映射，,，set 是集合,,都是关联型的数据结构，因此可用关联容器来表达

集合这种数据结构是用位集来描述数据集(可能以一个数组的形式存在),通过移位和位运算

线性表就是不能随机存储的表，像数组就是线性的，堆栈和队列也是线性的,有双向链表的存在(list),,dequece(double end quece),
表 table 就对应关系数据中的模式,,也即一个记录是一个表，它的各个字段就是表的竖维聚合数据类型就是像图(map)啊,树啊之类的，树是一种特别的图(每二个节点都有通路)而且是简单图

图的存储结构主要由它的邻接矩阵来表示，或邻接表，而树的存储表示主要由以下三种表示：结点表示法，兄弟子女表示法，等

这就是关联,，关联一定有 key 和一个 value,,它们一同被存入数据结构，然后运用某种机制(可能是 hash)通过 key 来索引 value
图是一种离散结构而非一种数据结构
图的边就是顶点之间的关系，这种关系是单向的或者双向的
像 MFC 的消息系统用的就是表驱动方式(它的消息映射表就是一个静态表)
我们知道，栈是一端开口的，往往从高端压和出栈(称为栈顶)后进先出的（但是后出先进这种说法是不存的，因为当一个栈没有数据时，它就不能出任何东西，因此只能说后进先出，先假设它有至少一个数据存在）因此它有当前指针和栈顶指针这二个元素来表示(当前指针指)，栈往往用数组来模拟(++p,p-- 这样的形式)，栈其实是一种跟它的实现形式无关的思想而已(是一种逻辑结构)，因此可以说是数组（指针数组）来摸拟（顺序存储的存储结构)，可以用数组的一端，当，而队列是一种先进先出的，它二端开口，有三个描述元素(尾，头和当前指针)，它往往被实现作为一个管道（因为队列从意义上来讲它也其实就是一端进而从另一端出的数据结构啊）作为缓冲，或 forward 给其它处理
list 是列表的意思，有序列表 orderdlist，链表(linked list 是一种 orderd list)，堆栈，队列都是一种 orderd list
就像链表和数组都可以仿真堆栈一样,，树啊，图啊都是思想模型，链表和数组才是实际

存储的机制
对数据结构的讨论中经常用到递归，特别是树中，因为树本质就是一个递归结构，在回溯节点时就是回溯同一种节点(因此这个节点可用递归描述)

递归跟回溯,栈

递推与迭代还是有区别的，递归就是用自己来定义自己，，一般不需要一个循环，，而迭代需要从 1 开始，将这个循环变量一直自加到最大值(循环不变量？)，，需要一个循环，一般来说，迭代比递归更有效率(在某些专门对递归进行了优化的环境中除外)

对一种数据结构的讨论常常不但要明白它们的工作原理，还要明白它们的操作，如查找，排序等，数据结构存储结构和这些操作就构成了 ADT

《数据结构 C 语言版》

前言:这是我在 2005.7 月 - 2005.9 月署假看《数据结构 C 语言版 - 清华大学出版社 黄国瑜叶芳菁著》时写的读书笔记,现在把它发布出来,希望对大家有用,也算是作个备忘录吧,不科学之处,还望高手斧正.

7.20 堆栈与队列

堆栈与队列都是数据结构(更复杂的还有树,图)在关系上是平等的数据结构,实际上堆栈与队列都是"内含在空间里的数据块",堆栈,队列的本质就是"数据块",不过它们都是包含在特定内存空间里的"有序数据块",如下图(4.bmp,用数组模拟"特定内存空间")
"特定内存空间"可以用数组表示,也可以用链表表示,数组是内存中的线形空间,也就是说,数组可以在内存中开辟一段空间,该空间是线性连续的,对该空间里任一元素的存取要根据"索引值"来进行,这些索引取从 0~MS-1(如定义一个 int queue[ms] 或 int stack[ms])是线性递增的,与数组空间从低地到高地的排列一一对应,数组的每一个空间都可有数据,也可无数据,每个空间的大小为一个 int,整个数组大小就是 ms 个 int,但是无论如何,对其中任何一个元素的存取(存是往数组任意位置里存入一个大小为 int 的空间,或在某个无数据内容的数组空单元空间里赋值,显然,这个数组空间是本来就存在于数组内的,而不同于链表要动态开辟一个新空间,而如果是前一种情况,数组就不再是静态的空间了,因为它的大小由 ms 变成了 ms+1,而这是不可行的,同理,取是释放一个空间单元,这就使数组总空间大小由 ms 变为 ms-1,或在某个空间里赋值 0,术语称置 0,而事先不管这个空间里有无值,如果有值,有的是什么值),其实,对数组索引的描述不但可用 0~MS-1,当然也可用 1~MS,但是为了方便考虑,把前一种看作为常用的,专业的方式,当然还可用 2~ms+1(3 种方式在定义了一个大小为 ms 的数组的情况下都可行可用),因为数组空间是一定的,对其的表示方式当然可以自由地使用不同的方法,一切的一切,只要保证能正确存取到所需的数据为程序所用为准,因为这是数组这种数据结构要最终达到的作用和总则,另外还要保持易用

(像 0~ms-1 就显得专业并且简单易用,1~ms 就人性化,而 2~ms+1 就什么都不是),上图中的数组示意图都开了"口",是表示只能从数组的开口的那一端存取数据(数组每个元素都是空间里包含的内容值,即数据,请搞清"元素","数组每个空间单元","内容值"等的说法),是一种形象的表示方法,而标准的数组图示方法可如下表示:(5.bmp)哪为什么要开口呢?一个数组为什么能开口呢?这是因为前面提到,这个"数组开辟的空间"将作"堆栈空间"使用,也就是"用数组模拟堆栈",因此标准数组示意图就要经过一些变形以适应能正确表示堆栈(空间)的要求,首先第一点变化就是"开了口",第二个变化就是还加入了一个 top 指针,这二个变化适应"能正确地表示堆栈(空间)"而生,其实要说 top 指针,它还不是标准意义上的指针,指针是一个 32 位的整型,而这里的 top 本质还是索引值,而非内存或外存地址单元名称代号,只是因为它发挥了类似指针"寻址"的作用,因此将其看作为"索引型"的指针,数组"堆栈"相比链表"堆栈",数组"堆栈"中的 top 是索引,而链表"堆栈"中的 top 才是真正的指针,因此数组的索引本质上是一种线性循序查找,而链表"堆栈"的 top 才是指针,分散在内存空间非线性不密集,只能由指针指定查找,这里就比较了数组与链表的本质(数组是内存中一段紧密的空间块,各个空间单元的连接是线性紧密的,这是对数组的低层讨论,反应到数组中就是其中的各个元素,将上一个元素的索引加 1 就可定向到下一个元素的索引位置,将上一个元素的内存空间地址加一个空间单元长度可定向到下一个元素在内存空间的位置,用 0~ms-1 或 1~ms 这样的递增性的索引就可表示数组的各个空间并引用它们,存取其中的空间或元素),有高地址和低地址之分,索引由小到大递增的方向就是内存地址低地到高地的线性递增方向,而堆栈是内存中各个分散的节点数据,各个节点数据就是元素或者更确切地讲,各个节点数据中的内容值,或称数据值而非指针值,就是链表的各个元素,相比起数组的空间单元的"连接",各个链表单元空间,也即节点空间的链表使用一种指定式的数值指定法而非数组采用的循序式的查找定位法,各个无素之间分散的链接由内含在各个元素(节点数据)中的指针字段而非内容字段,数据字段来完成,因此对其每个元素的存取前首先确定元素位置时是不能像数组中循序进行的,而是已被指定的,当前元素的内存位置就在上一个节点数据(元素)的指针字段里,而链表结构里,显然就没有高地与低地这种数组里才有的特征.

另外,要注意堆栈和队列中所说,它们都是有序列表,上面讲到,4.bmp 中各个空间里的"有序数据块"才是确切意义上的"有序列表",即堆栈,队列这二个词语作为概念所指的实体所在,那么,它们的有序性是靠什么来体现的呢?堆栈靠的是游离于 0~ms-1 之间的索引值 top,那么堆栈就是指 0 到 top 的数据块,top 有一个特殊情况,top 也可等于-1,显然,此时它指向序列为 0 的空间的更低地址的一个空间,表示堆栈为空(即堆栈中没有元素再供出栈了,出栈<=>从栈中输出一个元素<=>从堆栈中释放一个元素<=>删除一个元素),前面谈到,出口和 top 的引入都为用一个标准数组变为"堆栈"数组提供了可能,堆栈数据块是出入有序的,因此称为有序列表,那么,堆栈是如何依靠 top 来实现其空间里数据块元素出入的有序性的呢?在堆栈里,出口是唯一的数据输入输出(压入即输入 push)通道,而队列有 2 个数据出入口,准确的说法是一个出口,一个入口,而堆栈的出口和入口集中在数组空间的一端而已,示意图就是只能从数组空间的高地址方向端输入输出数据(而堆栈有 2 端前端后端或称头端尾端,即 front,end 与 head,rear),从 rear 端入栈,从 f 端出栈,堆栈数据实体就是从 f 到 r 的数据块,在数组表示的图示中 f 在下,r 在上,在链表表堆栈中,f 在前,r 在后(f 此时也可称为头,r 也可称作后端),在数组表示中,各个数据实体是各个数组空间里的内容

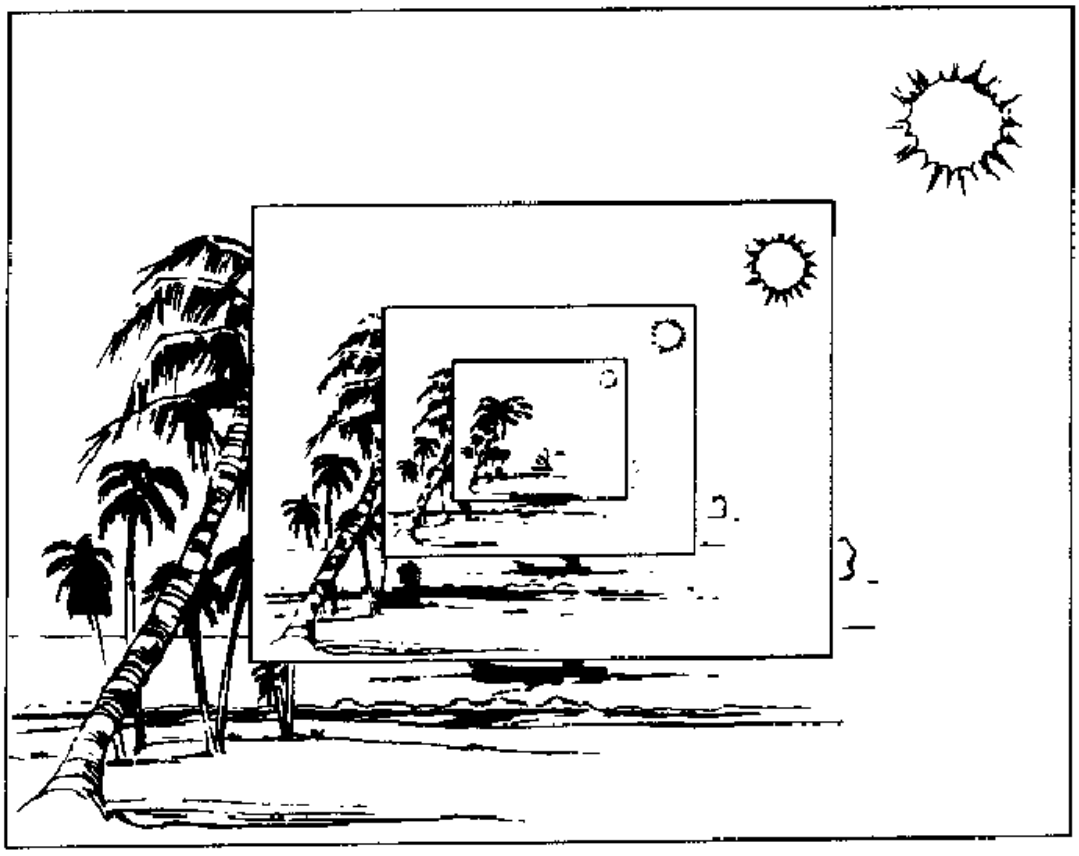
值,而在链表表堆栈中,各个数据实体(f 到 r),堆栈就是各个节点的内容字段链接而成的,这些内容字段链接起来构成的一串数据值就是堆栈数据块实体,从 f 到 r,可见 f 到 r 不是从低地到高地,也不是从高地到低地,因为链表整个空间是分散于内存中的"节点空间"链接起来的,各个节点空间是分散布列在内存中的,因此,各个节点空间的 value 字段也是分散于内存里的,链表的各个空间间实际上没有一条一条的链,这只是示意图中的形象表示,链表的各个空间的链接桥梁就是内含在各个节点空间中的指针字段,链表与数组的一个重要区别与优点就是链表使用 指针而非索引来寻址,这样就可以通过改变指针的值来重新形成链表空间或增删元素(实际上也是重新形成链表空间),这样链表就是一种动态配置的空间,而数组就是一种静态配置的空间,数组中的每个空间在数组被定义后就存在了,其中的任一个空间都不能被增删,只能向其赋值内容值 0,表示置空此空间单元,这就是静态配置的空间.
图片在我的 Q 空间的像册里.

数据结构浅探・其它
评论(0)发表时间：2006 年 6 月 2 日 0 时 53 分

7.21 真正的递归

递归在整个数据结构中和数据结构之外都频频出现。
递归是一种思想，只要问题本身满足递归你才能,递归涉及到三个概念，回溯
递归定义
即用递归来定义一些离散结构(集合，函数),3.4 递归算法，用递归思想来解决问题的一般化步骤(算法即解决特定问题的一般化步骤，”停机问题”证明不存在解决所有问题的算法)
这种关系就如同二叉树的定义和查找，因为二叉权的定义就是指明它的左右节点存在次序关系，所以可以用这种定义作为思想来对二叉树进行查找，这并不难理解
用对象本身来定义对象就是递归，，这是递归的描述性定义，，是不确定的，，，非形式的，集合的定义和算法的定义只有在形式语言里面才有形式定义，，(特别是图灵机证明不存在所有问题的一般化算法时用到的集合和算法的概念)

递归是用自身来定义自身这种说法成立吗，先来看一个问题



你能描述以上一副画吗？？（如果它的中心是无限循环的）

你可以这样描述，，，一副画的中心区域的内容是它自身(也是一副画，而且具有跟前面描述的一样的性质)，这样一副画就是如上的画的定义

显然递归的说法在这个例子是成立的

序列函数的定义比较容易理解，，集合的递归定义常常用来产生一些合式公式

迭代与递归在不同的情况下各有其优势

兔子和斐波那契数

例 4 兔子和斐波那契数 考虑如下问题，一对刚出生的兔子(一公一母)被放到岛上，每对兔子出生后两个月后才开始繁殖后代，如下表所示，在出生两个月后，每对兔子在每个月都将繁殖一对新的兔子，假定兔子不会死去，找出 n 个月后关于岛上兔子对数的递推关系。

月份	已有的对数	新生的对数	总对数
1	1	0	1
2	1	0	1
3	1	1	2
4	2	1	3
5	3	2	5
6	5	3	8

如何理解该月新生兔子即为距该月二个月前的兔子总对数?如 6 月新生的兔子为 4 月总兔数,4 月新生的兔子为 2 月总兔数， 3 月新生兔数即为 1 月总兔数？

对某个月的讨论要追到与它的前二个月的情况(题目意思如此:每对兔子出生后二月才开始生育),这里是某个月的新生兔子与它二月前兔子总数 “相等”的情况(注意这是一种递推说法,在任意差为二的月份间都存在这种联系,比如 3-1,4-2,5-3,6-4，而 6-2 则不能考虑,因为它超出了递推为 2 阶的阶数 2),从第三个月开始，放上岛的第一对兔子 A+A-生下了一对兔子 B+B-(假设每生下来的一对兔子都是一对公母可生育),在 B+B-被生育下来的这个三月份，B+B-并不生育，这对 A+A-在第四个月继续生育 C+C-,而 B+B-在第四个月也不生育，第五个月 A+A-继续生育， B+B-终于开始生育出一对兔子 D+D-,,

上面描述的可以作为典型,因为从第三个月开始，它就满足“每对兔子出生后二个月才开始繁殖后代”“任意出生在 a 月份的兔子在二个月后的 b 月后才生育,b-a=2”的题目要求,作为典型的 3-1,5-3 的情况被提出，后来处理任意相差二个月的兔子情况都可参照以上的描述了.

这样问题便被缩小到相邻二个月之间的情况，,

(为什么这是对的呢，作为典型的 3-1,5-3 为什么可代表一切相邻二月的情况,上面说了,这是题目意思告诉我们的,而不是递推， n 个月后的兔子跟 n-1 总数与 n-2 总数才是递推,,这个递推也是在假设不知道存在这个递推的情况下列出的一个式子，列出之后才发现它是一个线性组合的递归,列出这个式子之间设了 an,这并不表明事先就知道 an 一定是个满足某种递推的通项,只是在列出式子之后才明白是一个递推,而列出式子的过程仅仅依赖于题目意思)

既然已经得知,5 月兔子与 3 月兔子间存在联系，那么这种联系是什么呢？重要的是知道这个联系本身,这个联系可用于任意相邻二月之间(题目意思)

从以上 3-1,5-3 的描述中容易看出,

汉诺塔问题：

例 5　汉诺塔。19 世纪后期一个著名的游戏叫做汉诺塔，它是由安装在一个板上的 3 根杆子和若干大小不同的盘子构成。开始时，这些盘子按照大小的次序放在第一根柱子上，使得大盘子在底下（如图 5-2 所示）。游戏的规则是：每一次把 1 个盘子从一根柱子移动到另一根柱子，但是不允许这个盘子放在比它小的盘子上面。游戏的目标是把所有的盘子按照大小的次序都放到第二根柱子上，并且将最大的盘子放在底部。

图 1　初始状态

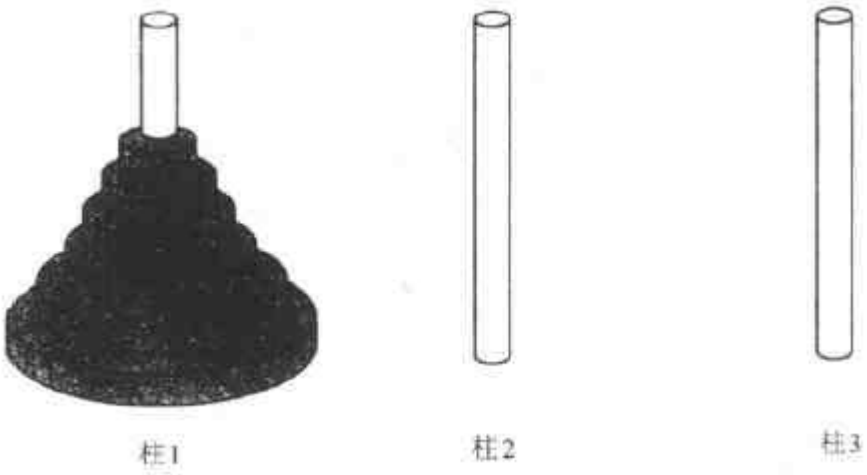
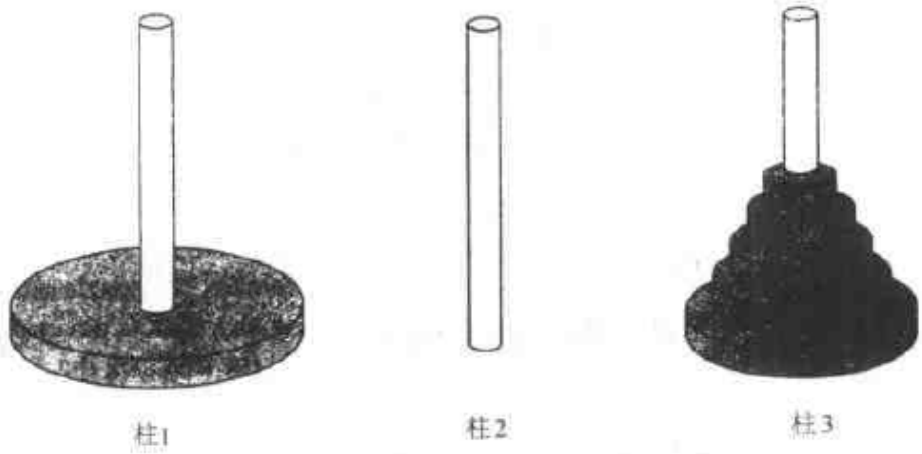


图 2　柱 1 n-1 个盘移到柱 3 后的情形



我们的目标是计数移动步数，，，并不是如题目所说得出移动的具体方法和每个书面步骤因为等我们得出步数这个结论后，就会发现如果具体去得出移动步骤会多傻

而且，尊照规则(一次只能移动一块盘子，最后全部原样移动到柱 2 而不是柱 3,并且最重要的大在下小在上)把 n 个盘子成功转移,移动的方法也可以千千万万，,所以我们要求的移动步数是最少的移动步数,那么对这个问题(求出最少移动步数)该如何建模呢?

首先，我们设想这样一个步骤:第一步,把柱 1 的 n-1 个盘子移到柱 3,保持小在上大在下

的顺序(如图 2 所示),保留最下面一个底盘，第二步，把底盘移到柱 2，第三步，再把从柱 1 移过来的 n-1 个盘按步骤 1 的方法和顺序移动柱 2,至此完成(注意方法和顺序的说法，方法:步骤 1 是怎么样移盘的这次也怎么移,顺序:保持小在上大在下)

容易看出，使用更小的步数是不可能求解这个难题的

下面具体建模
按照上面设想的所产生最少移动步数的移法,首先，设 Sn 是把 n 个盘子从一根柱子移动到另一根柱子的需要的总步数,(问题是什么我们就设什么,,虽然我们并不知道这其实是一个为了满足递推关系的设法,虽然我们也不知道有了以上设法,再用递推关系就可以很好地解此题,因为从形式来看 Sn 像一个序列的通项,它指明 Sn 就是把 n 个盘子从柱 1 到柱 3 所需要的 Step,S2 就是 2 个,S10 就是 10 所需要的 Step,这个设法假定 Sn 与 n 之间存在序列关系),按照步骤 1，移动次数可用 Sn-1 来表示,步骤 2 可用 1 来表示，步骤 3 亦为 Sn-1,故有
Sn=2Sn-1+1　　(Sn 为把 n 盘从柱 1 移到柱 3 所需步数或直接就是)
注意 Sn 为什么不直接说是“设 Sn 是把 n 盘从柱 1 移到柱 2 所需步数”呢,这样说当然没有错，不过为了严紧性和通用性考虑还是这样设(本题当然是从柱 1 移到柱 3,如果没有规定其实移到柱 2 作为中转也可,最后求移动到柱 3 上的步数同样满足 Sn=2Sn-1+1,,况且还有很多同类的题目，移奶酪到盘子里什么的,所以为了通用性考虑还是如上设)

最后的问题：我们用迭代方法来求解这个递推关系

5.2 求解递推关系

有一类重要的递推关系可以用一种系统的方法明确地求解,,在这种递推关系中（好像我们见过的都是这样的)，序列的项由它前面的线性组合来表示

请注意,递归，递推，迭代在措词上的区别

递归是用自身定义自身的,或过程调用自身，用在序列通项表示上,，形如，an= an-1,用在过程体内，这仅仅是一种思想，而不能说是一种算法,，显然这种思想是成立的,,,递推关系和迭代是用了递归思想的 2 个概念而已,是用到了递归思想的所有东西的集合中的二个元素,
要对它们三者定性的话，递归是一种思想，递推只是一种说法，迭代是一种算法(常用来求解满足递推关系问题)

递推是一个序列的某个通项可由它的前几项组合推导而来的关系,，强调的是通项和推出它的前几项之间的关系满足这个“递归推导而出”说法,因此它一定用到了递归的思想,满足递归

序列中,递推是发生在某个项和它的相领项之间的关系,,但是指定了初始项和这个递推关系,可以得出所有的项,即通项,,实际上“某个项”的定义一旦被提出,它就表示了通项的意义,,因此序列所有项=由通项关系得出每个项后,这些项的组合=初始项+满足递推的相领的某几项

(我们设这里是二项,而不是前几项)所以,所谓递归只是二个项之间的关系,,但是由于这二个项可以是任意项,,因此,递归是所有项中任意二相领项的关系,可用来求解整个序列每一项,求此也求解所有项

定义 X : 一切序列可以用到递推要求它满足递归,这永远是前提(实际上并不是所有的序列都满足递推和递归)

迭代是用迭次代换,(常用来求解一个“线性组合”型递推关系的方法,也可求解不满足线性组合的一些递推关系)把一个项逐次展开,,用到一个迭代器 i,比如求解 an,必须使 i 从第 n 项到第 k 项逐次递减,(k 通常情况下是 1,即第一项),,迭代也只在几个相邻项之间进行,因为它也借用了递归和递推思想而已,这一切都是因为用到迭代的序列(或其它问题)也要由定义 x 而来,,不再赘述

.

7.22 树与单链表,图

除了单链表之外还有高级链表,包括双链表和循环链表,但是它们的基础依然是单链表,对单链表的讨论可运用于高级链表,也即无论单,高级链表,本质都是一样的数据结构,都是链表,即链表的本质是节点的单向或双向串连,注意"串连",这是所有链表区别于树这种数据结构的所在,因为树是一种节点的"分支"链接,(树是较线性表,图更为复杂的数据结构,它的任何二个结点间都可以发生联系)它们的共同点就是:这些节点都是分散于内存中的节点,由上一个节点(链表)或父的一方节点(树)的指段字段指向,树和链表的示意图中,圆形就代表节点,包含数据字段和指向下一个节点的指针字段,这是对链表来说的,而对于树来说(这里说的是二叉树)圆形节点就包含数据字段和指向其左右节点的指针字段,或称左右子树,因为这些左右子树是以这个节点为根的子树,注意左右是有顺序性的,不能颠倒,用 Left,Right 区别.无论是链表还是树示意图中的直线都是指针,有方向的,不是互逆的,这处互逆性决定了对单链表的"首->尾"遍历和对二叉树的"二叉查找"遍历方式(中序,左右序)的单向性,可见树与链表的实体数据都存在节点中,这是节点的主体存储空间,链表,树作为数据结构用于组织程序中要用到的数据,它们的节点单元的 Data 字段发挥的是主体存储作用,而它们节点的指针字段是辅助性且必要的,用于树,链表的查找,遍历,插入,删除等操作,严格来说,图并非一种数据结构,书中对图的讨论只能称为对"与线长,面积大小"无关的点线面的拓朴讨论,而由点边组成的图形中居然没有一个点或边发挥数据存储作用,而这是一种数据结构首要完成的任务.

3.数组,链表为什么能仿真堆栈?

其实这个问题被提出来一点都不可笑,对这个问题的讨论很有意义,它能让我们明白一些细微而重要的东西.
数组,链表,堆栈,队列,是四大数据结构,在关系是并列的,为什么又有"用数组和链表仿真堆 栈"之说呢,其实数组,链表是较高级的复杂的数据结构,还记得本书序言中的一句话吗?数据结构是人类在长期的编程过程中总结归纳出来的一套科学有序的方法,数组是,链表是,堆栈和队列也是,但是在人类总结归纳这些数据结构并形成术语进而形成数据结构这门计算机科学时,永远是从简单低级的数组和链表开始的,对它的讨论和总结归纳先于堆栈,队列之前进行,当人们总结出数组和链表数据结构时,并得出对数组和链表的删除,复制,插入等操作后,形成了数组链表等数据结构概念及其操作的一整套理论后,人们进一步探索数据结构,发现了堆栈的存在,而堆栈又是基于数组,链表的高一级的数据结构,对堆栈,队列的研究与讨论经历了与对数组链表相同的过程,对数组和链表的研究和技术总结业已成型的情况下,就应该采用现在的知识来描述新出现的知识,因此有以上的说法.

7.23 树

树是一种数据结构,称为树状结构,简称树,树的本质是一个或多个节点的有限集合,树只有一个节点的情况是例外的特殊的情况但也是允许的树的形式,而一般情况下,树都有不止一个的节点,有限集合这四个字指明:一棵树,无论它由多少个节点构成,这些节点的数量都是有限的,可以计数的,也即,从来没有一棵树,它的节点个数为无限不确定的.
在一棵树的所有节点中,它们的地位是不一样的,每棵树必定有一个特定的节点,称为根节点(root),根节点与这棵树的其它节点构成了这棵树(当然这些节点并不是单独地存在无联系的,不然就无法从这些节点中搞出一个为根的地位节点,这些就构不成一棵树),可见,"根"是相对于"树"来说的,根这个概念是树一级的概念,只要有树,便会有根,根的本质是一棵树的"特定节点"(第一个节点,其它节点由它分支而来),而树又是一种特定的数据结构,请记住,根是依附在树概念上的概念,脱离了树便无所谓根概念,总之,根是与树直接挂钩的一对概念,为什么我要在这里这么花心思地说明根是相对于树的概念呢?因为这是理解 7.1 节中关于树的其它概念的一个基本点,掌握了它,你便不会在理解这些概念中迷失.
说完了根,再来说其它节点,其它节点(实际上这里说的其它节点准确的意义不是说除了根节点外的一棵树的其它所有节点,而是说根节点的下一级节点,可以有 0 个,可以有 n 个,0 个就是没有下一级节点,而 n 的大小便决定了这棵树的分类性质,如果 n=2,便是二叉树,后面会谈到)是根的子节点.(8.bmp)
从树结构的图示来看,树与现实中的树虽然类似,但是,数据结构中的树是一棵倒树,根是相对于树的概念与树直接挂钩,而子节点的概念直接与根节点的概念挂钩,而与树不挂钩,根节点与子节点的概念仅仅存在于一个节点与该节点下一级节点之间,与树不挂钩,这就像主程序与子程序的概念,主程序与子程序永远是相对的概念,如果前面说的子程序它又有它的下一级程序,那么主程序与子程序有主子关系之外,还可以说前面谈到的子程序与

该子程序调用的子程序也有主子关系,此时这里的子程序是主程序而子程序调用的子程序就是子程序,这 2 种说法都是可行的,因为主子关系是相对的可变的而非绝对的,主子这种说法存在于只要满足一方是调用者而另一方是被调用者之间,而非绝对不变的,再接着上图讲,那么 B,C,D,M 就是 A 的子节点,而不能说 B,C,D,M 是树 T 的子节点,没有这种说法,树只有根和子树,叶节点与其挂钩,而只能说(因为 B 作为根节点其下还有 P,Q2 个子节点)B,C 是树 T 的子树(D,M 不是),这里,B,C,D,M 作为 A 的子节点,同时 B,C 又是树 T 的子树,子树 B(以它的根节点为名称故根节点就是 B)的根节点 B 又有自己的子节点,T 的树 C 的根节点 C 又有自己的子节点 R,若一棵树中的任意一级(根)节点最多有 n 个子节点,则称这样的树为 n 元树,二叉树的得名也来源于此.

12.树的本质是什么?

树是一种数据结构,所以树的本质是一种组织内存空间的方法,就像数组是静态配置内存空间的方法,且数组配置出来的空间不但是静态的,而且是紧密相连排列的线性的各个"小空间",是一块内存中的块状数据静态的存储区,由它的一个小空间的地址可以推导出下一个小空间在内存中的地址,而链表是一种动态的配置内存的方法,程序中,数组空间在数组被定义出来时就被开辟,在它的生命期内从此不能更改大小,而链表被定义出来时还要用内存开辟函数 malloc()来实际分配内存,所以它开辟出来的空间是动态的,而且这些空间单元不是线性紧密排列的而是分散的,分散于内存中的各个"节点"空间,是不可通过索引下标循序查找定位的,只能通过内含于各个节点数据内的指针字段来"指定查找",形象示意如下:(1.bmp)那么树呢?树的本质是"节点的非空有限集合",跟链表有一定的相似性,首先,链表与树(基础树,书中所讲为二叉树,这属于简单树而非广度意义上的树,但是对二叉树的讨论可以延用扩展为整个树的范畴)都是动态内存配置的方式,链表空间间连接依赖于各个节点数据的指针字段,而树的节点空间连接的方法为"格式上的约定",树的节点空间和链表的节点空间都是这二种数据结构实际存储实体数据的存储场所,是怎么样一种"格式上的约定"呢?以二叉树为例(其实树各个空间之间没有什么链接,链表各空间发生关系的纽带与桥梁是指针字段,而树的各个节点空间间发生关系的钮带与桥梁为"格式上的约定","约定俗成的方法来格定"),二叉树的每一个节点空间实际各各分散布列于内存中,它们发生联系的手段就在于各个节点的性质,前面谈到,树是节点的非空有限集,如根就是树的第一个节点空间,如果是二叉树,这个节点自然会跟其下一阶的二个子节点发生关系,这就是父子节点关系,当然不止根与其下一阶节点,在其它节点间也存在父子关系,除此之外还有兄弟关系,树->子树关系这种"关系机制"是远远不同于链表依赖指针表示各节点空间联系的手段的,所以这是一种"节点关系"上的"约定",是一种全新的组织节点空间的手段和方法,而没有用到指针,当然可以用指针来模拟和仿真,这就涉及到"用链表来表示二叉树"的知识点,这是后来的内容.

13.图的本质是什么?

要说图是一种数据结构实在很难理解,因为我第一次碰到图的概念,根本就没有发现图形结构中的哪部分用于存储数据,378 页对图形的定义"在图形 G 中包含了 2 个集合,一个是

由顶点所构成的有限非空集,另一个是由边所构成的有限非空集,可用 G(V,E) 表示",按照这个定义,图形就是顶点和边的有限集合(至少要有一顶点一边),晕死,那么实体数据存在哪?顶点中?不是,边的描述值权吗?好像也不是,署假我只看了这本书 17 天,所以对图我没太多深入研究.

7.24 真正的散列表

Hash 表就是 hashed list,,,它是逻辑上的 list(所以也有 hash map,hash set 等),但按 hash 方式存储地址作为存储,

Hash 是一种利于搜索的启发过程,一般的搜索是对搜索空间 uniform 的,而 hash function 是对搜索空间的目标问题建立的一种启发机制的函数。

Hash 的最重要的意思不是提供一个映射函数,那反而是 hash 解决的第二个问题,即地址问题,,它最重要的意义,即第一个解决的问题是基于统计的本质。进行的对搜索空间的一种抽象(即 hashtable 而不是 hashfunction 问题,比如元素会出现一次,还是二次,这样抽象对操作元素有好处,但显然此时并不出现 hasing function 的意思),

一般设计散列时,说的都是设计散列表,然后处理冲突处理,,,如何散列所用的函数是附带问题。

也即其实 hash table 的第一层意思是 table,,即形成数据结构才是他的第一层意义,,,而不是如何 hash,,,并提供一套 hash 机制,,,

它着手于"在解空间和目标数据空间建立一套具有 inform 关系"的数据结构,,,这才是它的第一层意思,即名字中的 table 一词

至于如何 hash 并解决冲突,,那反而是它解决的第二个问题..即名字中的 hash 一词

只有理解了这点之后,你才会明白 hash table 是如何来的,以及为什么存在,,与其它数据结构作比较时所呈现的那些不一样的特点(比如为什么要进行 hash,,为什么要处理冲突,而其它的数据结构则不需要这样的分析过程).下面我们来详细解释。

首先,对于一个 hash 数据结构来说,它首先必须解决基于统计的本质把数据放到一个可供常数时间内索引到的地方,即解决这个数据结构的形成问题是首先要进行的,至于 hash 的方法以及在每一种方法下解决冲突的途径,并不是唯一的,而是可以慢慢借助不同的方法来实现的,而显然,这两个问题并不是毫无关系的,怎么样形成一个散列数据结构,总要后来涉及到如何散列的问题及如何解决冲突的问题。

对于第一个问题，所谓散列数据结构的形成，总是会涉及到两个动态的因素，即待散列的数据(的变化)和散列数据结构本身大小的变化，这两者的比值就是负载系数，它揭式了一种什么道理呢？即：如果数据量大于后者，我们可以重调整数据结构大小获得新的系数。

这样我们就讨论了数据结构的问题，下面再来讨论这种数据结构下的操作及算法效率(别忘了，这是一本数据结构方面的书导出一种特定数据结构的一般步骤)，这之前我们先来讨论 hash 的第二个问题。

对于第二个问题，我们可以使用线性的方法，(从这里开始慢慢出现对这种数据结构效率的讨论了，注意，散列是一种特别的数据结构，所以它是如何 hash 的也要算进效率方面的考虑，而我们其实知道，多亏这个如何散列动作在效率方面的 overhead，散列所有的操作是常数时间)这种方法的优点是 hash 的方法简单，但会不可避免地出现一个主集团的问题。

7.25 快速排序思想

交换排序是置换顺序，插入排序更理应被称为交换排序。

快排是一种交换排序，这种交换发生在待排序列本身之上，首先，将序列按中间位置分为二部分，随便从整个序列中取一个临时成员，这个成员就是“待排”的对象，这个待排对象完成它的一次排序后，那么整个快排也就完成了，因为它最终到了它应该被排在的位置。

取出这个待排元素后，以中间位置为基准，分别以从左趋向中间，从右趋向中间的顺序找二个对象与这个待排对象比较，,，这所谓的二个对象是满足条件的(从左趋向中间找的元素要大于待排对象，从右向中间找的元素要小于待排对象，首先是从右向中间找，然后才是。。)，而且可能有多个（显然在二边可能会有多个大于或小于待排对象的值）,，所以这个过程要持续几次,，这个待排对象才能最终到他应该去的位置。

那么这是一种什么样的“交换”排序呢，我们知道待排元素所在位置空出后,，从右边找的值填充（右端第一个大于待排对象的值）这个位置后，那么这个值所在的位置我们置空它，到这里为止，待快排对象原来所处的位置被填充,，只是它的值尚未被放入一个确定位置，而且，这里又空出来一个位置,，就是在右端比较的那个值填充待排对象原来空位后空出来的位置。

现在在左端进行与待排对象比较的过程，同样在找到第一个小于待排对象的对象后，将这个左端这个空出来的位置空出来，将这个值填入上面右端比较时空出来的位，那么到

现在为止，第一套左右值被找出来了，而且到现在为止，待排对象没有被置入一个位置（因为只是比较了第一套左右值），而且左端又空出来一个位置(但是此时还不是结束快排的机会，因为还存在其它的左右值对可供与待排对象比较并产生新的空位—左边或右边的)。。、

那么在进行第二套，最后一套左右值比较时，一定最终会多出一个左空位或右空位，此时将待排对象置入，，就算完成了快排。

7.26 算法设计方法

摊还分析不仅是解释数据结构性能的工具，而且也是设计时要考虑的因素。就跟 NP 完全一样（逊于图灵不可计算机的停机问题）。如果复杂度超过了 10 的确良 8 次方这个计算机的数量级就没有意义了。

存在很多类型的问题，比如最优化问题，，找点问题，理解诸如贪婪算法的前提是理解这些问题在先，比如最优化问题可以很好地解释什么是贪婪设计。。

迭代是方程求根的一种方法，并且它也体现了一种算法设计方法。

递归体现了分治的算法设计思想，但它提出的子问题一定要跟原问题接口一致。。 递归的终结条件是不需要递归也可直接求解的条件，，因此是终止条件，，当以从下到上的眼光来看时，它是超始条件（直到问题规模 n）
其实并不是只有所有明显递归性质的问题才可以用递归来解答，而是只有能把原问题分解为子问题并能制造一个接口的情况下就可以利用递归来求解它。

递归是从上而下，所以有时要求用辅助栈来保存中间结果，而递推只有一个函数，并不发生欠套的函数调用，并没有出入栈的时空开销。复杂的递归结构转化成递推时，需要回 SU 处理，二个概念仅一字之差，一个归，是向下，一个推是向上，

第 8 章 代码抽象之高级语法机制(C++,Python)

8.1 真正的 OO

初识 OO,我不过认为那是一种编程语言支持的工具，，真正稍微懂得它时，我发现我

走入另一个迷惑，一个更深的迷惑，，如果 OO 是一种思想，，，所以我要怎么用语言去联系并理解它？

而其实，正如我从书一直写到这里所秉承的，只有从抽象角度去理解一门语言机制，只有认识 OO 的程度到了把它跟抽象拿来讨论，你才能真正开始理解了 OO。

比如为什么会有 OO 的出现呢？首先，最初级的说法，因为 OO 是对现实世界“Object(物体)”的抽象(不可否认我们周围的世界的确是一个一个的对象，注意这是用 OO 眼光看待问题域),我们可以在抽象的基础上构建抽象，进而发展出大型的系统（由土和石到房子，由房子到，不可否认我们一直在做这些的事和思想上的活动），而表现在 OO 编程工具上，我们用 Class 来表示一个 Objects(注意这是应用领域，虽然这种 Class 对于表达现实的确显得有点单位过小-----而且是运行时不可控的抽象，但我们可以通过不断地包装抽象和组合抽象，或者通过策略---设计期可控的抽象来达到更大的抽象)，，Class 的出现是历史必然的，以前编码时是分开代码和数据的，整个代码被硬生生分成了代码和数据二部分，一个 CLASS 就是对于代码和数据的整合抽象(这样就需要只面对一种代码模式了)

其次，面向对象的确带来了效益，面向对象并不是一种凭空冒出来没有根据的概念，相反，，没有它的存在倒是很令人不自然的，面向对象和基于面向对象的设计模式使软件设计进入了一种高速发展的状况，甚至个人独立开发体系复杂的软件都成可能，这就是面向对象的作用之一所用所在，，，因为本质上，面向对象是符合现实生活的，，，那就是：正如上面一段所讲，面向对象特别类似于建筑学的过程，需要不同的原料并组合它们才能构成最终的软件或建筑，我们只要发明了砖和水泥这样的原料，就有可能发明楼房，但在不必首先一开始就得造出一大幢房子的需求下，我总可以慢慢积累砖吧，否则，一切细节都在开始被考虑，这不太现实(往往有时候，，，开发一个程序是一个真正工程规模级的活动，不是指那种声明几个类，创建几个对象就完事的开发过程，，而是指那种预计到未来扩展的需要而预留了很多发展余地的大型开发过程)

那么，OO 的类如果是一种新的代码模式，数据类型，那么它是较原来简单类型下什么样的增强类型呢？

Class 可用来派生子类型也可用来产生对象(我们用措词上的区别来区分这二者),派生的子类型 subclass,此时它也是一个 class,，产生的 object,我们称一个对象为 class 的 instance.class 为 object model

这其中存在一些特例，比如纯抽象的 class(用来实现 C++版本的 interface),,或者静态类，，

为了产生逻辑，我们可以继承，也可以组合，继承就是从相对基类中(除了单根继续的语言，基类都是严格相对的)单层继承或多层继续，产生新的类，这样的子类在抽象上

继续了父类的所有属性，在底层实现上其实是叠加了子类的代码在基类上的组合代码，是发生在类间的逻辑。也可以由类产生对象以产生逻辑，是发生在类和对象之间的逻辑，，，注意多层继续在现代 OO 中是不受鼓励的。。

而发生在对象之间的逻辑呢，是采用将一个对象作为另一个对象的数据成员来作为产生逻辑的方式，，对象之间交互的方式就称为消息，C 开发的函数范式在这里被消息所代替，实际上我们知道消息也是函数调用，只不过这里的函数是加了访问控制和多态的特殊性函数。。

比起 OO 语言之前的类型来说，在数据类型的眼光上看，有二大改变，1，以前的简单类型现在可以杂合数据与代码，所以是一种既不能称为数据或代码的 ADT 或 UDT，2，类与类之间可以发生联系，这种 ADT 与 UDT 通过组合或继承来构造整个系统，程序员之间只需共享对于这些 ADT 的理解。

什么是系统？系统是关于一个事物的逻辑上的有机组成部分，因此系统分析师的工作就是用面向对象去表达系统(划分对象)，对象之间是分配了责任的，如何把这些责任更好地分配呢？(即对象内逻辑和对象间的交互逻辑，UML 中的对象消息)这就是 OO 用来进行设计的意思。

OO 化并不强求以靠近现实的模型来设计应用,而更多地是利用 OO 的访问控制以同一种方式看待代码与数据(设计代码，定义数据)

面向对象语言提出的 OO,,,一方面既是抽象机制,,,一方面也是接口机制..

如何才能用 OO 进行科学的设计呢？这要求有很好地用类来表达抽象或实体的能力，一个分配得好的对象组合(兼顾了当前使用和未来扩展能力)往往不是现实生活一个具体的模型，而且有些时候你一点现实生活中的对应物影子都找不到，而往往是一个思想模型

基于 OO 之上的设计模式的学习也是很重要的，对设计模式的学习是无比重要的，只有掌握了设计模式，才能不会让思想局限于现有的代码，才会在一个更广泛的领域里去拆解对象并进行设计(原语领域去看待事物,因此程序设计实际上是一个反映你心智和哲学水平的活动)，往往有些时候，，，真正的再编程能力不再是语言细节和问题细节，，而是在掌握了语言细节和系统细节之上的高层构架的学习(设计上的学习).

先来看看面向对象几种层次抽象

我们知道在 C++实现的 OO 机制中，存在 class,class=type,即 template 中的 type,但是我们知道 template 更像是一种建立在 C++类型机制上的脚本语言，其语法古怪，不接近 C++本身，，C++用 class 作为派生 type 的类型，用 type 作为 template 中的类型。

而设计上的学习往往要求你掌握关于此问题的所有细节（在目前的科学技术水平下对该领域的现解），和与此问题有关的很多周边问题，所以要从原语领域去看待此问题和进行基于对象拆解此事物并构造这些对象之间的逻辑的系统活动，

8.2 抽象眼光看 OO

抽象使高层置为顶端，面向对象就是一种抽象，这种抽象使我们看不到我们不想用到的事物的一些方面，而把那些我们能用到的事物的方面用来作为描述此对象的全部，，（即抽象了的对象根本不能完整地反映对象本身而且压根就不能），接口是关于如何应用对象提供的服务的全部抽象，如果说面向对象和接口是代码级复用的机制，那么构件是二进制级真正的复用机制，接口把一个系统的可用部分按不同的形式透露给复用者，

抽象使高层置为顶端，而它的可定制部分都集中在顶端作为应用，这有点像网络的七层模型，只有定义了一个抽象而且是合理的抽象，才能为未来的应用预留扩展空间

一个 OO 程序本质是一些类数据(jar 文件包里全是 class 文件)，类是真正意义上的数据（类型），然而在 JAVA 中，JAVA 提供了数值类型对象的对象封装型和非对象封装型，(因为对所有的东西进行对象封装有时不符合现实而且对象机制一定程序上比面向过程慢--这就是 C 向 C++转型中有人担心 C++其实由于它的封装技术而导致的速度问题，你可以把整型看作一个对象但是传统观念里已经把它作为数值看待，一般程序设计语言都将数据类型的数据作为 first class 来看待，就是那种 Java 库 src 包内的那些对象，这些对象是高级对象，因为程序设计语言一般为他们装备了足够多的功能和运算符，可以由它们派生很多其它的数据类型)

面向对象数据库 db4o 的出现，使我们开发的重心不再集中在如何存储和声明数据本身，转而将注意力集中在这些业已创建的对象的逻辑上,如果说 UML 建模工具可以用来规划这个过程，那么 DB4o 就是实际产生和持久存储了这些对象，

8.3 真正的对象

在当初 DOS 下用汇编开发程序的时候，一般都是先考虑这个程序会用到什么数据，要用什么样的数据结构在程序内部去表示它，然后才考虑开始写代码，，这个时候数据与代码是紧密相连的，再后来出现了模块化但是面向过程编程，这个时候，利用临时自动变量可以降低子程序间的藉合度，而当出现面向对象后，我们完全可以先定义一个预对象（什么是预呢，就是说这个类被写出来的那一瞬那，它并不像面向过程子程序一样存

在某个一个 rotuine 执行路径中，而是作为预定义的一个程序组件，除非你定义并引用了一个该类的实例，这个以前定义的类中的代码才会进入某个执行路径），即类，一个类是对象是预定对象变量与对象实例(注意这二个概念，变量可以是一个指针，是声明性的并不分配内存，而对象实例是一块业已经过 new 分配的内存块，实例也可以是一个句柄,是指向对象实例的指针，因此它是指针的指针，形如 VOID**或 SOMEOBJ**,指针与句柄的区别即来源于此)的对象模板，而类模板是预定义的类模板，类这个东东来源于结构体，其实一个结构体也有它们的构造函数，在这种意义下，类与结构体很接近。

面向对象的方法成员广泛上的意义就是“对数据的操作”，也即计算机指令，，狭义上的意思就是“表示对象的主动或被动性作用的方法或函数”，，，面向对象的数据成员就是“数据或变量”，也即计算机要操作的数据，，这对应于面向过程中的函数跟变量，，，而面向对象将它们推入一个更泛化的境界 (一切计算机数据都可成为数据 ,特别是 OO 对象)。

8.4 真正的继承

面向对象的缺点不在 OO 本身，而在于人们对 OO 的滥用，这主要体现在人们对继承的概念的偏解上，设计模式指出，对于复用，我们应该尽量用聚合而不是多层继承（单层继承和对等继承还是允许和鼓励的），JAVA 不支持多重继承，一般也提倡不超过五层以上的继承(然而在策略的提出中，多重继承却起了不可替换的作用)

不可复用问题的出现绝对不是面向对象才有的问题，，面向过程更甚，实际上对象机制的确在一定程序上也促进了模块间的藕合，然而设计良好的对象构架可以很好地为以后的更改预留空间，并且似乎也并没有更好的比面向对象更好的机制出现，比如完全面向复用语言,,极端化地求完全可复用性又是不对的，因为复用这个概念本身完全是一个相对的概念,,,考虑一下 MFC，它内部的各个组件都不能拆开来用，只有在作为 MFC 这个整体的前提下各个组件才有效，在这个意义上它是不可复用的，，，然而，就其内部来说，当你在 MFC 下使用 MFC 编程，你真的可以做到重用其中的一个组件可以避开不使到另外一个组件（这就是说，可不可复用从来都有一个最小可复用单元的概念存在），stdafx.h 就允许条件编译和预编译 (stdafx.h 是 VC 给你的标准预编译头，它可以通过 project wizard 为你清理出一套专用于 win32 app,win32 mfc app,win32 console..etc 的文件头,当然你也可通过 project.settings,C++.precomplierheader 来定义自己的预编译头)，条件编译常被用在一些大型库和软件系统的编译上，比如一个由很多可 plugable 的 Dependencie 组件组成的库，常常提供有具体的宏条件编译来让你自定义哪些组件要被编译进来，哪些组件不被编译进来，而每一套编译出来的库都是有效的,,还比如一些用到 include 了 dx 头文件的库，有时为了让没有安装 dx sdk 的用户能正常编译，就提供有宏让自用户选择不 inc 进这些东西。

8.5 真正的 OOP

在一个常见的 C++编译器中，总是会带一个精心安排的简化程序设计的公用库，比如 MFC，VCL 等等，MFC 显得有点过时了而且它的实现比较繁复，C++标准把 STL 加进来了编译标准，一般 C++编译器都支持 STL，

库跟语言的关系是什么呢，，语言可以通过库来扩展，，《C++沉思录》中说“库就是语言”，“语言就是库”，，比如数组是语言内含类型，但是基于面向对象的需要，，就提供了 Vector 类来 wrapper 数组，，这就是库对语言的扩展作用。。(相比来说，如果语言是一种使用规范，那么一个库是为了某个特定领域—比如 OO，而精心设计的，一般要求精细的设计)

OOP 不仅仅再是指单纯的继承啊，多态啊（这样面向对象的机制本身），还指一些公用的 oo 程序设计手段 idioms，，比如容器(用来盛放对象)，迭代器(用来遍历对象)，适配器(统一对象的接口)也即 OOP 包括面向对象和 Generic programe 技术

模板的初级应用是作为容器（异型或同型）来使用的，实际上它还有作为 Generic programe 的很多高级应用，如编译期策略

STL 被称为总数据结构（因为它的元素是数值和对象都可以-异型或同型容器,而且它包含了对于一种数据结构的存储，，访问的方方面面的抽象）
这就是对数据结构的整合(stl)还有策略(可以称得上是对设计模式的整合)
stl 甚至都抽象了多种数据结构的使用方式,stl 被作为数据类型的模板（但是它显然还有更更高级的应用,,作为 container 的模板只是模板的初级应用而已）

这些原语程序设计手段也作为一个库(即 stl 也跟 mfc 一样是 OOP 的一个重要方面，，，当然还有构件库比如 ATL 等等)被提供在一门语言的编译器中，但是.net 就整合了这二种库，，而且它是跟本地脱钩的

8.6 真正的私有，保护和公有

权限修饰出现在二个地方，1,在一个类内部对成员的修饰，，2,用基类派生子类时，对基类作何修饰，，这是修饰派生方式的修饰，，这二个因素共同决定了外界以如何权限组合，，读写一个类的成员（这才是我们最终要访问的目标，但是有二重门要过），当这个相对基类是。

8.7 真正的重载与复写

重载是多态的简单形式，其实对 O O 来说，要谈到重载就要淡到多态，Overroad 与 Override,重载是根据已有事物定义新事物（这个新事物是与已有事物貌合神离的，通俗来说就是在一个具体类中，可以定义多个函数名相同但参数不同的成员函数，前面可以用 Virtual 或没有这个关键字），覆写是将旧事物升级为新事物，因此重载时的旧事物与新事物都有效的而覆写时的旧事物作费(通俗来说，就是在一对具有基类与派生类的二个类中，派生类中可定义一个与基类成员函数名和参数均相同的成员函数，此时基类函数前必有 Virtual 关键字)，诚然覆写可让基类同名函数作费，然而在 C++中还存在其它函数作费规则,这里“作费”是指派生类的函数屏蔽了与其同名的基类函数，规则如下：

（1）如果派生类的函数与基类的函数同名，但是参数不同。此时，不论有无 virtual 关键字，基类的函数将被隐藏（注意别与重载混淆）。

（2）如果派生类的函数与基类的函数同名，并且参数也相同，但是基类函数没有 virtual 关键字。此时，基类的函数被隐藏（注意别与覆盖混淆）。

覆写跟复写不一样，这两者都会导致多态，然而，覆写会导致“父类在调用它的一个成员函数时，实际上在调用子类的一个成员函数，因为它被子类覆盖了”，这也就是 c++著名的“用虚函数实现运行期多态”， 而复写仅会导致简单的“在父类和子类之间存在同名但参数不一的成员函数”这种结果

8.8 真正的构造函数

构造函数不会常常被声明为 virtual,,,因为子类不一定非要实现它自己的构造函数(而往往在它的基类中就有过实现)

基本类型数值是没有构造函数的，而 Java 中对于数值类型的封装形式就有构造函数，这是它们二者本质的不同，像 C++中用 new 动态声明一个对象，它一定用到了这个对象所属类的某个构造函数，可以说 C 与 C++对待程序的“数据”的最大区别就是 C++对他们统一看作为对象 ADT（于是连简单类型都套上个构造函数），而 C 没用到(因此 C 压根不需要构造函数来着)

8.9 OO 的缺点

OO 与函数的比较是 C 与 C++比较的问题。

比起函数式语言来说,因为在一门 OO 语言内它的一等公民不是函数体而是 OO 对象,而且 OO 直接跟我们的设计思想中出现的元素挂钩,因此 OO 成为代码复用和软工时

代最好的语言,(软工这个字眼就指出它是计算机离散倾向人的结合.OO 的提出就是为了解决编程离散形式与人思想中出现的因素的对接,解决复用和开发中人人的交流等问题,所以离开 OO 是不可想象的),但是构建在 OO 之上的设计模式却导致了越来越深的复杂性,这是因为人们没有想到 OO 以后的思想世界和现实世界应用,其实比 OO 这种形式要复杂得多,比计算机内部的离散形式还要复杂得多,换言之,OO 并非一切,相反,在一定程序上提供方便性的同时,在另外一些维度和应用上导致了另外的复杂度,因为思想永远是复杂的嘛(杀鸡用了杀牛的刀)

而函数语言或动态语言却可以绕过设计模式实现同等的功能,因为函数作为 first class 是一种最接近计算机处理的形式(而 OO 同时接近计算机和人的思想,或者更确切地说是接近人的思想,看 OO 以后的 GP,DP 就知道了),,反而可以用这种唯一的形式绕过很多 DP 要花很多劲而函数式语言轻易就能搞定的一些东西.

但是不要认为动态语言就能提代 OO 静态语言(这种说法真可笑,复杂性有它的好处,复杂性同时也是相对的而已,RUBY 在一些领域导致的复杂性比起 JAVA 来说会更多,只是不同情境的问题而已),哲学指出,任何问题都要实事求是,我们需要在问题需要什么类型的语言才去考虑选择什么语言 ,,考虑一门语言优劣时,并非纯粹技术问题,有时是多维的问题,,比如语言的优劣还得用它适用不适用软工这个指标来考虑呢

还比如 JAVA 的动态机制(反射加代理),这种历史复杂性是为了实现动态语言的"由行为决定类型",,,这就是 AOP,,JAVA 居然站在它本身是个静态语言的基础上去做动态语言做的事,造成了很多历史复杂性,讨厌

而且,RUBY 这些宽形式,自由的语言,正是因为太自由了,就像 C 语言一样,太自由,反而不能成为软工的首先,因为软工要解决复用和简单意味着统一的形式的需要,即编程语言的离散形式和人的关系(所幸 RUBY 本身也提供了 OO)..

所以,JAVA 在 WEB 开发中,OO 以后的那些 WEB 框架,是现实所导致的,并非 JAVA 本身导致的,,,改变 J2EE 框架就行了,并不要把 JAVA 改造或加个 JRUBY 就行的,,,相反,我们应更多地解决现实世界的思想模型,而非增大或扩展 JAVA 或 JVM 平台自身,感觉在 JVM 上集成 JRUBY 实在是个蠢的作法,凭空增大了抽象和运行时开销,就像把车拆了放到车库里明天又拼装起来再开一样..

其实以我来看,所有语言都是一种形式,要解决的问题才要弄明白,如果这种形式不能表达更深的问题,那么问题就出现在形式上,,,,OO 最大的特点不是继承,不是多态,而是"OO 接上了计算机离散和人的思想中的因素体,促成了软工时代的出现"

所以,不要去为了技术而探索技术,,,现实世界才是我们要最终解决的问题,而不是那些表达现实的形式语言..语言终究是形式.技术问题的复杂有时是历史原因导致的,,,,细节和理论,形式和应用,,是二个可以并行发展的维度 ...其中现实应用更重要 ...RUBY 声称它能提供编程对于的简洁性,这就是一种很好的作法。

8.10 模板与泛型编程

看过 Bjarne Stroustrup 的 C++简史的人都知道，模板最初是用来取代宏(预编译，条件编译，#define 等)的，这说明它的地位是靠近于编译期写代码时的设计过程的，(虽

然它也一直参加编译以及后来的代码生成，这就是它跟宏不一样的地方)。而且模板也是类型严格的所以也是语法的一部分，既然是语法，就是语言的一部分（而宏函数虽然被冠于函数之名，但其实他不是语句也不是函数，根本不参于编译）。。

模板，是在编译期靠近编译前端那一部分对类型进行控制和抽象的语言机制(即部分地在语法层次上直接写程序,针对编译时的实例化写程序不需等到运行期才检验结果，而且更不必专门针对运行期语意和运行期语言机制设施写程序。比如 C++运行期 00 的多态)，是 C++建立在它的类型机制(特别是 class 通用类型)上的语法机制(所以不光有函数模板，结构模板，还有类模板)，在这个语法机制上。C++可以做很多强大的事情(特别是设计上的，集中体现为泛型编程与元编程),,运用这种编译期的能力我们可以将 C++视为另外一种语言。即编译期以模板为脚本机制的语言(而模板是主要针对处理类型的)。。.

但其实编译非运行，我们知道运行期才能申请到系统资源，才能进行计算机的所谓运算，

其实对编译期和运行期的误解一切的罪原是解释语言跟编译语言的区别（如果没有编译后端，那么运行期的目标就是中间代码即源程序初步变换了的形式而不是目标平台的程序），我们知道，在编译前端完全之后，代码就被生成了，对于解释器来说，编译后端是不必需的，此时它就是逻辑上可运行的一个整体。。即设计逻辑 = 编译结果 = 最终运行逻辑。。只有等编译后端是为具体平台生成代码时，，这个时间才出现第二个运行期,,要打乱设计逻辑，将语法级的编译时的设计逻辑转化为变相的平台逻辑，运行时就可以申请到系统资源了。。(因为类似 C++编译器它的编译部分和运行部分是他大爷的分开的),，这才是运行时的标准定义。。

而模板绝非为后一个运行期而存在，它在中间代码层次上就可以工作了，因此模板中可以不需要分配变量这些运行资源(只要 typedef，提供编译符号给它就可以，当然，写模板逻辑时也可以面向运行期分配变量定义实例，前者导致元编程，后者导致普通运行期的模板编程即泛型编程)，它可以只是关于类型的纯语法逻辑。因此类型也可以仅仅是一个语法占位符(很难想象类型占位符也能参与运算吧)。就跟宏一样。。在写模板时并不产生代码，只有模板逻辑被实际调用时才被生成代码(即编译实例化时，类型将不再是一个占位符，而是获得指明的一个具体类型，可以进入编译了)。。

这对用语言来进行设计尤为有好处。因为他可以站在范类型的角度上进行脱离了运行考虑的编程(泛型，比如可以写出 function type 这样的语言结构，就跟纯虚函数一样，只是一个接口抽象。)，

泛型的好处在于：它以型的前提“泛”，所以兼有静态弱类型和自然语言的双重特点

这说明，设计越来越靠近编译期(甚至是设计期，比如对类型进行 template 化的策

略行为)，而非运行期，等到运行期才去验证设计，，，这种用设计(而且是符合语法的设计)来制导编译和代码生成的过程就像语言中内置了一个 UML 一样。。

而且模板是编译期的，可以控制编译器对逻辑的生成动作(元编程，比如它可以刻意不让某些类型的模板进入编译，你写有关模板的代码时，并不算待编译的一部分，只有那些实际发生作用的类型的模板代码才被编译才被生成代码，因此是一个很好的设计时机..

对类型的设计才是设计。

为什么写代码需要设计呢，因为代码是人写给人看的，所以对代码逻辑的控制是需要的，而这就是设计，，设计更多指一种人类活动，比如艺术设计，所以它包括测试，重构等诸多过程组成的与编码相对的过程。设计首先是一种对问题的积极抽象过程，booch 甚至说抽象是解决设计问题最有效的方法之一，当然，维护，重构也是，所以说抽象问题只是设计的一部分然而是最重要部分。

在 C++中，设计首先是对类型进行设计进行抽象（泛型这个字眼本身就表明了对各种类型其功能通用，所以是一种设计抽象），有 OO。有 template。OO 是类型化即面向对象，template 是泛型化即主要用 C++的基于对象机制来工作。

泛型编程中对型进行的抽象，有 make types to be a list,有 mapping type to sth,有 get traits from types，尽量在编译期间将型别抽象到应用，形成设计，因为静态语言的编译期间正好提供强大的类型功能，，，而这里谈到的 typelist 就是一种。。

对类型作了这么多抽象之后，再提出 iterate，等设计手法用于 stl，提出 policy 用于 policy based design，，学习范型编程，始终要提醒自己把握这个精神（即一般泛型设计会分成三个层次，第一层是型别抽象，第二层在第一层的基础上提出邻域相关的设计手法，第三问题本身，STL 和 LOKI 中都是这样）。。

那么编译期运算(就是实例化，所以它会发生代码膨胀问题)是怎么回事呢？

8.11 模板的实例化

首先我们要弄懂实例化

一种实例化是利用了模板的产生式编程范式（比如泛型编程），一种实例化是利用了模板产生式，但面向编译器的元编程范式。产生跟元是不是一样的。

注意，你写模板源程序的时候你其实是针对编译写的，你还不能确定你的源程序哪部分会进入编译器(比面向运行写源程序是更高的层次，面向运行写的源程序肯定只会进入运行期)，什么是模板会产生代码呢，这个说法其实无比简单，只是当我们摆出诸如“为设计而设计代码”，“编译期而非运行期多型”” C++的元编程技术”这样的别称时，它就变得远离我们的理解变得面目生疏了，

我们知道高级语言的过程,词法分析和语法分析构成了编译器的前端（得出一个抽象语法树）,然后是语义分析,然后是中间码或目标码生成(如果有一个虚拟机,比如 JVM 和 JAVA 语言,那么这个目标码就是 JVM 中解释器要执行的目标,如果是裸机器本身,那么就是一种接近二进制但不是最终二进制的表示形式,当然一般在生成最终目标码之前要先生成中间码),然后是对生成的目标码进行优化,优化之后进行汇编形成真正的二进制。

我们知道，在编译型语言中，比如 c 和 C++中，写代码就是为运行做设计,程序设计期 design time 的就法就因此而来，写代码的本质就是在 C 的数据封装抽象基础上，C++的基于对象抽象上，C++的 OO 抽象上这些语言机制逻辑上+操作系统封装了的 CPU 逻辑上，写最终面向 CPU 的二进制代码，这就是我们说的，系统编程语言的本质在于基于系统逻辑（并且语言本身也是基于系统逻辑的,我们把计算机系统离散基础，语言实现离散基础， CPU 汇编语言所涉及到的那些机器离散基础都称为系统逻辑），写面向系统逻辑的代码。（与它对应的脚本语言更多地是为了直接为应用服务因为它极力让开发者绕过系统逻辑）。程序运行的本质是什么呢，，就是向操作系统和 CPU 请求资源， CPU 执行二进制，（任何我们写出的源程序，在经编译优化后形成目标平台二进制码，我们向 PC 编程以构造应用的过程只是系统逻辑到应用逻辑的的一个维度变换），这是指比如 C++这的以本地平台为运行环境的系统编程语言，而非指自带 VM 的，向 VM 编程的脚本语言而言的。

设计期就是写代码期，是用户动作，发生在编译期前，编译期不是用户动作，运行期又不是用户动作。。

模板实例化是生成采用特定模板参数组合的具体类或函数（实例）。例如，编译器生成一个采用 Array<int> 的类，另外生成一个采用 Array<double> 的类。通过用模板参数替换模板类定义中的模板参数，可以定义这些新的类。当然它跟模板特例化不一样,具体的模板技术细节请去参考其它书
先来说一下编译语言

再说一下 C++中的模板技术,一般用 typedef 来实现模板中的假变量定义-因为模板中不能直接定义全局变量(以下提供的例子是一个数据结构模板)

```
template <class T, class U>

struct Typelist
```

```
{
    typedef T Head;
    typedef U Tail;
};
```

显然， Typelist 没有任何状态，也未定义任何操作，其作用只在于携带类型信息，它并未打算被实例化，因此，对于 Typelist 的任何处理都必然发生于编译期而非运行期

然而如果我们对编译期编程(静态编译期),这将发展出一种元编程的技术.(利用模板实例化机制于编译期执行一些计算。这种通过模板实例化而执行的编译期计算技术即被称为模板元编程。)

模板是一种参数化的泛型，泛型这二个字决定了模板没有一个类型(在写模板函数，或模板类时的设计期，能难想象没有类型的空东西能够堂里皇堂的参加运算吧^-^，只有编译器技术到家就可以),,在写出这个函数模板或类模板后经编译时它并不产生代码（机器动作时），只有在接下来实例化时(注意，这个实例化是指在设计期用户动作期，用户为这个泛型占位符指定一个具体类型并写出关于它的代码，这个实例化跟我们在 OO 程序设计中，由一个 class 定义一个对象类似，接下来，在编译期机器动作时才会在编译期实例化产生实际代码到中间目标文件中比如 obj 中(非飞行模式),

什么是实例化呢？注意这个字眼

在 OO 程序设计中，由一个 class 定义一个对象并不实际动态分配内存，因为这还是指为运行期作规划的设计期并没有实际进入到运行期并分配了一个内存，如果用了 new 关键字，还是没有分配内存，编译期永远不分配内存，只是为程序将来在运行期如何获得内存作分配上的控制而已，,

只不过一个实例化用了不同的情景而已，实例化并不一定指为运行期实际实例化（也就是说实例化并不一定就是一种“为运行而作设计”的设计，，它还可以是一种“为设计期而作设计”的设计期动作，）它其实就是一种编译期的宏，只不过它有类型的判断，因此远远不是简单的文字替换性质的宏，，而是一种编译技术，模板由于有这个特性，第一，因为它是泛型，所以为 C++带来泛型编程机制，第二，模板给 C++带来元编译技术。

这样仅仅是一个软件内部的考虑，如果不在源程序的级别，你要提供你的软件为别人所用，或者是一个类或者是函数，你不可能向别人提供源程序，你就要在二进制级别上向

别人提供面向对象的特性(即可复用特性),,OO 的可复用能力正是体现在二进制级别。而 template 是源程序级别。因为它的定型在代码刚被编译时。

承接我上面一篇文章《为设计产生代码》

如果实例化不是为运行进行设计，，而是为了为设计期产生代码这样的目的而进行的设计期的设计（很显然，它本身也是一种设计，并不面向运行实例化，这是一种静态实例化，编译期的实例化,即它向前推进了一个意义层次），而且当模板机制的编译技术实现涉及到词法处理，语法分析这样的编译原理知识时，，，那么模板就是一种地道的编译器的编译器了，是一种元编程技术。

8.12 模板的继承

8.13 模板的偏特化

8.14 模板与元编程

编译期解释是被发现的，而非被发明的，在一次国际会议上，有人发现编译输出的错误中居然也有运行期相似的结果，，这是模板实例化的结果(我们知道一旦编译出错，源码不可能得以运行,我们知道一旦编译出错，源码不可能得以运行,,而如果编译时就得出运行期才能得出的结果,,说明它经过运算了也即"图灵完备"了),，于是 C++的这种语言机制就被“发现”了。并最终发展出 Boost Mpl 这样的元编程库。

我们知道 C++主要是在动态运行期发挥作用的，OO 如此，模板 STL 也如此(具备编译后端的语言皆如此，因为需要进入运行时，获得系统给的资源映射为平台逻辑，而元编程程序并不需要)，理解模板元编程与模板 STL 前者是静态而后者是动态的区别，就像理解模板与数据结构的区别一样，根本在于得首先理解模板，再理解模板运用于这二个地方到底产生了什么不同。而这一切的一切，其实都是因为模板仅仅是一种语法机制，而后两者是模板在不同情景下的应用形式而已。

所以，所谓模板元编程，就是把编译器当成了更高层次的解释器和运行时而已。
模板编程是产生式编程(比如泛型编程)，而模板元编程是另一种跟产生式不同的编程范式(因为它面向受限编译期)。

lisp语言常常被作为教学语言,因为它源于严格的图灵理论,lamaba高阶函数理论,它的变体shemhe语言采用list,即广义表作为数据结构,这种数据结构可演化成任何一种数据结构,采用函数作为语言内first class来作为主要的开发范式,,,因为这种开发方式是图灵完备的,实际上它的计算能力是等价于任何一门语言的。理论上可产生一切逻辑。
C++中的函数并非first class,但它的元编程实现了一种metafunction,,这使得function(实际上是变形了的类模板)可以使C++在编译期实现“函数式编程”,(在这种意义上C++的模板就成了图灵完备的子语言了,实际上C++所谓的元编程主要是在模拟函数式编程)当然这种动作是深受限制的故而需要作抽象迂回的,(比如它的metafunction 不像stl的仿函数是利用运行期支持的设施实现的,BOOST MPL库中的每个function 独成一个文件,都需要include才能用,这主要是因为模板实现的函数并非类型,编译期并不认识函数调用及函数原型不能以函数(参数1,参数2)的形式进行调用,事实上它只认识模板以及相关机制,实际上你还可以看到编译期C++的好多奇特的语法都是源于编译期不完善的设施在看相关书籍的时候一定要处处提醒自己把握这个要点(除非改良C++标准以使编译器厂商支持编译期待性比如最新的C++09的提出),
我们下面再举一些编译期的语言设施及导致的迂回方式(某某说过,抽象是解决问题的一切方法)。
1.编译期只识别递归(比如在递归中一个特化可以指定递归结束条件,要知道递归首先是一种思想,,它成立的本质有二,1,自身用自身来定义2,结束条件,,,如果说template<类型>是递归的第一层意义,那么template<0>这样的特化体就是递归的第二层意义使得递归得于成立),
2,enum(变量就不能用,只能用类型和常数这样的编译期就确定下来的值比如typedef),struct(用struct而不用class是因为struct默认它的成员是public的)这样的语言设施,而且只有int,bool作为模板参数可用。
2,模板不能以模板为参数,,除非“类模板模板参数”,这样就不能直接产生foo(class foo1,class foo2),这样的结构,,,要绕一些弯子,比如tag dispatch function.
3,循环展开通常采用了复杂的迂回技术,
4,让模板的成员带value,模拟函数式编程的返回值。
5,因为元编程实际上是对产生关于类型逻辑的一种编译期策略(要知道多态化是设计的一个重要方面,编译期模板使C++变成动态语言或无类型语言,而这使C++成为跟Ruby,Python一样的高抽象设计语言),它将要做的事情要完成的逻辑提早到编译期完成而不用到运行期的资源(因为C++模板赋于它这个能力可以这样做),所以通常可以用它来进行设计,模拟设计模式等等。因为模板是语法导向的,所以它的源程序是需要按源程序文件的发行的,进行语法级的复用(模板程序只需要编译前端就逻辑成立),而面向运行期的程序是二进制导向和复用的(事实上除了C++,没有那门语言有这样的能力吧),,这就是所谓的二进制复用,因为变量,class的内存映射,,全部是运行期的编译后端的事情。。
6,用list数据结构实现typelist.

写在后面的话:其实 C++的元编程属于十分高级的抽象能力,我们不应该把精力放在这样的语言机制上面,基础薄弱的读者和程序员只需要把精力放在学好 C++的运行期 OO和数据结构就行了。因为大部分公司开发中甚至都不会用到这些技术。

8.15 元编程的意义

面向对象在复用工作方面做得很好(比如它提供了继承，多态，还提供了二进制复用比如 COM，还提倡用类聚合代替继承而不是 tear off，还出现了诸如设计模式这样的复用经验),但是这是相对现实生活的那一端做的工作,,然而它对于编程工具端(编译器)本身来说是不友好的(程序源码必须要进入 runtime 才能让我们看到这所有的 OO 里面发生的事,在编译阶段 (一般也称为 design - time) 我们不能控制这些 OO 对于问题域的实现)，我们应该在没有让程序进入某种具体编译器之前，，就让它得以被控制,而不仅仅是预测这些编译的文件进入 runtime 以后会形成怎么样的逻辑
也即，类的职责单位是类文件，这种机制有一些的缺陷性，问题域是巨大的，如果我们动手一项工程，我们不希望被无穷的细节所困扰(现实问题总要分解为一些类，最终要形成类文件,一般每个职责形成一个类)，我们希望有一种介于编译器和现实问题之间的更大的整合层来考虑事物(而不是一个一个的类文件),,也即，我们不需要考虑现实问题到类的实现路径，我们希望在设计期就考虑现实问题到一个“比类还大的”，“更接近现实问题”的逻辑层上去，再由这个逻辑层到最终的类实现路径(比如单例模式，就是指代设计中只能出现一个实例的逻辑实体，这已经十分接近设计了)
如果这个层面被提出来，它甚至不占用到运行的时间，，即增加这项抽象，并不耗费运期间任何成本（因为它只发生在编译期）
因此它是语法导向的，而不是虚拟函数集导向的
这个整合层就是策略，，，模板技术允许我们在编译期就用“策略组合”加“模板技术”来生成源程序，这实际上也是编写库为用户所用时所要考虑到的问题
用户希望能从库中提取功能的子集，这势必产生这里提到一个 trait 的概念，简单描述一下先

相比 C#和 JAVA,RUBY 这样的语言来说，实际上即使是 C++也没有直接在语言级上提供太多的抽象机制，而其它的语言，比如 JAVA 在语言级就有代理，RUBY 在语言级就有混入。相比之下 C++中只有 OO 和模板,（故称 C++是第三代语言，而前者是第四代语言），而且 C++的 OO 是二进制导向的，一直为后人稍稍诟病，但 C++的模板技术却大受吹捧，它的抽象能力不小，首先它正用能得到 generic programming，偏用能得到 meta programming,而且丝毫不比第四代语言的那些抽象弱。这迎来了 C++的再一次发展高峰。

模板中不需要变量这些运行期的设施，只需要 type 和常量,, 它又是语法导向的（所以其复用的手段往往是提供源程序，而且其缺点是不能提供强大的运行期能力，比如 Croba），而不是二进制导向的,, 因为运行期才有变量的概念，我们知道 C++是个强类型的语言，因此无法做到运行期的 duck typing,但模板使它具有编译期的 duck typing 功能，OO 的那种产生类型的机制是不纯粹的，可侵入的(我们称这种语言为面向对象)，而 duck typing 才是好的产生类型的机制(我们称 Ruby，Python 动态语言实现的 OO 为基于对象)，而 C++对于类型控制的能力在于它的编译期（因为它是强类型的），，而不是运行期的 RTTI，实际上 C++的 RTTI 是很弱的，它用来支持 OO 是需要花费运行期资源的，而作为一门静

态类型语言的 C++在编译期就有了关于类型的一切资讯（比如它实现多态就把这称为静态绑定），而如果将这种决策推迟到运行期(来实现运行期的 OO 多态)，那么就必须借助于语言的 RTTI 机制。

meta programming 是对模板的一种偏用，即 trick 而不是 C++的一种技术，它是被发现而不是被发明的只有范型，人们发现编译期也能实现一些运算，比如模板中也可使用 if else 和递归这样的语言机制，但实际上模板不是语言，无法直接识别 if else 和递归，实际上它只是傻傻地实例化(即使是这样也能图灵完备)，但是从另一种眼光来看，因为模板机制能间接支持 if else 和递归，，所以它终究是一种图灵完备的系统，即模板实际上可以看成是 C++的一种子语言，另一种意义上它也是 C++的 embeded language)。(比如它用这些 tricks 艰难地实现了一套编译期的 stl，详见 c++ meta programming 那本书，而我们在用的源于 HP 的 stl 是运行期的)

最新的 C++0x 标准或许可以改变这种局面。C++0x 标准为 C++提出了一系列改革，的目的就是为能使元编程成为模板的正用。。

8.16 真正的策略

为什么需要 policy 呢，，因为我们知道在应用开发中设计往往是做多选题,对应于应用域在解域和语言域中有大量可供选择的方案，所以可复用组件最好是给用户小的设计组件，用户才能借以组合它们形成更强大的设计方案（具体到每个领域，它的设计都应该如此,比如 loki,的智能指针，仿函数等具体领域都是策略 based 的）。因为设计元素只能分解而后才能复合，而不应该是一开始就复合了。。如果一开始就提出一个 do it all 全功能型的接口，那么往往复用性从一开始就固化了（OO 就是如此，单根继承往往涉及大量不必需的东西进入设计，组合才是科学的机制）。这往往很不好。因为它只能向前发展不能向后发展。那么组合大量小 policy 形成的对某个瓴域的某套组合 polices，给了我们后退的空间，我们可以组合需要的去除不需要的，这才是我们需要的，即设计中可以在此做选择题的能力和场所。。

策略为什么跟设计模式有关？

我们知道设计是高于编码的，这就是说在我们进行设计时，进行的是一种纯思想的活动(非常重要的是这里的设计二字远非设计模式中的设计那么狭隘)，并不预先想到要面向编码而设计，然而我们得出的某种设计方案(比如一种架构逻辑，，设计模式)并不是不可以在技术上用一种方式来实现，因为狭隘的设计模式之说的设计不过是根据设计方案产生一堆类文件而已，设计模式高于成码跟设计模式可以用一种技术来实现并不矛盾，关键是找不找得到一种技术实现手段的问题，你可以联系 ISO 网络参考模型来考虑，一种思想或协议的确可以用来实现。。就凭这点，我就十分佩服策略的提出者

策略是属于设计的而不是编码的（虽然策略也是写代码）,,，这是本质，明白这点非常重要

策略就是这样一种技术，模板技术天然就用于参数化类（这是模板的本质抽象）并产生类文件，这在设计期是一个天然的技术实现，比如一个类文件可以产生一个职责或对应设计中的一个对象，而模板可以跟据对象本身的特征(比如对象的类型)去决定产生的表示这个对象的类逻辑，，而这正是策略要解决的问题

策略是多范型设计的较高境界,虽然 OO 也可很好地表达设计和思想，其它范型也可以，然而策略更接近设计,比如 DP 的 Singtion,Boost 的 Enable If,,从这些字眼上看就直接跟思想和设计已经十分接近了。（不要忘记 DP 中一种是模板方法设计模式，，这更一步反映了模板与设计的深层渊源）

策略与 OO 是什么关系呢，（OO 和 OOP 出现在前，，然后是 DP,GP,Policy 这些范型），因为模板就是一系列可运行的参数化类的逻辑，那么对这个过程也可以用 OO(OO 与策略并不是包含或被包含之类的关系)来实现，，因为模板也支持 OO，比如 STL 的源码中，对容器，通用算法这些的策略大都没用到 OO 而是一堆面向过程的过程模板 (函数模板就是子过程模板，类模板就是用到了一点 OO 的模板),，(如果策略本身也是用 OO 范型(通常说的 OO 就是类文件级的 OO)来实现的，那么这可以称之为二阶 OO，第一阶是 1,模板产生类的过程中用到了 OO，2 类本身也是一种 OO,)

因为策略就是模板技术描述的设计思想，设计思想可以是 Gof,Boost,Loki,Core J2EE Spced DPS 这样的大思路，也可以是一切反映设计的小思想，，因此策略不仅仅是我们现在已经见过的上述一些实现，,，它可以是千千万万的不同思想的模板实现形式（因为思想有千千万万嘛，抽象是无底的）

什么是编程，什么是抽象能力，编程能力，什么是学编程(即使这样看似最最简单的问题也其实不会简单，网上的确有太多这样的争论，那么这里将会给你一个有形式描述的概念，关于我在一个维度的认识，因为我们在上面提出了诸如设计，抽象，原语之类的用词，因此接下来的描性会变得容易界定，这就跟设计模式内部的那些功能分工的说法一样，，只有给出了能形式化的中间逻辑用词，那么才能不会在争辩时连最细小的中间共识都不能达成，那么也就不可能得出一个最后的结论)，比如我在第五部分会写到一个世界逻辑库，因为要在这个逻辑世界里产生很多 Actor,,而 Actor 本身又有 Type,,Ablility 之分，那么我可以用模板表示，,（ectc），如果你能像我这样想，那么恭喜，你已经在非常有成效地学编程了，,，而你在看我写的代码，，那么再一次恭喜你，你已经在实践了

所以什么是编程，，编程就是用各种范型来实现设计，,，

学编程就是在大范型之下学大量细节的小范型（所以说 JAVA 并不是一种很好的拿来作为教学语言的语言。因为它在 OO 方面和 OO 后面的事做的很好，然而在 OO 前面的那

些范型全部被忽略了）

抽象能力就是对外来架构的理解能力，和掌握大幅中间逻辑，，以构造新逻辑的能力（计算机与人脑的区别就在这里，人脑可以主观能动地发明抽象，，而计算机只是我们用来反映抽象的工具，它本身并没有主动意识和主动构造抽象的能力）。

8.17 C++的基于对象设计：模板与设计

不幸的是，理解 STL 深层的原理是需要懂与模板相关的设计的，比如仿函数的本质，迭代器，配接器的本质，模板导致的泛型开发与它提出的这些设计相关的东西可以另外写成一本书。学习 STL 首先是学习这些设计手法，再学习其数据结构和算法的实现。

泛型有二层意思，第一，基础泛化，它把泛型参数化，用于动态产生关于不同型别组合的相同逻辑（可以联系函数声明和函数定义来理解），这也就是一般泛化了，第二，它把一切设计中可能出现的因素都类型化(template class 化)，即在 template class 这个字眼中不主要强调 template 泛化而是 class 类型化，（只不过它也会用到泛型的第一层基础泛化作用而已）比如，迭代器，仿函数实际上都是(模板)类,这其实更像是 C++的概念而不是泛型的概念。(因为 class 是 c++的而 stl 及它导致的 template 手法是另外一个人发明的)

为什么需要把指针，函数封装为 class 呢，这是因为在 C++中，class 几乎就是一种逻辑粘剂(即将数据成员和函数成员，，当然在模板中也可以是模板成员数据和函数，封装为 ADT)，在这里并不强调这些 Class 运行于 runtime 的那些特征，比如多态，等，而是强调 class 封装逻辑成 adt 并提供 private,public,protect 修饰机制的能力（相比之下 C++的 struct 太简陋因为它只能提供全 public,而且不能成为 adt,因此没有 adt 的诸多好处，比如 C++的只对 class 有效的运算符重载，，而 class+oper overloading+template class 你呆会会看到，，这在泛型设计中是多么有用处的东西。)，，所以在 C++中，相比面向对象来说，这些基于对象的开发范式也需要被重视。。

一句话，class 化可以获得语义级的 value, 只要给该 class 一个 copy ctor 就可以复制并传统它，给它一个重载的括号就可以成为跟函数动作一样的东西出现在 C++语法相容的东西（虽然语义实际跟标准的对应物不一样），，，

template class 是泛型设计中的重头武器，因为：
函数+oper () overloading+template class 化 = function objects，这就是 stl 中的仿函数。
pointer+oper * overloading(和->)+template class 化 = smartpointer，这就相当 stl 中的

interate.
Multiple inhert + template class 作为另一个 template 的参数=policy，即 loki 中的策略。
Struct+template=nested type,, 内欠型别。。
元编程里的 metafunction

模板特化与实例化是不一样的，，，其实任何一个模板，都存在二套参数，一套是泛用的，在tempalte关键字后面，另一套是特化或偏特化用的，在具体的模板后。特化与实例化的区别在于实例化不需要人去干预。。

8.18 lesson000x - 演示了多继承与虚拟继承

```
//这个源程序演示了多继承与虚拟继承
#include <iostream>

using namespace std;

class geo
{
    public:
    //这些 public 接口都不应该在其参数表中出现私有成员，也不该直接出现数据成员。
    //所以 posx,posy 只是临时的形式参数。
    void virtual init(int posx,int posy);
    //x 和 y 方向上的偏移量
    void virtual move(int corx,int cory);

    private:
    //私有成员只有自己和友元用。
    int x,y;
}; //注意，作为一个声明，无论是声明 class 还是 struct,都需要在这个位置加入帽号。

void geo::init(int posx,int posy)
{
    //只有在定现中，才能出现私有成员，在定义 geo 的 class 接口时不应出现。
    x=posx;
    y=posy;
} //定义就不需要加入帽号了。
```

```
void geo::move(int corx,int cory)
{
    x+=corx;
    y+=cory;

}

//再来定义一个从 geo 公共方式派生的 line class

class line : public geo
{
    //继承并重载(重载而非重写)
    public:
    //因为处在不同的 class 声明中，所以 posx,posy 这样的临时参量可以混用。
    void init(int posx,int posy);
    void move(int corx,int cory);

};

//下面来实现它

void line::init(int posx,int posy)
{
    cout<<"call from line::init()"<<endl;
    cout<<"now my (i am line) posx,posy are"<<posx<<posy<<endl;
    //当然，你还可以加入其它的逻辑

}

void line::move(int corx,int cory)
{
    //跟 init 一样，如果 line 有它自己不同于基类 geo 的 move 行为，你可以在这里加入
你自己的逻辑。
}

//再定义一个公共派生自 geo 的子类，以讨论多继承的特点

class circle : public geo
{
    public:
```

```
    void init(int posx,int posy);
    void move(int corx,int cory);
};

void circle::init(int posx,int posy)
{
    cout<<"call from circle::init()"<<endl;
    cout<<"now my (i am circle) posx,posy are"<<posx<<posy<<endl;
}

void circle::move(int corx,int cory)
{

}


//定义一个通用函数，它以 geo 为操作对象，以演示其子类会有什么样的行为
//看到了吗，initx,inity 跟 geo,line,circle 的 init()参数几乎一致，但其实含义不同
//后三者是声明的 class 内的 public 接口的参数，而这里的全域函数 mycommoninit 的参数。
//一般来说，class 内的接口要做到设置参数时的最小职责原则。
void mycommoninit(geo& _geo,int initx,int inity)
{
    _geo.init(initx,inity);
}

//void mycommonmove

int main()
{
    line myline;
    circle mycircle[5];

    mycommoninit(myline,10,10);

    for (int i=0;i<5;i++)
    {
        mycommoninit(mycircle[i],20,20);
    }
```

```
    return 0;
}
```

8.19 仿函数

仿函数是 C++基于对象编程的典型，，它把对象 class 化{也可能是 template class 化}，使之具有 copy ctor 可被复制，再给它提供一个重载的小括号这样在语法上就可以跟普通函数一样写了，整个过程并不需要面向运行期（面向对象），，，所谓模板加基于对象的基于对象之说只适合发生于编译期。

STL 中为什么需要仿函数呢？它为了成为算法的某种策略，，loki 中的仿函数用来实现一种 command 的设计模式，因为仿函数可以用来封装一系列请求；

8.20 traits

Traits 是剥离器，是一种设计抽象（往往人们也把它称为 concept,,满足 concept 的实现就是它的一个 model，即 concept 是编译期关于类型的 interface），广泛应用于 stl,loki,等设计理念中（剥离器一般只用于泛型设计，因为需要从泛化了的型剥离并获得它的 traits,实际上这个词更多地强调的是结果），是成就 stl，loki 等的支撑逻辑。因为类型有“一般 type”,作为 template class 的迭代器，等等，所以也有相应的 type traits,iterate traits.即模板的参数可以是什么，那么泛型也可泛化什么，形成相应的泛化意义（比如 traits），因此泛型可以将型别泛化，可以将操作泛化，甚至可以将一大类的操作和型别泛化。更甚至，可以将“template template 参数”泛化，注意我并没有多打字，现列举可能作为 template 所有参数的情况，并一一加以解释：

8.21 iterater

Stl 中，与 template 相关的一个设计手法就是迭代器，迭代器成为表达各种数据结构以及跟它们有关的各种算法操作的支持概念。当然也是一种泛型手法。Idioms 其实也是一种设计模式，即迭代器模式（traits,adapter 泛型设计手法可用于广泛目的，相比之下 iterate 好像只用于数据结构的设计手法设计模式），

既然是泛型编程，迭代器是在什么样的泛化需求下以什么样的泛化方式被提出来的呢？

我们知道数据结构都是某种区间，把数据结构视为区间这本身就体现了某种泛化（能泛即能提供通用性，可复用性，所以是一种对代码趋近于人的设计抽象），某种抽象，实际上无论是以何种结构形成的关联式(key+value=pair 对)还是非关联式数据结构（），，迭代

器都将提供一种游动于元素（一般来说元素只是 value 的说法）或节点（一般来说 node=key,value）之间并能对算法提供迭代支持。只不过迭代器作为泛型的型它也可以有多种 iterate associated traits 而已，有的是 input,有的是单向，有的是双向，有的是 const,有的是 mutable 而已。

8.22 adapter

配接器的说法很形象， 你可以联系现实生活中把 ps/2 鼠标加一个 ps/2 2 usb 接口，，把它转为 USB 接口的这样一种动作，，这种动作就是改变接口的机制使之由旧接口变成新接口，一种设计策略（改变原有代码，使之适应某种复用考虑，，所以是人控制代码的过程，，是设计抽象，，这个动作也称为重构，即不改变原有系统设计的情况下，利用设计手段修补式地改造原系统，因此跟 DP 相关），，往往把它归为专门的一门设计模式。

因为客户(你的电脑 ps/2 口坏了不能插 ps/2 鼠标了)只能使用某种接口的东西，所以需要对原有接口（原有代码）进行接口重新封装，使之向现呈现客户能用的接口。这是典型的设计模式应用于给代码打补丁的情形即复用的情形。（当然设计模式也可一开始用于纯设计的地方）

那么 stl 中的这些配接器都是些什么呢，又怎么样在 stl 的整个设计中发挥了作用呢？？

8.23 lesson000x - 演示了 vector 及一般的 stl 容器操作

```
//演示了 vector 及一般的 stl 容器操作

#include <iostream>

#include <vector>

using namespace std;

int main()
{

    //初始化 myvector 内容为 10 个 2
    vector<int> myvector(10,2);

    //连续从后端推进 3,4,5,以利于讨论
```

```
    myvector.push_back(3);
    myvector.push_back(4);
    myvector.push_back(5);

    //打引出 myvector 内容,这是用 size()版本作为步进量的循环

    for(int i=0;i<myvector.size();i++)
    {
        cout<<myvector[i]<<endl;
    }

    cout<<"=========================="<<endl;

    //打引出 myvector 内容，这是用迭代器版本的

    vector<int>::iterator myvectoriter;

    for(myvectoriter=myvector.begin();myvectoriter!=myvector.end();myvectoriter++)
    {
        cout<<*myvectoriter<<endl;
    }


    cout<<"现在演示利用迭代器在 vector 里寻找某个值的数，并利用迭代器打引出这个刚刚找到的值"<<endl;


    //迭代器本质上就是一个指针变量，所以可以不断在 vector 里游走

    //find 是 stl 的名字空间函数
    //查找值为 3 的数，find 将返回指向 3 的这个迭代器本身。
    myvectoriter=find(myvector.begin(),myvector.end(),3);

    //因为迭代器就是指针本质，所以可以提领其值。
    //为了安全，先验证一下是不是已经成功找到了 3
    //if(myvectoriter!=0)
    {
        cout<<"刚刚迭代器游到了指向"<<*myvectoriter<<"所在的 vector 位置"<<endl;
    }
```

```
    cout<<"现在演示删除这个刚找到的 3"<<endl;

    myvector.erase(myvectoriter);

    //为了验证是否成功删除 3，打引出整个 vector 内内容，此时应该时 10 个 2,4,5 这十二个无素


    for(myvectoriter=myvector.begin();myvectoriter!=myvector.end();myvectoriter++)
    {
        cout<<*myvectoriter<<endl;
    }



    return 0;
}
```

8.24 泛语法编程而不仅仅是泛型编程

我们知道，其实普通的类也可作为一个函数的参数，形成以“类型”为参数，但是这跟真正的泛型是有很大差别的，普通的类以类为参数，那个类是一个具体类。并且只能是函数有这样的功能

而模板类，即泛型类，那个参数可以不是一个具体类，并且，可以有类模板，函数模板这样的概念。

现如今，对于类型的抽象已经越来越完善了，经历了一个从基本类型到结构类型到类类型再到泛类型的过程，这当中，只有泛型的型才是最广泛的。那么泛字究竟指什么呢？其实是指“类型所能表达的意义”的泛

面向对象的类类型更多地表实体概念，当然也可表抽象概念，然而它要命的地方是:无论是表实体还是抽象，它都把所有的东西都从代码上形式化为一个 class，把代码看成数据加操作，而这两种行为都是具有极大局限的。

而从泛型开发中我们越来越多地看到，代码可以是函数，可以是类，可以是结构(实际上，模版是一个泛化了的代码表现模式，并不强求把所有的代码逻辑都写成类，但可以提供对类的操作)，这是对第一种行为的突破。

换言之，面向对象一律把代码看成数据与操作，这很傻

对于第二种行为，泛型所表达的型不但可以是实体，而且可以是关于另一个类的类型，所以泛型的型中，其内部有型，而外部的型可以是任何概念，不成强求为数据与操作。

这样，在语言用它的类型表达 dsl 逻辑这个层面上，只有泛型的型才是这种逻辑的最好表达工具，因为对某一种逻辑来说，任何东西都不必体现为数据和操作，而可以是其他接口。

熟悉了模版你才会发现它的科学性。

8.25 Python 的数据类型

很值得一提的是，python 很高层，直接就用 dict 这样的东西穿插起了代码，而我们知道 dict 是数据结构抽象，它用数据结构来表现类，比如一个类实际上是一种 dict,它的方法和类是一对 pair,它的函数和函数参数也是某种 dict 的 pair.
所以说，python 是很高层的，它有别于 c++这样的用库来一点点表现逻辑的方法（C++在语法内并不直接支持 dict 吧？），而是在语言级就直接内置了高级的逻辑。使得你可以用高级的东西（数据抽象的，数据结构的，代码结构的）来构建逻辑。这也使得语言要素很少，学习曲线很平滑。不必从一开始就学习基本的东西。(比如用 C 来表现 notepad，要实现基础的打开，保存逻辑，SDK 方式开发的话，你需要写很多代码，而 python 可能只需要几 K 代码)

在 python 中，有二种类型的数据，value 的和 referrence 的，list 又有 frozenset,tuple 的和可变动的类型。

8.26 Python 的动态类型和动态代码

在静态编译语言中，关于类型的逻辑一般都是全程固化的，首先，在编译期一定是被固化的，所以只可在运行期作一些变动以符合编程工作，然而，其即便怎么变动（比如 C++的 RTTI，java 的反射),，也还是有局限的（一是难于实现，二是会造成效益下降）。

而动态语言一切都发生在运行期，它在运行期变动语言要素（数据类型和代码结构）的能力是天生的。

首先，类型无须声明再使用，不必再出现 int myintvar;这样的东东了，因为变量可以直接脱离声明(编译器才需要声明获得类型信息)而被定义出来，而运行期的变量一定是某个类型的某个值对象，所以可以直接用。

其次就代码结构来说，类的方法成员等可以在运行期被替换。。

而 lamada 这样的东西，，饰符，yield 这样的东西，，就进一步体现了 python 在运行期具有强大可变动的能力了。

而其实，编译期是机器能发挥它的能力的地方（是最适合设计的，比如模板策略），只有运行期才是最靠人的应用的（比如实时显示结果，但调试困难）这就是一对矛盾。

8.27 Python 的内省

python 模拟了函数式编程，和实现了一个 OO，由于 python 强大的内省能力，它可以动态改变程序元素(type,udt class,module)的结构，，比如替换类方法，当然，这全部都是在运行期完成的。

内省就相当于 C++的 rtti,它提供动态能力，即运行时的那些可变动性支持以什么样的方式和逻辑编程的能力。

当然内省是一套机制，而非 python 的一种机制。

8.28 Python 的迭代器与生成器

迭代器是具有迭代能力的语言要素，比如具有.next 方法的类，或者其它非类的语言要素(比如 module 类型的 list，当然从类的眼光来看，list 也是一个类型，不过是 builtin 的,python 有二种类，一种是 built in，一种是 classic class)，python 把这样的东西（设计模式的东西）直接做到语言级而不是库级，，这使得 python 这门语言本身很 high level,可以与设计和应用框架这样的东西直接挂钩。

在 python 中还有饰符这样的语言级直接支持的设计模式逻辑。这一切都说明派森是作为一种高层语言存在的。

8.29 Python 的闭包

函数编程范式绝非操作单纯的函数，，而是一种泛函数，是一种抽象函数(过程编程的代码结构抽象)，比如闭包，它就是函数编程语言内的泛化体，它可以整合数据，如果说 00 编程范式中的对象是“整合了函数的泛数据”，那么闭包就是“整合了数据的泛函数”闭包充分体现了 python 的函数式编程范式。

8.30 Python 的饰符

饰符主要是为了实现 00 中的多态，当然，据说它不光对普通的 python“对象”起作用（比如一个普通函数），还对类方法起作用。这也其实是设计模式在 python 语言级的体现，

其实也是一种语法糖,,就像 python 里的.next,,作为一种语法模糊器,隐藏底层的工作机制,在写法而非语法层面上给人一种自然的感觉,,比如.next 的迭代器,.read 的 file-like 对象,这符合 python 要产生一种自然语言编程的效果的要求.

第 9 章 综合抽象之设计模式

9.1 真正的设计模式

我讨厌在讲授一些思想的时候提供大量的代码，因为我觉得接受思想的过程应该是一种快乐的如同阅读一本小说的过程，而不是花大量脑力研究某个细节的过程

而且这个世界，往往知识都只是相互转载，国内没有多少人会像那些欧美大师特立独立发明一些新的思想和论述，而我愿意写出我的一些思想与你们共享

每一个设计模式中都出现了一些角色，然而使用某个设计模式的个体(Client)不属于设计模式的某个角色,而只是使用这个设计模式的客户，，设计模式的目的就是为客户提供一

个好的对现有对象的访问方法,设计模式是一种高于任何成码的思想和经验模式，因此不能直接用某个工具建模下来，在使用设计模式的过程中，,，总会产生一些新的抽象(而且有时不只一层抽象),,这些抽象隔离和解偶了客户(Client)与现有代码之间的关系，,，在它们中间作为中间抽象出现，而所谓抽象，，往往都以一个类的方式存在，(因为 JAVA 中一个类默认只承担一项责任或实现一个对象数据描述，因此一个抽象往往就是一个类，当然，抽象有时以方法的形式存在，某个设计模式也会以方法的形式存在，比如工厂“方法”模式，一般来说，设计模式都会形成某几个抽象类，对应该设计模式中的几个角色 Actor)，,

设计模式终归是一种迂回的方法(因为增加了抽象所以代码变得有点难于理解而且类层次增加这变得运行时变慢了一点)，，然而这种方法成全了一种好处，，那就是：它部分或完全都解偶了使用者与现有代码之间(实际上设计模式可用在开发的各个阶段)的关系，,，这使得以后的对软件的维护工作和修改需求变得易管理和易实现，使软件不致于由于当初设计上的欠缺而变得难于修改而濒于死去.

9.2 设计模式与数据结构

对于设计模式，一般有下面几种理解： 重构时的设计模式。修补式的设计模式(client 角色浓重) 大设计时的设计模式。全新式的设计模式（可以没有 client 角色在某一套模式中）

其实。不妨把设计模式称为抽象模式更好(我们知道抽象问题领域是设计中的一个重要步骤)，，因为它更多地跟着眼于解决具体事物有关（就跟数据结构一样，不是跟具体语言有关，不是属于某种代码结构。）正如数据结构是择“数据”这个维度来抽象对现实事物映射到计算机解法的做法一样，设计模式是择“模式中的各个角色和关系”来映射对现实事物的模型，从而求得一个现实问题到计算机的解法一样（就目前所提出的一些设计模式来看，他们都是抽象现实事物模型的初步组件，一般有倾向于用面向对象语言来实现的趋势比如四人帮那书）。数据结构和设计模式都不会跟某种语言和语言机制有关，跟“面向对象”这样的代码抽象有本质上的差别，是实现模式，实现结构，而不是代码结构。着眼于如何解决和抽象问题，而不是如何抽象代码以进行更好能被复用这样的软工目的（当然，这二者是不分家的）。

我在《OO 为什么不是银弹-过度抽象的利与弊》中谈到，OO 并不是银弹，银弹是那些能统一人类思想，形成契约文化，经验的东西（比如我们写小说的那些套路），而不是简单的 class 这种面向复用的小技俩。 设计模式正是上述所谓“契约文化，经验”之类的初步体现（不可否认，我们所看到的设计模式跟具体现实事物还是有很大距离的），等到有一天，所有的问题都用设计模式来抽象的时候，成千上万的设计模式会被提出来。人们会倾向于用大大小小的设计模式来解决问题。那么设计模式就会到达它的颠峰。

然而对于程序员来说不利的是，数据结构已经被很好地映射到 C 语言中，而设计模式几乎在 C 语言中找不到它的影子。这正是它不成熟的地方。也许有一天会有一套“设计模式”专用语言出现。

9.3 设计模式之基础

四人帮的那本书是基于面向对象来谈设计模式的，因此它先提出一些面向对象的知识，比如一个类的 class 和 type(interface)的区别，提倡对接口编程而不是对 class 定义即实现编程，提倡对象的组合而不是继承。而且它稍后提到的诸多设计模式中，都有对象，职责，请求，之说，这些都是 OO 里面的知识。。

不要小看了这里的对象组合它实际上是对接口编程的小化说法

9.4 真正的开闭原则

我们应该对扩展开放，的同时(注意这三个字),,,保证对修改的关闭(一个工程，应该在设计时就要考虑到将来修改的需要，而且要保证未来修改时能尽量降低工作量，对于一个真正的工程级的规模，人力管理工程应该尽量简化),

几乎没有接确过设计模式的人(除了完全外行和真正的编程高手外)看到这句话都会感到疑惑,而且会产生一个很普遍的疑问：不修改何来扩展？

高手与低手（我没说新手因为我觉得低手这词好用）的差别就在于这里，高手往往看重的思想(即设计能力，,，识别架构和建立构架的能力，但是因为计算机能理解的设计只能是多范型设计，因此高手着重的这种思想往往也是受计算机实现的限制的设计)，而低手考虑问题的第一切入点就是源码本身，因此产生“不修改源码何来功能扩展”的疑问也就很自然了

而其实，在高手的眼里，只要定义一些抽象，产生一些迂回就可以解决问题了,这些迂回(实际上就是产生一些高层逻辑，这些逻辑就是具体某个设计模式中的某些角色 Acotr)可以让我们 clients 通过这些高层迂回避免直接接确到低层的实现（虽然我们 client 无论如何最终是要进入到具体实现的，但我们可以不直接而间接迂回地进入啊！！这些低层的实现就是现有代码了，是实现部分(可能是某个你要使用到的第三方库代码)，我们经常要对实现部分修改，或者说对现有代码的修改,而要求要有最少的工作量，而一个没有定义抽象或者没有定义好合理抽象的工程要涉及到很多修改工作），而这，，真真实实就是解偶的意义所在。。

我们再来说这些逻辑，其实这些逻辑都可以称为中间逻辑，，然而这些逻辑的地位又是不

同的，，有与具体实现接近的那一端的逻辑，，这些逻辑也是高层逻辑，，但是把与接近 client 使用者的逻辑看作为相对更高层的逻辑。。

说个故事吧！
《西游记》大闹天空时，要求当天庭大官，太白金星向玉皇献记说让孙悟空当弼马温，太白金星的智慧就体现了开闭原则，一方面，在孙悟空方面，太白向孙悟空说明玉皇已同意他上天(对扩展开放)，，另一方面实际上只是给了他一个放马的差，实际上按天庭规则(系统原有结构)孙悟空是不能上天的(),然而迫于孙悟空的力量(修改的需要),太白只是稍微迂回了弯子(增加了一层抽象)，就暂时平息了玉皇(玉皇本人不知道如何扩展这个需求，因为这是与天规相背的)和孙悟空二边.

9.5 真正的通米特原则

之所以不称通米特原则为通米特法则是因为在设计模式领域内实在不存在一个法则之说，通米特法则也称为最小知识原则，一个事物对另外一个事物知道得越少，那么它本身就越安全(这可以联系武侠小说里小人物碰巧目睹了对杀手杀人的整个过程，那么这个小人物就会有杀身之祸，)，

这里的安全是指对修改关闭

实际上无论对象组合还是继承都会造成类与类之间的引用，都会造成不可复用的问题，然而，相比继承来说，组合可以极大地减少这种复用的偶合程序，而继承压根就是不可分离的，因为本质上组合是一种 Has-A 的关系(组合对象与被组合对象)，而继承关系是一种 Is-A 的关系(基类与继承类，或称父类与子类，注意这二个概念还是有点区别的，一般说到父与子关系时就是指父对象与子对象，而说到基类与继承类时往往描述类与类之间关系的用词~~)

还有一种关系是 Link-A 的关系，这种情况下的不可复用性按情况下来定，，Is-A 的准确意思是什么呢（这里的意思指语义）？如 B is a A,,那么“B 是一个 A”，，可能是一个 A,但是不一定必定是一个 A.而且如果 B 是一个 A,，那么反过来就不能成立（子类化，虽然站在类型转换的场合下可以但是现实生活中这样理解不通）

一个代码的修改量应只取决于它最低层的实现，如果某个低层引用了过多高层逻辑接口的实现，那么这只能说明，对这个实现的解偶还没有规划到家，理想的情况是，应该只让这个实现的修改不触动到任何间接使用它的高层逻辑！！(因为自顶向下的引用对于顶来说，如果底部被修改顶部是不用作任何改变的，而如果是自底向顶引用，那么当底发生改变时，一定要涉及到顶部也要改变，，而这就是不恰当的高层抽象，违背了好莱坞原则和通米特原则)

.

9.6 真正的好莱鸽原则

好莱坞原则（不要给我打电话，我会打电话给你们）强调高层对低层的主动作用，即低层应该只管好自己的工作（具体实现），而高层自有它自己的工作（这就是管理低层的逻辑们，或者说从 client 到具体实现的一系列中间逻辑），在不需要到某个低层的时候，高层并不会调用到这个具体低层，低层永远不需要向高层作出表示，，说它需要被调用，，(即在所有的处于使用者与现有代码的中间的,用于隔离和解偶二者的,那些中间逻辑中，低层逻辑永远不要涉入高层的实现，而只要高层通过某个逻辑去涉入低层的实现，也即低层应不要调用高层，只有高层才会去调用低层，这才是合理的，我们应尽量避免向上调用和相互调用).

9.7 真正的策略模式

Open 和 close 一点也不矛盾，当它用在同一个架构上，open 指出这个架构的可扩展性，而 close 指出这个构架的内敛性，open 是相对高层来说的，，而 colse 是相对内部实现来说的，，，一个构架应对高层 open,而对内部实现 close,,,

策略模式将可变的行为集封装起来，这符合 OO 封装“可变部分”的原则，可变部分就是实现，我们修改一个软件直接修改的就是实现，而非抽象(实际上也不应该也没有必要对抽象进行修改，如果你的工程存在对抽象的修改，那就只能说明，当初在定义抽象的时候压根定义的抽象就是不合理的抽象,真正合理的抽象将使用者客户和现有代码极大地解偶，这使得以后的修改工作只需在低端实现进行而无须触动高端).

9.8 真正的观察者模式

好莱坞原则指出，类之间应尽量避免低层(实现)向高层(抽象,逻辑)的引用，
观察者模式中，观察者，被观察者，一个被观察者管理诸多对象(观察者)，这些观察者通过

9.9 真正的装饰模式

装饰模式就像是一个用类来修饰类的机制(这就添加了新的职责到被修饰的类，，这里说的修饰本质是什么呢？就是类的组合，让一个类被修饰者成为修饰者的一个实例变量)，，，这要求修饰类(可能是多个)和被修饰的类有一致的接口(也即它们同共都曾实现 implent 了某个接口，或者继承了某个有接口作用的抽象类 extend,,这样一来，就可以在动态运行时用一方代替另一方，然而客户并不会知晓其发生过内部的替换)

装饰模式可以让很多具有对等地位而且拥有共同接口的类进行有穷互饰，这样可叠合多个类进行某个共同的接口作用,并获得最终的修饰过的这个成员作用

.

9.10 真正的单例模式

某些只能够拥有一个实例的类对象必须通过某些方法来保证它在程序运行期只有一个单例，而且，更重要的，，必须提供一个全局域访问入口，这个入口必须是类层的，，

通过这个全局域访问点，你可以直接调用类的某个机

因为它的产生实例的构造函数是私有的，只能从类的内部去产生和获取这个实例,换言之，你不可以通过继承或组合的方法去获得一个实例，而且这个方法往往被定义为 fina,,也就是 C++语言中的 CONST,即子类不能覆盖它，

因此，可以用类方法(也即静态方法)，这种方法下，从继承

9.11 真正的迭代器模式

如果你知道什么叫递归和递推，那么迭代器本身这个概念你是很容易理解的，迭代器跟集合(集合就是通俗意义上的对象集合，虽然存在很多不同质的集合，比如用数据结构表达的对象集，或者用函数索引的 hash 集，但是只要是集合，它的内在总有一些对象及对象逻辑，对象逻辑就是操作这些对象的根据，比如遍历算法，而至于本象本身，可以是无意义的对象，或者同性质的内存节点，或者离散的东东，然而上面说了，这些集合内部必有一种方法作为逻辑可用来遍历他们各自内部的对象，)的关系就是：无论是什么集合，它都可以把一种抽象抽象归纳出来，就是遍历它们各自内部对象的算法
所以，对抽象的提取，往往是找相同的部分，把这些相同的部分提到高层，而用这处抽象来封装可变的部分(这里指各个集合内置的不同的遍历算法),这样就形成了一个所有集合能共享的遍历接口(当然这个接口并不为集合所用，集合自有它们自己的遍历算法，而是为 client 所用，不同的 client 都能面向和共享一个共同的,使者这些集合来进行遍历集合的算法，而不必管这些集合自身具体是如何遍历它们自身的元素的)

9.12 真正的工厂模式

工厂模式用来实例化对象，,，可被形象理解为一个封装了专门用来产生对象的某种逻辑(这种逻辑可以是一个方法的形式存在-这就是工厂方法模式，也可以是一个类的形式存在-这就是简单工厂模式),因为大凡产生对象的过程都是低层的(调用 New 方法实际创建实例对象，属于实现)，它压根就不应该跟高层(这里的高层指的是需要引用那些实

例或间接引用到那些实例的抽象或更高层抽象，由于一个类只能负责一种责任，一个抽象只能被作为一个类，因此当有多个抽象存在时，有必要将它们按职责分成不同的抽象层次，形成不同的层次类放在一起.

这个道理就像：我们生产出一系列的东西(我们当然可以把这个产生过程直接放置到某个未来应用中-这个未来应用要使用到产生过程中产生的对象，这样一来所谓的“某个具体未来应用”就会跟产生对象过程直接挂钩，因此我们把产生对象的过程独立出来，归纳它为专门的产生对象实例的过程，而应用这些对象的一些应用--虽然不知道未来会有多少应用会存在，而这个“不知道”的说法，本身就反应了它符合未来的扩展性--放置到另外一层去)，然而会有其它一系列

9.13 真正的门面模式

门面模式也称为外观模式，它提供一个易使用的接口作为它的外观，只是为了使现有代码 client 和要使用到的对象集(往往是多个具有不同行为不同接口的对象)通过这个接口(制造出的目标接口)能被更简单地使用而已，也即打包某些对象行为(并透露出一个基于高层应用逻辑上的接口),常跟适配器模式放在一起被讨论，因为它们都是为了提供接口而存在的，，适配器模式是转换接口为了“能够被使用”，而门面模式是简化接口为了“更好地被使用”(让被适配对象被 client 被使用，通过一个目标接口-注意这后半部分的说法才最最重要的)

9.14 真正的命令模式

将命令本身封装起来作为一个对象，让它的调用者(注意这个调用者不是客户 Client,Client 是模式之外的使用者，而是命令模式中的一员 Actor，是这个命令模式抽象层中的一层)和命令对象通过对象组合的方法

9.15 真正的模板方法模式

模板方法用一套模式作为定义方法和行为的大致框架(注意是大致，而不是全部，这个机制就允许挂钩，和一些需要它的继承子类实现的抽象方法),这跟策略有一点相似之处，因为他们都封装了作为可变部分的行为，然而它们之间还是有差别的，

然而，模板方法使用继承模式，而不对象组合模式，模板方法因为是一个抽象方法，因此如果有子类继承它，那么这个子类必须要实现这个抽象方法

9.16 真正的适配器模式

Adapter，不是接口的意思，它更准确的意义应该是适配，真正的“接口”在不同的应用场景下有不同的意义，现例举如下：
1,Java 的一种机制，这种 interface 语法是一套抽象机制，如果实现
2,接口类，这些接口类往往是抽象类，
3，二进制复用的接口,比如 COM,也就是构件接口
4,接口方法，某个 class 非 private 的方法（无论是抽象的还是带有实现的都可以称得上是一个接口,一般是指抽象的成员方法）API 都可以是一种接口
5,逻辑模型，通俗意义上的“抽取归纳”某个接口，或者说是高层入口，通过这个高层入口，所有的

以上只是为了不跟 OO 中的接口相混淆，所以强行把适配器模式说成是适配，，其实适配就是适配二个拥有不同接口对象的接口对象(也即这个产生的目标对象“接口对象”也是一个接口，)

9.17 业务与逻辑分开

业务就是你做软工的设计阶段时,所要明确的"逻辑本身",界面就是"表现此逻辑的应用形式"(面向用户的一端),也即逻辑是"我们要搞清的问题"(面向低层的一端),要解决和面向的问题领域,这个所谓的"问题"是严格的,它决定了我们在编码要体现什么样什么维度上的功能.因此在设计中,"搞清你要实现的问题"永远是重要的
另外一个概念是数据,数据处理逻辑要做成独立于界面和逻辑的,此时要提出一个架构,进行新旧系统的分合与整离,让应用统一于某种低层逻辑或界面形式,或由这种架构创建新的应用.
现在的网页,即使它用到 xml 做数据源,也是不完全的"业务与界面分开",我们应保证"如果一个页面被刷新,那么那些不与数据相关的界面元素根本无须变动",这就是彻底的分开
这种理念可以让网页反映速度提高很多倍,,应用的多元化绝对是可以被统一的,只要你能提出一种合理的架构,架构的提出不仅是一种 IT 观念改革(对某个抽象有了新的认识),而且是一种极大创新的活动(人们可以由此发展出很多改变了形式的应用).
多少人明白 domain 这个词的意义呢,如果泛化起来,会是什么意义,任何问题求本溯源就是一个世界,一个领
域)其实 domain 这个词是在原语领域描述事物分类的,每个事物都有一个 name,受某个 domain 类 name 来管理,因此,以什么粒度以什么元 meta 来分类事物并命名以产生一个命名机制呢,就是 domain name 这个词的由来,(元是老子提出来的,古人希望把世界的本质用元这个形式形式化下来)

Java 的源程序文件夹也是这样,你难得找到一种命名为你的所有大大小小的逻辑命名并

人为区别,所以 sun 找了一个 domainname 形式,,分类学与命名学是对软工尤为有意义的,,只有 sun 意识到了它

9.18 你能理解 XP 编程吗

敏捷方法极限编程 XP 和 RUP（ROSE 公司提供的大型软件开发“方法学”）是二种软件开发的方法学。

大设计是一种动用一切资源，从整个思想领域去设计计算机应用的过短，，这完全是一种预设计，编码过程变成了纯粹的被设计预见的集成的一个过程，，这种设计往往首先从思想出发，，完全不考虑计算机实现，语言机制对应用的限制或表达能力，，，提出一种标准和理想模型，把思维过程出现的任何一个过程都作为设计的一部分，设计过程中任何动作都不跟计算机和程序语言相关，，最后仅留一点余地作为编码，在编码时考虑其跟计算机实现和语言机制实现的结合，编码的地位很低很终端。。

XP 编程出来的时候，，人们大呼设计已死，，因为这种边设计边编码（在编码中形成设计）的方法大大忽略了设计超越“编码”的“预",，设计变成了跟编码并行的过程。。

实际上，该如何处理设计呢？？比如设计游戏。。

我们知道设计是无底的，这种无底性决定了我们应有限地把思维中出现的理想设计和想法体现到计算机逻辑和语言机制能表达的逻辑中，，而且应尽早地这样做，，任何应用领的逻辑都要最终被转化成计算机逻辑和语言逻辑。。也即，我们不必做超级设计和完美设计。。

游戏是什么呢？如何设计一个游戏呢，，游戏这个字眼可以无限被放大（根据我们在《应用与抽象》中说到的，应用领域可以无限深化），WEB 论坛可以是文字游戏，3D 游戏也是游戏，，网游也是游戏，，是不是要在你的设计中体现这所有的元素呢(一个具体的设计总是针对某个应用域寻求它在计算机和语言域的对应，如果你知道算法和数据结构你就深刻理解这个说法了，我们总是向语言和 OS 中寻找某种可能适应我们问题的数据结构，即使再通用的逻辑，比如库的设计，我们也不应)，并用一种“设计模式”中的模式来组织这所有的元素呢，，不能，，而且不应该。。你不可能在有生之年把它们(设计中出现的需要组织的逻辑们)的地位作一个组织或你自认为合理的排列。。

你可能会说我不直接提供这些无素的实现，，不直接在设计中体现这些，我只需预见它们，，并在设计中为他们预留接口，，但这样也是不行的

那么最后出来了，，什么是 XP 编程

预设计，大设计是一种“一次性设计”，企图把应用设计中的大大小小所有过程整合到一个大设计中，，这样的代价当实际编程开始时如果遇到不能前进的错误会损失很大，而且设计本身花费精力也不少

而 XP 编程先是提出一个不够完美的框架(针对某个应用，有应用本身和它产生大大小小的其它应用问题，这不够完美的框架是针对整个应用本身来说的)，或者不提出思想模型，，它并不试图分析整个应用，，以及对它们的设计（因为它相信设计不可能是一种大而全的，只能具体问题具体分析设计，人们不应把所有可预见或在后来出现的问题整合到同一个设计中），，并不着手预见可能出现的问题和对它们的大大小小的设计过程，当具体应用问题中的大大小小问题出现时，就着手一个即时设计(比如设计游戏时，这是个具体的大的应用问题，针对游戏本身可提出一个不够完美的框架，，当在他下面遇到有很多小问题，，比如网游时间同步，，我就看语言中提供了什么线程和语言机制，或者如上面说的数据结构或算法，来进行一个小设计)

这就是 XP 编程的全部意义所在。。

9.19 实践方法之极限编程

极限方法只是敏捷开发中的一种,,,软工指明软件开发不只是软件本身,,而是软件跟人的关系,,因此这出现了设计与编码,,,设计与编码是软工的二大主体,,,,好了,,实践方法的出现就是为了解决这二者之间的矛盾
?开放的标准使我们的规范保持中立，所有人都可以接受，而不受某一个开发商的控制，而开源可以使我们得到大家都接受的一个实现，而不受某个开发商的控制，这两者的结合非常有力。这个实现你可以不使用它
开发中的分布要求,产生了二个复杂度,1 网络开发,2 软件要独立于平台和硬件架构,,这就是 WEB 的特征

.

9.20 设计模式复用与框架复用

设计模式是可复用策略,,是思想级的,,但不是不可以用代码来表现(编程即换维重现,即将思想级的东西转变为语言级的东西),,软件设计的终极目标就是符合软工，扩展软件的可扩展能力和生命力，，设计模式就是服务这个的,,设计模式因此有一些原则，比如 LOC，控制反转原则，不重复自身原则,,这就是设计模式对于软工，所要达到的目的,常见的设计模式有哪些呢？，比如 MVC，工厂方法,工厂,单件等，，MVC 可以说是一种框架，，也可以说是一种设计模式，，因为 MVC 是设计模式的组合,,它被作为一种框架时，比如 strcut,spring，也是成立的,,可以说现在的一些 WEB 开发框架比如 STRUCT，

SPRING，ROR 都是设计模式的实作品，而设计模式是一种思想,,设计模式这种思想，这种设计目标和设计手段,，被体现在代码上,，就是用了设计模式的软件,，或用了设计模式的可复用框架,,比如设计模式表现为一种补丁时， 什么时候表现为补丁？就是老总说，某某公司要求我们开发一个软件，但是这个公司提供了一个库，要让我们现有的代码，用一种方法能使它跟这个库协同工作,，因此要利用到设计模式 ,,就是现有代码,，跟可复用的别人的库,,,，在这二者之间用设计模式进行连接,,,，发展出一种可运行的逻辑,,而非补丁式的设计模式的应用，则是在产品没有出来之前,，不需要适配既存可复用库和要写出的代码这二者,，采用的一种预先的,，大而全的设计方法,,非补丁式的设计模式,，是一种真正的设计,,，此时模式二字反而可删掉,,是一种预先想到可能想到的所有扩展能力,，决定采用什么设计模式来编码,，在这编码之前 ,，决定采用什么设计模式，或采用什么别的方法,,，这就是真正的设计,,预先的,，如果可能，尽量大而全地考虑,,,当然设计不仅是面向可复用,，还面向应用设计,，如何设计应用,，如何设计用户界面，如何分析业务逻辑以便于扩展出关于这个业务逻辑的新逻辑,,不仅是在设计如何编码了，还在于调动计算机资源的能力、思维的建模能力、分解和搭架能力,,很多人以为设计模式是补丁其实是很狭隘的东西,,其实设计模式本来就不是编码,,，只是当人们站在编码角度来理解设计模式时,，他立马就错了,,设计模式是一种流于建筑和软件界，通用的可复用策略,,是思想级的，虽然它不是不可以在代码上被表而而已

第三部分 工具与材料篇：Python 代码阅读与控制

如果说在本书第二部分里我力求向你们提出一个思想体系，那么在这本书的这部分我将要讲述的就是代码控制和实践能力了（从这章开始，所有的讲解都会带有代码，这是跟第二部分不同的,这是程序设计的基础二部曲。这本书完成之后，你应该具备基础的编码实践能力（或阅读别人代码的能力,python 代码阅读与控制能力，实践能力）。

从最微小的代码写起，一个看不起小代码，和不注重从微小到巨量积累过程的人，他的实践编程水平一定不怎么样。每一篇代码后都有作业。有的是练习型，有的是创新型的。

这部分主体是,Python 语法,，及初级标准库的分析与使用(如果说语法是工具，那么库里面的类型和接口，就是材料了)，整个第三部分有预计有 4W - 5W 的代码。。

第 10 章 基本 Python 语言

以下的资料暂时摘自《dive into python》，我会慢慢用自己的例子替换掉，这里只是给大

家展示一个样例范本。。

版权声明：

10.1 code fragment 1：定义一个 Dict

```
>>> d = {"server":"mpilgrim", "database":"master"}
circled_1_delcric
>>> d
{'server': 'mpilgrim', 'database': 'master'}
>>> d["server"]
circled_2_delcric
'mpilgrim'
>>> d["database"]
circled_3_delcric
'master'
>>> d["mpilgrim"]
circled_4_delcric
Traceback (innermost last):
  File "<interactive input>", line 1, in ?
KeyError: mpilgrim
```

10.2 code fragment 2：修改一个 Dict

```
>>> d
{'server': 'mpilgrim', 'database': 'master'}
>>> d["database"] = "pubs" circled_1_delcric
>>> d
{'server': 'mpilgrim', 'database': 'pubs'}
>>> d["uid"] = "sa"        circled_2_delcric
>>> d
{'server': 'mpilgrim', 'uid': 'sa', 'database': 'pubs'}
```

10.3 code fragment 3：Dict 的 key 是大小写敏感的

```
>>> d = {}
>>> d["key"] = "value"
>>> d["key"] = "other value" circled_1_delcric
>>> d
{'key': 'other value'}
>>> d["Key"] = "third value" circled_2_delcric
>>> d
{'Key': 'third value', 'key': 'other value'}
```

10.4 code fragment 4

10.5 code fragment 5

10.6 code fragment 6

10.7 code fragment 7：week sumary

10.8 code fragment 8

10.9 code fragment 9

10.10 code fragment 10

10.11 code fragment 11

10.12 code fragment 12

10.13 code fragment 13

10.14 code fragment 14：week sumary

10.15 code fragment 15

10.16 code fragment 16

10.17 code fragment 17

10.18 code fragment 18

10.19 code fragment 19

10.20 code fragment 20

10.21 code fragment 21：week sumary

10.22 code fragment 22

10.23 code fragment 23

10.24 code fragment 24

10.25 code fragment 25

10.26 code fragment 26

10.27 code fragment 27

10.28 code fragment 28 : week sumary

10.29 code fragment 29

10.30 code fragment 30

10.31 code fragment 31

10.32 code fragment 32

10.33 code fragment 33

10.34 code fragment 34

10.35 code fragment 35 : week sumary

10.36 code fragment 36

10.37 code fragment 37

10.38 code fragment 38

10.39 code fragment 39

10.40 code fragment 40

10.41 code fragment 41

10.42 code fragment 42 : week sumary

10.43 code fragment 43

10.44 code fragment 44

10.45 code fragment 45

10.46 code fragment 46

10.47 code fragment 47

10.48 code fragment 48

10.49 code fragment 49 : week sumary

10.50 code fragment 50

10.51 code fragment 51

10.52 code fragment 52

10.53 code fragment 53

10.54 code fragment 54

10.55 code fragment 55

10.56 code fragment 56 : week sumary

10.57 code fragment 57

10.58 code fragment 58

10.59 code fragment 59

10.60 code fragment 60

10.61 code fragment 61

10.62 code fragment 62

10.63 code fragment 63 : week sumary

10.64 code fragment 64

10.65 code fragment 65

10.66 code fragment 66

10.67 code fragment 67

10.68 code fragment 68

10.69 code fragment 69

10.70 code fragment 70：week sumary

10.71 code fragment 71

10.72 code fragment 72

10.500 。。

10.x more coming 。。

第 11 章 Python 标准库

11.1 code fragment 1

11.2 code fragment 2

11.3 code fragment 3

11.4 code fragment 4

11.5 code fragment 5

11.6 code fragment 6

11.7 code fragment 7

11.8 code fragment 8

11.500 。。。

11.x more coming 。。

第四部分 应用篇：Webile 系统开发

是使用本书前面三部分这些知识的时候了，在对这个 webile 的设计中，我将向你呈现大部分出现在本书中的思想或细节，并给出作者写这个框架过程中的一些合理或不合理尝试，模拟一次真正的软件工程(系统开发的 p2p web server 和 high level 开发的 web blog 过程..)！我们的目标是做一个 mobile web 手机！！然而它不是传统的基于 http 的 web, 而是基于某种 p2p 协议的“web 协议”。。

第 12 章 设计（领域分析与抽象）

12.1 了解问题和目标

我们要设计一个 webile 系统,顾名思议,web mobile 系统,这是一个类似 google 手机平台的东西,当然,为了便于能简单地讨论,我们不打算把这个系统作为多应用系统,而作为一个 p2p web 专用系统

12.2 了解领域相关抽象

你理想中的游戏是什么样子呢？什么又是游戏(你对游戏的概念直接影响了你对游戏的设计)，每个人的看法都会不同吧（反问一下，什么又不是游戏，连 BBS 也可以是一种文字游戏，但是我们不做这种超级泛化,我们只对 3D 网游作泛化）？以下就是我对游戏尤其是网游的看法，更准确地说它是框架设计,这就导致了框架的出现。

再来看 Web 和 Web 框架

任何一个抽象世界（无论其大小）都可以形成一个领域逻辑，可被某种语言提供的类型和语法表达，这种语言就是这个领域的 dsl. 或者由某种语言的库来表达，可以称为 ds library. 如果库用到 mvc 等东西，一般流行称之为 ds framework 。

请参照本书选读部分的《真正的 sun 策略》

在 web 这个领域，从开发角度来看，提出了一系列新的理念和框架，还有中间件。除去核心，其它的编程动作往往最后成为配置动作，XML 就是通用的配置语言了。

wsdl 是一种用 xml 表示的东西(首先是一种标准，就跟 soap 一样是种协议，当然现在是 web 编程时代，不要提到网页编程就想到从 http 协议开始，那是上个年代的事了，或者说，是 C++语言的事，它事事都要从头做起。),，是一种配置大于编码的动作。

第 13 章 搭建虚拟开发平台

13.1 Vmware 与 arm 交叉编译环境

第 14 章 系统开发之 Web Server

14.1 Pyjxta

第 15 章 高层开发之 Blog

15.1 Django(DS Framework)

15.2 。。

当然，python 是通用语言，但它拥有 web serverside 库，所以也算是一个 web dsl
无论大小，任何一个抽象世界都可以形成一个领域逻辑，可被某种语言提供的类型和语法表达，这种语言就是这个领域的 dsl.或者由某种语言的库来表达，可以称为 ds library.
如果库用到 mvc 等东西，一般流行称之为 ds framewo。

第 16 章 实现

16.1 demo

这个 Demo 的功能有：

第五部分

选读

会学习的人首先要学历史

任何东西和知识,,到最后都是思想,,所谓道,,,就是思想的意思,,所有的错综复杂的应用细节后面,,支撑它的后台不过几条简单的理论(所以为应用构架的构架要越简单但又不失开放性),当然,一个人不可能直接去领会这些理论,但在学习细节之前,如果能接确到这些思想,,那么这对指导学习细节的过程将是十分有益的,,因为学习如果能是一种有指导的过程,,所谓人的主观能动性,,,,在有理论指导的情况下自然会比没有理论来得有效

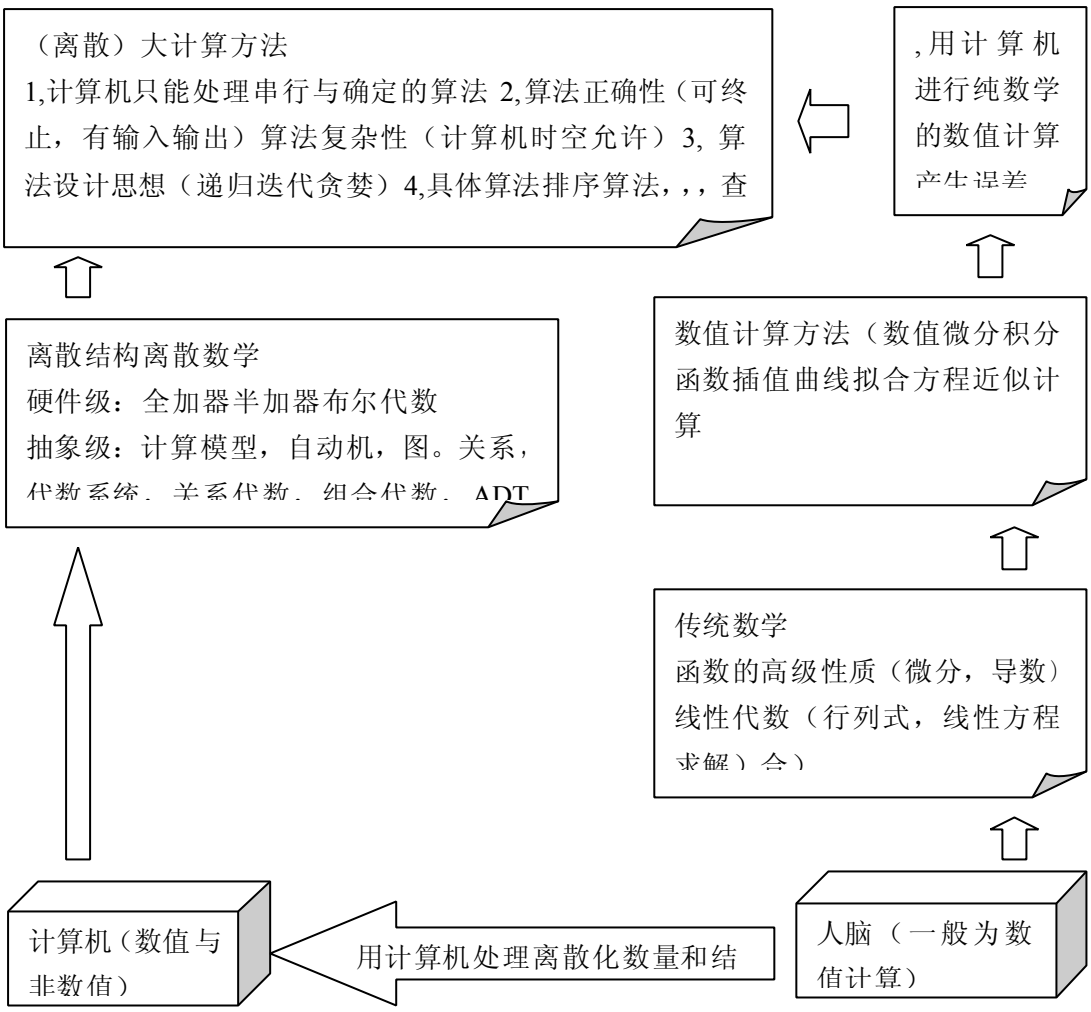
所谓细节和思想,,这个道理就像学历史和社会学,从某个维度某个意义来说,历史学照给我们史实,社会学给我们一些大量历史现象之后的道理,,学习历史就是学习"细节抽象",,,领会社会学条条框框的总结性言论就是"思想抽象",,,由此可见,最好的学习方法是先熟悉一个＂领域＂（应用的全部）的全部细节,，这之前如果有某些＂思想抽象＂作指导,,那么这二个过程可以相辅相成，达到很完美的境界,,
所以，在一本讲解ＩＴ技术的书中,,,我觉得一开始要讲＂历史抽象＂，即某个领域的历史由来,,然后讲各种历史和实现细节，你才能在这个基础上发展出思想级的真知..只

有完成了这步，，学习才是最有效的，最后是形成实际的工作和动手，实践能力
应用与设计

设计模式只是设计中的一个子集,设计模式就是现在的编程技术所体现的那些设计思想,对某个领域抽象的实现,那些不能体现的呢？？？都在原语设计中.我们所看到的概念，领域术语，如果重新被设计，，会产生很多新的抽象，，和由此而生很多应用,并可以形成一门专门的语言,这称为领域专用语言,所以,不光对低层的设计是重要的,光是对"应用本身"的设计都是重要的,,这是软工的需求分析的一个重要步骤,

所以设计绝对不是设计模式,设计模式是编译语言为了脚本引擎的不适配而出现的,是为了让编译语言适应不断变化的需求出现的畸形产物,不要被"设计是为了适应需求"这种软工的局限思想所局限(虽然这是现实的要求),,有时候，，你得承认，应用的复杂性甚至超过低层实现,,我们要站在一个最高的维度去看待设计,,,此时设计不用于重构,因此不会表现为设计模式,,,它更多地是一种预设计,,着力于大而全地考虑所有细节应用的设计(以 IT 人士的眼光来加入特点关注点的抽象世界,,,也称为原语设计),,,一想到设计模式就想到实现,,一想到编码你就错了,只有一开始把设计模式想象成为一种思想,你才能真正理解设计和设计模式

那样这是一种什么样的思想,具体可以表现为领域,原语,泛化,组合,分离,整合这些思想,我会一一讲来
数学不但是算法和解决一切领域问题的基础，别忘了，计算机本来就是一个离散结构！宜结合数据结构阅读本节！它与 OO 的结合点就是利用了 ADT 的算法！



数学与编程

离散数学与代数系统

把本来自成整体的系统（为了让它能适合计算机处理而进行计算模型化）即离散化为计算机能识别和处理的结构的过程称为离散，，用计算机来进行纯数学的数值分析就频频反映了这种现象，

离散数学,,就是把数学理论,,形式化为计算机能处理的东西,什么是离散??计算机本身就是一个巨大的离散概体,,能让计算机来处理数学,就必须把纯理论数学的一些因素形式化为计算机部件能理解的东西,因此会有形式语言的出现,因为很多形式语言的低层知识还是图等数学知识,等它与计算机结合成为形式语言时,形式语言这门学问也就成了离散数不的范围

计算机是一个离散结构，因为它的计算模型是属于离散数学的,但是有人会问了，我们现在在使用的计算机跟图灵机有什么联系，错，还是图灵机，因为现在计算机是在图灵机上构建的更高层抽象（我很庆幸你在第一章的抽象思想没忘），就像冯诺依曼思想一样,它还是现代计算机体系结构的基础，请大胆发散你的思维！

离散数学研究离散了的量，什么是离散?现在我们有足够的知识来更细致地说明它了，这是因为计算机与数学结合后，，传统的数学不再是数值计算了（而且即使是数值计算，，当它与计算机结合后必须被离散化），，，而且一些非数值的计算，，，传统的脱离了计算机处理的数学不可能完成，只能靠计算机，，，因此这个过程中更加提出了一些新的计算形式和计算量，，，统称为离散的量，，也即，，传统数学已经演化到了与计算机结合的离散数学了，，，所有的传统数学问题都可以用离散数学来研究（也即用计算机来参与数学），，，具体表现如下

集合，纯数学中的集合，用计算机来表达最好不过了，因为有布尔代数，，这个代数很好地解释了计算机的运算器的工作方式
逻辑，，图论（不但被计算机，，而且被各个领域的各个分支使用）
代数系统，，关系代数（与数据库有关），，，有限自动机，，正规集（与计算原理有关）

也即，用计算机研究数学和用数学研究计算数，，，，这产生的所有知识都是数学，，都是离散数学

线代与矩阵

什么是矢量，一切量都是矢量，线性代数将传统的代数（主要是数值计算）上升到一个封闭的代数系统（离散数学的一分支），线性代数的提出跟方程组求解这个传统的数学数值方法密切相关，法国的一个青年证明五次以上的方程不可能得出代数解，除非提出一个新的代数抽象或扩展已有的代数系统，线代指出，在给定一组基的情况下，向量空间，比如三维空间与线性空间同构（证明这个结论是代数系统中的内容），因此，用线性代数的知识（线性方程组，行列式）研究向量或用向量空间的知识（矢量啊，矩阵啊）来研究线性代数系统中的量跟传统代数中量是传统和高级的关系，向量代数系统等价于线性代数系统，但一般就说向量代数系统是线性代数系统不分开来说，，这里有一个导出顺序的历史关系，，下面一一道来

以上在高数中我们建立了函数分析性质之间的知识结构，这里在线代中我们同样建立线代知识体系

什么是线性代数呢？
线性代数研究有限维线性空间的结构和线性空间的线性变换(多项式就是一种线性)，
所以线性代数中这个“线性”实际上指“线性空间的结构”和“线性空间的线性变换”
这二个短语中出现的“线性”,(1)对于线性空间来说，这根本是因为 N 维向量空间 F^n 与 N 维线性空间是同构的(代数系统中的一个概念)，如果存在具有同构关系的线性空间 V(F)与 F^n,通常把线性空间也称为向量空间，线性空间中的元素也称为向量

n 维向量空间中的向量关于线性运算的线性相关性（线性空间与向量空间发生联系的地方）

(2)对于线性变换来说，它可以用矩阵来表示，由于线性变换与矩阵之间一一对应(在给定基的前提下)，而且保持运算关系不变，因此，可以用矩阵研究线性变换，也可以用线性变换研究矩阵。

向量空间有（关于它的线性）变换，，线性空间也有变换，，变换即矩阵，故矩阵是它们发生联系的因子.

Core learning and min learing 编程

应该如何写一本关于编程的书呢,保证通俗易懂,而且最核心的部分都要讲到,而且看完本书能形成实际的能力,
1,x86 架构的一部分,给出我们的计算机就是寄存器机器这样的堆栈机概念,那么什么是堆栈呢,也要讲到,因为 CPU 对内存进行控制,导致了 OS 关于内存的逻辑,分段分页,OS 模型抽件抽象层,桌面环境,消息机制,绘图机制导致的开发机制,WINDOWS 中的消息和句柄这样的源程序不公开不好研究,
CPU,堆栈机，软件栈,浮点标准
2,汇编语言的一部分,跳转异常导致的语言机制,
3,高级语言原理,自动机,图灵语言模型,
4,操作系统,进程和并发,
５手工开发时代，C 语言和（数据结构）,算法,创造时代
C 跟 OS 的结合,所以有算法这种概念,当语言发展到 OO 之后就没了,因为平台逻辑没有,算法就是关于平台的特定逻辑.OO 就是关于高级思维模式抽象的逻辑
6,,应用与设计，软工与人,软工开发时代,可复用为一切,算法都有库提供,,复用时代
7,ＯＯ与ＲＵＢＹ
一个实例
中间一棵树,,就是编程,四周出来的根绪是其它领域知识,,这是一片连木林,知识就像树林

真正的二进制

请注意，人们说 JAVA 在语言级就实现了它的多线程，这句话与我们平常说的多线程是不一样的，,它的多线程是针对 JVM 的多线程，而非具体针对我们日常使用的某个平台，，比如 WIN32 的线程

另外 JAVA 也不需要指针（虽然它提供了仿指针），这是因为 JAVA 一般不用于对本地编程(即它不面向 CPU，，也不面向运行它的某个 OS 编程)，，因此指针都变得可有可无了！

为什么我们需要二进制格式呢？因为一方面这是为了存放的需要，二进制相对 ASCII 码格式，也即文本格式的文档(用户的软件叫文档，，但是这里只表示 ASCII 码的文档)存储占用的空间少，装载它所花的内存镜像少，WEB 上传输的统一格式是 XML 文档，可是 XML 文档也有它的二进制格式（这是因为 xml 通常是面向文档的，它跟二进制对比面向人们端的处理方便，，但是面向计算机端的运行效益不如二进制，，因此 xml 通常作为一种中间格式而不是最终的格式来被应用），另一方面，因为二进制就是计算机表示信息的方法（ASCII 则是向人们提供可识别信息的手段,ASCII 更准确来说是一种编码标准而已，有更新的替代方案如 Unicode ）在处理上会快得多，PE 文件就是一种典型的二进制文件，更更一个方面，，二进制格式可以保密（说实话有一些网上传输的字节流只能编码成二进制以加密比如 base64，，base64 是将二进制加密成文本在网络上传输的方法），，当人们需要一种加密的文件格式时，通常都会选择用二进制表示的某种格式，当然你也可以把二进制作为 node 嵌入到 XML 中，这样 XML 就作为外部的统一打包格式（这就是人们常说的“XML 是一种数据的数据,,meta Data” 因为 XML 语法可以用来描述一切领域的数据）

先来介绍各种数据和数值二进制在内存中表示的机制，一般采用二进制的补码形式表示(有原码，反码，补码这几种编码方式,整型最普遍用的是它们在内存中的补码形式,即先取它的绝对值的二进制表示，再按位取反再加 1)

记住只有整型才有有无符号的说法，无符号数和有符号数的这个符号就指“正或负"（不是指二进制表示的符号位但是显然直接影响了它的二进制表示中的这个“符号位"），所以无符号数永远只能表正数(这种“正数”实际上更精确的说法就是“无符号"，而在世俗的眼光里，无符号就是默认的正数，只有负数才需要强加一个符号去表示它为负)，由于只有有符号数才有正负之分,,因此有符号数就有正，0，负这三种“有符号”数，（因为整型数最终要按某种形式，比如原码，补码，反码表示成二进制），无符号数不必考虑符号位，而有符号整型数（按它的补码形式在内存中被表示）就需考虑符号位，进位分为逻辑进位和算术进位，如果你知道汇编中有 cf 和 of 就知道了，CF 就表示无符号数的进位,OF 就表示有符号数的进位，当溢出时就退回到最初(表值范围就是一个周期圈),,

一个多字节的数据的二进制表示有三种方式，书面表示，输出表示和内存表示

比如 0xABCDEF12 这个 16 进制数，AB 是高位，越到后面越是低位(要记住这个,这是书面上的降序说法，一般按内存表示为准采用降序或升序说法，比如 intel 在内存中就是升序，而按输出形式就是降序)，这是它的书面表示(它的输出表示为 12 EF CD AB 由于就是按它在内存中被存储的方式被输出因此就是内存表示)，那么这种存储方式是什么呢，下面道来

内存总是从”前左低”到”后右高”位按递增数值编码的字节嘛,,那么如何把多字节数据的二进制表示存放入线性内存呢？有二种方式,一种是"bigdian（降序，把 big 高位放前面低位放后面就是降的趋势了）"，另一种是"little dian(升序)"，这二种方式是指 CPU 或 OS 的规范(intel 采用降序 big,这样书面表示就跟内存表示不统一了而是相反的)，也即如果从左往右是从高位到低位，就是 small,如果从左往右是从低位到到高位，那么就是 big

真正的整型数

任何一个软件都可采用与 CPU 表示或 OS 平台采用的数值格式不同的数值格式（以符合它特定的要求），比如 JVM 就用全部长为 32 的浮点型，整型等等,,,OPENGL 也有专供它自己使用的数值类型(语言中可用 TYPEDEF 来定义各种数值类型的子集)

数值型是计算机从 CPU 级能直接提供的类型 type,因此可直接拿来使用

BYTE，SHORT，INT，LONG，一般表示 8，16，32 位有符号整型（在计算机内部以其二进制补码形式来表示的形式）

而 CHAR,，WORD,，DWORD 一般表示无符号整型

C++中，int 有 2 种意义,宏观上它指通常意义上的"整型数据类型",包括以下提到的 6 种整型类型,微观上它指[signed] int(注意:以下凡用到[]都表示该括号内的内容是可选的,有就表示它对后面那个词有修饰作用,没有则反之,因此[signed] int=int,单纯的一个"int"都是表"整型"的意思,这是固定不变的),整型变量共有 6 种(1)[有符号]基本整型,即[signed] int,它就是微观意义上的整型类型 int,整型变量最普通的字母表示形式,6 种整型类型之一(2)[无符号]基本类型,即 unsigned int(可以看到 unsigned 没被括上[],它是不可省略的),注意,系统有一个默认的规则,如果不显式地指定基本类型为 unsigned int,即如果不在基本类型 int 前显式地加一个前缀修饰 unsigned------这有 2 种情况:a.写成 signed int(2)写成 int,前面不加任何修饰符(modifier),系统都将把它认为是[signed] int(前面提到[signed] int 与微观意义上的 int 等价)因此,[signed] int 与 int 都表示微观意义上的有符号基本整型,

其它四种同理(3)有符号基本整型:[signed] short [int](4)无符号短整型 unsigned short [int](5)[signed] long [int](6)unsigned long [int]
再来看整数数值数据与它们的对应关系(如何判断一个现实中的整数数值数据的类型,是以上 6 种的哪一种),先要明白计算机的表数范围是由位长决定的(很明显整型在 C 中用 16 位 2 进制数表示因此整型的表数大小为负"2 的 15 次方"到 2 的"15 次方,有符号或无符号实际意义指的是"整数在内存中的二进制表示是否带有符号位",很明显,现实中的整数数值有无正负符号跟整数本身有无符号(是 signed 还是 unsigned)可能有关,但却跟它们在内存中的表示形式,存储形式是否有符号位有直接的关系,判断一个整数是有符号或无符号的决定因素和唯一依据和充要条件是它的 2 进制表示是否有符号位(这里有必要提一下,数值在内存中是按它的 2 进制补码来存储的如 10,它的 16 位 2 进制就是 0000000000001010,正数的 2 进制补码是它的原形,负数的 2 进制补码书上说得很清),而不是它的其它进制表示形式是否有符号或未指定任何符号.

Java 中没有 unsighed 类型，其实归根结底，有没有 signed 或 unsigned 是跟位操作和二进制进位有关的，JVM 中没有二进制的有无符号表示，因此不需要这二种标志符来标识有无进位。

浮点标准与实现

浮点有它的人为标准备,如 IEEE 标准
浮点数可以提供很强的精确度，因为它的表数据范围很大(决定精确度和表数范围的都是浮点数的指数部分)，而且它的小数部分可以精确很多，因此可以表示相差不大或相差很大的东西（比如真实宇宙中二个靠得很近的点，或远几光年距离的二个点）,而整型数就不行(但是在计算货币的时候，浮点数的舍入操作反而不如整型数来得精确)

一种误差形式就是接断误差，用计算机数值来表示纯数学中的数时（或一切现实生活中进行到的数值计算总会产生误差）,,，所以在计算机数值时要避免大成数和小数除数

浮点数在书面上和内存中是如何被表示和存储的呢？在书面上有二种表示方式 1 是用纯粹的十进制表示(但必须带有一个小数点，如 123.这样的整的浮点数)2 是用带有指数的形式来表示，如 12.3E1
当然在存储上，小数点本身和指数符 E 是不必实际存储的(因为它们只用于书面标示并不是一个浮点数本质内容，浮点数本质内容是“小数点的位置应该在哪”“决定这个小数点位置的指数有多大，是正还是负”从这句话可以看出，浮点数的存储表示必须要包含 1 指明小数点的相对位置 2 指明具体的指数部分，实际上 1 和 2 是统一的)，指数正负是必须要被存储的,,,又有一个浮点数书面表示的规范形式，就是小数点前有一位非零的数字,,,,

规格化的浮点数格式主要由三部分组成,阶码(确定小数点的位置)，数符（表示正负号），

尾数（表示机器数字长）

在内存中（只有浮点数的指数形式，因此分为二部分被存储），浮点数除去小数点的数字序列是作为二进制被存储的，一般占 24 位，而包含指数正负的指数部分占 8 位(这个长度为 8 位就决定了浮点数的精确度和实际表数的大小范围),，因此体现在书面表示上，从左边数只有 6-7 位数字序列是有效位数。再长就是没有意义的。

这是因为浮点数在计算机内部或虚拟机内部是以二进制表示的，解决方法是用一种 ADT 而非 primate type 来表示不能舍入的金融数据,, 如用 BigDecimal 包装类型

CPU 中集成对浮点数的处理单元，一般被集成一个被称为 FPU 的元件上协处理器,它是一个栈式的机器元件
浮点数可以有 64 位，因此它的表数范围和精确度都可以达到很大 single ,,,double,,,long int 也只能 32 位而已
在计算机和现实中，浮点数都是用二个部分来表示的，，一个是它的基数部分，,，一个是它的伸缩表示，,比如 23.45 和 2345 都是浮点数，,它们的基数部分是 2345 或者 0.2345,, 如果是 2345,那么 23.45 就是 2345*1/100,,2345 就是 2345*1/1,如果是 0.2345,那么 23.45 就是....
那么在计算机内部，，移位就是 2 的一个幂,这种处理对于计算机来说处理过程是很快的

由于不论基数和小数点后面的尾数部分都是最终的浮点值的表示部分，因此有几位有效位数（不要把有效位数跟尾数部分和指数部分弄混,有效位数实际上就是基数加尾数再除去小数点的那一串数字序列,单浮点数一般为 6-7 位，双浮点数一般为 15 位,超出这些的数字序列就不称为有效位数了）实际上对最终浮点值的影响是绝对不会次于指数部分的，这是因为指数部分仅仅是伸缩度的表示，改变成另外一个伸缩度，有效位数部分就会产生新的基数和尾数部分

另外，整型数被 0 除和浮点数被 0 除的结果是不一样的，整型数除以一个 0 只会产生个异常，而浮点数除于 0 却会产生一个 NaN 或无穷大

字符与字符串

一门语言的字符逻辑涉及到数据结构与 IO，所以是一门语言非常重要的部分。

对于字符，包括基本字符和复用字符(不是字符串，比如'abcd'实际是一个四字节

字符)

记住，，有多字节字符，就是所谓的 unicde 宽字符

由于计算机系统都用数值来编码字符，，因此它们在计算机内部的实现形式通常是 int 值，，，存在不同的标准，比如 ansi,unicode，各个平台也不尽相同

因此会带来不同的赚容问题，在写源程时要注意这种赚容性，，在可执行程序移

殖到不同平台时也得注意。。

在 C 语言的眼光中，字符的本质是 int,字符吕的本质是数组，或指针

真正的 Unicode

原先对字符编码的标准有美国的 ASCII 和西欧的 ISO,中国的 GB 和 BIG5 等，当然还有其它国家的

unicode 组织和 iso 的 ucs(统一字符集 uni character set)小组都是为了统一字符的一致努力而存在的，，当它们走到一起的时候，他们相互妥协并整合各自的成果推出了一些东东
unicode 原来用 2 字节编码 65535 个字符(注意，这只是对字符的编码，并非实际的在计算机内部表示并存储字符，由于 2 的 16 次方为 65535,因此它只能在概念上抽象对应 65535 个字符)，
然而，光就汉字(包括康希字典中的繁体字)来说，65535 个表示就显得少了点,更别说表示其它非拼音语言的字符了
ucs 原来用四字字编码字符，，
后来他们共同制定出来一个“统汉字”
即把 CJK(chinese,janpanese,krean 这些都是汉字的变体)称为统汉字 unihan(han 就是汉字的意思,uni 就是统一的意思)，，把它们当中的一些字符进行整合，简化，删除，硬是把三国汉字弄到了 65536 个之内,即 U+0000 到 U+FFFF

注意这 65535 还是 utf16 的基本代码级别(code plane,想象大型油轮的一等创，二等创，，，,就是 VS 编译时常出现的 C4819 号警告中的那个 code page),还有超出这些分(即 U+10000 开始的)的就用其它级别表示，UTF16 有 1-17 个级别，其它 16 个级别称为辅助级别

这些辅助字符是用先前 65535 个位中其中 2048 个(D800-DBFF1024 个和 DC00-

DFFF1024 个)没用到的空闲位的其中二个组合而成的，并把他们的编码统一到 U+10000 开始的地方作为编码点 code point

关于\u 这种转义符可以直接出现在 Java 源程序中，并且可以脱离单引号(char)和双引号(string)在源程序文本中存在，当然也可放置其中(如果不放置其中就是普通的文本，放入单或双引号内就是作为常量串)

转义字符就是不作为普通字符（可打引，可显示）的控制字符(可是本质上都是字符因此一同编码)，它们也跟普通字符一样被 unicode 编码（因此可作为书写源程序清单中出现的普通文本字符---本来整个就是 UNICODE 文本文档嘛，，也可以作为变量标识名---本来就是字母字符和数字字符和下划线字符的组合嘛,况且 JAVA 还支持 unicode 变量呢）。

并弄出了诸如 utf8,utf16（转换机制,8,16 表位数）,,ucs2,ucs4(存储机制，2，4 表字节数)这样的东东
utf(即统一字符转换格式)是一种转换机制,把非统一字符转换成统一字符的机制(包括一切字符),有 utf8,utf16,这里的 8,16 是 bit,,由于这里不涉及到存储,这里的 bit 只是表示“用多少 bit 就可以表示一个字符的统一字符码”,比如 utf8 就是用 8 位就可以编码达成，而不是表示一个经过 utf8 转换的字符实际只占 8 位,,实际上，经过 utf8 编码的字可能有三个字节长(最小 2-最长 4 个字节，即一定要满足 ucs2,ucs4)
U+0x0000 到 U+0xFFFF 的格式是什么意思呢(U+就表示 unicode 代码点的意思)
0x0000 到 0xFFFF 是编码区位(即在 unicode 码表中的区位，有 2 字节，因此有 65535 个位置供原始字符用，称为代码点 code point)，而 U 就是关于一个原始字的它的 utf8 格式,,它们结合起来就是该原始字的统一编码的存储形式

本地化和语言编码

一谈到编程就学习一门语言的人是正确的做法,因为以语言为突破口(而且要很透)当然存在其它突破口,然而学习语言还是比较行之有效而且大家通用的途径,那么学习其它东西就会势如破处主,虽然语言跟语言要编写的应用逻辑是工具跟作品的关系,但是学习工具,无疑将是一个很好的拐棍,可以助你学到其它重要的知识,这些知识将在各种各样要编制的软件中出现,但绝对不要以为学习语言就是编程的终极目的
就像本地化和语言编码,学习语言的编码机制时,我们就会涉及到计算机平台上的编码标准和规范的学习,

真正的布尔

在 C++中的布尔可以是一切整型值,这其中为 0 即 false,非 0 即 true,然而在 JAVA 中只有二种值 true 或 false

比如 C++中的指针，由于它是一个整型值(可能是 0 或不是 0)，我们可以直接用一个表示非的叹号置于一指针前，然后用整个的取值作为 if 的算式

为什么布尔可以有这么多值或者只有二种值呢，，boolean 只是 int 的 typedef 子集，一般用 0 表假用 1 表真，

下面说一下布尔运算符，布尔运算符即逻辑运算符，（程序语言中还有关系运算符，算术运算符,位操作符等，，然而这里的关系运算符却不是《关系代数》中的运算符，关系代数是一个离数中的代数系统，它自有它封闭而自成系统的数据集合和运算符,《布尔代数》研究 0 和 1 的运算，比如累加器的实现等等）

关系运算符就是比较二个或多个数之间大小，是否相等等关系的运算符
位操作符跟逻辑运算符比较类似（不一定只有二个运算子参与逻辑运算或位运算，而且它们的运算规则也比较类似，而且运算符也比较类似逻辑是双体号而位操作是单体号），它们之间明显的区别在：逻辑是命题相关的，因此跟短路规则有关(这是说在 if 这样的条件句中)，而位操作符跟位操作有关，因此不进行短路规则的判断

逻辑与即逻辑乘，见 0 为 0，全 1 为 1, 逻辑或即逻辑加，见 1 为 1,全 0 为 0, 还有逻辑和位操作中的 XOR，也即只有运算子不同时才能取到 1,相同时为 0

位运算主要是利用屏蔽技术取得整型数（的二进制表示）的某些特定位,运算规则跟逻辑运算是一致的

因此在位屏蔽中，用 1 与“与”的组合可判断某位是 0 还是 1,因此在位屏蔽中，用 0 与“或”的组合可判断某位是 0 还是 1

位移和位段,, 移位有单向和循环移位，

Minlearn(1)平台之持久

把内存逻辑模式转为硬盘可持久模式的机制就是可持久机制,,像文件啊,数据库,都是可持久,,,对数据库的开发(对其开发)经历了一个从偏向数学,机器到靠近人脑的 OO 过程,,

平台之多媒体编程

实际上如果你有过写小说的经验就好了,写小说可以先构思好故事内容再以后慢慢添加,厉害的作者甚至可以边写边想,到最后居然也能形成一种故事体系完备的书,当然,这因人而异,我们从中可以发现一些对软件开发的道理
大设计就是这样的一种设计,它是一种预设计,(在软件设计中的设计)关于此应用的考虑到所有的因素,并 OO 化,或者用其它范型实现,当然这种设计过于复杂,而敏捷方法说比较接近即时设计即时编码这样的方法..
为什么 Linux 下没有很多游戏呢,这原因有以下几点:
1,历史原因地,Linux 在桌面的发展没有 WINDOWS 好,导致了这个现象
2,技术上,Linux 下的多媒体编程多么差劲(而多媒体应用是桌面应用中多么重要的一部分啊),是个人都知道,,多媒体编程跟游戏编程直接相关,这又导致了 Linux 没有流行,没有一个类 DX 的媒体访问底层来被用于制作游戏(SDL 算是一个),你应该知道 linux 下播放 MP3 都卡,那是因为没有一个好的直接媒体访问层,没有供 MP3 应用可用的支持库,还有,Linux 下的游戏居然用 QT+,GTK+ 这样的界面库来开发(比如 CALL TO POWER),QT,GTK 是什么东西啊,这相当于 WINDOWS 里面的 GDI 啊,多么差劲啊,我觉得 linux 下的 SDL 甚至比 WINDOWS 下的 DX 还好,因为它没有采用 OO 的机制,IDL 的 COM 机制等,是一个纯 C 的东西,如果用 RUBY 来封装它,进行上层开发比较合理
在我眼中,跟硬件相关的都用 C,影响机器执行效立的都用 C,非得用 C 的都用 C,而属于人的逻辑,一切高层应用逻辑都应用 RUBY,世界上只有统一了这二种语言就行了,最后将 RUBY 发展为软工语言,将 C 发展为标准的硬件语言..我觉得语言统一了,应用才能被统一.
但是我后来否决了这种看法,,历史上地,一种应用一种标准应由某种语言撑起来，比如 QT 撑起 GNOME，往往不是语言来决定应用，而是反过来,故 C+RUBY 不全,,,不能期许这二种语言成为唯二的通用语言..

图形原理之位图，图象和字体

一般情况下ＤＸ这样的"直接访问硬件"的库不宜用来作一个ＯＳ中的桌面开发，,，为什么用ＧＤＩ这样的库来完成呢，（比如ＷＩＮＤＯＷＳ中),，这是因为ＧＤＩ是高层的，可以跨平台，在开发时不需要动不动就去访问硬件，而一旦访问硬件，，这种说法一般都体现了它跟平台的相关性，而桌面开发应该属于高级应用，，不跟游戏开发一样，必须要去直接访问硬件才能体现其运行的效立．这就是什么应用适合用什么逻辑来开发的道理．.

像位图的显示，字体的矩阵显示，等等，都是计算机图形学的内容了(要是你知道为什么在画图中划一条斜线有锯齿形状就知道线是用数学矩阵模拟的)，有硬件的相关性和复杂的数学基础，但一般在显示字时并不访问硬件，，而是用了高级的库

真正的 GUI

GUI
图形用户接口， Window 的界面就是一种 GUI，而不是一种 console,(UNIX 下的 GTK 等也是一种 GUI)用在游戏中的 GUI 是基于 DX 来实现的,因此它们也参与渲染叫 DXGUI（要知道，在某些要求十分实时的大型应用中，有时只能 CONSOLE 界面，GUI 界面会成为一种瓶颈）
可是它们是 DX 下的 GUI，虽然可用 GDI 来摸拟（但是这不是一个纯净的基于 DX 的 GUI，而只是一个，比如，在 GDI 中绘制，然后绘到 DX 页中的东东）而且 DX 下的 GUI 控件有它自己的属性（比如特效啊），这些都不是一个 GDI 控件所能达到的
，
JAVA 中有它自己的一套界面机制，这就是 SWING,它的原理是 MVC(界面-视图-控制机制)
但是这些东东要独立处理消息，你发现一些游戏中 GUI 很难响应消息，这是因为信息不能及时到达(界面有它跟线程和图形相关的代价)

window frame 是指一个窗口的基类，它存有这个窗口的矩区大小信息，工具栏，菜单栏的位置信息，所以它并不仅仅是一个容器，前面说了，它是一个逻辑上的基类，而不是一个物理上的容器
window base 就是指 edit 啊， button 这些组件

通常在一个基于 DX（或某种其它的渲染接口）的应用(utility 类 APP)或游戏(Game 类 APP，这类 APP 较 utility 不同的时它要求实时消息密集型刷新)中时，这二者所要求的 GUI 机制是不一样的,这就是为什么窗口模式较之全屏模式来说，全屏模式下可以应用渲染的大部分资源 assert,而窗口只能应用一部分适合用做 Utility 类 DX APP 开发

本质上你也可把 gui 作为场景中的 static mesh 对象(ogre 中不可 moveable 对象实体)进行渲染，就跟渲染静态 terrain mesh 一样，控制面板,top view 常常被作为静态 mesh 来渲染(Render to texure 技术常常把全景地图绘到一个 texture 上作微缩地图，因为一个 texture 本质上承载它的 assert 也是一个 surface 而已)

Linux 与 3D

在 Windows 系统的内核级就集成了 hal 层，这是一个硬件抽象层，Gdi 和 dx 作为二维图形用户界面和三维多媒体加速虽不属于 Windows 内核的，，但都工作在系统底层。。想象一下，gdi 和 dx 是 Windows 系统的系统组件。。
GUI 机制只需要软件加速(在没安装显驱时我们也可以看到桌面，这是因为 GUI 机制是硬件独立的软件上的东西,是一门 OS 必须要实现的，而三维加速可以放到后面发展，为了展现多媒体它可以作为硬件相关的或软件渲染，因为它毕竟不是 os 表现 GUI 必须的，一般情况下，操作系统为这二个目的开发不同的库，Windows 是 gdi 和 dx,类 linux 是 x11 和 gl)

3d 图形加速为什么不能做入系统内核呢，，这是因为操作系统的桌面环境和 GUI 机制只能是用户级的(而 Windows 在内核就集成了一个 hal)，，，因为只有在用户级，才能不出现因为崩溃而把整个操作系统弄崩的现象出现(而 opengl 等属于硬件直接访问的)。

而 linux 系统的内核级除了驱动程序之外，完全没有集成硬件抽象集，，x11 工作在应用层，，加速库只有 opengl,,一般说到 linux 系统的 opengl 加速，就是指它的软件加速(opengl 有三种工作模式)，，linux 下除非你下载到了支持显卡硬件加速的驱动时，并且在配置好了桌面环境 x11 的情况下(通过开启一个 dri 的东西)，，才有可能真正开启硬件加速。。一个很可惜的事实是，linux 下支持硬件加速的显卡和显驱少见。。这二个情况造成了 3d 游戏在 linux 上的不普及。。

设备环境

首先要明白一个常识,Windows 是早于 C++的(这里说的是 C++的 ANSI 标准和它的 OO 泛型的引入,诚然用 C 也可实现 C++所有能做的事情，但它比起 OO 离我们的思想要远得多得多),并独立于 C++之外,因此,Windows 的某些对象(注意这个说法"Windows 的对象")并不都属于 C++封装技术的规范之内,即 Windows 的这些对象是 Windows 作为一个 OS 本身的对象,不是由某个 C++的类创建而来,但是在 C++里却可以用一个类封装这个(或这些)对象,并创建它们,MFC 的 Objects 就是 C++的 Objects,(设备环境就属于上述的对象之一,它是 Windows 的对象,MFC 用 CDC 类封装了设备环境对象,注意有些教科书也称设备环境为设备场景);

字体都是 GDI 的，像素点都是矩阵来模拟的因此会有锯齿状
如果存在这样一个编程需要:要求用户自己编写一个函数在屏幕上画一条直线,在不使用到 MFC 等任何既存的绘图操作函数的条件下或者说在 MFC 出现之前,用户要独立编写这个函数,一般的建模样式是:函数名(起点 x,起点 y,终点 x,终点 y,线条颜色,线条粗

细),即:DrawLine(x1,y1,x2,y2,color,width),为了方便,还要考虑将直线绘制在任何设备上比如打印机图形文件等,而不仅仅是屏幕上,另外,还希望不必考虑同类设备的型号等因素,如果综合考虑上面的功能上的条件,函数就成了:DrawLine(x1,y1,x2,y2,color,width,device,limits,units),这个函数虽然相当直观,但对它的多次引用将造成巨大的资源消费(系统的开销分为时空二部分,时间指 CPU 的处理时间,空间指内存和硬盘的占据空间,多操作系统给每个用户分配限量的资源配额,如 Linux),每当执行这个函数绘制一条直线时,上述参数作为变量被压入内存区,如果要绘制第二条直线,这些参数将重新被压入内存,当需要画很多条直线时,这些变量将频繁出入内存,CPU 的处理时间和内存的占用显然比较大,另外,还有一个问题,上述函数如果要扩展其它的功能,必须对 DrawLine()编辑参数项,这将造成函数的冗长及封装上的不便,即函数 DrawLine()没有保留扩展功能的余地,有没有一种方法,当系统引用该函数画完一条线后,以后不管再画多少条线,将不再重新将这些函数压入内存,同时让这个函数有保留扩展功能的余地??

一种方法是,把这些参数中的大部分作为一个 C++类的属性来进行封装,(对象包含二部分,行为和属性,通俗来说,行为就是函数,属性就是变量,对象和类都是面向对象的程序设计语言模拟现实生活万千事物的技术之一,类是对同属性同行为的对象集合的封装,类也包含它的成员函数和成员变量,类对相关对象的封装是逻辑上的,而在物理上,一些可以归类的类可以构成类库,类库就是软件包如 MFC),从而可以由这个 C++类派生对象,在 Windows 的绘图操作中如 DrawLine()中,再将这个对象作为 DrawLine()的一个参数,这样,具体到程序中,应用 DrawLine()画绘制一条直线,只需要一个这个对象的指针(由于对象在内存中的地址是不断变动的,系统用一个指针来指向它的地址)和提供画线的起点和终点坐标就行了,即:DrawLine2(对象的指针,x1,y1,x2,y2), 于是不管画多少条线,在上述绘制直线函数 DrawLine2()中,注意不是 DrawLine()只要将除 x1,y1,x2,y2 之外的变量压入内存一次就行了,并且这个函数有扩展功能的余地;

这个方法("一种方法是..")便是 MFC 采用的方法,这个 C++类便是 MFC 的 CDC 类,可以看到,上述"这些参数中的大部分"往往是一些 Windows 绘图操作函数中如画线画形状填充要调用的全部变量的公共不可变部分,其它的刚是可变部分..

以上说明了把 API 封装为对象的好处所在

平台之数据库

应用是严格的，所以一个真正产品级的数据库系统都要有很强的回滚机制啊，提供快速的查询反映。数据库系统是用它的应用来定义的。比如它要支持什么能力的应用。

数据库系统（这里提到的是狭义的，指具体某种数据库比如 SQL 及基于它的某个数据库管理系统比如 SQLServer，广义的数据库系统包括人员，硬件，软件等，而数据库管理系统是一个数据库系统的核心软件，因为某种数据库的某个数据库管理系统几乎是这种数据库的代表，因此在说到数据库系统时一般就是指它的某个数据库管理系统）与

文件系统都是为了对数据的存取和管理更方便.注意这里提到了存取和管理，更准确地说，数据库(管理)系统是科学地组织数据(用三个模式来对数据进行抽象)+高效地存取和维护数据(查询处理器和存储管理器,在 SQLServer 中是查询分析器)
因此一个数据库(管理)系统首先要抽象数据，再向用户提供它的存取和维护接口(维护比如转化，导出，备份)
对数据进行抽象不但包括对数据本身进行抽象(概念模式)，还要对数据的存储结构进行抽象(内模式),，还要对数据基于用户的那部分进行抽象(外模式)

数据库结构对泛指意义上的所有数据库的结构进行描述，为了提出一个数据库结构，及为了对这种具体的数据库结构进行讨论，发展了数据库的三级结构,,即内外和概念模式
概念模式：
数据模型是数据库结构的基础，所以提出了数据模型这一概念，它用来描述数据的一组概念和定义，最常用的数据模型分为概念数据模型和基本数据模型,,前者是用用户的观点对数据和信息进行建模，是现实世界到信息世界的第一层抽象，其中最著名的是实体-联系模型，简称 E-R 模型.
后者是用计算机的观点对数据和信息进行建模,有层次型，网状模型，关系模型，面向对象的数据模型,,对象关系数据模型(最后二种是最新出现的高级模型)

概念模式被简称为模式，与数据模型相关，不实现访问控制等这些外模式做的事情,比如关系数据库的概念模型就是关系模式的集合。在SQL 中，关系模式称为“基本表 table”,,,内模式即存储模式被称为“存储文件”(也即数据库系统所支持的数据类型比如 varbinary,varchar,image 等)，外模式即子模式称为视图 View(一个或多个表的集合),元组称为行(一个记录)，属性称为列（一个字段）

内模式：
数据库是与文件系统密切相关的，也称为子模式，数据库按它的内模式存在硬盘上，与文件系统有关，微软的下一个文件系统就是一种被称为 WinFS 的数据库文件系统

虽然说数据库系统克服了文件系统数据联系弱，数据不一致和数据冗余大的特点，但当数据库在几百 TB 大时，文件系统的优势就比一般比数据库还好(几百 T 大的数据库在大型应用中是很常见的要注意)，故虽然在数据库中可存储图像，二进制数据，但一般在数据库过大时（并且图像最好直接放硬盘上进行编辑的需要),,，并不将这些东西放进数据库中

在 SQL 中，内模式即存储模式被称为“存储文件”(也即数据库系统所支持的数据类型比如 varbinary,varchar,image 等)

外模式也称用户模式或子模式，是用户与数据库系统的接口，是用户用到的那部分数据的描述(抽象),用户通过这个模式与数据库系统进行交互

数据库结构的三个模式（外模式，模式和内模式）都是对数据库结构的数据部分定义，在数据库管理系统中一般有三个功能 1 数据定义功能 2 数据操纵功能 3 数据库的运行管理,数据库的三模式即第一部分功能(这就是数据库系统的数据字典).

既然，在一个数据库系统中，数据库结构的数据定义部分是由数据库管理系统来提供的，,因此数据库管理系统不仅仅是一种软件而更像是一种规范(比如 SQL).

本书第一部分主要讨论了跟关系数据库有关的一些关系代数的运算来调用. .

真正的可持久化

文件就是可持久，，普通文件和数据库都可以用来可持久

比如 delphi 的二进制窗体文件，它将控件 OO 对象存入一个硬盘文件，以后编译时可加入到代码中来，这就涉及到 OO 的持久化问题！！专门用来定义可持久 OO 对象的语言称为 ODL!!
有一个于 2002 年解散的组织创立了一个 OQL,对象查询语言，然而它被放到 OO 数据库就是不必要的中间抽象,这是为什么呢？

UDT 是用户数据类型，新建数据类型,类,这是一切对象的基础，归根结底程序是操作和数据，，但是程序语言发展到现在，现代的程序只有一种元素那就是数据，和数据接口，（操作和数据在现在被数据和接口代替）因为面向对象已经把操作和数据都封装起来作为“一般数据类型”（这就是 class,通常意义上的 typename 就是 class ）一个 INT 是一个数据类型(如果 INT 和 ADT 都是对象，那么就可以为 ADT 定义跟 INT 一样的支持动态 bind 的算术操作符，如加减，自加自减)，一个 vector 是一个类型，一个自定义的 class 也是类型，甚至一个函数也是类型(因为存在函数对象的说法),过程对象等等,即，，在面向对象的技术规范里，一切皆对象数据。下面开始把数据类型说成 datatype

左值与右值的概念就来源于此，考虑以下一个式子，int a= int b,,, 把数据 b 的值赋给 a 的地址所指的一个空间，datatype 的值部分就称为 datatype 的左值，,，datatype 的地址部分就称为 datatype 的右值
数据库处理对象数据的持久化，在数据密集型应用中，关系型数据库可以办到，，OO 数据库就更好了
在内存中是流式对象，对象会变成字节流，通常会发给另一台机器。而对于固定对象，不同语言实现的对象之间是不同的，同一个语言不同编译器也会有所不同,对象保存在磁盘中。即使程序中止运行，它们仍可保持自己的状态不变。对于这些类型的数据存储，一个特别有用的技巧就是它们能存在于其他媒体中。一旦需要，甚至能将它们恢复成普

通的、基于 RAM 的对象。这就是 OO 数据库要解决的问题

也即，Datatype 就是泛化意化上的数据，它跟持久合结合产生数据库，跟内存结合产生数据结构学

为了使用某样东西而设立使用者把使用对象之间的抽象，，称为对接口编程

在关系数据库时代，过程性语言跟关系数据库的程序逻辑中的数据模式是相对应的，然而当程序语言发展到了今天的 OO 时代，如果再使用关系型数据库就像把车放进车库里之前把它拆散了，明天开的时候再把它装起来(一辆车本来就是一个整体，而且是具备一个逻辑可运行的整体，仅仅是为了存储的需要而把它拆散是不必要的抽象，因为明天还要把车推出车库再开嘛),,OO 数据库里的数据都是 OO 对象的实例,这种数据模型逻辑完全对应了 OO 程序的程序逻辑，而且省了中间查询语言这层为了使用而设立的高层抽象,,用编程语言的本地语言就可以，并且可以保证代码的类型安全，

而在关系模式中，查询语句往往作为字符串被传递到程序，，虽然有一些机制可以保证它的代码安全性，(有用某种语言写的存储过程,比如用一个 Java 会话 bean 写成的会话 bean)然而大部分情况下,一些东西还是不能被避免的，比如数据库存在一个程序中并不存在的实例引用(对象应该只包括变量与操作，但是考虑其与本地的接入点，内存引用指针来说，一个对象就应该包括这三部分了)，数据库依然会为它产生一个查询(而只有在 type safe 机制下才能保证一个未被引用的变量引用在编译时被调试到)

真正的文件

文件被存放在硬盘上(作为二进制)，它有一套地址(可以用一个十六进制工具直接打开，并针对某个内容-这个内容必定有一个外存地址，进行编辑并保存，注意不能用记事本之类的，因为计算机只能以 ASCII 格式保存，会破坏原文件的地址)，文件被加载入内存之后(map)，又有一套内存中的地址,,一些静态分析工具就会求出这个地址对应它在硬盘上的地址(偏移地址加镜像基)

在 Java 的眼中，文件跟内存跟网络一样都是一个流源，有索引文件，Hash 文件，顺序文件，块文件等这跟数据结构有关

文件跟算法的关系更密切，因为计算机的算法本质上不是纯数学的（它是真实以计算机的时空资源为代价的）,,，就像计算机的数值学只能是离散的而不能是脱离计算机的（实际上数值跟计算机学结合是扩展了还是？一句话：数学这个概念并不是我们想象中的纯数学，与计算机结合了的数学才是真正的数学）文件就是空间资源

真正的数据

从很大意义上来说,计算机是一种模拟现实的工具,计算机对现实中的数据(字符,音频,视频)的模拟实现手段是"编码",如音频,视频就是编码实现的,字符包括字母和符号,也是用ASCII 码来表示的,用 0 和 1 的不同组合来表示的 ASCII 码是一种低层的二进制级的编码,音频,视频较之可能是高一级的编码方式(基于算法的，基于某种其它抽象的),但是数据不只有字符数据,还有数组数值,它们称为"普通数据",现实中的数值在数学(专门研究"数"的科学)中分类为:(2.bmp)
或者有理数分为整数和分数,其实给实数和有理数分类的方法有多种,关键看什么标准,由上可以看出,实数是用小数来描述的,也就是说,可以把所有的实数理解看作为小数,换句话来说,数的本质是小数,数的小数形式可以描述和表示一切可能存在的实数数值,换言之,小数与实数等价,在数值上有一一对应的关系,因此用计算机来表示数值数据只要表示了小数就成功地表示了一切数学中可能出现的实数(实际上这也是由计算机内部结构决定的，因为 CPU 内置浮点器,而且对浮点数的处理足够高效),也就表示了现实生活中一切可能用到的实数,因此在计算机也称实型.

就简单数据类型来说有数值型， 字符型，（货币型，日期型），

但是只有 256 种字符型能直接被 CPU 处理（与整型一一对应的那些，比如字母，转义符等），其它需要被通过其它抽象编码的字符，比如汉字，是不能直接被 CPU 处理的

XML

xml 是文本语义的 WEB 语言
XML 首先是一种想法,一种标准,然后在由它发展的支节逻辑中,才有图灵完备的语言存在,XML 本身并非一种具体的技术,而是一种标准
我们知道 XML 并非现代意义上的语言,它只是标识语言,现代语言的模型标准是图灵机,在实现上具备图灵完备性的语言才能被称作语言图,,,换言之灵完备性的语言才叫真正的语言，而 XML，HTML，SQL，之类的不算
DTD 和 shema

这就是元语言，

问题域永远是多元的，因此需要元语言，数据格式是多样的，因此会有表示数据的通用方法 XML,一般用 XML 格式的 metadata 来封装其它格式(即：不可能把所有格式的

数据包括二进制啊，PE 啊，，TXT 啊都做成 XML，但我们可以把这些数据格式放进一个 XML 的字段内，XML 就成了一种通用的数据打包形式并用它用于进行统一数据交换过程，然后在使用时从这些这段中取出各个格式的数据再解码成所需的格式,W3C 组织正在策起一个关于 XML 二进制格式的方案)。

xml 是什么东西，，文档交互的标准，，更标准的说法是 dom,,(dom 是一种技术实现,xml 是一种能发展为领域语言的应用领域的综合，sun 很喜欢做构架，因此它从思想级泛化出一个＂标准＂，，一个架构级的东西，然后慢慢发展成软件实物和编码产物)，
不要小看这个ＤＯＭ,这个 dom 可了不得了,，有些复杂性是历史原因，所以可以重新设计一个架构解决这种历史遗留问题，这个 xml 架构是符合简单性，开放性和实效性的，因此 xml 是历史所趋的．以至于最后要发展出很多术语比如 node,root,，这些术语就是 dom 领域的＂专用抽象＂（在更多维度上可能会有更多新的未发现的抽象，但人们出于应用的目的，仅仅将他们的眼光投放在这些维度点上)，，这些领域用词可以被用来设计出一些计算机语言能具现的ＤＳＬ，，因此有很多更多的ＸＭＬ语言级的抽象出现了，高阶低阶的，，比如 schma 等等（我个人是讨厌学习新的东西的）
以前在互联网上文本有各种专用格式（我这里指文档而不是二进制）,,ＩＥ浏览器和 nestape 支持不同的格式，?ＸＭＬ格式的文档能迅速和有效地被计算机处理，因为它是一种属于图的离散结构，但是ＸＭＬ厉害的地方不仅在于它实现了这个功能，而是在它实现了这些功能之后．．因为ＸＭＬ将文档结构发展成一个领域，以至于后来有二进制ＸＭＬ的出现等等，所以应用是无限扩展的，作为学习者，如果不能掌握这些＂细节抽象＂，我们就只能掌握＂思想抽象＂和＂历史抽象＂

ＸＭＬ是不断发展的应用的产物，而且是特定应用的产物（文档交换），因此有很多其它方面，它就是显得有点畸形和笨，而ＸＭＬ本身是面向计算机理解的，它对人的可读性也不是很好的，，比如 web 开发的部署中，我们常用 xml 来实现一种＂按需更改，即时修改即时启用＂的效果，这样就会导致不好的现象．

真正的 XML

XML 其实是先是面向解决数据异构问题出现的，，如果想理解 XML，不理解它这个历史背景的话，那么你得到的认识会以为 XML 仅是一种新的理论加实践，而不知道它产生的根本和存在的本质。

xml 是什么东西，，文档交互的标准，，更标准的说法是 dom,,(dom 是一种技术实现,xml 是一种能发展为领域语言的应用领域的综合，sun 很喜欢做构架，因此它从思想级泛化出一个＂标准＂，，一个架构级的东西，然后慢慢发展成软件实物和编码产物)，

不要小看这个ＤＯＭ,这个 dom 可了不得了,，有些复杂性是历史原因，所以可以重新设计一个架构解决这种历史遗留问题，这个 xml 架构是符合简单性，开放性和实效性的，因此 xml 是历史所趋的．以至于最后要发展出很多术语比如 node,root,，　这些术语就是 dom 领域的＂专用抽象＂（在更多维度上可能会有更多新的未发现的抽象，但人们出于应用的目的，仅仅将他们的眼光投放在这些维度点上），，这些领域用词可以被用来设计出一些计算机语言能具现的ＤＳＬ，，因此有很多更多的ＸＭＬ语言级的抽象出现了，高阶低阶的，，比如 schma 等等（我个人是讨厌学习新的东西的）

以前在互联网上文本有各种专用格式（我这里指文档而不是二进制）,,ＩＥ浏览器和 nestape 支持不同的格式，　ＸＭＬ格式的文档能迅速和有效地被计算机处理，因为它是一种属于图的离散结构，但是ＸＭＬ厉害的地方不仅在于它实现了这个功能，而是在它实现了这些功能之后．．因为ＸＭＬ将文档结构发展成一个领域，以至于后来有二进制ＸＭＬ的出现等等，所以应用是无限扩展的，作为学习者，如果不能掌握这些＂细节抽象＂，我们就只能掌握＂思想抽象＂和＂历史抽象＂

ＸＭＬ是不断发展的应用的产物，而且是特定应用的产物（文档交换），因此有很多其它方面，它就是显得有点畸形和笨，而ＸＭＬ本身是面向计算机理解的，它对人的可读性也不是很好的，，比如 web 开发的部署中，我们常用 xml 来实现一种＂按需更改，即时修改即时启用＂的效果，这样就会导致不好的现象．

Minlearn(2) 平台与并发

各种平台上的并发逻辑,,一般 OS 基于 CPU,而应用程序的基于开发它的语言内部的并发支持,OO 语言和它后面的东西,设计模式,,,,OO 语言跟过程式 C 语言的关系,就跟 Windows 跟 LINUX 的关系一样,,他把一切逻辑都用界面来封装,人们操作界面来调用计算机的内部逻辑,但实际上,Linux 的 SHELL 能提供的逻辑才最接近计算机的内部逻辑和强大的逻辑,CUI 比 GUI 有着更强大的逻辑,就是学电脑的新手不够适应,因为 GUI 出现的更多的逻辑也就导致了,为了解决 GUI 问题而出现的线程问题是线程的应用之一一也就是其中一个..有人说 UNIX 比何比 WINDOW 好,实际上 WINDOW 就是在 CUI 上套个 GUI,再对这个 GUI 做多了文章,导致了 GUI 之后的很多特定逻辑,而这些逻辑在 CUI 下根本不被考虑,,,开发中不需考虑这样的逻辑..

真正的并发性

并发性可用进程或线程来实现

线程被称为轻量级的进程,，一般由软件或用户来实现（当然也可由 OS 来抽象集成）,,

相比来说进程是重量级的，因此进程被用来进行重量的基于任务的资源分配等，而线程被用来进行轻量的基于函数的任务，，进程一般由 OS 直接抽象集成，，其实有一些软件会完全采用它们自己的进程和线程模型，，，，因为这二者在底层是依赖一些相同的东西比如长跳转,非局部跳转等机器操作。

在语言级，C 和 C++都没有实现一个并行库(并行库并不一定要在 OS 级实现)，，因此很多软件都自行实现了它们(比如 Java 就会有它自己的线程支持)，如 posix api,

信号量就是旗语,Db4o 中的旗语就是信号量,像数据库，网络,企业集成中间件(比如 EJB 容器),这样的要求多对一的应用总是会用到并发控制，并发控制其实在很多场合都用到，只是它们往往被高度封装因此我们看不到它们而已。

Java 中对线程的讲解

Volatile 原来在汇编中出现时，就是指对硬件内存的保护，如 AGP 内存（shadow memery），volatile 是用来修饰访问硬件内存的(硬件以控制器作为它的 CPU，，以内存作为它的存储器与中央 CPU 交互，，当然 GPU 是独立有显存的，因此称它为硬件内存,,硬件缓冲)，利用这个关键字可以禁止编译器对该段代码进行访问

EJB 容器在一定程序上就像一个数据库(还有容器查询语言来着)，，，因为容器也要涉及到持久机制，也即对容器内的对象进行持久，因此它也要提供跟数据库系统一样的持久机制，，事务机制和安全机制等等

而一个完整的数据库对于这些机制也是必不可少的(基于一个数据库系统还要处理跟线程有关的并发机制)

Minlearn(3)载体逻辑

可执行文件在 Linux 下是预编译包,比如 DEB 下是 DEB 包,Linux 有二种生成可执行逻辑的方式,一种是 DEB 包分发,一种是从源程序构建(这得益于 Linux shell 下强大的 COMMAND TOOLS,比如 MAKE,这说明 OS 跟开发之间的密切关系,WINDOWS 似乎并不是一个为开发而结合的系统,这就是人们常说 LINUX 是怪人开发的给怪人用的 OS 的道理),
可是在另一方面,WINDOWS GUI 后面的那些东西给我们学习最简单的计算机原理带来了障碍(甚至是偏离计算机本来面目的惯性思维,相比之下,对 LINUX 的学习就比较接近计算机本身),当要对计算机进行再高阶的开发时,又不可避免地要涉及到计算机底层(你

看,再高级的 RUBY 都要学习数据结构这些底层思想),,这造成的结果是罪恶的,,这就相当于穿上一双做得不彻底的鞋,跳出去,,却最终没能跳得彻底,只能跌倒，照样使得脚受伤. . WINDOWS 在封装复杂性上做得不够彻底，，导致了二面性. . 所以,评论 LINUX 和 WINDOWS 谁好谁坏,只能是一种在各自的领域里为自己说话的动作,,,实在不值得为这个花时间

由于ＬＩＮＵＸ跟开发直接相关，，可见这种机制不但用于分发,而且还用于执行,,为什么 Linux 下提供这种机制呢,要知道,我们这是个软工时代,库之间的引用逻辑错综复杂,任何一个单位或个人都不可能不用别人的逻辑来进行开发,LINUX 是跟开源紧密相关的,而非 WINDOWS 大商业大教堂式的开发,这是开不开源导致的最根本的区别所在,,linux 的 deb 预编译说明什么问题呢,,,所有的移殖问题都是逻辑兼容问题，，有的移殖直接跟ＣＰＵ＋ＯＳ有关，有的移殖跟库逻辑有关，,移殖时，不光要考虑 CPU+OS 的这个大的平台逻辑,而是要考虑它间接或直接涉及到的库的逻辑的移殖问题,比如 WINDOWS 上 VC 的编译器还带了一个 msvcr80.dll 的特定编译器相关的逻辑,需要被动态链接在用 VC8 产生的 EXE 逻辑的体中,,,,是完全按它的全部逻辑来的，问个问题，是不是 ubnutu 上的一切软件，只要 unbuntu 本身被移殖了，那么它上面的一切 APP 都可移殖，不是这样的,这需要要讨论一下 UBNUTU 本身由什么逻辑组成,比如 UBNUT 的桌面用 GTK+/GNOME 实现,那么 KDE 程序就不能运行在这个桌面上,除非你的 UBUNTU 上有了 QT 库逻辑,才能拥有二个桌面,GNOME 逻辑不包括应用程序,,而 KDE 包括 QT 库,包括 KDE 桌面不说,还有 KOFFICE,KDEVELOPER 这样的应用逻辑,所以 KDE 不单是一个桌面环境逻辑,还实现了这个桌面环境上的很多应用.

Minlearn Ruby(4) 字符串与 WEB

字符串是各大语言要考虑的对象,首先各个平台上字符串逻辑都不一样(有的用四个字节内存,有的二个),,即使同一平台不同的语言实现也不相同(你看 C 在字符串后加一个终止符),,而且,也有关于字符串处理超大逻辑(正规表达式),甚至于一门语言本身的词法处理,,,也涉及到自动机(当然,并非正规式)和正规式(自动机对正规式产生的正规集进行处理)

?

而且,在 WEB 开发上,字符串跟 I/O 又有关,是 WEB 开发的重中之重..

互联网与企业开发

互联网的最初灵感来自对学术自由和开放的向往，而现在它已成为由企业和运营商控制的商业平台。企业应用与网络的发展密不可分,这二者相互发展,成为影响软件工程界的

二大主力

多维这个字眼本身就提倡从多个方面,而企业应用与网络(更多地是网络抽象中的 WEB 抽象领域)在多个维度多个方面结合演变,因此产生很多新的知识.,它代表了最复杂的软工领域,

企业应用是最复杂的,而它又是跟网络技术相关的,因此企业开发中所涉及到的应用是无比复杂的,这种特性使得企业开发成为重中之重一个很重要的表现就是 SOA 的出现,这也就是 JAVAEE 关注和擅长的领域所在.

Minlearn Ruby (5) 网络原理与 P2P

PC 上的网络标准就是以太网卡作为接口的逻辑,,当然 MODEM 的时代已经过去,,(有三种通信媒体,网卡,我们的网卡就被称为以太网卡,因此有以太网标准,无线网卡的无线标准等)
COM 与 IDL,IDL 努力实现一种与语言无关的构件通讯标准,,而 COM 是一种组件,这种基于二进制级的构件技术(要跨语言跨平台,它的最主要的体现在于分布)跟 SOA 这样的源程序级的构件技术还是不同的,,虽然这二者都是构造大件逻辑的技术..比类高一层的逻辑的技术..
P2p 甚至不跟 TCP 有关,，它从 IP 协议开始，有自己的一套协议，再慢慢发展出上层应用，,JXTA 的 CORE 提供了一些模型，,因此 JXTA 是一种平台（存在有很多 P2P 平台），,它提供的消息发现机制只有发展到成员加入这些具体的应用逻辑中，，才能被称得上是应用逻辑了。。在 JXTA 中只能称之为平台部分

为 Windows 说些好话

1,Windows 在面向开发方面,,极力封闭在对其内开发(想象一下 MFC 那种拉及),这是其商业战略,但是 WINDOWS 下的 IDE 和移殖过去的语言也发展得已经非常完善,如果不是 WINDOWS 开了这个先河,恐怕时至今日,类 LINUX 的桌面平台都没有一个能学习的榜样.在 Windows 下开发,,在一些方面,微软极力使人们的开发过程变得简单,,微软也的确做到了,,但是另一些方面,它的确也导致了另一些维度上的更大复杂性,但是总体上,,WINDOWS 下的开发人员是很非常容易找到它所需要的资料的.
2,诚然,并非所有的计算机逻辑都需要套上一个 GUI,但是有时候奇怪得很,GUI 能办得到的事情,CUI 却办不到,,CUI 能玩游戏吗?游戏就不是应用吗?诚然,这二者根本就没有可比性,那些说 WINDOWS 不如 LINUX 的人主要是开发人员,但是生活是生活,,我们并不总用电脑进行开发,而只是应用..
3,WINDOWS 首创的菜单,桌面任务栏这些 GUI 理念和设计深刻地影响了人们,被用在

PC 界面的方方面面,被很多后来模仿者所抄袭,诚然技术总是技术,LINUX 也可实现 XP 的桌面,但任务栏,窗口管理器这些理念总是 WINDOWS 设计出来的吧.(linux 上的 Windows 叫 window,x11 标准也是 window)

4,WINDOWS 在其内核部分就整合了 GUI 逻辑,这使得它的桌面反应很快捷,用户体验甚好,而不像 LINUX 上外挂一个 GNOME,KDE 的方式,这种理念使得 LINUX 不适合多媒体发挥,比如游戏,LINUX 上的 SDL 很简单很粗级,而且是 C 的,没有多少接近游戏设计的高层逻辑存在,相反只是面向 PC 通用多媒体设计,即使是游戏,也采用 X11 机制和 GTK 这种非硬件直接访问的库,这使得 LINUX 下的游戏历来就少,而微软对游戏的支持是不遗余力的.

5,直接学习 linux 和 unix 固然是开发人员学习编程最好的方式,然而学习 WINDOWS 给了你一根通往底层知识的拐棍,如果不是在温室里长大(温都室),很多人恐怕一看到 UNIX 文字界面就早已没有了兴趣吧,不要说 UNIX 和 LINUX 多么多么科学,反正一句话,真正计算机的人是需要 WINDOWS 上的那些应用程序,而不是开发人员天天津津乐道的 VI,MACE

6.Windows 越来越大,吃硬盘吃内存,微软补丁不断,但是要知道,PC 硬件是在不断发展的,手机平台是越做越小为宜,但是 PC 的发展势态符合这种改革,而且,在开发方面,OO 方面的设计思想,框架,SOA 构件,这些未尝又不是越来越占内存和硬件的逻辑,,,如果开发逻辑都可以向复杂度发展,作为 OS 运行这些逻辑的本身为什么不能向复杂性发展呢?

真正的 Windows 消息

消息机制和绘图机制是微软 Windows 及其周边其它产品与生俱来的，是 Win 系列 OS 作为一个操作系统进行微机内部实现的二大支柱和特征，消息系统是 Windows 下一切应用程序间，包括 Windows 自身，进行交互和通讯的渠道(有些程序并不理会消息，因为它不是标准的 Win32 程序)，是 Windows 实现对运行在其下的所有应用程序进行控制及应用程序对 Windows 进行响应的解决手段，因此对 Windows 的编程，无论是在哪种 语言规范和 IDE 下，都不可避免地要涉及到消息处理，虽然有些编程语言如 VB 用事件驱动编程机制在很大程度上封装了消息的复杂性,但若要深入 Win32 编程，就必须学习 Windows 的消息系统，正如游戏编程要掌握 Win 的绘图机制一样,而只要你一旦深韵了这二大支柱和基本,你就掌握了 Win32 编程的根本。。

消息的产生来源于系统事情(包括计时器事件)和用户事情,Windows 用消息来调入和关闭(还有其它处理,如绘制一个窗口等)应用程序,一个典型表现是在关机操作中，Windows 发一个关机的消息给所有正在运行的应用程序，告知它们退出内存，此时，应用程序用回应消息的方法来响应 OS，因此，消息是应用程序与 WinOS 交互的手段..

消息的主体是应用程序之间和应用程序与 OS 之间，(这是通俗的说法，其实在一个应用程序的内部，各“窗口”组件之间也存在着消息的流动，窗口组件与它们的父窗口和上层窗口之间当然也有消息的传递过程(如"命令传递"，后面在跟踪一个消息的路径中将会详谈)，Windows 内部即时流动的消息数量是如此的宠大，程序实现之外的手

工分析是一种很自不量力的事情)消息的最终主体却是窗口与窗口之间,窗口与 OS 之间 - 因为在 MFC 的技术规范里,只有窗口进程才能发送和接收一个消息并处理它，当然一些非界面窗口类如文档类也能处理一个消息，消息的最终归宿是某个窗口类的成员函数,也就是进入消息处理函数被处理,或被某个非界面类也就是内部处理类如文档类处理,系统中默认的窗口类和用户注册的窗口类都有进程，都能在内存中创建实在的窗口对象,窗口对象和窗口类接收和处理(千万注意:接收一个消息和处理一个消息是相差甚大的二个过程,后面将在讨论重定向一个消息技术时将谈到)发往它或由它主动发往别的窗口进程或 OS 的消息,修改窗口进程干涉窗口进程对消息的处理过程(而不是接收过程，这个区别的详细解释请参见后面从"注意消息泵并不是一个.."起的文字)是可能的(窗口进程只是一段函数)，但是如果这个窗口进程属于别人，如系统的窗口类，你将没有源程序进行修改，但却可以用消息重定的技术加以干涉，比如用户自定义的窗口类，用户完全可以自定义它的窗口进程，编写自己的消息泵，实现对消息的重定向，编写用户自己的消息泵属于 Win32 编程中重定向一个消息的七大技术之一。

MFC 中有七种技术可以用来重定向一个消息，它们是：1，子分类 2，超分类 3,OnCmdMsg(),4,SetCapture5,编写自己的消息泵，6，SetWindowsHookEx(),人们常说的钩子函数,便是其中之一.

在谈完消息泵的概念后,我们将一步一步追踪一个消息在系统中的路径,然后才能讨论对它的重定向。

消息泵并不是一个窗口类的窗口进程,虽然它们都是函数,同样都对注入到这个窗口进程的消息进行工作,而并不最终处理消息本身(上面已经说到原因),消息泵是一个通俗的说法，它只与消息被发往窗口进程后的接收工作有关而不与处理过程有关(上面也已经说到消息的接收和处理是二不同过程),而窗口进程恰恰相反它只与处理有关不与接收有关下面开始详述。。

消息泵被包含在 CWinApp 的成员函数 Run()中..... 写得太累了,,,未完待续.

实际上在 UML 的规范里，消息是一切对象发生联系形成系统的手段。

入队消息与非入队消息
走出 Windows

真正的 MFC

32 位的图形化操作系统 Windows 的出现,曾使一直在 Dos 下编制 16 位控制台应用程序的程序员们产生不小的恐慌,Windows 的易用性和在 Windows 下进行软件开发工作的复杂性是很大的一对矛盾,在微软没有推出它的解决方案之前(前面说到,这是一种必要),程序员要实现从 Dos 平台到 Windows 系列平台的转型门槛很高,在降价这个技术门槛的工作中,微软首先要解决的是实现应用程序的图形化界面编制,C++ 加 API 的解决方案仍然很复杂,虽然有效但却留给程序员们太多的工作,除此之外,程序员还要考虑

Windows 的消息机制,消息机制是 Windows 的内部实现,对 Windows 的编程不可避免地要涉及到消息,于是,微软的工作还包括在其产品中考虑增加对消息机制的编程支持(后来我们知道微软的产品是 VC++).

在 VC++出现之后,一切在 Windows 下的软件编程工作似乎都变得那么自然了,VC++加 MFC 的解决手段使 Windows 下的程序员多了起来,同时也让 Windows 的用户多了起来,一切尽在微软的掌握之中,VC++如愿成为 Windows 程序员专业的 SDK,一方面,VC++使用的语言规范是 C++,这是最有前途和最流行的语言,另一方面,MFC 提供了对 C++下图形化界面编程和消息处理机制的编程实现,这曾经是 Windows 下软件开发的 2 大技术难点,MFC 虽然不是完美的,因为它在对 API 函数封装的过程中出现了一定的冗余,但正因为 MFC 是微软自己对 Windows 下软件编程的解决方案,无疑它是最好的,在出现了如 BCB 这样的 IDE 后(VC++是基于对象,BCB 是面向对象),软件界面的设计可以纯手工搭建,但是这种方式是以一定的功能要素为代价的,要深入 Win32 编程,深谙 Windows 的消息编程技术,最好使用 MFC.

MFC 实现对界面编程的工作是通过封装 API 中用来进行界面开发的函数来进行的,可以想象,创建一个窗口对象所必需的参数无论是在 MFC 环境中,还是在 API 环境中,本质上都是相同的,对消息机制编程的支持,微软是这样做到的:消息被发送到某个类的窗口进程中,让消息最终进入 Windows 的一个类的成员函数中(注意:不要被诸如"Windows 的类"这样的概念感到不解,不要以为类的概念只出现在 C++的技术规范里),让消息最终被这个类的这个成员函数处理掉,此时成员函数名通常就是"On_"+消息标识(如 WM_Paint)的样式,这显然是符合 C++的封装技术的.

MFC 应用程序用窗口进程来接收和处理一个发往它的消息,(前面说到,接收和处理是二不同过程,又说到,消息泵和窗口进程是二个不同的概念),其它的非 MFC 应用程序,如用 VB,Dephi 开发的普通应用程序,包括非图形界面的 32 位应用程序,以及其它一些运行在 Win OS 下的应用程序,也都存在着与其它应用程序或操作系统的消息交互,消息的有无与开发环境无关,但是消息(在程序代码中是一个大写的标识)的流转方式在 MFC 环境下和 API 环境下是明显不一样的,根本原因在于 MFC 应用程序对消息进行了更复杂的回旋处理,刻意让消息的流转走了好多弯路,即不断地调用函数让消息进行流转,而这,是微软基于 C++的封装技术要求考虑的,是十分合理的,这就是 MFC 技术之一的"消息映射","消息映射"模式无疑走了一些弯路,带来了一点复杂性,但却又是一种需要,因为 MFC 是 for C++的,"消息映射"就是要让 MFC 满足 C++的类的封装技术(文完).

MFC,微软基础类(Microsoft Foundation Classes),实际上是微软提供的,用于在 C++环境下编写应用程序的一个框架和引擎,VC++是 WinOS 下开发人员使用的专业 C++ SDK(SDK,Standard SoftWare Develop Kit,专业软件开发平台),MFC 就是挂在它之上的一个输助软件开发包,MFC 作为与 VC++血肉相连的部分(注意 C++和 VC++的区别:C++是一种程序设计语言,是一种大家都承认的软件编制的通用规范,而 VC++只是一个编译器,或者说是一种编译器+源程序编辑器的 IDE,WS,PlatForm,这跟 Pascal 和 Dephi 的关系一个道理,Pascal 是 Dephi 的语言基础,Dephi 使用 Pascal 规范来进行 Win 下应用程序的开发和编译,却不同于 Basic 语言和 VB 的关系,Basic 语言在 VB 开发出来被应用的年代已经成了 Basic 语言的新规范,VB 新加的 Basic 语言要素,如面对对象程序设计的要素,

是一种性质上的飞跃,使 VB 既是一个 IDE,又成长成一个新的程序设计语言),MFC 同 BC++集成的 VCL 一样是一个非外挂式的软件包,类库,只不过 MFC 类是微软为 VC++专配的..

MFC 是 Win API 与 C++的结合,所以 MFC 是 C++封装实现下的 Objects,API,即微软提供的 WinOS 下应用程序的编程语言接口,是一种软件编程的规范,但不是一种程序开发语言本身,可以允许用户使用各种各样的第三方(如我是一方,微软是一方,Borland 就是第三方)的编程语言来进行对 Win OS 下应用程序的开发,使这些被开发出来的应用程序能在 WinOS 下运行,比如 VB,VC++,Java,Dehpi 编程语言函数本质上全部源于 API,因此用它们开发出来的应用程序都能工作在 WinOS 的消息机制和绘图里,遵守 WinOS 作为一个操作系统的内部实现,这其实也是一种必要,微软如果不提供 API,这个世上对 Win 编程的工作就不会存在,微软的产品就会迅速从时尚变成垃圾,上面说到 MFC 是微软对 API 函数的专用 C++封装,这种结合一方面让用户使用微软的专业 C++ SDK 来进行 Win 下应用程序的开发变得容易,因为 MFC 是对 API 的封装,微软做了大量的工作,隐藏了好多内节程序开发人员在 Win 下用 C++ & MFC 编制软件时的大量内节,如应用程序实现消息的处理,设备环境绘图,这种结合是以方便为目的的,必定要付出一定代价(这是微软的一向作风),因此就造成了 MFC 对类封装中的一定程度的的冗余和迂回,但这是可以接受的..

最后要明白 MFC 不只是一个功能单纯的界面开发系统,它提供的类绝大部分用来进行界面开发,关联一个窗口的动作,但它提供的类中有好多类不与一个窗口关联,即类的作用不是一个界面类,不实现对一个窗口对象的控制(如创建,销毁),而是一些在 WinOS(用 MFC 编写的程序绝大部分都在 WinOS 中运行)中实现内部处理的类,如数据库的管理类等,学习中最应花费时间的是消息和设备环境,对 C++和 MFC 的学习中最难的部分是指针,C++面向对像程序设计的其它部分,如数据类型,流程控制都不难,建议学习数据结构 C++版..

(2)

所有在程序中用按钮窗口类创建的窗口对象都共享一个共同的按钮窗口进程,注意:类有进程,用这个类创建的对象同样也有进程,这个进程便是它所属类(类的属别关系有二种,基类和派生类即父类和子类，即逻辑结构上的类的属别关系。上层类和下层类也就是物理结构上的类的属别,如在一个 MDI 中，文档类是视图类的下层类视图类和文档类又是框架类的下层类,再比如一个普通的对话框类对话框类本身是 CWnd 类的一个子类，对话框上的某个按钮是通用窗口的子类，但却是这个对话框类的下次类不能称它是该对话框类的子类)的进程，如果一个 WM-PAINT 消息被发送到某个窗口对象，比如一个对话框上的某个按钮，术语称:消息被发送到这个按钮的窗口进程里,而实际上:消息被发送到了通用按钮窗口类（按钮窗口作为 Windows OS 下的窗口三大类别之一的子窗口，与弹出式窗口如对话框窗口重叠式窗口如一个 SDI 或 MDI 应用程序的主窗口并行.),其内部实现是:消息被发送到了通用按钮窗口类之后，通用按钮类窗口进程根据这个预窗口对象指定的大小和位置风格(在程序中这些参数被传递给通用按钮窗口类进程)画出确切的按钮对象即该按钮在屏幕上的物理表示，,

可见界面类往往都与一个窗口对象实际关联着，窗口对象在程序中定义并被加载时它在

计算机内存中有确切的物理表示.并通常用一个指针变量来指向它的地址.

窗口进程到底是什么东西呢，(回调函数是一种异步消息)它们实际上是一段函数,(这段函数在这个类被定义时就被定义了如 CWnd 类的窗口进程,用户可以定义自己的窗口进程然后进行任意的修改，但操作系统的类,如 CWnd 的进程不能被修改，要干涉它的消息处理过程就要学到消息重定向了,消息重定向是 MFC 中最诱人的技术)MFC 编程机制中用它来实现对消息的处理,实现 Win32 的消息机制,这里千万注意:窗口进程不是普通意义上的“类的消息处理函数”,它们是进入到某“类”(这一类的窗口对象都共享这个“类”的窗口进程,)窗口消息的总的管理者,这有别于作为一个类的成员函数的“类的消息处理函数”，因为这些函数才是消息的最终归宿和被最终处理的源 头.(未完待续.)

真正的虚拟机

你会很是奇怪？为什么 Java 不作为游戏的脚本语言被广泛使用呢，这跟 Java 的性质有关，Java 自它被设计出来就作为非本机语言来提出的,它的数据类型全都不是 win32 使用的，Java 虚拟机独立平台，Java 虚拟机的出现完全是为了配合 Java 语言而出现的，Java 语言中的 jni(Java native interface)就是为了让 Java 本地化而出现的东东

在学习 JAVA，。NET 类库时,,实际上就是学习 sun 和 microsoft 如何对编程领域与现实世界结合点的那些事物(或称对象)的架构进行划分与整合的过程(注意这里提到了划分与整合),，在原来的程序设计语言中，往往这些事物(窗体,，IO,，，WEB,，DB)是分开来学习和使用在 OO 代码中的(想像一下在 VC6 的 MFC 中，需要使用一个库对象时跟在 C#中的类包中的区别)，然而，在.NET 类库和 JFC 中对他们进行了很多重整合与重划分

也即，有时，不只是创建新事物要求定义接口和定义抽象层次,，有时对现有构架的集成与分解的需要使我们需要接口和重定义抽象层次(这个从 JAVAEE 的一系列架构中可以大量被看到,客户端层，表现层，业务层，数据层，整合层-也即 Middle ware 层，资源层)

但是千万不要做超级设计，不要走入完美化的怪圈，更科学的整合与分离模型总是存在，但不要走入哲学化的深处(以满足当前认知能力和应用需要之间的平衡为准)。

实际上 API 的概念就在这里扩展了,,，类库，方法，类，某个库内定义的某个结构体，都可以被称作是 API(API 就是面向应用级的接口，实际上大凡接口不是面向应用的还真少，所以 API 一般意义上还可作为一般意义上的接口),,，用户使用这些实现，通过一些迂回的办法，也即定义一系列使用它们的接口，最终可以形成很复杂的程序,，也即，我们编程大部分情况下只是使用别人的东西，我们只是在写接口粘合代码，而这，就是真正意义上的编程（一个程序往往只是一些使用某些特定代码产品的使用逻辑本身，并不实现某个算法或提供给别人可使用的某个 API),，当然，也不能说人家提供 API 的人写

这些 API 的过程就不是实现(只不过它们是写的目的是供别人使用而写，而我们是为了使用而写,应用永远是重要的嘛，因为毕竟我们不需要重新发明一些别人已经写得很好的轮子,，也即我们相对是开发者而人家相对是产品供应者)

说了这些你可能还是不明白我真正要表达什么？我是说：可以由几个 API 写成一个巨大的程序，而程序中往往真正实现某个算法的 API（这些 API 有理由说它们是真正的实现，通俗来说就是面向过程里的某个算法过程），而我们天天看到的程序,，真正的技术算法部分 API 可能也就那么几个，说了这么多，我只是说：我们天天看到的程序，其实组成他们的大部分只是非算法的架构逻辑们!!（这就是被称为接口逻辑的逻辑,，因为它们主要提供为以后的可复用性提供的架构逻辑，是真正的发挥桥接口作用的中间逻辑，那些能发挥实际作用的算法逻辑中间逻辑往往由这些架构串联起来,，离最终的应用逻辑还远呢，）

OO 编程时，在分类事物时，我们时常碰到中间抽象的分类边界,,，这就是粒度，元(元是老子提出来的词),用在 JAVA,。NET 库中就是名字空间，接口,MeteData 等

想像一下 Sun 和 Microsoft 在划分和整合这些对象时的情形，选择一个好的粒度比率显然是很重要的(为这些粒度命一个好名往往更是一个很艰巨的过程，常常采用接口+able 后缀形容词的形式,如 IConvertible 表示可 cast 对象或类的集合,现在的大型类库都跟接口息息相关，因此它的低层 IConvertible 是一些描述性的高抽象形容词表示的一个接口对象，)，聚合跟分离的边界划分，(如果过于强调整合就会造成装箱开销过大,,如果过于强调分离就会导致学习和使用上的不便) ,对于名字空间（Java 居然用了文件夹和公司域名表达与名字空间的对应，实在是绝！这体现了 WEB 这个粒度越来越靠近其它的类库粒度,所有编程中出现的事物都趋于跟 WEB 整合，人们正在寻求所有本地事物与 WEB 的结合点，其它一些例子比如 DCOM，XML,RMI 也是这样）

语言宿主

程序等于指令加数据(指令是被硬化到 CPU 中的)，实际上就是指 CPU 加内存，内存器，因为程序运行的过程永远不在乎这三个东西，在诸如 C 语言这样的结构化函数语言中，深克地提现了数据结构跟算法这样的东西。。因为这可是冯氏架构规定的啊，

宿主代码，脚本宿主机指的是一种什么意思呢,，因为各种语言有不同的运行时，同一种语言不同的编译器结果也有不同的运行时(编译原理有专门一章讲解运行时)，因此其'指令加内存'的运行时内存分布情况是不一样的。。当可欠入式脚本语言要

真正的脚本

ogre 数据驱动式编程与可拔插组件与脚本驱动

为什么说架构是重要的?架构使抽象置于高层，使实现置于低层末端或者完全独立高层逻辑成为 plugable 的东西。(想象 WIN 的 HAL.DLL)
往往有时候，,真正的再编程能力在于对架构的掌握与分析与学习,,,而语言层面的细节比较容易学习，对物件事物的对象拆解也是仁者见仁智者见智但在有文档的前提下也不会差到哪里去的问题。

在进行一个大架构开发时，应该为使用它的用户隐藏一切底层逻辑，而向用户提供高层应用逻辑(注意这里的说法与前面不一样)，而即使是这些高层逻辑，也不应让用户完全理解它们的实现逻辑，系统应该隐藏那些实现接口，而向用户提供那些直接能驱使系统逻辑工作的接口，，比如参数填充，或数据驱动，这个应用一个具体的例子就是 OGRE 图形引擎，它的 rendersys 的 render()接口是一个 callback,,,仅仅在用户指定加载到场景中的资源并由 rendersys 确定渲染后它才开始自动渲染(由 ogre 调用)，无须用户调用，

而且，如果要为一个大构架增加调用第三方组件时，最好能保证这些组件是可拔插的(而系统原先为它们预留过接口)，，即这些组件与主构架是相互独立的 Dependencies,portable,plugable 的

一个工程往往同时用到编译和解释，这就是脚本语言的由来，脚本语言通常是解释性的，因此可以动态被测试（可在运行时修改），而且它的语法比编译语言简单，因此可以 wrapper 调用 C++的库，然后在应用中动态测试

JAVA 也是一种脚本语言(一般把能 JIT 或解释的语言都看作 Script,因为它们能实时成型)，JAVA 语言实际上并不是对计算机编程(而是对 JVM 这台虚拟机器编程),,，所以说它不是本地语言，JAVA 语言的出现就是为了配合它的 JVM

脚本被用来进行一些诸如数据驱动支持之类的工作(想象 War3 中的开关编辑器)，脚本有这个功能是因为脚本是动态解释的
脚本机与游戏高层通用逻辑接口设计,其实脚本这个概念自古而始之，在 unix shell 下就经常编写 scripts 编译程序，等等，这个机制实际上可应用到包括在游戏在内的一切大中型应用中，这些所谓的逻辑接口可以是一切应用级的高级接口(比如在场景中用脚本控制形成一段 CS)，也可以是低层接口(比如控制一块纹理的赋给。。,Ogre 在引擎的一开始就集成了脚本支持，等等)

微软的自动对象也可以用脚本来控制，比如 Microsoft Word OLE automation objects.,,利用 vbscript(VB 的一个子集)来控制,而 Windows 提供了对于它们的脚本虚拟机支持.

另外一种比较通俗的游戏脚本机就是读取一些 txt 文件，游戏程序内内置一些函数，然后遇到 txt 内的某个字串就中断然后执行某个函数.

action-schedule-motion

真正的.NET

winword 曾经一路引领应用程序界面的潮流，， 这不，微软也对它的 IDE 要进行一番手术，在最新的 vs9 中，微软用了大量的流行色素，浮动工具栏之类的元素，，当然这只是界面层次的，，实际上 vs9 被蓄谋设计成一个整合关于微软 win 平台下所有编程工作的工作台(和整合所有编程工具的工作台,,.net 更是蓄谋要形成通用的虚拟平台下的编程语言规范),,，VS9 整合了.net 托管代码编程集，，win32 本地 native 代码编程集，还有 web 动态网页编程集,xml 数据库,web,建模,所有帮助文档，甚至还有 device sdk,和 tablepc sdk,然而整合与分离永远是二个二个方向发展的东西，,嵌入 embeded 系统的出现就表示，，往往有时候过于集中的整合并不适用大多数人(所幸 vs9 并不是完全整合得死死的,它的各个部件如 vc ,vcse 都可分离来用).

下面阐述一些容易混淆的概念

sdk 往往指代例子加文档的集合(当然更准确的意义是它应该还包括文件头),

1.platform sdk,,Windows OS 平台软件包,,封装了 win32 所有 api 的头文件集，附大容量的说明文档，即 msdn，缺省安装一般出现在 Program Files\Microsoft SDKs\Windows\v6.0A 中

2..net sdk 托管代码 sdk,往往也带有一个大的说明文档，这就是.net framework 虚拟机的 sdk(虚拟机是虚拟机，,，一个虚拟机的编程语言就是一个虚拟机的编程语言，这个虚拟机上的编程语言往往是为虚拟机写它的本地代码而出现的，因此.net framework 并不是虚拟机本身的 sdk,我们永远别想得到微软的.net 的源程序或编程接口 - 盖只可没有跟 Java 一样发布它的虚拟机源程序的破力,什么是 native 呢？就是说用对一个虚拟机编程，用的是这个虚拟机的支持语言，而不是什么其它的高级语言，比如 high language，),运行在.net framework 下的应用程序和代码和运行在原生 win32 下的本地代码是有区别的，，前者是.net framwork 托管的(是在一个脚本机下运行的，应用程序接受来自它分配的内存,在它的框架内运行)，而后者是直接在 OS 下执行的(在 c 的 rt 下运行),,这二种运行方式分配内存的方式明显不同，其它的不同点就更多了

人们常说从 win32 到.net 是编程平台的转变，，意思就是从本地到.net 虚拟平台的转变

3.device sdk,,,驱动开发软件包

个人感觉.net 是微软拿来与 Java 抗争的东西，，虽然会在一定程度上降低开发软件的难度，但是使程序运行在.net framework 下是要付出相当代价的(框架和架构本身是需要代价的，一是因为增加了迂回所以理解上会有代价，二是运行起来计算机要对代价作解析，程序速度也会变慢)，，程序运行速度明显变慢（虚拟机普遍都很慢，因为它是软件的机器），而在 win32 下开发程序唯一的缺点就是除非 Windows 升级到下一个更高级的核心，否则你的程序是不用更改的，，而运行在.net 的程序只要保证 Windows 一直支持.net 就可永远在 Windows 下运行，，这是它唯一的好处之一

VS7 以上的开发环境一般需要安装以下几种 SDK，Windows SDK(Windows 相关比如 Windows.h),PlatFormSDK(平台逻辑相关),DOTNETSDK(可托管代码相关),VC SDK(VC 开发环境有关),还要安装一个 SQL 精简(数据的可持续化已集成在编程环境中了)

COM 与 DCOM

DCOM 调用 RPC(远程过程调用,Windows 基础服务之一，services.msc 可看到,这种机制不仅提供对本机远程过程的调用机制，还提供对网络上不同位置的二个对象之间的相互调用过程)，它的平台实现基于 ole2(object link and embed),,,有专门的 idl 语言(接口描述语言)和编译器(vc 下是 midl)来编写 com,com 有它自己与线程和 TLS（thread local storage）的操作与属性，这就是套间，因为 COM 其实是一种较原来的 PE 文件和 DLL 文件更加高级的文件装载机制定义和对象数据持久化机制定义，下面会有具体描述，接口描述语言是一种类 swig 的东东(即 code generate 类工具而不是 code complier 工具，接口转化工具，语言粘合兼容器)，生成 COM DLL 过程要用到类源程序和接口定义或接口文件,，但是它较之 swig 类工具更加高级和复杂，因为接口定义成为了源程序作为一个类的真正接口部分(而不仅仅是为了生成其它语言用的关于此源程序的可有可无的包装器,,包装器 wrapper 是 swig 用来将类源程序重新封装使它适配其它语言的对象模型工具)

那么 COM 的本质到底是一种什么东西呢？它是一种对 PE 和 DLL 的完善!!如果说存在 C 与 C++的不同的话，那么用 C 和 C++编写的 EXE 和 DLL 也有不同，如 USER32 不是用 C++写的，它的 API 是 C 的，而假设 GDI32 是用 C++写的（那么它的 API 就是 C++的，是某个对象的某个成员函数，诚然，用 C++也可写基于过程的 EXE 和 DLL，那么

较之前二者 COM 就是更加高级的 C++的 API，因为它还支持直接对二进制对象数据持久化的元素即接口,这就是面向构件的开发）前者在二进制级的文件内不内含对象及其持久化机制，而后者提供这种支持机制，而 COM 就是这种机制的集大成与更加完善,因为它还支持接口,PE 是一种可移殖的可执行文件，DLL 文件可以用于提供 API，而这也是原 WIN32 底层 API 实现方式，一个 EXE 或 DLL 的装载器可以将文件内的.text,.data,.stack 定义正确装入内存形成进程(OS 段式内存管理)，线程，而 COM 的进程自有它的特点，这就是套间

打开 VC SDK 中的 objbase.h，可以看到所有接口对象的基对象是 iUnknown,接口是一个对象,而一个接口作为 smart ptr 对象（在一切皆对象的说法里，指针对象也是对象)，，，它封装了所指对象向用户提供的使用层面的接口（隐藏了实现)，所以在接口下也有一些对象，，，接口与对象并不一一对应，，，往往一个对象的类会有多个接口,,,从一个接口类定义出一个接口对象，，实际上就定义了这个接口作为指针所指的那个背后的真正对象，故 I 其实是#define ptr interface 之类的定义

com 基于 ole2,最初被微软用来解决复合文档而出现，是 win 上的一种二进制组件标准，既然它是为了复用而出现，因此它必须提供一些复用手段，面向对象有三种机制，那么面向构件也会有一些，比如它使用 qureinterface()实现多态
这实际上就是面向对象里面的 cast,,,属于 RTTI 的内容
为什么说这些仅仅是 RPC 呢？因为接口只是行为集，函数阵列(marshalling 技术可以序列化一个远程对象)，因此被称为远程过程
DCOM 是微软用来对抗 OMG 的 CORBA 和 SUN 的 RMI，而CORBA 基于 ORB(对象请求代理)
实例是对象的指针，句柄就是指针的指针,,,形如 void**,someobj**,这经常出现在 com 里
CORBA 很大程序上依赖 orb,因为它是集中管理的，而 COM组件是各自为致的，
从 COM 到 DCOM，，主要用 C++来实现，因此它必定充满了指针(com 的 idl 语言就是 C 的风格),,微软的很多操作系统底层就是用 COM 和 DCOM 来实现的，
DLL 只是一种文件装载机制而已，COM 也可以以 DLL 的方式存在
COM 和 DCOM 跟接口语言息息相关，一个是服务端的桩，一个是客户端的代理
omg 有它自己的 idl,不跟具体语言相关，因此 Java,c++什么的都可以实现
因为 dcom,com 用于 window 的平台之内，因为它是一种更高级的文件装载机制，，像 inproc 注册 dll 形式的 COM，而进程映象 exe 就需要另外一个机制来注册
实际上 COM 组件就是一个 COM 对象，由不同的接口对象组成，
从下面开始就是 J2EE 大领域学，对它的学习可以让我们更加了解软件架构

以上的图中不过都是些抽象罢了,对抽象的实现就是编程(用语言或其它范型的实现)

无穷的细节之下不过几条简单的思想,看 Sun 的策略就知道了。
有了三种思维(泛化，抽象，组合)，我们就可以很好地理解 J2EE 的架构了

理想

三权分立是一种政治理想,,提出它的人觉得这样的社会很合理并立著成说,,,于是就有了今天我们看到的资本主义社会的政治结构,在这种构架下发展出来的人和国家的关系很融洽,社会经济充足发展,很多人认为这是一种终极的社会理想,而其实它只是万元中的一元,,这个世界永远都是多元的,,,资本主义价值观跟世界观并不能代表一切,,,,那更多地是一种科学家的偏执认识

软件也是一样,,,Windows 窗内的世界足够美好,,,linux 在开始就没有能打过 Windows,,那么在今天 Windows 统一桌面时,它就成了一种合理存在.

linux 跟 Windows 为什么有不一样呢,linux 跟 Windows 最大的不同点在于什么呢,,不是核心也不是外面的桌面,而是开源(这是个几乎可以决定这二个系统本质的区别,,正是这二个大方向决定了一切和它们的发展态势),,,我们知道在 linux 下一个很重要的组件就是 rpm,dkpg 这样的软件包,其实这样的逻辑跟本就不需要用户来维护,,Windows 不考虑这个,因此在它上面,只要下载并运行 exe 就可以执行文件了(这就是运行载体的巨大区别,这是个对用户至为重要的门槛),而 linux 下居然让用户维护这层逻辑(linux 的用户下载开源软件,这就是 linux 要考虑的"开源",LINUX 其实可以让这个工作的最终逻辑结果发展到跟 WINDOWS 一样,而彻底放弃什么新立德软件包都要做到跟 Windows 一样,最好的结果是,用户根本体会不到一个叫新立德的在管理维护和下载一系列应用程序,,用户不必知道这个,),而用户不是程序员,学会不了如何使用 dkpg(软件的发布方式为什么要让最终的用户去涉入呢,这是个很不明致的做法,你可以比较 web 的 XML 的 WEB 程序发布方式,它是对机器的,对人而言都显得很复杂,除了程序员人们都普遍觉得 XML 过难),,,,这就是说,在用 C++生成软件并在一个 OS 上得以最终运行的这个逻辑路径上,,,,,WINDOWS 和 LINUX 做的工作是不一样的,,,LINUX 为了它的开源而将工作保守在提供一个软件包管理器的逻辑程度上(其实在程序员的眼里,LINUX 才最最为简洁和有效的,而普通用户则觉得 LINUX 是程序怪人为程序怪人开发的 OS,哈哈).而普通用户才是最主要的用户群,是最强大的基础,而不是精英

当然,软件包只是一个方面,,在其它方面,,,LINUX 也有对普通最终用户造成门槛过高的地方,因为它面向开源,,,程序员几乎可以直接对靠近内核的 OS 系统逻辑进行操作和影响,而 WINDOWS 只是提供最最高级的用户逻辑和 API 供人们使用,,,这种发展上的巨大差别使 LINUX 成为高级用户才使用的系统,,因为它没有 WINDOWS 封装了一切最后面逻辑的逻辑,,反而是简单和可扩展的,,也因为它极力要提供这个可扩展性,,,因为最最后面的用户逻辑做到一定程序就中止了,,,这给普通水平低的用户造成了门槛过高..

思想，维度，细节

陈陈 13:19:36 掌握思想和细节的辩证关系了
MinLearn 13:19:47 比尔盖之说了，，商业，不就是把有的东西卖别给人，然后收别人的钱，，难道还有其它的东西吗
MinLearn 13:20:14 当然，如果逻辑扩展开来，在每个维度上又可产生新知，新的逻辑
MinLearn 13:20:35 怎么卖，，怎么有自己的东西，，怎么光明正大收别人钱，，怎么多收？？怎么收了别人钱还不被骂？？
MinLearn 13:20:53 很多时间，，学习就是处理一个中心跟许多支节的关系
MinLearn 13:21:07 所谓中心就是思想，所以支节，就是你说的细节
MinLearn 13:21:34 比如 XML，，中心就是一种标记语言的规范，可是它的实现有 DOM，也有其它的
MinLearn 13:22:19 比如 XML 的用途，，可以跟数据库结合产生 XML 数据库，可以跟查询结合产生新的 XPATH 逻辑，，XLINK 逻辑，，甚至跟浏览器结合产生 XUL 逻辑
MinLearn 13:22:53 一切都是有一个本源的，只是它被在概念上泛化了，然后产生了实现他们的新知，，需要被学习
MinLearn 13:23:11 所以我说，人类的东西都是把简单的东西弄复杂了，，不过这也是必要的 MinLearn 13:23:24 计算机再强大，如果只是一台裸机，你说有什么用？？
陈陈 13:23:44 你用这个来解释它
陈陈 13:23:48 其实对我的用户体验不好
陈陈 13:24:07 里面的 DOM，XPATH，XUL 这些名词，对于我都是陌生的
MinLearn 13:24:09 只有在它上面发展逻辑，一层一层来，，首先 OS，然后 PE 载体逻辑，然后应用程序，然后网站前端，然后最终应用，然后用户本身
陈陈 13:24:45 所以如果要做商业，就要从别人的角度想问题
陈陈 13:24:52 商业的本质，在于你说的为他人提供价值
MinLearn 13:24:58 我知道，那是另外一个维度的事情，，
MinLearn 13:25:03 那是经济学说的
陈陈 13:25:10 换位思考，智慧交换，这是商业的核心
MinLearn 13:25:27 而我是站在这个历史时刻为了跟你说明中心跟支节的关系提出的说法 MinLearn 13:25:31 不矛盾
MinLearn 13:25:56 对的，换位思考即换维思考，，，，不同的角度可以得出不一样的看法
MinLearn 13:26:14 每一种看法都是偏见，但都是需要的偏见
陈陈 13:26:50 疯了
陈陈 13:26:57 是不是太极端了
陈陈 13:27:15 你试图用所有的、一个的逻辑，去整合所有的想法，即使只是术语的偷换。呵呵
MinLearn 13:27:27 你不可能从经济学，从哲学，从人的观点，再从历史学观点去仅仅说明“商业”，事实上这二个字大得很，，因为当我们是一个历史学者时，就只说明历史

角度的事，当我们是哲学者是就从哲学角度去学，，
MinLearn 13:27:48 不是，恰恰相反，，我用了一种很浅显的道理去说明，，这样我就 min learn
MinLearn 13:28:06 选择我需要学习的去学习，而不是理解人类为止关于这个事物的所有认识
MinLearn 13:28:37 如果我是程序员，我并不把苹果的水份由什么组成这些化学知识拿来考虑一次，我只会把平果封装为一个对象而已
MinLearn 13:28:40 这也就够了
陈陈 13:29:29 ？
MinLearn 13:29:35 如果你能知道这些东西就是：“逻辑”，而逻辑就是软件功能的说法，，，设计软件，就是对逻辑要有一个非常清的规划，否则就过不去，
陈陈 13:29:46 也是，你的这种认知是特有的、惯性的。适合你的，那么它就是最好的

在形式主义与直觉主义之间：数学与后现代思想的根源

首先，从历史上看，某些哲学基本问题和数学一直有着复杂的共鸣。我们可以从一个古老的问题开始，这一问题与数学的本质有关。理性主义者认为，数学仅仅是理智的发明，那么问题是，为什么数学会具有实践上的效能？问题也可以这样理解，即在纯粹的推理和逻辑之外，是否还有其它的内容。如果数学只是纯粹的逻辑关系，并没有其他的内容，那么它只是无足轻重的同义反复。笛卡尔采用了辩证的解决办法。他认为，在认识过程中，除了理性，还需要经验的内容，经验与理性不能完全分离，我们在观察的基础上提出一个理论，反过来，观察又在某种循环中证明了这一理论。这就是著名的笛卡儿循环。对他来说，真理不纯粹是理性，也不纯粹是经验，而是在理性与经验之间的循环。到 18 世纪后期，哲学家们已经明确认识到，对于数学来说，存在着某些超逻辑的、纯粹理性之外的内容。

康德对这一问题进行了极其重要的探索。他明确把主体意识摆在认识过程中的核心地位。一方面是外在的世界本身，一方面是主观的王国，知识从何而来？康德认为存在某些超逻辑的东西成为知识的先决条件，就像维柯的“具有想象力的普遍本质”（imaginative??universal）那样，把经验和概念综合为知识。他指出，人们先天具有的空间直观和时间直观发挥着模型的功能，塑造着我们的经验，人们据此提出描述这些经验的概念框架，如数学就是这样的概念框架。

但困难接着出现了。按照康德的说法，我们的大脑先天具有某种能力，但是我们如何确定事情就是如此？这个问题意味着我自己成为我的认识的对象，因此我不得不面对自己。因此，我们需要比内省更多的东西来支持对这些内容的确认。康德的解决办法求助于共同体，把自我的认识交于他人来裁决。但显然康德本人对这一办法并不十分满意，因为他认为如果个体毫无批判性地接受共同体的信念就是“不成熟”或“未开化”的。

形式维度

提供形式化是因为人只能掌握有限的维度(,最大的形式就是计算机和形式语言,我们不可能说话英语时直接说字母,我们只能用字母上面的单词这层抽象来交流,字母是一种形式,单词也是一种形式,应用时,而不是开发时,我们应尽量选择一种离我们要达到的最终逻辑近的形式来表达,然而这个说法永远是相对的,所有的技术问题到最终都是哲学问题,我并不会想到把知识全部学完,而只是想要完成什么功能就去学习什么细节,因为哲学指出,维度只是人类的认知,当我需要这种认识为我服务时,我就去学习它,否则只需要知道它是人类的维度就行了),而实际上任何功能都是多维的,任何一种简单的形式,如果在上面搭配足够好的抽象,也可以达到很多抽象,比如用 OO 实现面向过程,用面向过程实现 OO,用 Lua meta table 实现 OO,等,但是正如不用在 JVM 上实现一个 Windows 一样,有些抽象是不现实的和没有功用的

探索和解释抽象的达成是学习的最好方法,知道今天我们用高级语言写程序是如何由编译原理和形式语言的抽象一层一层而来的吗?为什么要写循环流程结构,而不是其它形式的内容?了解这些才能到一个新的高度,当然,抽象的隔离性可以让我们不再考虑这些透明的东西,(透明就是你可以装做看不见,意识不到底层起作用的东西,而只用与高层抽象直接互动),,,但是只有了解这些才能达到一个高度,想问题的时候,就会在这个维度上进行,从而不必一步一趋,眼光封闭

?程序员常常津津乐道于某个底层的知识,计算机内部如何运作,如何科学和精巧,并回味很久,一种逻辑达成如何千转百回,错!应用才是重要的,提供一个最上层简单的形式尽量避免不必要逻辑才是重要的

但是即使在现在为止,我所说的维度,抽象,,离直接的编写代码这个抽象跨度还是太大了.

维度

事物和术语是简单的,因为它总要属于某个最终抽象,总可以泛化成某个维度上的"最高境界",这就是本质,抓住了这点，学习这个事物就会开始变得简单，另外一方面，事物是复杂的．因为在这个本质下发展起来和正在发展着的抽象是巨大的，只有了解了这些我们才能真正这个事物或术语，因此我们可以在本质下求细节，这并不是形而上学，我并不是在提倡先学本质再学实践，我们都是站在人类认知的领域和全部维度之内学习（这个术语或事物）的，并未站在别的维度，虽然当我站在这个维度上说话的同时，我也将不可避免地失去某些维度．而你也可以提出很多站在你的维度，来根据你的切身体验，提出一系列反驳的说法来反驳我

维度和形式是重要的，设计模式提出了一系列形式用词，界定和规范了一系列逻辑＂形

式"，，思想是无限的，设计模式其实还可以无限细化和朝某个维度和角色无限深入，但是，即便这个思想"形式"也是重要的，有了这个形式，我们可以有共同的语言来讨论和解决一些问题，而这些问题，以前我们只能绝对而言，涉及到很大的维度，而有了设计模式之后，我们可以"相对而言"，在有限的维度内，为达成更有效的共识而开展讨论.

任何问题都不简单，这也正是因为事物其实是属于某个抽象领域的，当这个邻域与某它事物所在的领域发生联系时，这个事物也会发生改变，对这个事物的全面和正确理解就要涉及到了解另外一个事物，学习就会呈现几何负担的增长

意识到这点是重要的，?所以我们总是可以确定:知识是无限的,逻辑是复杂的,多维的,作为人我们只能在某个维度上,某个我们能把握的复杂程度上,以某种形式去把握它.我们必须相信这个,才能将我们认识事物的行动变成一种时时有理论指导的行动,这样的行动比一般的学习行动更有价值..

关于逻辑的逻辑

很不幸,在进行软件设计的时候,的确需要很强的逻辑思维能力,这种 OO 之后的设计比起 C 语言的代码更难懂(因此写 OO 代码的人写起 OO 程序来很容易明白,但是别人来分析他的 OO 代码往往不知所然,因为不知道他到底写了什么样的 OO 对象在内,需要人们去分析每个 OO 对象的意义,作者加了什么样的逻辑模型在这个 OO 对象上,而 C 语言的程序,作者和读者看的都是同一个东西,都是广为人知的算法,同样的 I/O 操作,没有任何一丝关于封装的迷惑)

所以关于逻辑的逻辑这样的东西就出来了,在设计微内核时,微内核本身就是关于内核的逻辑作为内核,因此它只是一个逻辑模型,微内核是关于内核的架构代码,本身是一种逻辑,是真正的内核的前端逻辑,而不是像 LINUX 内核那样,本身就内置了可独立运行的驱动,进程管理等是整个内核(真正意义上的内核就是这样的内核,而微内核只是称呼上的内核,实际上它是内核前端,是关于内核的逻辑),这是关于普遍意义上的"逻辑的逻辑(接下来的逻辑要怎么写,这前面的逻辑就是关于怎么写后面逻辑的逻辑)",类 C 过程语言和 OO 语言都要在设计时涉及到这样的问题

而 OO 语言中体现的设计思想就更甚,要深刻体会设计中出现中的各种逻辑的关系,,那些逻辑是前端,那些逻辑是模型中首先要实现的,比如移殖时,就要把各种平台的底层处理做到整个应用逻辑的最底层,只要先考虑了并解决了这样原逻辑,那些后来的高级逻辑才能被后来各个实现,有些逻辑是虚无的逻辑,有些逻辑是关于设计的实现,是实作品

逻辑可以在各个角度发散形成设计,就像说话时,我们站在任何一方,以任意一种姿态都可以表达一种意义,形成一句语句,这就是逻辑的多样性,一个形容词就是关于一个名词的词,是词的词,因此本身也是词,也是逻辑,也是一种意义.

与软工有关的哲学 唯物

没有绝对科学的东西科学,维度给了我们一切答案,永远有更偏更僻的维度存在,因此拿唯物主义来说,如果它仅仅是表达那么一种"理"(它仅仅只需要说明这点),而没有宣扬它是正确的(实际上唯物论本身站在另外一些唯度上看就是片面的,而片面就是一定程序上的不正确),那么唯物论在这个维度上就算是做到科学了

唯物主义表明事物是不以人的意志为转移的,但是这里面的最基础的一个说法都不科学,什么是事物?哲学都没有从一种形式上去说明这个问题(就像字典内对"强间"应如何定义呢,法律范式内应如何对这个词进行界定呢)

当然我们讨论问题时并不需要把所有涉及到的逻辑都拿来解释一通,但是作为哲学维度的"事物"是应该被深入和专门界定的,而且"事物"这个字眼本身就是不严格的,而对象对象中的 OO 中的 O,在计算机内存中,它是一个运行单元,在表达出的思想所指时,它又是不可形式化的.(在内存中总是那种形式,然而却可以产生不同的逻辑,这就是形式与逻辑的关系)

如果我们曾把生命中一段时间用于考虑这些哲学知识,那么我们就会与常人"分裂",正如写小说的人如果想说出与众不同的小说故事到达到无人深入之境,就必定要付出和得到一些灵魂,,产生排拆他人的想法,但是要知道,世俗的维度是我们生活所在的维度集,我们如果能认识到这种客观性,并尊重它,那么我们就不会分裂,

人只能是简单的,人只能是一个行者,在有生之年接受有限的知识,进行自认为或世俗认为正确的观点和作法并行动.而不可能永远是一个哲学者,没有人有足够的生命力来最终解释自己,得到的解释也只能在一个维度上成立而在另外一个维度显得可笑,,选择需要学习的人类知识去学,不要做一个大而全的学习者,在利用所得知识进行设计自己的应用时,应根据经济学所指,做别人没有的东西,才会显得有优势,你不必在任何一个方面都出色,但一定要在一个方面最出色(实践要达到"别无它,唯手熟而这样的境界"),这就是你的资源优势,可以转化为经济优势

逻辑

很多逻辑的意思都是不可言传的或者难以言传的，所以要看别人的代码时，除非别人在类文件形式的逻辑，组件形式逻辑的命名上直接让你明白很多信息，?否则你就不能有效地明白作者写这些程序时的想法，所以读人家的程序是难的，因此虽然你是面向源程序形式，但你其实是在求索别人的思想，更要命的是，你明白了这个类文件是什么意思，你还不能很有效地告诉别人这个类写了什么东西，，体现了什么逻辑，这是感觉是窒息的软件的本质就是一种逻辑的叠成物，，某某牛人说过，一切问题和功能的达成都是逻辑!!软件的眼光里，一切皆逻辑，，在这个软工年代，最终的产品是多人合作代码的产物，因此相互之间能明白对方的源程序里到底能提供什么功能是重要的，，在以前的程序=＂数据结构+算法＂的年代，程序员之间很容易在这方面获得沟通，而在现在这个软工年代，程序=数据结构+算法+架构＂的年代，你还得明白人家程序逻辑是如何组织的，哪些是功能级的实现，哪些是架构级思想级的东西（这些东西为了以后的可扩展而存在，虽然一定维度上也可以称为功能级的实现），,

所以逻辑该如何命名，，我们只能满足于用特定的，固定的形式去描述性地表达它们，比如用类文件的名字，组件的名字，，设计模式的中间用词，等等.

与软工有关的哲学 联系

范意上的设计是广泛的,不仅限于计算机的,也不限于软工抽象(软工和计算机是二个完全不同的抽象,虽然没有人提出过计算机抽象到底是什么,软工抽象到底里面有哪些抽象存在,我们仅能站在某个或某些维度上给出一个描述性的概念而不是有限集,如果能站在一个大全的维度上说明到软工的全部抽象,虽然这是不可能的,但我们还是给得出的这个结果取个名字,叫范式,范式总是某些维度上的产物而不是大全的维度产生,学习计算机的很多哲学思维可以解决其它域的哲学),设计并不仅仅面向于创新,有时是形式的重组,而不是内容的创新,应用形式的改观,设计出的产品,要源端是面向人的,因此要提供足够简单的使用和访问形式,在目标端是要达到足够丰富的应用逻辑(比如 XML 统一文档交换,但可由 node,root 这些形式导致足够丰富的深层功能,深层这里是指上层),因此越复杂越大而全越好(但是如果没有足够人力,我们应考虑设计出别人想不到的商机),应用形式和应用逻辑作为设计中应主要考虑到的问题,
在设计软件时,我们主要用 UML 工具,但是这东西是静态语言用的

与软工有关的哲学 辩证

因为我们总能不断地提出答案，所以根本就没有答案,这思想正确吗?对,是正确的,任何思想,当用语言这种形式来言传时,都不可能做到绝对精确地为人所知,虽然从哲学上讲,也不存在绝对精确的东西,因此,我们只需要意识到这种思想本身就够了,而不需要去精确地理解,,,这就是所谓的"每个人心中都有一个...",但因为大家都处在人类的维度范式内,因此大家的所思所想也就那些,任何人对同一个事物的想法都差不到哪儿去,没人能走出人类的内心..能做到不同程序的分裂而已,
分裂就是人不再是他自己,行动时没有按照自己的意愿而来,行动上想要干什么而思想上却不能因行动受控,或者行动上不受思想控制,或者偏离了社会准则,实际上任何人都是分裂的(每个人都有不同于世俗的想法),但进了精神病院的那些人就是真正的分裂的,因为能明白以上我说的这个道理的人,他就不会轻易分裂.处在二个分裂处的二个极端
我们永远只能是自己,世界永远有它存在的样子,我们要培养一种与整个人类并齐的人格,并不是任何问题中每个细节用词都是形式的,即每个抽象都是无穷维的,不可能精确地讨论它,只能在某个维度上某个情境上说它是正确的,指代物和关于它的指代永远只是相对的关系,一者是唯物的,另一者却是人的认识,是唯心的,唯物

合理性

什么是合理的,这是个站在哲学全面维度上也解决不了的问题,在其它维度上有不同的解.比如人的维度,当然人的维度也分每个人的维度,所以,这不可能有解,只能在给定一个具体维度和角度的情况下,我们才能去着手解决这个问题,在解出这个问题之后,我们也不能说这是科学的,因为在更多的维度上,任何既存的解答都是无力的.

任何问题都是平等的,没有那个事物哪个问题比哪个事物哪个问题更重要的说法,当你说重要这个字眼时,你已经用情了,就不再是道理了,因此,关心则乱(但是站在说话人的维度来说,每个人说话的维度都是关于自己的维度,是他自认为合理的理),但是作为人来说,因为我们是作为人的身份来考虑问题的,因此我们应在自身端放置足够形式简单的逻辑,而在要达到逻辑和功能端求索非常复杂的逻辑.,即用一种简单的逻辑形式,去成就后来复杂的逻辑.
每个问题都是平等的,仅仅在一个范式下,才有地位之分,才有细节和思想之分.
因此,维度永远存在,相信事物有解的前提是相信事物在其它方面和维度还存在未知,,学习知识的最终目的,是相信不可能有一个事物可以解决所有的逻辑,即,总有人类不明白地方或未触及到的地方,,即,问题无解,在这种认识下,我们就只需要掌握作为人需要学习的东西和只能学到的东西,,但是另一方面,不要因为理解了事物有无限个维度这个事实就去抵制学习维度,做人的维度或做事的维度,那样你就会迅速不适应社全,变得毫无用处.我在这个 CSDN 里提到的所有关于维度的哲学学说等等,都是某一个维度上的,并不足以成为你一生尊守或认同的信条,否则就会害了你.
缺点是事物优点的某些方面,这就是一种辩证的思想，这种思想,当你以为你自己找到一种形式能解释世上一切问题时,你在显得复杂的同时也就显得越简单(活着的人不可以是一个维上的人),单纯往往导致狭隘,复杂和多样才是合理的,我们的做法只能随境随地,不要企图站在一个维度脱离具体问题的维度.虽然你意识到维度这个词的确可以解释一切联系,但假如你当一个脱离生活的哲学家,这显然是可笑和没有功用的.
哲学就在这里发挥了尤为重要的作用.下面将介绍马克思主义哲学论

语言，道理和感情

英语适合被用来写合同，而汉语不适合，这是因为汉语是表意文字，因此汉字数量众大．而英语只有简单的最基本的２６个字母形式，然而这二者的表达力却大不相同，在表达力上，表达自然事物这二者相等，而在表达感情上，英语显然不如汉语，，比如英语用＂you made me sick＂来表达＂你很恶心＂这样的意思，当然，在足够的造字量和文化背景的情况下，英语可以和汉语一样达到同等功能的表意能力.

但是正如我们无法格定道理和感情一样，（有些感情就是道理，有些道理就是感情），因此有很多事情越话越糊，是因为有的人想就事论事，就理论理，有些人却忙于借讲理的幌子为私心护航(而不自知)．有时候说理就是传意,传意就会带情
英语是一种＂底层逻辑很精小＂的语言，英语只有 26 个字母(有限的最底层的形式，但可构建到巨大的后来逻辑)，然而却可以提供无数个单词，因此英语用键盘输入是非常简单的，在显示上，２６个字母的组合体可以表达一切,而汉语虽然有笔划,然而却不能直接用笔划的组合来表达意思,而是用笔划的更上一层抽象,汉字来表达书面意思,比如外国用 0 来表数倍数,而中间就只能用百,亿这些意义表单位和倍数
?感情和道理永远是一门语言分不开的东西，有些逻辑是人的，有些逻辑不是人的，并没有在一门语言上给他们区分，比如褒义词，贬义词，中性词，这应该是一门语言中很优先的逻辑，然而包括汉语在内大多数的语言都没有首先考虑这个逻辑，一门科学的语言，应首先给每个词定性，无论它是形容词还是别的其它什么词性．
哲学不是人的科学，就像数学一样，没有感情的成份，只有理性的成份（科学家得出数学中其实也有感情的成份存在，然而那不是严格意义上的数学）．
接口

学习的一个重要动作是尽量把自己置于一种无知，单纯的状态，倒干你杯子中的水你才能接受，这是合理的，因为我们要理解一个术语，，这个术语本身必定属于一个领域（在它的最高 IT 境界），因此必定可以泛化出一个本质，虽然在这个本质下，人们得出了和正在得出很多复杂的实践认识和细节实现，但这个本质本身是不会变的，有点形而上学啊，，不管了
也就是说，一个术语必定到它的最高境界就是一个很单纯的东西，因此我们可以用足够单纯的心态去看待它
在理解接口的本质时，你只要幻想它是一个适配器和接口卡物，，但过了这步显然不够，，你还需要经过很多的实践才能最终明白接口到底是什么，，可以是什么(tomato 你可能永远不知道它是什么，但你吃了一口过后一定永世不忘，就是这个道理)

下面我就举出很多ＩＴ界的例子
代码库，设计模式中某个角色逻辑，类库，ＣＯＭ组件，一般组件，一个能在二个逻辑之间发挥承接作用的类文件形式的逻辑，都是接口，这些条条框框就是＂细节，就是实践维度的认识＂，而只有在同时了解了接口的本质，和这些实践维度上的认知之后，你才能说你是彻底了解（仅仅是）编程领域适合的这个特指的＂接口＂

编程领域的接口从来都不是泛义的，而是特指的，但一开始要获得一种泛识，才能更好地学习细节和实践

开源与开放

Spring,strcuts.MVC 都是框架和模型,思维模型,rails,spring 是 web 框架,是一种"接口"中间件,应用的复杂性要求我们使用一系列中间逻辑和中间件实现了的中间逻辑,直接在这些实现上发展新的,叠加了的新逻辑.
在我写文章的时候我并不会用故事的形式,因为在讲述道理的时候,除非就事论事(而不是用故事的形式),否则故事这种形式将会带给人们更多其它维度的困惑
设计中的泛化本质上就对应了应用可以泛化这个道理,我们可以整合一些现有应用成一种标准,一种架构,或者提出一个新的大架构,在这上面发展很多其它新的未端应用.一切在于我们如何泛化,在哪个维度上泛化,要由这种泛化得到什么样的实现,这也就是设计中的应用分析,架构设计
开源永远是好的,不光是因为产生它的力量大,从最本质的原因来说,是因为开源的起因是源于统一流行应用的要求而出现的,它的目的可能是提出一个架构,形成一种流行的应用规范(API 是否开放源代码就是一种),避免技术团体或公司由于历史原因产生的可绕过的历史复杂性
因此 open jdk 的形式是简单的(比如它的类文件组织形式),提出了一种简单的形式,无论它有多么简单,我们总可以在这种形式上发展很多新的逻辑,形式的简单性跟要达到应用复杂性无关,有时,形式的简单只是为了使"复而杂"变得"复而不杂",但这种改变仅仅是形式上的,应用的复杂性是另外一个维度上的事情

编程设计与经济

你已经在卓有成效地学习计算机和编程了,但是要知道,哲学和经济学是一个人一生必须要学习的东西.首先学习的东西.首先意识内要解决的逻辑.如果说技术独立意味着生存独立,那么哲学独立意味着精神独立. 经济独立让你真正地独立,
技能独立让你生存独立，哲学独立让你人格独立，经济独立让你一生独立，文学独立让你精神独立,,哲学的东西，是可以作为一生的信条去信仰和执守的，而人的东西，不过另外的东西，哲学并非人的东西
知道沃尔玛为什么有钱吗？人家做Ｂ２Ｂ，它为什么做Ｃ２Ｃ也可以生财？这就是策划,做与别人不同的东西. 经济学告诉我们,如果别人够强大,就不要跟人家对着干,除非你有与对方相当的财力与人力,你永远有你自己的理念和设计.有自己的资源和独特的想法.这也是这个世界变得日益合理和多样化的根本.宽架构, 实现的多样化,永远是最好的.

这反映在编程上也是一样,只有意识到(编程)设计并学习它，才可以充分发挥你的主观能动性，不拘泥于追求公共技术细节，独立他们发展你自己的策划,实现你自己不同于别人的功能并期待更好地商业化. 但是现在为止，大多数人还是在追求公共技术细节的实现，根本没意识到编程界有"大设计"这个东西存在,对编程之外的用户体验这个最重要

的东西都没有注意到，这就是腾讯的成功之处之一啊

伪码语言

无语法语言设计
伪码语言
这种语言没有类型，甚至没有语法，有一套语法，但是不尊守此套语法来写文章也是可以的
，在源程序书写错误时并不会出现编译不通过的情况，比如多打二个逗号会帮你消掉一个，这是指语法上的，就跟自然语言一样，虽然隐瞒得过编译器，但人一看还是知道哪错了哪没错的,, 去除一切需要严格尊守的语法,，作者保证逻辑和语义的原密性

语言本身可以扩展，比如增加词汇,,,这门语言的词汇要特别多
编程的唯一范式是造句,，
只需提供描述就可以编程,
Idioms 就是一切,每一个 idioms 要么是一个元 C 逻辑，要么是一个普通逻辑
仅仅提供了对 C 语言的接口,,这下面全是平台逻辑,比如 Windows 逻辑，C 库逻辑

逻辑单元不要函数。。。因为那样偶合,，会造成复用上的难题
没有上下文环境,,任何一句语句都可直接拿来复用

整个一个完整的程序就是一个静态网页，超链系统,,分这几大块，词会文件加，描述文件加，资源文件夹，启动器

每个词会都是一个可独立运行的逻辑

在解决把机器逻辑映射到应用逻辑上,，这个语言提供了什么样的机制呢。。
字符串并不停在目前抽象的水平，而应形成，在屏幕上打字这样的水平。

重复的逻辑并不会造成双份的运行负担

---------- 程序范例

制作一个程序;Windows,console,
它是一个子

为什么我说 Java 是脚本语言

与其说 Java 是一种静态系统编程语言，不如说它是一种脚本语言、
历史上对什么是脚本语言什么不是脚本语言没有一个定论，至少它与下面列出的几点没有一个必然的联系
1.此语言是静态类型的，还是动态类型的语言，
2，此语言是强类型的，还是弱类型的
3，此语言是面向本地的，还是面向一个 VM 的语言
4，此语言是编译，或者编译成中间码被 VM 解释的，，还是直接解释的
5,此语言是图灵完备的，还是类 SQL，HTML，XML 的标识码。
综合以上几点，可以发现，其实任何语言都不存在脚本不脚本的绝对说法，
所谓脚本，只是说此语言有没有 DSL 的意思，即该语言是不是专门用来开发某个领域逻辑的。

因此脚本语言也可是，图灵完备的，静态类型的，静态编译的，强类型的语言。。
1，如果是图灵不完备的，就可以是 sql
2,如果是静态类型的，可以是 cint,动态类型的 Lua
3,如果是解释类型的，可以是 underc,编译执行的 cint
4,如果是带虚拟机的 Lua,不带虚拟机的 underc,
5，如果是半编译半解释性质的 Java,纯解释的 underc

这是为什么呢，因为脚本语言是相对系统编程语言来说
的，，系统编程语言就是指 C 这一类语言，，C 最重要的特点就是它用数据结构加算法来解释系统底层开发的手段，，开发 C 绝对离不开对系统原理的理解，，而这些原理，包括 C 本身，都是基于数据结构跟算法的
而其它语言，可以基于更高层的开发逻辑，比如 Java，可以应用包括 OO 在内的高层语言逻辑，，程序员不必涉入数据结构这些与底层相关的东西(当然，我们这里不是指数据结构的逻辑模型，而是指被用于 OS 和 C 语言运行时，这些开发时时克克要面对到的底层数据结构，这就是说，数据结构可以是一种思想模型，可以不一定用于底层实现，还可用于应用和高层设计，，但是，OS 实现，C 语言运行时实现所关系到的 C 语言机制，，C 面向开发的可复用库，，都是基于数据结构和算法实现的，，这就是系统底层基于数据结构算法，因此系统编程语言开发系统逻辑要时时克克明白这二种大思想这个道理)

从这个意义上说，虽然 Java 是一种通用语言，，但其实它被主要用于 WEB，也算是一种 DSL 了，也算是一种脚本语言。。因为它主要用来进行非底层的逻辑配置式开发。。是一种功能配置式语言，而不是底层实现式语言。。

宽松语法,无语法语言

我觉得对于一个平台的架构设计，有 C 语言和 C++(我这里说的 C++并非普通的 C++)就够了

如果能把 underc 作成模糊其类型，，并且能作成类 html 高度通用的非图灵语言就好了，这样的语言要叫 C++。

我们提倡把 C 当成库语言和低层语言，，把解释执行的 C++当成应用语言和终端语言，在这些脚本语言中(如上所说，我们应首先把脚本语言发展成非图灵必要宽语法的)，应尽量向程序员模糊语法和类型。。

但类型机制你首先要知道他是怎么来的。。

类型机制几乎是从讲解一门语言的语义就开始涉及了，这导致了在使用此语言的一个实现时进程编程时，，如果没有弱类型机制(我们知道，强类型机制几乎是保证系统编程质量的要求之一，相比之下动态类型太偏向程序员而远离了需要严格控制类型的底层)，那么我们编程处处就得涉及类型机制，变量，直到对数据结构的掌握。。

这样的编程工作也就是数据结构加算法的方式，适用于开发底层逻辑。虽然 C 也可用来开发应用，但用 C 实现从 C 能表现的那些机器逻辑到应用逻辑的维度变换处处受制于对数据结构和算法的制约(我们知道运行时本身就是一个大数据结构，用 C 编程必须涉及到，而且 OS 实现中，因为用 C 实现，所以也存在很多用数据结构加算法实现的 C 的接口，要利用这些接口进行系统编程，必须了解数构和算法知识)

而 C 语言不单要求严格的类型和语法，而且提出了跟底层直接相关的指针机制，这种用内存地址来指代机器并表现系统问题的机制，深克地反映在用 C 实现的字符串，数组，这些东西上，一旦用 C 编程，，你就需要站在 C 系统底层的立场上设计现实问题，，而显然，用 C 来表现系统是最佳的。。

运行时,是源程序跟平台 (OS,CPU) 的接口,,运行时驱动该语言代码映射到此平台逻辑对系统编程,,就是用"平台能干什么事情"..来构造应用...一个一开始就学 JAVA 而不了解低层的人,不可能深克知道如何从低层驱动计算机构成逻

对于最真实的机器逻辑，只有机器语言是最直接的反映者，C 语言毕竟还属于机器语言的封装，，它也并没有反映绝对真实的机器逻辑，除了它的指针和用指针表现的东西，，因为它终究是某种抽象品，比如 C 的函数，所有一切语言的子过程，在汇编语言和机器逻辑中不存在这样一个概念，函数和子过程相对汇编语言来说就是一种高级语言机制，而并非想指针那样的底层相关语言机制，而 C++就更是差系统底层远去了更多，C++除

去 C 的那部分语言机制，比如 OO，只有当它封装了 C 的函数时，才能控制底层，否则 C++根本不能称为系统编程语言，还有 STL 这样的机制，，根本在系统编程阶段用不着。。

有没有一种编程理念能让我们彻底不考虑这些的东西，而使编出的语句和程序能够映射到机器逻辑呢，，这就是说，我们想隐藏机器逻辑，和语言对于机器的逻辑，，只有这二个东西被屏蔽了，我们才能站在非机器的立场去设计现实问题，]

虚拟机提出了一个理想的机器环境，，JAVA 语言更是隐藏了对 JVM 的逻辑，但是还是做得不够。因为 Java 是有严格类型和严厉语法的，，这样的复杂性跟开发底层要掌握的东西也差不多，我们应尽量去掉。脚本语言应弱类型或者干脆动态类型。。

语言机制作为库实现，，

最强大的语言原来是预处理

我们知道，在编译器的开发中，实际上做了二件事，1，为语言定义规范，2 产生到目标语言的翻译。要让语言支持越来越复杂的语法机制的确可以增加语言功能，然而势必给编译器的编写直接带来负担。

那么，不妨换个角度看问题，，我们真需要那么强大的语法吗，，我们也许只是需要那么强大的写法而已，，如果只是写法，为什么一定要加入到语法中去呢，不但给该语言的编译器编写造成负担，而且使学习这门语言，利用它进行开发和复用这种语言写的逻辑都变得门槛过高。

预处理就是这样一种聪明的过程。C 的语法可以很简单(甚至于流程图可以体现出精确对应到 C 语句的伪码)。但搭配预处理实际上可以达到超强大的功能。预处理可以给你提供强大的表现机制 show 机制，让你有能力将复杂的逻辑表现出为简洁的语句格式，如果这些语句格式给达到给于语言一种语法机制的效果（因为预处理所代替的语句部分也是符合语法的）。那么预处理实际上是相当成功的。比如人们所需要的为码形式的语法，英语编程语言都可以用它来达到。

但是我们知道预处理本身并非语法，它只是简单的替换以达到便于书写，但是这种精神和意义是异常重大的，首先，它不复杂化语言语法，其次，它简化了语法，使人们不接语法形式去写程序（但实际上符合语法）。比如甚至可以用预处理简化掉三种流程控制。

它才是真正的胶水语言。

或者，我们可以把预处理做进语法，比如模板，但是这又会增加语言的复杂度，抵消了预处理最重要的意义所在。

比如 C++的操作符重载，一个类可以重载括号操作符变成函数形式（语法有效，语义不同),，预处理完全也可以达到这样的效果，如果它所代替的语句是语法有效的，即使语义不同也没有关系，因为我们是将程序发布供复用者使用的。它才是最好的接口语言。

上面谈到的用预处理统一消除语法的复杂性是一个方面。
降低语言难度的另一方面不是语法，而是语言数据，面向对象的语言用类来抽象数据。可以用语法上的类来表达现实的实体，或抽象概念。而只有数据统一了，那么，程序员之间就会达成一种共识，复用就会基于理解这些形象的类。

预处理和数据统一，是这种“可定制语法”语言最重要的二个方面。人们发布程序库时可以发布预处理说明。所有的库逻辑接口都被预处理为简单的为代码。而且是语句形式，这样复用就可以基于语句加数据了。而语言本身并没有得到加强，根本不需要变动其语法。

如果说预处理是从语法上减少语言的复杂度，数据结构是从“解决问题的数据抽象角度”统一解决问题的方法，而面向对象的类，是从“代码角度抽象现实事物模型”来使抽象映射到语言，是代码结构。。。那么设计模式，就是从“基于语言代码结构，比如类，从解决问题的一般性设计方案”入手，同样是简化用语言解决问题的整个过程。

即预处理是简化，数据结构是简化，类是统一和简化，设计模式也是简化，这四者结合，无坚不摧。

其实我觉得 C++之父不必把它的 c 前端版本的 C++弄成一个独立的编译器，因为不这样的话，写出来的 OO 代码也是 C 的，这样就能极大范围地做到语法上统一，也能将接口保持在 C 函数的级别供脚本语言使用。

泛型编程其实是一定程度上的自然语言编程

如果我说泛型编程是 C++最简单的语法机制（甚至比 OO 要简单一百倍），你相信吗？？实际上只要懂得 C++的发展趋势（C++的所有库，包括标准库，BOOST，都是以模板和泛型实作的)，我们就会了然于胸了。学习应该“深入浅出”，只要先下功夫明白了“模板与泛型”，那么我们在使用 STL，BOOST 库时，就会猛然发现，原来学习起来复杂的东西，，其最终目的是为了带来使用上极大的简单性。

如果我们将泛型编程手法的意义再拓展一下，我们就会发现，它其实是一种最灵活的编程范式。甚至可以产生自然语言编程。一切的一切，源于类型可以只是一个空的占位符

（其实，整个模板世界就相当于是一台一台的炸汁机，在本书第二部分会谈到 type traits。只要向其中投进去一个具体类型，模板就会为你产生一个真实的数据逻辑，如果你只是使用模板而不是开发模板，那么你就会发现这对一门语言的优越性，这使得用模板设计的 C++库在使用门槛上很低），设计的时候可以不用考虑被如何编译(当然，要求所有的模板是语法正确的而不管它是不是会被编译，这点似乎不好，新的语言应解决这个问题)，我们要做的工作，就是把模板内部的那些过程式逻辑也弄成占位符(泛语法编程而不仅仅是泛型)，只有先从这个角度考虑问题并解决它，其它的讨论才会开始变得有意义起来。首先来谈这个所谓的“泛型”，再谈泛语法的问题。

我们知道，模板是一种型，虽然它可以以函数，类，结构的代码结构形式被体现，但它首先是一个模板，首先是一种模板“泛”型别(这种代码结构)，也就是说，它不但可表数据，在意义上还可以表其它东西比如一个动作，一个随意发挥的逻辑。因为它只是一个占位符。

泛型与动态型别的区别是：动态型别模糊了程序员对型别操作的必要(变量可以不用声明其所属型别就可以拿来用)，是一种进步，然而，其并非泛型，也只是具体类型而已，写源程序时是跟考虑这种型别在运行期有什么动作紧密挂钩的。

于是我们可以把泛型看成 dsl 中的领域词汇，自然语言中的词汇，当然这个词汇可以是一个名词，可以是一个动词，也可以是助词，也可以是某个句子结构，比如一个由助词构成的短语，英语中的不定式。但无一例外地，这些形式都是领域词汇的变形(所以，我们正是企图把自然语言编程和 dsl 结合起来)。

如果你看到这里就以为他跟面向对象用类来表示领域逻辑一样，那么你就错了，因为模板决不是类意义上狭隘的类型。我们可以借用模板元编程的那些手段，将一个模板的形式发挥到我们想要的复用形式。而不仅是一个数据加代码的类类型形式。

这就是说，定义一个模板的时候，我们不仅定义了模板逻辑本身(即定义了领域逻辑)，还声明了模板如何能被使用的形式(比如不定式 to do sth，这三个词中的 sth 可以是领域词汇,do 也可以用模板表示一个领域动作,to 就表示一个领域助词，所以，模板决不仅仅是数据加操作的这层意义上的组合)。这些都可以通过给模板增加多态化机制来进行。加多态可以表现为任何代码，这样每个领域词汇可以随意地组合而构成实际的意义了。

到底怎么样加多态呢，我们先在脑海上想象一下元编程把模板变成函数，重载括号后形成的泛函数形式，而我们知道，重载是多态的形式之一，就拿这个例子来说吧，对于一个 to do sth 不定式，不防我们把 to 这个模板领域助词的尖括号看成省略号的三点，表示“可变”参数类型。这个可变参数可以是源程序中的其它泛型。

这样的泛型语言，对每类应用都有一个词汇集形成的框架.所以是一种 DSL 语言，而且这

种语言本身就是一种设计模式语言。也不必发展出一个所谓的数据结构来解决现实问题。因为我们可以在这种语言中 bind C++的 dll 为领域词汇。这样，所有的问题都是复用问题了，稍后会详细讲解这种语言的这个方面。

最后，最重要的一点是，这种语言得改动一下 C++的模板写作时符合语法的要求(因为我们这种语言是无语法语言，当然，无语法语言是否图灵完备以构成完整的逻辑？下面我们会谈到)，我们可以写出一个不符合语法的机模板，即真正泛语法版的泛型模板。这首先得解决泛语法的问题。

我们来看如何解决泛语法的问题。

因为我们想通过泛语法解决泛型的问题，所以这种泛型模板也就是一种真正的无类型了。我们不防认为泛语法也就是无语法。

本文谈到的泛语法就是这个意思，在写源程序时，语器词，拟声词等在源程序中不起作用，先假想这种语言的编译器是个预处理程序，是一个自然语言到标准 c++语言的翻译器，最小的翻译单位就是这种领域词汇。所以它并不会翻译那些不是领域词汇的词，比如上面谈到的语气词，拟声词，以及不是领域词汇的动作，名词。所以这种语言的起作用的领域词汇部分跟源码注释是混合在一起的。整个源程序就是一个可读性超强的文本。自然语言形成的文章。

什么是泛动词呢，想象 stl 里面的通用算法就可以了，这种语言的源程序里，每一句翻译系统都会把它看成主谓宾，或谓宾这样的句子结构(当然还可以是其它的)，最后翻译成 c++的源程序。

而其实，这种语言也可不作为 C++的前端，而作为一般的编译器，即编译为平台码，下面我们给出这种语言下开发程序的详细策略：

首先在语言内部，因为其语法上支持泛语法(无语法)和泛型，所以我们可以在设计阶段进行自由，脱离语法的编程(因为此时并无语法)，只有源程序中那些领域词汇会进入到后来的严格编译过程中（此时出现了语法)。这种语言唯一的语法就是领域词汇构成的那部分源程序的正确性。是否能编译通过。而由程序员来保证每个领域词汇能正确编译，由人来保证逻辑的正确性。因为这属于领域词汇是人的东西而非机器的东西(整个语言没有机器的任何因素存在，没有变量，没有类型)，所以由人来保证它们逻辑上的正确性是合理的。

所以整个语言将会是这样的：I to do sth,he goes these.这样的语句
我们可以把这样的语句也封装为更大一级的领域词汇。

如果这个世上有朝一天会出现这样的语言，那么我就是它的鼻祖了。

一种可行的自然编程语言的实现法

如果站在最高的角度看编程，就会看到编程者，冯氏机系统模型，图灵语言模型，应用，如何将四者结合产生冯氏机上基于图灵开发模型的 dsl 的，自然语言编程。

只有这样想面对四大问题并解决它，才能使一切讨论开始变得有现实意义

首先我们来看待问题本身，编译原理告诉我们，

图灵机语言模型提供形式语法但不提供非形式形象的写法，

冯氏提供类型，它主导代码，为什么说冯氏是数据主导代码的呢，因冯氏机就是一台本质上的 io 机，开发者眼睛里只有类型（就是数据模板），在过程式语言内就是 plain old dates,在 00 语言内就是 abstract datas,函数也是一种型，不过他只能传指针的方法跟其它数据类型交互，class 就更是一种高级型，一方面是 adt,一方面是 udt(比如可以重载给以操作符),而模板呢，在 udt 的基础上 generic 了，但模板绝不是一种型，而是一种表现形式，代码结构，或写法结构都可由它来表现，与类搭配的模板就是表现类代码，函数模板就是表现函数

编程者需要写法而不是语法，
dsl 需要写法上的 idoms,领域相关的 logics

我们可以整理一下上述所有概念的对应关系，

编程者（需要泛语法跟泛型） 写法 --- adt,udt
冯氏机 (需要类型 pod 或 class)
图灵开发模型(需要形式语法)形式语法对应编程者的泛语法，即写法
应用(需要 dslogics) (就对应冯氏的型和编程者方面的 udt,adt)

而现在的语言，都只是解决了上面中的四个当中的一个。 那么我们如何把他们结合起来呢？对，是抽象！

抽象是解决问题的最好哲学，对于这四个问题，我们可以抽象它们，因为编程就是遵照语法写代码，根据我们在前面谈到的对四大问题的理解，我们可以首先把它抽象为：在写法下的类型，于是我们可以设置两大抽象，即抽象写法，和抽象的代码就是数据(冯氏是用数据主导代码的)

具体如何抽象呢？一种抽象的方法是参数化，参数化类型就参数化了冯氏机，参数化语法就参数化了图灵开发模型，这样就做到了泛语法加泛型，写法和类型，这两者都可用模板来呈现，力求 idoms 跟 logics 的书写产生一致的编程泛式，比如我们可以结合 yacc，和模板，形成泛语法(这就需要把 yacc 改成一个模板库逻辑，使它能以参数化的方法创建具体语言定义)和泛型编程。

什么是 idoms,什么是 logics 呢?
idoms 是语法规则，泛语法就是 idoms(写法习惯)， 比如怎么组合 logics.是参数语法，logics 是参数类

比如 idoms 也可定义词性，特化某个词性，甚至某个词

于是这种编译器就是一种参数化了语法的和图灵形式的通用编译器，而不是跟普通编译器一样，有一套严格的语法而是语法被参数化可变了。而且它是参数化了类型的，即是参数化了冯氏机的。而它又可以很自然地跟 dsl,编程者相结合。

只有 4 种文件在这个工程中：.idioms,.logics,progs(projects).docs

下面给出一个程序的样本

```
//演示了 C++下受限的自然语言编程
//关于基本的思想，请参看文章《一种可行的自然语言编程方法》

//把#include 形象化为 need;
#define #include need;

//把包形象化为字典，词典
#define package dict;

//需要 win32 平台上的 dict，一种 package.会带入 int type,float 等。
need dict_native_win32;

//参数化的编程范式,巧合的是，这里的 para 一词相关，既可表示参数，又可表示范式
#define template paradism;

//每个 typename 实际上都是一种 domain spe logic
#define typename logics;

//每个 class 实际上都是某种 domain spe logic
```

```
#define class logic;


//每个 typename 又实际上都是一种 domain spe idiom
#define typename idioms;

//每个 class 实际上又都是某种 domain spe idiom
#define class idiom;

//====================领域逻辑=====================================

//下面演示一个队列领域逻辑,它包括队列和队列中的项这二大逻辑,以 any 为参可以产生一个队列项;
//参数可以是其它 logics,也可是 idioms
paradism <logics any>
logic 队列
{
    //df
};

paradism <logics any>
logic 队列项
{
    //df
};

//=============习惯用法=================================

//下面演示一个 idioms,“如何使用队列和队列项”

//idioms 实际上是冯氏语言中的 code,但是因为被 data 统一了，所以这里也用 adt,udt 的形式
//参数可以是其它 idioms，也可以是 logics
paradism <idioms any>
idiom 队列用法 1
{
    //df
};

//这是特化版本的。
```

```
//"建交一个<队列>"
paradism <logics any>
idiom 队列用法 2
{
    //df
};

//"将一个<队列项>插入一个<队列>"
paradism <idioms any>
idiom 队列用法 3
{
    //df
};


//==================程序=====================================
//下面来进行具体的领域编程和自然语言编程

int main()
{


    建交一个<队列>;
    将一个<队列项 4>插入一个<队列>;
    将一个<队列项 5>插入一个<队列>;
    将一个<队列项 6>插入一个<队列>;

    //当然，你可以完善上面的写法为 idioms 或 logics,使它变得更为可观

    return 0;
}
```

计算机与编程

计算机与编程的联系点在哪里？为什么计算机会支持编程抽象中的算法等？
计算模型学指出，，，计算机只能处理一种被称为计算机接受“确定的串行算法”，（未确定的比如 AI,,,并行的算法正慢慢被发展**我们所接触到的书中的算法都是串行的，即一次**

只能进行一次运算）串行就是面向过程中的子过程，给定一个输入就能产生一个输出（有出口和入口),，整个过程不产生死循环,,（算法正确性分析,，算法复杂性分析,，这样的课题保证以上过程能进行）这样就能最终够成一个系统
为什么在一门语言中总会有控制结构，数据类型这样的通用概念呢，有控制结构是因为如果你在汇编下编过程，就会知道这其实是所有机器（比如堆栈机啊）和虚拟机进行处理的基本方式（源于离散数学的计算模型），有数据类型(primate type 而非 oo wrapper datatype---也即 ADT)是因为 CPU 也就只能按二进制的方式表达整型啊，浮点啊这些东东

一门语言更高级的就是，数据结构，线程，网络，界面，数据库，组件啊这些高级内容了，下面一一道来

有数据结构就是因为如果你在汇编下编过程，就知道程序是数据跟代码的结合体(计算机就是用代码来处理数据嘛)，数据本身从数值数据，字串数据啊什么的泛化到了 oo 的 class,,,那么在内存中这些数据该如何存放(及如何更高效地存放)以进行程序执行时的动态内存(就是数据流分析工程)呢，这就出现了数据结构,,，那么当这些数据不用时要等到下次用到的时候怎么办呢(就是字节流)，这就是数据模型的持久化称为数据库技术(由于数据从基本型转到了 OO 的 CLASS 型，因此数据库技术也就进化到了 OO 的数据库)，线程的泛化是并发，并发在很多时候都被要求用到，比如操作系统的多任务，网络中的一对多，数据库中的同步等等，因此一门语言必须不能回避并发性

注意数据跟资源还是有点区别的，打开任务管理器看到一个进程主程序只占一点内存(这往往是数据)，可是你发现却占了很多 pagefile(这往往是资源),，资源跟数据还是有点区别的，在 PE 文件中，资源文件指菜单啊，硬盘上的媒体文件啊，这些视窗资源，数据则指内存中的数据结构等

那么界面呢，这是因为现在用的都是 GUI（GUI 也就是表现抽象，一些时候它也跟并发性有关）了，WINDOWS 用自己的 OS 级的 GUI，类UNIX 的操作系统有它的 GTK 等等，因此一门语言或库(语言跟库几乎是同重要的东西)也不能不考虑对某个具体平台的界面支持，那么 WINDOWS 的消息机制呢，这主要是因为 WINDOWS 是一种基于事件的异步系统（也就是等用户来触发某个消息，WINDOWS 才能知道如何对它进行反映），因为又有不同的窗口(Windows 就是窗口啊)，这些消息(鼠标动作啊，键盘动作啊，计时器事件触发的消息啊，更新时的绘图消息啊，用户自定义消息啊)被发往不同的窗口，对于界面和消息 Windows 都提供了 C 语言级的 API(因此用汇编都可以调用这些 API，只要知道它们的调用方式)，而如果用某个库比如 MFC 来封装这些 API 以简化对 Windows 平台依赖性的编程就更好了

网络提供了分布式计算，它与 OO 结合点就是 RPC,RMI 啊---这里体现了组合思想（注意这只是在网络间传送接口而非传送整个 OO 对象的方式来实现二个异地对象之间的交互），它与 XML 结合点就是 JAX，OO 与 WEB 的结合就是 SOAP 啊，这些编程领域的

东西都在随着人们的认识进行不同的分解与重整合，而且轴渐脱离某个平台。向跨平台发展,,这也就导致了构件的产生，人们希望用一种类似搭积木的方式(相对一门语言外部来说)来 assemble(组合)一个应用，这些积木就是用不同语言开发的构件，因此需要提供一个交互用的统一接口(因此经常需要跟 IDL 技术结合),构件要相互协作发生作用形成最终的可运行系统因此需要部署，一般用 XML 来表达这种部署。比如我们一般说部属 EJBS

也即，语言内部的只能是一个一个的 CLASS，当这些 CLASS 脱离语言环境被分发（在二进制级以库的形式供外界使用）时，往往是以一个一个的组件的形式存在(而且要向外透露接口供使用它的客户使用)的，，

这些构件都是零碎的不能独立发生作用的，而且要实现比如对象同步，负载平衡，安全，持久等需要(这些需要统称为 SOA 需要)，因此发展出一系列的中间件（对 SOA 的实现）来管理这些构件。

Scheme 程序语言介绍之一

Scheme 的历史和沿革在网上有这样一句有趣的评论：计算机科学的大部分，就是在重复发现很久以前别人就早已发现过的东西。当然，这是一句玩笑。不过我们可以给这句玩笑接个下巴：对于程序语言中的每一个重要概念，你都可以先在 Lisp 当中发明一次，再在 C++ 里面发明一次，再在 Java 里面发明一次，再在 Python 里面发明一次，再在 Perl 里面发明一次，再在 Ruby 里面发明一次，当然，最后还要在 C# 里面再发明一次。我们以此开始我们对 Scheme 的介绍。^_^Scheme 的前身是 Lisp。和 Scheme 一样，这也是一门诞生在 MIT 人工智能实验室的语言。据说 Lisp 在程序语言的族谱上，班辈仅次于 Fortran，是第二古老的语言。但和 Fortran 不同，Fortran 经常被大名鼎鼎的计算机科学家批评，作为反面教材，这些计算机科学家当中有著名的图灵奖获得者 Edsger Dijkstra。而 Lisp 和 Scheme 恰恰相反，它们常被计算机科学家作为正面例子，一个优秀作品的例子。赞扬 Lisp 的人当中有 SmallTalk 和图形用户界面的发明人之一 Alan Kay。Lisp 由图灵奖获得者 John McCarthy 发明。据说一开始 McCarthy 只想把这门他正在设计的语言的语法的设计，往后拖一拖，等到后面有趣的工作做完了，再回头来给这门基于 Lambda 演算的程序语言加上为数学家们所熟悉的语法。可是 McCarthy 的一个学生很快发现，直接在还没有正式语法的抽象语法里面写程序，感觉非常好。就用不着一个正式的语法了。于是 Lisp 诞生了。Lisp 重要的特征就是：第一，基于 Lambda 演算的计算模型；第二，加上 List processing，这也是 Lisp 名称的由来；第三，直接在抽象语法里面工作，这是非常特别的。前两个重要特征，是 McCarthy 天才的设计，第三个特征则是有趣的巧合。又过了十多年，还在 MIT 人工智能实验室，不过这次不是 McCarthy，而是两个更年轻的计算机科学家。Guy Steele, Jr. 和他的老师 Gerald Sussman 合作对古典 Lisp 做了两个重要改进。一是把 Lisp 从

Dynamic scope 变成了 Lexical scope。现在大家熟悉的几乎所有的语言都是 Lexical scope，所以大家见怪不怪了。后来 Steele 成为 Common Lisp 设计的主力，Common Lisp 把 Scheme 的 Lexical scope、还有其它一些由 Scheme 所创造的特征，都加入到主流 Lisp 语言当中，Dynamic scope 终于成为了历史。Steele 和 Sussman 做的另一个主要改进是把 Continuation 这个概念引入到程序语言里面。这样一门新的程序语言就此诞生。他们按照人工智能实验室的传统，把它命名为 Scheme。这个 Continuation 据说是计算机科学里面，在程序语言设计这个领域，最让人感到激动，并且在开始学习的时候，也是最让人感到困惑的概念。或许有些读者朋友听说过 Continuation，这些读者朋友可以尝试分析一下下面由 David Madore 提出的著名的"阴阳迷题"。David Madore 是有名的 Unlambda 语言的发明人。(let* ((yin ((lambda (foo) (newline) foo) (call/cc (lambda (bar) bar)))) (yang ((lambda (foo) (write-char #*) foo) (call/cc (lambda (bar) bar))))) (yin yang))初学 Scheme 的读者朋友可以不管这个迷题。这是个专门写出来让人伤脑筋的东西。不过，如果能自己弄懂这个迷题，那说明你的确弄懂了 Continuation 是怎么回事了。对于一般使用 Continuation 来说，并没有这么样的古怪和晦涩。读者朋友大可放心。至于那个 Unlambda 语言也是如此。Unlambda 据说是函数式程序设计语言家族里面的 Intercal。和 Intercal 一样，也是一个专门折磨人的语言。Steele 和 Sussman 发明了 Scheme 以后，写了份 Report on Scheme。后来经过修改，又发布了 Revised Report on Scheme，这样不停的 Revise 下去。根据大师们历史悠久的找乐子的光荣传统，这一系列的 Report 被依次命名为 R2RS、R3RS、R4RS 和目前最新的 R5RS。现在还没有听说 R6RS 的消息。R4RS 是 IEEE 制定的 Scheme 语言标准的基础。 R5RS 比起这个 IEEE 标准，主要增添了"卫生（Hygenic）"的 Macro 支持。R5RS 之前，Lisp 的 Macro 就早已是程序员非常喜欢的、非常强大的语言特征。前面说过， Lisp 没有正式的语法，程序员直接在抽象语法树里面写程序。他们为什么喜欢这样呢？原因主要是他们发现，直接在抽象语法树上工作，其实是个非常大的便利，这使得 Lisp 可以拥有强大的 Macro。据后来 Steele 和 Richard Gabriel 所回忆，在 Lisp 历史上，不断有人想给 Lisp 加上正式的语法，这些努力和尝试包括 Apple 公司支持的 Dylan 语言，可是这些努力每一次都失败了。主要原因，就是 Lisp 的 Macro 浑身上下都散发出让人头晕目眩的迷人魅力。它拥有无比的能量，又让初学者感到无比的困惑。传统 Lisp 的 Macro，在 R5RS 的作者们看来，有严重的缺陷，这促使他们在 R5RS 中发明了"卫生"的 Macro。可是这个卫生的 Macro，反过来又遭到了传统 Lisp 支持者的严厉批评。对于不太熟悉程序语言理论的读者朋友来说，上面讲到的内容， Lexical scope 和 Dynamic scope，还有基于 Lambda 演算的 List processing，以及 Continuation 和卫生的 Macro，这些概念可能一时让人摸不着头脑，不过不要紧，这些内容我们以后都会讲到。读者朋友弄明白了这些概念以后，可以回过头来看看这里的历史介绍。回页首 Lambda 首先介绍 Lambda。许多非常精彩、非常重要、也非常困难的概念，随着时间发展，慢慢变成了日常生活中丝毫不引起人注意的事情。比如火和轮子这样关系到人类文明进程的重大发明，数字"零"的发现，等等。计算机科学里面也是如此。计算机科学发展的幼儿时期，数理逻辑中的 s-m-n 定理和通用图灵机等的发现，都被认为是重要成果，可是今天，许多熟悉电脑的中学生都知道，用软件模拟通用计算机是可能的。关于 Lambda 演算的理论也差

不多是这样。如果不是专门研究 Lambda 演算的理论，Lambda 对于今天的程序员来说，几乎是个透明而不可见的概念。实在是太普通，都很难把它说的有趣一点，或者看上去深奥一点。因为所谓 Lambda，其实就是表达了区区一个"函数"的概念而已。不过，在 Scheme 里面，Lambda 还是表达了两个值得注意的重要特征。第一个，就是广泛的允许匿名对象的存在。这很难说和正宗的 Lambda 演算的理论有特别的联系，它更像是由 Lambda 演算的理论所衍生出来的编程风格。第二个特征，就是所谓的高阶函数。这个特征和 Lambda 演算理论息息相关。高阶函数是 Lambda 演算理论的精髓，是由 Lisp 首先介绍到程序语言这个世界的。也是大量的现代语言，比如流行的 Python 当中一个似乎是不那么引人注目的特征。下面我们分头介绍这两个和 Lambda 演算理论紧密相关的 Scheme 特征。回页首高阶函数 Python 发明人 Guido van Rossum 和 Fred Drake 编写的 Python Tutorial 第 4.6 节列举了下面的例子。 >>> fib<function object at 10042ed0>>>> f = fib>>> f(100)1 1 2 3 5 8 13 21 34 55 89 对于事先不了解 Lambda 演算理论的读者朋友来说，第一次看到上面例子的时候，哪会想到背后深刻的理论基础和悠久的历史发展呢？这似乎就是公路上数不清的普通的轮子当中的普通的又一个而已，谁会想起生活在石器时代的我们的先祖们第一次看到这个滚动的玩意儿的时候是怎样的兴奋呢？不过，计算机科学就是不一样，如果你当真想亲眼看到有人对"轮子"发出由衷的赞叹的话，可以找一个 C 或者 Pascal 语言的程序员来碰碰运气。不过，如果你的运气实在不好，也许会听到类似下面的话的哦。"轮子？没有我家的小叫驴好呀！"玩笑说了这么多，我们下面讲点干巴巴的"理论"。^_^高阶函数有两点内容。第一是让函数对象成为程序语言当中所谓"第一等公民"。我们所说程序语言当中的"第一等公民"，指的是可以对这样的数据对象进行赋值、传递等操作。就像我们可以对整数型变量所做的那样。如果函数对象也是程序语言当中的第一等公民，我们就可以像上面列举的 Python 的例子那样，把函数对象在变量之间进行赋值和传递等操作。高阶函数的第二点内容是像下面这样。既然函数本身，就像整数或者浮点数那样，成了我们所谓"第一等公民"，我们当然就希望可以像以前能够做的那样，在我们需要的时候，把所有这些"第一等公民"，拿在手上揉来揉去、捏来捏去，无论它们是整数型数据、或者是浮点型数据、还是函数型数据。我们要把它们这样变换过来，再那样变换过去。把这些"第一等公民"放到我们的变换函数里面，对它们进行任意的、我们所需要的各种各样的操作。换句话说，我们可以像对待整数和浮点型数据那样，把函数本身也作为某个函数的输入数据和输出数据。也就是说，我们的函数可以对函数本身进行操作。这些可以操作别的函数的函数的地位不就是更高级了吗？这就是所谓"高阶"这个词的由来。回页首匿名函数

shell 编程和交互式语句编程

一切逻辑于语言为主，这是命令行的意思,,vim 也是一个配备了 awk 的环境，我们知道，其实在语言的眼光里，如果逻辑不是本身语言就有的，，那么用此语言开发出来的逻辑就是库，官方的库叫标准库，私人或组织的就是普通库了。除此之外，一门语言也可用其它语言产生的库，不过往往是在二进制级复用。。是隔了一层的逻辑。。

操作系统本身会提供二种 API，一种是 win32 api,,这样的 api,一种是 system call,比如 posix,,当然更多地是普通库，比如 x11 库，gdi 库，如果库本来就是由该语言写成，那么在语法级就可以 include 进他们的头文件，,在二级制级用它们的 dll 或 lib(我们知道 os core,posix,x11,gnome,gdi,opengl 这样的东西是 C 的，因此很容易给 C 语言提供它们的接口),,,,这是当逻辑库是用该语言写成的情况下，如果不是，有时语言也可在二进制级复用这样的库。。这就是第一段说到的隔了一层的逻辑，往往会另外写出一个该语言的 dll,来调用原语言的 dll,,前面 dll 是调用逻辑和接口转换逻辑，后面的 dll 逻辑才是实体逻辑，这就是被称为脚本粘合接口的技术。

shell 也是一种库，因此可实现一种语言 shell，此时用该语言开发 shell 逻辑就成了地道的 shellprogramming.

shell 给你一个窗口是为了调用系统(开发系统调用逻辑)，那么交互式编程往往给你一个脚本窗口是为了开发语言逻辑，这也就是脚本编程里面的交互写语句方式

Debug，编译期断言

对了,只是为了跨平台,所以一般发展一个虚拟机(这个虚拟机为一种语言的源程序提供了运行环境,更更重要的是,一般虚拟机内的解释器就是翻译器,如果说虚拟机本身是为了执行,那么内置的解释器就是翻译器了,,因为要把作为高级语言的这种中间语言翻译为虚拟机能直接执行的目标语言,目标语言这是目标平台能直接执行的语言的简称)
像 JVM 就是这种虚拟机,架在它上面的高级语言就是 JAVA,JAVA 跟 JVM 的关系很爱昧,JVM 是为了成全 JAVA 出现的,只要 JVM 实现了跨平台(实际上跨平台不是真正的跨平台，而正是在每个平台上都有它的实现),那么 JAVA 写的代码就能跨平台,但 JVM 的意义不单单是作为跨平台的目的来出现的,实际上,JVM 实现了一种标准的运行环境,里面有完善的 GC,有完善的跳转,是一种面向 JAVA 语言的高级的软件机器,更更而且的是,为了让 JAVA 成为工业语言,为了让大家实现完美的可复用,为了让大家普遍面对一种"RUN ONCE"的语言,JVM 对 JAVA 作了很多适应,
JAVA 中,是一种综合了编译跟解释的合体,怎么说呢,首先,JVM 能执行 JVM 目标代码,首先把 *.Java 这种高级语言源程序翻译为(这里是通过编译翻译,JVM 中也有编译器支持)*.class 这种中间代码,然后 JVM 中的解释器把*.class 再翻译为 jvm 的目标码,这样就能达到二个目的,1 比纯解释要快 2 使得 JAVA 代码能 complilr once ,run once
总而言之,编译与解释只是翻译过程,是对高级语言码到目标平台码之间的一种过程称法,动态静态语言这种说法是指运行期,,静态语言也可以被解释(像 JVM 执行通过其内置的解释器翻译中间码再由 JVM 来执行),反过来,动态语言也可以被编译,

真正的 STL

要注意，泛型算法是与模板分开的二个概念

就像宏用于条件汇编和重复汇编和宏函数,作为伪操作一样

泛型算法是一种思想，而模板是 C++的一种技术，,，只是泛型算法作为一个思想被指导于实际是通过模板来实现的而已

泛型算法有迭代器，函数 objects,,container,,它们之间独立而穿插

而模板有类模板,，函数模板,,，当然也可有其它模板

先来分清一些概念

C 和 C++的类型有 class(类),,funtion(函数也是语言的一种类型),,int ,,double,

这些类型按不同分法可归纳为纯量数据类型，结构数据类型，等等

注意以上说的都是“具体类型",，所谓具体类型就是可以用它们直接派生这种类型的数据,,

就像具体的人必须用人名来进行 ID 一样,，上面所述的类型都有它们各自的名字

C++的模板技术里<typename xxx>,这里的 typename 可以写成 class，但并不表示类的意思，它的准确的意思应该是“类型名”

typename 与作为“类”意思的 class 是一种全体与个体之间的关系,,，typename 是一种通用的类型,，即未指明名称的“一般类型",，可以

用具体的类型来参数化它

故我们采取一种说法，说如果模板是函数，那么模板的形参就是“一般类型"，当模板调用发生后，它的形参实参化为某种“具体类型”

产生只关于这种类型的这种模板类或模板函数,

typename 与 class（作为类）都是类型的名字,，前者是泛指,，后者是特指

所以说，，如果类是产生对象的模板，，那么模板就是关于类的模板，，注意这个产生与关于的区别

产生是实际分配内存而关于是编译时的参数替换(与预编译不同的是它进行了强类型检查是足够安全的)

再来说说模板函数的特化，

一个特化的模板函数会同时指明它特化和普通时的参数，，，特化时的参数放在<>内，而普通时的参数

放在()内，，虽然参数个数不一样，，但它们共享同一个函数体，，，当调用此特化模板函数时，当参数个数与作为普通模板函数提出时的参数个数一样时此模板函数作为普通模板函数来处理，，反之作为特化模板函数来处理，

真正的容器

用模板实作设计就是策略，实际上用模板实作的 STL 也是策略，然而它不是用模板来表达设计模式的机制，因此不能称为策略，然而， STL 却跟策略用了同样的思想，即用模板具有产生类文件和构造思想的能力，模板在 STL 中构造了什么思想呢，这就是总数据结构（数据结构有数据和对数据的算法，而总数据结构有容器和通用算法，而模板有很强表达设计思想的能力，很轻构地就表达出了容器作为数据存储节点的抽象，通用算法作为操作数据节点的抽象，，并实现了这二者的分离逻辑，）
所以， STL 的出现并不奇怪，，，如果我们能想到以上这样，说不定我们也会自觉地用它来实作出 STL 的，，说不定世界上第一套 STL 就是我们实作出来的呢

这再一次说明，，只要具备了某些思想，我们才能理解别人的架构，才能明白别人为什么用那套逻辑去架构库，就像我们懂得了上述思想才能明白如果有模板但却没有 STL 那才叫一个奇怪呢，所以说思想是优于细节的，而且这个思想显然可以直接被传递

策略的模板实现在“设计与策略”那一章中被提到

真正的智能指针

也即作为对象的指针

智能指针是一个类，，，可以由它派生对象
在支持 OO 的编译的眼里，，行为和数据集就是对象，，可是对象代表的意义可以千差万别，，这个对象可以是一个指针型的，，也可以是一个 INT 型的，，在这个意义上(注意这几个字，在某种意义上)，，对象可以是任意类型，当然指针意义的对象也就是指针对象了

上面说到的某种意义，就是某种角度，，更专业的术语叫做某种维度，，，维度是一种意义，，，比如三维空间，，，那么如果三维空间中的三维表时间，，，那么我再提出一个维表时间，，那么它就是一个四维（而不能说是四维空间因为这四种意义不全是作为空间的意义）

第一种说到的抽象，，，我们在进行抽象时，，，，只能得到一个维度的抽象（我们在进行设计时往往抽取一个维度就立马写代码），，，或称抽象集的一个维度，，，再如实现与逻辑分离，，这只能是一个具体维度上的可达性，，我们不能保证复用在任何方面都可以做到更好。。

智能指针也带有指针的意思，虽然它是一个对象，因为这个对象提供了->这样的类似一般指针的重载操作符，因此它在形式上下班模拟了一般指针，而且，更重要的，，这是一个智能指针
这是因为一般指针只是指示器，当它指向的实体发生变化时，这个指针并不发生变化，甚至可以独立于变体发生改变（也就是对指针的重新赋值）
，一句话，一般指针仅仅是作为地址意义存在的这就是所谓的左值
而智能指针有值的意义,,,它维护着它所指对象的一切并把它当值 value 右值，当对象本身发生改变时，智能指针能够自行适应这种改革者变，而且能够自行维护这个对象的销毁等操作
为编译期动态产生类的方法称为策略，，，动态期的称为 RTTI

真正的数组索引

指针经常跟数组在一起，因此它也经常跟索引相关
因为字符串也是一个数组(更准确来说是可以用数组来实现的线性表，只不过它的最后一个字符是/0 而已)，所以形如 char*的字符串也一定跟数组有关
如果一个指针指向一个 new type[num] 形式开辟的数组，那么经常有下面的形式出现，
数组名=指针=数组的索引 1 的地址=用双引号括起来的一串字符串
char* myvar = “iloveu”是用字符指针指向字符串第一个字符在内存中的位置，char[] myvar2=”iloveu2”也是成立的(字符串往往以字符指针或这种字符数组来表达)。
指针加 1 等价于索引也加 1，但是这其中发生的本质是不一样的，，指针值加了

sizeof(type)的值，，而索引只是向后一个索引递进而已
指针只能在堆上存在，，，而不能在一个函数的栈帧上存在,alloca 可在一个栈上声明
“同一”与“等价”的区别就在这里出现了
实际上数组的地址的确等于它的第一个元素的地址，，然而数组的地址并不是它的第一个元素的地址，而是这个完整数组的地址开头
mfc 的消息机制就是一种表驱动

真正的类库

为什么使用 JAVA，，这个原因不在于 JAVA 是一门纯 OO 语言这么简单(实际上它的支持系统比 JAVA 本身还重要,,一般把.NET,JAVA 看作真正的 OO 语言，因为它们提供的 OO 类库实在是全)，而在于它的 JFC 满足复用，扩展，这个世界低价的，多功能的替代品很多了，然而一个在设计上就预见了将来扩展需要的类库只有 JAVA Foundation Class（JAVA 的基础类，虽然 JFC 一般指 SWT 界面库但是我们这里把它作为 JDK 的全部库来看待），深入 JAVA，深入 JAVA 的类库，你必须拿出系统分析师的神经质，来领略其设计上的架构，而不仅仅是依照流传的文档进行依样画胡的设计，

记得在看谭的 C++第二版时时文件是流，然而会有多少人明白这是一个含义颇深的话，如果你去看 JAVA 的 IO，，它完全把文件，数据库，网络，内存,流视为同一个东西（甚至对象也可序列化，，可序列的意思就是，，内存中的对象，，文件啊都是内存中面向流的 stream flow,,,通过序列化就可以把它反持久化为结构化的对象，，一般是指 XML,,,,即二进制到文本格式的转化），，这就是原语泛化学习的好处(但是在使用上起初并不令人很容易上手，但是一段时间之后，你就会收获很大)，因为它们本来就是同质的东西，就像网络数据流其实跟文件十六进制码同是一样，，反工程它们也是相近的二个过程。

这就是整合与分解的体现之处，也是人们认识世界共性的一种体现(用于编程领域)。

以上是 IO，在一些细节和公用概念方面，JAVA 类库也有它们自己的一套概念，如 MVC(界面设计中的模型控制视图)，设计模式等，
如果要学习 JAVA，就细节方面来说，知道其 SDK 实现是必须的过程,JAVA 的类库是科学封装和归类的，经过学习，你就会惊诧它与时俱进的架构
不要惊诧为什么在你看过的所有代码中，几乎个个都是难解的而且都是遵守设计模式式的具体应用，它们的作者一定看过诸如 Gang of Four 系列的大部头，一个程序员必定是一个不断学习而且接受极快的人
而且，午万不要认为这些都是新知识，虽然从年代上来说这些东西出现的确没有经过多少年（STL 也才几年时间吧，面向对象也才十几年吧），但是，不可用年代来衡量一个东西的年纪和志向，对于一个在计算机界存在了十几年的东西来说，OO 算是老的了，，大凡是程序员（不包括那些机器程员序只会写代码不会学理论）都对 OO 有一个它自己

的知识结构，下一个十年，也不会出现诸如面向例子编程或面向接口编程这样的东东，因为它们都是面向对象的旧酒换新瓶(思想一般比较稳定，，虽然反映思想的细节千变万化，就像软件开发的工程学比较稳定，但是方法学却有很多一样)

可恶 OO

1,OO 其实并不是一种符合人类思维的理念,并不是任何东西都可以用 OO,或者被适合用 OO 来看待,并反映为编程中的原型,比如基本数据类型,还比如一些大的逻辑概念,比如 COM(OO 中的类在源程序中一般作为最小逻辑单位)
2,存在少量小逻辑的编程活动中,OO 可以胜任,一方面,每个类都可以写上有限的属性和方法,但是当软件中需要的对象数量众多时,就需要产生对象的策略(比如 C++的类模板),这说明纯 OO 的语言不能光提供 OO 机制,OO 并不能解决一切,多范型的语言才是历史所趋,C++这方面就做得很好,另一方面,当用 OO 实现的逻辑过大时,这就是被称为构件的逻辑,我们并不通常用 OO 来实现构件.
3,相比起 C 语言,OO 语言远离计算机底层,一些简单的操作不必封装为 OO 代码,而且就连设计模式中的一些模式的 OO 实现,也被 RUBY 证明为用简单的几行代码就可以搞定(这说明有些时候写代码是算法问题-就是 C 语言实现算法,不是 OO 之后的基于封装和为别人再利用这样的目的的设计问题)
4,如果说 C 强迫了人们向机器靠拢,那么 OO 则强迫人们向另一种很奇怪的"任何事物都是对象,都可拆,怎么拆"这样的思路发展,,未免有点傻了去了

真正的 DSL

DSL 领域专用语言
实际上语言也分门别类，,，有的是描述性语言（IDL,WDSL），有的是高阶语言，，也即领域语言（SQL，IDL）,,有的是实现语言如 C++,JAVA

声明性的语言是建立在编译语言之上的，编译语言的目标就是要产生目标中间码(比如 vc 的*.obj)，，机器码
像 C++，ALGO，，PROLG 这些语言都可以称之为领域语言，,，而 YACC 就是建立在他们上面的更高阶语言

如果你有过写编译器或语言的经历，你就会知道语义在这个过程中所发挥的作用了，,,,不同的语言用于不同的领域，，所采用的语义也就不同，，数据库有它的 SQL 语言，，建

模有 UML 语言，接口有 IDL 语言，，，等等，，，如果存在一种语义（并由此发展出一种语言），可以用来表示所有领域的语义，，那么由这种语义发展而来的一种语言就是元语言，，而 XML 就可以做到元数据，，YACC(编译器语义的编译器)在一定程序上实现了元语言，，而多范型可以称得上元设计了，，

作用何在呢？

每一个领域都有它自己的字母表(领域字典，，具体应如何理解呢，，比如在本书最后一部分提出的“虚拟世界逻辑”，在进行设计时我提到了很多诸如 Actor,,,,Mission 之类的用词，如果虚拟世界逻辑问题是一个领域，那么这些就是领域用词，如果设计模式是一个领域，那么出现在其中的各个模式和模式内用词都是 DP 的领域用词,但是这些要形成语法中的关键字),,,每个领域应当专门用一种语言来开发..有它不同于其它领域的语义(不跟其它语言一样),，语义是语言之上的一层逻辑，XXX 学指出，，用一门语言比如 C，C++ 或者仅仅用 OO 来描述全部领域都是不完整的
继承和虚函数都可以产生多态，
我们应该不要在接口中包含数据成员（像 VC++的.h 文件中就允许这种形式，这导致 C++ 的头文件并非一个良好的接口）
接口往往是函数阵列，而且往往接口本身也是一个对象，包含这些函数阵列作为它的成员，虚函数的意义就在于它不必定义它的实现，，正是因为没有实现，所以它可以被 override 而形成新的接口供以后的继承类来实现它
对接口编程就是对函数编程，，定义行为集代表的逻辑层之间的关系，而数据分派在子类中作为实现
判断一个类的成员是不是能被继承，，最终的途征是看它是静态绑定的还是动态绑定的，如果动态的，就可以，(static 成员)

要深克明白这个绑定与实例化的意义所在，，

推迟绑定时间到编译期(设计期)，就是元编程的意义所在的一个重要方面
元编程就是设计期实现和运行期实现的结合(也即在编码时同时进行设计期和运行逻辑的工作)

真正的多范型设计

编程主要涉及三个主体，，，现实世界(方案领域),，，，编程工具也即通用的表达现实抽象的工具（应用领域）范型就是编程界(编程界的范型都基于计算机这个底层实现)对设计(设计都源于人脑对现实世界的模拟和用计算机来表达它的需求)的表达因子

从工程的角度来考虑一个领域里的最高高度就是多范型设计(请注意这还只是设计而非

编码,但是设计中的某些因素最终要在代码中最现体现出来-当然并不是设计中的全部中间过程与细节因为设计过程中某些用词只是辅助用词,这“最终在代码中表现出来”称为对设计的实作，这实际上也可称为设计的编码，然而标准意义上的编码是写最终的应用而不是写中间逻辑比如库逻辑)

如何把你的设计代码化表现呢，用策略是其中之一，策略只是范型之一，当然还可用某种语言的一切范型，比如 C++是一种多范型的语言，范型就是问题域对应于方案域(schma)的具体解法这种统用说法，即设计通过范型才能在代码中体现，我们所有的编程工作就是利用语言级或非语言级的编程界范型(比如二进制的复用 IDL 等，这通常是部署设计)去实现问题。（因此要学习语言的范型和对问题的语言范型实现）

进行原语设计时可以不考虑范型(因为此时的原语设计不是面向编码的)，但是当用范型来实现时，就是面向编码的工作，此时要考虑设计中的哪些部分(去掉思想过程中的辅助过程，即那些不需在代码级表达出来的细节或用词)用哪种范型来表达比较合适。

原语设计与多范型设计有着不同的出发点（这是区别），但是它们的联系才是主要的，原语设计的出发点是现实世界，最终的目的是为了提出一种构架，,，这种构架可以为未来扩展提供足够余地接口(这是思想级的接口并不实际是我们常说的“面向接口编程”的接口,这里这里的设计跟编程没有多大关系，多范型设计中的设计扩展接口才慢慢跟编程和计算机走到一起)，多范型设计的出发点是计算机的范型，，最终目的是产生源码文件，并体现原语设计时提出的思想架构(这句话才是重点，这里再次重申，原语设计的架构仅仅是按人类的思考来定义的架构并不是以范型的观点来看待的)，但是二者是统一的(这是联系)，只要思想上科学的架构，当它被多范型用计算机的眼光被体现出来时，多范型因此没道理不会具有当初思想级架构所具有的扩展余地(或可复用余地)，也即它们是统一的，，然而，最终的扩展瓶颈是由范型设计所决定的,即接口余地=范型设计最大扩展余地=范型设计扩展余地+原语设计扩展余地。

为什么需要明白以上这些呢？明白以上这些，将极大地使程序员不满足借用编程界现有的设计，而是创立自己的设计方案并实现它（有意识的行动比没有意识的行动有效得多，而有上述思想作为指导是必要的,而如果连自身的问题都没有找准根本就不行）

你可能在问，明白这些思想的东东有什么用？然而你如果真不明白这些思想级的障碍，那么代码级的障碍你也许根本过不去,一个直接的例子就是，在看到一些技术文章的时候就会碰到这些用词时会用上

问题领域总可以分成它的子领域，，子领域又可分为它的子子领域，每个领域都可用一些范型来表示，对每个问题域的每个中间抽象层次都要考虑扩展和可复用（而且是在设计期就考虑），或者每个领域各用一种范型，或者
C++本身就是一种提供了多范型的语言（注意除了它的 OO 机制范型以外，它的模板啊，指针和引用啊，虚函数啊,GP 啊都是其它意义上的范型）

对多范型的考虑要充分考虑复用，因此必定要考虑到二个原则：找到方案领域的变化与，不变点，不变点可以做到基层(高层抽象，，，并不一定要用继承这种范型来实现，因为面向对象这种范型只是多范型的一个子集而已)，而在多范型的眼里，任何领域工程都不只通过一种范型来解决(比如的面向对象，比如某一种语言的面向对象)，，因此它可称为 metadesign,,注意，它是设计 metaprogram 是关于语言的语言，是语言之上的更高一级抽象，

分析过程往往
往往
找出事物的不变点与变化点这是多范型里面常常提到的二个词，这二个词反映了什么事实呢？
在“设计与策略”一节的开头有讲解

真正的反工程

反工程与软件工程是相对的

在软件工程里，一切皆对象 (这是指用了 OO 解的软件工程)，在反工程里一切皆比特，反工程过程就是从数据流中分析出一些它的逻辑(所谓的数据流分析),甚至推出它的系统工程化方法之一

轻易不要去学反工程，因为如果连软件工程学都不甚明白做不到知彼知己百战百胜，最初级的反工程往往是求得一个注册码之类的，发现运行出错之处，高级的反工程学就是求得作者写这个工程时的整个架构思想。

这里说是之一是因为得出的逻辑可以用多种语言多种工程化的方法比如 UML,booch,表示，

注意，除了能分析出一些数据结构与原程序一一对应外（字段名也可能不一样)，，，分析出的逻辑一般是重新用语言构造的，，这根本上是因为：

汇编器生成汇编码的时候并不以源码作为参考，而是以源码被执行时的逻辑作为参考，来生成汇编码的，这个道理就像优化编译器并不以源码作为参考生成与源码一一对应的机器码一样，而是生成同等作用的机器码，不同的编译器可以生成不同的机器码，但是这些不同的机器码都会有类似的逻辑，这说明，编译也是以源码被执行时的逻辑来生成机器码的(但是这个逻辑与源码的逻辑有惊人的相似而已。汇编码与它的机器码往往有一一对应关系，因为前者实际上就是根据后者来的)

明白这个道有用，这样我们的说法就是“从汇编推出算法，数据结构等，再模拟源程序的代码而不是标准重现(因为根本就是一个不可逆的过程)”

一个 EXE 或 DLL 的导出库往往是它在二进制级所能提供的全部接口，但是它用到的导入库(即 win32 sdk 库,mfc 库，或其它第三方代码库)函数可能是很多的

接口一般就是 C++的.h 文件里面 public 透露出的那些类和函数，在 .h 的开头一般都有__sth_h 字样，实际上就代表__sth_interface

所以接口的概念是无比重要的，它使第三方代码库隐藏了其实现内节，如果我们要用该库中的一个类，我们只能定义一个这个类的变量，再根据该第三方代码的文档来利用它，一般这个类仅仅提供了 create 接口（一个类能被实例化它就进入内存，这是最基本也是需要提供给使用者全部的接口，因为该类所有其它的接口都是该类 protected 的，都可以通过你变例化得到的这个类的对象来引用得出，比如一个成员函数或变量），,但是在文档中描述了使用该库的很多信息,,,这成为你获得该库信息并使用它的唯一途径，所以需要反工程法超越文档，或者在根本就没有提供文档的情况下反工程更为重要,,因为这个类的源码永远是不可得的，只要从比特流级别去弄清它的一些逻辑,正如我前面所说的，如果你是一个反工程高手，你甚至都能重构出它的一种工程化的逻辑

一些提供 SDK 的软件或库让你更好地使用该库并直接提供了在一种语言下的某些使用范例或继续开发所需的一些东东，或者即使在没有提供 SDK 的情况下，你也可以在具体使用它的层面上更好地使用它(比如继承这个类并 override 它的虚行为,或者利用其它设计模式中更加高级的手段)

overroad 是重载,override 是覆写

sdk 是什么？就是开发过程用到的所有东东， Win32 有它的 SDK，被集成在 VC++中，一些大中型软件也有 SDK，如 DX SDK，MAX SDK，等等，一般大型第三方库除了提供一个运行库之外，一般还发布一个可定制的编程集，这就是 SDK

所以，反工程很少是为了使用需要而去反工程此库（或者网流信息流，同样是 HEX 码），而是为了弄清它的一部分逻辑，得出它的一些数据结构，或者搞清它的某个重要算法如加密法

MFC 库不会提高我们的编程艺术,它只会让我们不了解 SDK，不了解 OO，当然这是为了学习而言的，,,，而如果你足够急功近利想要开发一个什么东西，那么就用 MFC 吧,微软好像就是倾向于把我们变成傻子！（所以要走出窗口放眼世界啊,无论是从库来说，还是从其它计算机的方方面面来说我们都要始终认识到我们所处的这个平台只是万花丛中的一朵，平台也有 CPU 的说法，，WIN32 就是 WIN 软件平台加 32 位 CPU 硬件平台

的说法）

父类与基类都是相对的概念,差异就在于多层和单重继承，,，父类一般用在单层中，基类的说法一般用在多重断承中，理解这个很有用，，分析 RTTI 时，归根结底，类都是 predefined behaviors 和属性，并不像过程一样立即被使用，

一个函数必定要用一种调用协议，就像 Python 要用 C++的函数要经过一个叫 swig 的改造一样(再比如将 Lua 集成到 C++的源程序中，必须向其宿主 C++代码提供 Lua-call 方式，因为 Lua 的函数调用协议有它自己的一套方法和数据类型作为函数参数，每个虚拟机也都有它具体的入栈出栈方式)，在一种语言所能支持的所有调用协议中，也有不同的协议，一般有 stdcall,fastcall,pascal,

stdcall 主要是用于 api,这个协议由子程序控制出入栈,参数由右向左入栈，fastcall 用了 ecx,edx 寄存器以加快参数传送速度，因为寄存器的速度要高于最快的内存和外存,,,,类的成员方法不同于一般的 api,类的成员方法这种 call 协议一般只用 ecx 作为它的一个隐含参数,即用 ecx 传 this 指针，this 即自引用，考虑 rose 中对象交互的消息机制有自身向自身发消息的那种

如果你发现一个函数调用(类的构造函数)，传递过去的 this 指针(指向该类，this 只用来指类体在内存中的指代，而类的实例自有变量来指代。所以不用 this 这样的设施)居然是指向了栈中一个未被初始化的变量的化(作为 this 的 ecx 一般不被初始化)，那么你基上可以确定这个函数是一个对象的构造函数，因为没有程序会在栈中放数据.rdata，特别是指针数据

堆栈和堆栈帧

把栈想象成一个智能水桶，，当空时，，它要盛水，就要上端开口，下端关

当使用这桶水时，下端开，上端不需要再灌水了
这就是说，1 它是二端开口的，但读存操作只能通过一端进行,这二端一个是高地址，一个是内存低地，一般是将高地作为栈顶和读写源
2，所谓 FIFO，这里的 FIRST 或 LAST，是相对 I 或 O 一端的那些出入数据来说的
3,栈这种内存块是动态扩展生成的,,,每个函数都在它自己的栈作为 EXE 进程栈的一部分，，，，被称为一个栈帧

加密与解密

我们上安全网站的时候，比如 EBAY，会向你的 IE 传达一个证书和一串值(公钥),，用这个 EBAY 提供的值和证书，可以表明你正在浏览的网站就是真正的 EBAY
存在着二大种加密方式（什么是加密呢？就是用密钥对明文加密成密文,，或用密钥去显示加密后密文的密文），这就是对称和非对称,，但是在这二者中算法和密钥都是重要的,，只是分配密钥的体制不一样(当然算法只有一个供加解密，最终影响的还是密钥，算法处在上层，密钥才是直接跟你接确的),，故导致的结果迥异,，对称算法中，加解密都用一个密钥（故称之为公钥）,，掌握了公钥和算法(如果该算法可逆举或穷尽，即可破解)那么别人就可以拿到你的明文,,,,，但是在非对称中(这个对称不对称不是指算法可逆不可逆，而是指有没有二个相对密钥存在),，只有用私钥加密,，用公钥解密,,，这样，除非别人拿到了你的私钥先,，才能用公钥解密
理解时要先想到算法，密钥是第二个要解释的问题和概念。

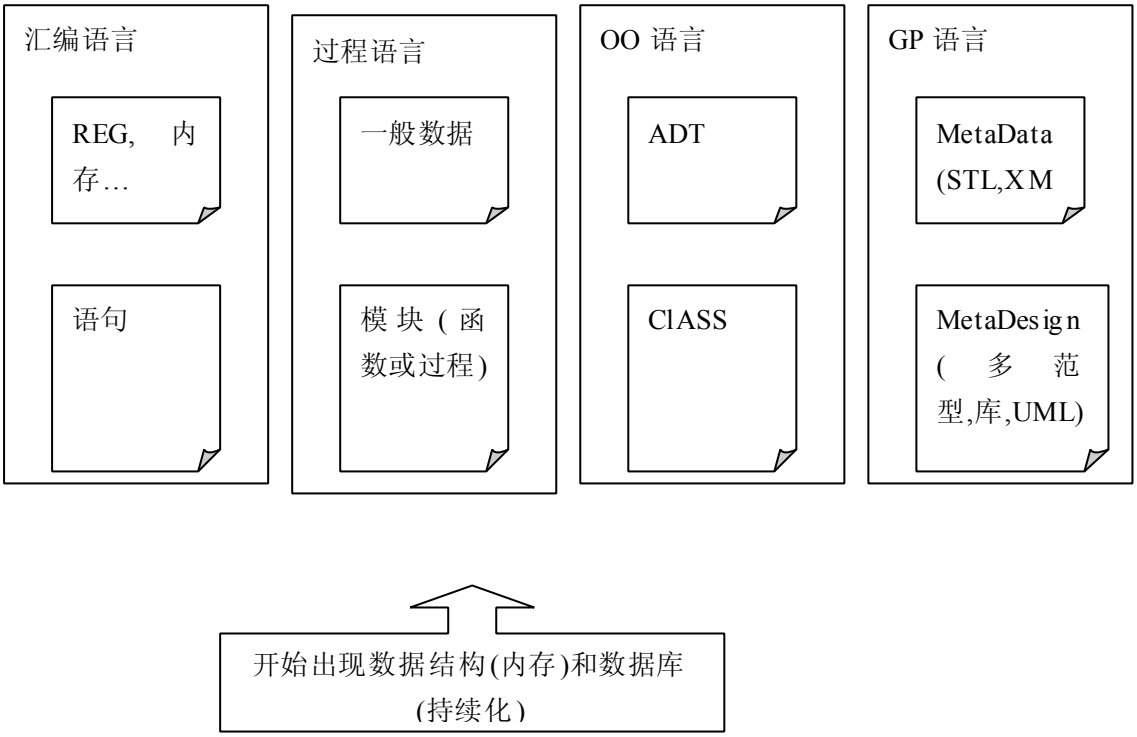
真正的调试

调试几乎是与设计，与实际编码地址相等的过程,,一个软件花在调试和维护上的时间几乎比实际编码的时间还要长(重构级的调试)

所以，千万不要以为调试只是一种与编译器打交道的活，实际上，它是软件工程学（特别是方法学）中一个很重要的部分，初期的调试就是 Assert,软工里面的调试还包括写 Log,写专门的测试案例(这样的测试可以称为事前调试,设计级的调试-希望这个词是我第一次用)

先来介绍一些高级编译技术
1.stlport 在 include 路径里置前，因为 VC6 的旧的 SDK 里带有同名的文件
2.dx8 置 dx9 前，并且置顶，因为需要一个 dxinput8.h 的文件
3.cpp 文件和 inc 文件到底有什么区别?本质上在 h 和 cpp 里都可写代码实现，然而 h 里面的代码经常是一些 inline,define 之类的代码，因为它们要被其它文件包含
4.编译头可加快编译速度
5.编译大型库时，注意条件编译，可选部分按需略掉
6.不需要 inc 的一个方法是用路径号./,,少用\
7.可在 link 选项卡里加路径
8.disable warning stl
9.直接把.lib 加进工程
10.macro redine 错误时用_下标
11.#include 的<>符号是优先在环境变量中查找，而“”是当前相对路径中查找

12.用 VS9 中的 VS8 来编译库
下面介绍一下大型开源库的编译技术,,比如 Yake 的编译
Dll,exe 会导出函数或资源，,,要引用其中的函数，,需要 import lib,,,,,,这个 lib 跟 static li b 不一样
对虚拟机编程是 JAVA,对本地编程是 C++,Java 早就由本地向服务器端转型了
C++依然是本地编程最好的语言！！它的优势就在于它的杂合性，杂合了多范型！！



编程与语言

真正的 RTTI

类型转换是程序设计语言中实现多态一个很重要的因素（经常需要在一个基类对象指针和一个派生类对象指针之间 Cast,,或者 cast 成 void），在 first class
cast 应该是 RTTI 的内容，为什么需要 rtti 呢？因为我们需要对对象在内存中的形式进行控制(比如 JAVA 提供的反射对 DB4O 的设计就有帮助，因为 DB4O 需要它控控制对象的引用并存入数据库),,
有同一个基类的对象之间可以相互 cast,
转型提供了多态
50 页下面(1,2 二句用于创建一个 Tobject 的子类对象,等价于
用子类名.creat)

try
1 aObj := TObject(TBase.NewInstance);
2 aBObj := TBase(aObj.Creat);
第 1 句分二步括号内是一部分，作用是
Tbase.NewInstance 等价于 TObject.Newinstance,由于 Tbase
未改写它的父类的 NewInstance(这个函数调用 initinstance
,与其它一些函数构造 VCL 创建对象服务的模型)这步过后，内
存中已有 TObject,,
第 1 句括号外 aObj :=TObject,作用是把“把创建的 TBase
对象的类型转化为一个指向 TObject 的指针 aObj”
（作者原话），
这意味着，在第一句前半括号内的内容过后 Tbase 对象也同时被创建了，
在内存中成形了(但是还没有达到像子类名.creat这种形一样完全成功地创建一个对象的效果),只是括号外的内容过后，aObj,aBObj 这二个
对象变量指针所指的内存范围不一样而已，由于被定义为 Tobject,aobj 指向图中实线体开始，而...
TObject
如何理解呢？？这个过程可硬性总结为，
第一句括号部分下来，实际上只创建了一个 tbase 在内存中的
模型而没有创建 tobject 的模型,毕竟没有显式地或独立地创建 tobject 在内存中的模型,它只是随 tbase 的创建而被创建了，而且即使被创建，它
也不是独立存在的而是作为 tbase 的一部分的,
即 tbase 的内存模型中包含 tobject 的内存模型(因为 tbase 是由 tobject 派生而来的，它的内存模式比 tobject 通常要大，你可以形象
内存中，tbase 包含 tobject,tobject 在 tbase 的上面，就像书中示意的那样
,这里我觉得 tbase 取名为 tcommon 更好)，因此在说法上，我们说 tobject 说被创建了，当然你也可以说 tbase 范围的上面部分不是 tobjct,
但是它确实符合 tobject 的内存模式，在道理上我们也可以说它是 tobject，并拿它加以讨论^^
括号外过后，把刚创建的 tbase 转化为一个 tobject,这样它的内存模式就缩小了，要注意 aobj 是始终指向 tbase 的(因为 tobject 实际不存在，只是
说法上说它存在于 tbase 的上面),，更准确地说指向 tbase 的全部部分还是它包含的那块 tobject 部分(道理上，我们也可以说 tobject 存在)
实际上到底有没创建呢？？这是一个哲学问题，不由我们来回答,,思考的角度不同，你可以说它存在，或者不存在
记住下面二句后面的就好理解了
tbase.newinstance 这种语句格式永远是指 .newinstance 前面的那个 tbase 在进行 newstance 操作
aobj:=tobject() 这种语句格式永远是指 aobj 指向括号内的那个 aobj 所指的 Tbase(******)

为什么这二者可以相互赋值呢？ tobject 和 tbase 是兼容的，因此可以相互赋值,,,只是经过上面第一句括号外的赋值过后， aobj 所指的范围就
变成了 tbase 内的 tobject 部分，相对于 tbase 的全部来说，这是缩小了，缩小或扩大的本质就是执行框架的变小或变大，即指向的内存模式的
范围大小，第二句过后，根本我上面打星号的那句话， aobj 指向的 tbase(别忘拉)
在完成 creat 的工作后变成了赋给 aBobj(兼容的指针嘛)，即 abobj 指向 tbase 的完整部分的开头,,
因此， aobj 的执行框架在第一句后指向 tbase 内的 tobject 部分(经过第二句后无变化，除非 aBObj := TBase(aObj);这表示把 aboj 类型转化为
tbase,,这样 aobj 就变得跟 abobj 一样，所指向范围一样)，而第二句后
仅仅改变了 abobj,使 abobj 指向 tbase 的全部部分的开头，即 aobj 和 abobj 的执行框架比起来，前者要小
现在来分析 53 未的例子,
001,aobj 指向 tbutton 内上面的 tcompent 部分
002.....
003,aobj 经过转化成为 abotton,,,指向 tbotton 完整部分的开头,,,因此 aobj 相对 001 来说，执行范围就变大了

Minlearn Ruby

首先要明白,技术是服务人的,学习技术要进得去,出得来,不要为一种特定的技术所迷,学习不但是学习知识而且是形成一种理念,开源就是这样一种理念,因为开源导致的理念是开放和协议的标准化,不会出现因为被那个公司控制垄断的历史复杂性，真正的学习是哲学，理念和技术细节的统一学习,,为什么人们说,XP 编程需要那么多条条的理念,又为什么又有这么一句话"电脑上的东西能不要删就不要删,即使你有最新的 bak". 唯有哲学,才能让你看清一切,从那些知识(人创造的逻辑)中解放出来,做回一个真正的人,并且更好地认识知识.
我将从编程的最初起源讲起,存在过很多平台和架构,在这些硬件平台和架构上又产生了软件架构(一般称 OS 为软件架构,但是称最终的 KERNEL 为构架,比如适合嵌入式的微内核，一般用在资源需求小的医了设备上， ＥＴＣ，和适合ＰＣ平台的单一内核),常见的硬件架构有 X86,PPC,ARM,等等,这些平台有特定的 CPU 和,因此有对"CPU 移殖,对 OS 移殖"的说法,往往移殖是一种重写多于改造的工作,因为针对不同的硬件,比如改造 linux 内核,实际上在 linux 内核中也存在一些与硬件相关的部分,因此你必须同时改造或重写这个部分(比如一种既存的平台架构),并针对特定你要移殖到的硬件作程序编写(比如目标机器上的,BIOS,BSP,主板驱动等等),,，开发软件，也有平台之分，平台导致的开发差异性,一般体现在,C 语言在各个平台上都不一样,适合 PC 平台(要知道 PC 只是硬件平台中的一种)C 标准库(这个标准只是 PC 标准)要被改造,在各个平台上要提供相应的开发包,这些工具包括：针对所用 CPU 的编译器/汇编器/连接器,相应的库工具,目标文件分析/管理工具,符号查看器等一般用 GNU C,在底层一般用 C 语言,现在的平台已经从硬件-桌面-到了 WEB,所谓 WEB 是一种以硬件以桌面为基础的更高层逻辑平台,如果说 X86 是硬件平

台,那么 WINDOWS 就是软件平台,那么 WEB 只是协议的逻辑平台,比如在 OS 内核中集成了网络协议(微内核一般只集成进程管理,视网络协议为后来要发展出的的目标逻辑原来网络协议是一个 OS 内核级考虑的东西，根本不是高层的，因此人们说 Linux 离开了网络就没有因为没有太多 linux 上的桌面应用程序,连装个 QQ 都要装个四不像的 QQ,那么 WEB 是一种什么样的平台呢,),

为什么会有嵌入式呢，因为智能手机平台，在硬件上都没有一个标准(甚至连 PC 上的电源管理也存在几套标准如 ACPI,DPM)，历史复杂性都来自于商业之争，而编程活动又服务于这个目标，因此，如果出现一家垄断公司，＂它带来的问题比它解决的问题还多＂，那么在这个平台上开发应用，简单就是犯罪. . 势必会在最后出现不利,只要标准统一了,平台移殖这个人类自身制造的大问题才会被根本解决,因为标准(硬件和软件的)被实现了之后,开发和基于开发之上的应用会很统一,想象一下,我们手机上是 Ubuntu,而 PC 上是 Ubuntu,飞机架驶室也用 Ubuntu 的情形,当然,这....

微内核的东西是考虑到有限的处理能力和内存,因此只优先发展框架逻辑,架构逻辑,其它支节逻辑作为后来的高层应用逻辑来发展,因此是非独立的,离开了应用逻辑它就没有意思(是关于内核的内核,这样一种逻辑),而 Linux 目前的内核是独立的,本身可以作为一种可运行程序来独立..

在上面提到的标准一字是什么意思呢,其实这是个很重要的词,比如 XML,,W3C,,ORB 这些都是专门提出标准的机构,,corba 为解决分布式处理环境(DCE)中，硬件和软件系统的互连而提出的一种解决方案；编程只是为了能为思想提出一个实作品,,J2EE 提出一套 WEB 服务器的标准,然后有各个商家去实现它,通用验证的"实作品"才能称为符合 J2EE 标准的,因此 J2EE 只是一套标准而已,再如 ISO 的 TCP/IP 协议,存在一种叫 RCF 的东西,人们提交标准,只有那些考虑到可以作为合理的,通用的标准才能被通过,并作为以后开发的标准,如果这个新标准在一种老标准之后提出来,那么基于老标准的开发统统都要"移殖",这样造成的历史复杂性就是人们给自身找的麻烦,只有事先提出一种好的架构(当然,这是一种设计,有时并不跟编程相关,因为在其它领域也存在设计,设计模式居然是建筑学部分大于软工学,可想而知,真正的设计是不分行业的,是一种考虑到历史,过去,现在,与实现无关的思想世界,)

在 PC 上,简单,高效的 Linux 没有获得跟 Windows 一样大的成功,而是窗内的世界也足够美好,可是在桌面市场上,,工业要求精准，Windows 可以做到，linux 就不行了,,而且 OS 一般被设计为通用的,一般这都会导致应用设计上的复杂性,而 linux 在服务器市场上还好我们提倡一种最清楚的逻辑，逻辑一层一层来(需要什么功能,可以由逻辑一层一层来,比如网络模型,比如 HAL,这些逻辑模式,由于每个逻辑都在某个层次上完成,只要完成了一层上的东西才能发展下一层,而且这样也造成了可隔离性,比如 OO,都是这样的思想,明白这个道理,你就明白了那些编程大师思想的根本),在最底层,它的逻辑应尽量小,而 plugable 的 embeded 是最好的,有一些 embeded 的 linux 还适合于 x86 的 PC 平台呢(智能手机,移动设备,MID 类设备也有各自的架构),因此,那种能可卸载的逻辑，模块式逻辑是最好的，因为可以按一种架构动态增删逻辑，，只在需要的时候加载，这样就比较合理，而且最底层要维护最小的逻辑，这样才能为有限的机器提供可定制的逻辑. . 同时适合 embeded 机器和普通ＰＣ. 甚至服务器级的机器，，在这些架构上使用同一种ＯＳ逻辑. .

最初的操作系统是怎么来的呢?因为它接近于底层,内核开发直接接近硬件,电路图,内核

中包括跟硬件相关的部分,比如驱动程序,实际上在驱动程序和内核之间还包括一个"驱动程序管理层"的中间逻辑层,这就是逻辑的意思
如果你查看 Linux Kernel 就知道(微内核是 kernel 的架构,因此是一种关于 OS 的 OS),它一般包括,首先要解决进程逻辑(对 CPU 任务逻辑的工作),然后是文件访问(比如如何 Mount 到 Flash 内存),硬件管理(而非驱动程序本身),而文件系统和网络协议相对是可选和可后来发展的逻辑,
从 C 语言的观点可以看清很多道理,下面谈一下移殖和交叉编译,一般
下面介绍几个平台逻辑达成路径
1,寄存器->CPU->KERNEL->OS->,,那么在开发中,与平台相关的逻辑会什么时候出现呢?什么时候给"可移殖"产生了障碍呢?
在设计应用的时候,,,在所有的逻辑中,那些逻辑会涉及到平台逻辑,,,在移殖时,哪些时候会考虑到平台逻辑?有些时候,开发一个应用程序的逻辑跟开发一个 OS 用到的逻辑量和复杂度相当,比如开发一个应用程序,我同样要用到 XML,并发逻辑等
而且,我们选取用 YAKE 作游戏开发库而不是 RAW OPENGL,是因为知道,YAKE 更接近游戏最终逻辑

真正的 Sun 策略

你真的以为计算机领域每一个知识点都可以被写成一本书吗，对，可以！但是这种现象反应了什么呢，这其实正反映了每一个知识点都是有限的（一个再小的问题都可以自成一个领域这是对的），只要在当把它放置到一个大领域内看待时，当它跟其它知识点或事物产生联系时才能发生更多的知识，这些联系和原来的事物本身便构成了这个知识点或事物，这样说终究难免有点深奥，但其实我真的有所指的，下面道来

记住：有时只是为了纯粹创立一种学说，而不管这种学说有没有用，很多理念与概念就会产生。

比如 XML，这是 Sun 递交给 W3C 的，知道由 XML 产生了多少技术和术语吗？反正我都不用再举例了，这种现象的本质恰恰正反映了 XML 本来并不是一个大领域，本来只是源于 Web 上文档交互的标准化，这是泛说(把 XML 泛化到了一个标准)，当它与数据库结合时便能产生 XML Native 数据库(还有 XML-Enable DB 等)，当它与 J2EE 的 JMS 结合时产生了 JMX,当它与数据打包结合便产生了 XML 作为最外层的 Wrapper(比如 WOW 用 XML+Lua,它的 XML 封装了界面编程代码等) XML 只是一种思想，所以为什么不能把它发展为一种数据库呢

再比如 J2EE 大领域架构的提出，结合设计模式和当前很多流行应用提出来的，这其实

是一种再亲切不过的技术规范了,当然并不是每个问题都大到跟 J2EE 相当，但是对 J2EE 的研究行为几乎就代表了其它事物的研究行为。

这就是 Sun 的泛化与组合的思想具体体现,他们玩的不过一种“抽象叠加”,“原语泛化”,“组合成繁”的玩意罢了，然而，这就是导致今天这个世界潮流的思想根本。

即，

感觉 Sun 公司全部都是思想家，他们特别喜欢做架构，而 Microsoft 的全是修理工，特别喜欢做细节，珠不知，有些思想一出来可以省去好多细节，有时不是缺少细节的问题而是缺少一种思想的问题，微软连这个道理都不明白吗？还是微软有它自己的更大智慧？

真正的 J2EE

首先要从 SOA 说起

SOA:面向服务的架构，这是一种为了直接提供高效能的服务而提出的中间件抽象(普经有一段时间各个企业都用不同的中间件，而当 EJB 提出后就统一了这种局面，而且一方面独立开发这些可伸缩的中间件的技术因素太多因此一般企业都选择直接购买然后在其上构建应用，这些 MOM 负责数据库池啊,，负载平衡啊，对象同步啊，线程啊，新旧系统集成啊，安全关机，重启啊)，这样人们就不能写这些抽象了，只要 wrapper 它们然后在上面直接写业务逻辑就行，比如 EJB 容器就是一种 SOA,，而在 EJB 容器内写 EJB 就是写业务逻辑了

EJB 可以相互之间调用，也可和 JSP 和 SERLET(Let 是小的意思，applet 是小应用程序的意思)发生联系。

一般直接把应用服务器跟容器服务器混为一谈，而SOA 一般是指面向 WEB 服务的 SOA

J2EE,是构建在 J2SE 之上的一种规范集合(主要是一些面向 SOA 的服务中间件规范),，如果说 J2SE 是一些库代码，那么 J2EE 就是规范集了，如果 EJB 容器规范啊这些应用服务规范，每个商空都有自己的 J2EE 产品，,SUN 会为满足这组规范的他们发一个 SUN 认证的 J2EE 兼容证。

要满足的 J2EE 技术和规范有：EJB，JAX——RPC，RMI-IIOP，JNDI，JDBC，JTA，JTS，JMS（一种 MOM），以上是应用服务端，还有后端的 SERLET，JSP，JAVA IDL,

JAVAMAIL，JCA，JAXP，JAAS，等等。。

也即，这些规范只是 SUN 定义给别人的“接口”，是抽象类，别人可以拿来实现。接口可以拿来实现（当它是一种思想的形式），接口也可拿来组合成软件（当它是一种代码或构件级的可复用形式比如 API 时）

现在我们来总结一下这里主要使用了多少三种思维的体现，反正我能找出这么多
1. df
2.

真正的 EJB

普通 Java bean 跟 ejb 有什么区别呢

普通 Java bean 跟 ejb 都是构件，都实现了它们作为 bean 的一些通用接口，但是正如上所说， ejb 是一种受 traits 的 bean(受约束是为了飞得更高),,是企业级的 API 类型，，而普通 bean 就是类似普通 vb 控件的可重用组件而已(提供了 intercepting 节获和一系列对应的访问接口 get,set)

什么是企业开发？j2ee 实际上比你想象的还要复杂，，它包括的技术实在太多了,,

message driver-bean 就是用消息(jms 技术)来驱动的 bean,,异步就是 Callback,,消息驱动编程

jms 一般出现在集成层，为了整合各个“微架构”而出现的机制，，就像 jxta 一样，，作为一种思想模型(Jxta 是一种协义,jms 是一种架构通信思想),,，在对 jms 的实现中，有“主题”,“订阅”这样的字眼，，而这些字眼就是观察者设计模式中出现的 characters

一个 j2ee 应用的持久层有很多东东要考虑，，比如 BMP(由 Bean 来负责持久),，Bean manager peresit,,,CMP,,container manager peresit 由容器来负责持久,POJO(plain old Java objects),数据库持久，，等等，，还有一些跟 XML 结合的技术比如 jax-rmi 等等，，

企业主要面向电子商务， B2B，B2C 等等，因此，它主要包括以下几个抽象层次，，客户端层，，表现层，，业备层，集成层（JMS，，注意 JDBC 是属于集成层的，因为它是一个封装中间件），，，资源层

注意， EJB 这些东西还只是业务层，实体 bean 不是数据库持久（即集成层）或资源层考虑的问题，，我们应该把实体 bean 与数据库分开（但的确有些遗留系统把业备做到了

资源层次）

SOA

面向服务架构,实际上在企业信息化过程中,企业的业务逻辑才是重要的,,EPR 方面的事能越简单就越好,因为业务上的事才是重要的,投入在 EPR 方面的资源要越少和响应突变的能力越强才最好
以前的软件是没有 GUI 接口的,,软工时代主要矛盾已经由性能低下的硬件与执行效率之间的矛盾转变为快速变化的市场需要与低效的开发工,internet 最初是学术交流的平台,后来成为企来控制的商业平台,,Web 的第一个目标是成为人们通过知识共享进行交流的媒介，第二个目标是成为人们协同工作的媒介，因此需要语义
当设计中出现的元素太依赖于业务了,就选择发展出一个领域模型
今天人们都很关注 SOA，我想说，我们的构件对外提供的就是不折不扣的 Services。”怎样解决粒度的问题？黄柳青说：http://home.donews.com/donews/article/7/79165.html 读一下这篇文章就知道 SOA 是什么,,,SOA 是一种比对象要大的构件,,,逻辑粒度大,,软工复杂度中的很多东西都会解决了,因为逻辑就是可拔插的

Desktop,web,internet,云计算不过 WEB 的集中化这种说法的偷换概念

其实 web 的构成就是由桌面构成的,因为 internet 上所有的桌面硬件都是有 OS 的,当这些"自治的,有芯的桌面"连结出现 internet 时,就形成了一个有 NOS 的大大小小的 internet 服务环境(B/S 集群等,这也是自治的,相联结的,),开发工作就不仅局限在桌面了,而是同时也同时在 internet 上,这种开发跟在本地机器上是不一样的,比如 WEB 服务就需要 deploy,还比如其它的 ftp 等,当然这当中最重要的是 WEB 开发与企业结合
因为 web 是 internet 之上的抽象,www 也被包括在 internet 协议中,当然也可以说 web 开发也属于 internet 开发,但一般把 web 独立出来说,现在的 web 开发在 internet 开发中的确是处于主要地位的,达到一种 browser 与 server 的应用环境)
注意,tcp,udp,p2p 指的是 TCP/IP 协议的更底层,,,跟应用层的 web,是高低抽象的关系,用来开发网络服务环境
所以我觉得,web 永远不可能完全取代桌面,因为在硬件基础上,桌面构成了 web(包括一些可移动 mobile 设备),一件东西不可能取代支撑形成它的唯物基础,,,就像一个人在未出

生前就谋杀自己的亲父母一样,,在架构上改变不可逆,,除非出现一个完全不需要冯氏 PC 架构(不需要 CPU 和 OS)的新 WEB 环境出现(即提供 web 服务的服务器是什么样的，人们需不需要将提供远程 web 服务的机子，和诸如浏览网页的功能，在功能上分开分为二台独立的机子，即服务器一台+浏览器一台，再比如网络游戏，提供世界逻辑的服务在服务器上，而提供图象处理和显示的功能在 PSP 上),,,或者在 PC 上只配备显示设备而不是带本地处理能力的功能(即根本没必要为本地硬件进行开发工作,因为它不需要软件,,也就是说,将我们现在能见到的 PC 演化成非智能手机,不带 CPU,也没有 OS,只提供存储如同 NC,而网络或 WEB 提供超速的传输功能,其传输速度和运算速度要接近超过原来的本地 OS+CPU 的处理速度才能符合应用,否则都没有以前快谁去用它啊.速度终究是一个很重要的用户体验啊)
在这种情况下,WEB 取代了 OS+CPU,本地的成本将大大降低(可能是一个类 PSP 只提供弱处理单元,或像非智能手机一些电路板就够了),,对本地的开发工作将几乎不需要进行,,,因为计算中心和计算逻辑由本地转移到 WEB 上了,本地成了一个无芯的电器
云计算实际上等于= web 2.0,或者 web x.x+ nc (这样本地就不需要进行开发,所有的开发工作都在 web)

晕计算

云计算实际上是个老概念,云计算就是=web 2.0,web x.x + nc,我们日常用的计算机会缩水成 nc,不必为一个没有 CPU，没有 OS 的“本地”编程和开发软件,,所有的软件都被开发在“远程”的 WEB 中去了，以及其它服务器,这就是云计算,，以后所有的云计算机才有 OS 和 CPU,,云计算机说将不必为本地开发软件，这除非本地计算机成为一个 NC 空壳,,否则，如果我们现在在用的计算机,，它有强大的 CPU，也有 OS,，那么对它的开发工作就会一直存在,,在云计算下,以后所有的云计算机才有 OS 和 CPU 这些云计算机仅是服务器，我们人手一台的 PC 将严格不能成为服务器，比如，因为我们的工作站没有 NT 操作系统，所以只能作为上网用的 broswer 这样的简单终端而不能成为提供服务的云计算机,NC 很多年前就被提出了，而 web2.0 是最新的，所以云计算就是不新不老的结合体
CPU+OS 的环境给了我们什么样的"本地"编程空间和限制呢,而 WEB"远程"计算又给了我们什么样的编程空间和编程限制呢,,,这首先要理解这二者的抽象基础
CPU 提供了冯氏架构,因此要有 IO,要有寄存器,OS 提供了进程,内存管理,API,进而提供了解释器,而编译则是直接面向硬件的,当然无绝对的编译语言或解释语言

云计算模糊了 PC 与终端,它倡导一种用网络来代替冯氏 PC 架构来进行计算的能力,未来计算资源将成为水电等基础设施而不是人手一台的带自处理功能的 PC 终端,(你只需用这种资源进行高层应用,不必直接掌握电能,电能本来就不必开发,,因为包括软件都是云计算所说的电,,切，而且这个电会智能为你个人定制服务，只要你一个要求,即 SaaS)
实际上以上概念还不是云计算的重点，它的另一个特征是我们日常意义说的软件将不存

在,人们不必为终端开放驱动计算的软件，所有的计算能力能给人类（个人或企业）的东西都表现为在 internet 上的服务，人们只需要一个终端（可能是不需要 CPU的NC,未来可能是GPU大行其道的时候..当然如果云的计算能力和运输能力足够强,那么本地不必配置图象处理设备,只需配置和实现显示抽象就行了）与之交互来获得这些服务就行了(提代访问控制和个人内容)，它包括二个方面,一个是富 INTERNET(还提供人类知识的全集和智能计算为使用者服务的能力),一个是 THIN 终端,然而到现在为止我写这文章的时候,,它现在为止还是一个概念,但是在技术上它一定会成立,不知道我有生之年能否有幸看到这种现实的形成,其实任何技术,,到最后都是哲学,,哲学到最后都是人心,我们能直接取人心,那么技术就是我们的搬运工,

最强大的力量不是精英而是小人物,最多的用户是没有学习过编程的用户,也即,应用现在还主要是面向愚民,这是微软的作风,Windows 到最后肯定不会跨掉,因为它做的不仅是技术,而是服务

如果就编程来说,为了自己的发展看,我还是比较喜欢形式统一的 linux(显然本人很违心地正在 Windows+IE7 上打出这些文字),这是技术人员的通病

据说云计算的目的并不是为了让企业计算机垄断网络,网络上人和企业是平等的,同享一朵对所有使用者开放的云,

而我觉得云计算是一种乌托帮的想象，这个概念更多地说的是一种生活体验，，而不是技术，，让人们远离技术和开发，，如果这种技术不是为所有人所掌握的(如果维护云计算能力的人或事物是由某些力量掌握的),那么云计算就是名不符其实的

富网页技术

界面是应用开发中一个永远的主题,虽然不是所有的逻辑都适合被套个 GUI 的外衣(WINDOWS 却干脆把 GUI 做进系统内核,使GUI 不再是一个外衣,而是一件内衣),网页的界面是个近年来越来越流行的话题,在桌面,界面已经有 GNOME 和 GDI,GTK 这样的大部头,对于浏览器中能展现的世界,也有相应的界面技术,有一个统称,这就是 RIA,主要有三个东西,HTML,AJAX,FLEX 等等

其实桌面和浏览器界面从历史上就是在一起的,最初的 INTERNET 应用程序全是用文字来展现的,比如客户端水木清华的 BBS TENLENT,后来 BBS 也可以用 HTML 来写,这说明基于 TCP/IP 的网络应用程序本身是不分桌面接口或浏览器接口的,因为这二者都基于一个 OS 核心(在这个 OS 核心内,提供有 TCP/IP 支持,,而且内核之上的桌面环境中,GNOME 提供了渲染网页的逻辑作为 APP 式的外挂,而 WINDOWS 则把 GDI 做进系统内核),,而浏览器端有 HTML 能更好地为这种应用造出界面而已

浏览器所使用的核心基于桌面环境中封装的一个专门逻辑,叫 XHTML,因此开发浏览器的逻辑是跟目标平台上的桌面环境逻辑相关的,富网页技术专注于为 WEB 套上更好更科学的 GUI,因此出现了 AJAX,FLEX,等技术

AJAX 是一种更科学的网页机制,在浏览器端,通过 Javascript 等客户端脚本技术,在发生

一次由 WEB server 到 broswer 的 pages pulling 时,不必重新 pull 整个服务器页面,只需要拉取数据发生了改变的部分(因此只需要改变当前网页上那部分的界面逻辑,是服务端的逻辑也是浏览器端的逻辑),
而 FLEX 刚是 ADOBE 用 FLASH 展示的网页系统,用 MXML 这样的 DSL 语言来专门开发.比较流行 .

附录：除了本书之外你还需要读的书

书籍：

一本数据结构方面的书。
你的工作所面临的现实问题方面的编程书。
最后，可能还有一本设计模式方面的书。

源码：

一切你能看到的源码。

附录：一些建议

1. 学会使用只配有键盘的电脑(不要使用鼠标)
2. 使用低配置的电脑
3. 读此书的正确方法是交叉读,但一定要明白本书的架构先

最后，对你死亡级的提醒，请不断写代码唯手熟而的方式是阅读代码，成千上 W，并实践 ! ! !

《本书完》

更多及后续更新请骚扰官网 : http://thousandd.appspot.com/